# Software Design Principles Notes

Contents

[Software Design Principles Notes 1](#_Toc184166283)

[Aim of software design patterns: 3](#_Toc184166284)

[State Design Pattern: 9](#_Toc184166285)

[Key Participants: 10](#_Toc184166286)

[**State:** 10](#_Toc184166287)

[**Concrete State:** 10](#_Toc184166288)

[**Context:** 10](#_Toc184166289)

[Strategy Pattern: 13](#_Toc184166290)

[Key Participants: 14](#_Toc184166291)

[**Strategy:** 14](#_Toc184166292)

[**Concept Strategy:** 14](#_Toc184166293)

[**Context:** 14](#_Toc184166294)

[Implementation with code: 15](#_Toc184166295)

[Observer Pattern: 17](#_Toc184166296)

[Key participants: 17](#_Toc184166297)

[**Subject** 17](#_Toc184166298)

[**Observer** 17](#_Toc184166299)

[**ConcreteSubject** 17](#_Toc184166300)

[**ConcreteObserver** 17](#_Toc184166301)

[Implementation w/ code: 18](#_Toc184166302)

[**Subject:** 18](#_Toc184166303)

[**Observer:** 18](#_Toc184166304)

[**CurrCondDisplay (Concrete Observer)** 18](#_Toc184166305)

[**Weather Data (Concrete Subject)** 19](#_Toc184166306)

[**Implementation for main programme:** 20](#_Toc184166307)

[Iterator Design Pattern: 22](#_Toc184166308)

[Key participants: 22](#_Toc184166309)

[**Iterator:** 22](#_Toc184166310)

[**Aggregate:** 22](#_Toc184166311)

[**Concreate Iterator:** 22](#_Toc184166312)

[**ConcreteAggregate:** 23](#_Toc184166313)

[Implementation w/ Code: 23](#_Toc184166314)

[**Iterator:** 23](#_Toc184166315)

[Interface: 24](#_Toc184166316)

[**Example Usage:** 24](#_Toc184166317)

[Decorator Pattern: 25](#_Toc184166318)

[Key Participants: 25](#_Toc184166319)

[**Component:** 25](#_Toc184166320)

[**ConcreteComponent:** 25](#_Toc184166321)

[**Decorator:** 25](#_Toc184166322)

[**ConcreteDecorator:** 26](#_Toc184166323)

[Implementation with Code: 26](#_Toc184166324)

[**Component:** 26](#_Toc184166325)

[**ConcreteComponent:** 26](#_Toc184166326)

[Decorator: 26](#_Toc184166327)

[**ConcreteDecorator:** 27](#_Toc184166328)

[Factory Method Pattern 28](#_Toc184166329)

[Key participants 28](#_Toc184166330)

[**Product** 28](#_Toc184166331)

[**ConcreteProduct** 28](#_Toc184166332)

[**Creator** 28](#_Toc184166333)

[**ConcreteCreator** 28](#_Toc184166334)

[Implementation w/ Code: 29](#_Toc184166335)

[**Product:** 29](#_Toc184166336)

[**ConcreteProduct:** 30](#_Toc184166337)

[**Creator:** 30](#_Toc184166338)

[**ConcreteCreator:** 31](#_Toc184166339)

[Abstract Factory Design Pattern: 33](#_Toc184166340)

[Key participants: 33](#_Toc184166341)

[**AbstractFactory** 33](#_Toc184166342)

[**ConcreteFactory** 33](#_Toc184166343)

[**AbstractProduct** 33](#_Toc184166344)

[- Declares an interface for a type of product object 33](#_Toc184166345)

[**ConcreteProduct** 33](#_Toc184166346)

[**Abstract Factory:** 34](#_Toc184166347)

[**ConcreteFactory:** 34](#_Toc184166348)

[**Abstract Product:** 35](#_Toc184166349)

[**ConcreteProduct:** 35](#_Toc184166350)

[Singleton Design Pattern: 36](#_Toc184166351)

[Command Design Pattern: 37](#_Toc184166352)

[Key participants: 38](#_Toc184166353)

## Aim of software design patterns:

1. **Buildable** - can be implemented without error
2. **Maintainable** - can be updated relatively easily
3. **Extendable** - can add new features without affecting existing features unnecessarily
4. **Reusable** - can be ported over to other similar projects

**Cohesion:**

- Describes how much a component contributes to a **single** purpose

- Low cohesion -> more complex, changes are harder to implement, therefore harder to reuse

Hence we want **high** cohesion.

1. Operation cohesion
2. Class Cohesion
3. Specialisation Cohesion

**Coupling:**

- Describes the amount of **interconnections** between components

If components have high coupling:

1. Changes in one component likely to affect the other
2. Implementing af unctionaility may be more difficult as multiple components are involved
3. Harder to test the system, due to issues arising from multiple components
4. Inheritance Coupling  
   - Try to move attributes to sub classes if only required by one subclass, reduces the need for unnecessary inheritance of attributes
5. Interaction Coupling  
   - Try to have as few parameters as possible between messages in a sequence diagram

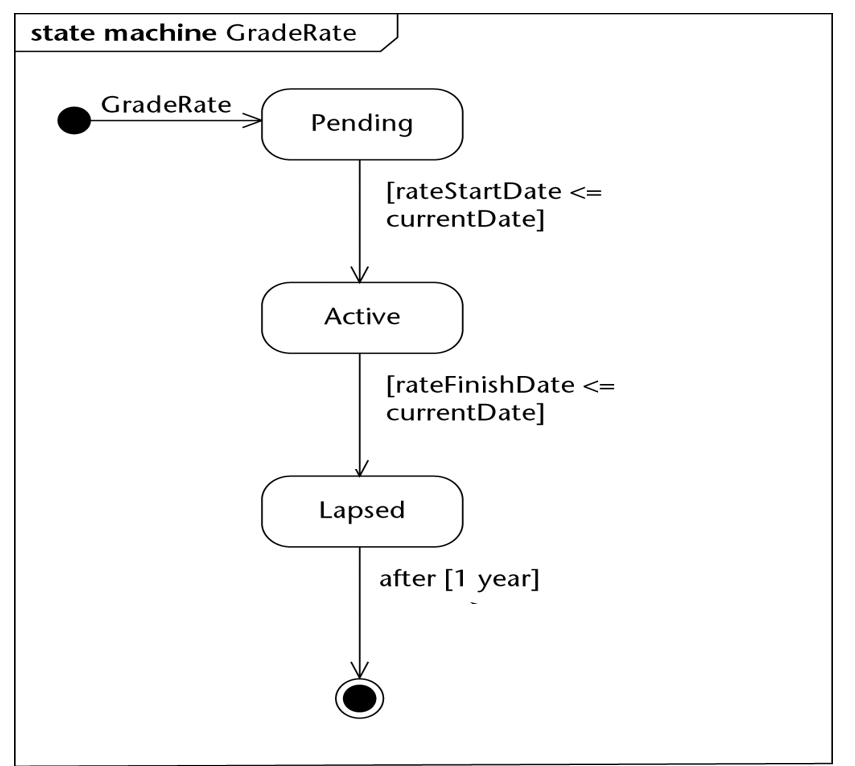
**State:**

- Current condition of an object

- Determined by:

1. Current value of the object’s **attributes**

2) **Links** it has with other objects



Types of trigger:

1. Change trigger

- Occurs when condition becomes true

1. Call trigger

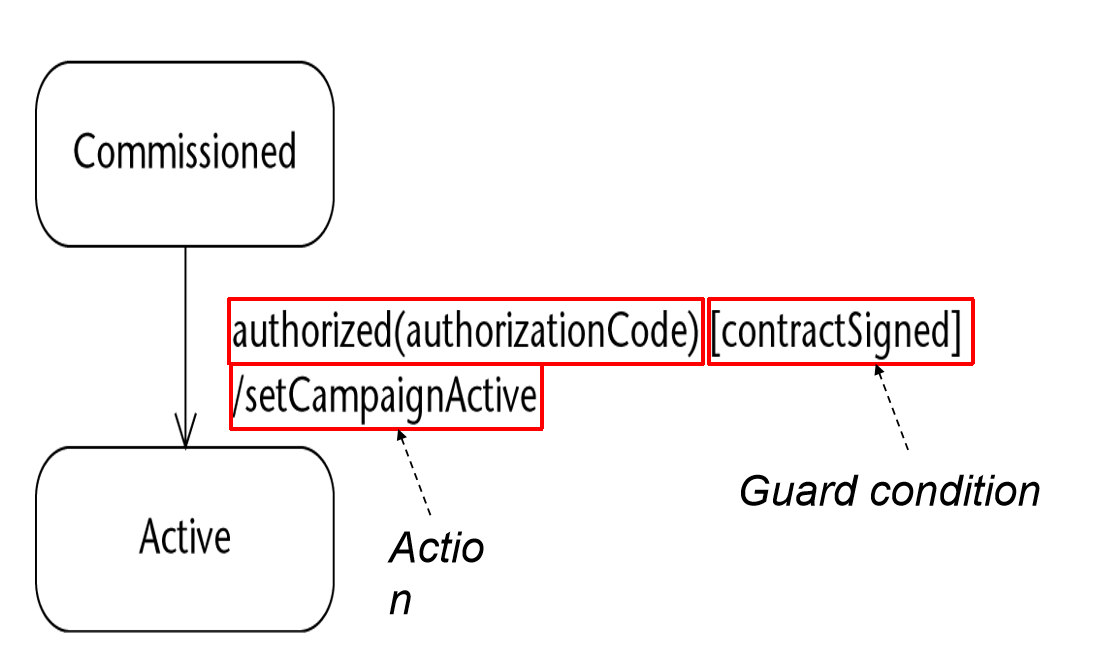
- Occurs when an object recieves a call of one of its operations either from another object or from itself

1. Relative-Time trigger

- Caused by the passage of a designated period of time after a specified event (usually entry to the current state)

1. Signal Trigger

- Occurs when an object receives a signal (async communication)

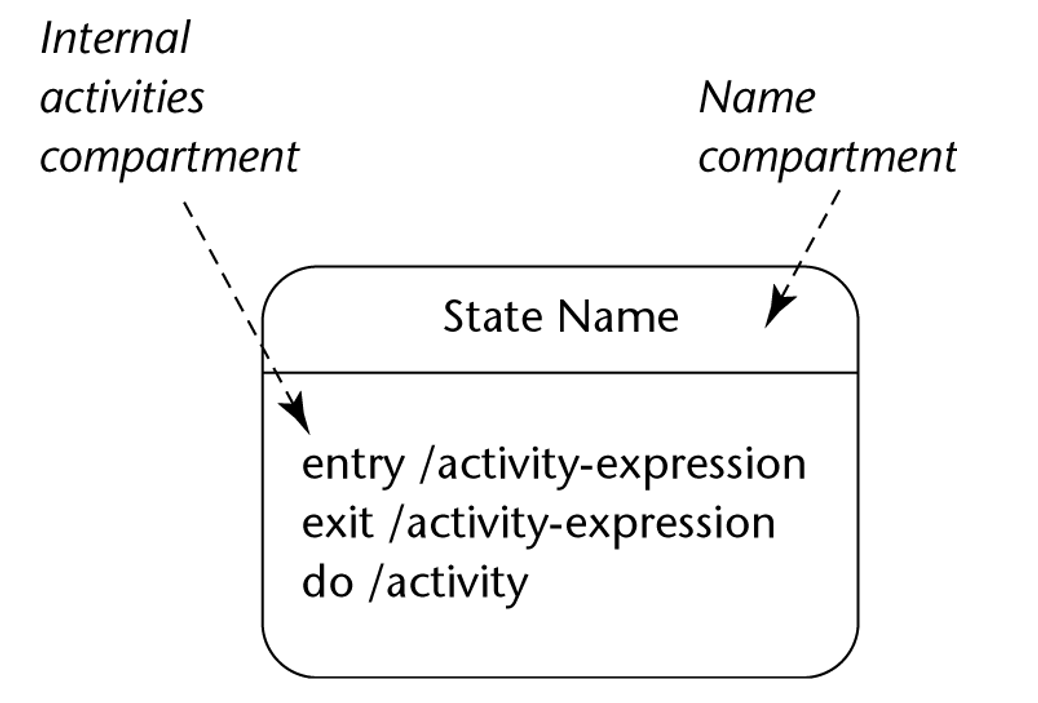


**Transitions:**  
- Transition is fired when following conditions are satisfied:

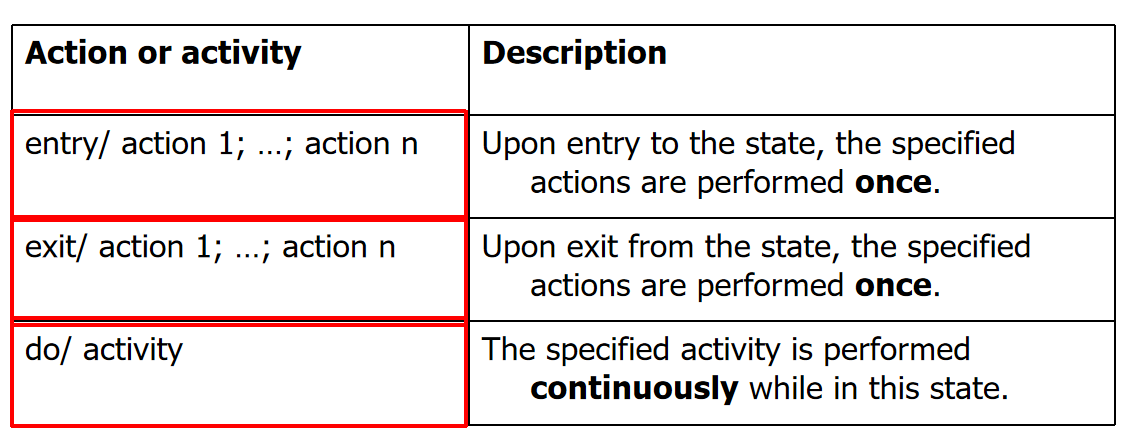
* Entity is in the state of the source state
* Event specified in label occurs
* Guard condition specified in the label is evaluated to be true

When a transition is fired, actions associated with it are executed

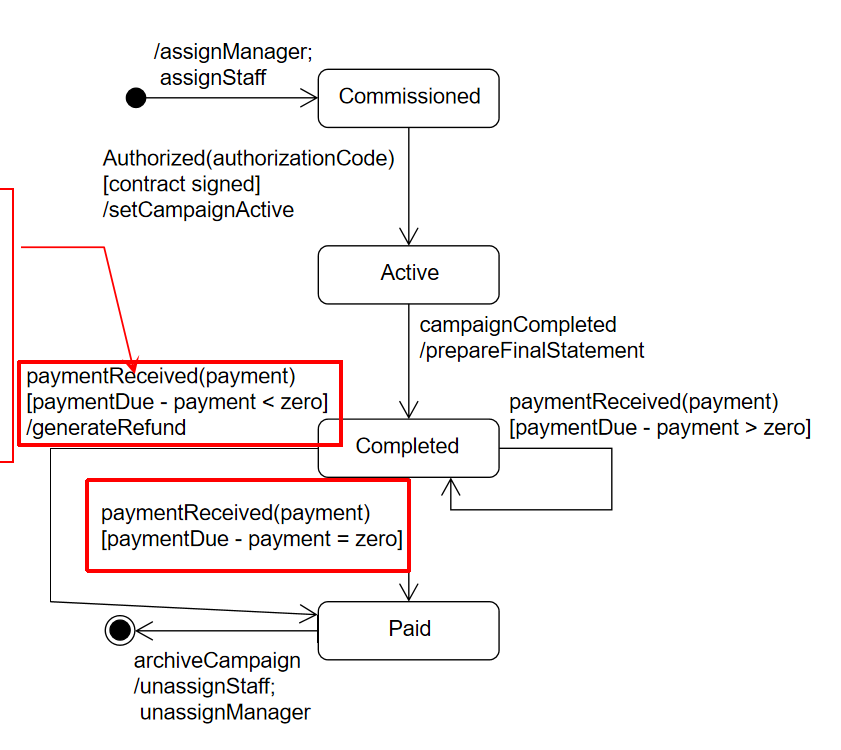
**Internal Activites:**



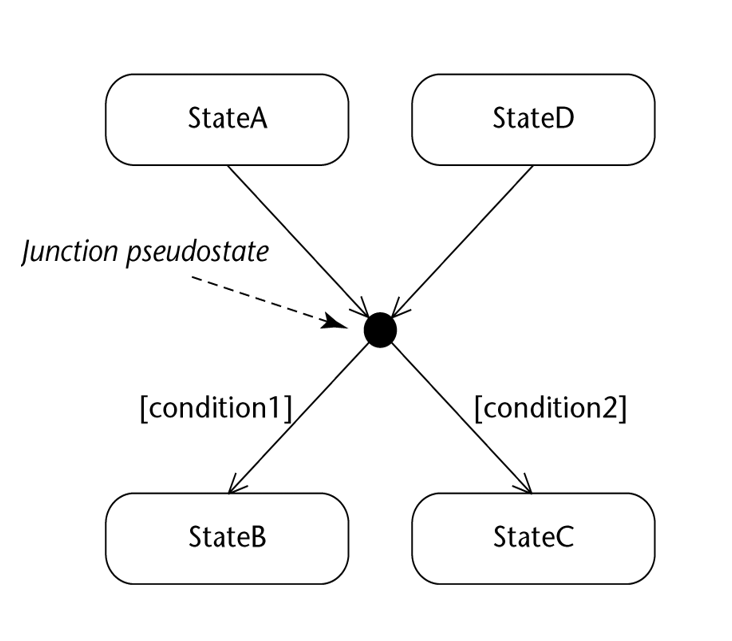
**UML Notation:**



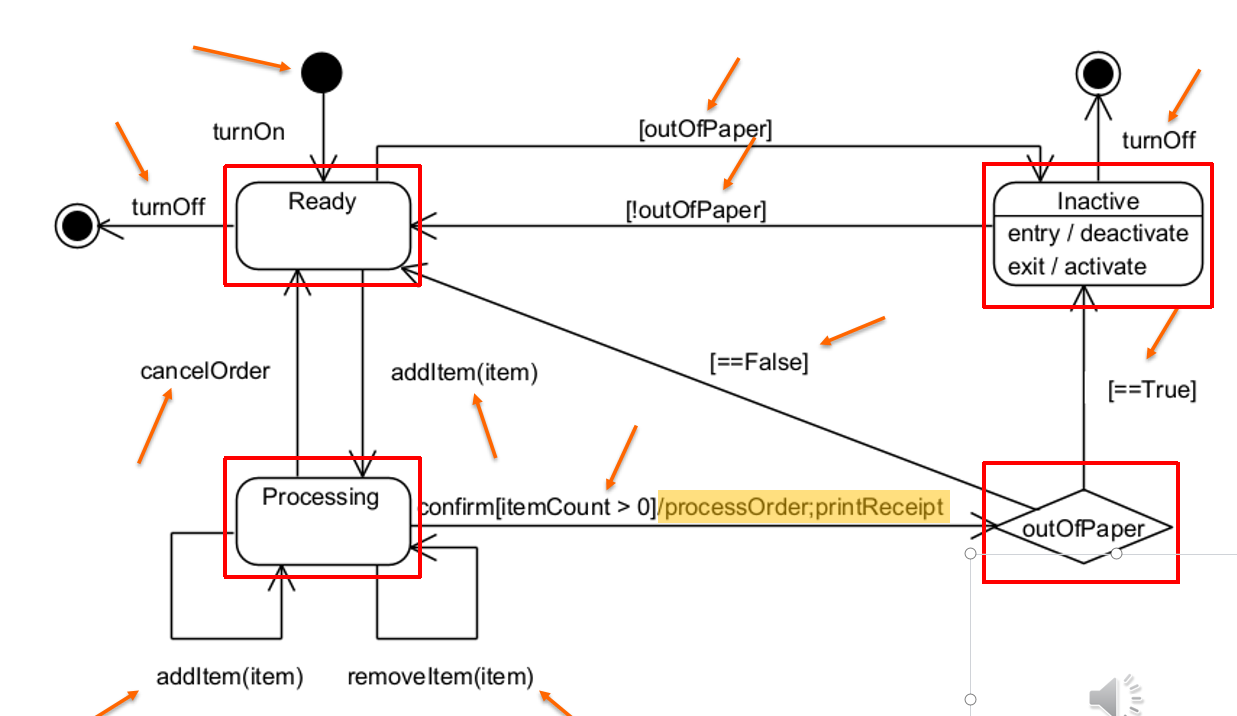
**Example:**



**Junction and Choice Psuedostates:**



- UML says that object can end up in **any** resulting state if there is no priority, would result in ill-formed state machine

**Full State Machine Example:**

# State Design Pattern:

Allows an object to alter its behaviour when its internal state changes

The object will appear to change its class (a.k.a objects for states)

Applicability:

* Object’s behaviour depends on its state, must change it’s behaviour at runtime depending on that state:
  + Operations have large, multi-part conditional statements that depend on that state
  + Same conditional structure over many operations
  + State design pattern puts each branch in the conditional as a separate class -> each state is an object, can vary independentlyA computer screen shot of a diagram

    Description automatically generated

## Key Participants:

### **State:**

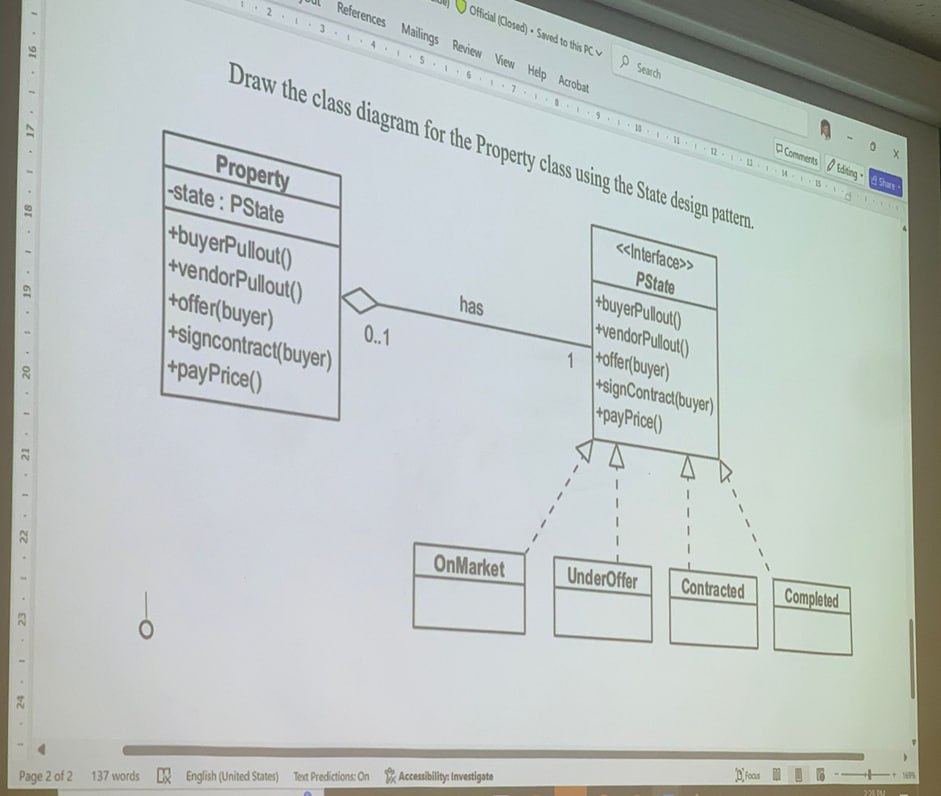
Defines an interface to encapsulate the behaviour of all states in the context

### **Concrete State:**

Implements the behaviour using the state interface

### **Context:**

Configures with state objects, maintains an instance to a ConcreteState object that represents its current state



  public class Property

    {

        private PState \_state;

        private bool \_success = false;

        public Property()

        {

            \_state = new OnMarket(); // Initial state

        }

        public void SetState(PState state)

        {

            \_state = state;

        }

        public bool IsSaleSuccessful()

        {

            return \_success;

        }

        public void MarkSaleSuccess()

        {

            \_success = true;

        }

        public void BuyerPullout()

        {

            \_state?.BuyerPullout(this);

        }

        public void VendorPullout()

        {

            if (\_success)

            {

                Console.WriteLine("Vendor cannot pull out - sale is complete with success = True");

            }

            else

            {

                \_state?.VendorPullout(this);

            }

        }

        public void Offer(Buyer buyer)

        {

            if (\_success)

            {

                Console.WriteLine("Cannot offer - sale is complete with success = False");

            }

            else

            {

                \_state?.Offer(this, buyer);

            }

        }

        public void SignContract(Buyer buyer)

        {

            \_state?.SignContract(this, buyer);

        }

        public void PayPrice()

        {

            \_state?.PayPrice(this);

        }

    }

-------

    public interface PState

    {

        void BuyerPullout(Property property);

        void VendorPullout(Property property);

        void Offer(Property property, Buyer buyer);

        void SignContract(Property property, Buyer buyer);

        void PayPrice(Property property);

    }

-------------

    public class OnMarket : PState

    {

        public void BuyerPullout(Property property)

        {

            Console.WriteLine("No buyer to pull out. The property is on the market.");

        }

        public void VendorPullout(Property property)

        {

            Console.WriteLine("Vendor pulls out. Sale is complete with success = False.");

            property.SetState(new Completed());

        }

        public void Offer(Property property, Buyer buyer)

        {

            Console.WriteLine("Offer made.");

            property.SetState(new UnderOffer());

        }

        public void SignContract(Property property, Buyer buyer)

        {

            Console.WriteLine("No buyer to sign contract.");

        }

        public void PayPrice(Property property)

        {

            Console.WriteLine("Contract not yet signed.");

        }

    }

# Strategy Pattern:

* Defines a family of algorithms, encapsulates each one and makes them interchangeable
* Algorithm may vary independently from the clients that use it
* Known as policy design

Applicability:  
- Sometimes there are many ways to achieve a task (such as payment)

* “Correct” way is dependent on circumstances
* Wish to change strategy used during runtime, or want to keep implementation details secret

A diagram of a strategy

Description automatically generated

## Key Participants:

### **Strategy:**

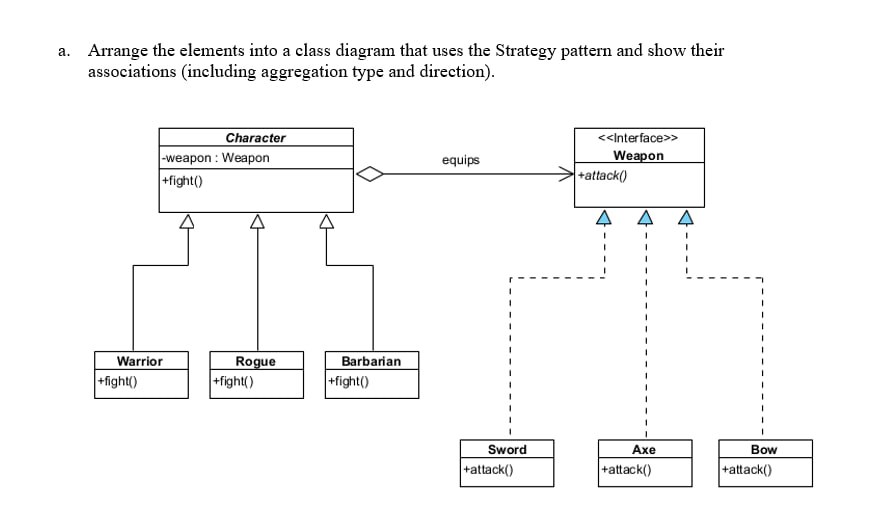
* Defines an interface common to all supported algorithms

### **Concept Strategy:**

* Implements the algorithm using Strategy Interface

### **Context:**

* Maintains a reference to a strategy object
* Configured with a ConcreteStrategy object
* May define an interface that lets Strategy access its data

**Example:**  


## Implementation with code:

 public abstract class Character

    {

        private Weapon \_weapon;

        // Empty constructor

        protected Character()

        {

        }

        // Property for accessing and changing the weapon

        public Weapon MyWeapon

        {

            get { return Weapon; }

            set { Weapon = value; }

        }

        public Weapon Weapon { get => \_weapon; set => \_weapon = value; }

        // Abstract method to be implemented by subclasses

        public abstract void Fight();

    }

 // Warrior.cs

    public class Warrior : Character

    {

        public Warrior()

        { }

        public override void Fight()

        {

            MyWeapon?.Attack(); // Call Attack if MyWeapon is not null

        }

    }

----------------------

# Observer Pattern:

Defines a one-to-many dependency between objects, so when one object changes state, all its dependents are notified and updated immediately

A diagram of a computer

Description automatically generated

## Key participants:

### **Subject**

* Provides interface to register/remove/notify observers

### **Observer**

* Defines interface for update notification

### **ConcreteSubject**

* Object being observed
* Sends notification to observers when state is changed

### **ConcreteObserver**

* Observing object
* Implements observer interface update() method

Example:  
A diagram of a server

Description automatically generated

## Implementation w/ code:

### **Subject:**

// Subject interface with methods to register, remove, and notify observers

interface Subject

{

    void RegisterObserver(Observer o); // Register an observer

    void RemoveObserver(Observer o);   // Remove an observer

    void NotifyObservers();            // Notify all observers of a change

}

### **Observer:**

// Observer interface with an update method to receive state changes

interface Observer

{

    void Update(float temp, float humidity, float pressure); // Update the observer with new data

}

### **CurrCondDisplay (Concrete Observer)**

// Concrete observer implementation that displays current conditions

class CurrentConditionsDisplay : Observer

{

    private float temperature; // Store the temperature for display

    private float humidity;    // Store the humidity for display

    private float pressure;    // Store the pressure for display

    private Subject weatherData; // Reference to the Subject to allow deregistration if needed

    // Constructor registers itself as an observer of the provided Subject

    public CurrentConditionsDisplay(Subject weatherData)

    {

        this.weatherData = weatherData;

        weatherData.RegisterObserver(this);

    }

    // Update method called by Subject with new data

    public void Update(float temperature, float humidity, float pressure)

    {

        this.temperature = temperature;

        this.humidity = humidity;

        this.pressure = pressure;

        Display(); // Display updated data

    }

    // Display method to print the current conditions, including pressure

    public void Display()

    {

        Console.WriteLine("Current conditions: " + temperature + "F degrees, "

                          + humidity + "% humidity, and "

                          + pressure + " pressure");

    }

}

### **Weather Data (Concrete Subject)**

// Concrete implementation of the Subject interface

class WeatherData : Subject

{

    private List<Observer> observers; // List to hold observers that are watching this subject

    private float temperature;        // Current temperature data

    private float humidity;           // Current humidity data

    private float pressure;           // Current pressure data

    // Constructor initializes the list of observers

    public WeatherData()

    {

        observers = new List<Observer>();

    }

    // Register an observer by adding it to the observers list

    public void RegisterObserver(Observer o)

    {

        observers.Add(o);

    }

    // Remove an observer from the observers list

    public void RemoveObserver(Observer o)

    {

        observers.Remove(o);

    }

    // Notify all registered observers by calling their update method

    public void NotifyObservers()

    {

        foreach (Observer observer in observers)

        {

            observer.Update(temperature, humidity, pressure);

        }

    }

    // This method is called when the measurements change

    public void MeasurementsChanged()

    {

        NotifyObservers(); // Notify observers about the change

    }

    // Method to simulate new weather data and notify observers

    public void SetMeasurements(float temperature, float humidity, float pressure)

    {

        this.temperature = temperature;

        this.humidity = humidity;

        this.pressure = pressure;

        MeasurementsChanged(); // Call MeasurementsChanged to update observers

    }

}

### **Implementation for main programme:**

// Test program to demonstrate the Observer pattern

class WeatherStation

{

    static void Main(string[] args)

    {

        WeatherData weatherData = new WeatherData(); // Create the Subject

        // Create an observer and register it to receive updates from WeatherData

        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);

        // Simulate new weather measurements

        weatherData.SetMeasurements(80, 65, 30.4f);

        weatherData.SetMeasurements(82, 70, 29.2f);

        weatherData.SetMeasurements(78, 90, 29.2f);

    }

}

# Iterator Design Pattern:

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation

* Loose coupling between client and aggregate/iterator classes

A diagram of a software

Description automatically generated

## Key participants:

### **Iterator:**

* Defines the interface for accessing and traversing elements
  + Next() advances to the next element and returns it as an Object
  + hasNext() returns true if there is another element in the aggregate, false otherwise

### **Aggregate:**

* Defines the interface for creating an iterator object

### **Concreate Iterator:**

* Implements the Iterator interface
* Keeps track of current position

### **ConcreteAggregate:**

* Implements the iterator creation interface to return an instance of ConcreteIterator

**A diagram of a program

Description automatically generated**

## Implementation w/ Code:

### **Iterator:**

class OddIterator : Iterator {

    List<int> values;

    int position = 0;

public OddIterator(List<int> values) {

    this.values = values;

    // Move position to first odd number, skip even numbers

    while ((position < values.Count) && (values[position] % 2 != 0)) {

        position++;

    }

}

public bool hasNext() {

    return position < values.Count;

}

public object next() {

    int value = values[position];

    position++;

    // Move position to next odd number, skip even numbers

    while ((position < values.Count) && (values[position] % 2 != 1)) {

        position++;

    }

    return value;

}

}

### Interface:

interface Iterator {

    public bool hasNext();

    public object next();

}

### **Example Usage:**

// Example usage

List<int> list = new List<int>();

list.Add(1);

list.Add(2);

list.Add(3);

// Create an iterator for odd numbers

OddIterator oddIterator = new OddIterator(list);

while (oddIterator.hasNext()) {

    Console.WriteLine((int)oddIterator.next());

}

# Decorator Pattern:

* Attaches additional responsibilities to an object dynamically
* Provides a flexible alternative to subclassing for extending functionality

A diagram of a decorator

Description automatically generated

## Key Participants:

### **Component:**

* Defines the interface for objects that can have responsibilities added to them dynamically

### **ConcreteComponent:**

* Defines an object to which additional responsibilities can be attached

### **Decorator:**

* Maintains a reference to a Component object
* Defines an interface that conforms to the Component

### **ConcreteDecorator:**

* Adds responsibilities to the component

## Implementation with Code:

### **Component:**

public abstract class Beverage {

    protected string description = "Unknown Beverage”;

    public virtual string getDescription() {

      return description;

    }

    public abstract double cost();

}

### **ConcreteComponent:**

public class HouseBlend: Beverage

{

    public HouseBlend() {

      base.description = "House Blend";

    }

    public override double cost() { return 4.50; }

}

public class DarkRoast: Beverage

{

    public DarkRoast() {

      base.description = "Dark Roast";

    }

    public override double cost() { return 4.00; }

}

### Decorator:

public abstract class CondimentDecorator: Beverage {

    public abstract override string getDescription();

}

### **ConcreteDecorator:**

public class Mocha: CondimentDecorator {

    private Beverage beverage;

    public Mocha(Beverage beverage) {

        this.beverage = beverage;

    }

    public override string getDescription() {

        return beverage.getDescription() + ", mocha";

    }

    public override double cost() {

        return 0.80 + beverage.cost();

    }

}

# Factory Method Pattern

* Defines an interface for creating an object, but lets the subclasses decide which class to instantiate
* Lets a class defer instantiation into subclasses

A diagram of a product

Description automatically generated

## Key participants

### **Product**

* Defines interface for objects that the factory method creates

### **ConcreteProduct**

* Implements the Product interface

### **Creator**

* Declares the factory method, which returns an object of type Product
* May also define a default implementation of factory method that returns a default ConcreteProduct

### **ConcreteCreator**

* Overrides the factory method to return ConcreteProduct

Example:

A screenshot of a computer

Description automatically generated

## Implementation w/ Code:

### **Product:**

abstract class Pizza

{

    public void Prepare()

    {

        Console.WriteLine($"Preparing {this.GetType().Name}");

    }

    public void Bake()

    {

        Console.WriteLine($"Baking {this.GetType().Name}");

    }

    public virtual void Cut()

    {

        Console.WriteLine($"Cutting {this.GetType().Name} in standard way");

    }

    public void Box()

    {

        Console.WriteLine($"Boxing {this.GetType().Name}");

    }

}

### **ConcreteProduct:**

class NYStyleCheesePizza : Pizza

{

    public override void Cut()

    {

        Console.WriteLine($"Cutting {this.GetType().Name} into traditional New York slices");

    }

}

class NYStyleHawaiianPizza : Pizza

{

    // Uses default Cut() implementation

}

class ChicagoStyleCheesePizza : Pizza

{

    public override void Cut()

    {

        Console.WriteLine($"Cutting {this.GetType().Name} into square slices (Chicago Style)");

    }

}

class ChicagoStyleHawaiianPizza : Pizza

{

    // Uses default Cut() implementation

}

### **Creator:**

abstract class PizzaStore

{

    // Factory method to create a pizza, implemented by subclasses

    protected abstract Pizza CreatePizza(string type);

    // Method to order a pizza

    public Pizza OrderPizza(string type)

    {

        Pizza pizza = CreatePizza(type);

        if (pizza != null)

        {

            pizza.Prepare();

            pizza.Bake();

            pizza.Cut();

            pizza.Box();

        }

        else

        {

            Console.WriteLine("Sorry, we don't have that type of pizza.");

        }

        return pizza;

    }

}

### **ConcreteCreator:**

class NYStylePizzaStore : PizzaStore

{

    protected override Pizza CreatePizza(string type)

    {

        if (type == "cheese")

        {

            return new NYStyleCheesePizza();

        }

        else if (type == "hawaiian")

        {

            return new NYStyleHawaiianPizza();

        }

        else

        {

            return null;

        }

    }

}

class ChicagoStylePizzaStore : PizzaStore

{

    protected override Pizza CreatePizza(string type)

    {

        if (type == "cheese")

        {

            return new ChicagoStyleCheesePizza();

        }

        else if (type == "hawaiian")

        {

            return new ChicagoStyleHawaiianPizza();

        }

        else

        {

            return null;

        }

    }

}

# Abstract Factory Design Pattern:

* Provides an interface for creating families of related or dependent objects without specifying their concrete classes

A screenshot of a computer

Description automatically generated

## Key participants:

### **AbstractFactory**

* Declares an interface for operations that create abstract product objects

### **ConcreteFactory**

* Implements the operations to create concrete product objects

### **AbstractProduct**

### Declares an interface for a type of product object

### **ConcreteProduct**

* Defines product object to be created by the corresponding concrete factory
* Implements the AbstractProduct interface

Example:  
A screenshot of a computer

Description automatically generated

**Implementation with Code:**

**Abstract Factory:**

<Interface>

public interface IPizzaIngredientFactory

{

    IDough CreateDough();

    ITopping CreateTopping();

}

**ConcreteFactory:**

public class NYPizzaIngredientFactory : IPizzaIngredientFactory

{

    public IDough CreateDough()

    {

        return new ThinCrustDough();

    }    public ITopping CreateTopping()

    {

        return new Pepperoni();

    }

}

public class ChicagoPizzaIngredientFactory : IPizzaIngredientFactory

{

    public IDough CreateDough()

    {

        return new ThickCrustDough();

    }    public ITopping CreateTopping()

    {

        return new Beef();

    }

}

**Abstract Product:**

<Interface>

// Abstract Products

public interface IDough { }

public interface ITopping { }

**ConcreteProduct:**

public class ThinCrustDough : IDough { }

public class ThickCrustDough : IDough { }

public abstract class Cheese : ITopping { }

public class ReggianoCheese : Cheese{ }

public class MozzarellaCheese : Cheese{ }

public class Pepperoni : ITopping { }

public class Beef : ITopping { }

**Client:**

**Differences between Factory Method and Abstract Factory**

* Factory Method
  + Uses subclasses that override a factory method
  + Creates a single object
* Abstract Factory
  + Uses object composition
  + Creates a family of objects

# Singleton Design Pattern:

* Ensures that only one instance of the class is created
* Provides a global point of access to the single instance
* Declares a private constructor

A diagram of a computer

Description automatically generated

Use Singleton when there must be at most one instance of a class

* Instance must be accessible to clients from a well-known access point

One instance is extensible by subclassing

* Clients can use the subclass without modifying code

**Player Instance:**

public class Player

    {

        private static Player uniqueInstance = null;

        private readonly Command[] commands = new Command[12];

        private Command lastCommand;

        private int hp;

        private int maxHp;

        private int mp;

        private int maxMp;

        private int magicPower;

        // Private constructor for Singleton

        private Player(int maxHp, int maxMp, int magicPower)

        {

            this.maxHp = maxHp;

            this.maxMp = maxMp;

            this.magicPower = magicPower;

            this.hp = maxHp;

            this.mp = maxMp;

            for (int i = 0; i < commands.Length; i++)

            {

                commands[i] = new NoCommand();

            }

            lastCommand = new NoCommand();

        }

**Set Instance:**

     // Singleton instance retrieval

        public static Player getInstance(int maxHp, int maxMp, int magicPower)

        {

            if (uniqueInstance == null)

            {

                uniqueInstance = new Player(maxHp, maxMp, magicPower);

            }

            return uniqueInstance;

        }

# Command Design Pattern:

* Encapsulates a request as an object
* Let’s parameterise clients with different requests
* Supports undoable operations

**A diagram of a computer program

Description automatically generated**

## Key participants:

## Command:

Declares an interface for executing an operation

## Concrete Command:

Defines a binding between a Receiver object and an action

Implements execute() by invoking corresponding operations on the Receiver

## Client:

Creates a ConcreteCommand object and sets it receiver

## Invoker:

Asks the Command to carry out the request by calling execute()

## Receiver:

Knows how to perform the operations associated with carrying out a request

Any class may serve as a Receiver

Example:

A computer screen shot of a diagram

Description automatically generated

Implementation with Code:

**Invoker (Player):**

    public class Player

    {

        private static Player uniqueInstance = null;

        private readonly Command[] commands = new Command[12];

        private Command lastCommand;

 // Setter for commands

        public void setCommand(Command command, int slot)

        {

            commands[slot] = command;

        }

        // Execute a command

        public void hotkeyPushed(int slot)

        {

            commands[slot].execute();

            lastCommand = commands[slot];

        }

        // Undo the last executed command

        public void undoKeyPushed()

        {

            lastCommand.undo();

        }

**Command & ConcreteCommand:**

public interface Command

    {

        void execute();

        void undo();

    }

    public class NoCommand : Command

    {

        public void execute()

        {

            Console.WriteLine("No command set!");

        }

        public void undo()

        {

            Console.WriteLine("No command set!");

        }

    }

    public class DrinkHealingPotionCommand : Command

    {

        private readonly HealingPotion healingPotion;

        public DrinkHealingPotionCommand(HealingPotion healingPotion)

        {

            this.healingPotion = healingPotion;

        }

        public void execute()

        {

            int healed = healingPotion.drink();

            healingPotion.getPlayer().heal(healed);

            Console.WriteLine($"Player is healed for {healed} HP and now has {healingPotion.getPlayer().getHp()} HP");

        }

        public void undo()

        {

            int healed = healingPotion.drink();

            healingPotion.getPlayer().takeDamage(healed);

            healingPotion.incrementDoses();

            Console.WriteLine($"Undoing drinking healing potion for {healed} HP.");

        }

    }

# Composite Pattern:

* Allows you to compose objects into tree structures to represent part-whole hierarchies
* Let’s client treat individual objects and compositions of objects uniformly

A diagram of a component

Description automatically generated

## Key participants:

## Component:

Declares an interface for objects in the composition

Implements default behaviour where appropriate

Declares an interface for managing its child components

## Leaf:

Represents objects in the comparison with no children

## Composite:

Defines behaviour for components with children

Stores child components

## Client:

Manipulates objects in the composition

A diagram of a component

Description automatically generated

Implementation with Code:

**MenuComponent:**

public abstract class MenuComponent

{

    public virtual void Add(MenuComponent menuComponent)

    {

        throw new NotSupportedException();

    }

    public virtual void Remove(MenuComponent menuComponent)

    {

        throw new NotSupportedException();

    }

    public virtual MenuComponent GetChild(int index)

    {

        throw new NotSupportedException();

    }

    public virtual string Name

    {

        get { throw new NotSupportedException(); }

    }

    public virtual string Description

    {

        get { throw new NotSupportedException(); }

    }

    public virtual double Price

    {

        get { throw new NotSupportedException(); }

    }

    public virtual bool Vegetarian

    {

        get { throw new NotSupportedException(); }

    }

    public virtual void Print()

    {

        throw new NotSupportedException();

    }

    public virtual IEnumerator<MenuComponent> CreateIterator()

    {

        throw new NotSupportedException();

    }

}

**MenuItem:**

public class MenuItem : MenuComponent

{

    private string name;

    private string description;

    private bool vegetarian;

    private double price;

    public MenuItem(string name, string description, bool vegetarian, double price)

    {

        this.name = name;

        this.description = description;

        this.vegetarian = vegetarian;

        this.price = price;

    }

    public override string Name => name;

    public override string Description => description;

    public override double Price => price;

    public override bool Vegetarian => vegetarian;

    public override void Print()

    {

        Console.Write($"  {Name}");

        if (Vegetarian)

        {

            Console.Write(" (veg.)");

        }

        Console.WriteLine($": ${Price:N2}");

        Console.WriteLine($"  -- {Description}");

    }

    public override IEnumerator<MenuComponent> CreateIterator()

    {

        return new NullIterator();

    }

}

**DinerMenu:**

public class DinerMenu : MenuComponent

{

    private List<MenuComponent> components = new List<MenuComponent>();

    private string name;

    public DinerMenu(string name)

    {

        this.name = name;

    }

    public override string Name => name;

    public override void Add(MenuComponent menuComponent)

    {

        components.Add(menuComponent);

    }

    public override void Remove(MenuComponent menuComponent)

    {

        components.Remove(menuComponent);

    }

    public override MenuComponent GetChild(int index)

    {

        return components[index];

    }

    public override void Print()

    {

        Console.WriteLine($"\n{Name.ToUpper()}");

        Console.WriteLine(new string('-', Name.Length));

        foreach (var component in components)

        {

            component.Print();

        }

    }

    public override IEnumerator<MenuComponent> CreateIterator()

    {

        return components.GetEnumerator();

    }

}

**DinerMenuIterator:**

public class DinerMenuIterator: Iterator {

    private List<MenuComponent> myMenu;

    private int position = 0;

    public DinerMenuIterator(List<MenuComponent> menu)

    {

        myMenu = menu;

    }

    public bool hasNext() {

        return position < myMenu.Count; }

    public Object next() { return myMenu[position++]; }

}

**NullIterator:**

public class NullIterator : IEnumerator<MenuComponent>

{

    public MenuComponent Current => null;

    object System.Collections.IEnumerator.Current => Current;

    public bool MoveNext() => false;

    public void Reset() {}

    public void Dispose() {}

}