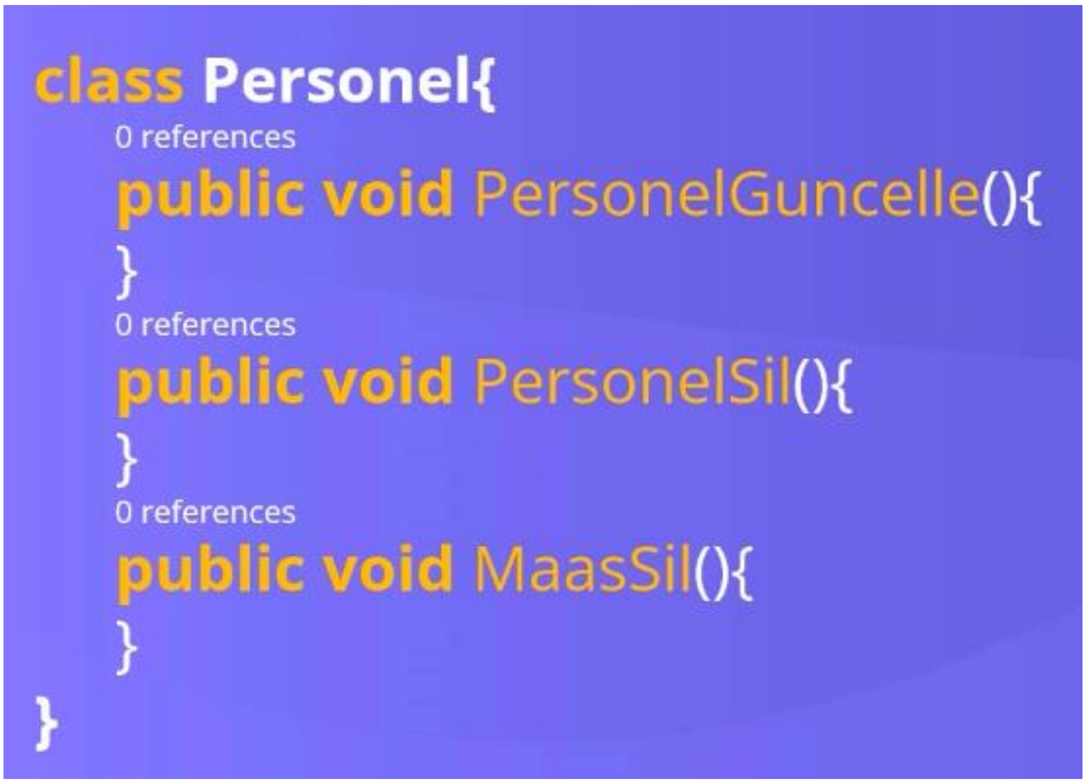
Solid Principles and Design Patterns

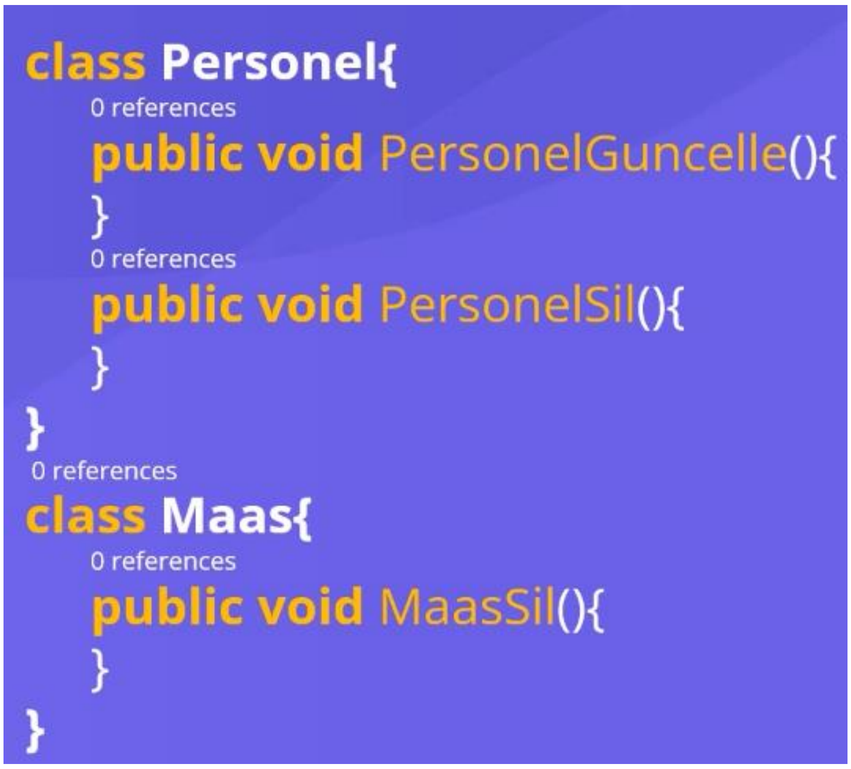
SOLID

- Created by Robert C Martin
- Built upon by Michael Feathers
- Create more modern and understandable software
- To allow developers to develop code and keep pace with better effort
- S – Single Responsibility
  - Every class should have responsibility over a single part of functionality provided by the software
  - Responsibility should be entirely encapsulated by the class
  - Class should contain only the class methods for each classes purpose
  - Without SRP



```
class Personel{
    0 references
    public void PersonelGuncelle(){
    }
    0 references
    public void PersonelSil(){
    }
    0 references
    public void MaasSil(){
    }
}
```

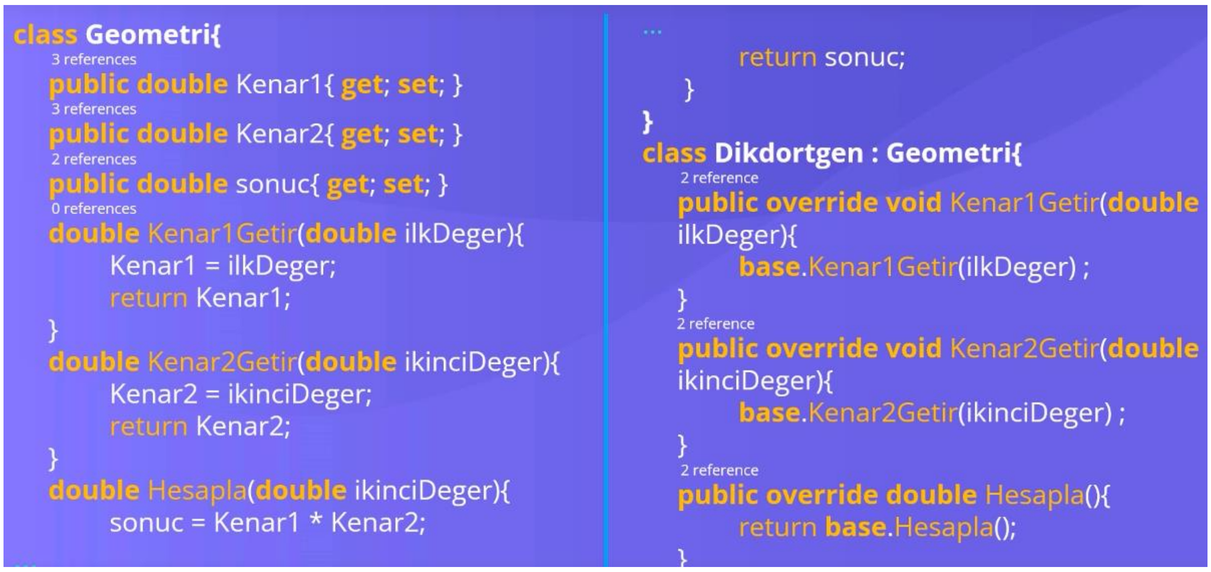
- With SRP



```
class Personel{
        0 references
        public void PersonelGuncelle(){
        }
        0 references
        public void PersonelSil(){
        }
}
0 references
class Maas{
        0 references
        public void MaasSil(){
        }
}
```

  - Example
- O – Open / Closed (OCP)
  - The softwares entities, classes or methods should be open for extension but closed for modification
  - Any code will change in the following time for new necessities
  - We have to protect all missions
  - Class should allow for its functionality to be extended, not allow any modifications to its own source code
  - Aims to reduce the introduction of bugs and other errors into your code by requiring classes not to change their own implementation unless absolutely necessary
  - Implement classes that can easily have their functionality to be extended
  - Allowing a class to be open for extensions, we allow for many real changes to occur without completely disrupting our design

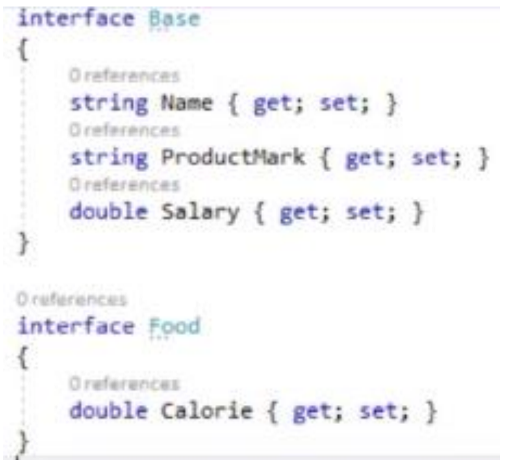
- L – Liskov Substitution (LSP)
  - Originally designed by Barbara Liskov
  - Should be able to treat a child class as though it was the parent class
  - All derived classes should retain their functionality of their parent class and cannot replace any functionality that the parent provides
  - To prevent block errors, need to override all methods like the following



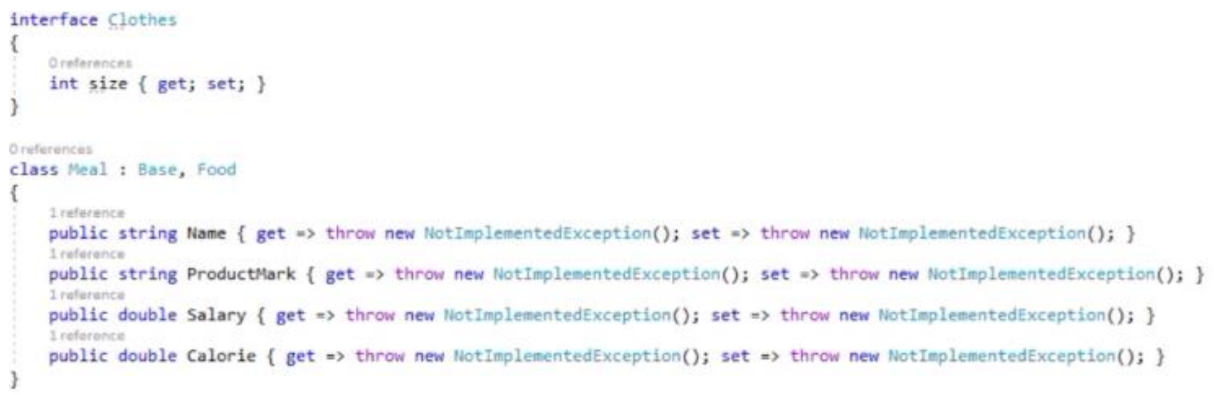
```
class Geometri{
    3 references
    public double Kenar1{ get; set; }
    3 references
    public double Kenar2{ get; set; }
    2 references
    public double sonuc{ get; set; }
    0 references
    double Kenar1Getir(double ilkDeger){
        Kenar1 = ilkDeger;
        return Kenar1;
    }
    double Kenar2Getir(double ikinciDeger){
        Kenar2 = ikinciDeger;
        return Kenar2;
    }
    double Hesapla(double ikinciDeger){
        sonuc = Kenar1 * Kenar2;
```

```
...
        return sonuc;
    }
}
class Dikdortgen : Geometri{
    2 reference
    public override void Kenar1Getir(double
    ilkDeger){
        base.Kenar1Getir(ilkDeger) ;
    }
    2 reference
    public override void Kenar2Getir(double
    ikinciDeger){
        base.Kenar2Getir(ikinciDeger) ;
    }
    2 reference
    public override double Hesapla(){
        return base.Hesapla();
    }
```

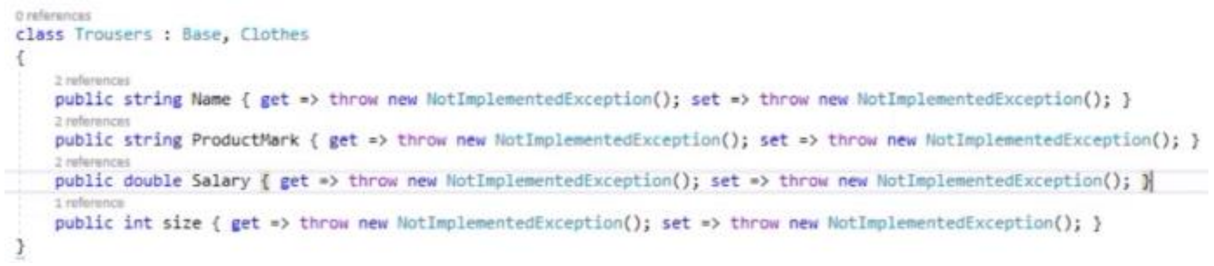
- I – Interface Segregation
  - No client code object should be forced to depend on methods it does not use
  - Each code object should only implement what it needs
  - Only use necessary interface otherwise the code will be inappropriate to the singular responsibility principle



```
interface Base
{
    0 references
    string Name { get; set; }
    0 references
    string ProductMark { get; set; }
    0 references
    double Salary { get; set; }
}

0 references
interface Food
{
    0 references
    double Calorie { get; set; }
}
```

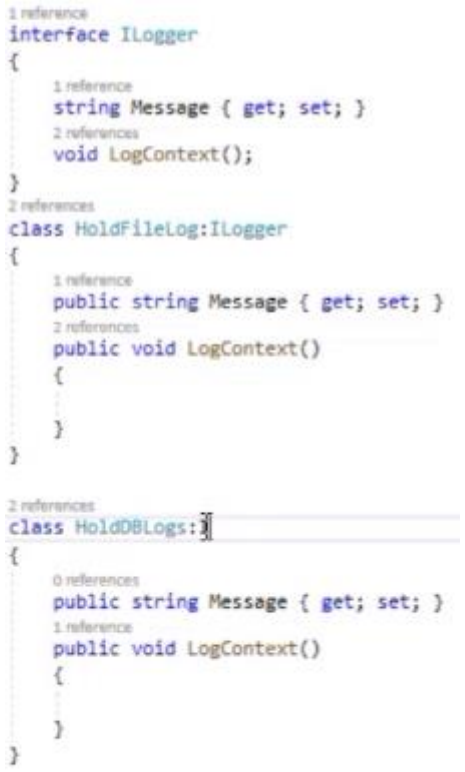


```
interface Clothes
{
    0 references
    int size { get; set; }
}

0 references
class Meal : Base, Food
{
    1 reference
    public string Name { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    1 reference
    public string ProductMark { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    1 reference
    public double Salary { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    1 reference
    public double Calorie { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
}
```

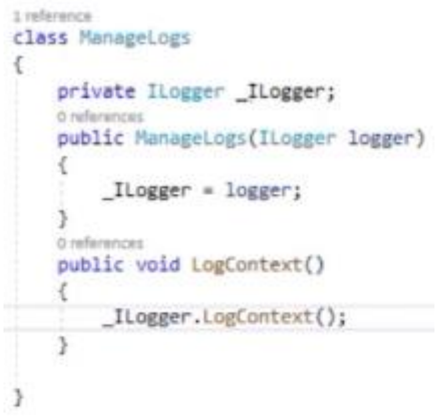


```
0 references
class Trousers : Base, Clothes
{
    2 references
    public string Name { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    2 references
    public string ProductMark { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    2 references
    public double Salary { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    1 reference
    public int size { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
}
```

- D – Dependency Inversion
    - High level modules should not depend on low level modules
        - Both should depend on abstractions
    - Abstractions should not depend on details
        - Details should depend on abstractions
    - Reducing dependencies amongst the modules
    - Shows the importance of interfaces and abstract classes



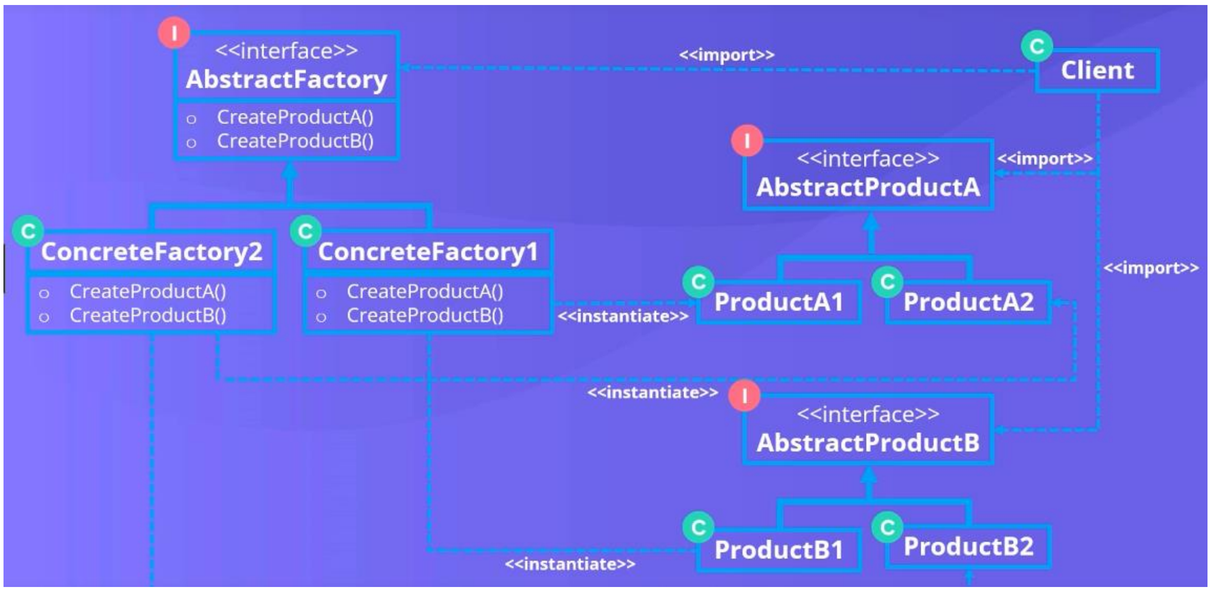
```
1 reference
interface ILogger
{
    1 reference
    string Message { get; set; }
    2 references
    void LogContext();
}
2 references
class HoldFileLog:ILogger
{
    1 reference
    public string Message { get; set; }
    2 references
    public void LogContext()
    {

    }
}

2 references
class HoldDBLogs:I
{
    0 references
    public string Message { get; set; }
    1 reference
    public void LogContext()
    {

    }
}
```

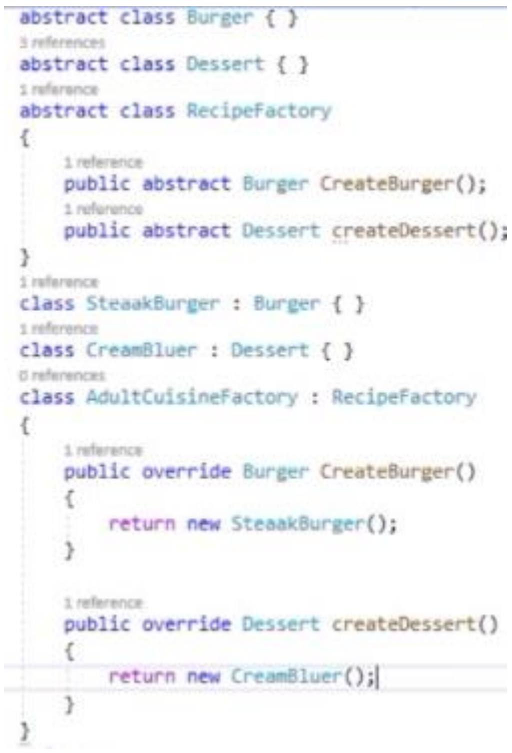


```
1 reference
class ManageLogs
{
    private ILogger _ILogger;
    0 references
    public ManageLogs(ILogger logger)
    {
        _ILogger = logger;
    }
    0 references
    public void LogContext()
    {
        _ILogger.LogContext();
    }

}
```

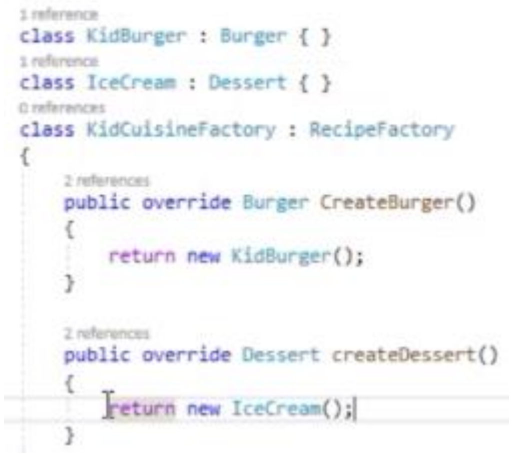
Design Patterns

- Deal with OOP exclusively
- Creational
  - Abstract Factory
    - Interfaces are defined for creating families of related objects without specifying their actual implementations
    - Create factories which return many kinds of related objects

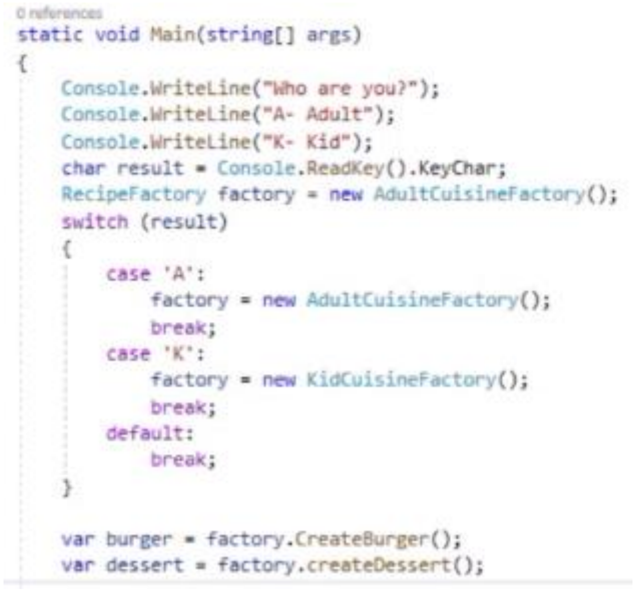




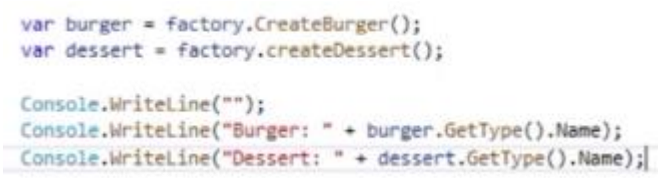
```
abstract class Burger { }
3 references
abstract class Dessert { }
1 reference
abstract class RecipeFactory
{
    1 reference
    public abstract Burger CreateBurger();
    1 reference
    public abstract Dessert createDessert();
}
1 reference
class SteaakBurger : Burger { }
1 reference
class CreamBluer : Dessert { }
0 references
class AdultCuisineFactory : RecipeFactory
{
    1 reference
    public override Burger CreateBurger()
    {
        return new SteaakBurger();
    }

    1 reference
    public override Dessert createDessert()
    {
        return new CreamBluer();
    }
}
```



```
1 reference
class KidBurger : Burger { }
1 reference
class IceCream : Dessert { }
0 references
class KidCuisineFactory : RecipeFactory
{
    2 references
    public override Burger CreateBurger()
    {
        return new KidBurger();
    }

    2 references
    public override Dessert createDessert()
    {
        return new IceCream();
    }
```

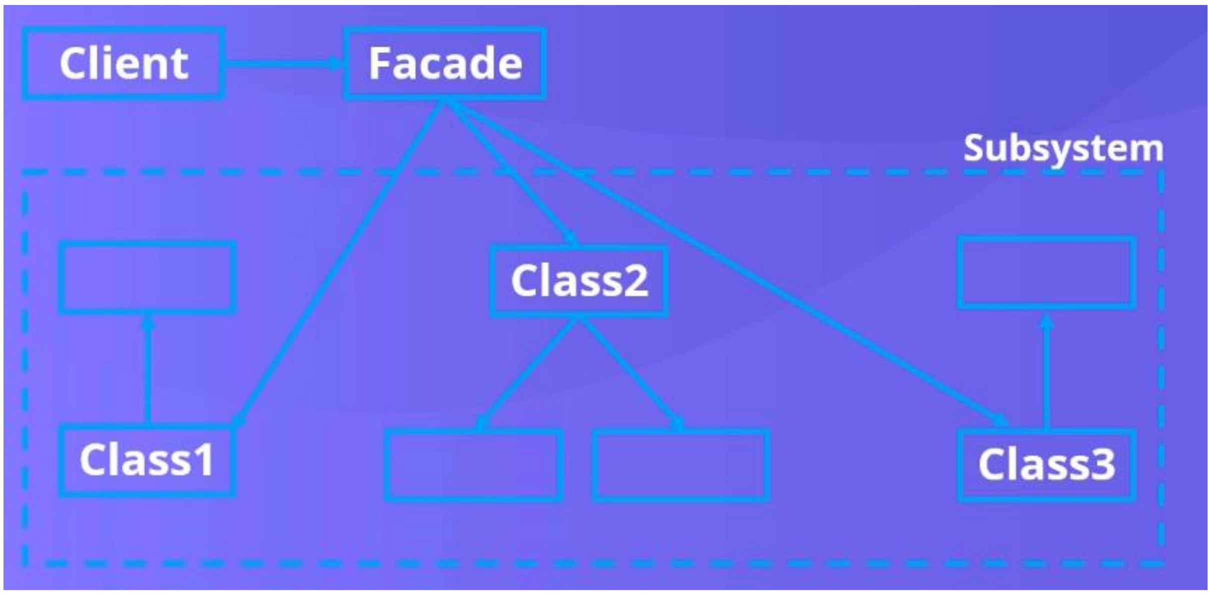


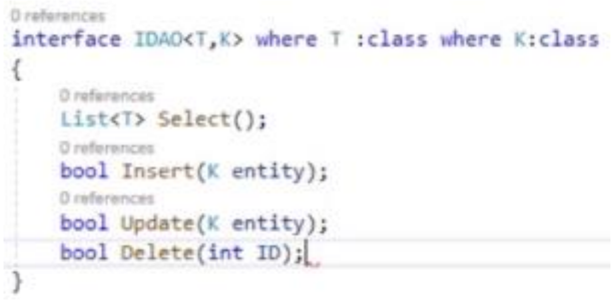
```
0 references
static void Main(string[] args)
{
    Console.WriteLine("Who are you?");
    Console.WriteLine("A- Adult");
    Console.WriteLine("K- Kid");
    char result = Console.ReadKey().KeyChar;
    RecipeFactory factory = new AdultCuisineFactory();
    switch (result)
    {
        case 'A':
            factory = new AdultCuisineFactory();
            break;
        case 'K':
            factory = new KidCuisineFactory();
            break;
        default:
            break;
    }

    var burger = factory.CreateBurger();
    var dessert = factory.createDessert();
```



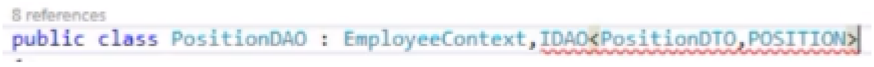
```
var burger = factory.CreateBurger();
var dessert = factory.createDessert();

Console.WriteLine("");
Console.WriteLine("Burger: " + burger.GetType().Name);
Console.WriteLine("Dessert: " + dessert.GetType().Name);
```

  - etc
- Structural
  - Façade
    - Gang of four design
    - Face of the building, don't know anything about the complexities of the building
    - Displays friendly, welcoming face
    - Gives better readability of the code

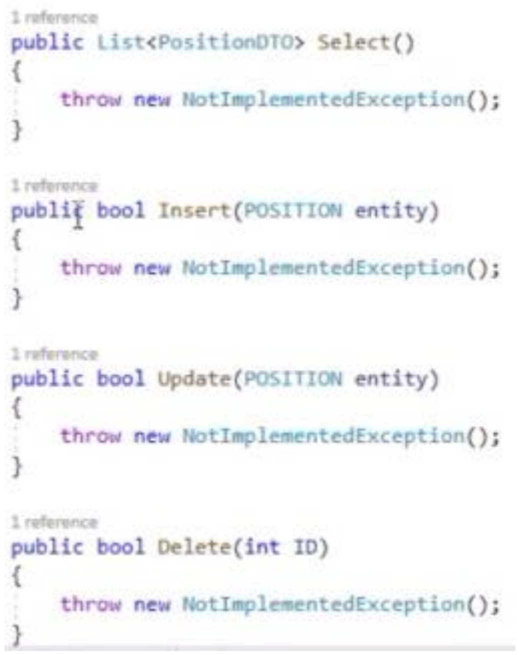




```
0 references
interface IDAO<T,K> where T :class where K:class
{
    0 references
    List<T> Select();
    0 references
    bool Insert(K entity);
    0 references
    bool Update(K entity);
    bool Delete(int ID);
}
```

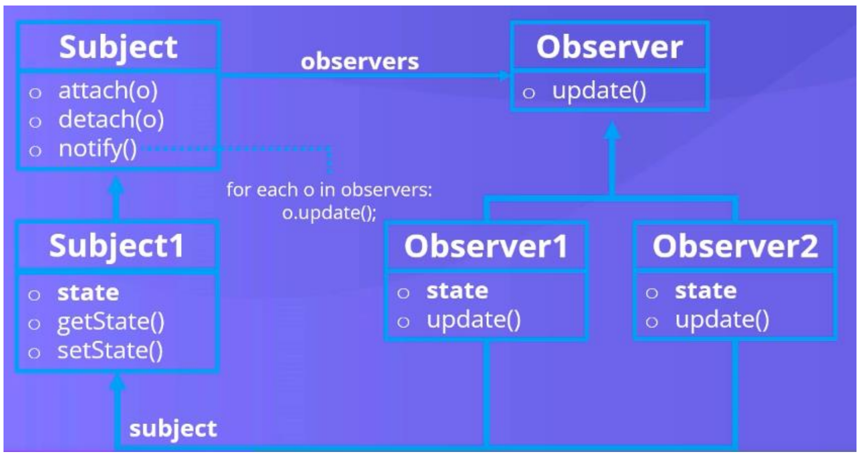


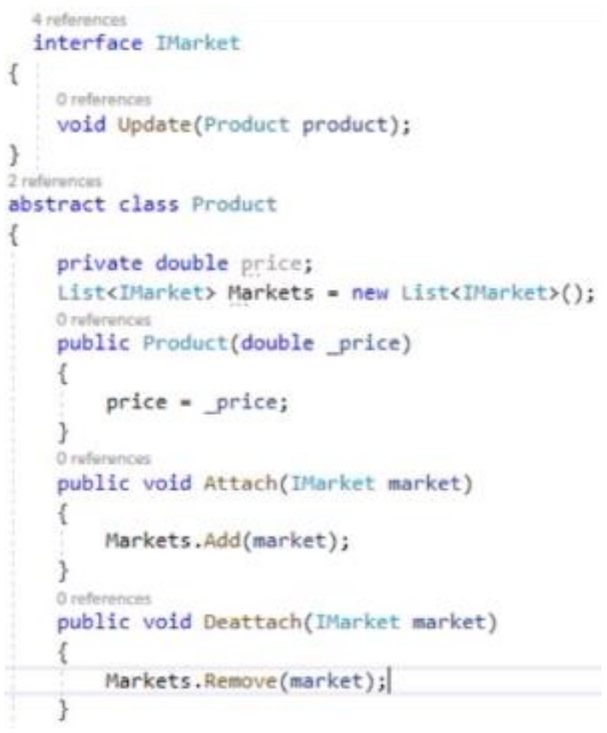
```
8 references
public class PositionDAO : EmployeeContext,IDAO<PositionDTO,POSITION>
```



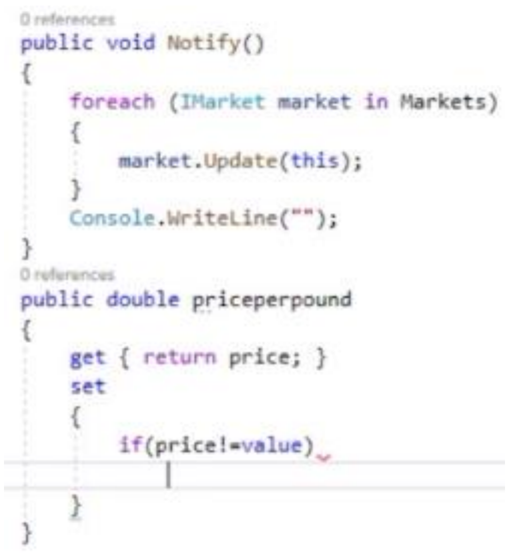
```
1 reference
public List<PositionDTO> Select()
{
    throw new NotImplementedException();
}

1 reference
public bool Insert(POSITION entity)
{
    throw new NotImplementedException();
}

1 reference
public bool Update(POSITION entity)
{
    throw new NotImplementedException();
}

1 reference
public bool Delete(int ID)
{
    throw new NotImplementedException();
}
```

  - etc
- Behavioral
  - Observer
    - To define a one too many dependencies between objects
    - One object changes state, all its dependencies are notified and updated automatically
    - Subject and observers define the one too many relationship
    - Observers are dependent on the subjects

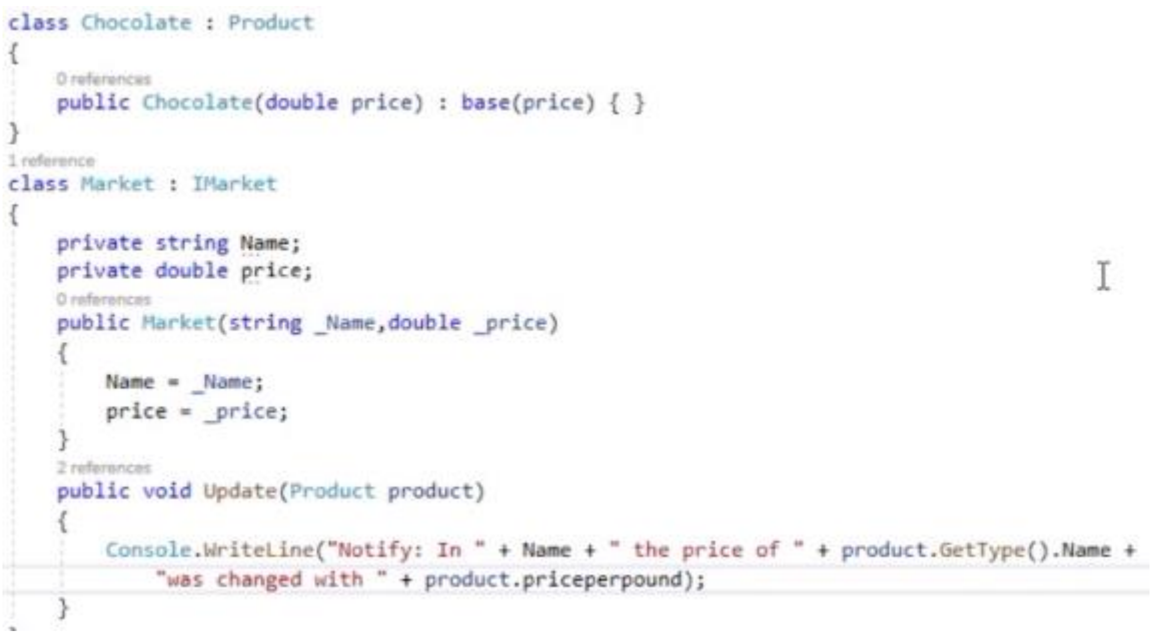




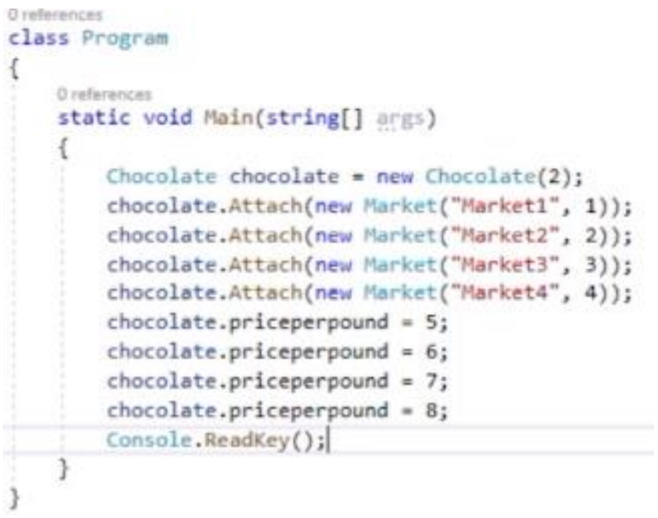
```
  4 references
  interface IMarket
{
    0 references
    void Update(Product product);
}
2 references
abstract class Product
{
    private double price;
    List<IMarket> Markets = new List<IMarket>();
    0 references
    public Product(double _price)
    {
        price = _price;
    }
    0 references
    public void Attach(IMarket market)
    {
        Markets.Add(market);
    }
    0 references
    public void Deattach(IMarket market)
    {
        Markets.Remove(market);
    }
```



```
    0 references
    public void Notify()
    {
        foreach (IMarket market in Markets)
        {
            market.Update(this);
        }
        Console.WriteLine("");
    }
    0 references
    public double priceperpound
    {
        get { return price; }
        set
        {
            if(price!=value)

        }
    }
```



```
class Chocolate : Product
{
    0 references
    public Chocolate(double price) : base(price) { }
}
1 reference
class Market : IMarket
{
    private string Name;
    private double price;
    0 references
    public Market(string _Name,double _price)
    {
        Name = _Name;
        price = _price;
    }
    2 references
    public void Update(Product product)
    {
        Console.WriteLine("Notify: In " + Name + " the price of " + product.GetType().Name +
            "was changed with " + product.priceperpound);
    }
```



```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Chocolate chocolate = new Chocolate(2);
        chocolate.Attach(new Market("Market1", 1));
        chocolate.Attach(new Market("Market2", 2));
        chocolate.Attach(new Market("Market3", 3));
        chocolate.Attach(new Market("Market4", 4));
        chocolate.priceperpound = 5;
        chocolate.priceperpound = 6;
        chocolate.priceperpound = 7;
        chocolate.priceperpound = 8;
        Console.ReadKey();
    }
}
```

- Etc