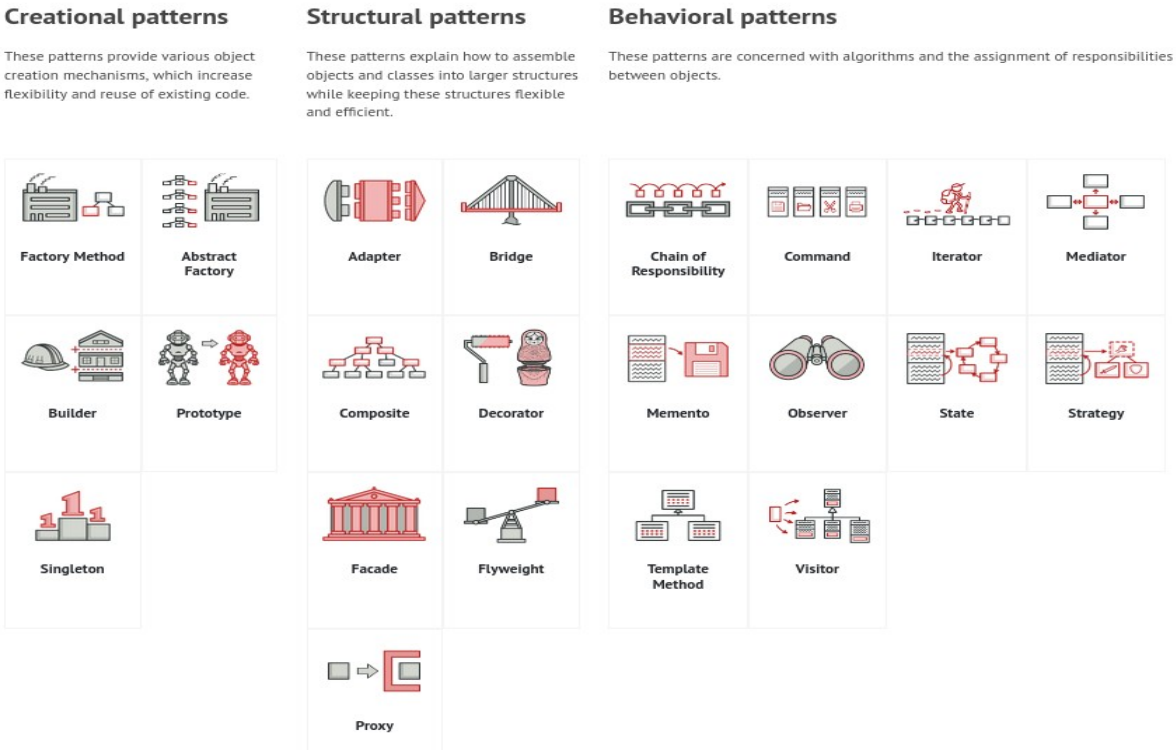
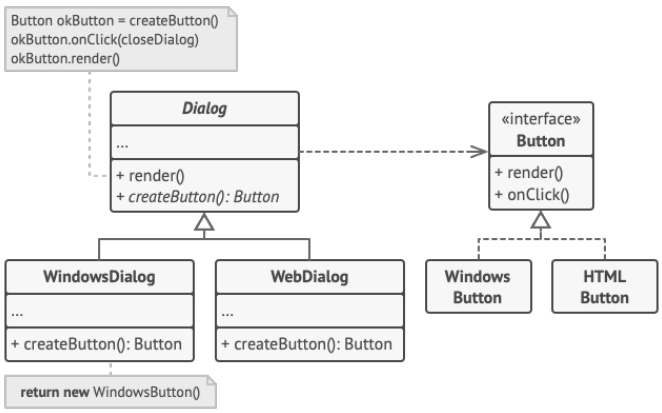
Design Patterns



CREATIONAL PATTERNS

Factory Method

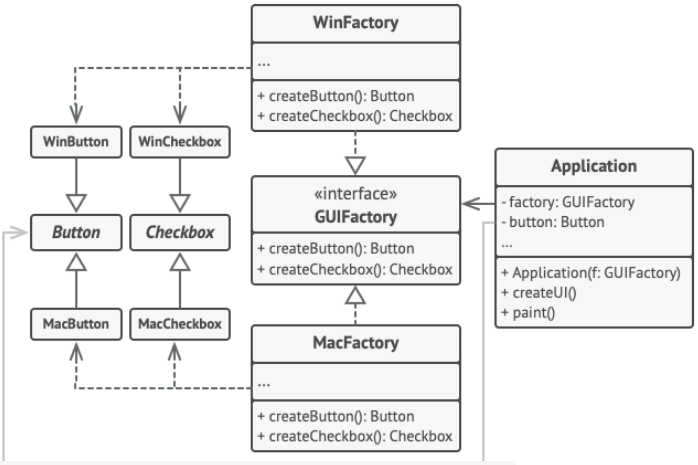


Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.

```
namespace FactoryPattern{
    interface Transport{
        abstract void doSomething();
    }

    class Truck: Transport{
        public void doSomething(){
            Console.WriteLine("I am a truck");
        }
    }

    class Ship: Transport{
        public void doSomething(){
            Console.WriteLine("I am a ship");
        }
    }

    abstract class Factory{
        public abstract Transport createTransport();
    }

    class ShipFactory: Factory{
        public override Transport createTransport()
        {
            return new Ship();
        }
    }

    class TruckFactory: Factory{
        public override Transport createTransport()
        {
            return new Truck();
        }
    }
}
```

Abstract Factory Method

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




```
interface IComputer{
    public abstract void run();
}

class ModernComputer : IComputer
{
    public void run()
    {
        Console.WriteLine("Modern Computer is running");
    }
}

class OldComputer : IComputer
{
    public void run()
    {
        Console.WriteLine("Old Computer is running");
    }
}

interface IPhone{
    public abstract void call();
}

class ModernPhone : IPhone
{
    public void call()
    {
        Console.WriteLine("Modern Phone is calling");
    }
}

class OldPhone : IPhone
{
    public void call()
    {
        Console.WriteLine("Old Phone is calling");
    }
}
```

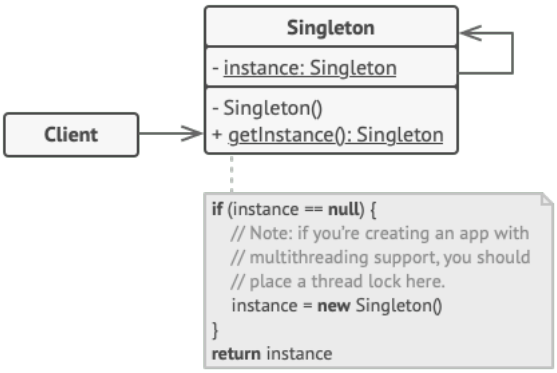
```
interface IFactory{
    public abstract IComputer createComputer();
    public abstract IPhone createPhone();
}

class ModernFactory : IFactory
{
    public IComputer createComputer()
    {
        return new ModernComputer();
    }

    public IPhone createPhone()
    {
        return new ModernPhone();
    }
}

class OldFactory: IFactory
{
    public IComputer createComputer()
    {
        return new OldComputer();
    }

    public IPhone createPhone()
    {
        return new OldPhone();
    }
}
```

Singleton Method

```
class Singleton{
    private static Singleton? instance;
    private Singleton(){}

    public static Singleton Instance{
        get{
            if(instance == null){
                instance = new Singleton();
            }
            return instance;
        }
    }
    public void instancePrint(){
        Console.WriteLine("Singleton Instance");
    }
}
```

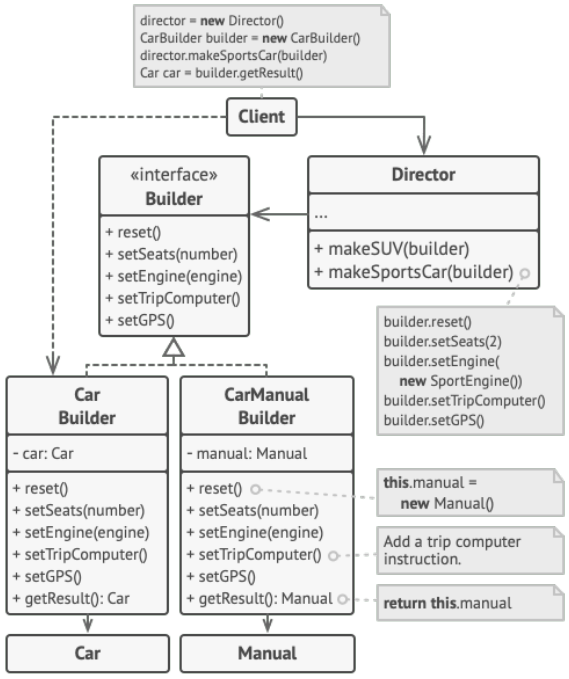
Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

The Singleton's constructor should be hidden from the client code.



Builder Method

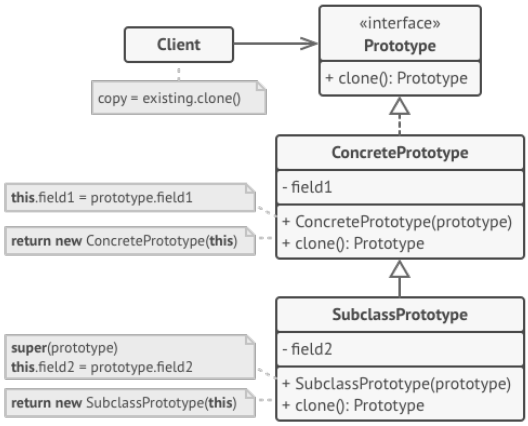
The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*. Use the Builder pattern when you want your code to be able to create different representations of same product (for example, stone and wooden houses).




```
interface IHomeBuilder{
        abstract public void BuildWalls();
        abstract public void BuildDoors();
        abstract public void BuildWindows();
    }

    class StoneHomeBuilder : IHomeBuilder
    {
        public void BuildDoors()
        {
            Console.WriteLine("Building Stone Doors");
        }

        public void BuildWalls()
        {
            Console.WriteLine("Building Stone Walls");
        }

        public void BuildWindows()
        {
            Console.WriteLine("Building Stone Windows");
        }
    }

    class WoodenHomeBuilder: IHomeBuilder{
        public void BuildDoors()
        {
            Console.WriteLine("Building Wooden Doors");
        }

        public void BuildWalls()
        {
            Console.WriteLine("Building Wooden Walls");
        }

        public void BuildWindows()
        {
            Console.WriteLine("Building Wooden Windows");
        }
    }

    class Director{
        private IHomeBuilder _builder;
        public IHomeBuilder Builder{
            set => _builder = value;
        }

        public void ConstructFullHome(){
            _builder.BuildWalls();
            _builder.BuildDoors();
            _builder.BuildWindows();
        }
    }
```

Prototype Method

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy. (when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects).

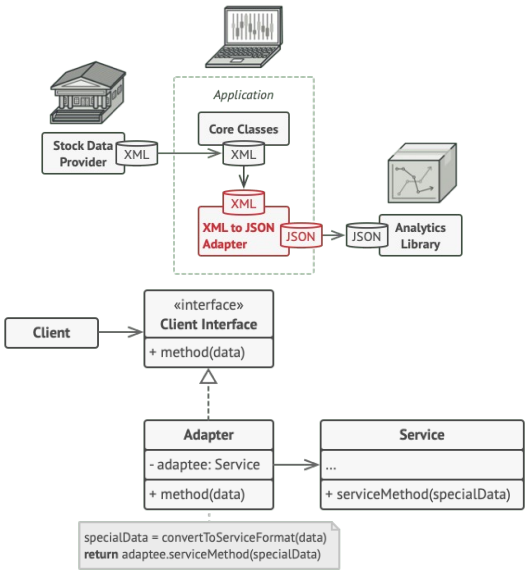



```
interface IPrototype<T>{
    T Clone();
}
class Home{
    public int homeID;
    public string city;
    public Home(int homeID, string city){
        this.homeID = homeID;
        this.city = city;
    }
}
class Person: IPrototype<Person>{
    public string Name;
    public int Age;
    public Home Home;
    public Person(string name, int age, int homeID, string city){
        Name = name;
        Age = age;
        Home = new Home(homeID, city);
    }
    public Person Clone(){
        Person p = new Person(Name, Age, Home.homeID, Home.city);
        return p;
    }

    /*Another way without interface and manual copying*/
    public Person ShallowCopy()
    {
        return (Person) this.MemberwiseClone();
    }

    public Person DeepCopy()
    {
        Person other = (Person) MemberwiseClone();
        other.Home = new Home(Home.homeID, Home.city);
        return other;
    }

    public void Print()
    {
        Console.WriteLine("Name: " + Name + " Age: " + Age + " HomeID: " + Home.homeID + " City: " + Home.city);
    }
}
```

STRUCTURAL PATTERNS

Adapter Method

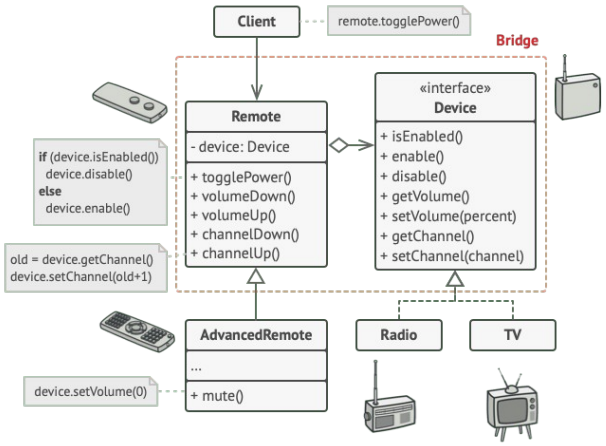
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

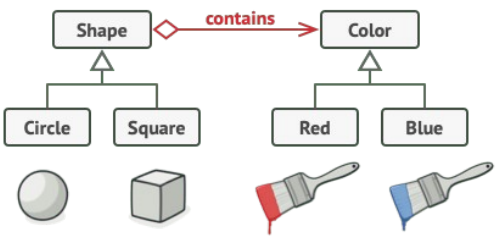



```
interface Request{
  public void sendRequest(string xml);
}

class XMLRequester: Request{
  public void sendRequest(string xml)
  {
      Console.WriteLine(xml);
  }
}

class JSONAdapter: Request{

  public void sendRequest(string xml)
  {
      string json = XMLtoJSON(xml); //pseudo
    Console.WriteLine(json);
  }
}
```

Bridge Method

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies

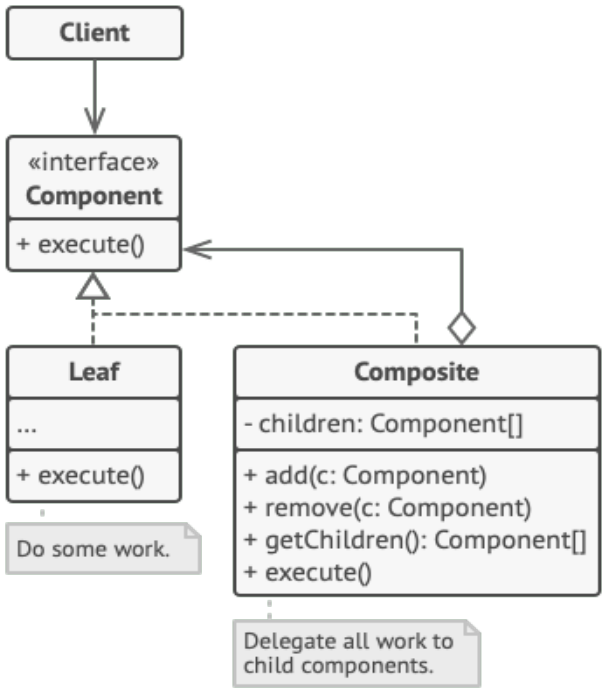





```
interface IElectronic{
    public void on();
    public void off();
}

class TV: IElectronic{
    public void on(){
        Console.WriteLine("TV - ON");
    }

    public void off(){
        Console.WriteLine("TV - OFF");
    }
}

class Air: IElectronic{
    public void on(){
        Console.WriteLine("Air Conditioner - ON");
    }

    public void off(){
        Console.WriteLine("Air Conditioner - OFF");
    }
}

abstract class Remote{
    private IElectronic device;

    public Remote(IElectronic dev)
    {
        this.device = dev;
    }

    abstract void turnOn();
    abstract void turnOff();
}

class ClassicRemote: Remote{
    public void turnOn(){
        this.device.on();
    }
    public void turnOff(){
        this.device.off();
    }
}
```

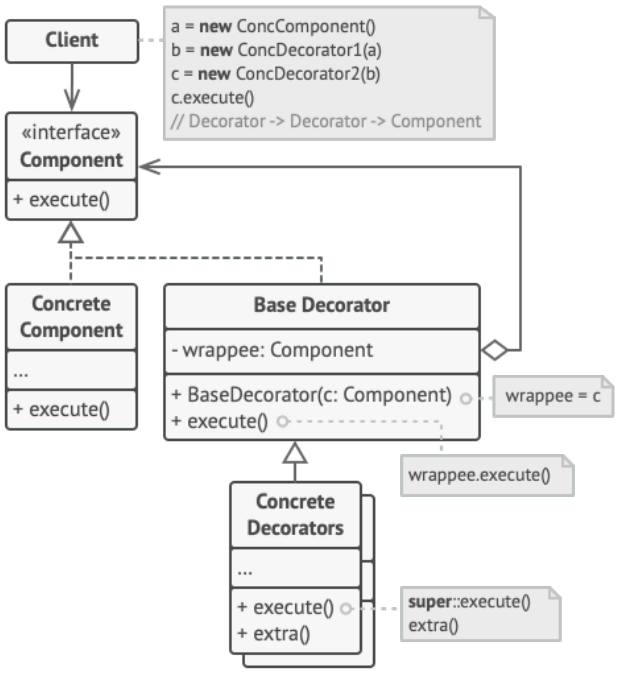
Composite Method

```
interface IItem{
    abstract double calculateProduct();
}

class Product: IItem{
        string productName;
    double price;
    public double calculateProduct(){
        return price;
    }
}

class Box: IItem{
    IItem[] products;

    public double calculateProduct(){
        double price = 0;

        foreach(IItem p in products){
            price += p.calculateProduct();
        }
        return (price);
    }
}
```



Decorator Method

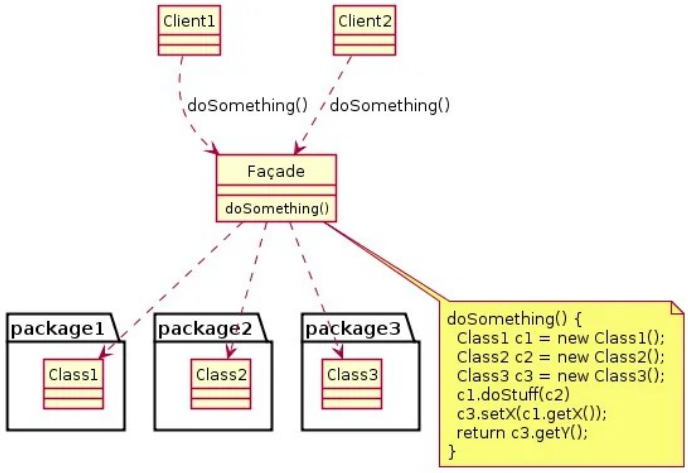
```
interface Shape{
    abstract void draw();
}

class Rectangle: Shape{
    public void draw(){
        Console.WriteLine("Rect Draw");
    }
}

class Circle: Shape{
    public void draw(){
        Console.WriteLine("Circle Draw");
    }
}

abstract class ShapeDecorate: Shape{
    private Shape shape;

    public ShapeDecorate(Shape s){
        this.shape = s;
    }

    public virtual void draw(){
        this.shape.draw();
    }
}

class BorderDecorate: ShapeDecorate{
    public override void draw(){
        base.draw();
        Console.WriteLine("Adding Borders");
    }
}
```



Facade Method

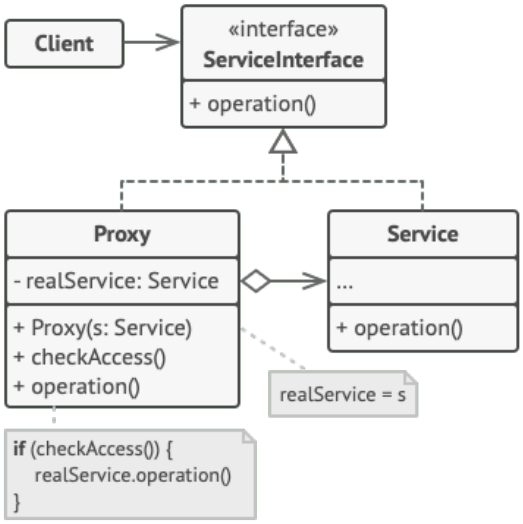
Facade arayüzünü kullandığımızda diğer classların metot detaylarını bilmek istemiyoruz. Yazacağımız bu arayüzde , alt sınıflar Facade sınıfımızdan bağımsız da çalışabilmeliler, Facade sadece bir kullanım kolaylığı sağlayan arayüz olacak tasarımımızda.

```
class A{
      public void AA(){
          Console.WriteLine("AA");
      }
  }
  class B{
      public void BB(){
          Console.WriteLine("BB");
      }
  }
  class C{
      public void CC(){
          Console.WriteLine("CC");
      }
  }
  class Facade{
      public void doSomething(){
            new A().AA();
          new B().BB();
          new C().CC();
      }
  }
```



Proxy Method

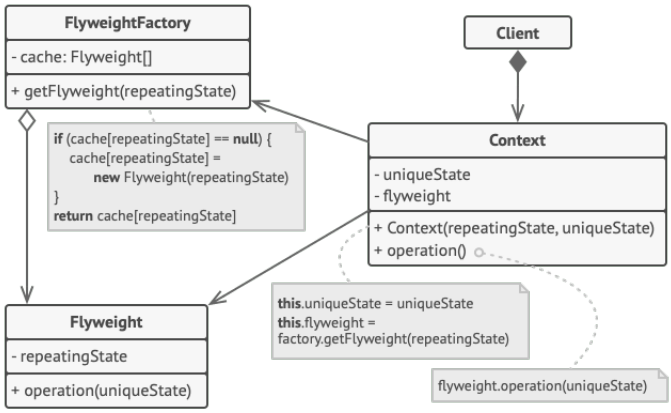
A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.




```
interface RequestSender{
    public void sendRequest();
}

class Real: RequestSender{
    public void sendRequest(){
        Console.WriteLine("Sending Request");
    }
}

class Proxy: RequestSender{
        RequestSender req;
    Proxy(RequestSender r)
    {
        this.req = r;
    }
    public void sendRequest(){
        Console.WriteLine("Proxy Checking");
        this.req.sendRequest();
    }
}
```

- **Flyweight Method**

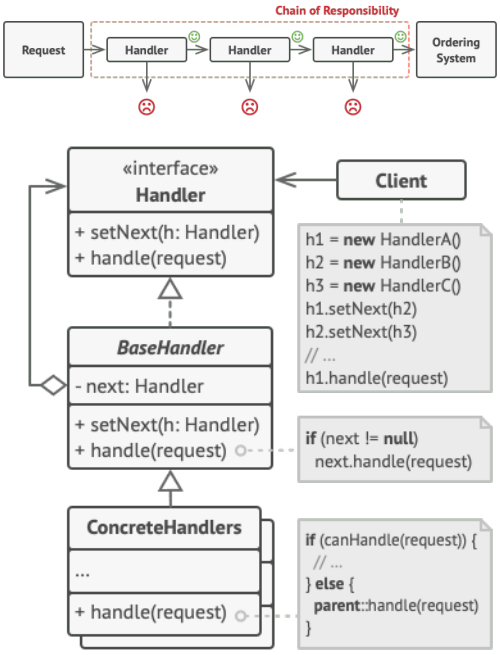
(HashMap kullanarak) objeyi herkes için tekrardan oluşturmaktansa doğrudan hashmap üzerinden alır.




```
abstract class Player{
        double health;
      string name;

      abstract void spawn();
  }

  class T: Player{
      T(){
          this.name = "CS_T";
          this.health = 100;
      }

      public void spawn(){
          Console.WriteLine("T spawn");
      }
  }

  class CT: Player{
      CT(){
          this.name = "CS_CT";
          this.health = 120;
      }

      public void spawn(){
          Console.WriteLine("CT spawn");
      }
  }

  class SoldierFactory{
      Dictionary<string, Player> players;

      SoldierFactory(){
          this.players = new Dictionary<string, Player>();
      }

      public Player getPlayer(string key)
      {
          if (!this.players.ContainsKey(key)){
              if (key == "T")
                  this.players.add(key, new T());
              else if (key == "CT")
                  this.players.add(key, new CT());
           }
          return (this.players[key]);
      }
  }
```

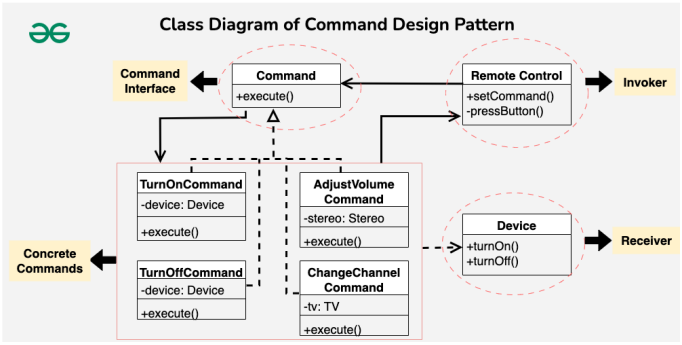
BEHAVIORAL PATTERNS

- **Chain of Responsibility Method**




```
abstract class Handler{
    private Handler next;

    public void setNext(Handler handler){
        this.next = handler;
    }

    public abstract void handle(Object o);
}

class AuthHandler: Handler{
    public override void handle(Object o)
    {
        if (o == null)
            Console.WriteLine("Auth Failed");
        else
        {
            Console.WriteLine("Auth OK");
          if (this.next != null)
                this.next.handle(o);
        }
    }
}

class ConnectionHandler: Handler{
    public override void handle(Object o)
    {
        if (o == null)
            Console.WriteLine("Connection Failed");
        else
        {
            Console.WriteLine("Connection OK");
          if (this.next != null)
                this.next.handle(o);
        }
    }
}
```

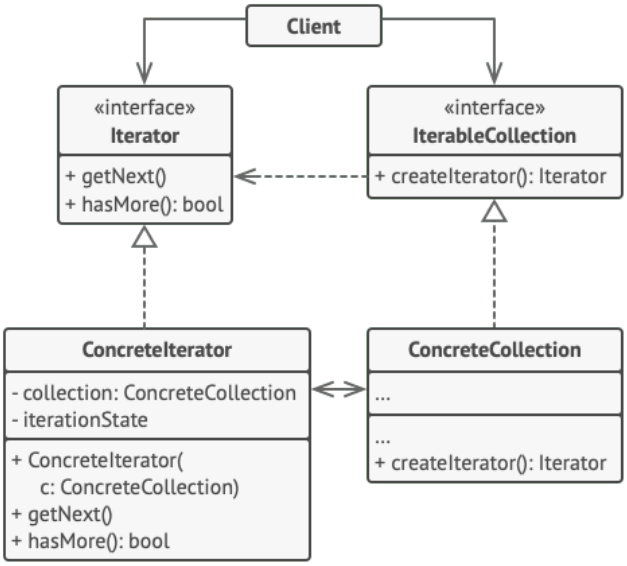
- **Command Method:** Çalıştırılan her komut için ayrı class açılır.




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

class TurnOnCommand: Command{
        Device dev;

        TurnOnCommand(Device d)
        {
            this.dev = d;
        }

        public void execute(){
            this.dev.turnOn();
        }
}

class TurnOffCommand: Command{
        Device dev;

        TurnOffCommand(Device d)
        {
            this.dev = d;
        }

        public void execute(){
            this.dev.turnOff();
        }
}
```

```
interface Device{
        abstract void turnOn();
        abstract void turnOff();
}

class TV: Device{
        public void turnOn(){
            Console.WriteLine("TV is ON");
        }

        public void turnOff(){
            Console.WriteLine("TV is Off");
        }
}

class Stereo: Device{
        public void turnOn(){
            Console.WriteLine("Stereo is ON");
        }

        public void turnOff(){
            Console.WriteLine("Stereo is Off")
        }
}

class Conroller{
        public void runCommand(Command c){
            c.execute();
        }
}
```

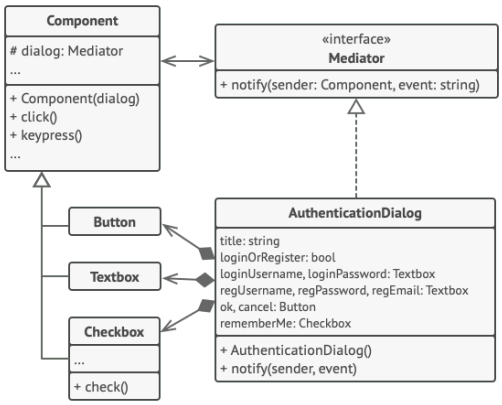
- **Iterator Method**




```
interface IIterable{
        IIterator getIterator();
}

interface IIterator<T>{
    bool moveNext();
    T getCurrent();
}

class Iterator<T>: IIterator<T>{
    T[] iterList;
    int index = -1;

    Iterator(T[] iterList){
            this.iterList = iterList;
    }

    public bool moveNext(){
            if (index >= iterList.Length)
                return false;
            index++;
            return true;
    }

    public T getCurrent(){
            return iterList[index];
    }
}

class MoneyBag: IIterable{
    int[] moneyBag= {1, 2, 3, 4, 5, 6, 7, 8, 9};

    public IIterator getIterator(){
            return new Iterator<int>(moneyBag);
    }
}
```

- **Mediator Method**

Obje ile bir işlem yapıldıktan sonra başka bir objeyi doğrudan triggerlamak yerine mediator üzerinden triggerlama yapılır

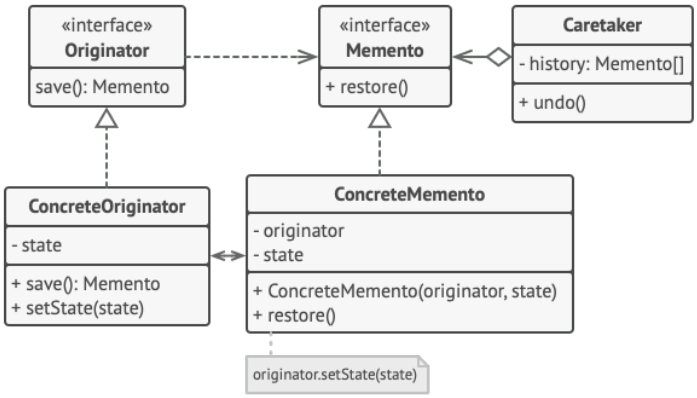



```
interface Mediator{
        public void notify(Component sender, string e);
}

class MainMediator: Mediator{
        ComponentA compA;
        ComponentB compB;

        MainMediator(ComponentA a, ComponentB b){
            this.compA = a;
            this.compB = b;
        }

        public void notify(Component sender, string e){
            if (e == "A")
            {
                    Console.WriteLine("Sender triggered event A");
                    if (sender.name == "A")
                        this.compB.executeA();
                    else
                        this.compA.executeA();
            }

            else if (e == "C")
            {
                    Console.WriteLine("Sender triggered event A");
                    if (sender.name == "C")
                        this.compB.executeC();
                    else
                        this.compA.executeC();
            }
        }
}
```

```
abstract class Component{
            public string name;
        public abstract void executeA();
        public abstract void executeC();

        Component(String name){
            this.name = name;
        }
}

class ComponentB: Component{
        public void executeA(){
            Console.WriteLine(name + " executed A");
        }

        public void executeC(){
            Console.WriteLine(name + " executed C");
        }
}

class ComponentA: Component{
        public void executeA(){
            Console.WriteLine(name + " executed A");
        }

        public void executeC(){
            Console.WriteLine(name + " executed C");
        }
}
```

- **Memento Method**

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation

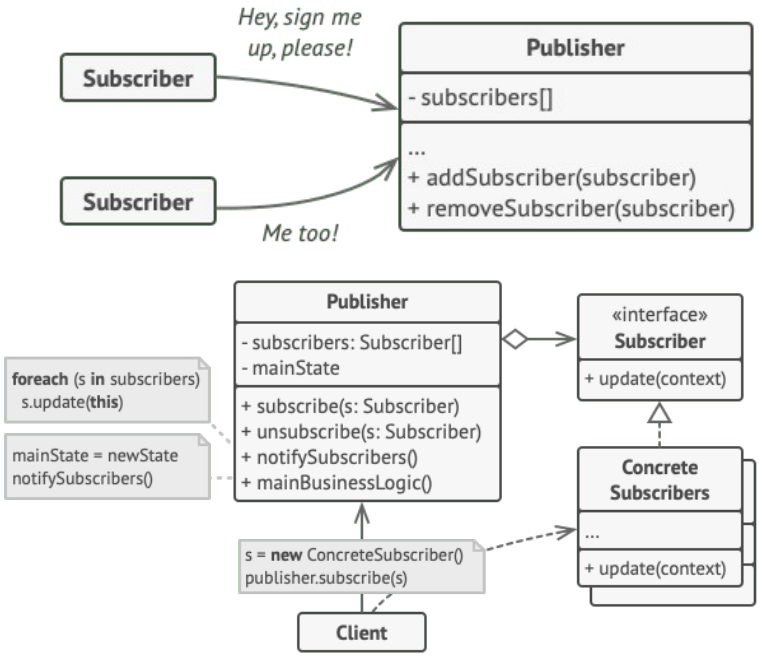



```
class MyObj{
        private DateTime date;
        private string currentState;

        /*SOME CODES HERE*/

        public void restore(Memento m)
        {
            this.date = m.getDate();
            this.currentState = m.getState();
        }

        public MyMemento save(){
            return new MyMemento(date, currentState);
        }
}

interface Memento{
    public DateTime getDate();
    public string getState();
}

class MyMemento: Memento{
    private DateTime date;
    private string state;

    MyMemento(DateTime date, string state){
        this.date = date;
        this.state = state;
    }

    public DateTime getDate() {return date;}

    public string getState() {return state;}
}

class BackUpGenerator{
    List<Memento> backups;
    MyObj origin;

    BackUpGenerator(MyObj o){
            this.origin = o;
    }

    public void backup(){
            backups.add(origin.save());
    }

    public void undo(){
        if (backups.Length > 1)
        {
            var memento = this.backups.Last();
            this.backups.Remove(memento);
            origin.restore(this.backups.Last());
        }
    }
}
```

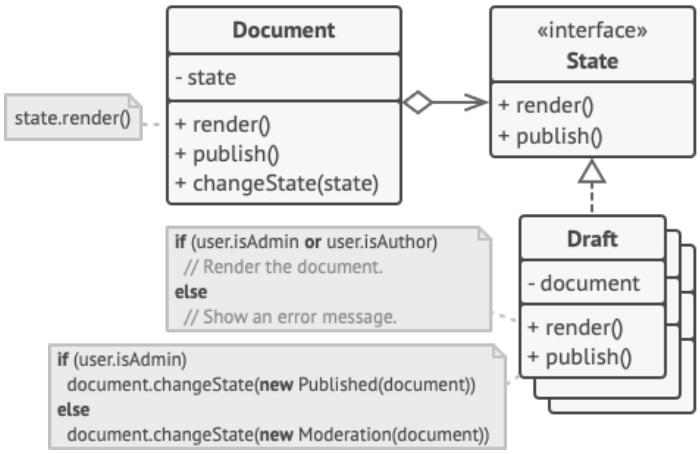
- **Observer Method**

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

```
interface Subscriber{
    public void sendEmail();
}

class Sub: Subscriber{
    int id;

    Sub(int id){
        this.id = id;
    }

    public void sendEmail(){
        Console.WriteLine("SENT to " + id);
    }
}

class Publisher{
    List<Sub> subs;

    public void addSub(){
        this.subs.Add(new Sub(subs.Count));
    }

    public void notifySubs(){
        foreach(var s in this.subs)
            s.sendEmail();
    }

    public void unSub(int id){
        this.subs.RemoveAt(id);
    }
}
```



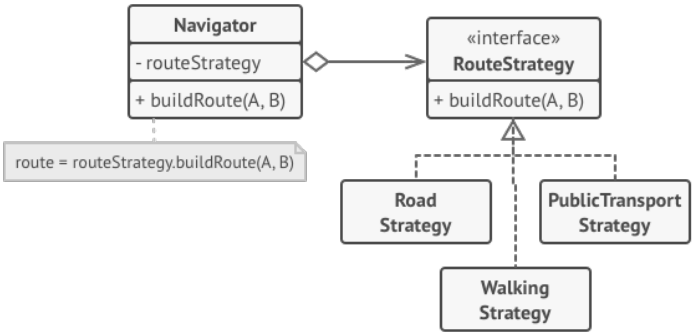
- **State Method:** if else kullanarak state' leri belirtmek yerine tüm stateleri ayrı obje olarak tanımlar.




```
class MediaPlayer{
        State state;

        public void setState(State state){
            this.state = state;
        }

        public void runMusic(){
            this.state.runMusic();
        }
}

interface State{
        public void runMusic();
}

class Bass: State{
        public void runMusic(){
            Console.WriteLine("Running in Bass Mode");
        }
}

class Low: State{
        public void runMusic(){
            Console.WriteLine("Running in Low Mode");
        }
}
```

- **Strategy Method**

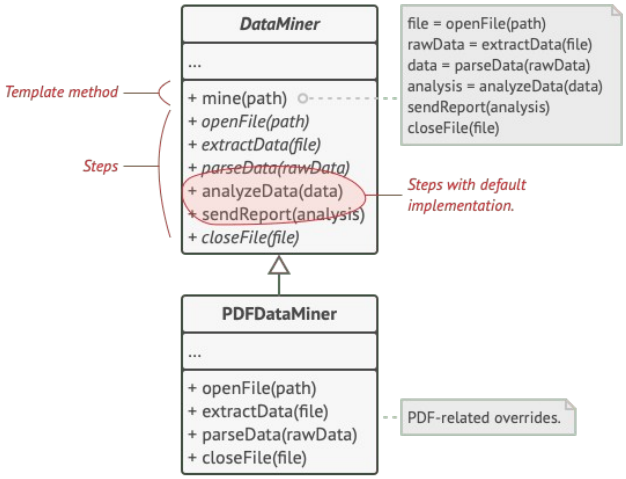
The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.




```
class Navigator{
        Method method;

        public void setMethod(Method method){
            this.method = method;
        }

        public void shortestWayTo(string place){
            method.shortestWayTo(place);
        }
}

interface Method{
        public void shortestWayTo(string place);
}

class PublicTransport: Method{
        public void shortestWayTo(string place){
            Console.WriteLine($"This is a shortes way to {place} with Public Transports");
        }
}

class Walking: Method{
        public void shortestWayTo(string place){
            Console.WriteLine($"This is a shortes way to {place} with Walking");
        }
}

class Car: Method{
        public void shortestWayTo(string place){
            Console.WriteLine($"This is a shortes way to {place} with Car");
        }
}
```

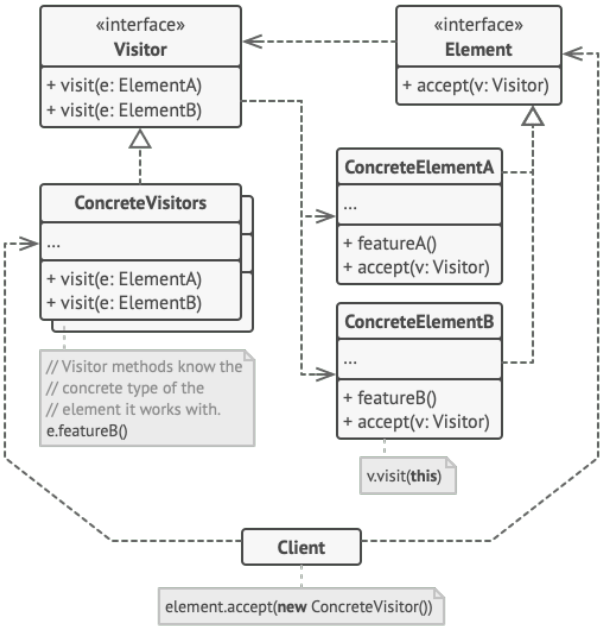
- **Template Method**

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method.* The steps may either be abstract, or have some default implementation.




```
abstract class PlayGame{
    public abstract void chooseGame();

    public virtual void openApp(){
          Console.WriteLine("Steam");
    }

    public virtual void playWith(){
          Console.WriteLine("Keyboard and Mouse");
    }

    public void play(){
        chooseGame();
        openApp();
        playWith();
    }
}

class PlayRE : PlayGame {
    public override void chooseGame(){
        Console.WriteLine("Resident Evil");
    }

    public override void openApp(){
        Console.WriteLine("Epic Games");
    }
}
```

- **Visitor Pattern:** Yeni özellik eklerken tüm classlara eklemek yerine visitor class' ına ekle.




```
abstract class Company{
        public string companyName;

      public void companyStuff(){
            Console.WriteLine(companyName + " stuff");
      }

      public void accept(Visitor v);
}

class CompanyA: Company{
        public void accept(Visitor v){
            v.visitCompanyA(this);
        }
}

class CompanyB: Company{
        public void accept(Visitor v){
            v.visitCompanyB(this);
        }
}

interface Visitor{
        public void visitCompanyA(CompanyA a);
      public void visitCompanyB(CompanyB b);
}

class EmailSenderVisitor: Visitor{
        public void visitCompanyA(CompanyA a){
            Console.WriteLine("Send Email to " + a.companyName);
        }

      public void visitCompanyB(CompanyB b){
            Console.WriteLine("Send Email to " + b.companyName);
      }
}
```

REFACTORING

Bloaters

- **Long Method:** A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- **Large Class:** A class contains many fields/methods/lines of code.
  When a class is wearing too many (functional) hats, think about splitting it up.
- **Primitive Obsession:**
  Use of primitives instead of small objects for simple tasks
  Use of constants for coding information
  Use of string constants as field names for use in data arrays
  If you have a large variety of primitive fields, it may be possible to logically group some of them into their own class. Even better, move the behavior associated with this data into the class too.
- **Long Parameter List:** More than three or four parameters for a method.
  if these parameters are coming from different sources, you can pass them as a single parameter object
  If some of the arguments are just results of method calls of another object replace parameter with method call.
- **Data Clumps:** Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.
  If repeating data comprises the fields of a class, use Extract Class to move the fields to their own class.
  If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields.

Object Orientation Abusers

- **Switch Statements:** You have a complex switch operator or sequence of if statements.
- **Temporary Fields:** Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty. Temporary fields and all code operating on them can be put in a separate class via Extract Class.
- **Refused Bequest:** If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter.

- **Alternative Classes with Different Interfaces:** Two classes perform identical functions but have different method names. Rename Methods to make them identical in all alternative classes.

Change Preventers

- **Divergent Change:** Divergent Change is when many changes are made to a single class.
  Split up the behavior of the class via Extract Class
  If different classes have the same behavior, you may want to combine the classes through inheritance
- **Shotgun Surgery:** when a single change is made to multiple classes simultaneously
  If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via Inline Class.
  move existing class behaviors into a single class. If there's no class appropriate for this, create a new one.
- **Parallel Inheritance Hierarchies:** Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.

Dispensables

- **Comments:** A method is filled with explanatory comments.
- **Duplicate Code:** Two code fragments look almost identical. Reference the method instead of rewriting.
- **Lazy Class:** if a class doesn't do enough to earn your attention, it should be deleted.
- **Data Class:** A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.
  Review the client code that uses the class. In it, you may find functionality that would be better located in the data class itself.

- **Dead Code:** A variable, parameter, field, method or class is no longer used (usually because it's obsolete).
  Delete unused code and unneeded files.
- **Speculative Generality:** There's an unused class, method, field or parameter. Remove them.

Couplers

- **Feature Envy:** A method accesses the data of another object more than its own data.
  This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.
- **Inappropriate Intimacy:** One class uses the internal fields and methods of another class.
  Keep a close eye on classes that spend too much time together. Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.
- **Message Chains:** In code you see a series of calls resembling $a→b()→c()→d()
  A message chain occurs when a client requests another object, that object requests yet another one, and so on.
  Instead of calling like this call method b, c, d inside a.
- **Middle Man:** If a class performs only one action, delegating work to another class, why does it exist at all.
  If most of a method's classes delegate to another class, Remove Middle Man.

Incomplete Library Class: The author of the library hasn't provided the features you need or has refused to implement them. Introduce a few methods to a library class, For big changes introduce local extension.