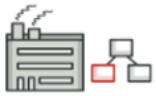
Design Pattern

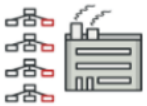
are typical solutions to common problems in software design.

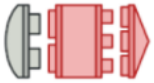
Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

https://refactoring.guru/design-patterns

Pattern Overview


Factory Method


Abstract Factory


Adapter

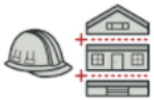

Bridge

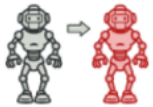

Chain of Responsibility


Command

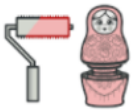

Iterator

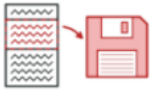

Mediator


Builder

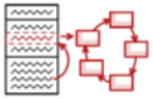

Prototype

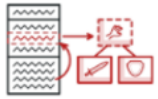

Composite


Decorator

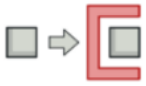

Memento


Observer

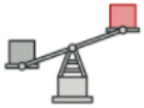

State


Strategy


Singleton


Proxy


Facade


Flyweight


Template Method


Visitor

Koppelung

Koppelung

Koppelung ist ein Maß für die **Abhängigkeit** zwischen 2 Objekten. Diese Abhängigkeit entsteht durch die Nutzung der Funktionalität des jeweiligen anderen Elements.

Eine geringe Koppelung der eingesetzten Elemente erhöht die **Wartbarkeit** eines Systems.

Kohäsion

Kohäsion

Kohäsion ist ein Maß für den **inneren Zusammenhalt** eines Objekts. Wird durch ein Element zuviel Funktionalität verwirklicht und ist das Element somit zu generell, nimmt die Kohäsion ab.

Arten der Kohäsion

- **Servicekohäsion:** Methoden einer Klasse sollten nicht zu wenig, aber auch nicht zu viel Funktionalität implementieren.
- **Klassenkohäsion:** Klassen sollten keine ungenutzen Attribute und Methoden besitzen.

Classification

- Design patterns
  - differ by their complexity
    level of detail and scale of applicability.
  - they can be categorized by their intent
    and divided into three groups:

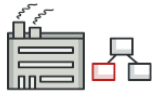
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

Benefits of patterns

- Patterns are
  - are a toolkit of **tried and tested solutions** to common problems in software design.
    - knowing patterns is useful because it teaches you how to solve all sorts of problems using principles of object-oriented design
  - Define a common language
    - Can be use to communicate more efficiently.
    - You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion.

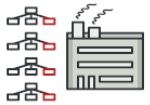
What does the pattern consist of?

- Sections that are usually present in a pattern description:
  - **Intent** of the pattern briefly describes both the problem and the solution.
  - **Motivation** further explains the problem and the solution the pattern makes possible.
  - **Structure** of classes shows each part of the pattern and how they are related.
  - **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.



Factory Method

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

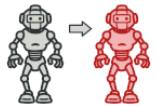


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

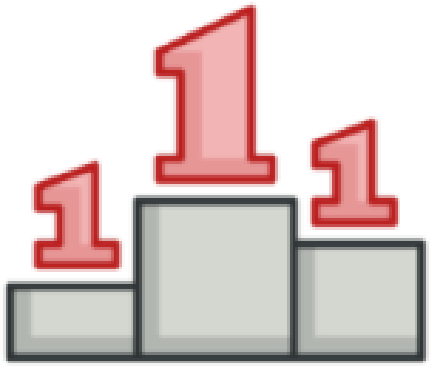
Lets you copy existing objects without making your code dependent on their classes.

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Creational Design Patterns

provide various object creation mechanisms, which increase flexibility and reuse of existing code.

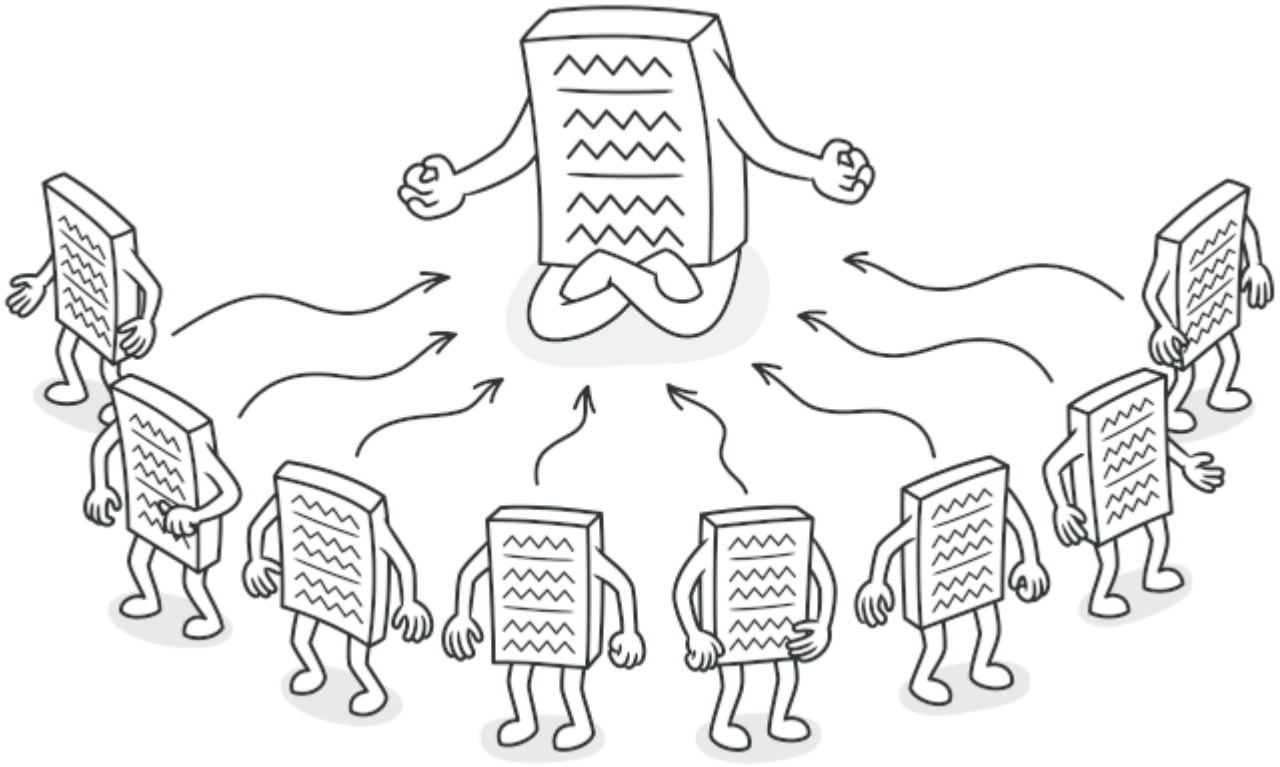


Singleton Pattern

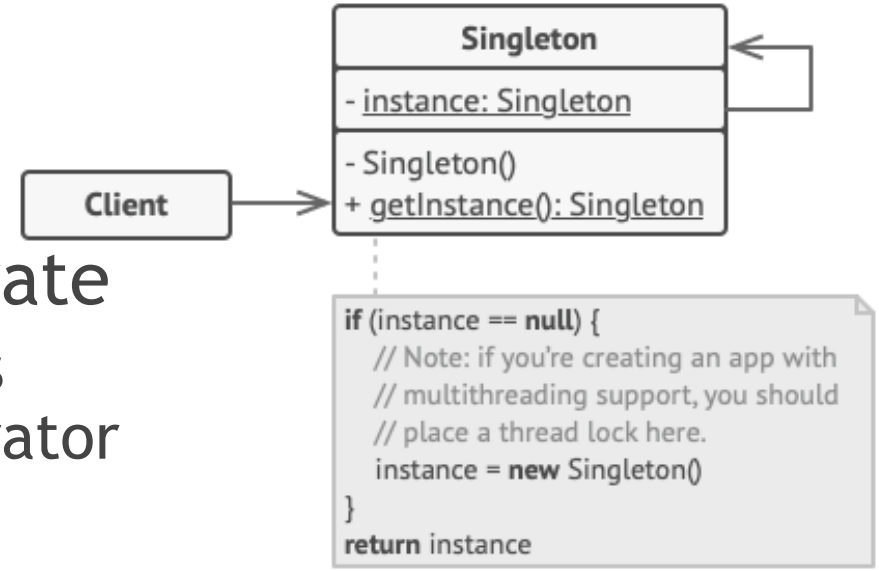
ensures that a class has only one instance

Singleton

- ensures that a class has only one instance
- providing a global access point to this instance



Singleton

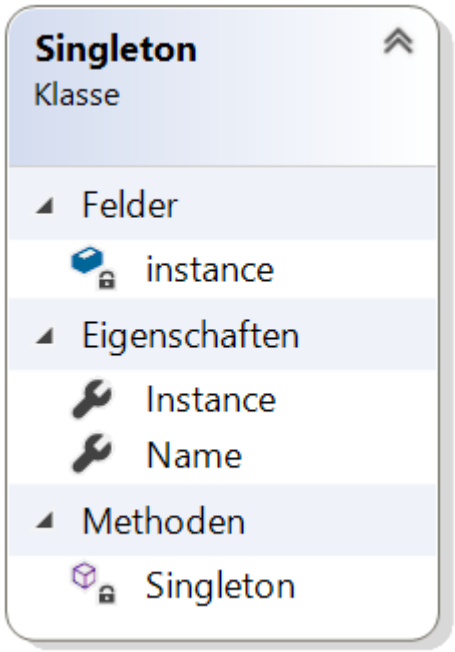


- default constructor private
  - to prevent other objects from using the new operator
- create a static creation method
  - that acts as a constructor
  - this method/property calls the private constructor to create an object and saves it in a static field
  - all following calls to this method return the cached object

UML - Singleton

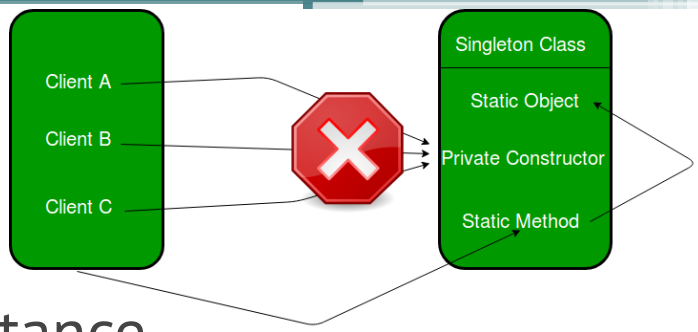
Implementation Singleton

```
class Singleton
{
    2 Verweise
    public string Name { get; set; }
    1-Verweis
    private Singleton() { }
    private static Singleton instance = null;

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

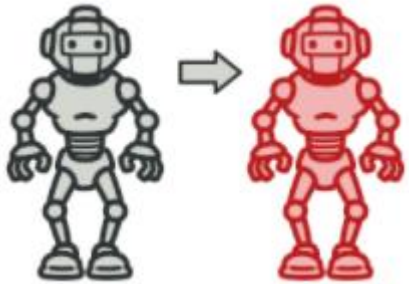



```
Singleton myInstance = Singleton.Instance;
myInstance.Name = "Kurt";
Console.WriteLine(myInstance.Name);
```

How to Implement



1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.



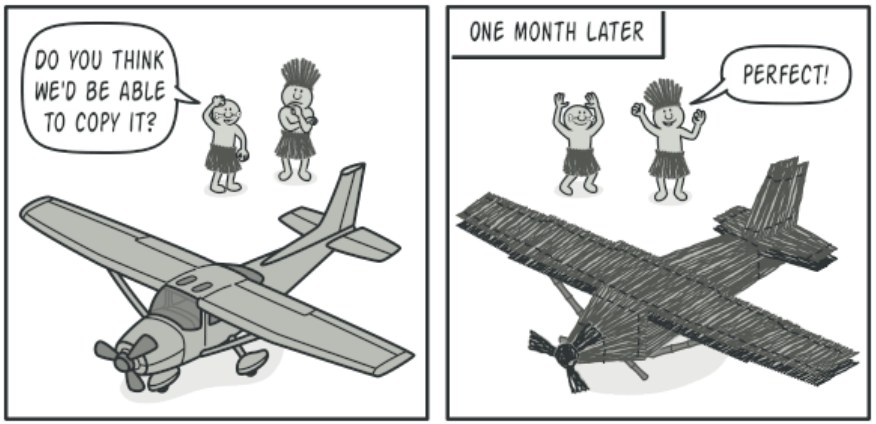
Prototype Pattern

creational design pattern that lets you copy existing objects without making your code dependent on their classes

also known as Clone

Problem

- Say you have an object, and you want to create an exact copy of it.



- How would you do it?
  - First, you have to create a new object of the same class.
  - Then you have to go through all the fields of the original object and copy their values over to the new object.

- Nice! But there's a catch.
  - Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.

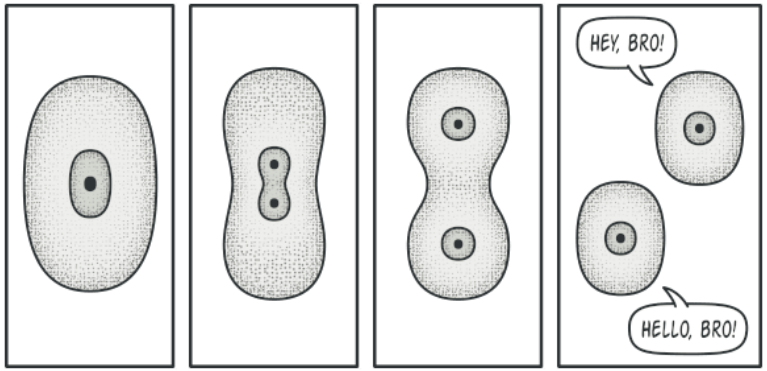
Solution

- delegates the cloning process to the actual objects that are being cloned

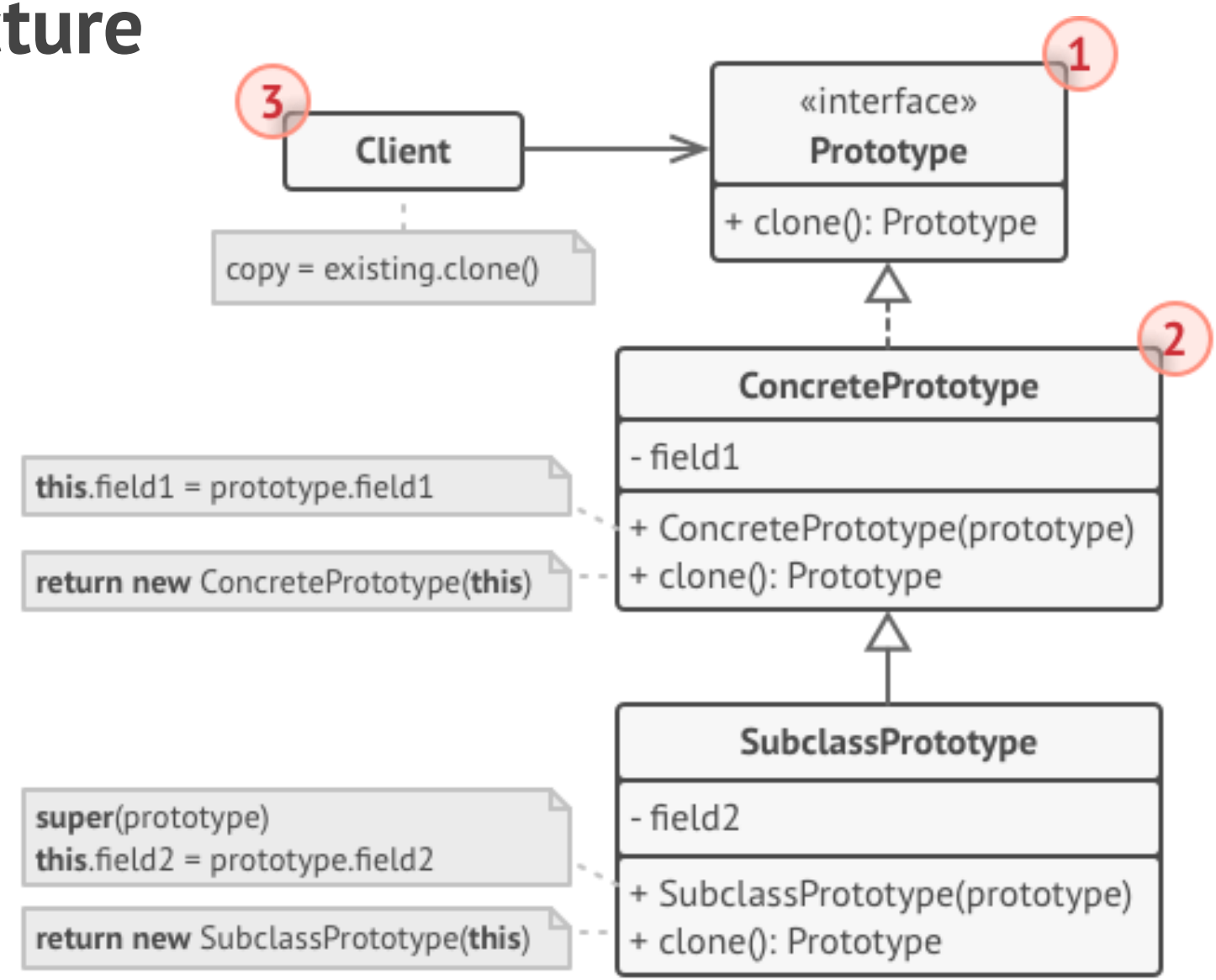
- declares a common interface for all objects that support cloning containing just a single clone method

Real-World Analogy

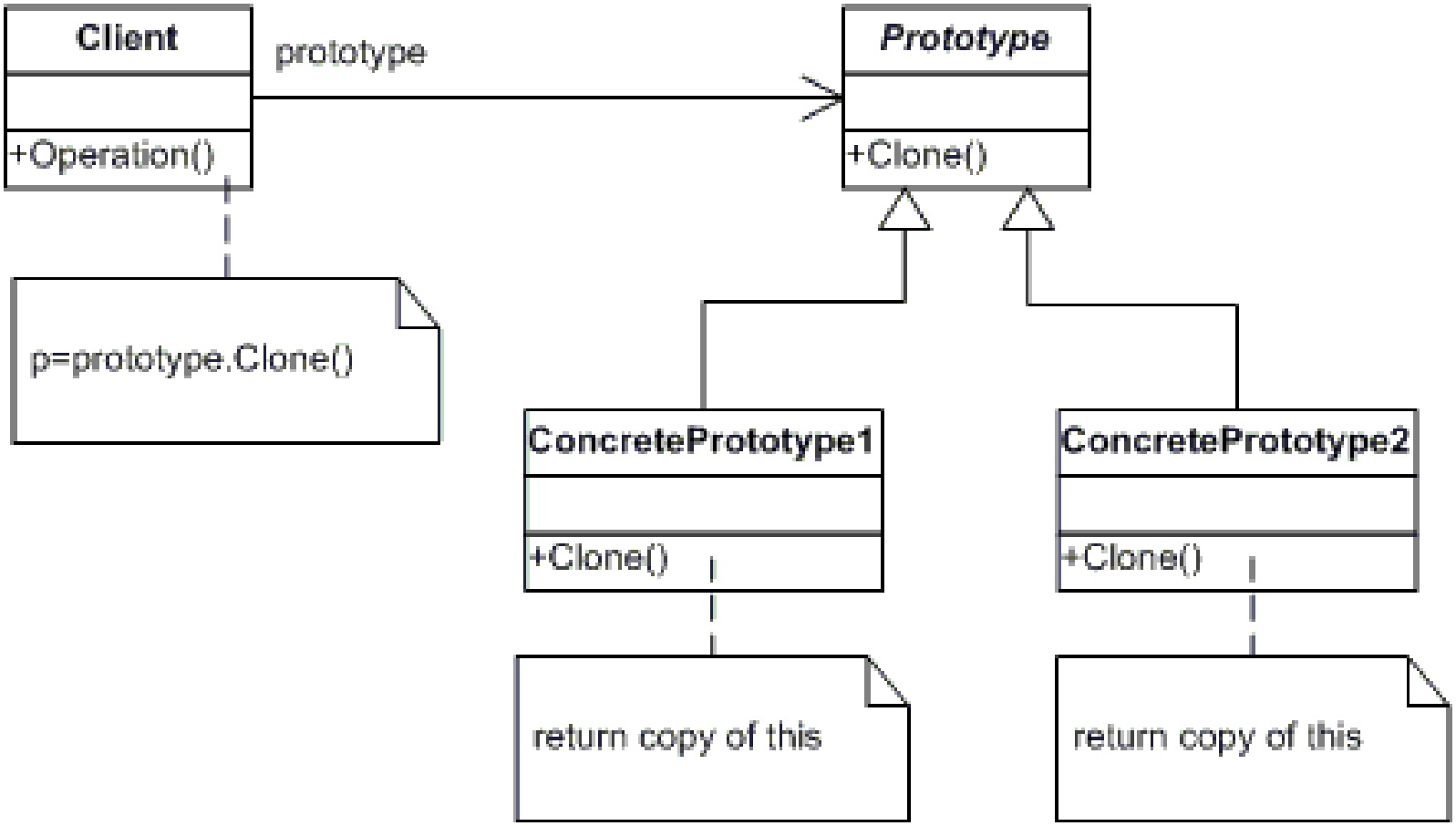
- industrial prototypes don’t really copy themselves
- closer analogy to the pattern is
  - the process of mitotic cell division (biology, remember?)
  - original cell acts as a prototype and takes an active role in creating the copy



Structure


3
Client
1
«interface»
Prototype
+ clone(): Prototype
copy = existing.clone()
2
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)

UML - Prototype



Structure

- The Prototype interface declares the cloning methods. In most cases, it's a single clone method.

- The Concrete Prototype class implements the cloning method.
  - In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

- The Client can produce a copy of any object that follows the prototype interface.

Clone a Person

```
Person p1 = new Person();
p1.Age = 42;
p1.BirthDate = Convert.
    ToDateTime("1977-01-01");
p1.Name = "Jack Daniels";
p1.IdInfo = new IdInfo(666);

// Perform a shallow copy
// of p1 and assign it to p2.
Person p2 = p1.ShallowCopy();
// Make a deep copy of p1
//and assign it to p3.
Person p3 = p1.DeepCopy();
```

```
public class IdInfo {
    public int IdNumber;

    public IdInfo(int idNumber) {
        this.IdNumber = idNumber;
    }
}

public class Person {
    public int Age;
    public DateTime BirthDate;
    public string Name;
    public IdInfo IdInfo;

    public Person ShallowCopy() {
        return (Person)this.MemberwiseClone();
    }

    public Person DeepCopy() {
        Person clone = (Person)this.MemberwiseClone();
        clone.IdInfo = new IdInfo(IdInfo.IdNumber);
        clone.Name = String.Copy(Name);
        return clone;
    }
}
```

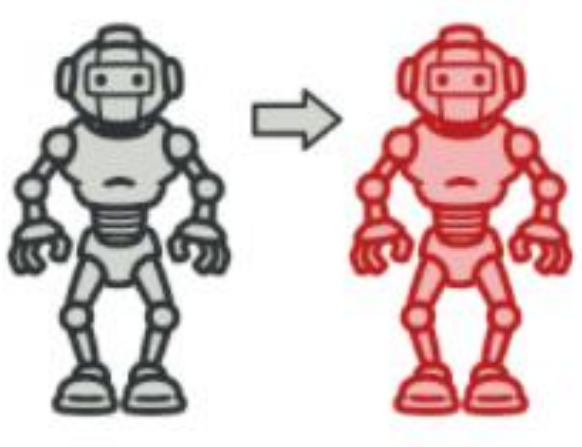
Output - Copy

```
Person p1 = new Person();
p1.Age = 42;
p1.BirthDate = Convert.ToDateTime("1977-01-01");
p1.Name = "Jack Daniels";
p1.IdInfo = new IdInfo(666);

// Perform a shallow copy of p1 and assign it to p2.
Person p2 = p1.ShallowCopy();
// Make a deep copy of p1 and assign it to p3.
Person p3 = p1.DeepCopy();

// Display values of p1, p2 and p3.
Console.WriteLine("Original values of p1, p2, p3:");
Console.WriteLine("   p1 instance values: ");
DisplayValues(p1);
Console.WriteLine("   p2 instance values:");
DisplayValues(p2);
Console.WriteLine("   p3 instance values:");
DisplayValues(p3);

// Change the value of p1 properties and display the values of p1,
// p2 and p3.
p1.Age = 32;
p1.BirthDate = Convert.ToDateTime("1900-01-01");
p1.Name = "Frank";
p1.IdInfo.IdNumber = 7878;
Console.WriteLine("\nValues of p1, p2 and p3 after changes to p1:");
Console.WriteLine("   p1 instance values: ");
DisplayValues(p1);
Console.WriteLine("   p2 instance values (reference values have changed):");
DisplayValues(p2);
Console.WriteLine("   p3 instance values (everything was kept the same):");
DisplayValues(p3);
```

```
Original values of p1, p2, p3:
   p1 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01.01.77
      ID#: 666
   p2 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01.01.77
      ID#: 666
   p3 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01.01.77
      ID#: 666

Values of p1, p2 and p3 after changes to p1:
   p1 instance values:
      Name: Frank, Age: 32, BirthDate: 01.01.00
      ID#: 7878
   p2 instance values (reference values have changed):
      Name: Jack Daniels, Age: 42, BirthDate: 01.01.77
      ID#: 7878
   p3 instance values (everything was kept the same):
      Name: Jack Daniels, Age: 42, BirthDate: 01.01.77
      ID#: 666
```

IClonable interface in C#

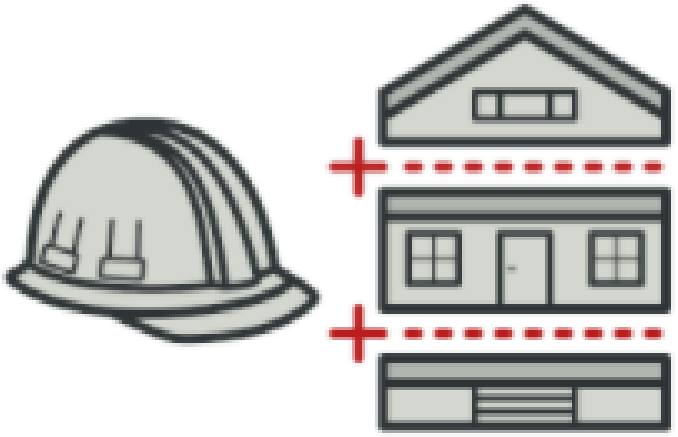
- enables you to provide a customized implementation that creates a copy of an existing object



- contains one member, the Clone method, which is intended to provide cloning support

  beyond that supplied by Object.MemberwiseClone.

https://docs.microsoft.com/en-us/dotnet/api/system.icloneable?view=net-5.0



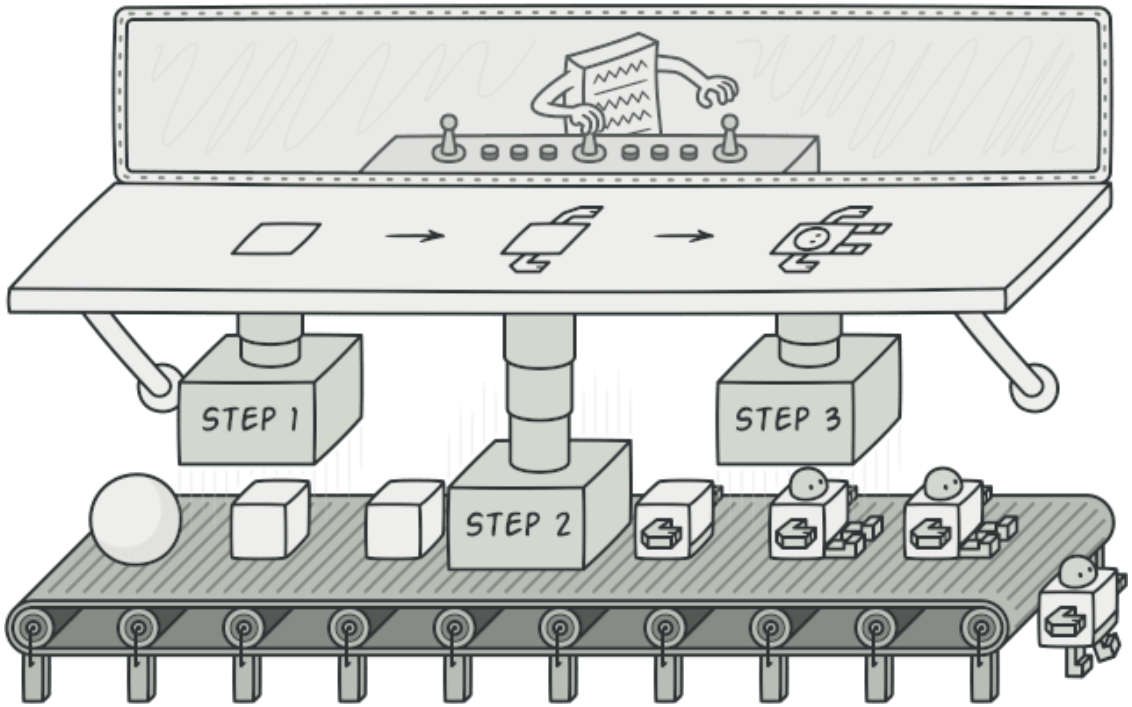
Builder Pattern

construct complex objects step by step

produce different types and representations of an object using the same construction code

Builder Pattern

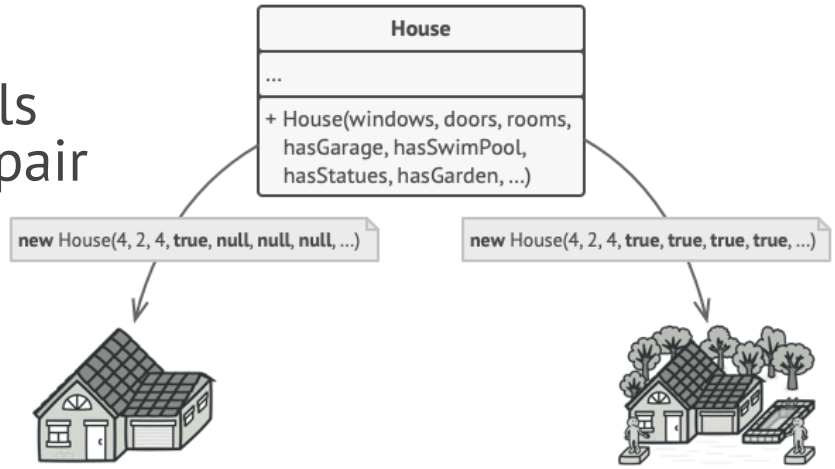
- Creating a complex object that requires laborious, step-by-step initialization of many fields and nested objects

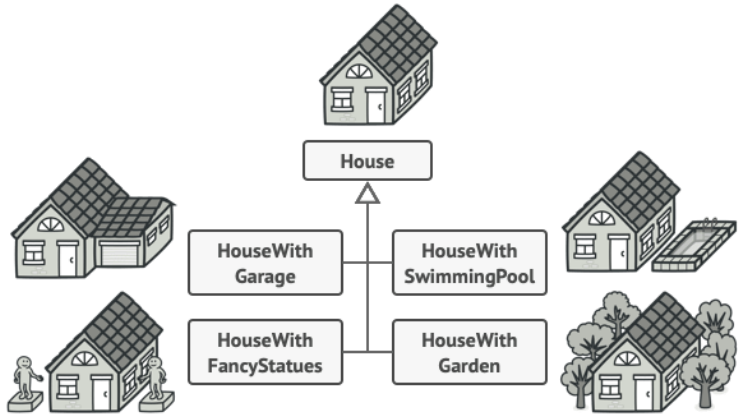


Problem

- build a simple house
  - you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof
  - *creating a subclass for every possible configuration of an object.*

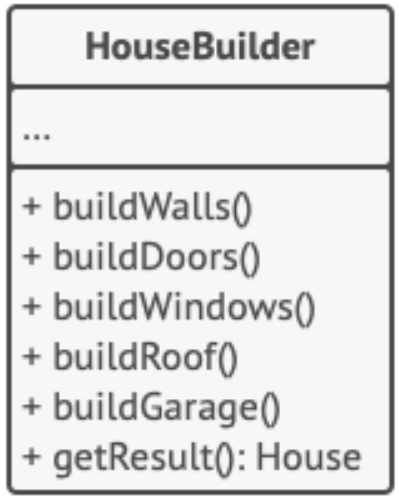
    or
  - *a constructor with lots of parameters: but not all the parameters are needed at all times*

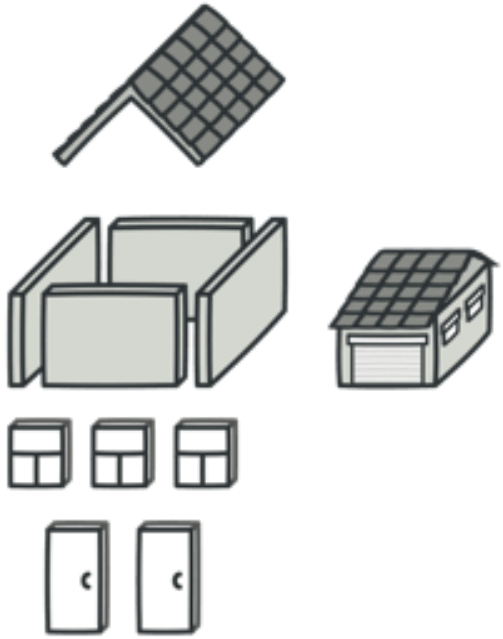




Solution

- extract the object construction code out of its own class and move it to separate objects called *builders*

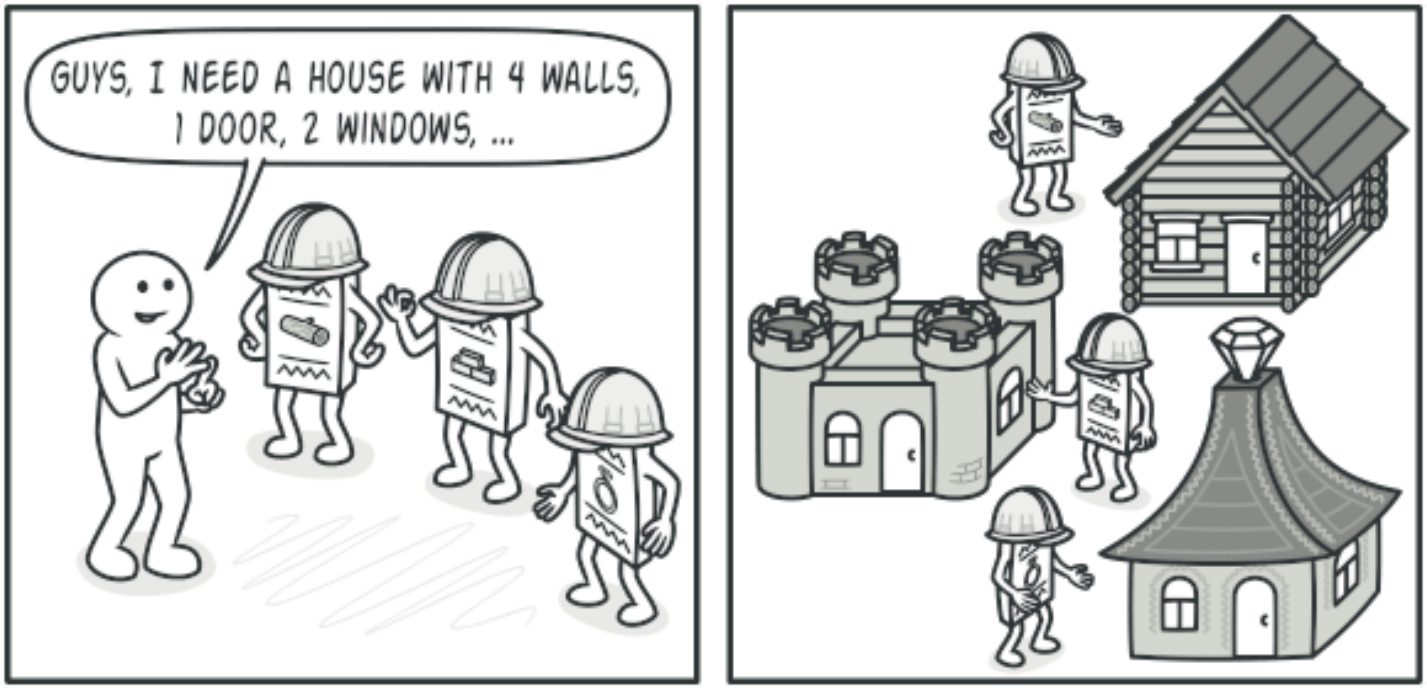




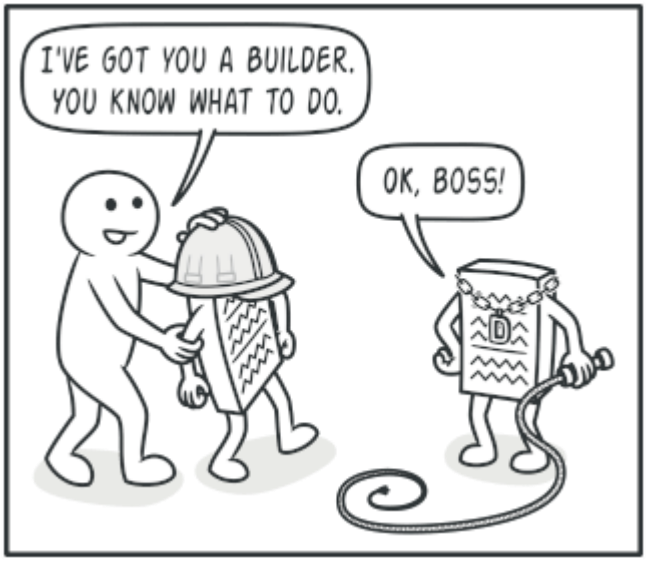
- important part is
- don’t need to call all of the steps
- only those steps that are necessary for producing a particular configuration of an object

Several different Builder Classes

- Different builders execute the same task in various ways



One Director

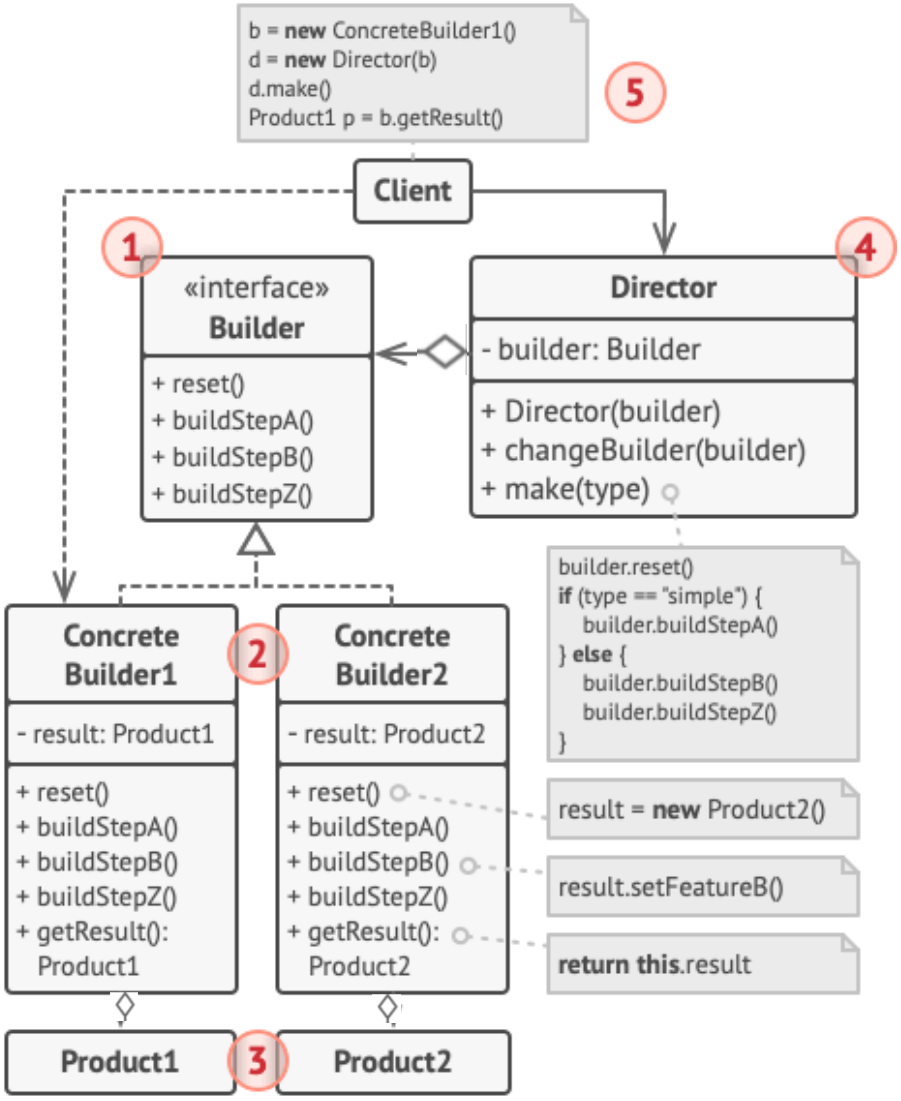


- *knows which building steps to execute to get a working product.*

- defines the order in which to execute the building steps, while the builder provides the implementation for those steps

- having a director class isn’t strictly necessary

Structure



1. Builder interface

- declares product construction steps
- that are common to all types of builders

2. Concrete Builders

- provide different implementations
- may produce products
  that don't follow the common interface

3. Products

- are resulting objects
- constructed by different builders
- don't have to belong to
  the same class hierarchy or interface

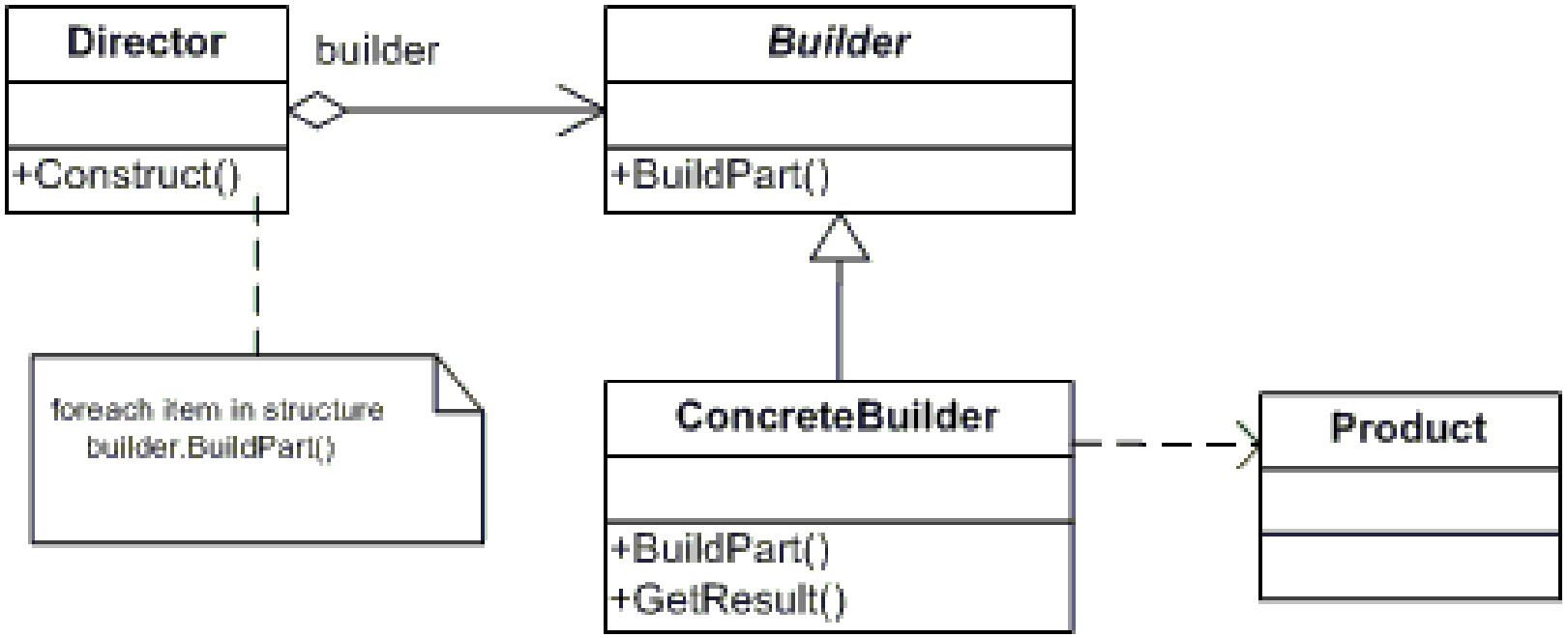
4. Director

- defines the order in which to call construction steps
- can create/reuse specific configurations of products

5. Client

- must associate one of the builder objects with the director → director uses that builder object for all further construction
- alternative approach
  client passes the builder object to the production method of the director → use a different builder each time you produce something with the director

UML – Builder


Director
builder
Builder
+Construct()
+BuildPart()
foreach item in structure
builder.BuildPart()
ConcreteBuilder
+BuildPart()
+GetResult()
Product

Building

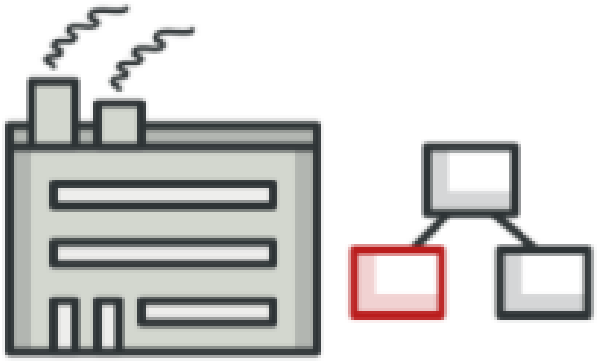
```
public class Building
{
    public string Name { get; set; }
    public int Floors { get; set; }
    public bool HasElevator { get; set; }

    public override string ToString()
    {
        return $"Building: {Name}, Floors: {Floors}, Has Elevator: {HasElevator}";
    }
}
```

BuildingBuilder

```
public class BuildingBuilder {
    private readonly Building _building;
    public BuildingBuilder() {
        _building = new Building();
    }
    public BuildingBuilder SetName(string name) {
        _building.Name = name;
        return this;
    }
    public BuildingBuilder SetFloors(int floors) {
        _building.Floors = floors;
        return this;
    }
    public BuildingBuilder SetHasElevator(bool hasElevator) {
        _building.HasElevator = hasElevator;
        return this;
    }
    public Building Build() {
        return _building;
    }
}
```

Using the Builder

```
public Building GetSchool() {
    return new BuildingBuilder()
        .SetName("Local School")
        .SetFloors(3)
        .SetHasElevator(true) // BuildingBuilder
        .Build(); // Building
}

public Building GetFireStation() {
    return new BuildingBuilder()
        .SetName("Fire Station")
        .SetFloors(2)
        .SetHasElevator(false) // BuildingBuilder
        .Build(); // Building
}

public Building GetHospital() {
    return new BuildingBuilder()
        .SetName("City Hospital")
        .SetFloors(6)
        .SetHasElevator(true) // BuildingBuilder
        .Build(); // Building
}
```

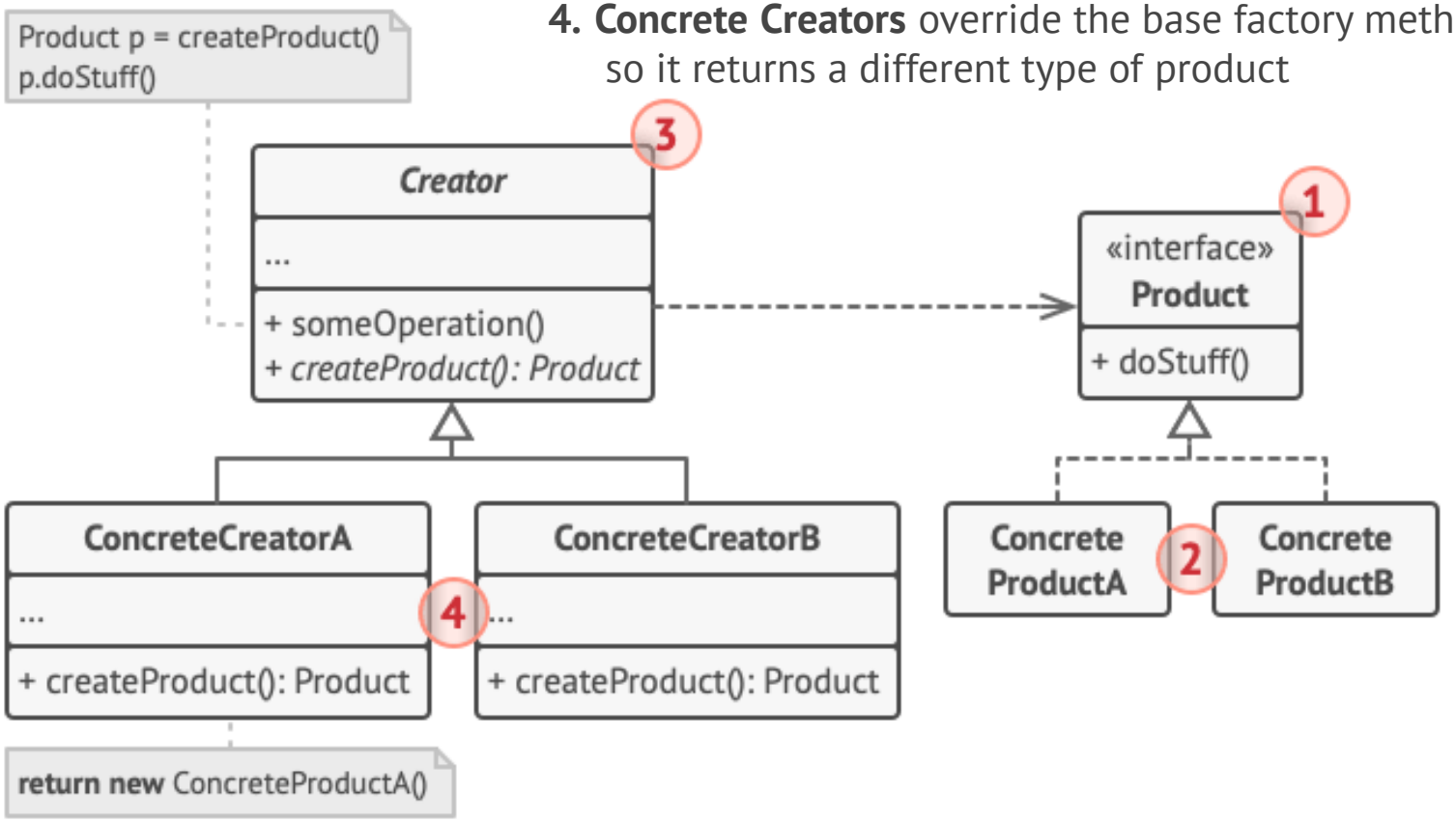


Factory Pattern

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

Structure

1. **Product** declares the interface, which is common to all product objects
2. **Concrete Products** implement the Product interface
3. **Creator** declares the factory method that returns new product objects.
4. **Concrete Creators** override the base factory method so it returns a different type of product



Building Factory

```
public class BuildingFactory {
    public Building GetSchool() {...}
    public Building GetFireStation() {
        return new Building {
            Name = "Fire Station",
            Floors = 2,
            HasElevator = false
        };
    }
    public Building GetHospital() {
        return new Building {
            Name = "City Hospital",
            Floors = 6,
            HasElevator = true
        };
    }
}
```

```
public class Building
{
    public string Name { get; init; }
    public int Floors { get; init; }
    public bool HasElevator { get; init; }
}

var factory = new BuildingFactory();

var school:Building = factory.GetSchool();
var fireStation:Building = factory.GetFireStation();
var hospital:Building = factory.GetHospital();
```

Building Factory

as Singleton

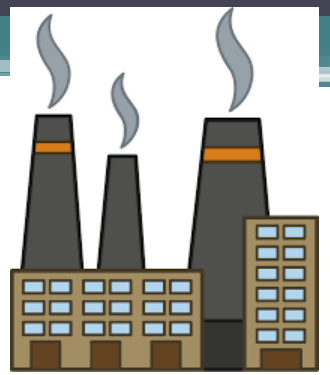
```
public class BuildingFactory {
    private static BuildingFactory _instance;
    private BuildingFactory() { }
    public static BuildingFactory Instance {
        get {
            if (_instance == null) _instance = new BuildingFactory();
            return _instance;
        }
    }
    public Building GetSchool() {...}
    public Building GetFireStation() {...}
    public Building GetHospital() {
        return new BuildingBuilder()
            .SetName("City Hospital")
            .SetFloors(6)
            .SetHasElevator(true) // BuildingBuilder
            .Build(); // Building
    }
```

using the BuildingBuilder

Get Instance from Factory...

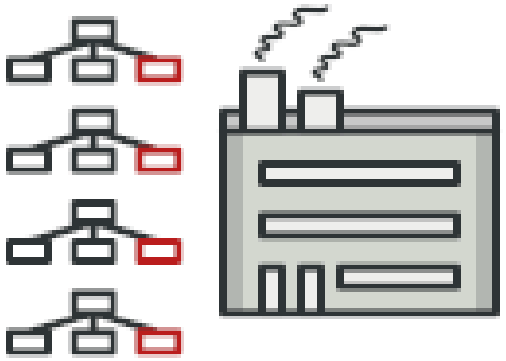
```
var factory = BuildingFactory_SB.Instance;

var school:Building = factory.GetSchool();
var fireStation:Building = factory.GetFireStation();
var hospital:Building = factory.GetHospital();

Console.WriteLine(school);
Console.WriteLine(fireStation);
Console.WriteLine(hospital);
```

Applicability



- **Use the Abstract Factory**
  - **when your code needs to work with various families of related products,**
  - **but you don't want it to depend on the concrete classes of those products**

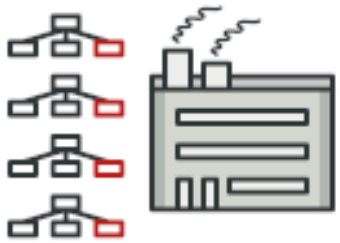
  - Consider implementing the Abstract Factory when you have a class with a set of **Factory Methods** that blur its primary responsibility.
    - In a well-designed program *each class is responsible only for one thing*.
    - When a class deals with multiple product types, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown Abstract Factory implementation.

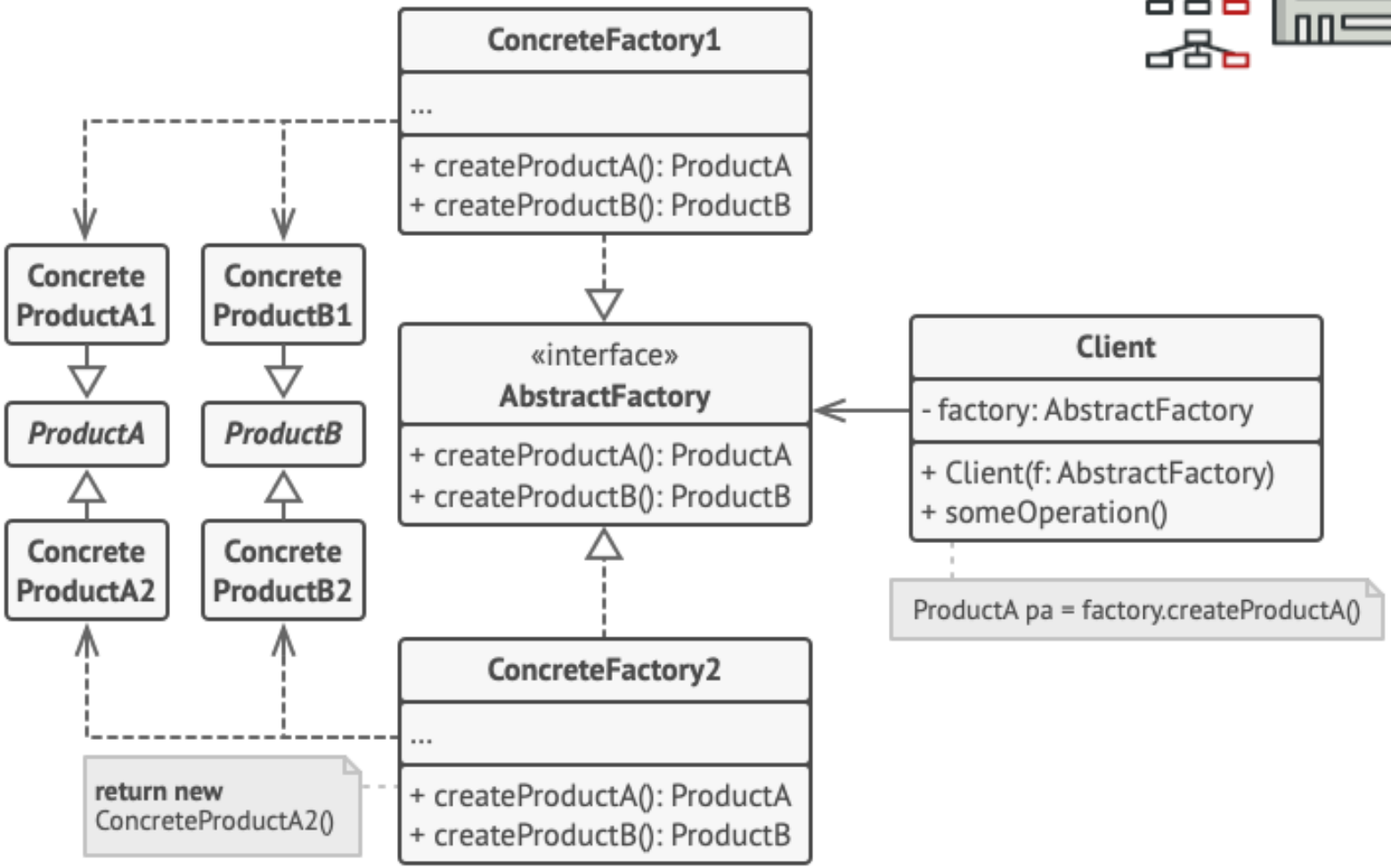


Abstract Factory

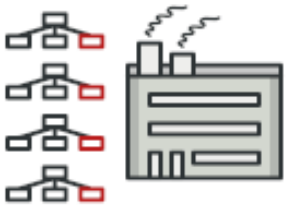
lets you produce families of related objects without specifying their concrete classes

Structure

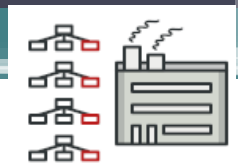



ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Participants in Abstract Factory

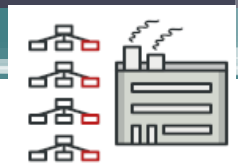


- **AbstractFactory**
  - declares an interface for operations that create abstract products
- **ConcreteFactory**
  - implements the operations to create concrete product objects
- **AbstractProduct**
  - declares an interface for a type of product object
- **Product**
  - defines a product object to be created by the corresponding concrete factory
  - implements the AbstractProduct interface
- **Client**
  - uses interfaces declared by AbstractFactory and AbstractProduct classes



Abstract & Concrete Product

```
// Abstract Factory
public abstract class ABuildingFactory {
    public abstract Building CreateSchool();
    public abstract Building CreateFireStation();
    public abstract Building CreateHospital();
}

// Konkrete Implementierung für städtische Gebäude
public class UrbanBuildingFactory : ABuildingFactory {...}

// Konkrete Implementierung für ländliche Gebäude
public class RuralBuildingFactory : ABuildingFactory {...}
```



Concrete Factory

```
// Konkrete Implementierung für städtische Gebäude
public class UrbanBuildingFactory : ABuildingFactory {
    public override Building CreateSchool() {
        return new Building {
            Name = "Urban School",
            Floors = 5,
            HasElevator = true
        };
    }
    public override Building CreateFireStation() {
        return new Building {
            Name = "Urban Fire Station",
            Floors = 3,
            HasElevator = true
        };
    }
    public override Building CreateHospital() {
        return new Building {
            Name = "Urban Hospital",
            Floors = 10,
            HasElevator = true
        };
    }
}
```

```
// Konkrete Implementierung für ländliche Gebäude
public class RuralBuildingFactory : ABuildingFactory {
    public override Building CreateSchool() {
        return new Building {
            Name = "Rural School",
            Floors = 2,
            HasElevator = false
        };
    }
    public override Building CreateFireStation() {
        return new Building {
            Name = "Rural Fire Station",
            Floors = 1,
            HasElevator = false
        };
    }
    public override Building CreateHospital() {
        return new Building {
            Name = "Rural Hospital",
            Floors = 4,
            HasElevator = true
        };
    }
}
```

Using Concrete Factories

```
ABuildingFactory urbanFactory = new UrbanBuildingFactory();
ABuildingFactory ruralFactory = new RuralBuildingFactory();

// Städtische Gebäude
var urbanSchool :Building = urbanFactory.CreateSchool();
var urbanFireStation :Building = urbanFactory.CreateFireStation();
var urbanHospital :Building = urbanFactory.CreateHospital();

Console.WriteLine($"Urban School: {urbanSchool}");
Console.WriteLine($"Urban Fire Station: {urbanFireStation}");
Console.WriteLine($"Urban Hospital: {urbanHospital}");

// Ländliche Gebäude
var ruralSchool :Building = ruralFactory.CreateSchool();
var ruralFireStation :Building = ruralFactory.CreateFireStation();
var ruralHospital :Building = ruralFactory.CreateHospital();

Console.WriteLine($"Rural School: {ruralSchool}");
Console.WriteLine($"Rural Fire Station: {ruralFireStation}");
Console.WriteLine($"Rural Hospital: {ruralHospital}");
```

Pizza Builder & Factory

- Create a PizzaBuilder, a Pizza has some default Incredience: Tomatosauce, Cheese
- Create a AddIncredient, use a Baseclass and derive from it for each concrete Incredient
- Implement the Factory as Singleton
- Write a PizzaFactory with 5 Pizza like:
  - Tonno
  - Quattro Fromaggie
  - Hawaii
  - Salami
  - Calzone
- Create each Pizza once for testing

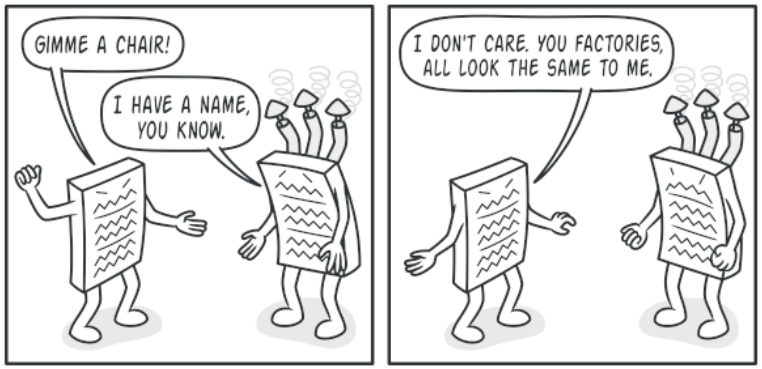
Pizza

```
public abstract class PizzaIngredient {
public string Name { get; }
protected PizzaIngredient(string name) {
  Name = name;
}
public override string ToString() {
  return Name; }
} // Derived concrete ingredient classes

public class TomatoSauce : PizzaIngredient {
  public TomatoSauce() : base("Tomato Sauce") { }

}
```

Abstract Factory VS Builder Pattern

Abstract Factory

- Emphasises objacts creation operations for families of related objects,
  - when each family is a set classess derived from a common base
- each object is returned **immediately** as a result of one call

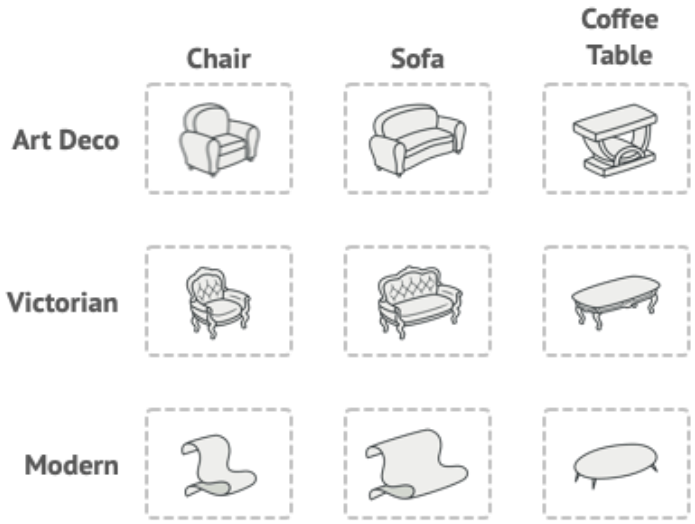


Builder Pattern

- Focuses on constructing a complex object **step by step**
- Formulates the logic of how to put together a complex object:
  - **Builder** object encapsulates configuration of the complex object
  - **Director** object knows the protocol of using the Builder, where the protocol defines all logical steps required to build the complex object

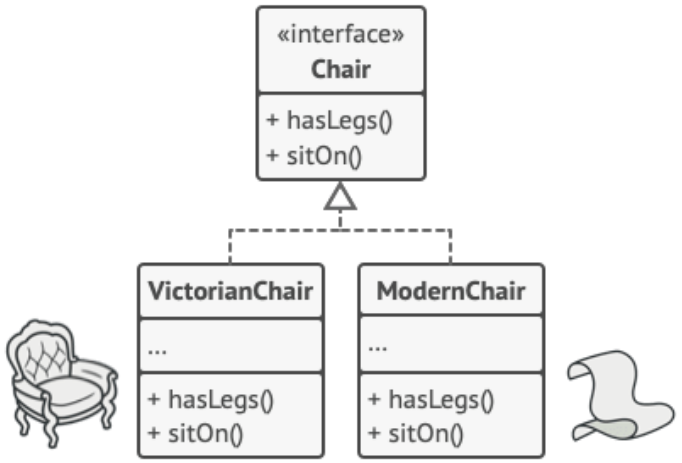
Problem

- Imagine a furniture shop simulator:
- A family of related products
  - Chair
  - Sofa
  - CoffeeTable
- Several variants of this family, products Chair + Sofa + CoffeeTable are available in these variants:
  - Modern
  - Victorian
  - ArtDeco



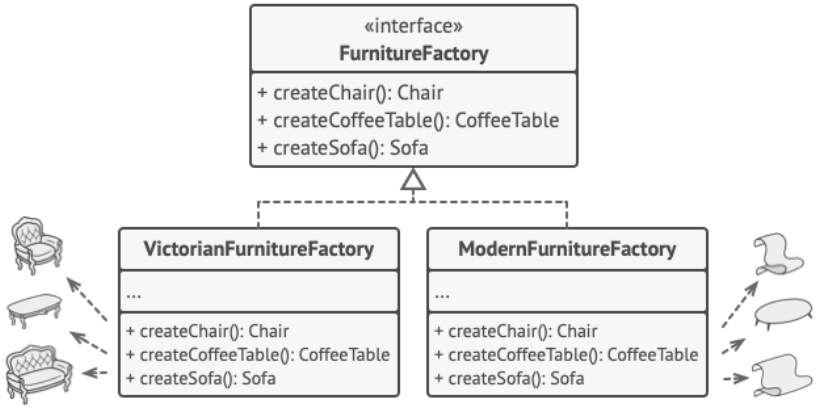
Create individual furniture objects that they match other objects of the same family

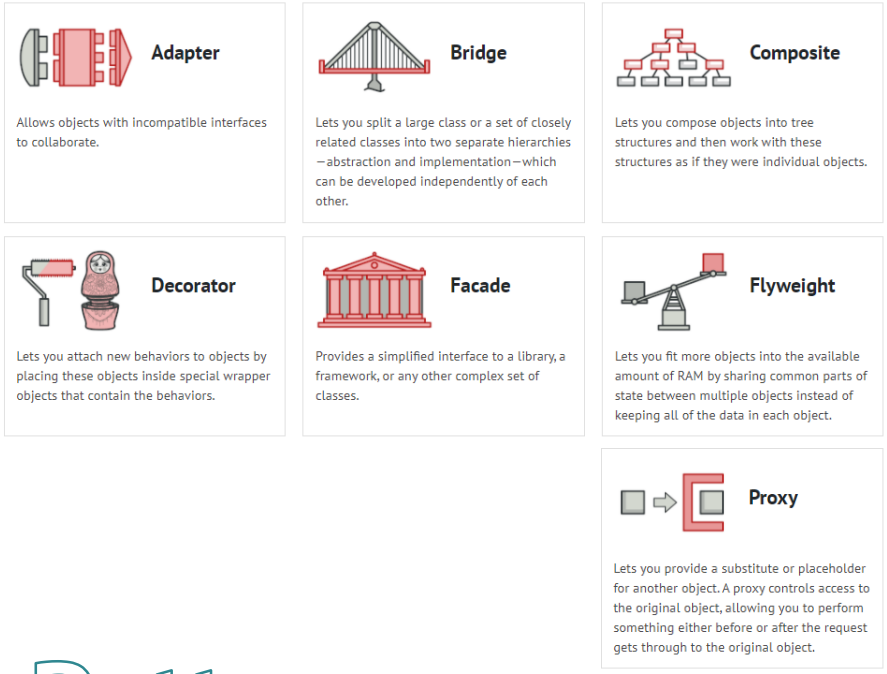
Excercise:



- Abstract Factory Pattern
  - explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table)
  - make all variants of products follow those interfaces
    - all chair variants can implement the Chair interface;
    - all coffee table variants can implement the CoffeeTable interface
    - and so on.

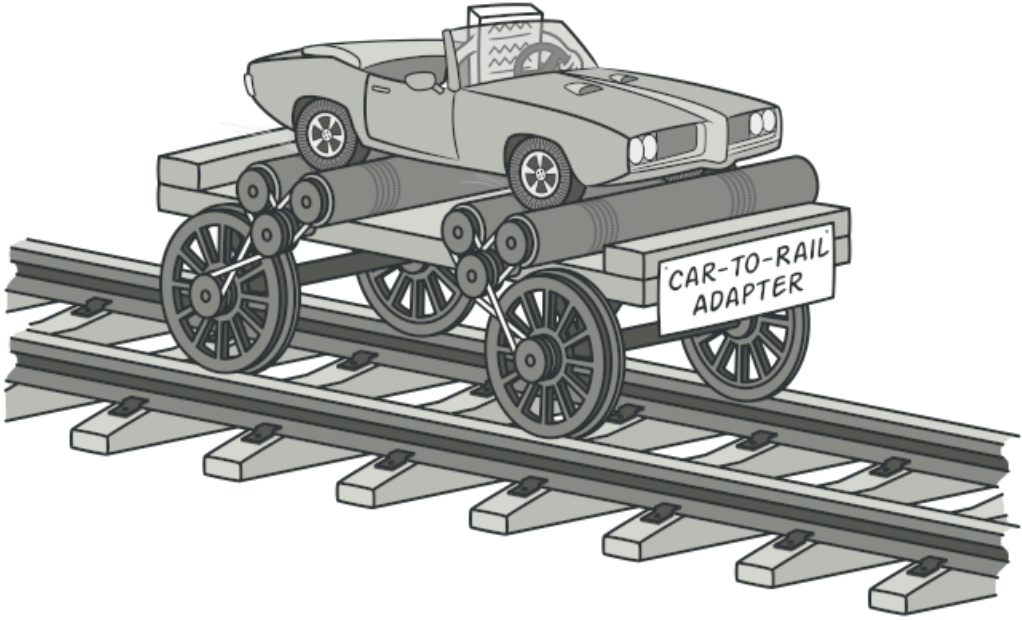
  - declare the Abstract Factory
    - an interface with a list of creation methods for all products that are part of the product family
    - for example, createChair, createSofa and createCoffeeTable
    - methods must return abstract product types represented by the interfaces we extracted previously:
      - Chair, Sofa, CoffeeTable and so on.





Structural Design Patterns

explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



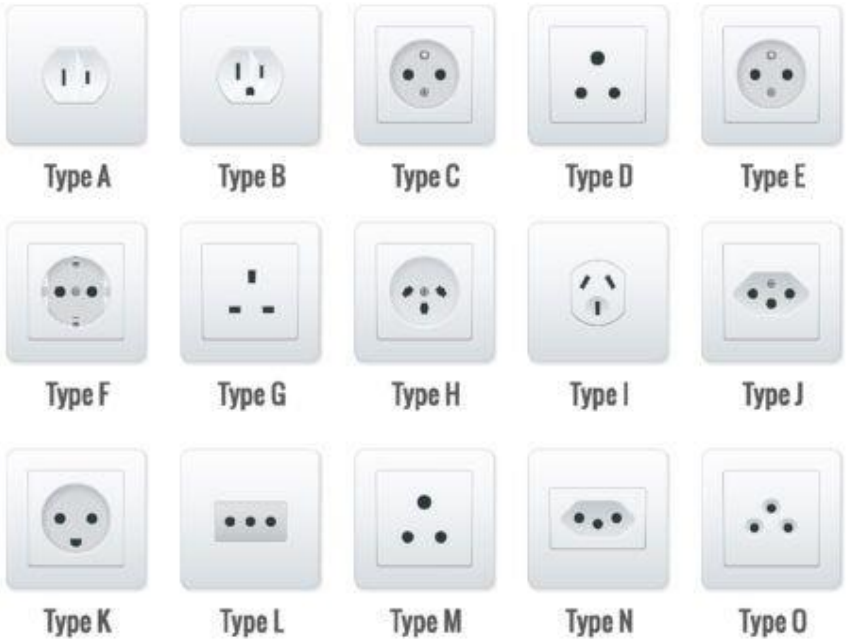
Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate

Real World

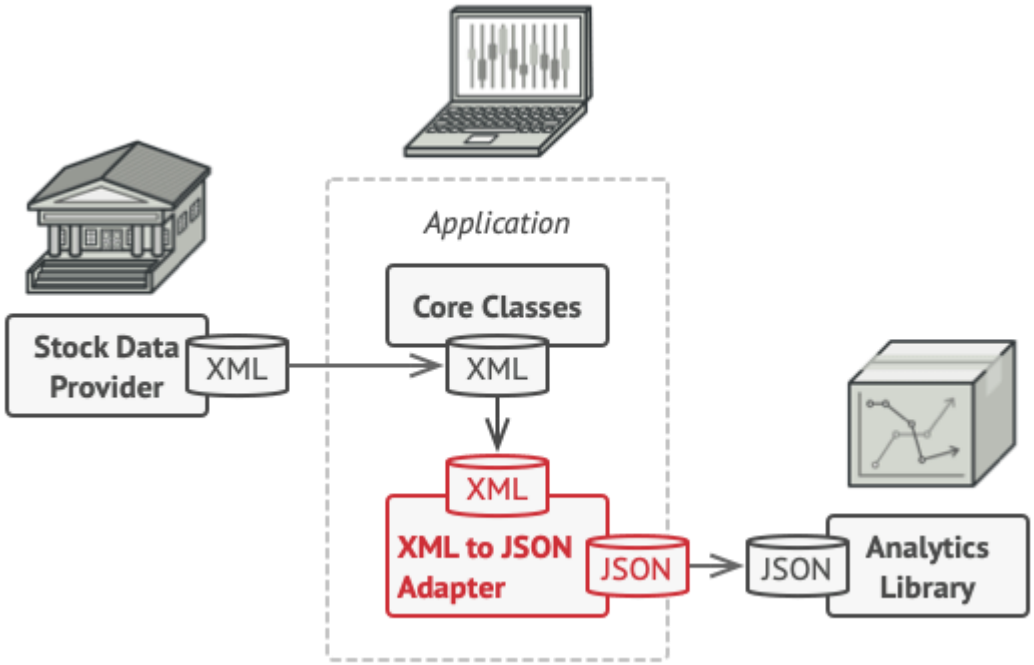
The power plug and sockets standards are different in different countries. That's why your plug won't fit an US socket.

The problem can be solved by using a power plug adapter.



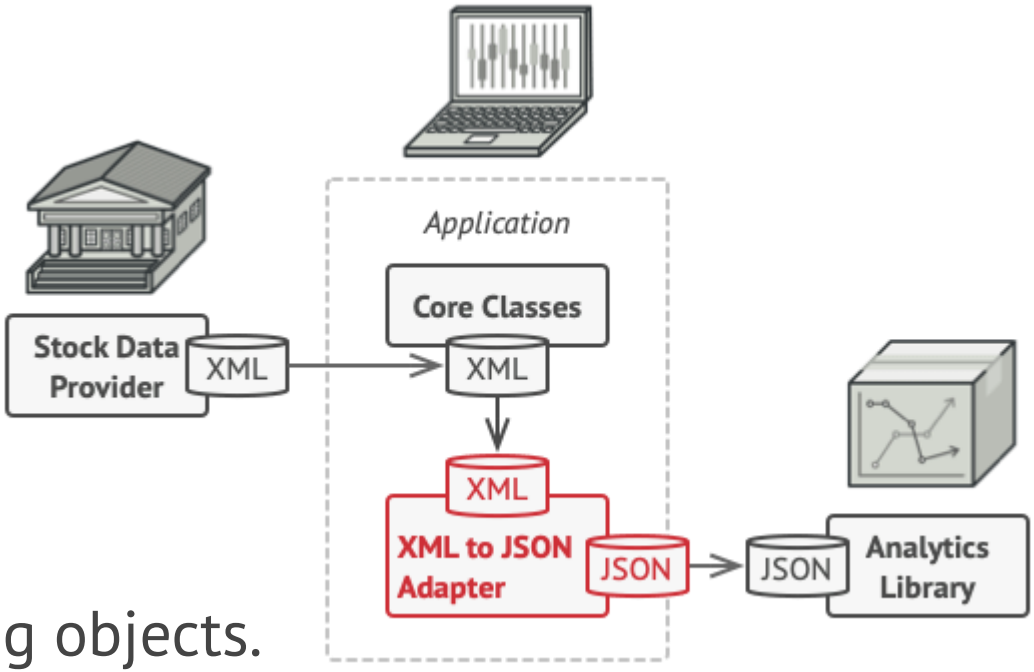
Problem

- Imagine that you're creating a stock market monitoring app
- App downloads in XML format and displays charts and diagrams for users



- At some point, you want to integrate a smart 3rd party analytics library, but the analytics library only works with data in JSON format.

Solution



1. The adapter gets an interface, compatible with one of the existing objects.

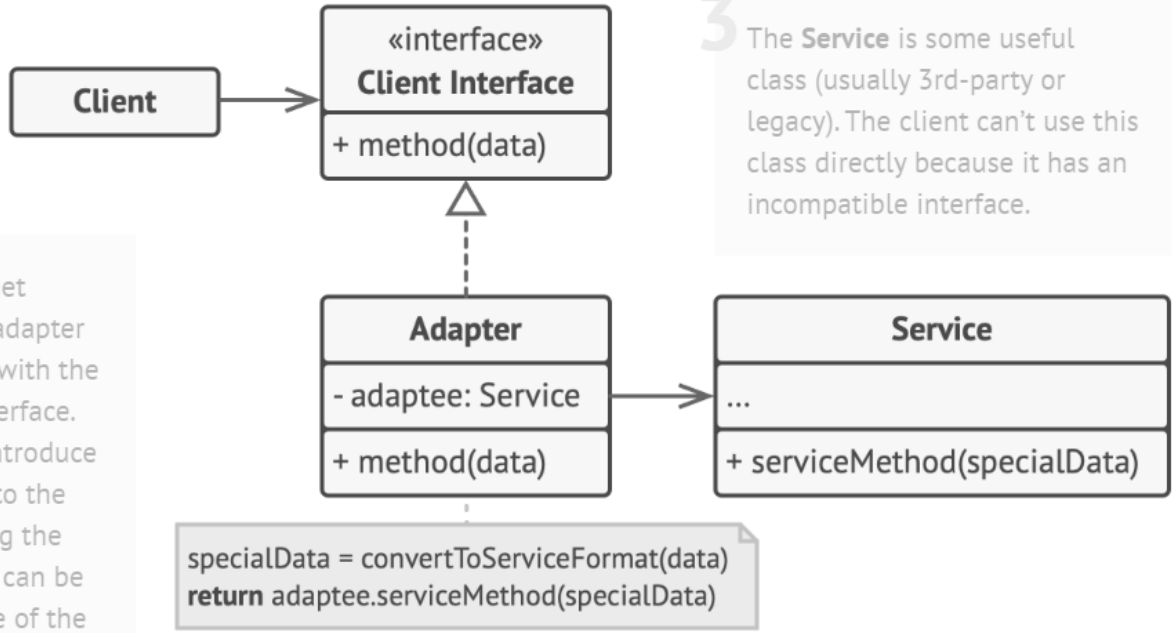
2. Using this interface, the existing object can safely call the adapter's methods.

3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

UML

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.

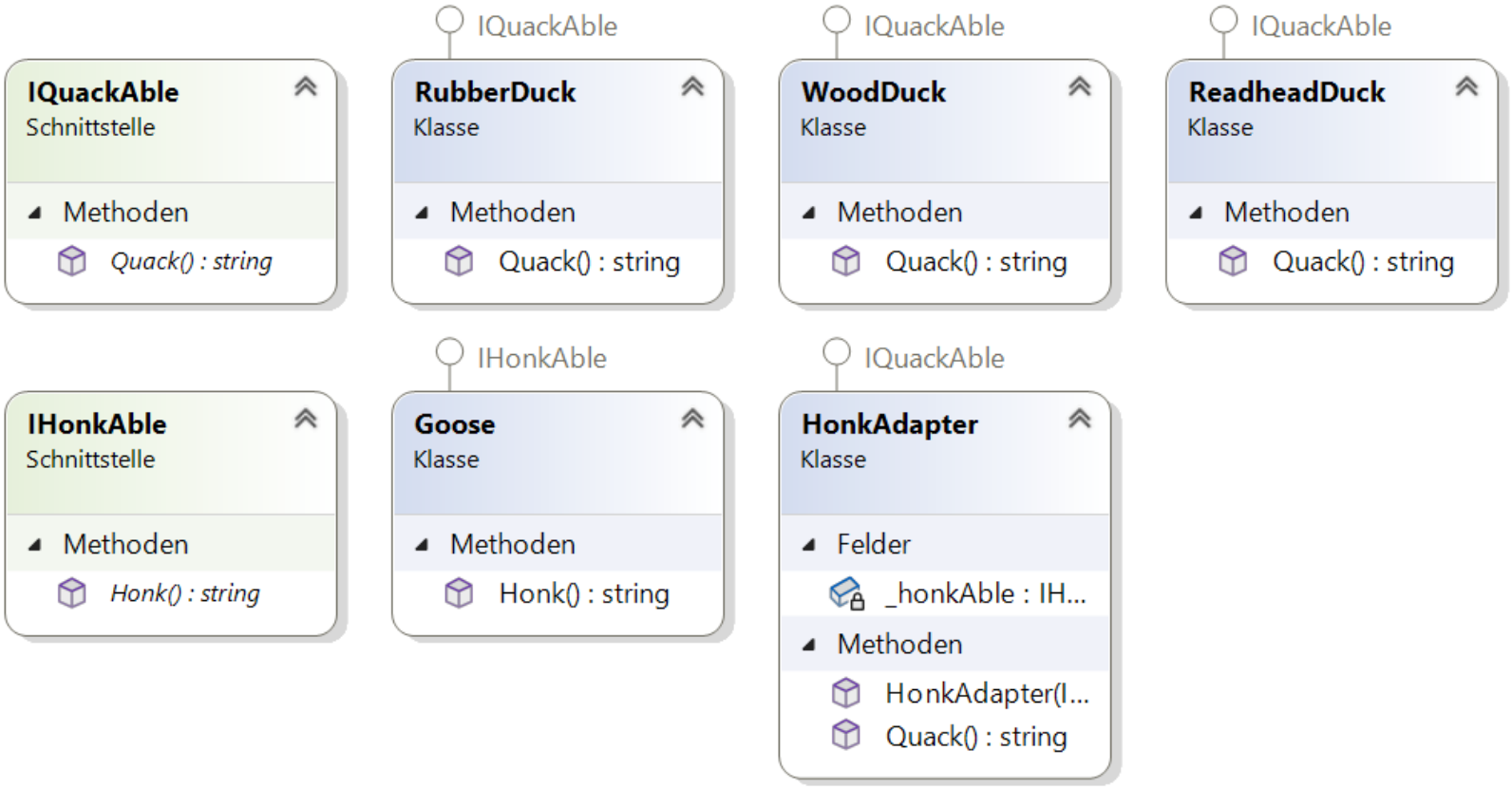
4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it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

Goose Adapter

The most common sound made by geese is **the honk**, which is similar to a duck's quack or a chicken's cluck

IQuackable & IHonkable



Quack

```
public interface IQuackAble {
    string Quack();
}

public class ReadheadDuck : IQuackAble {
    public string Quack() => "Quack";
}

public class RubberDuck : IQuackAble {
    public string Quack() => "Squeak";
}

public class WoodDuck : IQuackAble {
    public string Quack() => "quwook";
}
```

Honk

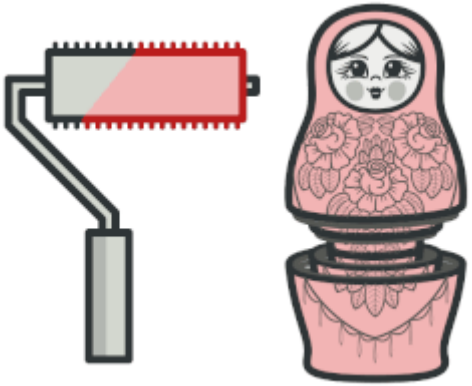
```
public interface IHonkAble {
    string Honk();
}

public class Goose : IHonkAble{
    public string Honk() {
        return "Honk";
    }
}
```

Goose Adapter

```
public class HonkAdapter : IQuackAble{
    // HonkToQuackAdapter
    // Apapter implements the target interface
    // -> Adapt the interface

    private readonly IHonkAble _honkAble;

    public HonkAdapter(IHonkAble honkable) {
        _honkAble = honkable;
    }

    public string Quack() => _honkAble.Honk();
}
```

Test Quackables

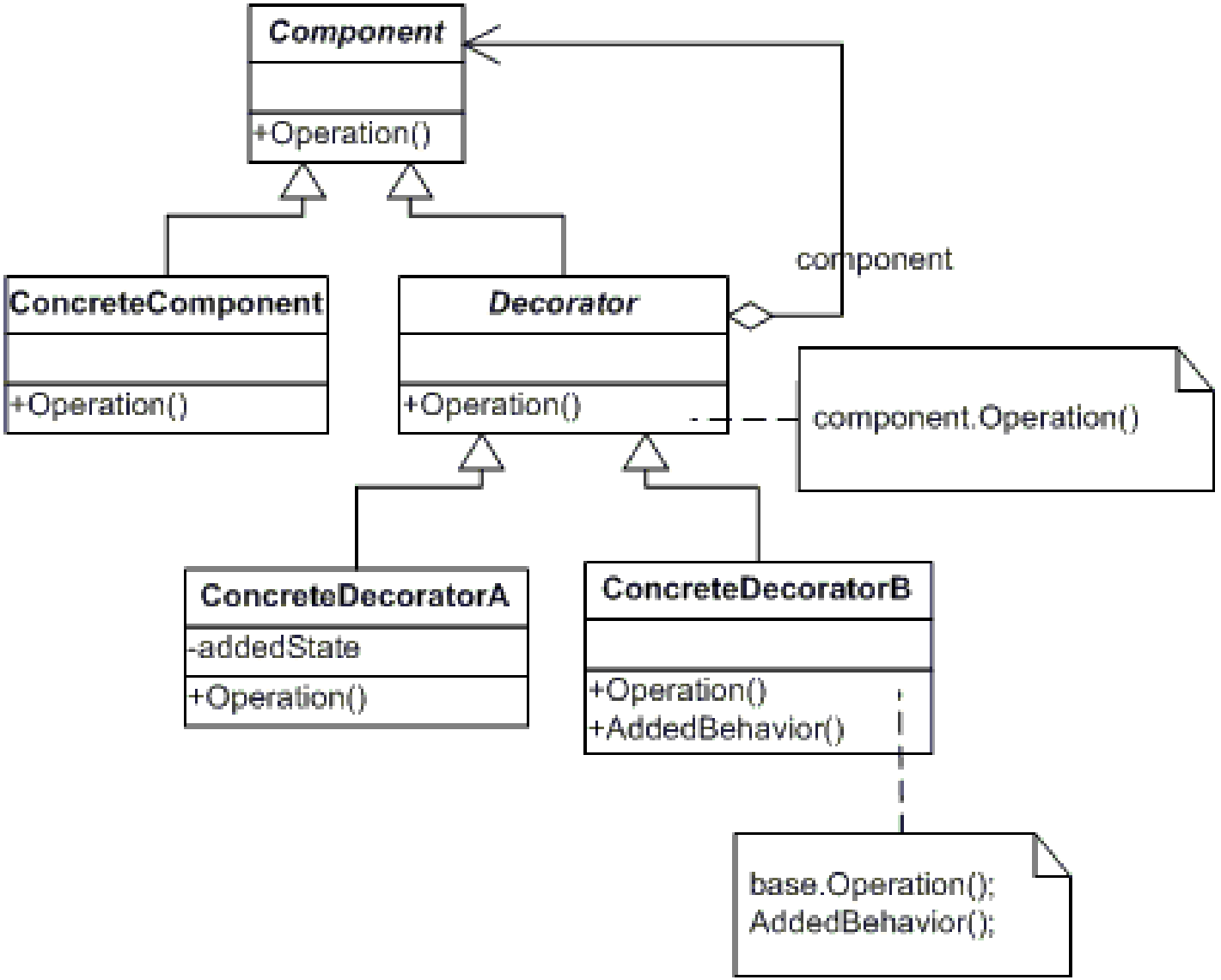
```
List<IQuackAble> quackAbles = new() {
    new ReadheadDuck(), new RubberDuck(),
    new WoodDuck(), new HonkAdapter(new Goose())};

var data = quackAbles.Select(q => q.Quack());
```



Decorator Pattern

attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors

UML - Decorator



IQuackable Decorator

```
public interface IQuackAble {
    string Quack();
}
```

- Count the amount of quacks independently who‘s quacking

```
public class QuackCountDecorator : IQuackAble {

    private readonly IQuackAble _quackAble;

    public static int Counter;

    public QuackCountDecorator(IQuackAble quackAble) {
        _quackAble = quackAble;
    }

    public string Quack() {
        Counter++;
        return _quackAble.Quack();
    }
}
```

Count the Quacking

```
List<IQuackAble> quackAbles = new() {
    new QuackCountDecorator(new ReadheadDuck()),
    new QuackCountDecorator(new RubberDuck()),
    new WoodDuck(),
    new QuackCountDecorator(new HonkAdapter(new Goose()))
};

var data = quackAbles.Select(q => q.Quack()).ToList();

Assert.That(quackAbles, Has.Count.EqualTo(4));
Assert.That(QuackCountDecorator.Counter, Is.EqualTo(3));
```

Quackcount Extention-Method

- An alternativ to the decorator would be an extention method, which counts the quacks

```
public static class QuackExtensions
{
    private static int quackCount = 0;

    public static string QuackAndCount(this IQuackable duck)
    {
        quackCount++;
        return $"{duck.Quack()} (Quack Count: {quackCount})";
    }

    public static int GetQuackCount()
    {
        return quackCount;
    }
}
```

Coffee Decorator Example

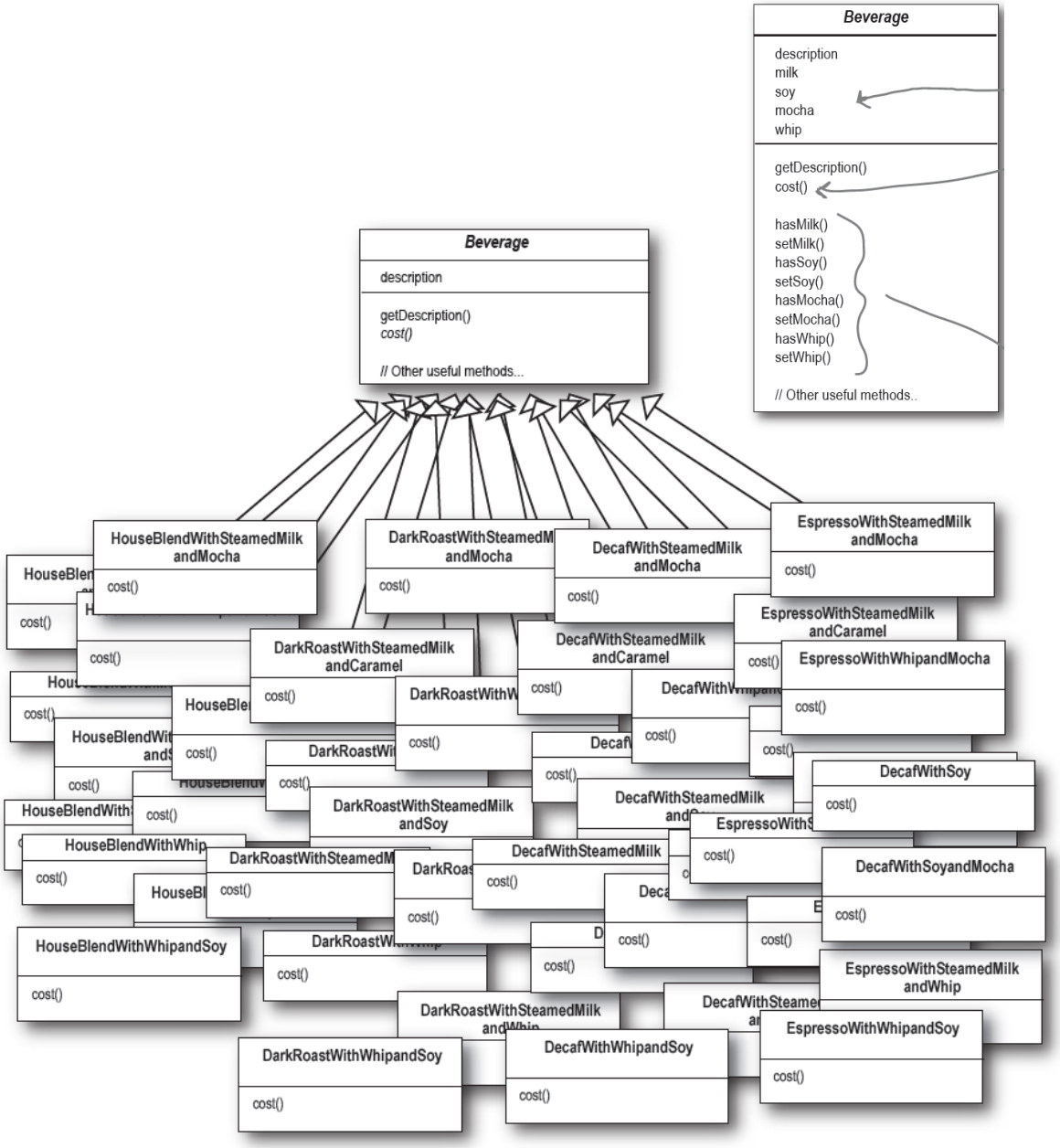
- How do you like your coffee?
- Do you even like coffee?
- Want some tee or coco?

Problem

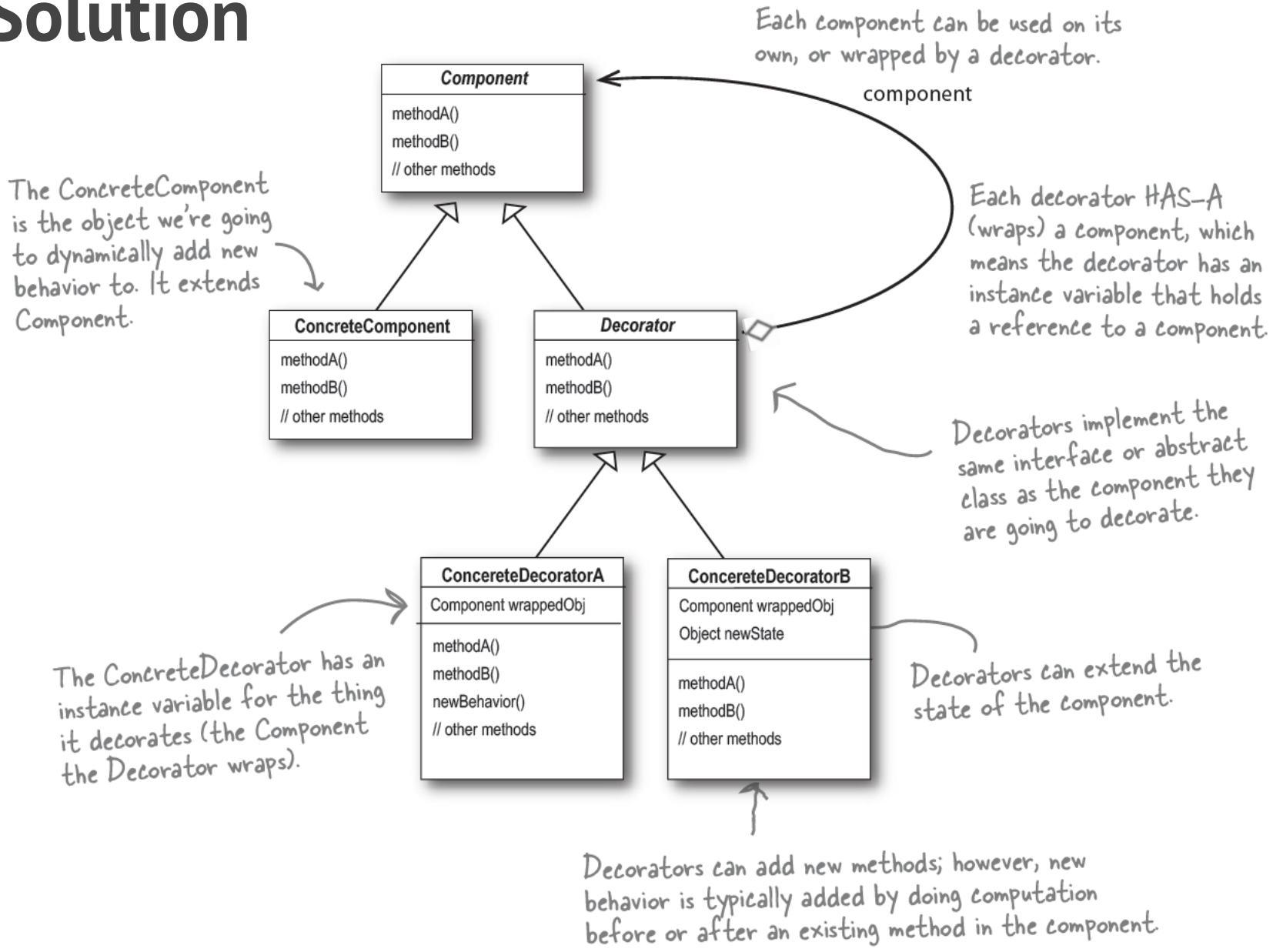
- Image you want to order a hot beverage… and add…

  e.g.: Sugar, Milk, Sojamilk, …

  all different kind of beverage should be able to be ordered and calculated



Solution


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Main & Output

```
static void Main(string[] args) {
    Console.WriteLine("Test Decorator");

    ABaverage coffee = new Coffee();
    coffee = new Sugar(coffee);
    coffee = new Milk(coffee);
    coffee = new MilkFoam(coffee);
    Console.WriteLine(coffee.GetDescription());
    Console.WriteLine(coffee.GetPrice());
    Console.WriteLine();

    ABaverage tea = new Tea();
    tea = new Sugar(tea);
    tea = new Sugar(tea);
    tea = new Sugar(tea);
    Console.WriteLine(tea.GetDescription());
    Console.WriteLine(tea.GetPrice());
    Console.WriteLine();

    ABaverage kakao = new Kakao();
    kakao = new Milk(kakao);
    kakao = new Milk(kakao);
    kakao = new MilkFoam(kakao);
    kakao = new Sugar(kakao);
    Console.WriteLine(kakao.GetDescription());
    Console.WriteLine(kakao.GetPrice());
    Console.WriteLine();
}
```

- Beverage can have a lots of different ingredient

```
Test Decorator
Kaffeepulver, Zucker, Milch, Milchschaum
0,85

Schwarztee, Zucker, Zucker, Zucker
0,7

Kakao, Milch, Milch, Milchschaum, Zucker
1,05
```

Beverage & Concrete Beverage

```
interface IBeverage {
    public string GetDescription();
    public double GetPrice();
}
```

```
abstract class ABeverage : IBeverage {
    protected string description;
    protected double price;

    protected ABeverage(string description, double price) {
        this.description = description;
        this.price = price;
    }
    public virtual string GetDescription() {
        return description;
    }
    public virtual double GetPrice() {
        return price;
    }

    public override string ToString() {
        return $"{description} kostet {price}";
    }
}
```

```
class Tea : ABaverage {
    public Tea()
        : base("Schwarztee", 0.40) { }
}
class Kakao : ABaverage {
    public Kakao()
        : base("Kakao", 0.60) { }
}
class Coffee : ABaverage {
    public Coffee()
        : base("Kaffeepulver", 0.50) { }
}
```

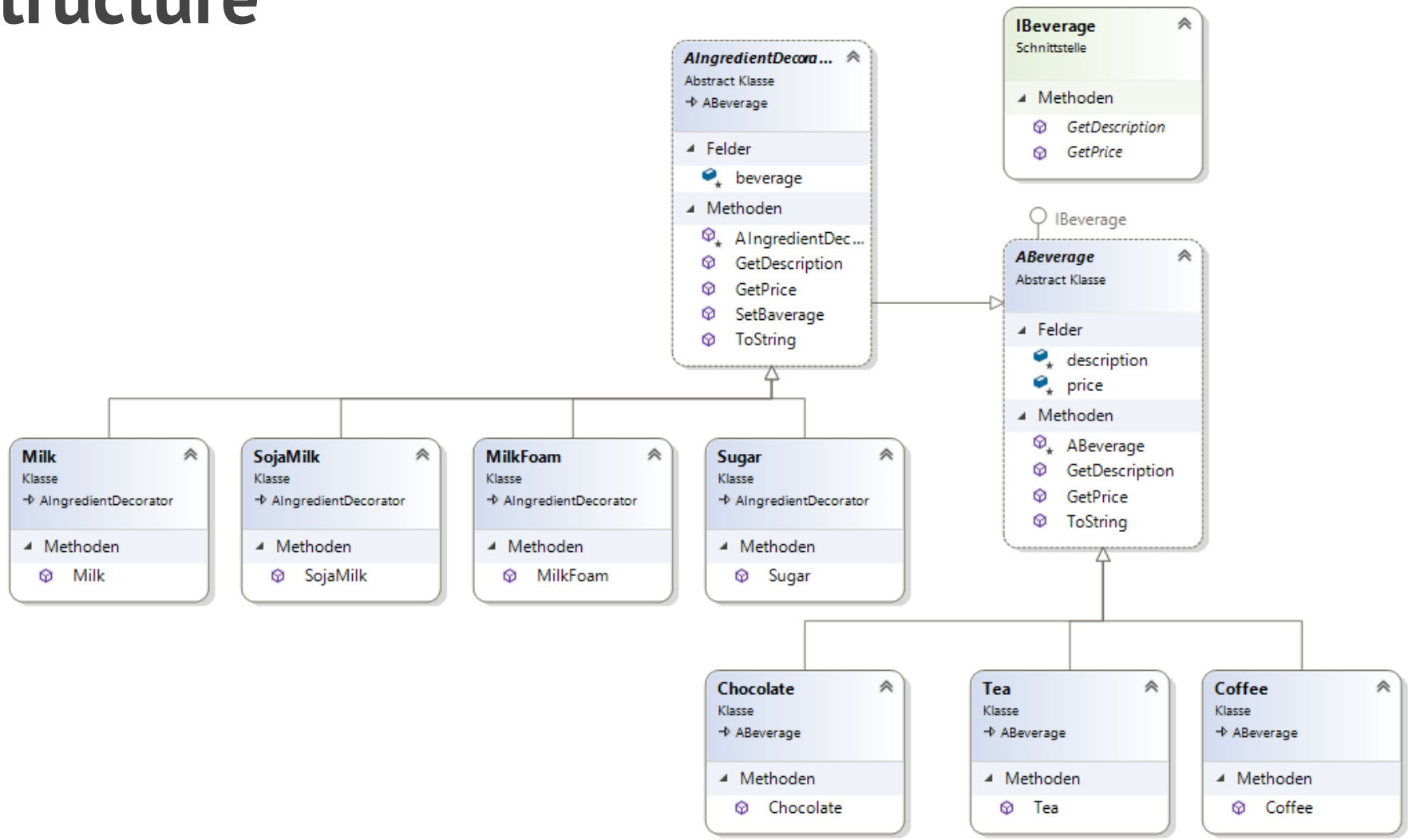
Ingredient as Decorator

```
abstract class AIngredientDecorator : ABeverage {
    protected IBeverage beverage;

    protected AIngredientDecorator(
        string description, double price,
        IBeverage baverage):base (description, price) {
            this.beverage = baverage;
    }
    public override string GetDescription() {
        return $"{beverage.GetDescription()}," +
            $" {this.description}";
    }
    public override double GetPrice() {
        return beverage.GetPrice() + price;
    }
    public void SetBaverage(IBeverage baverage) {
        this.beverage = baverage;
    }
    public override string ToString() {
        return $"{GetDescription()} " +
            $"kostet {GetPrice()}";
    }
}
```

Ingredient
is derived from
Beverage and has a
Beverage

```
class Milk : AIngredientDecorator {
    public Milk(IBeverage beverage)
        : base("Milch", 0.10, beverage) { }
}
class Sugar : AIngredientDecorator {
    public Sugar(IBeverage beverage)
        : base("Zucker", 0.10, beverage) { }
}
class MilkFoam : AIngredientDecorator {
    public MilkFoam(IBeverage beverage)
        : base("Milchschaum", 0.15, beverage) { }
}
class SojaMilk : AIngredientDecorator {
    public SojaMilk(IBeverage beverage)
        : base("Sojamilch", 0.20, beverage) { }
}
```

Structure


IBeverage
Schnittstelle
Methoden
GetDescription
GetPrice
AIngredientDecora...
Abstract Klasse
ABeverage
Felder
beverage
Methoden
AIngredientDec...
GetDescription
GetPrice
SetBaverage
ToString
IBeverage
ABeverage
Abstract Klasse
Felder
description
price
Methoden
ABeverage
GetDescription
GetPrice
ToString
Milk
Klasse
AIngredientDecorator
Methoden
Milk
SojaMilk
Klasse
AIngredientDecorator
Methoden
SojaMilk
MilkFoam
Klasse
AIngredientDecorator
Methoden
MilkFoam
Sugar
Klasse
AIngredientDecorator
Methoden
Sugar
Chocolate
Klasse
ABeverage
Methoden
Chocolate
Tea
Klasse
ABeverage
Methoden
Tea
Coffee
Klasse
ABeverage
Methoden
Coffee

CoffeeDispencer

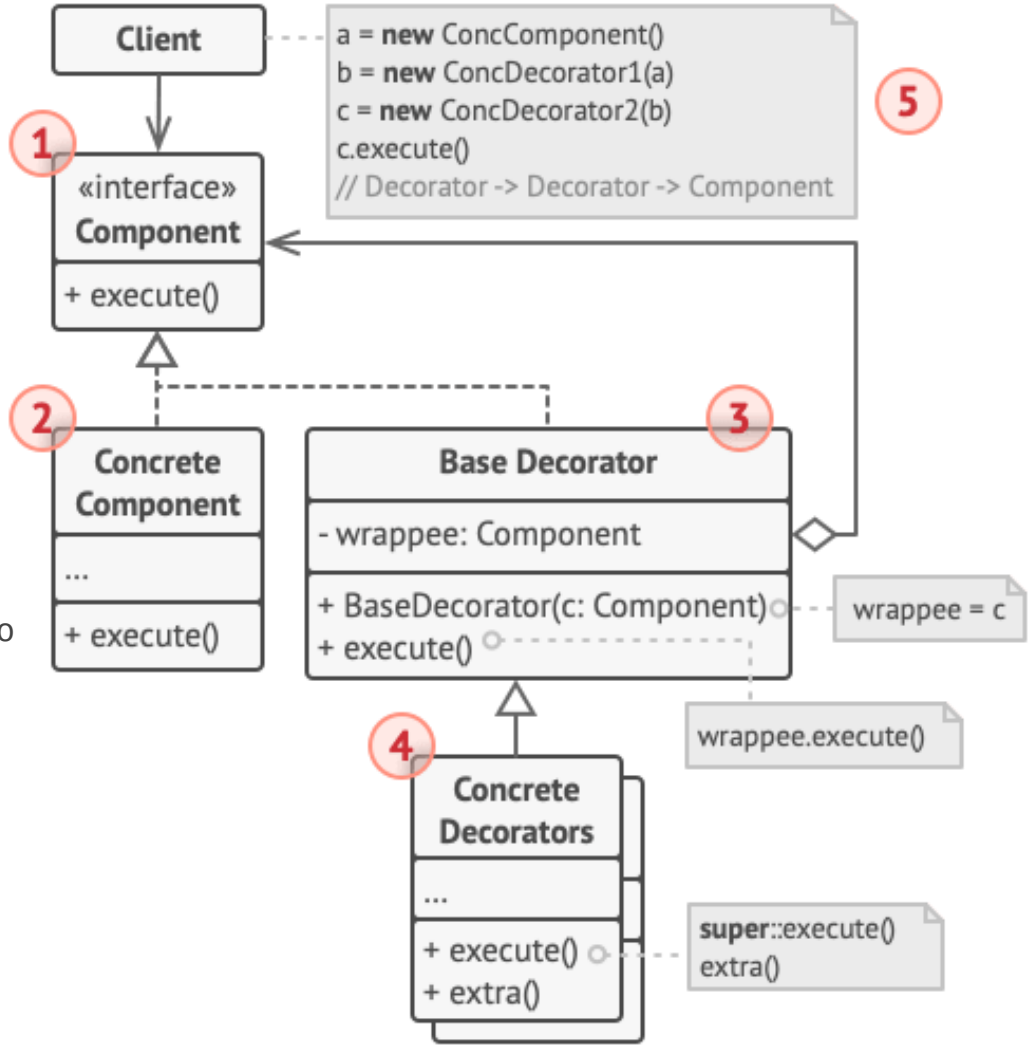
```
class CoffeeDispenser {
    private int amountOfCoffee;
    private int amountOfSugar;
    private int amountOfMilk;
    private int amountOfTea;
    private int amountOfChocolate;
    private double curAccountBalance; //verfügbares Geld

    public void Refill(int milk, int sugar,
        int coffee, int tea, int chocolate) ...
    DecreaseAmounts

    private IBeverage AddMilkSugar(IBeverage c) {
        DecreaseMilk();
        DecreaseSugar();
        c = new Sugar(c);
        c = new Milk(c);
        return c;
    }

    public IBeverage GetCoffee() {
        DecreaseCoffee();
        IBeverage c = new Coffee();
        c = AddMilkSugar(c);
        curAccountBalance += c.GetPrice();
        return c;
    }
    public IBeverage GetTea() ...
    public IBeverage GetChocolate() ...
    public double GetCurAccBalance() ...
}
```

```
CoffeeDispenser coffeeD = new CoffeeDispenser();
coffeeD.Refill(100, 100, 100, 100, 100);
IBeverage coffee1 = coffeeD.GetCoffee();
IBeverage coffee2 = coffeeD.GetCoffee();
IBeverage tea1 = coffeeD.GetTea();
IBeverage tea2 = coffeeD.GetTea();
IBeverage coco1 = coffeeD.GetChocolate();
IBeverage coco2 = coffeeD.GetChocolate();

IBeverage[] drinks = new IBeverage[6];
drinks[0] = coffee1;
drinks[1] = coffee2;
drinks[2] = tea1;
drinks[3] = tea2;
drinks[4] = coco1;
drinks[5] = coco2;

double price = 0;
for (int i = 0; i < 6; i++) {
    Console.WriteLine(drinks[i].ToString());
    price += drinks[i].GetPrice();
}
Console.WriteLine("Gesamtpreis: " + price + " Cent");
Console.WriteLine("Kontostand: " + coffeeD.GetCurAccBalance() + " Cent");
```

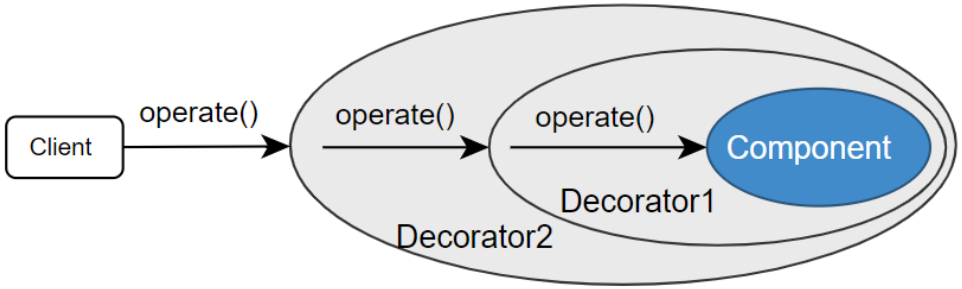
```
Kaffeepulver, Zucker, Milch kostet 0,7
Kaffeepulver, Zucker, Milch kostet 0,7
Schwarztee, Zucker, Milch kostet 0,6
Schwarztee, Zucker, Milch kostet 0,6
Kakao, Zucker, Milch kostet 0,7999999999999999
Kakao, Zucker, Milch kostet 0,7999999999999999
Gesamtpreis: 4,2 Cent
Kontostand: 4,2 Cent
```

Structure

1. **Component**
   - declares the common interface for both wrappers and wrapped objects
2. **Concrete Component**
   - defines the basic behavior, which can be altered by decorators
3. **Base Decorator**
   - has a field for referencing a wrapped object
   - field's type should be declared as the component interface so it can contain both concrete components and decorators
   - base decorator delegates all operations to the wrapped object
4. **Concrete Decorators**
   - defines extra behaviors that can be added to components dynamically
   - override methods of the base decorator
   - execute their behavior either before or after calling the parent method
5. **Client**
   - can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface



Main




```
public static void main(String[] args) {
    Component compA = new ConcreteComponentA();
    compA.operate();
    //ConcreteComponentA operates.

    compA = new ConcreteDecorator1(compA);
    compA.operate();
    //ConcreteComponentA operates.
    //ConcreteDecorator1 operates!

    Component compB = new ConcreteDecorator3(new ConcreteDecorator2(new ConcreteC
    compB.operate();
    //ConcreteComponentB operates.
    //ConcreteDecorator2 operates with a new State: 999
    //ConcreteDecorator3 operates with a new Operation: 12
}
```

ConcreteComponent

```java
public abstract class Component {
    //abstrakte Klasse oder Interface je nach Bedarf.
    public abstract void operate();
}
```

```java
public class ConcreteComponentA extends Component {
    public void operate() {
        System.out.println("ConcreteComponentA operates.");
    }
}

class ConcreteComponentB extends Component {
    public void operate() {
        System.out.println("ConcreteComponentB operates.");
    }
}
```

Decorator

```java
public abstract class Component {
    //abstrakte Klasse oder Interface je nach Bedarf.
    public abstract void operate();
}
```

```java
public abstract class Decorator extends Component {
    protected Component component;

    //Konstruktor zum komfortablen Initiieren am Client
    public Decorator(Component component) {
        this.component = component;
    }
    //operate() wird nicht implementiert.
}
```

ConcreteDecorator

```
class ConcreteDecorator1 extends Decorator {
    public ConcreteDecorator1(Component component) {
        super(component);
    }

    public void operate() {
        //operate() der dekorierten Kompontente aufrufen vor
        //oder nach der eigenen Funktionalitäst.
        component.operate();
        //eigene Funktionalitäst:
        System.out.println("ConcreteDecorator1 operates!");
    }

}
```

Concrete Decorator

```
class ConcreteDecorator2 extends Decorator {
    //ConcreteDecorator2 fügt der Komponente einen neuen Zustand hinzu.
    private int newState;

    public ConcreteDecorator2(Component component) {
        super(component);
        newState = 999;
    }

    public void operate() {
        component.operate();
        System.out.println("ConcreteDecorator2 operates with a new State: " + newS
    }
}
```

Concrete Decorator

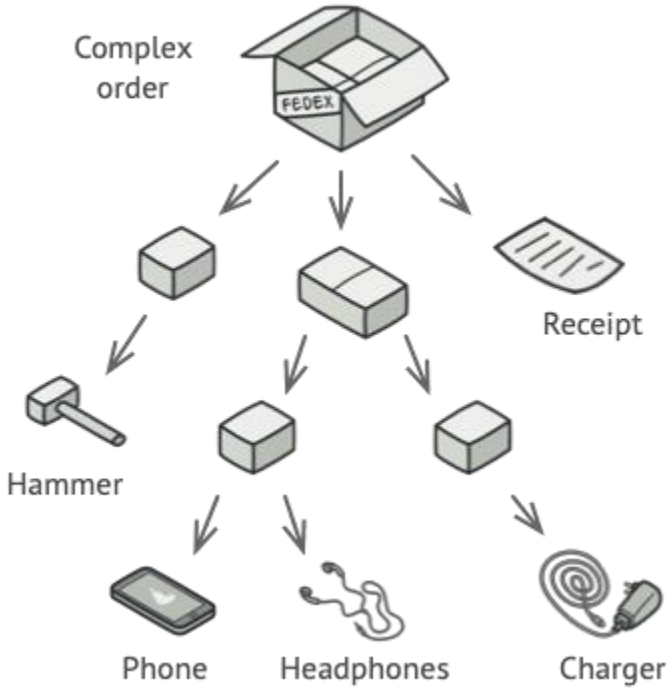
```
class ConcreteDecorator3 extends Decorator {
    public ConcreteDecorator3(Component component) {
        super(component);
    }

    public void operate() {
        component.operate();
        System.out.println("ConcreteDecorator3 operates with a new Operation: "+ne
    }

    //ConcreteDecorator3 fügt der Komponente eine neue Methode hinzu und
    //erweitert seine Schnittstelle nach auÃüen.
    public int newOperation() {
        return (int)(Math.random() * 20);
    }
}
```

Composite Pattern

compose objects into tree structures and then work with these structures as if they were individual objects

Problem

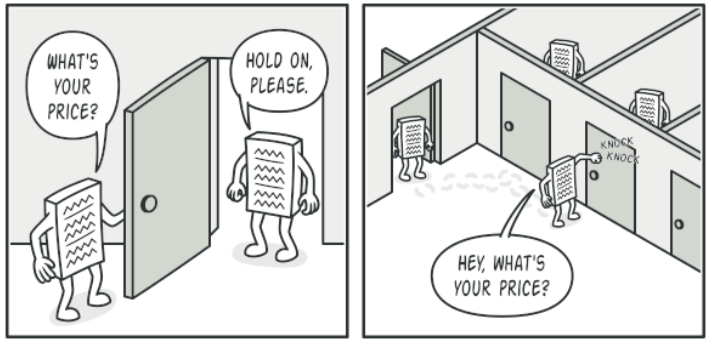
- imagine that you have two types of objects:
- **Products and Boxes**
  - a Box can contain several Products as well as a number of smaller Boxes
  - little Boxes can also hold some Products or even smaller Boxes
  - and so on



Solution

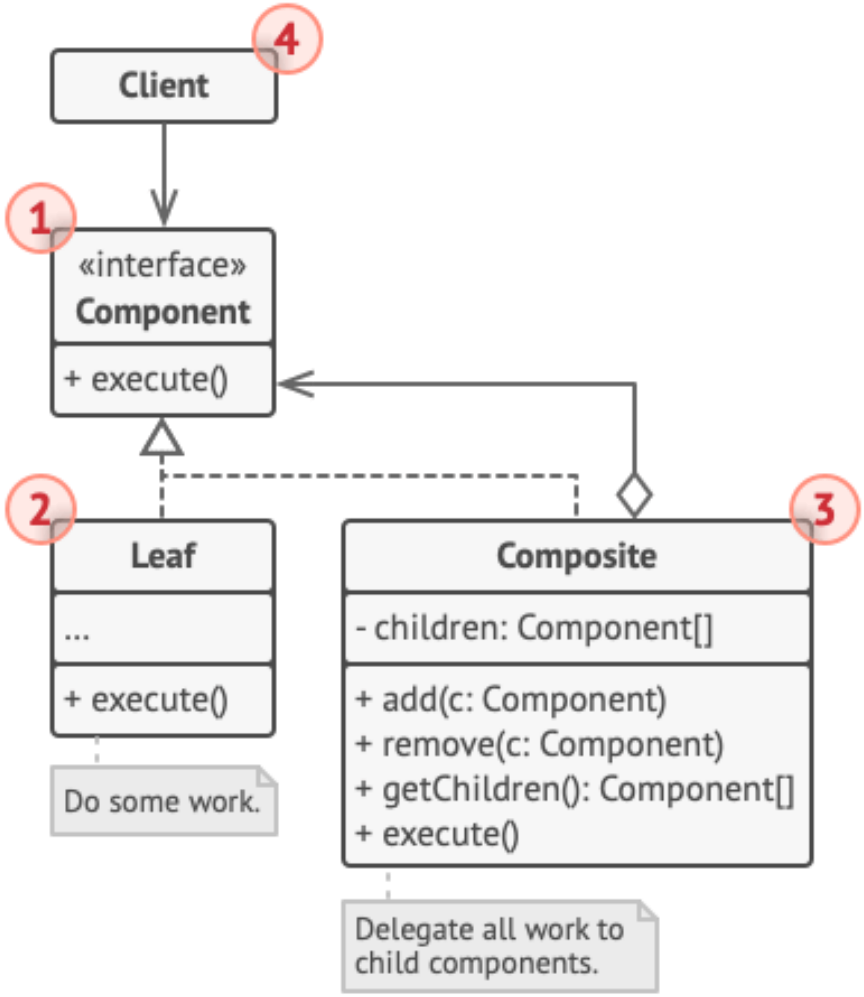
- work with Products and Boxes through a common interface which declares a method for calculating the total price

- How would this method work?
  - For a product, return the product's price.
  - For a box, it'd go over each item the box contains, ask its price and then return a total for this box.
    - If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated
    - A box could even add some extra cost to the final price, such as packaging cost

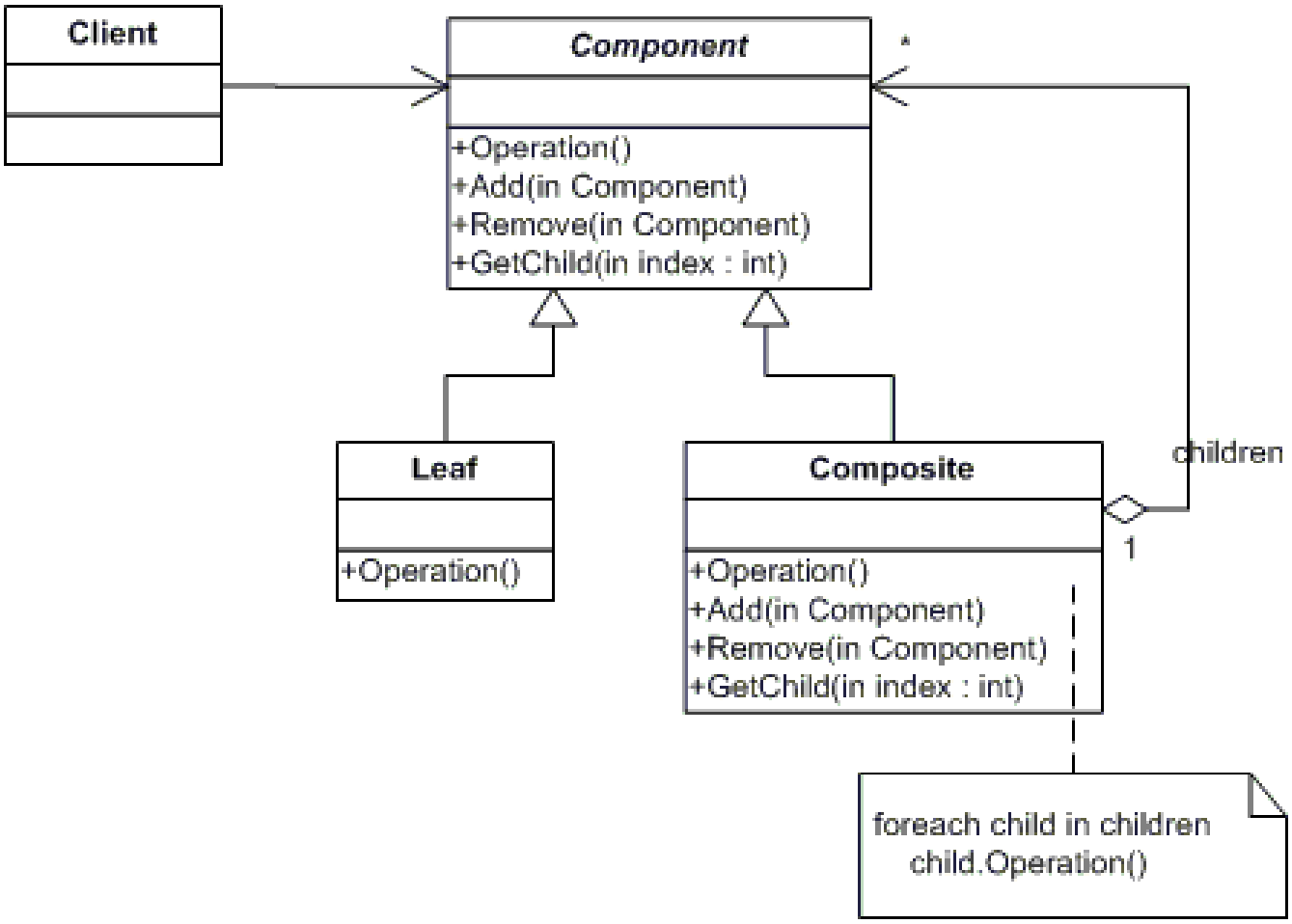


Structure

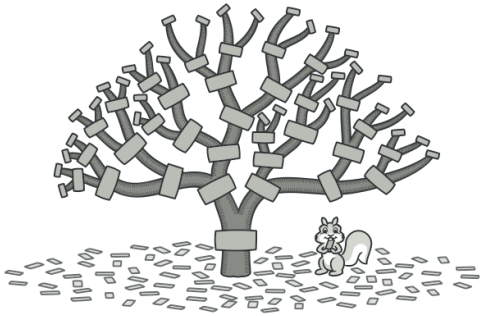
- Component interface
  - operations that are common to both simple and complex elements of the tree
- Leaf
  - basic element of a tree that doesn't have subelements
  - do most of the real work, since they don't have anyone to delegate the work to
- Container (aka composite)
  - has sub-elements: leaves or containers
  - works with all sub-elements only via the component interface - doesn't know the concrete classes of its children
  - delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client
- Client
  - works with all elements through the component interface
  - works in the same way with both simple or complex elements of the tree



UML - Composite



Composite Implementation



```
class Composite : AComponent
{
    private List<AComponent> childComponents = new List<AComponent>();
    public override void Operation()
    {
        Console.WriteLine("Hello this is a compontent and my childs are: ");
        foreach (AComponent item in childComponents)
        {
            item.Operation();
        }
    }
    public  void Add(AComponent comp)
    {
        childComponents.Add(comp);
    }
    public  void Remove(AComponent comp)
    {
        childComponents.Remove(comp);
    }
    public  AComponent GetChild(int index)
    {
        return childComponents[index];
    }

}
```

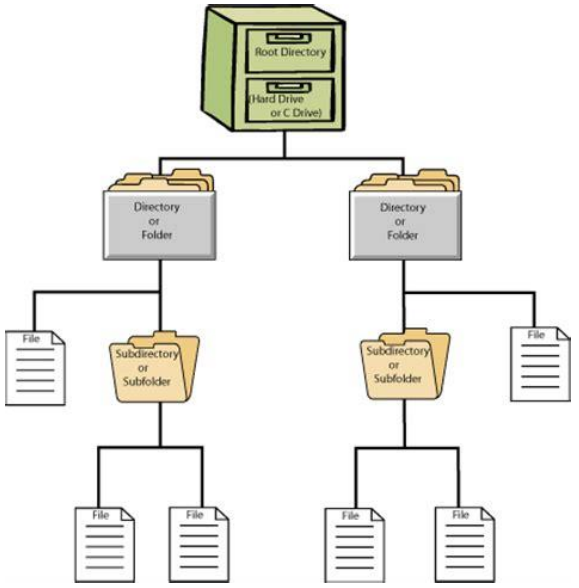
```
abstract class AComponent
{
    public abstract void Operation();
}
```

```
class Leaf : AComponent
{
    public override void Operation()
    {
        Console.WriteLine("Hello there, I am a leaf");
    }
}
```

Excercise Filesystem

```
class Program
{
    static void Main(string[] args)
    {
        Directory root = new Directory("Root");
        root.AddToDirectory(new File("Präsen.pot"));
        root.AddToDirectory(new File("IDK.txt"));
        root.AddToDirectory(new File("Zusammenfsg.docx"));

        root.Operation();
    }
}
```

Implement a program which describes a file-system, in which directories are the composites and files are the leafs. Via the methods operation, each object puts out who they are and what their children are (if they got one or more)



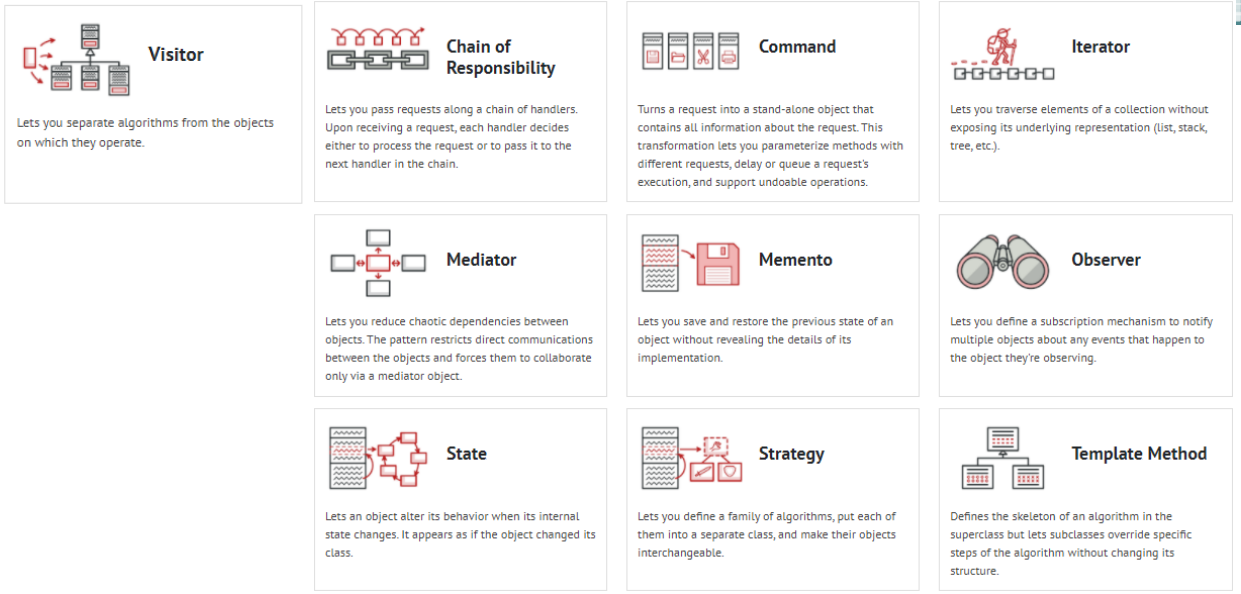
Directory & File

```
abstract class AComponent {
    public abstract void Operation();
}
class File : AComponent
{
    private string name;
    public File(string n){
        this.name = n;
    }
    public override void Operation(){
        Console.WriteLine("This is: " + name);
    }
}
```

```
class Directory : AComponent
{
    private string name;
    private List<AComponent> directoryContent;
    public Directory(string n) {
        this.name = n;
        directoryContent = new List<AComponent>();
    }
    public void AddToDirectory(AComponent c)  {
        directoryContent.Add(c);
    }
    public void RemoveFromDirectory(AComponent c) {
        directoryContent.Remove(c);
    }
    public override void Operation(){
        Console.WriteLine("This is foler: " + name+" and my files are: ");
        foreach(AComponent c in directoryContent)
        {
            c.Operation();
        }
    }
}
```

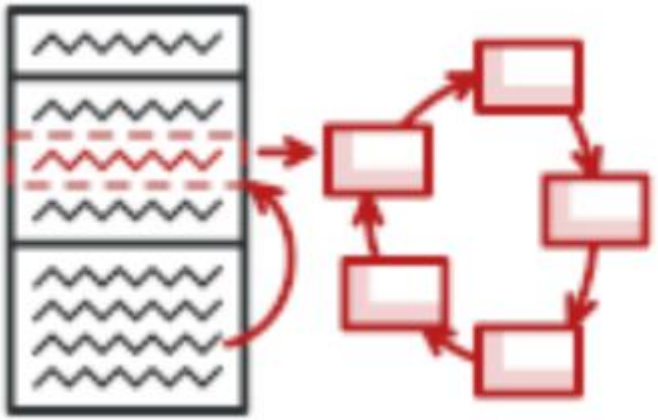
Relations with Other Patterns

- You can use **Builder** when creating complex Composite trees because you can program its construction steps to work recursively.
- **Chain of Responsibility** is often used in conjunction with Composite. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.
- You can use **Iterators** to traverse Composite trees.
- You can use **Visitor** to execute an operation over an entire Composite tree.
- Composite and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.



Behavioral Design Patterns

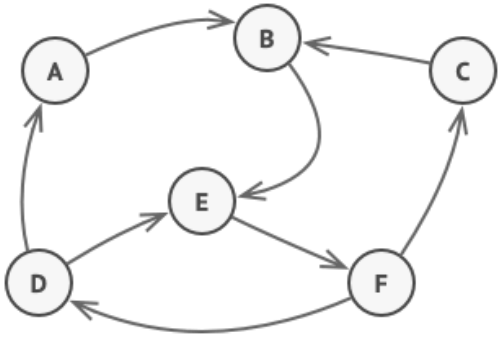
are concerned with algorithms and the assignment of responsibilities between objects.



State Pattern

lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Problem



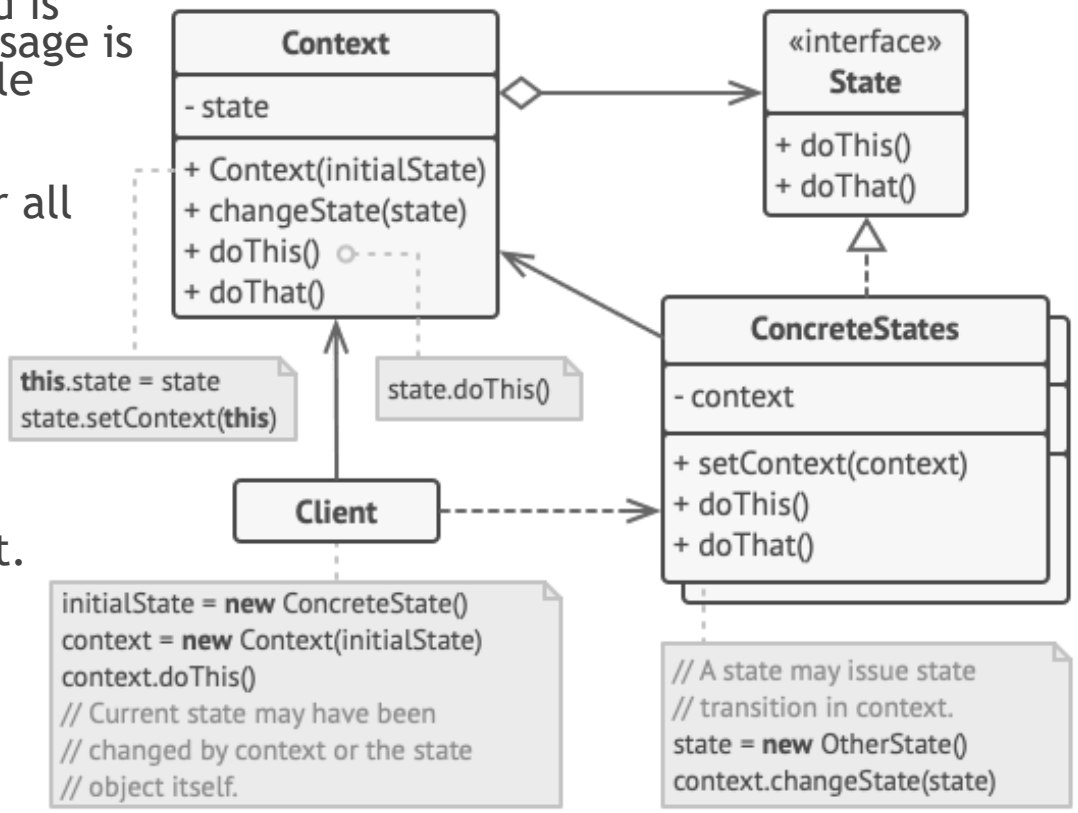
- is closely related to the concept of a **Finite-State Machine**
- any given moment, there's a finite number of states which a program can be in
- within any unique state the program behaves differently and the program can be switched from one state to another instantaneously
- depending on a current state, the program may or may not switch to certain other states
- these switching rules, called transitions, are also finite and predetermined

Solution

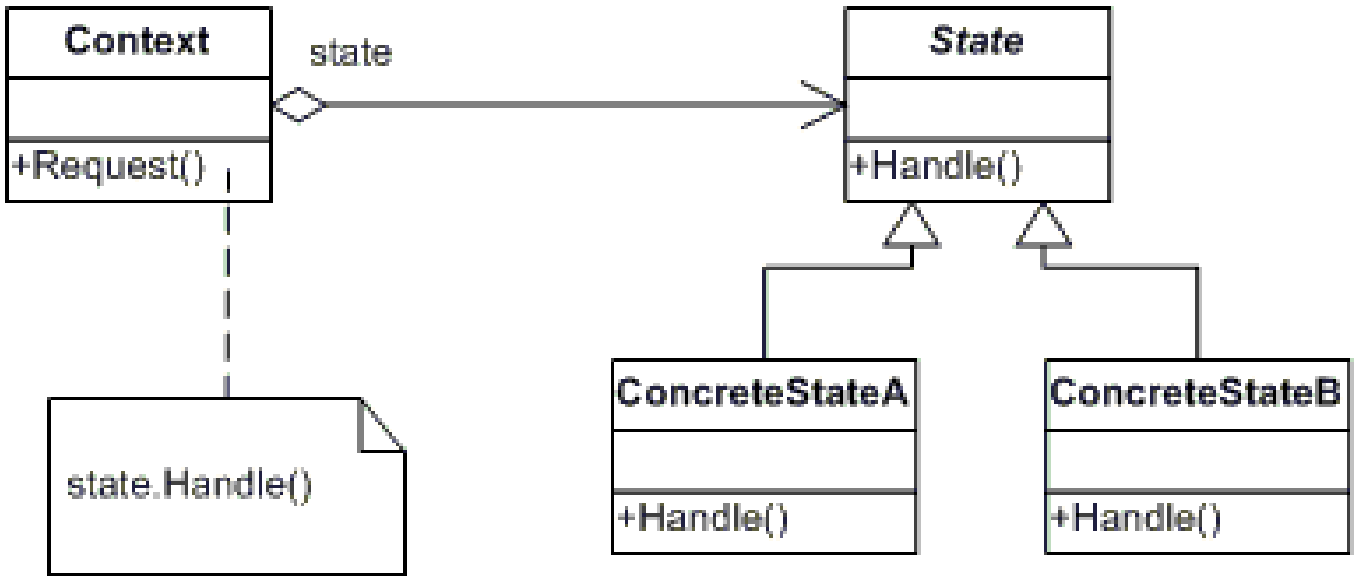
- create new classes for all possible states of an object and extract all state-specific behaviors into these classes
- Instead of implementing all behaviors on its own, the original object, called context, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object

State Pattern - Explanation

- **Context**
  - can have a number of internal states, whenever the request() method is called on the Context, the message is delegated to the State to handle
- **State** interface
  - defines a common interface for all concrete states, encapsulating all behavior associated with a particular state
- **ConcreteState**
  - implements it's own implementation for the request. When a Context changes state, what really happens is that we have a different ConcreteState associated with it



UML - State

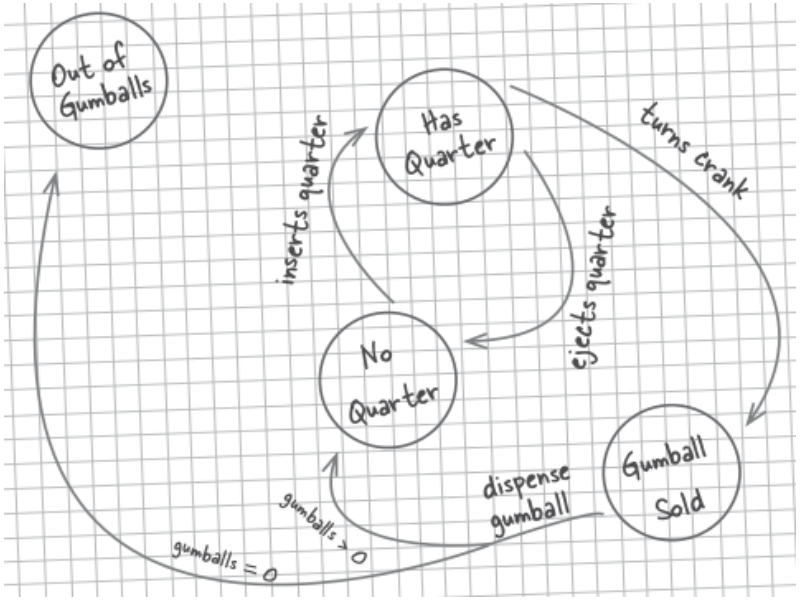

Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Excercise - Gumball

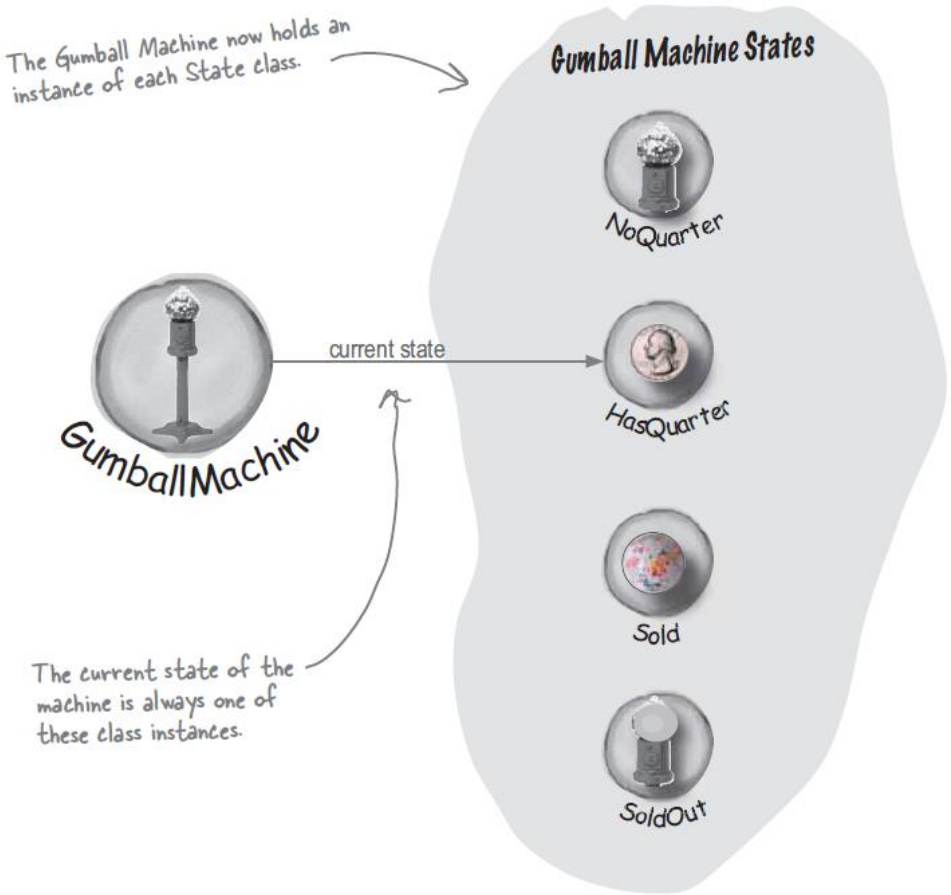
- Imagine you're working in B2B software company and Mighty Gumball, Inc. is asking you for help.

  They present you this diagram:

- And ask you to implement it in C#.



Gumball Machine States

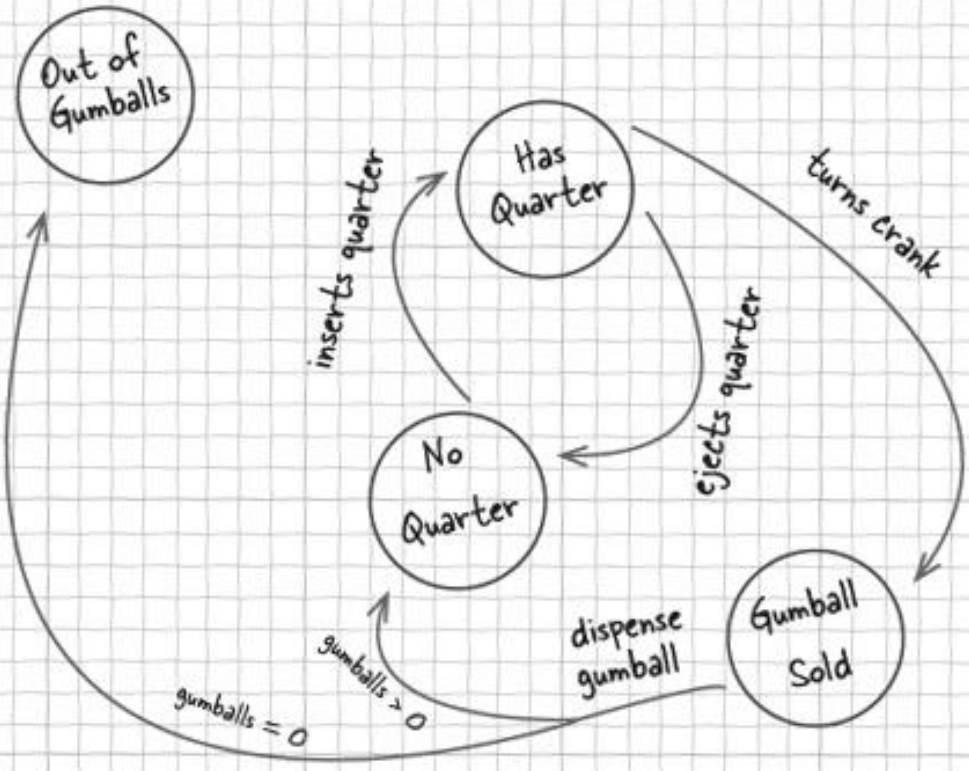


Mighty Gumball, Inc.

Where the Gumball Machine Is Never Half Empty

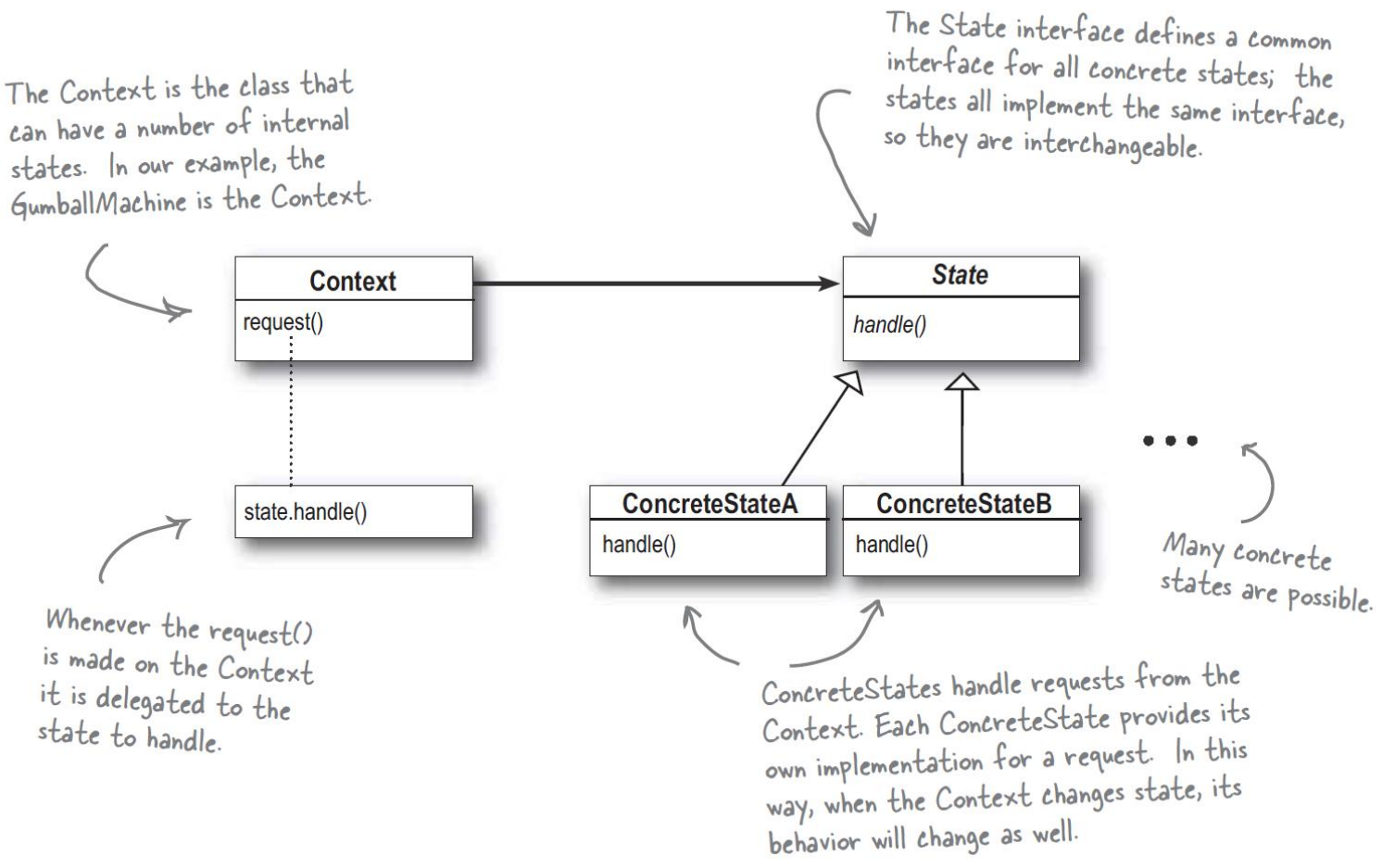
Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers

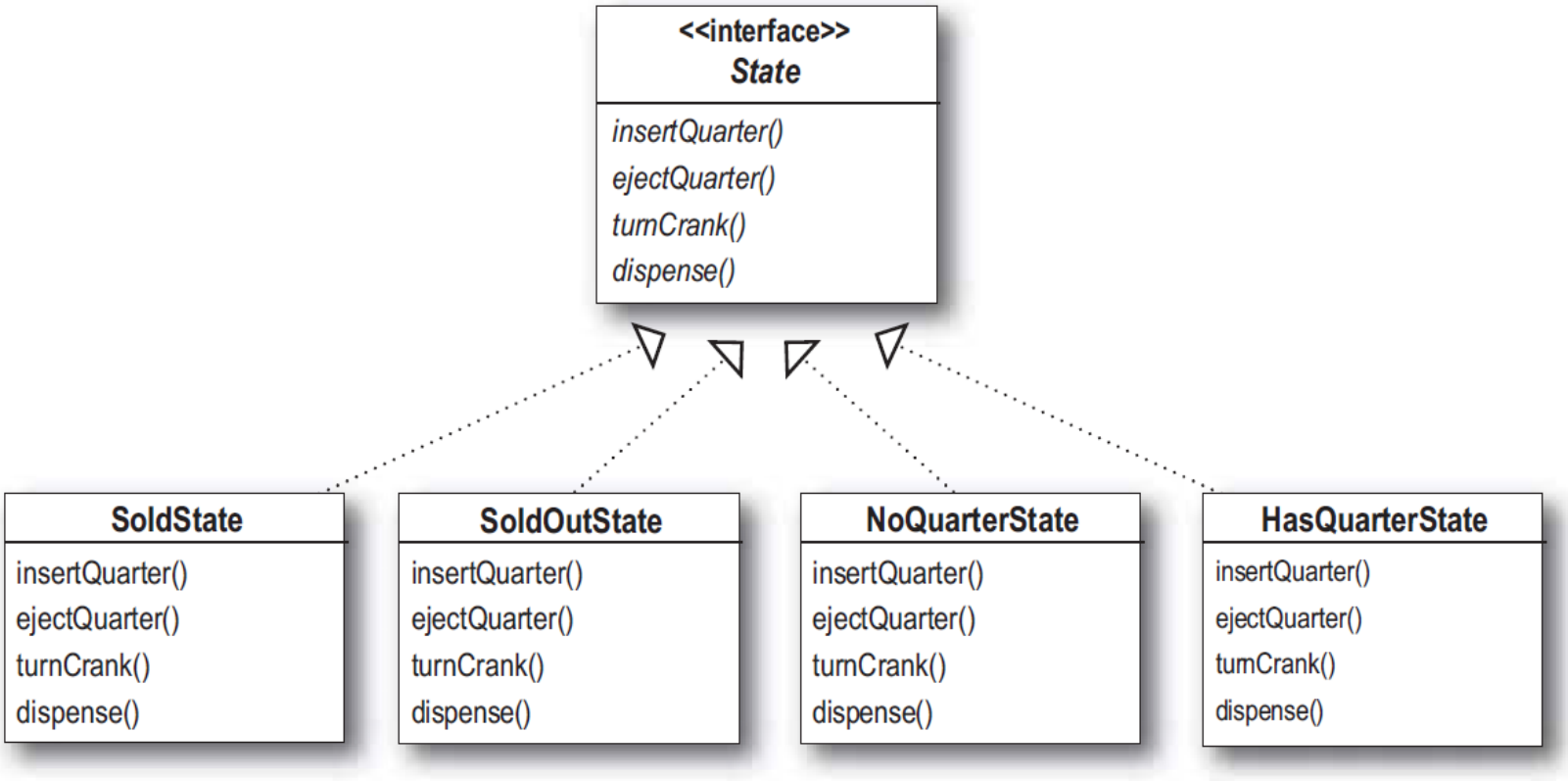


Uml – State Pattern

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



IState with Concrete States



IState & SoldStates

```
public interface IState
{
    void InsertQuarter();
    void EjectQuarter();
    void TurnCrank();
    void Dispense();
}
```

```
public class SoldState : IState
{
    GumBallMachine gumBallMachine;
    public SoldState(GumBallMachine gumBallMachine){
        this.gumBallMachine = gumBallMachine;
    }
    public void InsertQuarter(){
        Console.WriteLine("Please wait, we´re already giving you a gumball");
    }
    public void EjectQuarter(){
        Console.WriteLine("Sorry, you already turned the crank");
    }
    public void TurnCrank(){
        Console.WriteLine("Turning twice doesn´t get you another gumball!");
    }
    public void Dispense(){
        gumBallMachine.releaseBall();
        if (gumBallMachine.getCount() > 0)
            gumBallMachine.setState(gumBallMachine.getNoQuarterState());
        else
        {
            Console.WriteLine("Oops, out of gumballs!");
            gumBallMachine.setState(gumBallMachine.getSoldOutState());
        }
    }
}
```

SoldOutState

```
public interface IState
{
    void InsertQuarter();
    void EjectQuarter();
    void TurnCrank();
    void Dispense();
}
```

```
public class SoldOutState : IState {
    GumBallMachine gumBallMachine;
    public SoldOutState(GumBallMachine gumBallMachine){
        this.gumBallMachine = gumBallMachine;
    }
    public void InsertQuarter(){
        Console.WriteLine("You can´t insert another quarter");
    }
    public void EjectQuarter(){
        Console.WriteLine("Quarter returned");
        gumBallMachine.setState(gumBallMachine.getNoQuarterState());
    }
    public void TurnCrank(){
        Console.WriteLine("Sorry, we´re sold out");
    }
    public void Dispense(){
        Console.WriteLine("Sorry, we´re sold out");//425
    }
}
```

NoQuaterState

```
public interface IState
{
    void InsertQuarter();
    void EjectQuarter();
    void TurnCrank();
    void Dispense();
}
```

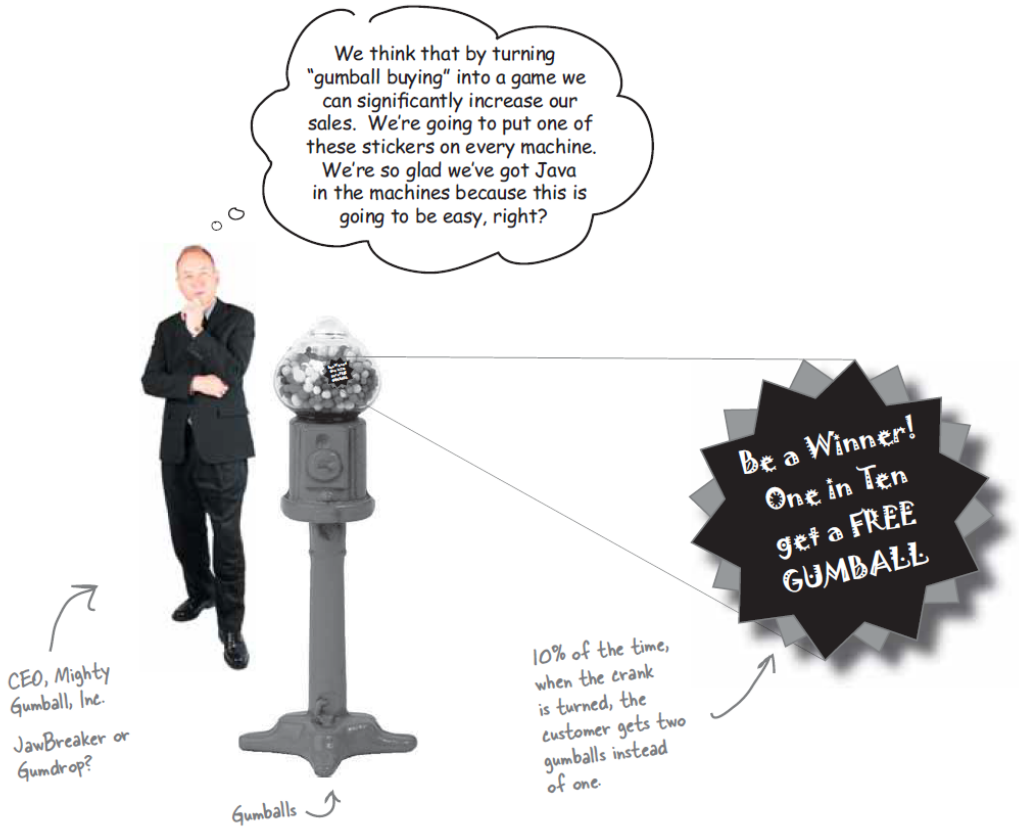
```
public class NoQuarterState : IState {
    GumBallMachine gumBallMachine;
    public NoQuarterState(GumBallMachine gumBallMachine){
        this.gumBallMachine = gumBallMachine;
    }
    public void InsertQuarter(){
        Console.WriteLine("You inserted a quarter");
        gumBallMachine.setState(gumBallMachine.getHasQuarterState());
    }
    public void EjectQuarter(){
        Console.WriteLine("You haven´t inserted a quarter");
    }
    public void TurnCrank(){
        Console.WriteLine("You turned, but there is no quarter");
    }
    public void Dispense(){
        Console.WriteLine("You need to pay first");
    }
}
```

HasQuaterState

```
public class HasQuarterState : IState{
    Random randomWinner = new Random((int)DateTimeOffset.UtcNow.ToUnixTimeMilliseconds());
    GumBallMachine gumBallMachine;
    public HasQuarterState(GumBallMachine g){
        gumBallMachine = g;
    }
    public void InsertQuarter(){
        Console.WriteLine("You can´t insert another quarter");
    }
    public void EjectQuarter(){
        Console.WriteLine("Quarter returned");
        gumBallMachine.setState(gumBallMachine.getNoQuarterState());
    }
    public void TurnCrank(){
        Console.WriteLine("You turned...");
        int winner = randomWinner.Next(10);
        if (winner == 0 && gumBallMachine.getCount() > 1)
            gumBallMachine.setState(gumBallMachine.getWinnerState());
        else
            gumBallMachine.setState(gumBallMachine.getSoldState());
    }
    public void Dispense(){
        Console.WriteLine("No gumball dispensed");
    }
}
```

```
public interface IState
{
    void InsertQuarter();
    void EjectQuarter();
    void TurnCrank();
    void Dispense();
}
```

Add a Winner State



Winner State

```
public interface IState
{
    void InsertQuarter();
    void EjectQuarter();
    void TurnCrank();
    void Dispense();
}
```

```
public class WinnerState : IState
{
    GumBallMachine gumBallMachine;
    public WinnerState(GumBallMachine g){
        gumBallMachine = g;
    }
    public void InsertQuarter(){
        Console.WriteLine("Please wait, we´re already giving you a gumball");
    }
    public void EjectQuarter(){
        Console.WriteLine("Sorry, you already turned the crank");
    }
    public void TurnCrank(){
        Console.WriteLine("Turning twice doesn´t get you another gumball");
    }
    public void Dispense(){
        Console.WriteLine("You're a Winner! You get two gumballs for your quarter"); //S. 413
        gumBallMachine.releaseBall();
        if (gumBallMachine.getCount() == 0) {
            gumBallMachine.setState(gumBallMachine.getSoldOutState());
        }else {
            gumBallMachine.releaseBall();
            if (gumBallMachine.getCount() > 0) {
                gumBallMachine.setState(gumBallMachine.getNoQuarterState());
            }else {
                Console.WriteLine("Oops, out of gumballs!");
                gumBallMachine.setState(gumBallMachine.getSoldOutState());
            }
        }
    }
}
```

GumBall Machine

```
public class GumBallMachine
{
    IState soldOutState;
    IState noQuarterState;
    IState hasQuarterState;
    IState soldState;
    IState winnerState;
    IState state;

    int count = 0;
    public GumBallMachine(int numberGumballs){
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        winnerState = new WinnerState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0){
            state = noQuarterState;
        }
    }
    public int GetCount()...
    public IState GetSoldOutState() ...
    public IState GetSoldState() ...
    public IState GetHasQuarterState() ...
    public IState GetWinnerState() ...
    public IState GetNoQuarterState() ...
    public void InsertQuarter(){
        state.InsertQuarter();
    }
    public void EjectQuarter()...
    public void TurnCrank()...
    public void SetState(IState state)...
    public void ReleaseBall(){
        Console.WriteLine("A gumball comes rolling out the slot...");
        if (count != 0)
            count -= 1;
    }
}
```

- Erstelle statt den Get Methoden Properties mit nur Leserechten

```
static void Main(string[] args) {
    GumBallMachine gumballMachine = new GumBallMachine(5);
    Console.WriteLine(gumballMachine);
    gumballMachine.InsertQuarter();
    gumballMachine.TurnCrank();
    Console.WriteLine(gumballMachine);
    gumballMachine.InsertQuarter();
    gumballMachine.TurnCrank();
    gumballMachine.InsertQuarter();
    gumballMachine.TurnCrank();
    Console.WriteLine(gumballMachine);
}
```

Main

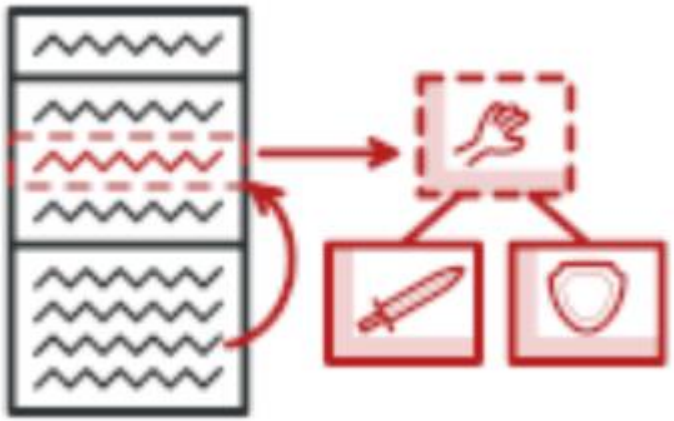
This code really hasn't changed at all; we just shortened it a bit.

Once, again, start with a gumball machine with 5 gumballs.

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

```
File  Edit  Window  Help  Whenisagumballajawbreaker?

%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```



Strategy Pattern

lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable

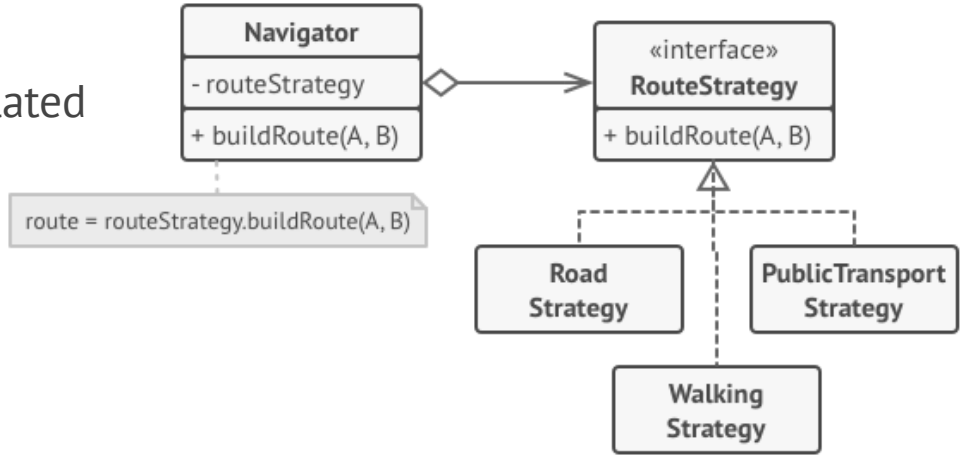
Problem

A story which describes the problem:

- One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.
- One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.
- The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.
- However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city's tourist attractions.
- While from a business perspective the app was a success, the technical part caused you many headaches. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.
- Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.
- In addition, teamwork became inefficient. Your teammates, who had been hired right after the successful release, complain that they spend too much time resolving merge conflicts. Implementing a new feature requires you to change the same huge class, conflicting with the code produced by other people.

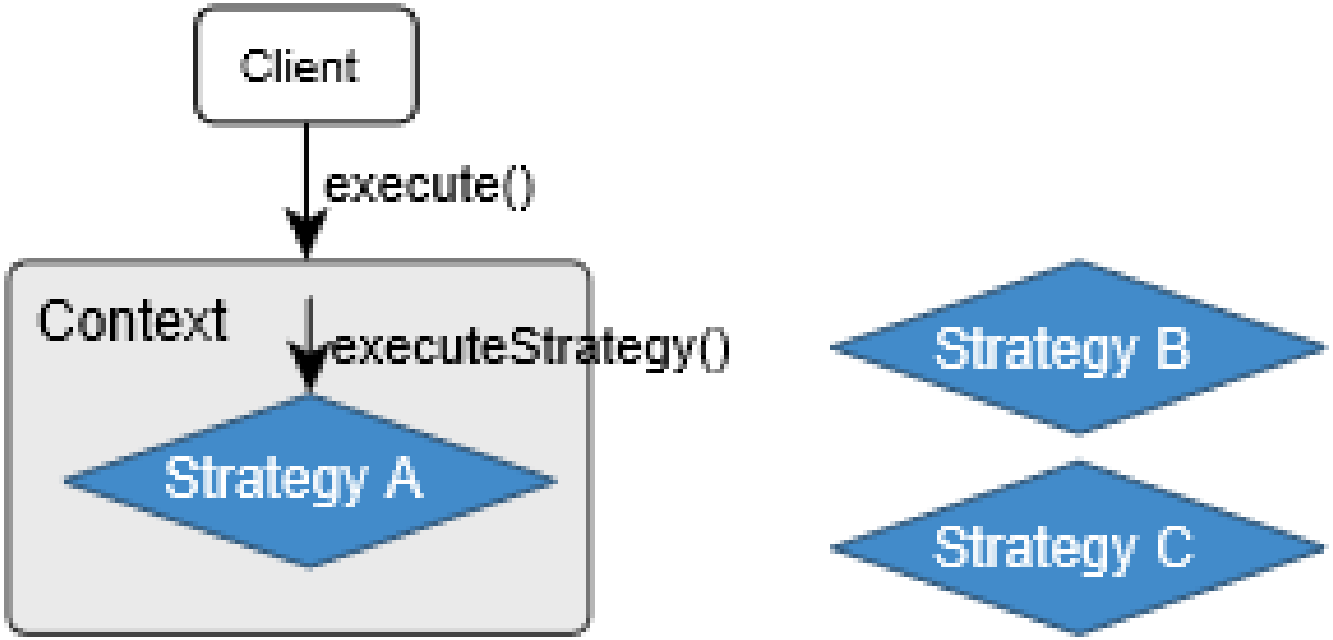
Solution

- Strategies
  - take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes
- C*ontext*
  - must have a field for storing a reference to one of the strategies
  - delegates the work to a linked strategy object instead of executing it on its own
  - context isn't responsible for selecting an appropriate algorithm for the job, it passes the desired strategy to the context
  - context doesn't know much about strategies
  - context works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy
  - context becomes independent of concrete strategies



Strategy Pattern

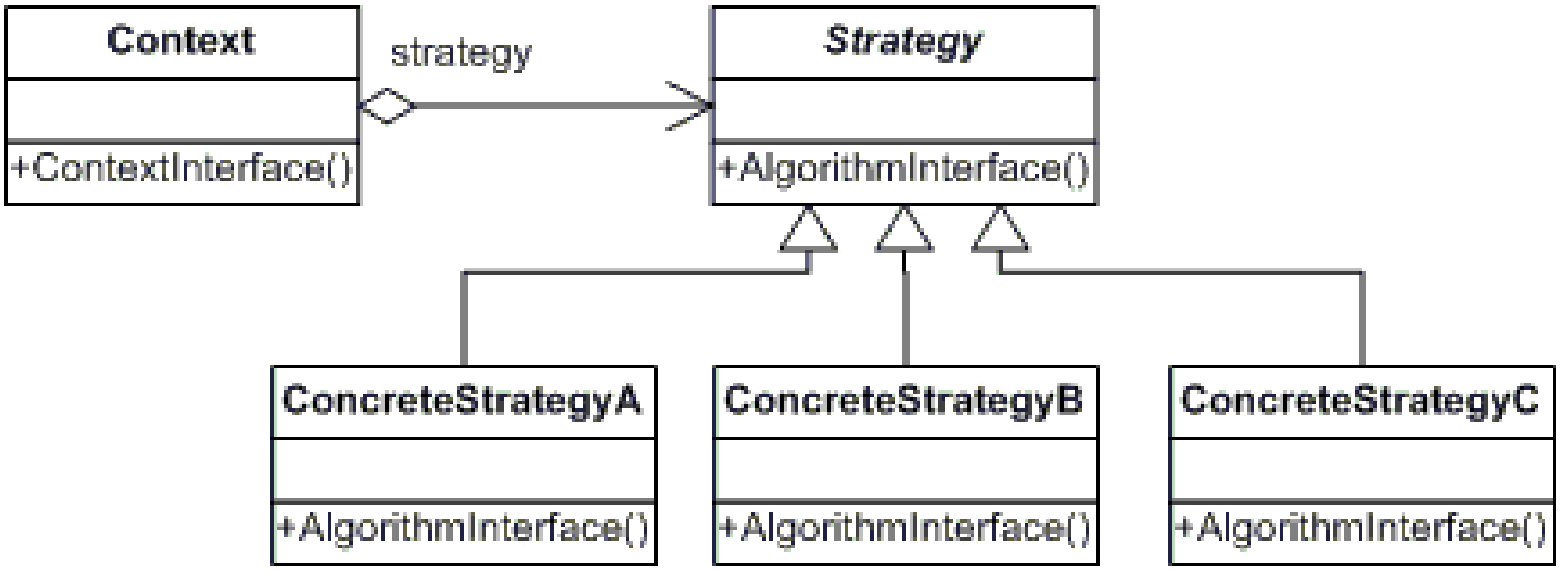
- Behavior is independable of the context
- context is independable of the imlementation



Class Diagramm

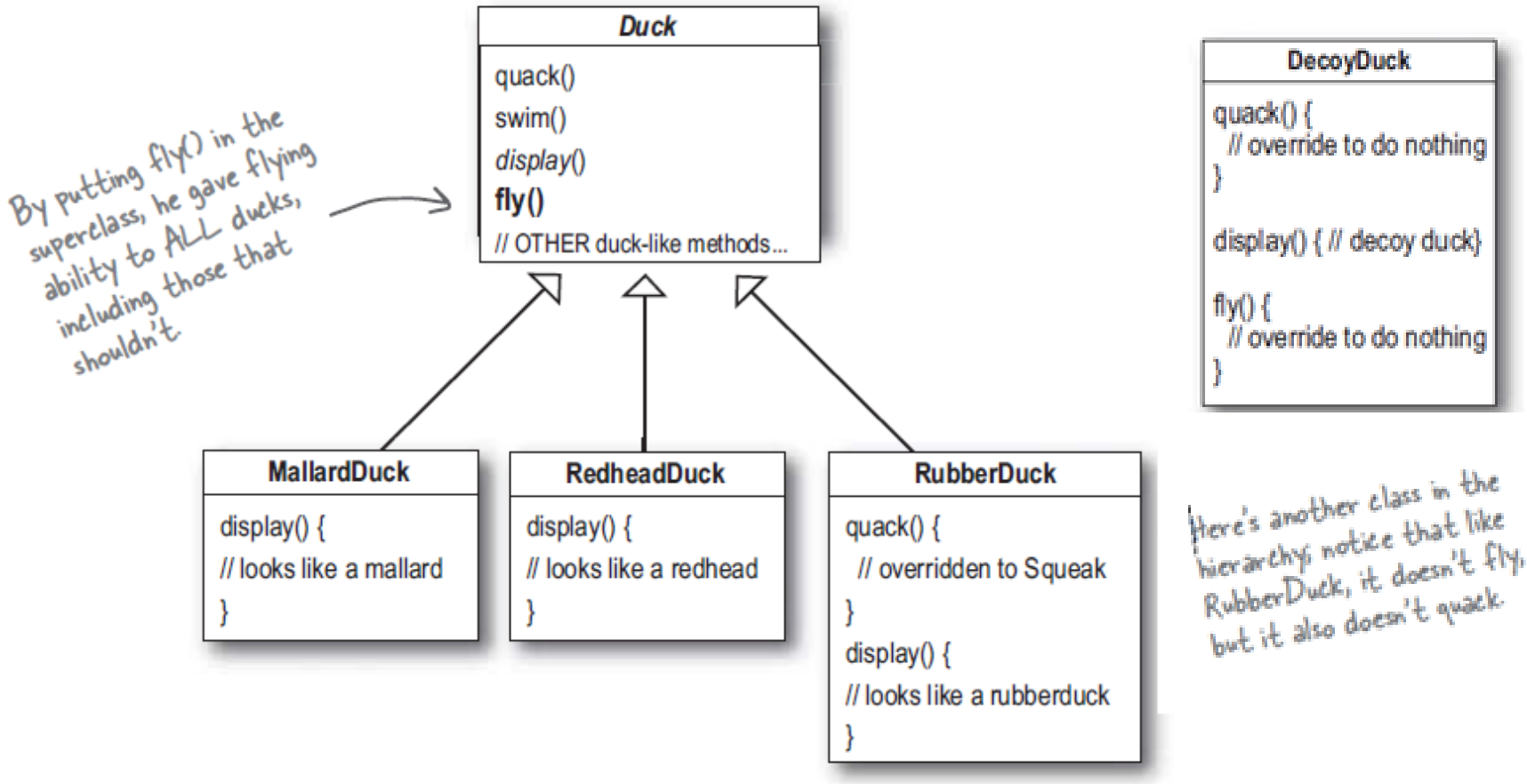
- IBehaviour (Strategy)
  - an interface that defines the behavior
- Conctete Strategies:
  - each of them defines a specific behavior.
- Context class
  - It keeps or gets context information and passes necessary information to the Strategy class

UML - Strategy

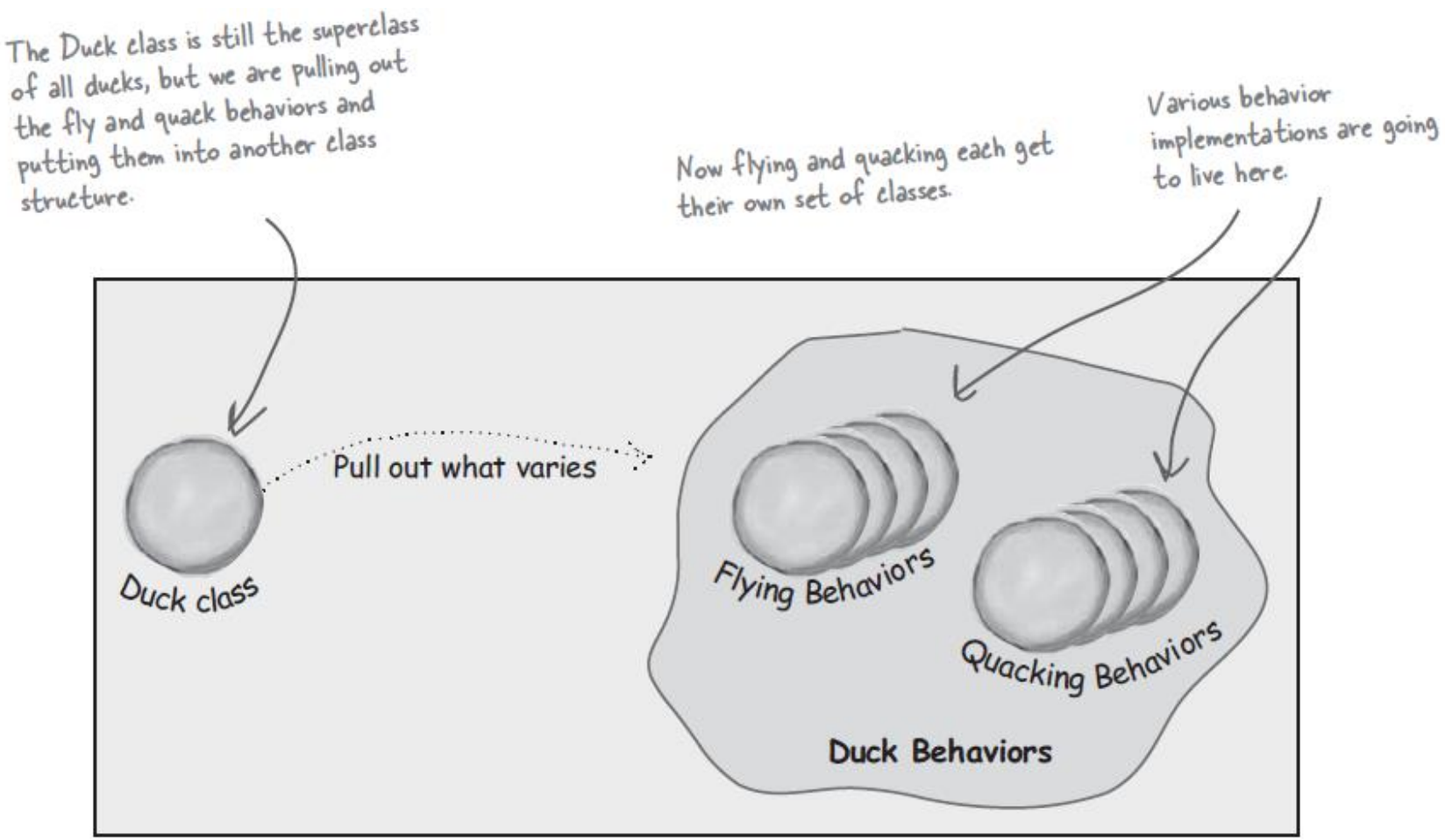


Strategy with Ducks

- Ducks can Swim, most of them can Quack, some of them can Fly.

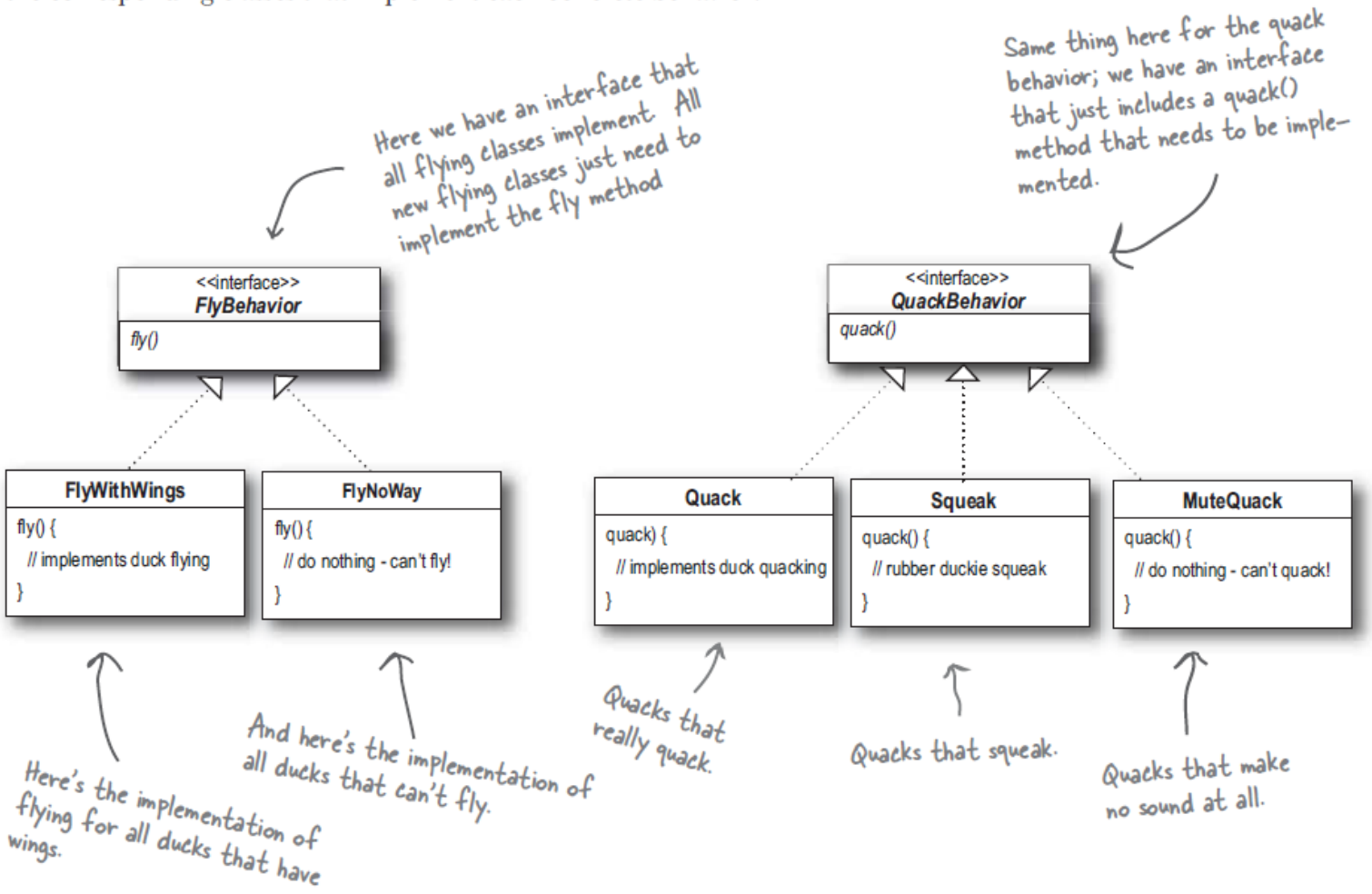


Strategy Solution

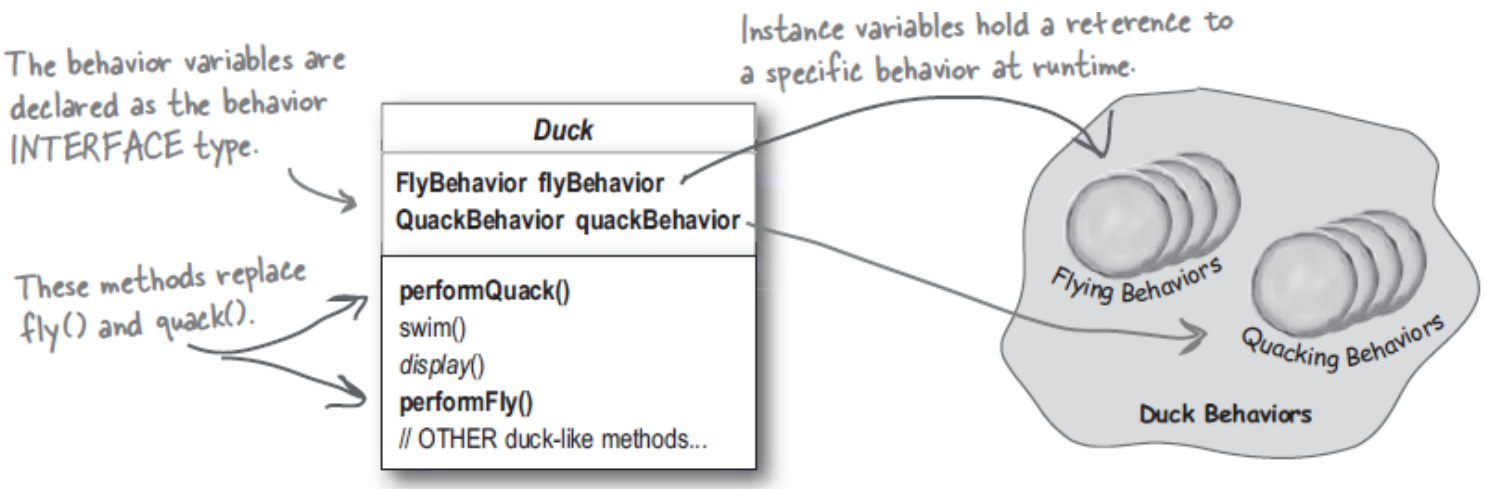


IFlyBehavior & IQuackBehavior

Here we have the two interfaces, FlyBehavior and QuackBehavior along with the corresponding classes that implement each concrete behavior:



Implementing Quack-Method



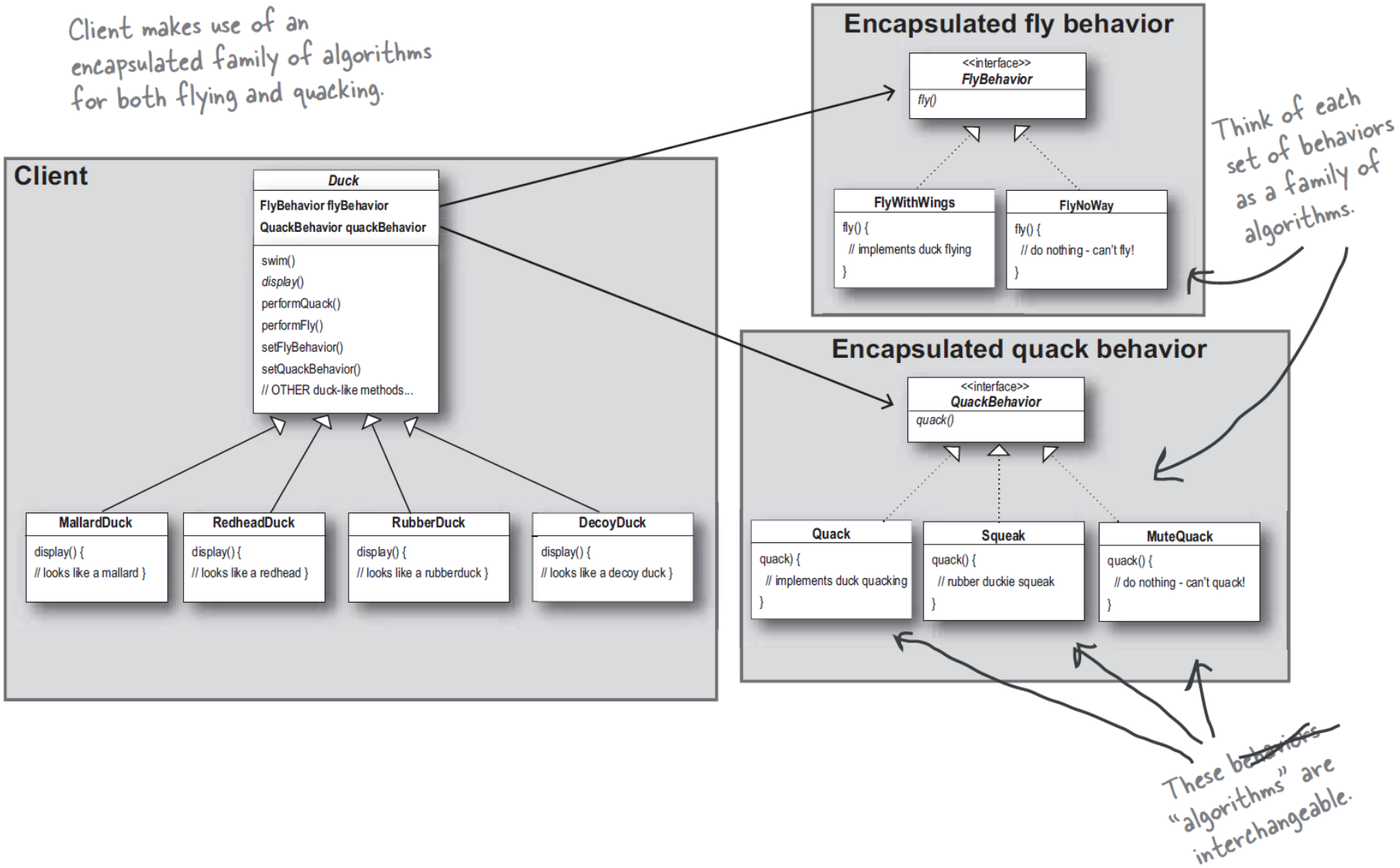
Now we implement performQuack():

```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

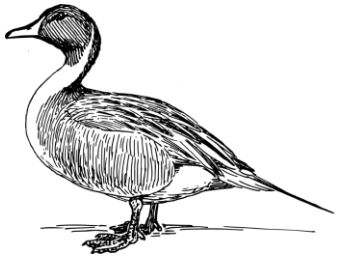
Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Strategy Pattern - Ducks

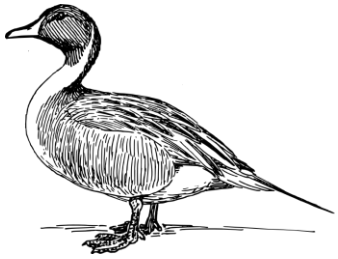


Abstract Duck



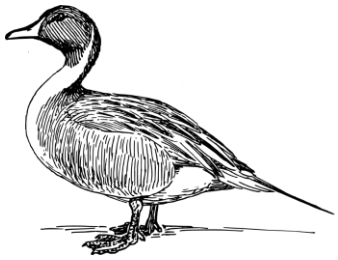
```
public abstract class ADuck {
    private readonly IFlyBehavior flyBehavior;
    private readonly IQuackBehavior quackBehavior;
    protected ADuck(IFlyBehavior flyBehavior, IQuackBehavior quackBehavior) {
        this.flyBehavior = flyBehavior;
        this.quackBehavior = quackBehavior;
    }

    public abstract void Display();

    public void PerformFly() {
        this.flyBehavior.Flying();
    }

    public void PerformQuack() {
        this.quackBehavior.Quacking();
    }

    public void Swim() {
        Console.WriteLine("All ducks float, even decoys!");
    }
}
```

Quack Behavior



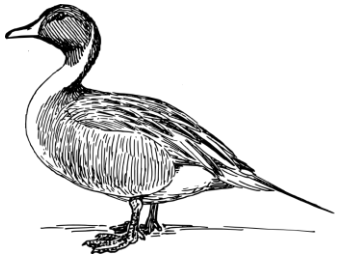
```
public interface IQuackBehavior {
    void Quacking();
}

public class Quack : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("Quack");
    }
}

public class MuteQuack : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("<<  Silence  >>");
    }
}

public class Squeak : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("Squeak");
    }
}
```

Fly Behavior



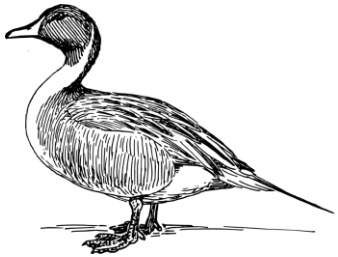
```
public interface IFlyBehavior {
    void Flying();
}

public class FlyWithWings : IFlyBehavior {
    public void Flying() {
        Console.WriteLine("I'm flying!");
    }
}

public class FlyNoWay : IFlyBehavior {
    public void Flying() {
        Console.WriteLine("I can't fly!");
    }
}
```

Mallard Duck & Decoy Duck



```
public class MallardDuck : ADuck {
    public MallardDuck() : base(new FlyWithWings(), new Quack()) {
    }

    public override void Display() {
        Console.WriteLine("I'm a real Mallard duck");
    }
}


public class DecoyDuck : ADuck {
    public DecoyDuck() : base(new FlyNoWay(), new MuteQuack()) {
    }

    public override void Display() {
        Console.WriteLine("I'm a real decoy duck");
    }
}
```

Using Ducks



```
public class MiniDuckSimulator {
    public static void Main(string[] args) {
        ADuck mallard = new MallardDuck();
        mallard.PerformQuack();
        mallard.PerformFly();
        mallard.Swim();
        Console.ReadLine();

        ADuck decoy = new DecoyDuck();
        decoy.PerformQuack();
        decoy.PerformFly();
        decoy.Swim();
        Console.ReadLine();
    }
}
```

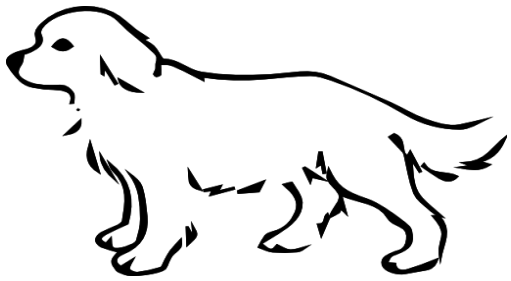
```
Quack
I'm flying!
All ducks float, even decoys!

<<  Silence  >>
I can't fly!
All ducks float, even decoys!
```

Strategy with Dogs

- Create an abstract class Dog, which can bark and walk. Each breed barks and walks differently:
  - Husky runs fast and barks loud
  - Bulldog limps and barks loud
  - DummyDog can't walk and has electronic barking
  - Poodle walks normal and barks quietly
  - …

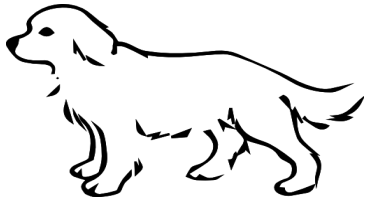
Main – Testing Dogs

```
static void Main(string[] args)
{
    ADog husky = new Husky();
    Console.WriteLine("husky");
    husky.Bark();
    husky.Walk();

    Console.WriteLine();

    ADog bulldog = new Bulldog();
    Console.WriteLine("bulldog");
    bulldog.Bark();
    bulldog.Walk();

    Console.WriteLine();

    ADog poodle = new Poodle();
    Console.WriteLine("poodle");
    poodle.Bark();
    poodle.Walk();

    Console.WriteLine();

    ADog dummy = new DummyDog();
    Console.WriteLine("dummy");
    dummy.Bark();
    dummy.Walk();
}
```



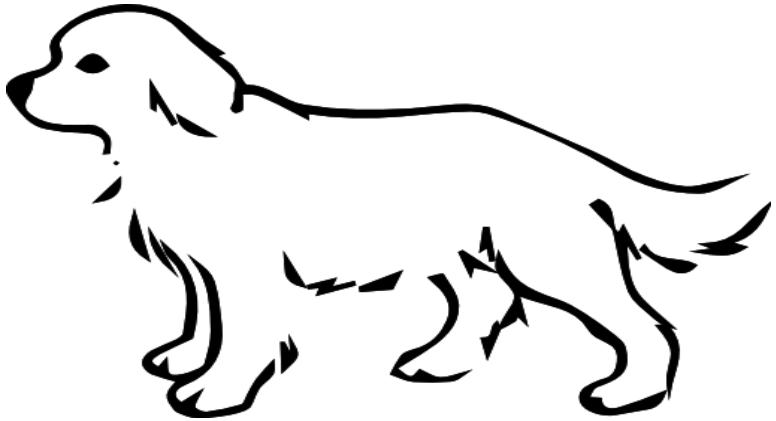
```
husky
loud woof
fast walking

bulldog
loud woof
limping walking

poodle
quiet woof
normal walking

dummy
electronic woof
no walking
```

Strategy Pattern with Dogs



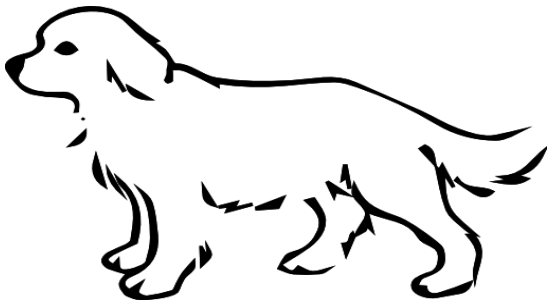
Class Dog

```
public abstract class ADog
{
    private IWalkingBehaviour walkingBehaviour;
    private IBarkingBehaviour barkingBehaviour;

    public ADog(IWalkingBehaviour w, IBarkingBehaviour b)
    {
        this.walkingBehaviour = w;
        this.barkingBehaviour = b;
    }

    public void Bark()
    {
        this.barkingBehaviour.Bark();
    }

    public void Walk()
    {
        this.walkingBehaviour.Walk();
    }
}
```



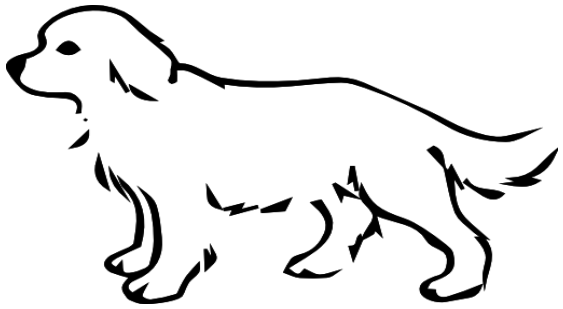
Bark Behaviour

```
public interface IBarkingBehaviour {
    public void Bark();
}
public class LoudBarking : IBarkingBehaviour {
    public void Bark() {
        Console.WriteLine("loud woof");
    }
}
public class QuietBarking : IBarkingBehaviour {
    public void Bark() {
        Console.WriteLine("quiet woof");
    }
}
public class ElectronicBarking : IBarkingBehaviour {
    public void Bark() {
        Console.WriteLine("electronic woof");
    }
}
```

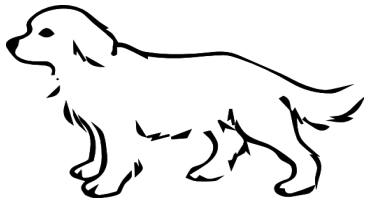


Walk Behaviour

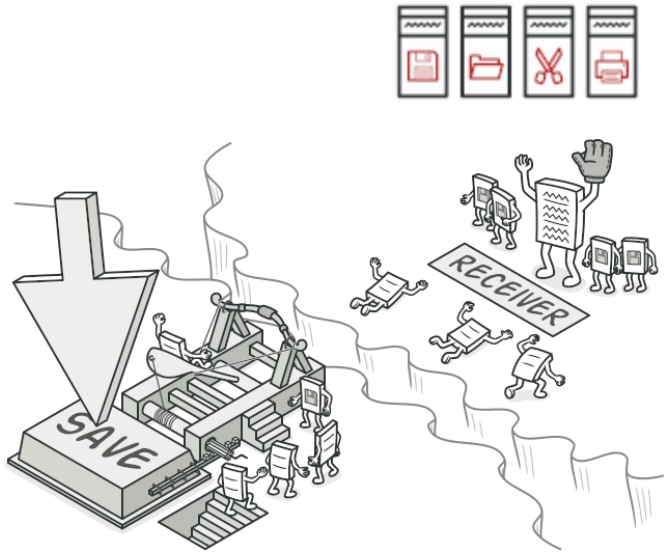
```
public interface IWalkingBehaviour {
    public void Walk();
}
public class FastWalking : IWalkingBehaviour {
    public void Walk() {
        Console.WriteLine("fast walking");
    }
}
public class NormalWalking : IWalkingBehaviour {
    public void Walk() {
        Console.WriteLine("normal walking");
    }
}
public class LimpingWalking : IWalkingBehaviour {
    public void Walk() {
        Console.WriteLine("limping walking");
    }
}
public class CannotWalk : IWalkingBehaviour {
    public void Walk() {
        Console.WriteLine("no walking");
    }
}
```



Concrete Dogs



```
public class Poodle : ADog
{
    public Poodle()
        :base(new NormalWalking(), new QuietBarking()) { }
}


public class DummyDog : ADog
{
    public DummyDog() :
        base(new CannotWalk(), new ElectronicBarking()) { }
}

public class Husky : ADog
{
    public Husky() :
        base(new FastWalking(), new LoudBarking()) { }
}
```

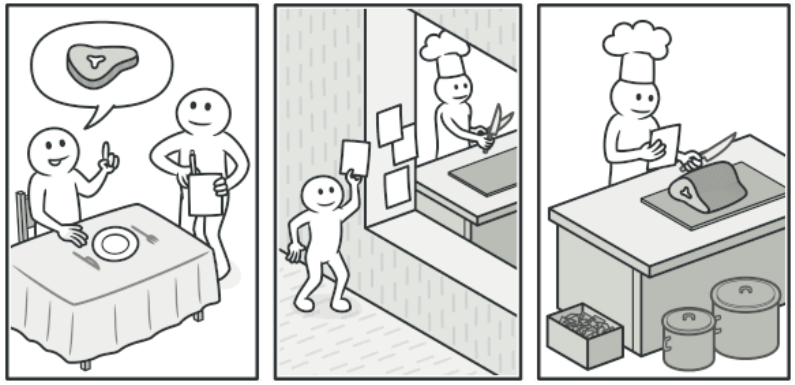


Command Pattern

turns a request into a stand-alone object that contains all information about the request.

transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

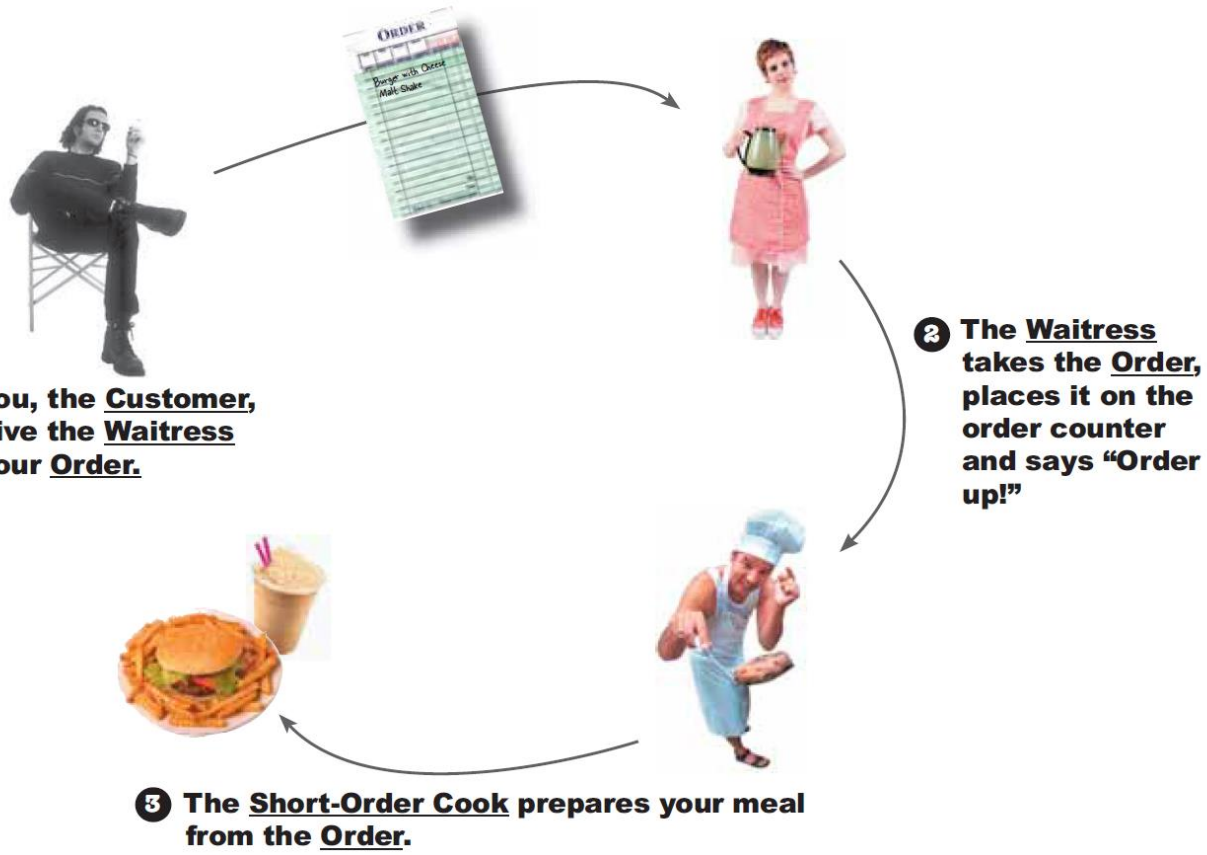
Real-World Analogy



- A friendly waiter approaches you and takes your order, writing it down on a piece of paper.
- The waiter goes to the kitchen and sticks the order on the wall.
- The order gets to the chef, who reads it and cooks the meal accordingly.
  The cook places the meal on a tray along with the order.
  The waiter discovers the tray, checks the order to make sure everything is as you wanted it, and brings everything to your table.
- The paper order serves as a command.
  It remains in a queue until the chef is ready to serve it.
  The order contains all the relevant information required to cook the meal. It allows the chef to start cooking right away instead of running around clarifying the order details from you directly.

Sequence

Okay, we all know how the Diner operates:

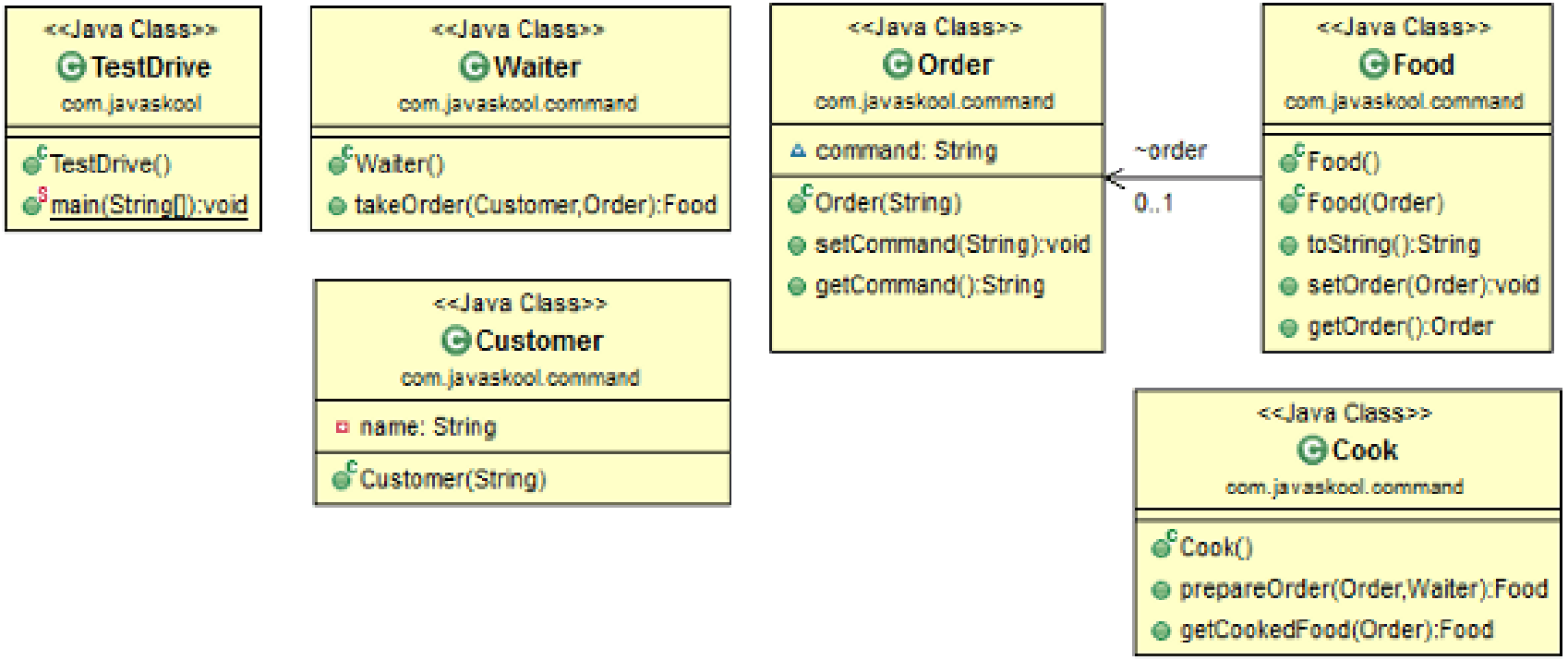


1 You, the **Customer**, give the **Waitress** your **Order.**

2 The **Waitress** takes the **Order**, places it on the order counter and says "Order up!"

3 The **Short-Order Cook** prepares your meal from the **Order**.

Class Diagram



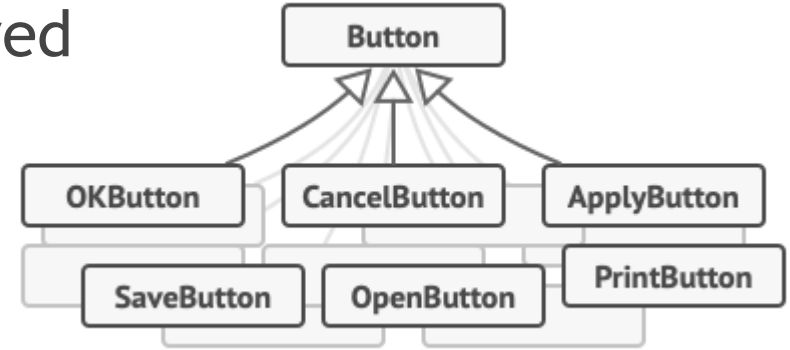
Purpose & Use

- Encapsulates a request allowing it to be treated as an object.

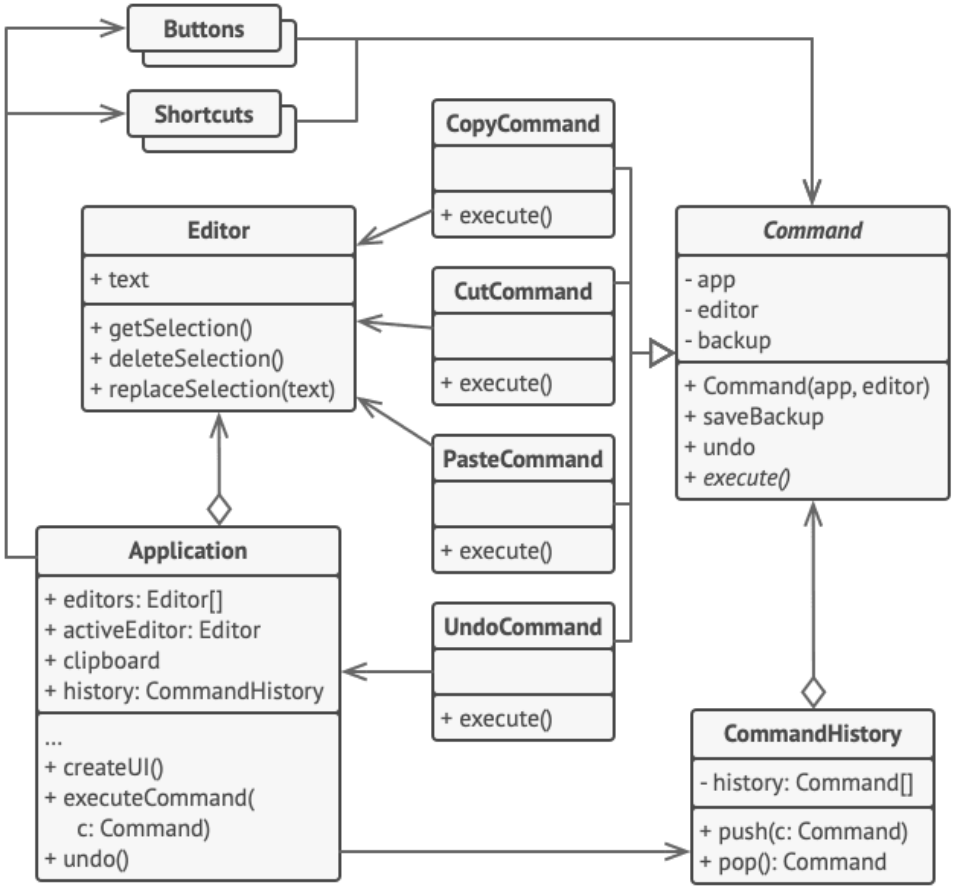
- **Use when**
  - You need to support logging.
  - A history management is needed (that is, undo/redo operations).
  - Source of the request should be decoupled from the object that actually handles the request.
  - …

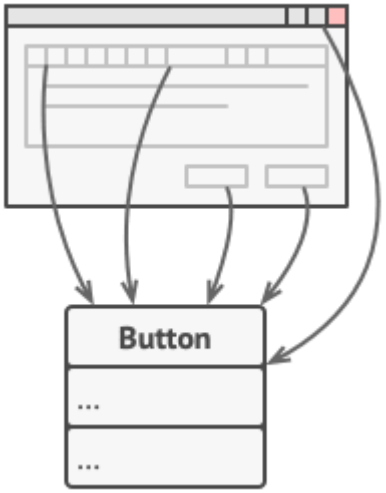
Problem

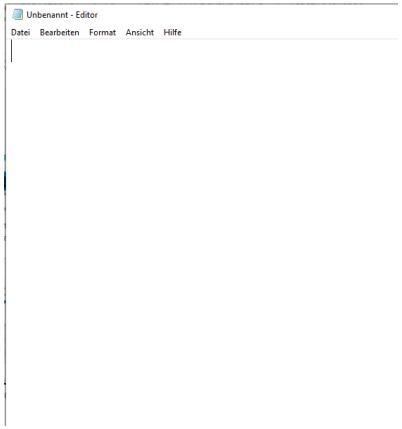
- Imagine that you're working on a new text-editor app.
- Current task is to create a toolbar with a bunch of buttons for various operations of the editor.
- All buttons of the app are derived from the same class.
- While all of these buttons look similar, they're all supposed to do different things.
- Several classes implement the same functionality.



Excercise Editor

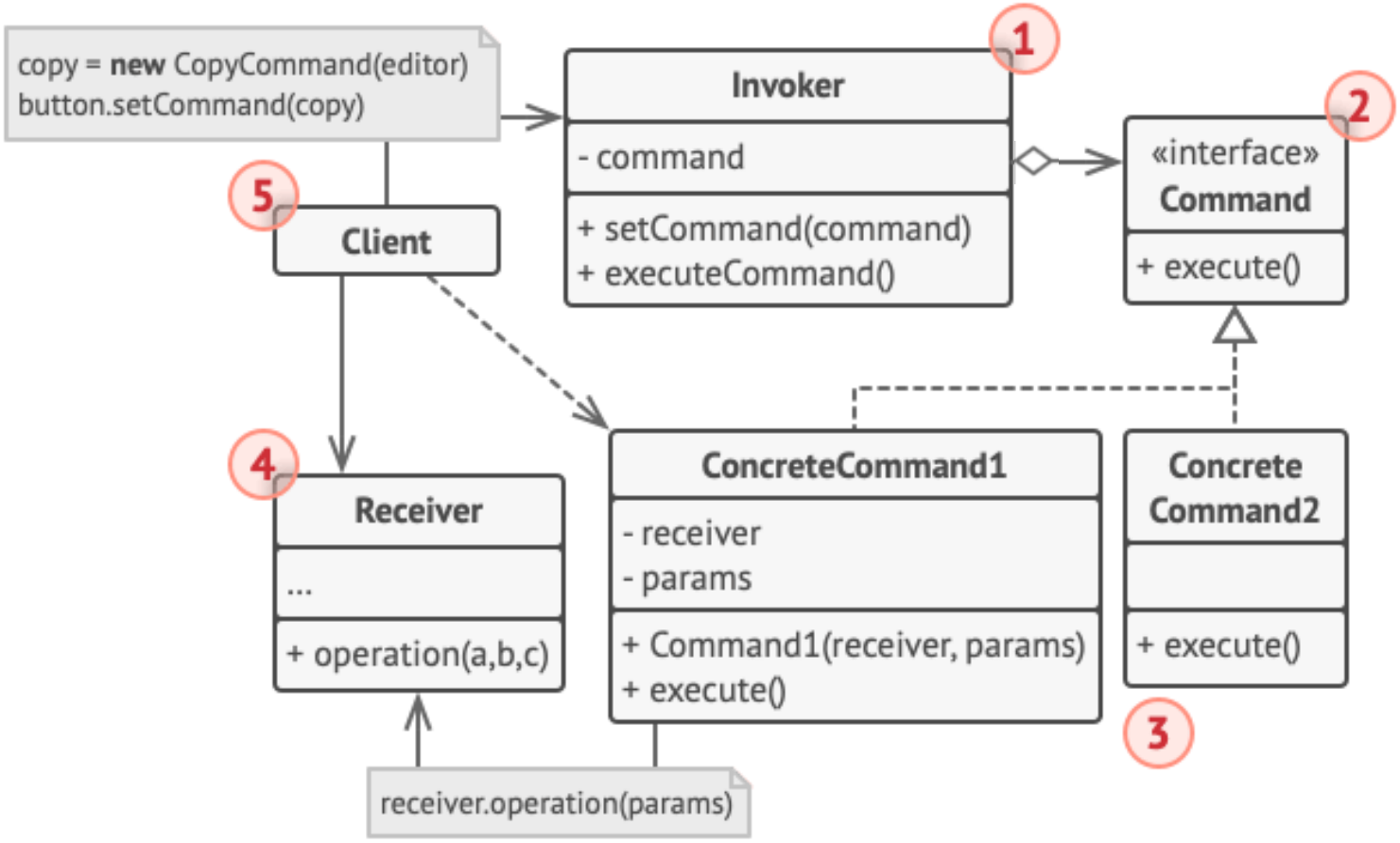






https://refactoring.guru/design-patterns/command

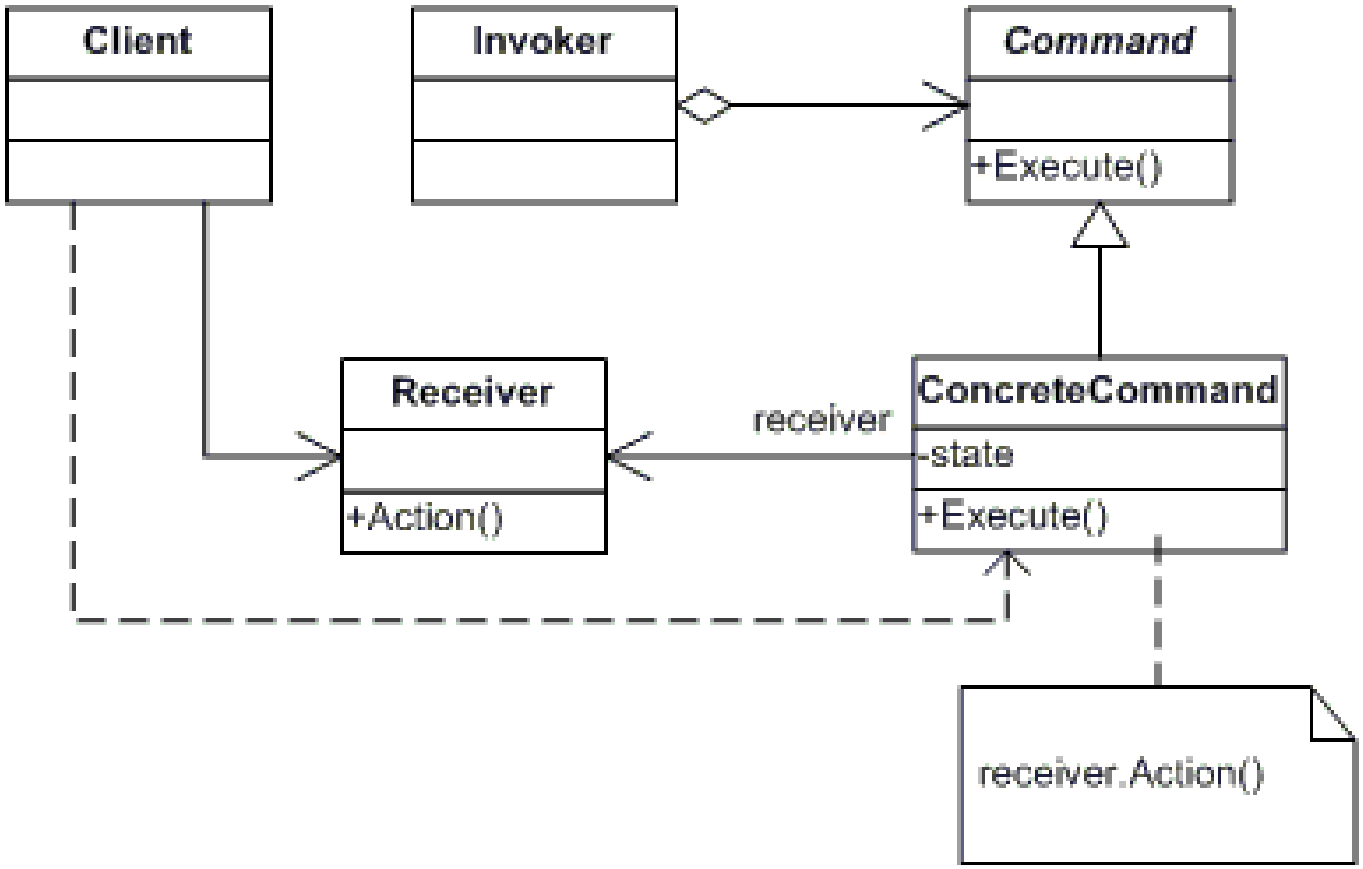
Structure


copy = new CopyCommand(editor)
button.setCommand(copy)
1
Invoker
- command
+ setCommand(command)
+ executeCommand()
2
«interface»
Command
+ execute()
5
Client
4
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
3
receiver.operation(params)

Class Diagramm

- **Command**
  - This is an interface for executing an operation.
- **ConcreteCommand**
  - This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
- **Client**
  - This class creates the ConcreteCommand class and associates it with the receiver.
- **Invoker** (Sender)
  - This class asks the command to carry out the request.
- **Receiver**
  - This class knows to perform the operation.

UML - Command

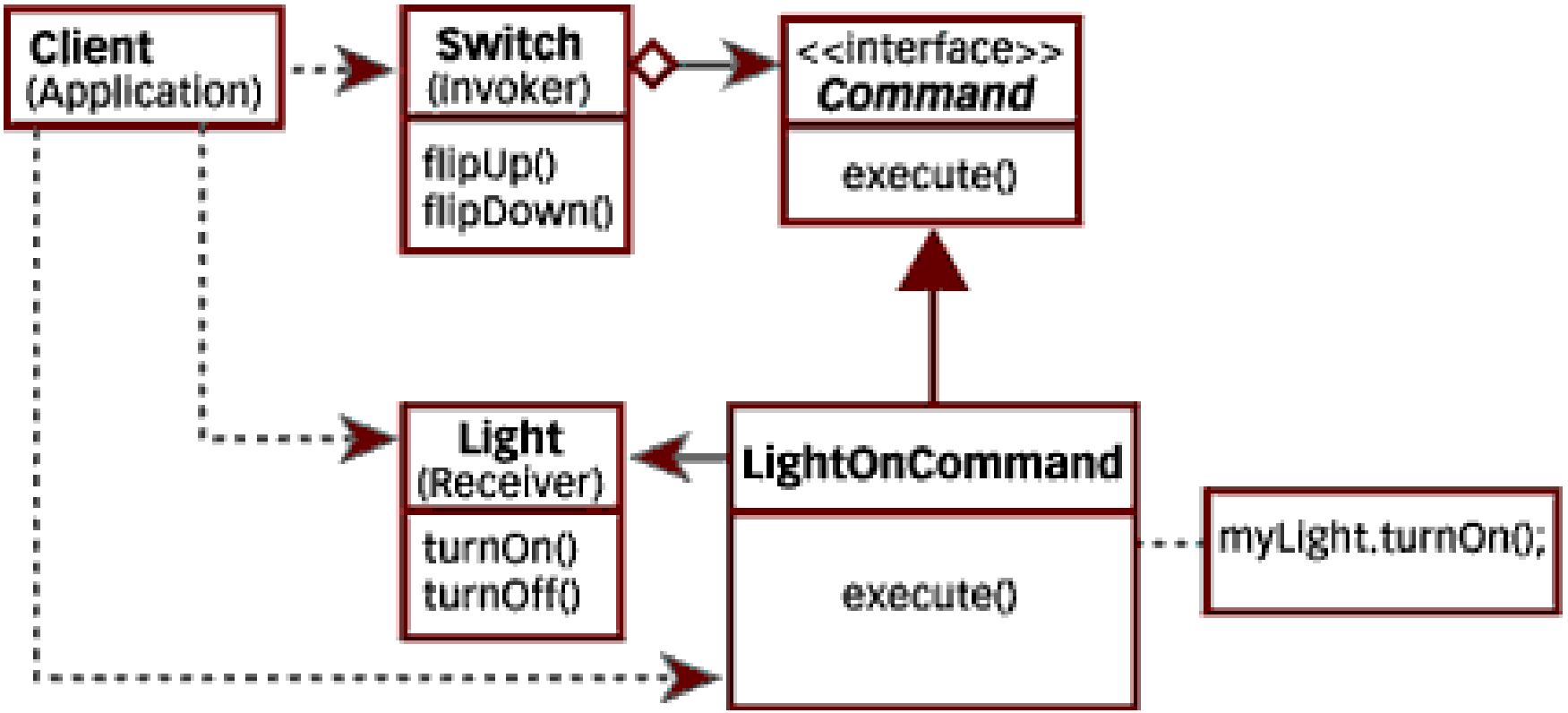

Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

Exercise Light & Fan

- The example shows a Fan and a Light.
  - Fan: startRoteate() & stopRotate()
  - Light: turnOn() & turnOff()
- Our objective is to develop a Switch that can turn either object on or off.

- We see that the Fan and the Light have different interfaces, which means the Switch has to be independent of the Receiver interface or it has no knowledge of the code -> Receiver's interface.

Example – Light On/Off



Explanation UML Diagram

- The Switch -- an aggregation of Command objects.
  - It has flipUp() and flipDown() operations in its interface.
  - Switch is called the invoker because it invokes the execute operation in the command interface.
- The concrete command, LightOnCommand, implements the execute operation of the command interface.
  - It has the knowledge to call the appropriate Receiver object's operation. It acts as an adapter in this case.

Example Code: Class Fan & Light

```
class Fan {
        public void startRotate() {
                System.out.println("Fan is rotating");
        }
        public void stopRotate() {
                System.out.println("Fan is not rotating");
        }
}
class Light {
        public void turnOn( ) {
                System.out.println("Light is on ");
        }
        public void turnOff( ) {
                System.out.println("Light is off");
        }
}
```

Command Interface

Command.java

```
public interface Command {
        public abstract void execute ( );
}
```

- How would the implementation of the command classes for fan and light look like?

Commands for Light

```
class LightOnCommand implements Command {
        private Light myLight;
        public LightOnCommand ( Light L) {
                myLight  =  L;
        }
        public void execute( ) {
                myLight . turnOn( );
        }
}
class LightOffCommand implements Command {
        private Light myLight;
        public LightOffCommand ( Light L) {
                myLight  =  L;
        }
        public void execute( ) {
                myLight . turnOff( );
        }
}
```

Commands for Fan

```
class FanOnCommand implements Command {
        private Fan myFan;
        public FanOnCommand ( Fan F) {
                myFan  =  F;
        }
        public void execute( ) {
                myFan . startRotate( );
        }
}
class FanOffCommand implements Command {
        private Fan myFan;
        public FanOffCommand ( Fan F) {
                myFan  =  F;
        }
        public void execute( ) {
                myFan . stopRotate( );
        }
}
```

Main zum Testen

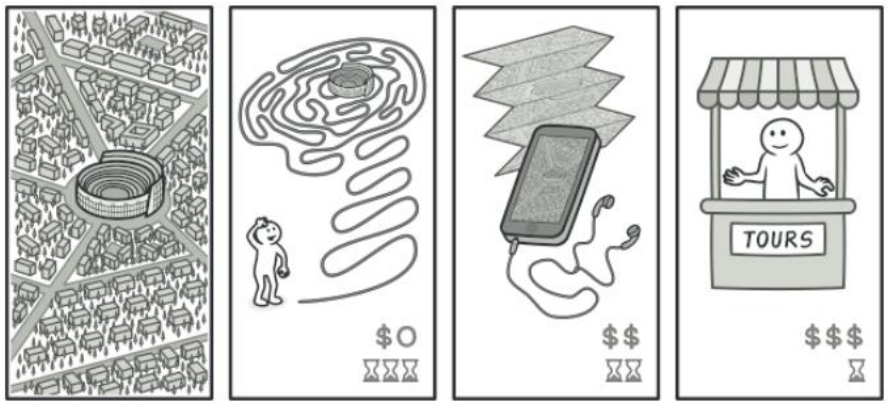
```
public class TestCommand {
                public static void main(String[] args) {
                        Light  testLight = new Light( );
                        LightOnCommand testLOC = new
LightOnCommand(testLight);
                        LightOffCommand testLFC = new
LightOffCommand(testLight);
                        Switch testSwitch = new Switch(
testLOC,testLFC);
                        testSwitch.flipUp( );
                        testSwitch.flipDown( );
                        Fan testFan = new Fan( );
                        FanOnCommand foc = new FanOnCommand(testFan);
                        FanOffCommand ffc = new FanOffCommand(testFan);
                        Switch ts = new Switch( foc,ffc);
                        ts.flipUp( );
                        ts.flipDown( );
                }
}
```

Iterator

This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

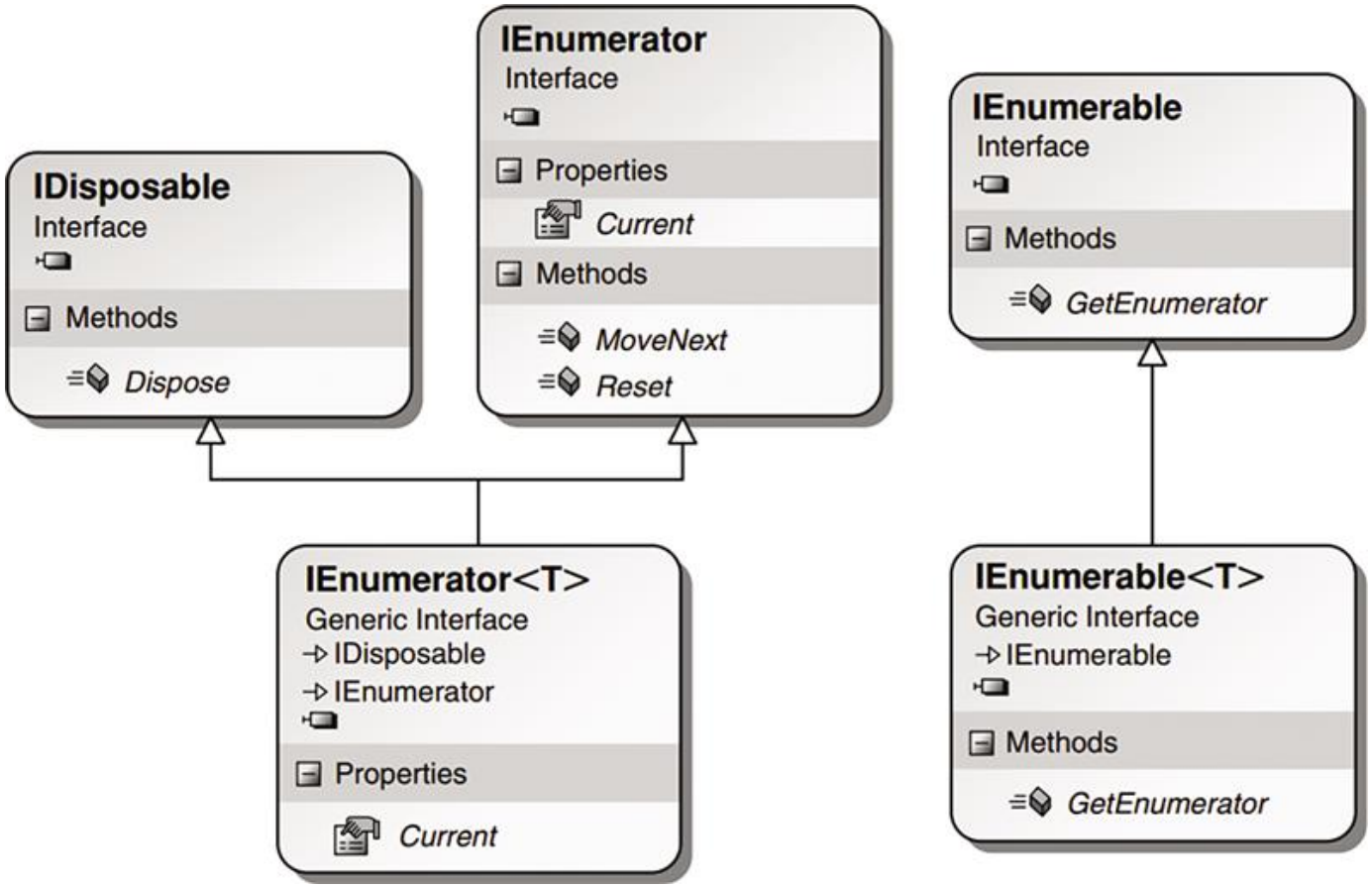
Intent

- Provide a way to
  access the elements of an aggregate object sequentially
  without exposing its underlying representation
- Promote to "full object status"
  the traversal of a collection



Various ways to walk around Rome.

IEnumerator & Ienumerable in C#



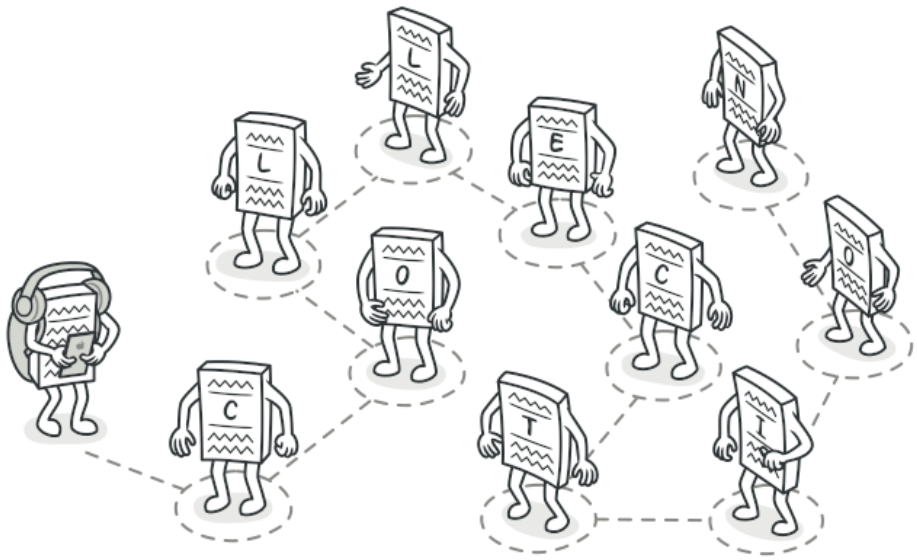
IEnumerator<T>

- IEnumerator<T> implements
  IEnumerator & IDisposable

```
public interface IEnumerator<out T> : IEnumerator, IDisposable {
    T Current { get; }
}
```

```
public interface IEnumerator {
    object Current { get; }

    bool MoveNext();
    void Reset();
}
```

```
public interface IDisposable {
    void Dispose();
}
```



Iterator for LinkList<T>

Using IEnumerable & IEnumerator Interface of C# Library System.Collections;

Iterator for LinkedLists

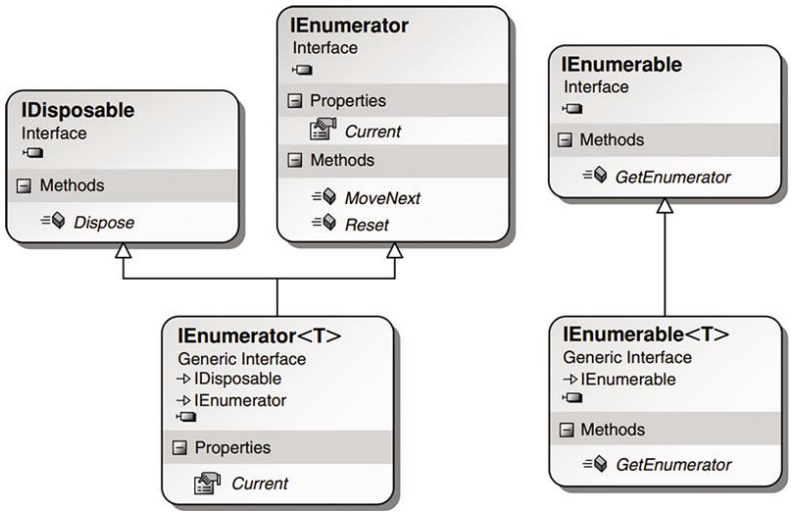
```
public interface IList<T> : IEnumerable<Node<T>> where T : IComparable<T>
```

```
public class GenericLinkedList<T> : IList<T>, IEnumerable<Node<T>> where T : IComparable<T> {
    public Node<T> Head { get; private set; }
    public Node<T> Tail { get; private set; }

    public IEnumerator<Node<T>> GetEnumerator() {
        return new GenIterator<T>(this);
    }

    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
```

```
public class GenIterator<T> : IEnumerator<Node<T>>
                    where T : IComparable<T> {
```

Generic Iterator for LinkedList

```
public class GenIterator<T> : IEnumerator<Node<T>>
                    where T : IComparable<T> {

    private IList<T> _list;
    private Node<T> _current;

    public GenIterator(IList<T> _list) {
        this._list = _list;
    }
    public bool MoveNext() {
        if(_current == null) {
            _current = _list.Head;
        } else {
            _current = _current.Next;
        }
        return _current != null;
    }
    public void Reset() {
        _current = null;
    }

    public Node<T> Current {
        get => _current;
    }
    object IEnumerator.Current => Current;

    public void Dispose() {
    }
}
```



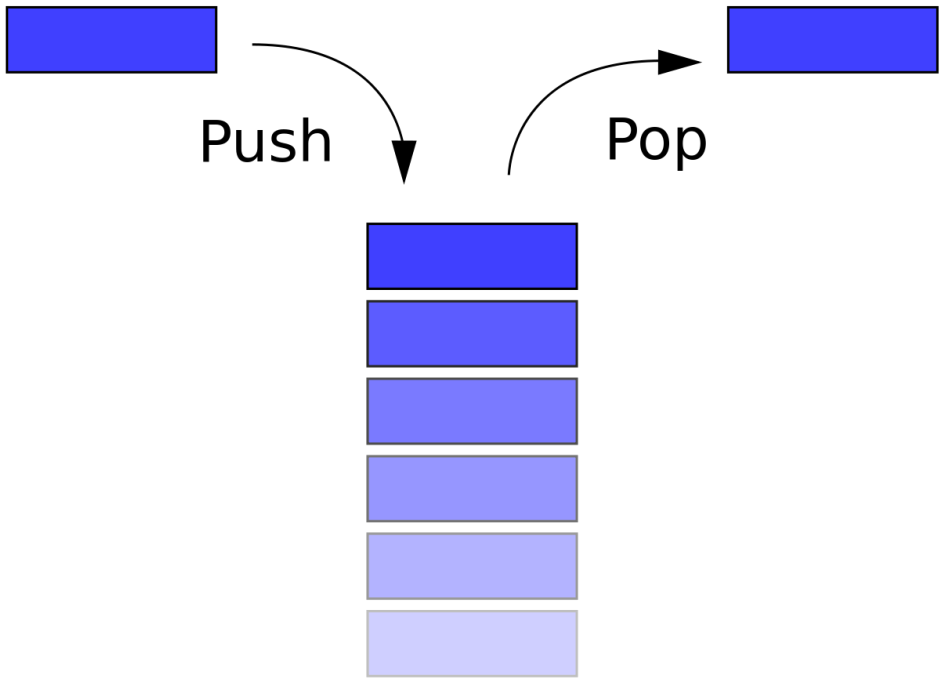
Test the Iterator

- Every Collection is Enummerable and can be used with a foreach:
- Now your LinkedList kann be used with a foreach:

```
int[] arr = { 5, 9, 3, 19, 45, 66, 22, 56 };

LinkList<int> l = new LinkList<int>();
l.AddArray(arr);

Console.WriteLine("Foreach:");
foreach (object item in l) {
    Console.WriteLine(item);
}
```

Short Version with yield

```
  îŋşîđê Ğêŋêsîç Ľîŋlêđ Ľîşʧ
řụčľîç ÍÉŋụŋêsắʧǫ̣s Ţ  ĞêʧÉŋụŋêsắʧǫ̣s


  Ǹǫ̣đê Ţ  ʧşắŵ   ḥêắđ
  xḥîľê  ʧsắŵ    ŋụľľ

    ỳîệľđ sêʧɥsŋ ʧsắŵ đắʧắ
    ʧsắŵ   ʧsắŵ ŋêẏʧ
```



Generic Stack

Implement a Generic Stack

Initialize the stack with a maximum amount

Push puts an element into the stack at the top

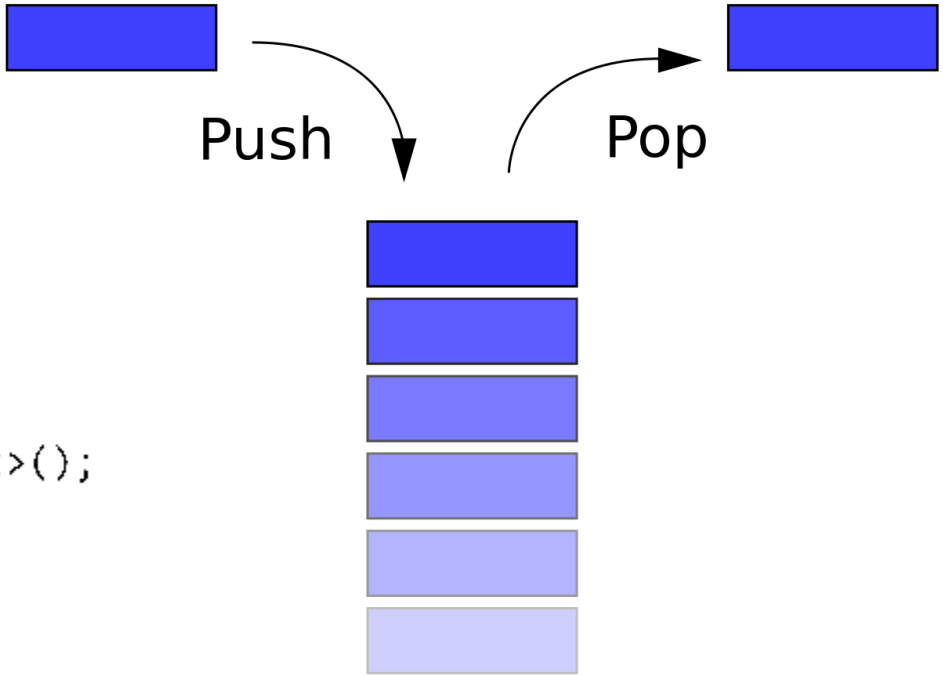
Pop gets an element of the stack from the top

Add an iterator for the Stack

Generic Stack

- For example, here is how you define and use a generic stack:

```
public class Stack<T>
{
    T[] m_Items;
    public void Push(T item)
    {...}
    public T Pop()
    {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```



Stack Implementation

```
class Stack<T>
{
    private readonly int size;
    private T[] elements;
    private int pointer = 0;

    public Stack(int size)...

    public void Push(T element)...

    public T Pop()...

    public int Length...
}
```

Stack<T>

Konstruktor

Push & Pop

Length

```
class Stack<T>
{
    private readonly int size;
    private T[] elements;
    private int pointer = 0;

    public Stack(int size)...

    public void Push(T element) {
        if (pointer >= this.size)
            throw new StackOverflowException();
        elements[pointer] = element;
        pointer++;
    }

    public T Pop() {
        pointer--;
        if (pointer >= 0)
            return elements[pointer];
        else {
            pointer = 0;
            throw new InvalidOperationException("Der Stack ist leer");
        }
    }

    public int Length {
        get { return this.pointer; }
    }
}
```

```
public Stack(int size)
{
    this.size = size;
    elements = new T[size];
}
```

Test the Stack & Output

```
Stack<int> stackInt = new Stack<int>(50);
stackInt.Push(1);
stackInt.Push(2);
Console.WriteLine(stackInt.Pop());
stackInt.Push(3);
stackInt.Push(4);
stackInt.Push(5);
stackInt.Push(6);
stackInt.Push(7);
Console.WriteLine(stackInt.Pop());
Console.WriteLine(stackInt.Pop());
Console.WriteLine(stackInt.Pop());
Console.WriteLine(stackInt.Pop());
Console.WriteLine(stackInt.Pop());
```

```
2
7
6
5
4
3
1
```

```
try {
    Stack<string> stackInt = new Stack<string>(5);
    stackInt.Push("Hallo");
    stackInt.Push("Griaß di");
    Console.WriteLine(stackInt.Pop());
    stackInt.Push("Servus");

    Console.WriteLine(stackInt.Pop());
    Console.WriteLine(stackInt.Pop());
}
catch (StackOverflowException e) {
    Console.WriteLine(e.Message);
}
catch (InvalidOperationException e) {
    Console.WriteLine(e.Message);
}
```

```
Griaß di
Servus
Hallo
```

Stack Iterator

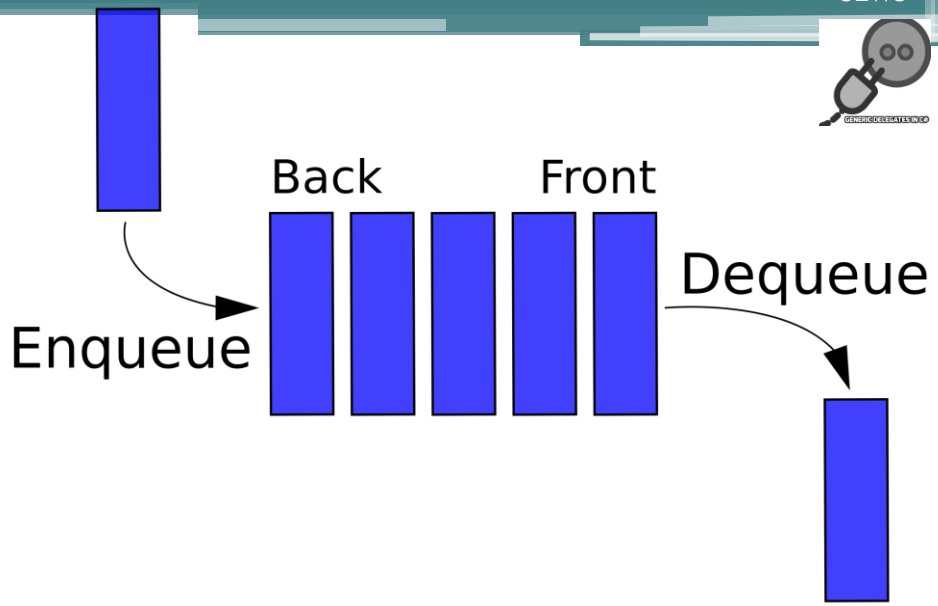
- Implement the IEnumerable for your Stack<T>
- Write a class StackIterator, which implements the IEnumerator<T> interface

```
MyStack<int> ms = new MyStack<int>();
ms.Push(5);
ms.Push(4);
ms.Push(5);
ms.Push(6);
ms.Push(8);
ms.Push(9);
foreach (var item in ms)
{
    Console.WriteLine(item);
}
```

Stack<T>

```
public class MyStack<T> : IEnumerable<T> {
    private T[] arr;
    private int current = 0;
    public MyStack(int size=100) {
        arr = new T[size];
        this.Size = size;
    }
    public int Size { get; init; }
    public int Length ...
    public void Push(T item) ...
    public T Pop() ...

    public IEnumerator<T> GetEnumerator() {
        return new GenIteratorStack<T>(arr, Length);
    }

    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

```
public class GenIteratorStack<T> : IEnumerator<T> {
    private T[] _stack;
    private T? _current;
    private int count;
    public GenIteratorStack(T[] stack, int length) {
        this._stack = stack;
        count = length - 1;
    }
    object? IEnumerator.Current => Current;
    public T? Current { get => _current;  }
    public bool MoveNext() {
        if (_current == null) {
            if (_stack.Length != 0) {
                _current = _stack[count];
                count--;
                return true;
            }
            return false;
        } else {
            if (count > -1) {
                _current = _stack[count];
                count--;
                return true;
            } else {
                return false;
            }
        }
    }
    public void Reset() {
        _current = default;
    }
  public void Dispose() { }
}
```



Generic Queue

Implement a generic queue
Use a List<T>
Dequeue – removes the first element
Enqueue – adds an element
Add an iterator for the queue

Queue with Enqueue and Dequeue

```
class Queue<T> {

    List<T> elements;
    public Queue() {
        elements = new List<T>();
    }

    public void Enqueue(T element) {
        elements.Add(element);
    }

    public T Dequeue() {
        T e = elements.ElementAt(0);
        elements.RemoveAt(0);
        return e;
    }
}
```

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(1);
queue.Enqueue(2);
queue.Enqueue(3);
Console.WriteLine(queue.Dequeue());
Console.WriteLine(queue.Dequeue());
Console.WriteLine(queue.Dequeue());
```

```
Queue<string> queueS = new Queue<string>();
queueS.Enqueue("Hallo");
queueS.Enqueue("Grias di");
queueS.Enqueue("Servus!");
Console.WriteLine(queueS.Dequeue());
Console.WriteLine(queueS.Dequeue());
Console.WriteLine(queueS.Dequeue());
```

Queue Iterator

- Queue<T> implements IEnumerable<T>
- Klasse QueueIterator implements IEnumerator<T>
  - with T Current und bool MoveNext()

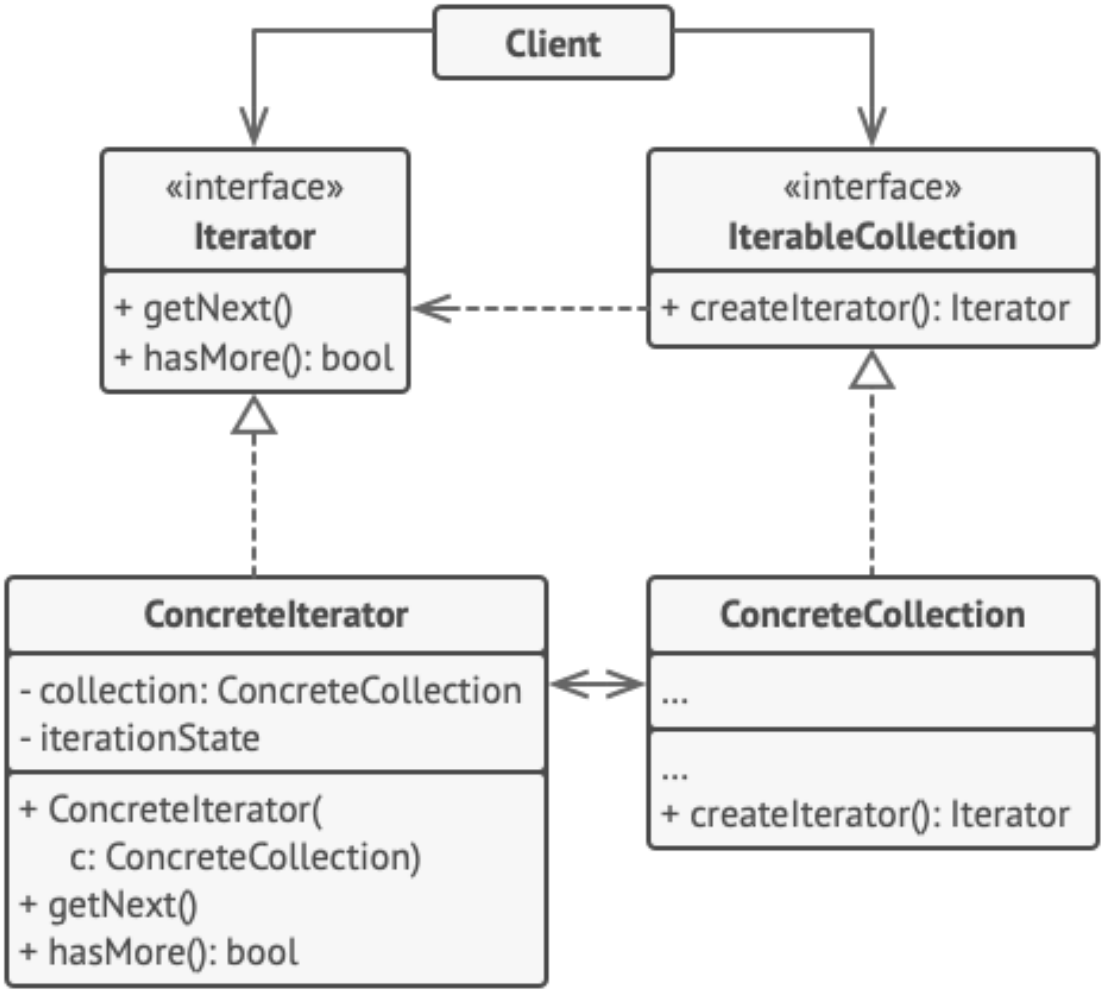
```
Queue<int> q = new Queue<int>();
q.Enqueue(4);
q.Enqueue(5);
q.Enqueue(8);
q.Enqueue(9);
q.Enqueue(10);
q.Enqueue(11);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

Queue<T> & QueueIterator<T>

```
public class MyQueue<T> : IEnumerable<T> {
    private List<T> list;
    public MyQueue() {
        list = new List<T>();
    }
    public int Length ...
    public void Enqueue(T item) ...
    public T Dequeue() ...
    public IEnumerator<T> GetEnumerator() {
        return new GenIteratorQueue<T>(list);
    }

    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

```
public class GenIteratorQueue<T> : IEnumerator<T> {
    private List<T> _queue;
    private T? _current;
    private int count = 0;
    public GenIteratorQueue(List<T> queue) {
        this._queue = queue;
    }
    object? IEnumerator.Current => this.Current;
    public T? Current {  get => _current; }
    public bool MoveNext() {
        if (_current == null) {
            if (_queue.Count != 0) {
                _current = _queue[count];
                count++;
                return true;
            }
            return false;
        } else {
            if (count < _queue.Count) {
                _current = _queue[count];
                count++;
                return true;
            } else {
                return false;
            }
        }
    }
    public void Reset() {
        _current = default;
    }
    public void Dispose() { }
}
```

UML Diagramm



Participants

- **Iterator** (IEnumerator)
  - defines an interface for accessing and traversing elements.
- ConcreteEnumerator (Iterator)
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **IterableCollection** (IEnumerable)
  - defines an interface for creating an Iterator object
- ConcreteAggregate (Aggregate)
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Implementation

```
/// <summary>
/// The 'Aggregate' abstract class
/// </summary>
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class ConcreteAggregate : Aggregate
{
    private ArrayList _items = new ArrayList();

    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }
```

Aggregate

```
    // Gets item count
    public int Count
    {
        get { return _items.Count; }
    }

    // Indexer
    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}
```

Iterator

```
/// <summary>
/// The 'Iterator' abstract class
/// </summary>
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

/// <summary>
/// The 'ConcreteIterator' class
/// </summary>
class ConcreteIterator : Iterator
{
    private ConcreteAggregate _aggregate;
    private int _current = 0;

    // Constructor
    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this._aggregate = aggregate;
    }

    // Gets first iteration item
    public override object First()
    {
        return _aggregate[0];
    }
```

```
    // Gets next iteration item
    public override object Next()
    {
        object ret = null;
        if (_current < _aggregate.Count - 1)
        {
            ret = _aggregate[++_current];
        }

        return ret;
    }

    // Gets current iteration item
    public override object CurrentItem()
    {
        return _aggregate[_current];
    }

    // Gets whether iterations are complete
    public override bool IsDone()
    {
        return _current >= _aggregate.Count;
    }
```

Main: Using an Iterator

```
static void Main()
{
    ConcreteAggregate a = new ConcreteAggregate();
    a[0] = "Item A";
    a[1] = "Item B";
    a[2] = "Item C";
    a[3] = "Item D";

    // Create Iterator and provide aggregate
    Iterator i = a.CreateIterator();

    Console.WriteLine("Iterating over collection:");

    object item = i.First();
    while (item != null)
    {
        Console.WriteLine(item);
        item = i.Next();
    }
```