TASARIM DESENLERİ

Tasarım Deseni Nedir?

Nesne yönelimli programlamanın ortaya çıkışıyla beraber, daha önceden yazılmış hazır bileşenlerin (component) yeniden kullanımı (reusing) konusunda oldukça önemli ilerlemeler sağlamıştır. Bunun paralelinde tecrübelerin (experience) yeniden kullanımı konusunda da çeşitli çalışmalar yapılmış ve desen (pattern) kavramı ortaya çıkmıştır. Desenlerin aksine anti-desen (anti-pattern) ise başarısızlık tecrübelerini anlatır.

Yazılım geliştirme öncesi, geliştirme aşması ve sonrasında daha önce yaşanmış çeşitli problemlere karşı birçok kimse tarafından çeşitli çözümler getirilmiştir. Desenler, daha önce geliştirme yapan programcıların yanlış yaparak doğru yapmayı öğrendikleri başarılı tecrübelerin anlatılması için bir araçtır. Gerçekleştirilen bu çözümlerin, daha sonradan yaşanacak benzer nitelikli problemlerde kullanılması amacıyla desenler kullanılır. Desen kavramının temelleri Mimar Christopher Alexander'ın 1970 sonlarında başlattığı çalışmalara dayanmaktadır[21].

1987 yılında uluslararası "Nesne Yönelimli Programlama, Sistemler, Diller ve Uygulamalar" konferansına kadar desenlerle ilgili bir çalışma ortaya çıkmamıştır[22]. Bu tarihten sonra ise başta *Grady Booch*, *Richard Helm*, *Erich Gamma* ve *Kent Beck* olmak üzere pek çok bilim adamı desenlerle ilgili makale ve sunumlar yayınlamışlardır. 1994 yılında *Erich Gamma, Richard Helm, Ralph Johnson* ve *John Vlissides* tarafından yayınlanan "Tasarım Desenleri: Tekrar kullanılabilir Nesne Yönelimli Yazılımın Temelleri" kitabı, tasarım desenlerin yazılımda kullanılmasında dönüm noktası olmuştur[23]. Yazılımlarda kullanılan desenler yedi ana başlıkta toplanırlar. Bunlar;

1. Mimari desenler (architectural pattern)
2. Analiz desenleri (analysis pattern)
3. Tasarım desenleri (design pattern)
4. Kodlama desenleri (implementation pattern)
5. Test desenleri (test pattern)
6. Çözüm desenleri (solution pattern)
7. Veri desenleri (data pattern)

Bu desenlerin en önemlisi ve ilk gelişenlerden birisi, tasarım desenleridir (design pattern). Tasarım desenleri aslında, Nesne Yönelimli Tasarım Prensiplerini, yazılımlara uygulamanın bir başka ifadesidir. Bu nedenle çok önemlidir. Tasarım desenleri ise üç ana başlıkta toplanmaktadırlar.

1. Nesne imali ile ilgili desenler (creational patterns)
2. Yapısal desenler (structural patterns)
3. Davranışlarla ilgili desenler (behavioral patterns)

Desenlerin çok sık kullanıldığı programlama yöntemine günümüzde desen yönelimli programlama (pattern oriented programming) adı verilmektedir[24]. İzleyen sayfalarda verilen desen UML diyagramlarının birçoğu Erich Gamma'nın Design Pattern CD yayınından alınmıştır[25].

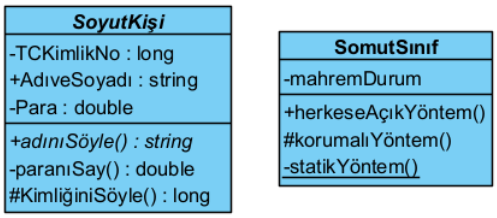
Bölümün devamında verilecek tasarın desenlerinin UML diyagramları visual-paradigm adresinden alınmıştır[26]. Bu bölümde verilen örnekleri iyi anlamak için

Sınıf diyagramları (class diagram), yazılımı geliştirilecek olan sisteme ait sınıfları (class) ve bu sınıflar arasındaki ilişkiyi (relationship) gösterir. Sınıf diyagramları durağandır (static). Yani sadece sınıfların

[21] http://gee.cs.oswego.edu/dl/ca/ca/ca.html
[22] OOPSLA, Object Oriented Programming, Systems, Languages, and Applications
[23] Design Patterns: Elements of Reusable Object-Oriented Software, ISBN 0-201-63361-2
[24] www.mathcs.sjsu.edu/faculty/pearce/patterns.html
[25] Design Pattern CD, Addison-Wesley,1998
[26] https://circle.visual-paradigm.com/category/uml-diagrams/gof-design/

birbiriyle etkileşimini (what interacts) gösterir, bu etkileşimler sonucunda ne olduğunu (what happens) göstermez.

Sınıf diyagramlarında sınıflar, bir dikdörtgen ile temsil edilir. Bu dikdörtgen üç parçaya ayrılır. En üstteki birincisine sınıf ismi yazılır. Ortadaki ikincisinde nitelikler (attribute), en alttaki ve sonuncusunda ise ve işlemler (operation) yer alır.



Şekil 33. Örnek Bir Sınıf Diyagramı

Sınıf isminin italik yazılması, sınıfın soyut (abstract class) sınıf olduğunu gösterir. Benzer şekilde işlemlerin yer aldığı kısımda, yöntemlerin italik yazılması halinde, ilgili yöntemin soyut yöntem (abstract method) olduğunu gösterir.

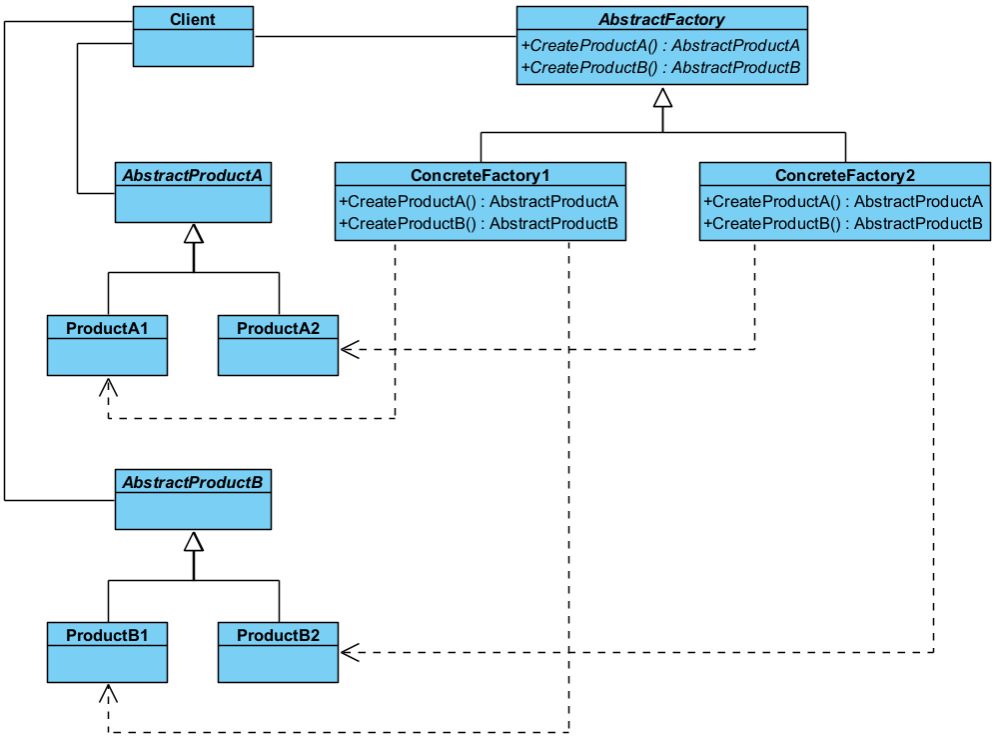
Sınıflarda erişim belirleyiciler (access modifier), özel karakterlerle birbirinden ayrılır. Burada "-" karakteri mahrem (private), "+" karakteri umuma açık (public), "#" karakteri ise korumalı (protected) olduğunu anlatmaktadır. Statik üyeler (static member), altı çizili olarak gösterilir.

Sınıflar Arası İlişkiler başlığını iyi özümsemek gerekir.

Nesne İmalatı ile İlgili Desenler

Nesne imalatı ile ilgili desenler (creational patterns), nesne (object) örnekleme ile ilgili ortaya çıkabilecek çeşitli problemlere çözüm getirmektedirler. Bu desenlerde nesneler dinamik olarak imal edilirler, çünkü değişim yönetiminde statik nesneler kullanılmazlar.

Soyut Fabrika



Şekil 34. Soyut Fabrika Deseni UML Diyagramı

Soyut fabrika deseninde (abstract factory pattern)amaç, nesnelerin imal edilmesi ve kullanılması ile ilgili olan kısımlarının birbirinden ayrılmasıdır. Bu amaca, birbiriyle ilgili ya da birbirine bağımlı olabilecek nesnelerin imal edilmesinde, nesnelerin somut (concrete) sınıflarını değil soyut (abstract) sınıfları kullanılarak ulaşılır. Somut sınıflar istemciden izole edilir ve programcıya somut sınıfları daha kolay değiştirme için olanak sağlar. Bu baslıktaki desenlerde, nesnelerin (objects) yapılandırılıp imal edildiği (creation) yer için fabrika (factory) kavramı, üretilen nesneler için ise ürün (product) kavramı kullanılmaktadır. Aşağıdaki UML diyagramında görüldüğü üzere istemci yalnızca soyut sınıflara bağımlı olduğundan somut ürüne olan bağımlılık azaltılmıştır (loosely coupling).

Şimdi bu deseni kodlamaya başlayalım. İlk önce `AbstractProductA` ve alt sınıflarını kodlayalım.

```
#include <iostream>
using namespace std;

class AbstractProductA {
public:
    AbstractProductA(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
private:
    string productName;
};
class ConreteProductA1: public AbstractProductA {
public:
    ConreteProductA1(string pAdi): AbstractProductA(pAdi) { }
};
class ConreteProductA2: public AbstractProductA {
public:
    ConreteProductA2(string pAdi): AbstractProductA(pAdi) { }
};
```

Ardından `AbstractProductB` ve alt sınıflarını kodlayalım. Bu sınıftan imal edilecek ürünler `AbstractProductA` ürünleriyle etkileşim yaptığı varsayılarak bir `interact` yöntemi eklenmiştir.

```
class AbstractProductB {
public:
    AbstractProductB(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
    void interact(AbstractProductA* a) {
        cout << this-> productName
             << " ürünü " << a->getUrunAdi()
             << " ile etkileşim halinde..." << endl;
    }
private:
    string productName;
};
class ConreteProductB1: public AbstractProductB {
public:
    ConreteProductB1(string pAdi): AbstractProductB(pAdi) { }
};
class ConreteProductB2: public AbstractProductB {
public:
    ConreteProductB2(string pAdi): AbstractProductB(pAdi) { }
};
```

Ardından ürünleri imal edecek fabrika sanal sınıfı ve somut sınıflarını tanımlayalım;

```
class AbstractFactory {
public:
    virtual AbstractProductA* createProductA()=0;
    virtual AbstractProductB* createProductB()=0;
};
class ConreteFactory1: public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ConreteProductA1("A1 ürünü");
    }
    AbstractProductB* createProductB() override {
        return new ConreteProductB1("B1 Ürünü");
    };
};
class ConreteFactory2: public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ConreteProductA2("A2 ürünü");
    }
    AbstractProductB* createProductB() override {
        return new ConreteProductB2("B2 Ürünü");
    };
};
```

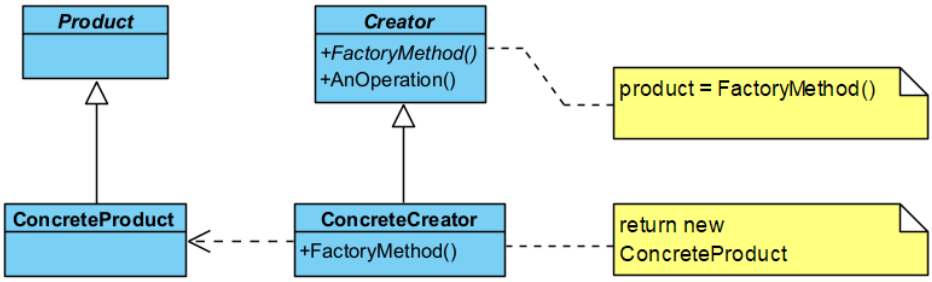
Son olarak fabrika ve ürün sınıflarını sanal olarak kullanan istemci programı kodlayalım;

```
int main() {// Client
    AbstractFactory* factory1=new ConreteFactory1();
    AbstractFactory* factory2=new ConreteFactory2();
    AbstractProductA* productA=factory1->createProductA();
    cout << "Fabrika 1 tarafından İmal Edilen Ürün:" << productA->getUrunAdi() << endl;
    AbstractProductB* productB=factory2->createProductB();
    cout << "Fabrika 2 tarafından İmal Edilen Ürün:" << productB->getUrunAdi() << endl;

    productB->interact(productA);

    delete productB;
    delete productA;
    delete factory2;
    delete factory1;
}
/* Program Çalıştığında;
Fabrika 1 tarafından İmal Edilen Ürün:A1 ürünü
Fabrika 2 tarafından İmal Edilen Ürün:B2 Ürünü
B2 Ürünü ürünü A1 ürünü ile etkileşim halinde...

...Program finished with exit code 0
*/
```

Fabrika Yöntemi

Fabrika yöntemi deseninde (factory method pattern) amaç, imal edilecek nesnenin sınıfını kesin olarak belirtmeden nesne yaratma işleminin gerçekleştirilmesidir. Bunu yapmak için fabrika yöntemi adında soyut (abstract) bir yöntem tanımlanır, fakat nesneleri imal etme (instantiation) işlemi alt sınıflara bırakılır. Yani sanal bir yapıcı (virtual constructor) kullanılır.

Soyut fabrika deseninde olduğu gibi dinamik nesne imal etme ile ilgili `new` anahtar kelimesi, fabrika yöntemi içinde kullanıldığından, istemcinin, ürünün yapıcısına (construtor) bağımlılığı ortadan kalkar. Aşağıdaki UML diyagramında görüldüğü üzere istemcinin somut ürünün yapıcısına bağımlılığı kaldırılmış ve nesne imal etme süreci kontrol altına alınmıştır.



Şekil 35. Fabrika Yöntemi Deseni UML Diyagramı

Şimdi bu deseni kodlamaya başlayalım. İlk önce `Pruduct` ve alt sınıflarını kodlayalım.

```
#include <iostream>
using namespace std;

class Product {
public:
    Product(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
private:
    string productName;
};
class ConreteProduct: public Product {
public:
    ConreteProduct(string pAdi): Product(pAdi) { }
};
```

Ardından ürün imal edecek olan imalatçı `Creator` sınıfını kodluyoruz;

```
class Creator {
public:
    virtual Product* factoryMethod() =0;
    virtual void anOperation() =0;
};
class ConreteCreator: public Creator {
public:
    Product* factoryMethod() override {
        return new ConreteProduct("Fabrika Yöntemi Ürünü");
    };
    void anOperation() override {
        cout << "Fabrika Yöntemini Kullanan Sınıf, Başka İşlemler de yapabilir..." << endl;
    }
};
```

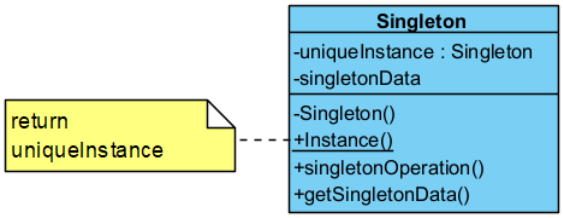
Son olarak ürün ve imalatçı sınıflarını sanal olarak kullanan istemci programı kodlayalım;

```
int main() {// Client
    Creator* imalatci=new ConreteCreator();
    Product* urun=imalatci->factoryMethod();

    cout << "İmalatçı tarafından imal edilen ürün:" << urun->getUrunAdi() << endl;
    imalatci ->anOperation();
    delete urun;
    delete imalatci;
}
/* Program Çıktısı:
İmalatçı tarafından imal edilen ürün:Fabrika Yöntemi Ürünü
Fabrika Yöntemini Kullanan Sınıf Başka, İşlemler de yapabilir...
```

```
...Program finished with exit code 0
*/
```

Tekil Nesne

Tekil Nesne deseninde (singleton pattern) amaç, bir sınıftan yalnızca bir örnek (instance) imal etmektir. Bir sınıfın yalnızca bir nesnesinin olmasına izin verilir, birden fazla olmasına izin verilmez. Yaratılan nesneye genel olarak evrensel (global) olarak erişilir.



Şekil 36. Tekil Deseni UML Diyagramı

Verilen UML diyagramı incelendiğinde, `Singleton` sınıfının `instance` yöntemi her seferinde aynı nesneye ilişkin referans geri döndürülür. Bu desen aynı zamanda, nesneye ilk değerlerin verilerek oluşturulduğu, imal edilme sürecini ve ilk kullanımını izole (encapsulate) eder. Böylece istemcilerin her biri aynı nesne ile muhatap olurlar.

Şimdi `Singleton` sınıfını kodlayalım;

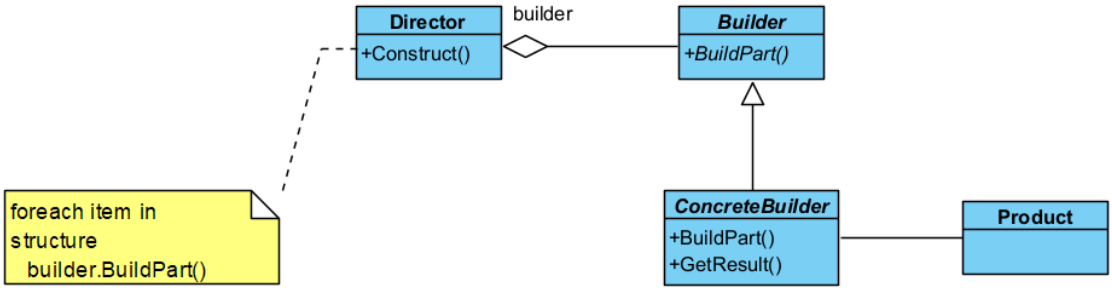
```
#include <iostream>
using namespace std;

class Singleton {
private:
    Singleton() { // Her seferinde başka nesne oluşmasın diye mahrem tanımlanmış
        singletonData=10;
    }
    static Singleton* uniqueInstance;
    int singletonData;
public:
    Singleton(Singleton &other) = delete; // Tekil nesne klonlanabilir olmamalıdır.
    void operator=(const Singleton &) = delete;
        // Tekil nesne atanabilir olmamalıdır.
    static Singleton *Instance() {
        /*
          Bu statik yyöntem;
          Tekil nesneye erişimi kontrol eden statik yöntemdir.
          İlk çalıştırmada, tekil bir nesne imal eder ve onu statik alana
          yerleştirir.
          Sonraki çalıştırmalarda, statik alanda depolanan nesneyi döndürür.
        */
        if(uniqueInstance==nullptr)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    void anOperation() {
        cout << "Tekil nesnenin davranışları da olabilir..." << endl;
    }
    int getSingletonData() const{
        return singletonData;
    }
};

Singleton* Singleton::uniqueInstance= nullptr; // ilk değer verilmeli.
```

Bu sınıfı kullanan istemci aşağıdaki gibi yazılabilir;

```
int main() {
    Singleton* singleton1 = Singleton::Instance();
    Singleton* singleton2 = Singleton::Instance();

    if (singleton1==singleton2)
        cout << "singleton1 ve singleton2 AYNI nesnedir." << endl;
    else
        cout << "singleton1 ve singleton2 FARKLI nesnedir." << endl;

    singleton1->anOperation();
    cout << "Tekil nesnenin verisi:" << singleton1->getSingletonData() << endl;
    delete singleton1;
}
/* Program Çıktısı:
singleton1 ve singleton2 AYNI nesnedir.
Tekil nesnenin davranışları da olabilir...
Tekil nesnenin verisi:10

...Program finished with exit code 0
*/
```

Kurucu

Kurucu deseninde (builder pattern) amaç, çok karmaşık bir nesnenin imal edilmesiyle ilgili işlemleri bir başka sınıfa bırakmaktır. Böylece nesnenin gösterimi ve kullanılması ilişkin kodlar ile yaratılmasına ilişkin kodlar birbirinden ayrılmış olur.



Şekil 37. Kurucu Deseni UML Diyagramı

Yukarıda verilen UML diyagramı incelendiğinde, Ürün imalatını yönetecek olan `Director` nesnesi, karmaşık nesneyi oluşturacak küçük ürünlerin (product) yapımını `Builder` nesnesine bırakır. `Director` nesnesi, oluşturulan bu küçük parçaları `Construct` yöntemi ile bir araya getirir.

İlk önce Product sınıfını tanımlayalım. Ürün birden fazla parçadan oluştuğu için burada temsilen `vector` kullanılmıştır.

```
#include <iostream>
#include <vector> // std::vector
#include <algorithm> //vector::iterator
using namespace std;
class Product {
public:
    Product(string pAdi):productName(pAdi) {
    };
    string getUrunAdi(){
        return productName;
    };
    void addPart(string pPart){
        parts.push_back(pPart);
    }
    void showParts() {
```

```
        cout << productName << " Ürününün parçaları:" << endl;
        vector<string>::iterator part = parts.begin();
        while( part != parts.end()) {
            cout  << *part << endl;
            part++;
        }
    }
private:
    string productName;
    vector<string> parts;
};
```

Buna ek olarak `Builder` ve alt sınıfını tanımlayalım;

```
class Builder {
public:
    virtual void buildPartKisim1()=0;
    virtual void buildPartKisim2()=0;
    virtual void buildPartKisim3()=0;
    virtual Product* getResult()=0;
};
class concreteBuilder:public Builder {
public:
    concreteBuilder() {
        product=new Product("Kitap");
    }
    void buildPartKisim1() override {
       product->addPart("Bölüm1: ...");
    };
    void buildPartKisim2() override {
       product->addPart("Bölüm2: ...");
    };
    void buildPartKisim3() override {
       product->addPart("Bölüm3: ...");
    };
    Product* getResult() override {
        return product;
    }
    ~concreteBuilder() {
        delete product;
    }
private:
    Product* product;
};
```

Şimdi de `Director` sınıfını tanımlayalım;

```
class Director {
public:
    void construct(Builder* builder){
        builder->buildPartKisim1();
        builder->buildPartKisim2();
        builder->buildPartKisim3();
    }
};
```

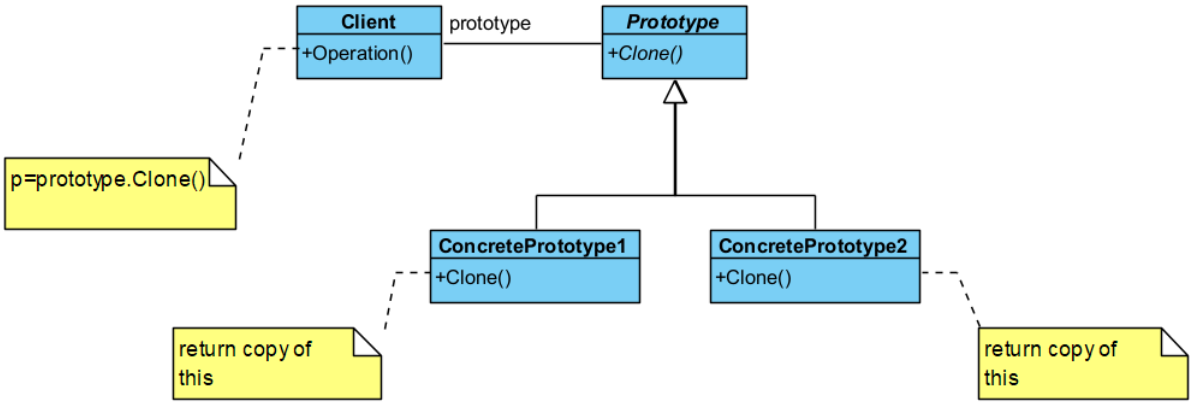
Son olarak istemciyi tanımlayalım;

```
int main() { // Client
    Director* director = new Director();
    Builder* builder = new concreteBuilder();
    director->construct(builder);
```

```
    Product* product = builder->getResult();
    product->showParts();
    delete director;
    delete builder;
}
/*Programın Çıktısı:
Kitap Ürününün parçaları:
Bölüm1: ...
Bölüm2: ...
Bölüm3: ...

...Program finished with exit code 0
*/
```

Prototip

Prototip deseni (prototype pattern), hâlihazırda mevcut olan nesnenin bir kopyasını çıkarmak için kullanılır. Bu desen, imal edilmesi güç olan nesnelere karar verilerek, bu nesneleri tekrar imal etmek için bir sürü işlem yapılması yerine, mevcut nesneden bir kopya çıkararak kullanılmasını sağlamaktadır.



Şekil 38. Prototip Deseni UML Diyagramı

Klonlama yoluyla var olan nesnelerin bir şablonuna dayalı yeni nesneler oluşturmak için kullanılan bu deseni kodlaması aşağıdaki şekilde yapılabilir;

```
#include <iostream>
using namespace std;
class Prototype {
protected:
    int data1;
    int data2;
public:
    void showData() {
        cout << "Object Data: data1=" << data1 <<",data2=" << data2 << endl;
    }
    virtual Prototype* clone()=0;
};
class ConcretePrototype1 : public Prototype {
//Birinci Şablon: Yalnızca data1 alanını kopyalayan prototip nesne imal eder
public:
    ConcretePrototype1(int pData1) {
        data1=pData1;
    };
    Prototype* clone() override {
        return new ConcretePrototype1(*this);
    }
};
class ConcretePrototype2 : public Prototype {
```

```
//İkinci Şablon: Yalnızca data2 alanını kopyalayan prototip nesne imal eder
public:
    ConcretePrototype2(int pData2) {
        data2=pData2;
    };
    Prototype* clone() override {
        return new ConcretePrototype2(*this);
    }
};
void operation() {
    Prototype* object1 = new ConcretePrototype1(40);
    object1->showData();
    Prototype* clone1 = object1->clone();
    clone1->showData();
    Prototype* object2 = new ConcretePrototype2(30);
    object2->showData();
    Prototype* clone2= object2->clone();
    clone2->showData();
    delete object1;
    delete clone1;
    delete object2;
    delete clone2;
}
int main() {  //İstemci
    operation();
}
/* Program Çıktısı:
Object Data: data1=40,data2=0
Object Data: data1=40,data2=0
Object Data: data1=0,data2=30
Object Data: data1=0,data2=30

...Program finished with exit code 0
*/
```

Nesneleri kopyalama söz konusu olduğunda, genel olarak üç tür kavramdan bahsedilir;

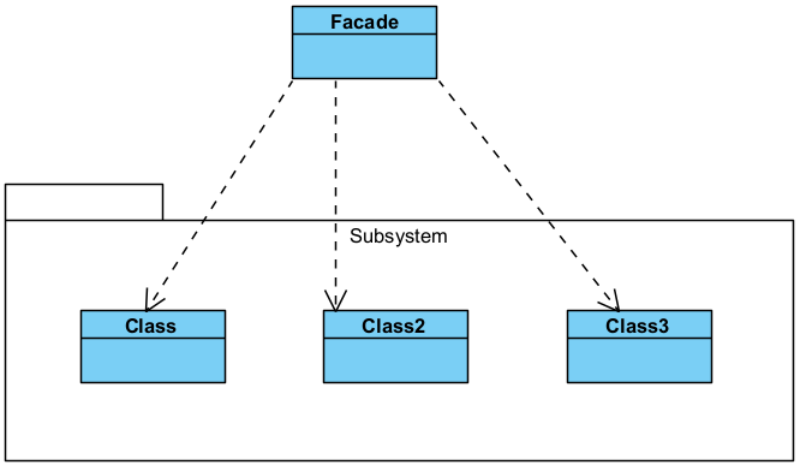
- Üstünkörü kopyalamada (shallow copy) yeni nesne eskisini referans gösterir. Bu durumda birinci nesnenin bir alanı (field) değiştiğinde ikincisinin ki de değişmiş olur. Yani birinde olan değişiklik diğerini etkiler. Bu kopyalamanın diğer adı da bit düzeyi kopyalamadır (bitwise copy).
- Derinlemesine kopyalamada (deep copy) ise birinci nesnenin aynısı bir başka bellek bölgesinde oluşturulur ve ikinci nesne yeni bellek bölgesini referans gösterir. Bu kopyalamanın diğer adı da üye düzeyi kopyalamadır (memberwise copy). Kopyalama sonrasında bir nesne üzerinde olan değişiklik yeni nesneyi etkilemez.
- Üçüncü tip bir kopyalama ise tembel kopyalamadır (lazy copy). Yukarıda bahsi geçen iki kopyalamanın iç içe girmiş şeklidir.

Yapısal Desenler

Yapısal desenler (structural pattern), sınıflar ve nesnelerin çok olduğu daha karmaşık programlarda getirilen yapısal çözüm yöntemleridir.

Vitrin

Vitrin deseninde (facade pattern) alt sistemlerin değişmesi halinde tüm sistemin yeniden derlenme olasılığını bertaraf etmek amacıyla alt sistemlere erişim tek bir sınıf üzerinden yapılır. Çok katmanlı yazılım mimarisinde, alt katmanların içinde olan değişiklikler sistem genelini etkilemez. Çünkü katmanlar birbirlerinin ön yüzünü (facade) görür ve kullanırlar.



Şekil 39. Vitrin Deseni UML Diyagramı

Bir sistem içindeki bir dizi ara yüze tek bir ara yüz üzerinden erişim sağlayacağımız program için bir dişi alt sistem kodunu yazalım;

```
#include <iostream>
using namespace std;
class SubSystem1 {
public:
    void operation11()  {
        cout << "Operation1 for subsystem1..." << endl;
    }
    void operation12()  {
        cout << "Operation2 for subsystem1..." << endl;
    }
    void operation13()  {
        cout << "Operation3 for subsystem1..." << endl;
    }
};
class SubSystem2 {
public:
    void operation21()  {
        cout << "Operation1 for subsystem2..." << endl;
    }
    void operation22()  {
        cout << "Operation2 for subsystem2..." << endl;
    }
    void operation23()  {
        cout << "Operation3 for subsystem2..." << endl;
    }
};
class SubSystem3 {
public:
    void operation31()  {
        cout << "Operation1 for subsystem3..." << endl;
    }
    void operation32()  {
        cout << "Operation2 for subsystem3..." << endl;
    }
    void operation33()  {
        cout << "Operation3 for subsystem3..." << endl;
    }
};
```

Şimdi de `Facade` sınıfını kodlayalım;

```
class Facade {
private:
```

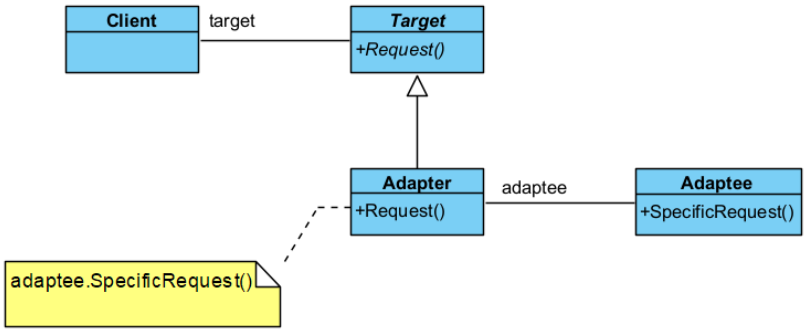
```
  SubSystem1 *subsystem1;
  SubSystem2 *subsystem2;
  SubSystem3 *subsystem3;
public:
  Facade() {
    this->subsystem1 = new SubSystem1();
    this->subsystem2 = new SubSystem2();
    this->subsystem3 = new SubSystem3();
  }
  ~Facade() {
    delete subsystem1;
    delete subsystem2;
    delete subsystem3;
  }
  void operation() {
    this->subsystem1->operation11();
    this->subsystem2->operation22();
    this->subsystem3->operation31();
    this->subsystem3->operation33();
    this->subsystem3->operation32();
  }
};
```

Son olarak istemci kodumuzu yazalım;

```
int main() { // Client
    Facade* facade = new Facade();
    facade->operation();
    delete facade;
}
/*Program Çalışdığında:
Operation1 for subsystem1...
Operation2 for subsystem2...
Operation1 for subsystem3...
Operation3 for subsystem3...
Operation2 for subsystem3...

...Program finished with exit code 0
*/
```

Adaptör

Adaptör deseni (adapter pattern), yabancı bir ara yüzü istemcinin anlayacağı bir ara yüze dönüştürür. Yani farklı ara yüzlere sahip sınıfların, iletişim ve etkileşim kurabilecekleri ortak bir nesne oluşturarak birlikte çalışmalarına izin verir. Bu desen, yabancı ara yüze yaptırılacak işin bir şekilde yapıldığı ve zaten yapılan bu işlemin istemcinin isteğine uygun hale getirildiği düşünülmelidir.



Şekil 40. Adaptör Deseni UML Diyagramı

Aşağıdaki örnekte `Target` nesnesi, istemci tarafından gelen isteği, bu işi yapabilme yeteneğine sahip `Adaptee` nesnesine yaptıracaktır. İşte burada `Adapter` nesnesi devreye girerek `Target` nesnesine gelen istekleri, `Adaptee` nesnesinin yapabileceği şekilde uyarlar.

İlk önce yabancı ara yüzü temsil eden `Adaptee` sınıfını kodluyoruz;

```
#include <iostream>
using namespace std;
class Adaptee {
public:
    void specificRequest() {
        cout << "Adaptee.SpecificRequest() yöntemi çağrıldı." << endl;
    }
};
```

Daha sonra soyut olan `Target` ve somut olan `Adapter` sınıfını kodluyoruz;

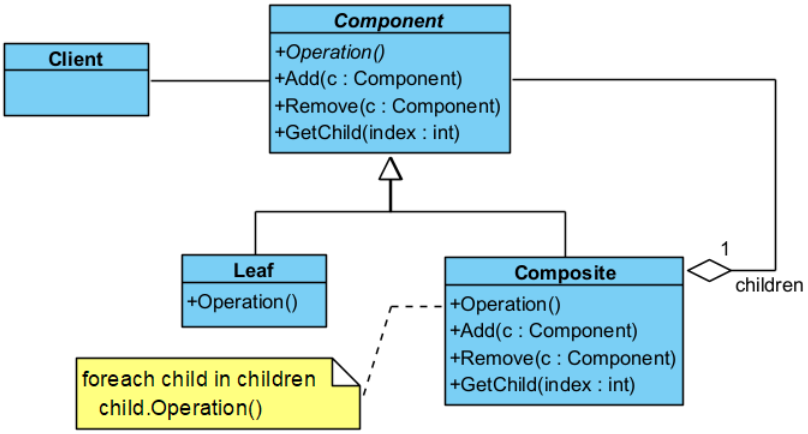
```
class Target {
public:
    virtual void request()=0;
};
class Adapter: public Target {
private:
    Adaptee* adaptee;
public:
    Adapter() {
        adaptee= new Adaptee();
    }
    ~Adapter() {
        delete adaptee;
    }
    void request() override {
        adaptee->specificRequest();
    };
};
```

Son olarak istemci tarafını kodluyoruz;

```
int main() { // Client
    Target* target = new Adapter();
    target->request();
    delete target;
}
/* Program Çıktısı:
Adaptee.SpecificRequest() yöntemi çağrıldı.

...Program finished with exit code 0
*/
```

Bileşik

Bileşik deseni (composite pattern), özyinelemeli (recursive) ya da hiyerarşik yapıya sahip nesne oluşturmak gerektiğinde yapısal olarak nasıl bir kodlama yöntemi izleneceğini söyler. Bu desen, her nesnenin bağımsız olarak veya aynı ara yüz üzerinden iç içe geçmiş nesneler kümesi olarak ele alınabileceği nesne hiyerarşilerinin oluşturulmasını kolaylaştırır. Aşağıda verilen UML diyagramı incelendiğinde; `Leaf`, ağaç yapısı içinde kullanılabilecek, parçalara ayrılamayan en küçük bileşen olan en ilkel (primitive) yapıdır. `Component`, bu ilkel yapılardan oluşturulacak karmaşık `Composite` nesnesine ilkel yapıların eklenip çıkarılmasını sağlayacak bileşendir.



Şekil 41. Bileşik Deseni UML Diyagramı

Şimdi sanal Component sınıfı ile somut sınıflarını tanımlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Component {
private:
    Component *ebeveyn;
public:
    void setEbeveyn(Component *pEbeveyn) {
        this->ebeveyn = pEbeveyn;
    }
    Component *getEbeveyn() const {
        return this->ebeveyn;
    }
    virtual bool IsComposite() const {
        return false;
    }
    virtual void addChild(Component *component) {}
    virtual void removeChild(Component *component) {}
    virtual string displayOperation() const = 0;
};
class Leaf : public Component {
public:
    string displayOperation() const override {
        return "Yaprak";
    }
};
class Composite : public Component {
private:
    list<Component*> cocuklar;
public:
    void addChild(Component* pComponent) override {
        this->cocuklar.push_back(pComponent);
        pComponent->setEbeveyn(this);
    }
    void removeChild(Component* pComponent) override {
        cocuklar.remove(pComponent);
        pComponent->setEbeveyn(nullptr);
    }
    bool IsComposite() const override {
        return true;
    }
    string displayOperation() const override {
```

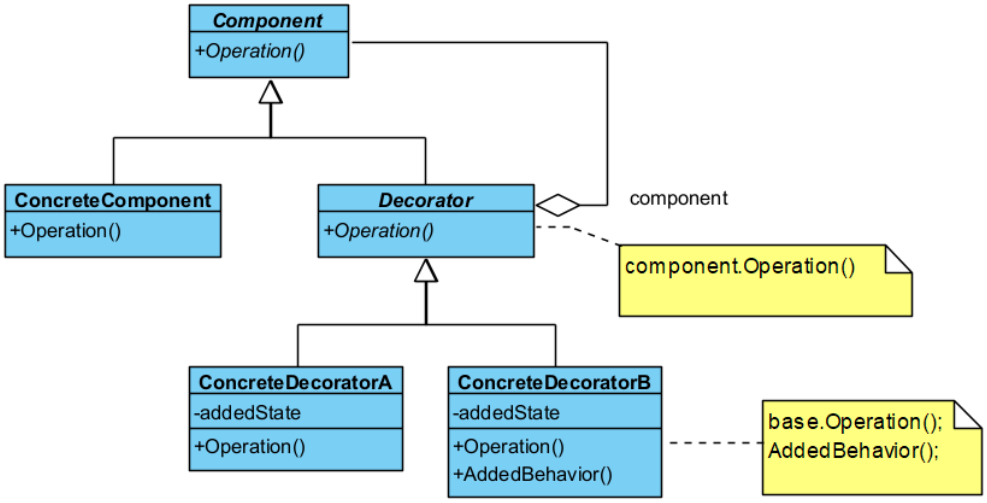
```
        string result;
        for (const Component *iterator : cocuklar) {
            if (iterator == cocuklar.back())
                result += iterator->displayOperation();
            else
            result += iterator->displayOperation() + "+";
        }
        return "Dal(" + result + ")";
    }
};
```

Şimdi istemci kodunu yazabiliriz;

```
int main() { // Client
    Component* yaprak = new Leaf;
    cout << "Sadece Yapraktan Oluşan Bileşik Nesne:" << endl;
    cout << yaprak->displayOperation();
    cout << endl;
    Component* kok = new Composite;
    Component* dal1 = new Composite;
    kok->addChild(dal1);
    Component *yaprak1 = new Leaf;
    dal1->addChild(yaprak1);
    Component *yaprak2 = new Leaf;
    dal1->addChild(yaprak2);
    cout << "Yapraktan ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    cout << endl;
    dal1->removeChild(yaprak2);
    cout << "Bir yaprağı Silinmiş ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    cout << endl;
    Component *dal2 = new Composite;
    kok->addChild(dal2);
    Component *yaprak3 = new Leaf;
    dal2->addChild(yaprak3);
    cout << "Bir dal daha eklenmiş ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    delete yaprak;
    delete kok;
    delete dal1;
    delete dal2;
    delete yaprak1;
    delete yaprak2;
    delete yaprak3;
}
/*Program Çıktısı:
Sadece Yapraktan Oluşan Bileşik Nesne:
Yaprak
Yapraktan ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak+Yaprak))
Bir yaprağı Silinmiş ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak))
Bir dal daha eklenmiş ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak)+Dal(Yaprak))

...Program finished with exit code 0
*/
```

Dekoratör

Dekoratör deseninde (decorator pattern), bir nesneye dinamik olarak yeni durum ve davranışları eklemek mümkündür. Yani çalıştırma anında, bir nesnenin sahip olduğu yeteneklere, yeni yeteneklerin eklenmesini sağlar.



Şekil 42. Dekoratör Deseni UML Diyagramı

UML diyagramı incelendiğinde, `Component` nesnesi, `Decorator` nesnesi aracılığı ile çalıştırma anında (run time), `addedBehavior` yöntemine ve `addedState` durumuna (state) sahip olur.

İlk olarak sanal sınıf olan `Component` ve somut `ConcreteComponent` bileşenlerini kodlayalım;

```
#include <iostream>
using namespace std;
class Component {
protected:
    int state;
public:
    Component(int pState):state(pState) {
    }
    int getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
    virtual void operation() =0;
};
class ConcreteComponent: public Component {
public:
    ConcreteComponent(int pState):Component(pState) {
    }
    void operation() override {
        cout << "Somut ürünün:" << endl
             << "state adında bir durumu var:" << state << endl;
    }
};
```

Şimdi de somut `Decorator` sınıfı ve alt sınıflarını kodlayalım;

```
class Decorator: public Component {
public:
    Decorator(int pState, Component* pComponent):Component(pState) {
        component=pComponent;
    }
```

```
    void operation() override {
        component->operation();
    }
protected:
    Component* component; // aggregation to Component
};
class ConcreteDecorator1: public Decorator {
public:
    ConcreteDecorator1(int pState, Component* pComponent):Decorator(pState, pComponent) {
        addedState=pState*100;
    }
    void operation() override {
       cout << "Decorator 1" << endl
            << "addedState adında yeni bir durumu var:" << addedState << endl;
    }
    int getAddedState() {
        return addedState;
    }
    void setAddedState(int pState) {
        addedState=pState;
    }
private:
    int addedState;
};
class ConcreteDecorator2: public Decorator {
public:
    ConcreteDecorator2(int pState, Component* pComponent):Decorator(pState, pComponent) {
        addedState=pState*200;
    }
    void operation() override {
        cout << "Decorator 2" << endl
             << "addedState adında yeni bir durumu var:" << addedState << endl;
        cout << "addedBehavior Davranışı Var:"
             << addedBehavior() << endl;
    }
    string addedBehavior() {
       return "Bu metin Yeni Davranışdan Geliyor...";
    }
    int getAddedState() {
        return addedState;
    }
    void setAddedState(int pState) {
        addedState=pState;
    }
private:
    int addedState;
};
```

Son olarak istemcimizi kodluyoruz;

```
int main() { // Client
    Component* component = new ConcreteComponent(1);
    component->operation();

    Component* decoratoratedComponent1 = new ConcreteDecorator1(2,component);
    decoratoratedComponent1->operation();

    Component* decoratoratedComponent2 = new ConcreteDecorator2(3,component);
    decoratoratedComponent2->operation();
    delete component,decoratoratedComponent1,decoratoratedComponent2;
}
/* Program Çıktısı:
```

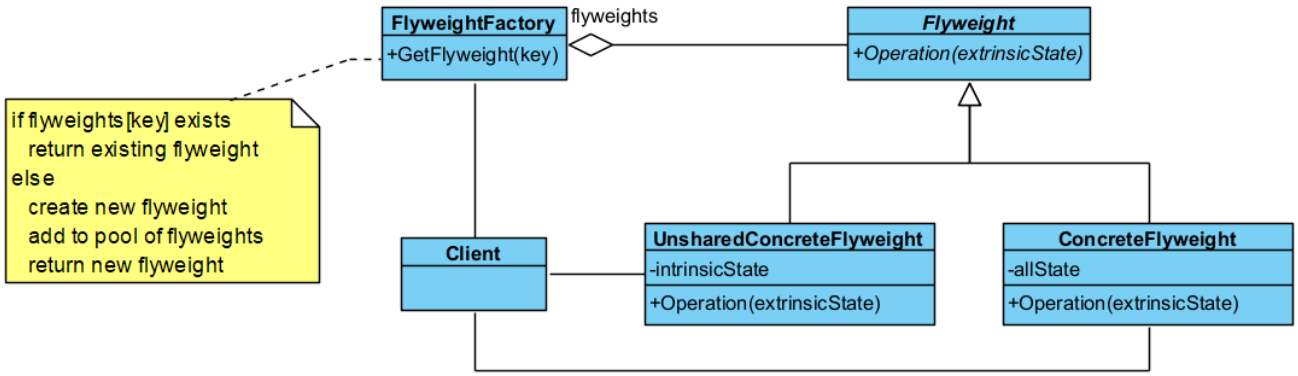
```
Somut ürünün:
state adında bir durumu var:1
Decorator 1
addedState adında yeni bir durumu var:200
Decorator 2
addedState adında yeni bir durumu var:600
addedBehavior Davranışı Var:Bu metin Yeni Davranışdan Geliyor...

...Program finished with exit code 0
*/
```

Sineksıklet

Sineksıklet deseninde (flyweight pattern), birçok küçük nesneden oluşan bir sistemi paylaşarak kullanma hedeflenmiştir. Temel olarak sineksıklet, paylaşılan bir nesnedir ve eş zamanlı olarak küçük nesneleri kullandırır. Yani sineksıklet nesnesi sözü geçen her bir küçük nesne gibi davranır.

Bu desen, bir metin editörü ile açıklanabilir. Metin editöründe yazılacak her bir harf, font bilgisine ve büyüklüğe sahiptir. İşte buradaki her bir harf ile kastedilen, sineksıklet deseninde konu olan küçük nesnelerdir. Bu nesneler bir araya gelerek satırları ve kolonları oluşturur. Sineksıklet nesnesi ise yerine göre bu harflerin her birinin yerine geçer.

Yukarıdaki UML dokümanı incelendiğinde; `FlyweightFactory` küçük nesnelerden büyük resmi oluşturan fabrikayı temsil etmektedir. Birçok uygulamada durumlar geçicidir (extrinsic state) ve bu geçici durumlar saklanmazlar. Geçici durumlarla yapılan işlemler sonunda, kalıcı durumlar (intrinsic state) olur ve nesnelerde saklanır. Geçici durumlarla şekillenen ortak bir nesne (shared object) tanımıyla küçük nesneler tanımlanır. Bu küçük nesnelerin her biri sineksıklet (flyweight) olarak adlandırılır. Ortak özellik taşımayan nesneler (unshared object) de büyük resim içinde yer alabilir. Bu nesneler de ortaklık dışı sineksıklet (unshared flyweight) olarak adlandırılır. Fabrika geçici durumlarla şekillenen sineksıklet nesneler üretir ve büyük resmi oluşturur.



Şekil 43. Sineksıklet Deseni UML Diyagramı

Sineksıklet nesneleri ortak kullanıldığından, uygulama nesnenin kimliğine bağımlı değildir. Bu desen çok sık kullanılmaz ancak aşağıdaki durumların çoğu varsa kullanılabilir;

- Bir uygulama geniş çaplı olup çok miktarda nesne içeriyorsa,
- Nesneleri durumlarıyla olduğu gibi saklamanın maliyeti yüksekse,
- Nesnelerin birçok durumu geçici ve saklanmasına gerek yoksa,
- Birçok nesneden oluşan bir nesne grubu, ortak kullanılan bir nesnedeki geçici değişime bağlı olarak değişebiliyorsa,

İlk önce soyut `Flyweight` sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
```

```
class Flyweight {
protected:
    string state;
public:
    Flyweight(string pState):state(pState) {
    }
    string getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
    virtual void operation(string extrinsicState) =0;
};
class UnsharedConcreteFlyweight: public Flyweight {
protected:
    string intrinsicState;
public:
    UnsharedConcreteFlyweight(string pState): Flyweight(pState) {
        intrinsicState=pState+"+Ek özelliker";
    }
    void operation(string extrinsicState) {
        cout << "Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:"
             << "Durumu:" << state << endl
             << "Geçici Durum:" << extrinsicState << endl
             << "Ortak Olmayan Durum:" << intrinsicState << endl;
    }
};
class ConcreteFlyweight: public Flyweight  {
public:
    ConcreteFlyweight(string pState): Flyweight(pState) {
    }
    void operation(string extrinsicState) {
        cout << "Somut Ortak Özellik Taşıyan Sinetsıklet:"
             << "Durumu:" << state << endl
             << "Geçici Durum:" << extrinsicState << endl;
    }
};
```

Şimdi de `FlyweightFactory` sınıfını kodlayalım;

```
class FlyweightFactory {
private:
    list<Flyweight*> flyweitghts; //Aggregation to Flyweight
public:
    FlyweightFactory(std::initializer_list<Flyweight*> list): flyweitghts(list) {
    }
    Flyweight* getFlyweitght(Flyweight* pFlyweight) {
        list<Flyweight*>::iterator iter = find(flyweitghts.begin(),
                                               flyweitghts.end(),
                                               pFlyweight);
        if (iter!=flyweitghts.end())
            return *iter;
        else {
            flyweitghts.push_back(pFlyweight);
            return pFlyweight;
        }
    }
    void showFlyweights() const
    {
        size_t count = flyweitghts.size();
        cout << "FlyweightFactory: " << count << " kadar flyweight sahibidir:" << endl;
```

```
        for (Flyweight* item : flyweitghts)
            item->operation("+Geçici Durumlar...");
    }
};
```

Son olarak de istemci sınıfı yazalım;

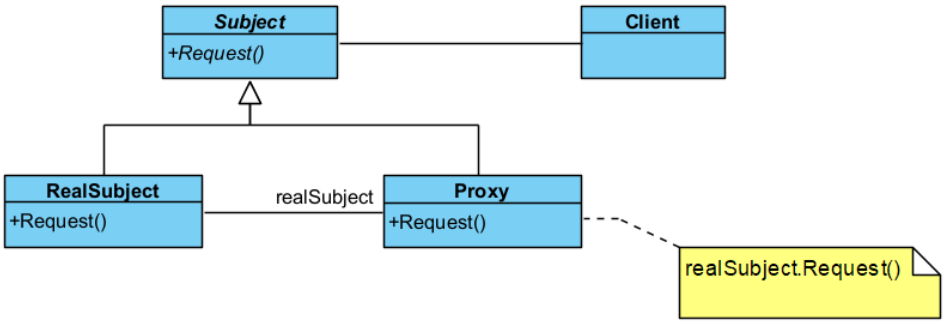
```
int main() { // Client
    Flyweight* ortakOlanSineksiklet1=new ConcreteFlyweight("Binek Araç 1");
    Flyweight* ortakOlanSineksiklet2=new ConcreteFlyweight("Binek Araç 2");
    Flyweight* ortakOlanSineksiklet3=new ConcreteFlyweight("Binek Araç 3");
    Flyweight* ortakOlmayanSineksiklet1=new UnsharedConcreteFlyweight("Kamyonet 1");
    Flyweight* ortakOlmayanSineksiklet2=new UnsharedConcreteFlyweight("Çekici 1");

    FlyweightFactory* fabrika1=new FlyweightFactory({ortakOlanSineksiklet1,
                                                     ortakOlanSineksiklet2,
                                                     ortakOlanSineksiklet3,
                                                     ortakOlmayanSineksiklet1});
    fabrika1->showFlyweights();

    FlyweightFactory* fabrika2=new FlyweightFactory({});
    fabrika2->getFlyweitght(ortakOlanSineksiklet1);
    fabrika2->getFlyweitght(ortakOlmayanSineksiklet2);
    fabrika2->getFlyweitght(ortakOlanSineksiklet1);
    fabrika2->showFlyweights();

    delete ortakOlanSineksiklet1,ortakOlanSineksiklet2,ortakOlanSineksiklet3;
    delete ortakOlmayanSineksiklet1,ortakOlmayanSineksiklet2;
    delete fabrika1,fabrika2;
}
/*Program Çalıştığında:
FlyweightFactory: 4 kadar flyweight sahibidir:
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 1
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 2
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 3
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:Durumu:Kamyonet 1
Geçici Durum:+Geçici Durumlar...
Ortak Olmayan Durum:Kamyonet 1+Ek özelliker
FlyweightFactory: 2 kadar flyweight sahibidir:
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 1
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:Durumu:Çekici 1
Geçici Durum:+Geçici Durumlar...
Ortak Olmayan Durum:Çekici 1+Ek özelliker

...Program finished with exit code 0
*/
```

Vekil

Vekil deseni (proxy pattern), kullanılacak ve erişilecek bir nesnenin yerine geçen bir başka nesneye ihtiyaç duyulduğunda kullanılır. İstemci (client), nesnenin doğrudan kendisine değil, vekil (proxy) nesneye ulaşır ve bu nesneye isteklerini iletir. Kısaca elinden her iş gelen veya diğer nesneler hakkında bilgi sahibi olan bir nesneye ihtiyaç duyulduğunda kullanılır;

– Bir nesne, bulunduğu yerden uzaktaki bir nesneyi kullanacaksa uzak vekil (remote proxy) kullanılır.

- Bir nesneye başka nesneler tarafından erişim kısıtlanacaksa koruma vekili (protection proxy) olarak adlandırılır.
- Bir nesneye erişim sırasında başka işlemler yapılacaksa vekil, akıllı referans (smart reference) olarak adlandırılır. Böyle durumda vekil, nesneye bir erişim yapıldığında, diğer nesnelerin erişmemesi için kullanılacak nesneye kilit koyabilir. Akıllı referans, akıllı gösterici (smart pointer) olarak da adlandırılır.



Şekil 44. Vekil Deseni UML Diyagramı

UML diyagramındaki `RealSubject`, bazı temel iş mantıklarını içerir. Genellikle, giriş verilerini düzeltme gibi hassas olabilen bazı yararlı işler yapma yeteneğine sahiptir. Bir `Proxy`, `RealSubject` kodunda herhangi bir değişiklik yapmadan bu sorunları çözebilir.

İstemci kodunun hem `RealSubject` hem de `Proxy` çalışabilmesi için tüm nesnelerle `Subject` ara yüzü üzerinden çalışması gerekir. Ancak gerçek hayatta, istemciler çoğunlukla `RealSubject` ile doğrudan çalışır. Bu durumda, deseni daha kolay uygulamak için `Proxy`, `RealSubject` sınıfından genişletebilirsiniz.

Şimdi `Subject` sanal sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;

class Subject {
public:
    virtual void request() const = 0;
};
class RealSubject : public Subject {
private:
    int state;
public:
    RealSubject(int pState):state(pState) {
    }
    void request() const override {
        cout << "RealSubject nesnesine gelen istek işleniyor..." << endl;
        cout << "RealSubject state: " << state << endl;
    }
};
class Proxy : public Subject {
private:
    bool CheckAccess() const {
        cout << "Proxy, realSubject nesnesinin erişimini kontrol ediyor..." << endl;
        return true;
    }
    void LogAccess() const {
        cout << "Proxy, gelen talebin iz kaydını tutuyor..." << endl;
    }
public:
    RealSubject *realSubject;
    Proxy()  {
        // RealSubject sınıfından bir örnek imal edip erişiyor.
        realSubject=new RealSubject(20);
```

```
    }
    Proxy(RealSubject* pRealSubject)  {
        // RealSubject sınıfından mevcut bir örneğin kopyası imal edip erişiyor.
        realSubject=new RealSubject(*pRealSubject);
    }
    ~Proxy() {
        delete realSubject;
    }
    void request() const override {
        if (this->CheckAccess()) {
            this->realSubject->request();
            this->LogAccess();
        }
    }
};
```

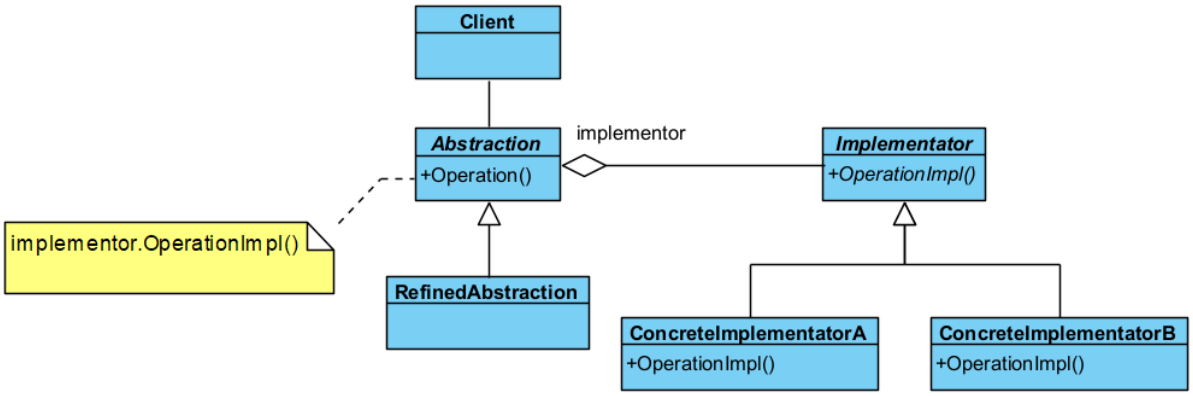
Son olarak istemci kodunu kodlayalım;

```
int main() { //Client-İstemci
    cout << "Client: gerçek özne nesnesine istek gönderiyor:" << endl;
    RealSubject *gercekOzne = new RealSubject(10);
    gercekOzne->request();
    cout << endl;

    cout << "Client: RealSubject sınıfından bir örnek imal ederek kullanıyor:" << endl;
    Proxy *vekil1 = new Proxy();
    vekil1->request();
    cout << endl;

    cout << "Client: gercekOzne nesnesinin kopyasını imal ederek kullanıyor:" << endl;
    Proxy *vekil2 = new Proxy(gercekOzne);
    vekil2->request();

    delete gercekOzne,vekil1,vekil2;
}
/* Program Çıktısı:
Client: gerçek özne nesnesine istek gönderiyor:
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 10

Client: RealSubject sınıfından bir örnek imal ederek kullanıyor:
Proxy, realSubject nesnesinin erişimini kontrol ediyor...
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 20
Proxy, gelen talebin iz kaydını tutuyor...

Client: gercekOzne nesnesinin kopyasını imal ederek kullanıyor:
Proxy, realSubject nesnesinin erişimini kontrol ediyor...
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 10
Proxy, gelen talebin iz kaydını tutuyor...

...Program finished with exit code 0
*/
```

Köprü

Köprü deseni (bridge pattern), işi yapan alt yüklenici veya taşeron (implementator) nesne ile işi yaptıran nesne (client) arasındaki bağımlılığı soyutlama ile ortadan kaldırır. Yani birbiriyle oldukça iç içe olan kodlama ile soyutlamayı ortak ara yüz ile birbirinden uzaklaştırır. Sistemin genişlemesine izin

verir. Bu desen, yazılımlarda oldukça çok kullanılan bir desendi. Çünkü taşeronlar değişse de yeni taşeron eklense de istemci tarafında bir şey değişmez. Köprü deseni aşağıdaki durumlarda kullanılır;



Şekil 45. Köprü Deseni UML Diyagramı

- Taşeron nesneye kalıcı bir bağımlılık istenmediği durumlar. Çalıştırma anında (run time) taşeronlar arasında geçiş yapmayı sağlar.
- Birden çok taşeron kullanarak sistemin genişlemesine ihtiyaç olduğu durumlar.
- Taşeronlar üzerindeki değişiklikler, istemciyi etkilemez. İstemci tarafında derleme olmadan yazılımın değiştirilmesi sağlanır.
- Projede aynı anda farklı birden çok taşeron tasarımı yapılabildiğinden, proje parçalara ayrılarak kendi içinde hızla sınıf tasarımı yapılabilir. Bu durum *Rumbaugh* tarafından iç içe genelleştirme (nested generalization) olarak adlandırılmıştır.

İlk önce taşeron olan soyut `Implementator` sınıfı ve somut alt sınıflarını tanımlayalım;

```
#include <iostream>
using namespace std;

class Implementator { // Taşeron arayüzü
public:
    virtual void operationImpl()=0;
};
class ConcreteImplementatorA: public Implementator{  // A Taşeronu
public:
    void operationImpl() override {
        cout << "A Taşeronu tarafından yapılan iş:..." << endl;
    };
};
class ConcreteImplementatorB: public Implementator{ // B Taşeronu
public:
    void operationImpl() override {
        cout << "B Taşeronu tarafından yapılan iş:..." << endl;
    };
};
```

Şimdi de yüklenici olan `Abstraction` sınıfını ve alt somut sınıfını kodlayalım;

```
class Abstraction { // soyut yüklenici
private:
    Implementator* implementor;
public:
    Abstraction(Implementator* pImplenmentor): implementor(pImplenmentor) {
    }
    void setImplementor(Implementator* pImplenmentor) {
        implementor=pImplenmentor;
    }
    Implementator* getImplementor() {
        return implementor;
```

```
    }
    void operation() {
        cout << "Yüklenici Taşerona iş yaptırıyor:" << endl;
        implementor->operationImpl();
    };
};
class RefinedAbstraction: public Abstraction { // somut yüklenici
public:
    RefinedAbstraction(Implementator* pImplenmentor): Abstraction(pImplenmentor) {
    }
};
```

Son olarak istemi tarafını kodlayalım;

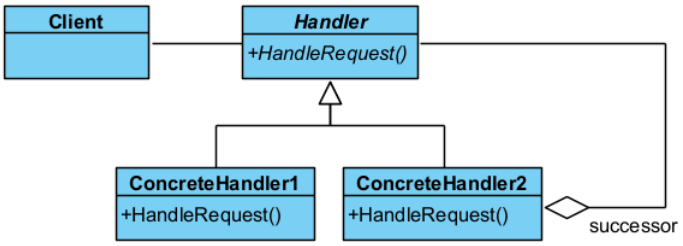
```
int main() { // İstemi-Client
    ConcreteImplementatorA* teseron1=new ConcreteImplementatorA();
    RefinedAbstraction* yuklenici=new RefinedAbstraction(teseron1);
    yuklenici->operation();

    ConcreteImplementatorB* teseron2=new ConcreteImplementatorB();
    yuklenici->setImplementor(teseron2); // çalıştırma anında taşeron değiştiriliyor.
    yuklenici->operation();

    delete teseron2, teseron1, yuklenici;
}
/* Program Çıktısı:
Yüklenici Taşerona iş yaptırıyor:
A Taşeronu tarafından yapılan iş:...
Yüklenici Taşerona iş yaptırıyor:
B Taşeronu tarafından yapılan iş:...

...Program finished with exit code 0
*/
```

Davranışla İlgili Desenler

Davranışla ilgili desenler (behavioral patterns) nesnelerin çeşitli olaylar karşısında gösterecekleri davranışlara çözüm getirmektedirler.

Sorumluluk Zinciri

Bazı durumlarda bir nesne, kendinin yapmayacağı bir işlemi halefine (successor) devredebilir. Ya da nesnenin kendi sorumluluğunu aşan durumlarda ilgili diğer nesneye işlem yaptırılır. İşte bu durumda sorumluluk zinciri deseni (chain of responsibility pattern) kullanılır.



Şekil 46. Sorumluluk Zinciri Deseni UML Diyagramı

Şimdi soyut `Handler` sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;
class Handler {
public:
    virtual void handleRequest(int pRequest)=0;
```

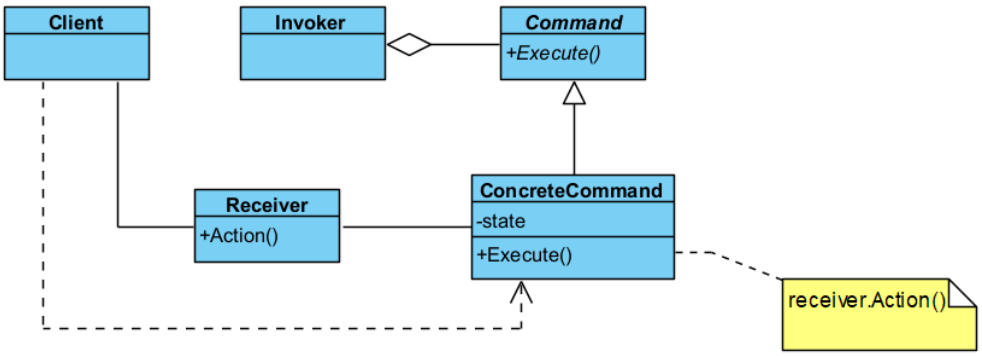
```
};
class ConcreteHandler1:public Handler {
public:
    void handleRequest(int pRequest) override {
        cout << "Yetki Devretmeyen ConcreteHandler1:" << endl;
        cout << pRequest << " ile gelen isteğin tamamını yerine getirdi." << endl;
    }
};
class ConcreteHandler2:public Handler {
private:
    Handler* successor;
public:
    //successor public hale getiriliyor: aggregatotion to Handler
    void setSuccessor(Handler* pSuccessor) {
        successor=pSuccessor;
    }
    Handler* getSuccessor() {
        return successor;
    }
    ConcreteHandler2(Handler* pSuccessor): successor(pSuccessor) {
    }
    void handleRequest(int pRequest) override {
        if (pRequest <100) {
            cout << "Yetki Devredebilen ConcreteHandler2:" << endl;
            cout << pRequest << " ile gelen isteğin tamamını yerine getirdi." << endl;
        } else  {
            cout << "Yetki Devredebilen ConcreteHandler2:" << endl;
            cout << pRequest << " ile gelen iyetkisini halefine devretti:" << endl;
            successor->handleRequest(pRequest);
        }
    }
};
```

Son olarak istemci kısmını kodlayalım;

```
int main() { //İstemci-Client
    Handler* yetkilimemur=new ConcreteHandler1();
    yetkilimemur->handleRequest(100);

    Handler* yetkisizmemur=new ConcreteHandler2(yetkilimemur);
    yetkisizmemur->handleRequest(50);
    yetkisizmemur->handleRequest(200);

    delete yetkilimemur,yetkisizmemur;
}
/* Program Çıktısı:
Yetki Devretmeyen ConcreteHandler1:
100 ile gelen isteğin tamamını yerine getirdi.
Yetki Devredebilen ConcreteHandler2:
50 ile gelen isteğin tamamını yerine getirdi.
Yetki Devredebilen ConcreteHandler2:
200 ile gelen iyetkisini halefine devretti:
Yetki Devretmeyen ConcreteHandler1:
200 ile gelen isteğin tamamını yerine getirdi.

...Program finished with exit code 0
*/
```

Komut

Bazı durumlarda bir nesne diğer bir nesneye ileti (message) verip bir işlem başlattığında, bu işlem bitmeden başka işlemler yapmak isteyebilir. İşte bu tür durumlarda komut deseni (command pattern) kullanılır.



Şekil 47. Komut Deseni UML Diyagramı

Bu desende istemcinin istekleri, `Invoker` nesnesi tarafından kuyruğa sokulur ve kuyruğa sokulan adımların her biri `Command` nesnesi tarafından asıl işi yapan `Receiver` nesnesine yaptırılır. Bu durum bize geri alma (undo) işlemi yapmamızı sağlar.

İlk önce asıl işi yapacak `Receiver` sınıfını kodluyoruz;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Receiver {
public:
    void action(string pParam){
        cout << pParam << " komutu işlemleri yapılıyor..." <<endl;
    }
    void actionA(string pParam){
        cout << pParam << " komutu için A işlemleri yapılıyor"  <<endl;
    }
    void actionB(string pParam){
        cout << pParam << " komutu için B işlemleri yapılıyor"  <<endl;
    }
};
```

Sonrasında her bir komutu tutacak soyut `Command` sınıfı ve somut sınıflarını yazıyoruz;

```
class Command {
public:
    virtual void execute()=0;
};
class SimpleConcreteCommand: public Command {
private:
    string state;
    Receiver* receiver; // private association to Receiver
public:
    SimpleConcreteCommand(string pState):state(pState) {
        receiver=new Receiver();
    }
    ~SimpleConcreteCommand() {
        delete receiver;
    }
    void execute() override {
        receiver->action(state);
```

```
    };
};
class ComplexConcreteCommand: public Command {
private:
    string state;
    Receiver* receiver; // private association to Receiver
public:
    ComplexConcreteCommand(string pState):state(pState) {
        receiver=new Receiver();
    }
    ~ComplexConcreteCommand() {
        delete receiver;
    }
    void execute() override {
        receiver->actionA(state);
        receiver->actionB(state);
    };
};
```

Daha sonra ihtiyaç olursa birden fazla komut tutan ve çalıştıran `Invoker` sınıfını tanımlıyoruz;

```
class Invoker {
private:
    list<Command*> commands;
public:
    Invoker(std::initializer_list<Command*> pCommandList): commands(pCommandList) {
    }
    void addCommand(Command* pCommmand)  {
        this->commands.push_back(pCommmand);
    }
    void doCommands() {
        for (Command* command : commands)
            command->execute();
    }
};
```

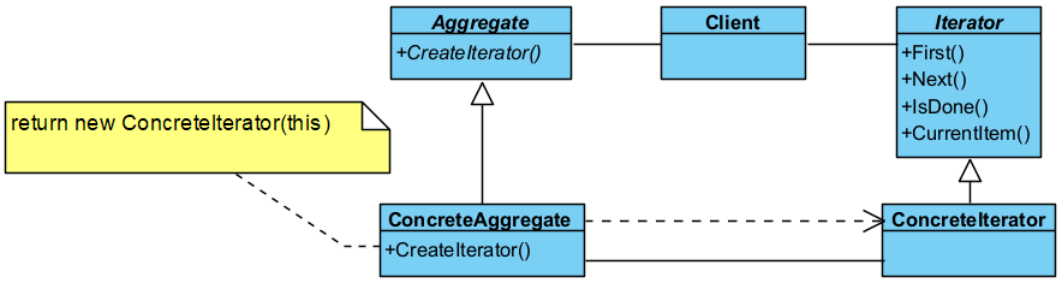
Son olarak istemci kodunu yazıyoruz;

```
int main() { // Client-İstemci
    Command* simpleCommand=new SimpleConcreteCommand("Yaz");
    simpleCommand->execute();

    Command* complexCommand=new ComplexConcreteCommand("Hem Dosyaya Hem Yazıcıya Yaz");
    Invoker* invoker=new Invoker({complexCommand,simpleCommand});
    invoker->doCommands();

    delete simpleCommand,complexCommand,invoker;
}
/* Program çalıştığında;
Yaz komutu işlemleri yapılıyor...
Hem Dosyaya Hem Yazıcıya Yaz komutu için A işlemleri yapılıyor
Hem Dosyaya Hem Yazıcıya Yaz komutu için B işlemleri yapılıyor
Yaz komutu işlemleri yapılıyor...

...Program finished with exit code 0
*/
```

Yineleyici

Yineleyici deseninde (iterator pattern), birden fazla elemandan oluşan küme (aggregate) içindeki her bir elemana sırayla erişim (iteration) sağlar. Erişim işleminde, istemcinin kümenin nasıl yapılandırıldığı konusunda bilgi sahibi olması gerekmez. Bu desende, istemci tarafından veri kümesi soyut (abstract)

olarak kullanılmış olur. Yani istemciye, veri yapısından (data structure) bağımsız bir şekilde verilere erişim sağlayan bir ara yüz (interface) sunulur. Aslında standart şablon kütüphanesindeki (Standart Template Library) yineleyiciler bu deseni kullanır.



Şekil 48. Yineleyici Deseni UML Diyagramı

Burada `Aggregate` ve `Iterator` sınıflarını ilk önce tanımlıyoruz;

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class Iterator; // Böyle bir sınıf tanımlanacak!
class Aggregate {  // double veri tipinde değerlar tutan küme
public:
    virtual void addItem(double pItem)=0;
    virtual int size()=0;
    virtual double getItem(int index)=0;
    virtual Iterator* createIterator()=0;
};
class Iterator {
public:
    virtual double first()=0;
    virtual double next()=0;
    virtual bool isDone()=0;
    virtual double currentItem()=0;
};
class ConcreteIterator:public Iterator {
private:
    Aggregate* aggregate;
    int current;
public:
    ConcreteIterator(Aggregate* pAggregate): aggregate(pAggregate) {
        current=0;
    }
    double first() override {
        current=0;
        return aggregate->getItem(current);
    }
    double next() override {
        current++;
        return aggregate->getItem(current);
    }
    bool isDone() override {
        return (current< aggregate->size()) ? true : false;
    }
    double currentItem() override {
        return aggregate->getItem(current);
    }
};
class ConcreteAggregate : public Aggregate {
private:
```

```
    vector<double> items;
public:
    void addItem(double pItem) override {
        items.push_back(pItem);
    }
    int size() override {
        return items.size();
    }
    double getItem(int index) override {
        return items[index];
    }
    Iterator* createIterator() override {
        return new ConcreteIterator(this);
    }
};
```

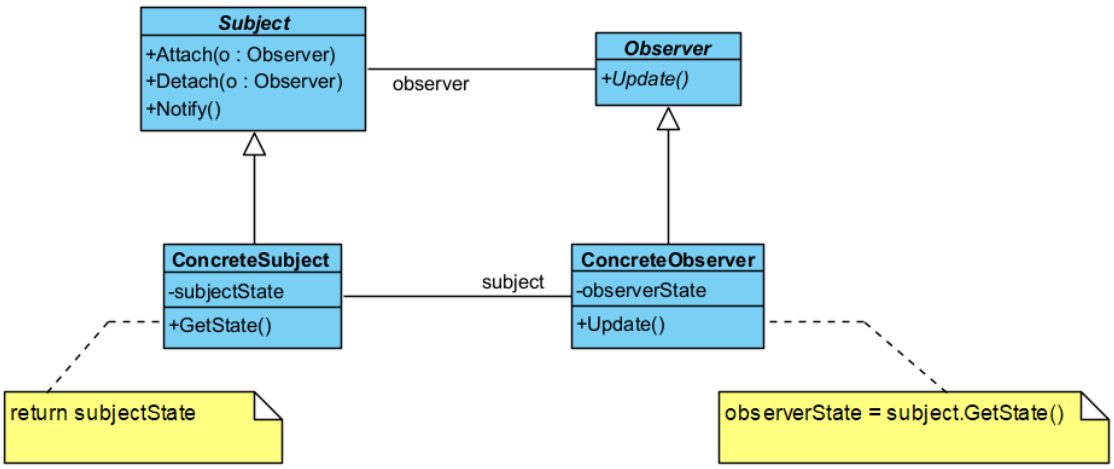
Şimdi de istemci kısmını kodluyoruz;

```
int main() { // İstemci-Client
    ConcreteAggregate* aggregate = new ConcreteAggregate();
    aggregate->addItem(10.0);
    aggregate->addItem(20.0);
    aggregate->addItem(30.0);

    Iterator* iterator = aggregate->createIterator();

    cout << "Aggregate nesneleri:" << endl;
    double item = iterator->first();
    if (iterator->isDone())
    while (iterator->isDone()){
        cout << item << endl;
        item = iterator->next();
    }

    delete aggregate, iterator;
}
/*
Aggregate nesneleri:
10
20
30

...Program finished with exit code 0
*/
```

Gözlemci

Gözlemci deseni (observer pattern), sistemdeki diğer nesnelerin durum değişiklikleri hakkında bir veya daha fazla nesnenin bilgilendirilmesini sağlar. Bağımlı olunan nesnenin durumu veya işin konusu (subject) değiştiğinde, bağımlı olan veya o işi izleyen gözlemcilerin (observer) kendileri de güncellenir. Bu desen aşağıdaki durumlarda kullanılır;

- Bir soyutlama (abstraction) sonucu ortaya çıkan bağımsız iki farklı yönün (aspect) ortaya çıktığı durumlarda, bu yönler birbirinden bağımsız olarak geliştirilebilir ve değiştirilebilir.
- Bir değişikliğin başka değişiklikleri gerektirdiği durumlar.
- Bir nesnenin, diğer nesnelerdeki değişiklikleri kendine vazife etmemesini sağlayan durumlar.



Şekil 49. Gözlemci Deseni UML Diyagramı

İlk önce sanal `Observer` sınıfını tanımlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Observer {
public:
    virtual void update()=0;
};
```

Daha sonra sanal `Subject` ve somut `ConcreteSubject` sınıfını kodlayalım;

```
class Subject {
protected:
    list<Observer*> observers;
public:
    void attach(Observer* pObserver){
        observers.push_back(pObserver);
    }
    void detach(Observer* pObserver){
        observers.remove(pObserver);
    }
    void notify() {
        for (Observer* observer : observers)
            observer->update();
    }
};
class ConcreteSubject: public Subject {
protected:
    int subjectState;
public:
    ConcreteSubject() {
        subjectState=0;
    }
    int getState() {
        return subjectState;
    }
    void setState(int pSubjectState) {
        subjectState=pSubjectState;
    }
};
```

Daha sonra somut `ConcreteObserver` Sınıfını tanımlayalım;

```
class ConcreteObserver:public Observer {
private:
    string gozlemciAdi;
protected:
    ConcreteSubject* subject;
    int observerState;
public:
    ConcreteObserver(string pGozlemciAdi,ConcreteSubject* pConcreteSubject) {
        gozlemciAdi=pGozlemciAdi;
        subject=pConcreteSubject;
    }
    void update() override {
        observerState=subject->getState();
        cout << gozlemciAdi << " gözlemcisinin yeni durumu:" << observerState << endl;
    };
};
```

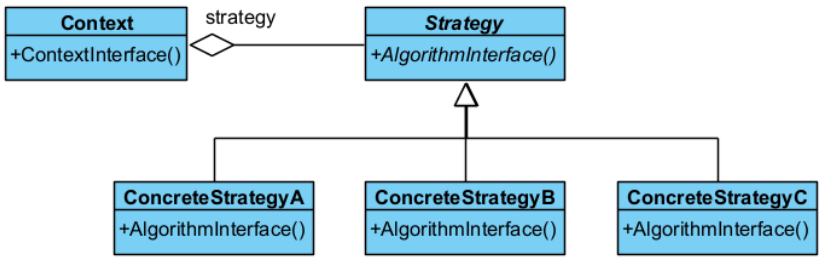
Son olarak istemci tarafını kodlayalım;

```
int main() { //İstemci
    ConcreteSubject* gozlemcileriDeğistirecek = new ConcreteSubject();
    Observer* observer1=new ConcreteObserver("A",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer1);
    Observer* observer2=new ConcreteObserver("B",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer2);
    Observer* observer3=new ConcreteObserver("C",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer3);
    gozlemcileriDeğistirecek->setState(1);
    gozlemcileriDeğistirecek->notify();
    gozlemcileriDeğistirecek->setState(2);
    gozlemcileriDeğistirecek->notify();
    delete gozlemcileriDeğistirecek,observer1,observer2,observer3;
}
/*Program Çıktısı:
A gözlemcisinin yeni durumu:1
B gözlemcisinin yeni durumu:1
C gözlemcisinin yeni durumu:1
A gözlemcisinin yeni durumu:2
B gözlemcisinin yeni durumu:2
C gözlemcisinin yeni durumu:2

...Program finished with exit code 0
*/
```

Strateji

Strateji deseni (strategy pattern), belirli bir davranışı gerçekleştirmek için değiştirilebilen sarmalanmış (encapsulated) algoritmalar kümesini tanımlar. Bu desen aşağıdaki durumda kullanılır;

- Aralarında ilişki bulunan birçok sınıfın davranışlara bağlı olarak anlaşmazlığa düşmesini önlemek için,
- Birden çok algoritmanın olması durumunda,
- Algoritma, istemcinin bilmeyeceği bir veri yapısına sahip olduğu durumlarda,
- Bir sınıf kendi yöntemlerinde çok fazla sayıda duruma göre seçim (conditional choice) kullanıyorsa.



Şekil 50. Strateji Deseni UML Diyagramı

Bu desenin bağlam (Context) nesnesi, diğer tasarım desenlerinin biri ile iç içe kullanılması halinde, sıfır desen (null pattern) olarak adlandırılır. Sıfır desende, `Context` nesnesi akıllıdır ve her zaman ne yapacağını bilir.

İlk önce `Strategy` somut sınıfı ve alt sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;
class Strategy{
public:
    virtual void algorithmInterface()=0;
};
class ConcreteStrategyA: public Strategy{
public:
    void algorithmInterface() override {
        cout << "A stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
class ConcreteStrategyB: public Strategy{
public:
    void algorithmInterface() override {
        cout << "B stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
class ConcreteStrategyC: public Strategy{
public:
    void algorithmInterface() override {
        cout << "C stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
```

Şimdi de `Context` sınıfını kodlayalım;

```
class Context {
private:
    Strategy* strategy;  // aggregation to Strategy
public:
    Context(Strategy* pStrategy):strategy(pStrategy) {

    }
    Strategy* getStrategy() {
        return strategy;
    }
    void setStrategy(Strategy* pStrategy) {
        strategy=pStrategy;
    }
    void contextInterface() {
        strategy->algorithmInterface();
    }
};
```

Son olarak istemciyi kodlayalım;

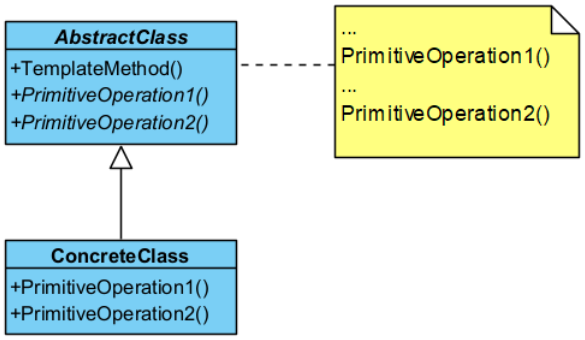
```
int main() { //istemci-client
    Strategy* strateji1=new ConcreteStrategyA();
    Context* baglam= new Context(strateji1);
    baglam->contextInterface();

    Strategy* strateji2=new ConcreteStrategyB();
    baglam->setStrategy(strateji2);
    baglam->contextInterface();

    Strategy* strateji3=new ConcreteStrategyC();
    baglam->setStrategy(strateji3);
    baglam->contextInterface();

    delete baglam,strateji1,strateji2,strateji3;
}
/*Program Çıktısı:
A stratejisinin Algoritması Çalıştırılıyor..
B stratejisinin Algoritması Çalıştırılıyor..
C stratejisinin Algoritması Çalıştırılıyor..

...Program finished with exit code 0
*/
```

Şablon Yöntem

Şablon yöntem deseninde (template method pattern), bir algoritmanın çerçevesini belirler ve uygulayıcı sınıfların gerçek davranışı tanımlamasına olanak tanır.

Kullanılacak algoritmalardan hangilerinin standart hangilerinin somut sınıflara özgü olacağı belirlenmelidir;

- İçinde "Bizi çağırmayın! Biz sizi çağıracağız." sloganını taşıyan yöntemler olan bir soyut bir taban sınıf tasarlanmalıdır.
- Taban sınıf içinde algoritmanın kabuğunu oluşturacak standart adımları içeren şablon yöntem tanımlanır.
- Somut sınıflarda detaylandırılacak yöntemlere ilişkin sanal yöntemler taban sınıfta tanımlanır.
- Şablon yöntem içinde algoritmayı oluşturan yöntemler çağrılır.
- Şablon yöntem içinde yer almayan tüm detay kodlamalar somut sınıflar için de yapılır.



Şekil 51. Şablon Yöntem Deseni UML Diyagramı

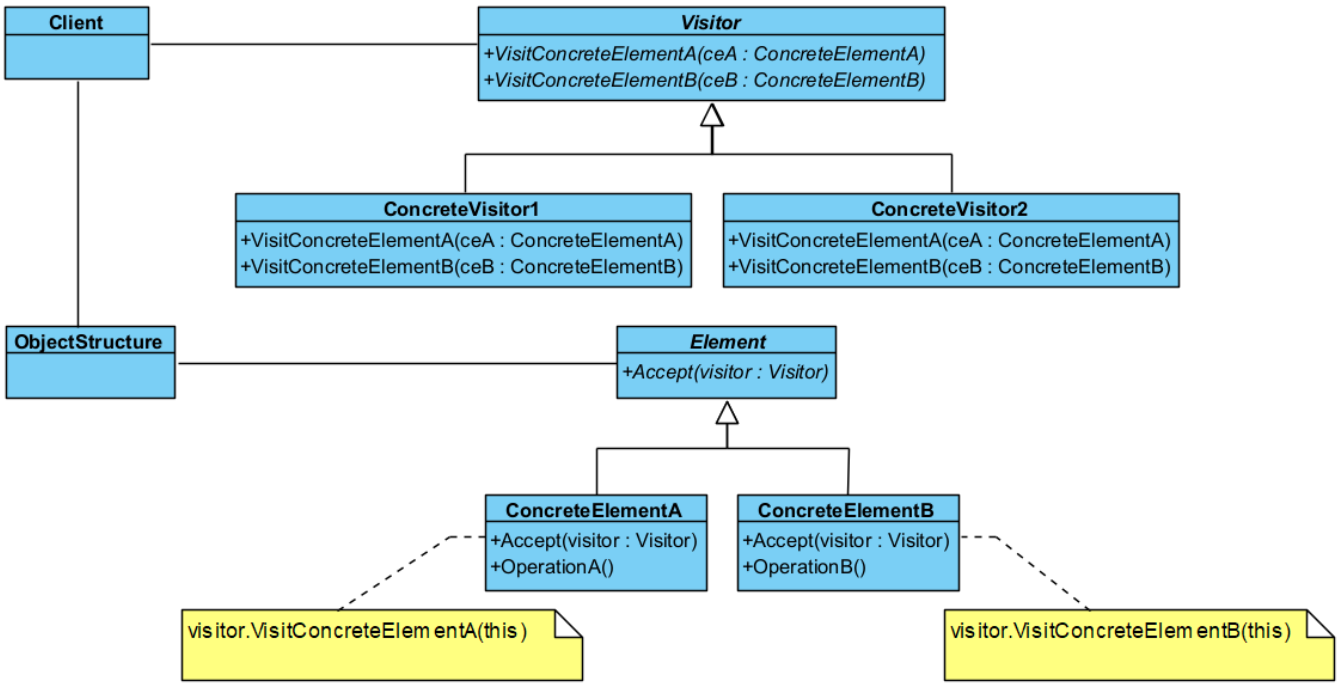
Şimdi Programı kodlayalım;

```
#include <iostream>
using namespace std;

class AbstractClass {
public:
    void templateMethod(){
        baseOperation1();
```

```
        primitiveOperation1();
        requiredOperations1();
        baseOperation2();
        primitiveOperation2();
        requiredOperations2();
    }
    virtual void primitiveOperation1()=0;
    virtual void primitiveOperation2()=0;
protected:
    void baseOperation1() const {
        std::cout << "Algotitmanın 1. TEMEL Adımı yapılıyor...\n";
    }
    void baseOperation2() const {
        std::cout << "Algotitmanın 2. TEMEL Adımı yapılıyor...\n";
    }
    virtual void requiredOperations1()=0;
    virtual void requiredOperations2()=0;
};
class ConcreteClass: public AbstractClass {
public:
    void primitiveOperation1(){
        cout << "Algoritmanın 1. adımı yapılıyor..." << endl;
    }
    void primitiveOperation2(){
        cout << "Algoritmanın 2. adımı yapılıyor..." << endl;
    }
protected:
    void requiredOperations1() override {
        cout << "Algoritmanın 1. GEREKLİ adımı yapılıyor..." << endl;
    }
    void requiredOperations2() override {
        cout << "Algoritmanın 2. GEREKLİ adımı yapılıyor..." << endl;
    }
};
int main() {
    AbstractClass* algoritma= new ConcreteClass();
    algoritma->templateMethod();

    delete algoritma;
}
/*Program Çıktısı:
Algotitmanın 1. TEMEL Adımı yapılıyor...
Algoritmanın 1. adımı yapılıyor...
Algoritmanın 1. GEREKLİ adımı yapılıyor...
Algotitmanın 2. TEMEL Adımı yapılıyor...
Algoritmanın 2. adımı yapılıyor...
Algoritmanın 2. GEREKLİ adımı yapılıyor...

...Program finished with exit code 0
*/
```

Ziyaretçi

Ziyaretçi deseni (visitor pattern), bir sınıfa ilişkin verileri kullanarak işlem yapan bir başka yeni sınıf tanımlanmak istendiğinde kullanılır. Yani çalışma zamanında bir veya daha fazla işlemin bir nesne kümesine uygulanmasına izin verir ve işlemleri nesne yapısından ayırır.



Şekil 52. Ziyaretçi Deseni UML Diyagramı

Başka amaçla tanımlanan yeni sınıfın (`Visitor`), mevcut sınıfın (`Element`) verilerine ulaşmak istemesi halinde bu desen kullanılır. Bu desen aşağıdaki üstünlükleri sağlar;

- Yeni işlemleri (operation) sisteme eklemeyi kolaylaştırır.
- Birbiriyle ilişkili olan işlemleri ilişkili olmayandan ayırmaya yardımcı olur.
- Yeni `ConcreteElement` nesnelerinin eklenmesini zorlaştırır.
- Bütün nesne hiyerarşisi baştanbaşa ziyaret edilebilir.
- Sınıflardaki tüm durumlar (state) biriktirilebilir.
- Bazı durumlarda sarma (encapsulation) işlemini engeller. Yani nesnenin iç durumlarından hareketle herkese açık işlemler tanımlanabilir.

Bu desen aşağıdaki durumlarda kullanılır;

- Nesneler birden fazla olup birçok ara yüze sahip olduğunda, bu nesnelerin somut sınıfları ile bir işlem yapmak gerektiğinde,
- Sınıflar üzerinde yapılacak işlemlerin, sınıflar üzerinde bir iz bırakmamasını sağlamak amacıyla,
- Nadir değişen sınıf yapılarına, sınıf yapısını değiştirmeden, yeni işlemler eklemek gerektiğinde.

İlk önce `Visitor` sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;

class ConcreteElementA; // Böyle bir sınıf tanımlanacak!
class ConcreteElementB; // Böyle bir sınıf tanımlanacak!
class Visitor {
public:
    virtual void visitConcreteElementA(ConcreteElementA* pConcreteElementA)=0;
    virtual void visitConcreteElementB(ConcreteElementB* pConcreteElementB)=0;
};
class ConcreteVisitor1 : public Visitor{
public:
    void visitConcreteElementA (ConcreteElementA* pConcreteElementA) override {
        cout << "ConcreteElementA, ConcreteVisitor1 tarafından ziyaret edildi" << endl;
    }
    void visitConcreteElementB (ConcreteElementB* pConcreteElementB) override {
        cout << "ConcreteElementB, ConcreteVisitor1 tarafından ziyaret edildi" << endl;
    }
```

```
};
class ConcreteVisitor2 : public Visitor{
    void visitConcreteElementA (ConcreteElementA* pConcreteElementA) override {
        cout << "ConcreteElementA, ConcreteVisitor2 tarafından ziyaret edildi" << endl;
    }
    void visitConcreteElementB (ConcreteElementB* pConcreteElementB) override {
        cout << "ConcreteElementB, ConcreteVisitor2 tarafından ziyaret edildi" << endl;
    }
};
```

Daha sonra `Element` sınıflarını kodlayalım;

```
class Element{
public:
    virtual void accept(Visitor* visitor) =0;
};
class ConcreteElementA : public Element{
public:
    void accept(Visitor* visitor) override {
        visitor->visitConcreteElementA(this);
        operationA();
    }
    void operationA() {
        cout << "ConcreteElementA, operationA işlemini yürütüyor..." << endl;
    }
};
class ConcreteElementB : public Element{
public:
    void accept(Visitor* visitor) override {
        visitor->visitConcreteElementB(this);
        operationB();
    }
    void operationB() {
        cout << "ConcreteElementB, operationB işlemini yürütüyor..." << endl;
    }
};
```

Bir de `ObjectStructure` sınıfını kodlayalım;

```
class ObjectStructure{
private:
    list<Element*> elements;
public:
    void attach(Element* pElement){
        elements.push_back(pElement);
    }
    void detach(Element* pElement){
        elements.remove(pElement);
    }
    void accept(Visitor* pVisitor){
        for (Element* element : elements)
            element->accept(pVisitor);
    }
};
```

Son olarak istemci kodumuzu kodlayalım;

```
int main() { //İstemci-Client
    ObjectStructure* nesneler = new ObjectStructure();
    nesneler->attach(new ConcreteElementA());
    nesneler->attach(new ConcreteElementB());
```

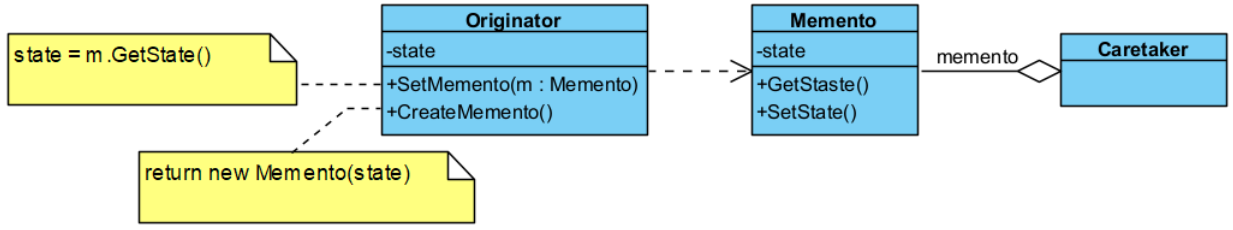
```
    Visitor* ziyaretci1 = new ConcreteVisitor1();
    Visitor* ziyaretci2 = new ConcreteVisitor2();
    nesneler->accept(ziyaretci1);
    nesneler->accept(ziyaretci2);
}
/* Programın Çıktısı:
ConcreteElementA, ConcreteVisitor1 tarafından ziyaret edildi
ConcreteElementA, operationA işlemini yürütüyor...
ConcreteElementB, ConcreteVisitor1 tarafından ziyaret edildi
ConcreteElementB, operationB işlemini yürütüyor...
ConcreteElementA, ConcreteVisitor2 tarafından ziyaret edildi
ConcreteElementA, operationA işlemini yürütüyor...
ConcreteElementB, ConcreteVisitor2 tarafından ziyaret edildi
ConcreteElementB, operationB işlemini yürütüyor...

...Program finished with exit code 0
*/
```

Ziyaretçi deseni uygulanırken iki kodlama (implementation) sorunu irdelenmelidir;

- Çifte sevkiyat (double dispatch), Bir sınıfı hiçbir şekilde değiştirmeden, söz konusu sınıfa etkin biçimde yöntem eklenebilmesi işlemidir. Bu çok bilinen bir tekniktir ve Common Lisp Object System (CLOS) gibi programlama dilleri tarafından desteklenir ve nesne üzerinde sarma (encapsulation) yapılmasına izin verilmez.
- C++, C#, Java gibi birçok nesne yönelimli programlama dili tek sevkiyat (single dispatch) yöntemi ile çalışır. Bu teknik ziyaretçi deseni için anahtardır. `Accept` yöntemi çifte sevkiyat sağlayan yöntemdir. Yöntemler, statik olarak yalnızca `Element` sınıfına bağlanmaz. `Element` sınıfı yapılacak işlemleri `Visitor` üzerinde birleştirir ve `Accept` yöntemi ile çalıştırma anında icra eder. İcra edilen yöntemler, `Visitor` ve `Element` veri tipi olmak üzere iki veri tipine bağlıdır.
- İşlemler dışarıda tutularak oluşturulan nesne yapısının sorumlusu kimdir? `Visitor`, oluşturulan nesne yapısının her bir elemanını tek tek ziyaret etmek zorundadır. Bunun nasıl ve hangi sırada yapılacağı belirgin değildir. Birçok durumda nesne yapısının kendisi bundan sorumlu tutulur. Nesne yapısı basit bir küme olarak tanımlanır ve `Accept` yöntemi her biri için çağrılır. Bazı durumlarda da yineleyici deseni (iterator pattern) kullanılır.

Hatıra

Hatıra deseni (memento pattern), nesnelerin bir andaki durumlarını (state) saklayıp bir başka zaman da bu duruma geri dönmek gerektiğinde kullanılır. Bir nesnenin iç durumunun yakalanmasına ve dışarda saklanmasına olanak tanır, böylece daha sonra geri yüklenebilir fakat tüm bunlar sarmalamayı (encapsulation) ihlal etmeden yapılır.



Şekil 53. Hatıra Deseni UML Diyagramı

Yukarıdaki diyagram incelendiğinde; `Originator` nesnesi, geçmişteki duruma dönecek nesneyi, `Memento` ise `Originator` nesnesinin bir anlık resmini çeken nesneyi, `Caretaker` ise `Memento` nesnelerini saklayan yardımcı nesneyi ifade etmektedir.

İlk olarak `Memento` sınıfını kodlayalım;

```
#include <iostream>
using namespace std;
class Memento {
```

```
private:
    int state;
public:
    Memento(int pState):state(pState) {
    }
    int getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
};
```

Daha sonra `Caretaker` sınıfını kodlayalım;

```
class CareTaker{
private:
    Memento* memento;
public:
    CareTaker(Memento* pMemento):memento(pMemento) {
    }
    Memento* getMemento() {
        return memento;
    }
    void setMemento(Memento* pMemento) {
        memento=pMemento;
    }
};
```

Bir de `Originator` sınıfını kodlayalım;

```
class Originator {
private:
    int state;
public:
    void setState(int pState) {
        state=pState;
        cout << "State Değiştirildi:" << state << endl;
    }
    int getState() {
        return state;
    }
    void setMemento(Memento* pMemento) {
        state=pMemento->getState();
        cout << "State geri alındı:" << state << endl;
    }
    Memento* createMemento() {
        cout << "State saklandı:" << state << endl;
        return new Memento(state);
    }
};
```

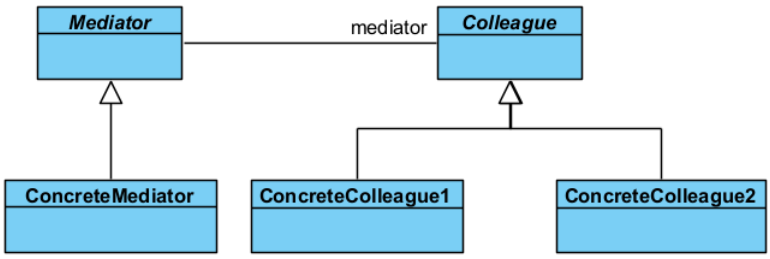
Son olarak istemci kodunu yazalım;

```
int main() { //istemci-client
    Originator* durumDegistiren= new Originator();
    durumDegistiren->setState(1);
    //Burada durumDegistiren nesnesinin durumu saklanıyor:
    CareTaker* durumuSaklayan = new CareTaker(durumDegistiren->createMemento());
    durumDegistiren->setState(-1);
    //Burada durumDegistiren nesnesinin durumu geri alınıyor:
    durumDegistiren->setState(durumuSaklayan->getMemento()->getState());
```

```
}
/*Program Çıktısı:
State Değiştirildi:1
State saklandı:1
State Değiştirildi:-1
State Değiştirildi:1

...Program finished with exit code 0
*/
```

Arabulucu

Arabulucu deseni (mediator pattern), sınıfların arasındaki iletişimi basitleştirir. Bir iletişim nesnesi tanımlanır, diğer nesneler bu nesne üzerinden birbirleriyle haberleşir. Diğer nesnelerin birbiriyle doğrudan iletişim kurmasına izin verilmez. Amaç, iletişimin kontrol altına alınmasıdır. Bu desende arabulucunun (mediator haberi olmadan hiçbir kişi (colleauge) birbiriyle haberleşmez. Ya da bir başka deyişle arabulucunun haberi olmadan hiç kimse birbiriyle görüşmez.



Şekil 54. Arabulucu Deseni UML Diyagramı

Bu desende, sistemde yalnızca bir adet arabulucu olacaksa soyut `Mediator` nesnesi tanımlanmaz. Ayrıca arabulucu ile kişiler arasındaki iletişim iki türlü yapılır;

- Birincisinde bu iletişim gözlemci deseni (observer pattern) ile sağlanır. `Colleague` sınıfı gözlemci desenindeki `Subject` sınıf gibi davranır.
- İkincisinde ise `Mediator` nesnesine, kişiler (colleauge) görüşecekleri kişileri belirterek doğrudan ileti gönderirler.

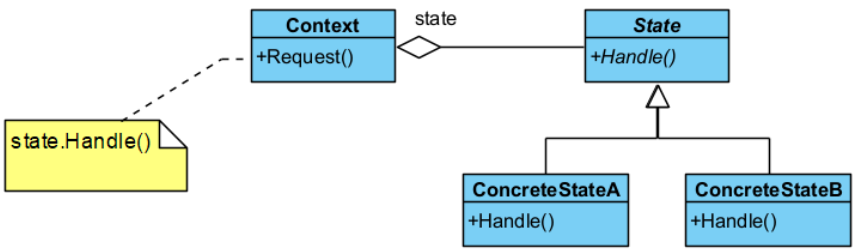
Aşağıda tek bir `Mediator` ve `ConcreteColleague` sınıf olan bir uygulama örneği verilmiştir;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Colleague;
class Mediator {
protected:
    list<Colleague*> colleauges;
public:
    virtual void attach(Colleague* pColleague)=0;
    virtual void send(string pMessage, Colleague* pColleauge)=0;
};
class Colleague {
protected:
    string name;
    Mediator* mediator;
public:
    Colleague(string pName,Mediator* pMediator):name(pName),mediator(pMediator) {
        mediator->attach(this);
    }
    string getName() {
        return name;
    }
```

```
    virtual void sendMessage(string pMessage) {
        mediator->send(pMessage, this);
    }
    virtual void getMessage(string pMessage) {
        cout << name <<":mesaj aldım:" << pMessage << endl;
    }
};
class ConcreteMediator: public Mediator {
public:
    virtual void attach(Colleague* pColleague) {
        colleauges.push_back(pColleague);
    }
    void send(string pMessage, Colleague* pColleauge){
        cout << "LOG:" << pColleauge->getName() << " mesaj gönderdi:" << pMessage << endl;
        for (Colleague* college:colleauges)
            college->getMessage(pMessage);
    }
};
class ConcreteColleague:public Colleague {
public:
    ConcreteColleague(string pName,Mediator* pMediator):Colleague(pName,pMediator) {
    }
};
int main() { //istemci-client
    Mediator* arabulucu=new ConcreteMediator();
    Colleague* ilhan=new ConcreteColleague("ilhan",arabulucu);
    Colleague* mehmet=new ConcreteColleague("mehmet",arabulucu);

    ilhan->sendMessage("Selam");
    mehmet->sendMessage("Merhabalar...");
    ilhan->sendMessage("Nasılsınız?");
}
/* Program Çıktısı:
LOG:ilhan mesaj gönderdi:Selam
ilhan:mesaj aldım:Selam
mehmet:mesaj aldım:Selam
LOG:mehmet mesaj gönderdi:Merhabalar...
ilhan:mesaj aldım:Merhabalar...
mehmet:mesaj aldım:Merhabalar...
LOG:ilhan mesaj gönderdi:Nasılsınız?
ilhan:mesaj aldım:Nasılsınız?
mehmet:mesaj aldım:Nasılsınız?

...Program finished with exit code 0
*/
```

Durum

Durum deseni (state pattern), nesnenin durumunu davranışına bağlar, nesnenin içsel durumuna göre farklı şekillerde davranmasına olanak tanır.



Şekil 55. Durum Deseni UML Diyagramı

Bu desenin bağlam (`Context`) nesnesinin, diğer tasarım desenleri ile iç içe kullanılması halinde, sıfır desen (null pattern) olarak adlandırılır. Bu desenle ilgili olarak iki önemli şeyden bahsetmek gerekir:

- Sahip olunan duruma göre icra edilecek davranış, tanımlanan `State` sınıfına bağlıdır. Yani ilgili davranış somut sınıflardan biri tarafından icra edilir. Bu nedenle somut `State` sınıfları içinde tanımlanan yöntemler durumu kullanan bağlam (`Context`) sınıf içinde tanımlanmak zorundadır.
- Durumu kullanan bağlam (`Context`) sınıf içinde hangi duruma geçildiğinin anlaşılması için ilgili duruma ilişkin tanımlanan özellik (property) içinde set ifadesi kullanılmak zorundadır.

Bu desen netice olarak, nesnenin duruma özgü davranış göstermesini sağlar veya farklı durumlara göre davranışları birbirinden ayırır, durum geçişlerini (state transition) açığa çıkarır. Bu deseni kullanırken aşağıda verilen durumlar özellikle irdelenmelidir;

- Durum geçişleri kim tarafından tanımlanmalıdır? Bu desen, hangi şartlarda durumların ayrıştırılacağını belirtmez. Eğer şartlar belirgin ise bu işlem `Context` nesnesi tarafından yapılır. Bu şartlar belirgin değil ve karmaşık ise bu işlem `State` nesnesi ve halefi (successor) tarafından yapılır.
- Tablo tabanlı alternatif: Bu durumda durum geçişleri bir tablo aracılığı ile yapılır. Tablo üzerine sağlanacak her bir durum yazılır. Bu durumda, durum geçişlerini sağlayacak değişiklikler yeniden kodlama gerektirmez. Ancak bu yöntemde hız düşer. Durum geçişleri tekdüze (uniform) tanımlanmak zorundadır.
- `State` nesnelerini yaratma ve yok etme: Bu nesnenler çalıştırma anında yaratılıp yok edildiklerinden, bu süre zarfında yapılacak işlemler için bir çözüm bulunmalıdır. İki yaklaşım vardır. Birincisi, `State` nesnesini ihtiyaç olduğunda yaratmak kullanmak ve öldürmektir. İkincisi ise bu nesneleri baştan yaratıp hiç öldürmemektir. Genellikle birinci yöntem tercih edilir.
- Dinamik kalıtım (dynamic inheritance) kullanma: Bazı durumlarda nesne, ait olduğu sınıfı çalıştırma anında değiştirir. Bu durumda talep edilen isteğin yerine getirilmesi tam olarak başarılamaz. Ancak birçok nesne yönelimli programlama dili bunu desteklemez.

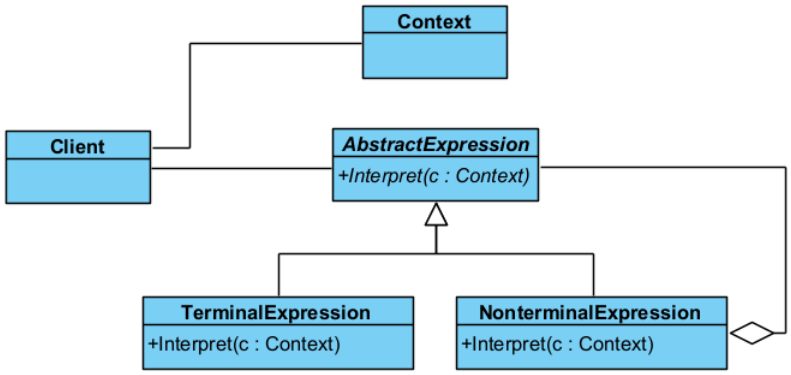
Aşağıda durum nesnelerini değiştiren basit bir önek verilmiştir;

```
#include <iostream>
using namespace std;

class State {
public:
    virtual void handle() = 0;
};
class Context {
private:
    State* state;
    State* state1;
    State* state2;
public:
    Context(State *pState1,State* pState2) : state1(pState1),state2(pState2) {
        state=pState1;
    }
    void durumDegistir() {
        if (state==state1)
            state=state2;
        else
            state=state1;
        cout << "Context: Durum nesnesi değişti!" << endl;
    }
    void request() {
        this->state->handle();
        durumDegistir();
    }
};
class ConcreteStateA : public State {
public:
```

```
    void handle() override {
        cout << "ConcreteStateA  request talebini yerine getiriyor..." << endl;
    };
};
class ConcreteStateB : public State {
public:
    void handle() override {
        cout << "ConcreteStateB  request talebini yerine getiriyor..." << endl;
    }
};
int main() {
    State* durumA=new ConcreteStateA();
    State* durumB=new ConcreteStateB();
    Context *context = new Context(durumA,durumB);
    context->request();
    context->request();
    context->request();
    delete durumA,durumB,context;
}
/* Program Çıktısı:
ConcreteStateA  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!
ConcreteStateB  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!
ConcreteStateA  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!

...Program finished with exit code 0
*/
```

Yorumlayıcı

Yorumlayıcı deseni (interpreter pattern), programlama dili yorumlama özelliğini uygulamalarımıza katmanın yolunu gösterir. Yani bir dilbilgisi için bir temsili ve aynı zamanda dilbilgisini anlayıp ona göre hareket etmeyi sağlayacak bir mekanizmayı tanımlar.



Şekil 56. Yorumlayıcı deseni UML Diyagramı

Bu desen, yorumlanacak dilin dilbilgisi (grammar) kurallarının basit olduğu durumlarda kullanılır. Karmaşık dilbilgisi kurallarına sahip diller için yaratılacak sınıfların yönetimi oldukça zorlaşır. Burada, bağlam (`Context`) nesnesi dilbilgisi kurallarına sahip yorumlanacak nesneyi gösterir. `TerminalExpression`, en ilkel dil öğesini yorumlayacak nesnedir. `NonterminalExpression` ise ilkel dil öğelerinden oluşacak karmaşık yapıyı yorumlayacak nesneyi belirtir.

Bu deseni uygularken aşağıda verilen durumlar irdelenmelidir;

- Soyut sözdizimin oluşturulması: Bu desen yorumlanacak dile ilişkin soyut bir sözdizimin nasıl olması gerektiğini açıklamaz. Tabloya dayalı, ağaç yapısı şeklinde el becerisi ile veya doğrudan istemci tarafından yapılabilir.
- Yorumlayacak işlemin tanımlanması: Eğer işlemler ortak bir yapıdaysa yeni bir `Expression` nesnesi tanımlanarak yapılabilir. Daha karmaşık durumlarda ziyaretçi deseni kullanılır. Programlama dillerinin çoğu soyut bir sözdizimine sahiptir ve yorumlanacak öğeler ayrı bir `Visitor` nesnesi tarafından yorumlanır.
- Cümle sonlarını bitiren ortak semboller için sineksıklet deseni kullanılabilir: Cümle sonlarındaki semboller genellikle bilgi saklamazlar. Yorumlamaya gerek duyulmazlar. Bu nedenle sineksıklet deseni (flyweight pattern) yorumlanacak yerler için bir eleme yapar.

Aşağıda kavramsal olarak bu desenin kolanmış örneği bulunmaktadır;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

class Context {
private:
    string context;
public:
    Context(string pContext): context(pContext) {
    }
};

class AbstractExpression {
public:
    virtual void interpret(Context* context)=0;
};

class TerminalExpression : public AbstractExpression {
public:
    void interpret(Context* context) override {
        cout << "Terminal Expression İşlemi Yorumluyor..." << endl;
    }
};

class NonterminalExpression : public AbstractExpression {
private:
    AbstractExpression* abstractexpression;
public:
    void interpret(Context* context) override {
        cout << "Non Terminal Expression İşlemi Yorumluyor..." << endl;
        abstractexpression = new TerminalExpression();
        abstractexpression->interpret(context);
        delete abstractexpression;
    }
};

int main(){ //İstemci-Client
    list<AbstractExpression*> expressions;
    AbstractExpression* nte=new NonterminalExpression();
    AbstractExpression* te=new TerminalExpression();
    expressions.push_back(te);
    expressions.push_back(nte);
    Context* context = new Context("987698");
    for (AbstractExpression* ae:expressions)
        ae->interpret(context);
}
```

```
/*Program Çıktısı:
Terminal Expression İşlemi Yorumluyor...
Non Terminal Expression İşlemi Yorumluyor...
Terminal Expression İşlemi Yorumluyor...

...Program finished with exit code 0
*/
```