Design Patterns

SOLID Principles, Design Patterns and Anti-Patterns with C++ Examples

Selim Uysal
Ankara, Turkey

Abstract

In this article, we discuss a topic that is crucial for programmers: how to write better code. The article consists of four sections: introduction, design patterns, anti-patterns and conclusion. First, we explain the SOLID principles and then examine design patterns, including their advantages and disadvantages. We explore each design pattern in detail and give an example that illustrates its application. Finally, we explain what anti-patterns are.

Patterns covered in this article (25)

Creational Patterns (6)	Structural Patterns (7)	Behavioral Patterns (12)
• Factory Method • Abstract Factory • Builder • Singleton • Prototype • Object Pool	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Template Method • Iterator • Mediator • Memento • Null Object • Observer • State • Strategy • Visitor • Interpreter

1. SOLID Principles

When developing software, it is recommended to apply the SOLID principles to improve the readability, maintainability and flexibility of the code.

- **Single Responsibility Principle:** A class or function should have only one responsibility.
- **Open/Closed Principle:** Classes and functions should be open for extension but closed for modification.
- **Liskov Substitution Principle:** Derived classes should be substitutable for their base classes.
- **Interface Segregation Principle:** Classes should not be forced to depend on interfaces they do not use.
- **Dependency Inversion Principle:** High-level modules should not depend on low-level modules; both should depend on abstractions.

These five principles should be applied to all code. In addition to them, there are design patterns, which are well-known, proven code structures.

2. Design Patterns

Design patterns are reusable solutions to common challenges in object-oriented software development. These solutions have been developed over time to address recurring problems. Design patterns are not libraries; they are code structures that are written within your own code. They can be categorized into three types: creational, structural and behavioral patterns.

Advantages	Disadvantages
• Reduces design errors and repetitive code. • Increases the reusability of the code. • Makes the design flexible, extensible and sustainable.	• In some cases, it may cause unnecessary complexity. • It may increase the cost, since additional classes and code are required.

A note on the code listings

All examples are written in C++. The identifiers in the code are in Turkish; the most common ones are translated below so that the listings are easy to follow.

Turkish	English	Turkish	English
Araba	car	Acik / Kapali, ac / kapat	open (on) / closed (off), turn on / turn off
Mermi	bullet	suankiState	current state
havuz	pool	Tost, Acili, Klasik	toast, spicy, classic
al / geriVer	take / give back	Guvenlik, Odeme, Envanter	security, payment, inventory
Sekil, Kare, Daire	shape, square, circle	AlisverisMerkezi	shopping center
Asker	soldier	Hizli, Luks	fast, luxury
kopyaOlustur	create copy	hamurHazirla, malzemeleriEkle	prepare dough, add ingredients
Lamba	lamp	pisir, servisEt	bake, serve

2.1. Creational Patterns

These patterns deal with object creation and the process of creating objects.

2.1.1 Factory Method Pattern

Object creation is handled in a single place (a factory method), and the other classes use the objects it produces. Its most important feature is that it makes the code flexible by letting subclasses decide which objects are created. It is usually used when the exact types of the objects to be created are not known in advance. The most important advantage is that when a new type is added, the code continues to work by adding only the necessary class instead of changing the whole code.

Example: There is a car class (Araba); the Hatchback and Sedan classes inherit from it and override its function. The objects are produced from a single class: the getAraba function of the ArabaFactory class creates a hatchback or a sedan according to the type string, using if/else conditions. In main, the objects are created as pointers by the factory class, allocating memory dynamically. If the team that wrote this code

wants to produce a new car model, they can add it by adding one new class (and one branch in the factory) without changing the rest of the code, which gives the code flexibility.

Code listing – Factory Method (part 1 of 2)

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

class Araba {
public:
    string getTip() {
        return "Bilinmeyen Araba Tipi";
    }
};

class Hatchback : public Araba {
public:
    string getTip() {
        return "Hatchback Tipinde Araba Seçildi.";
    }
};

class Sedan : public Araba {
public:
    string getTip() {
        return "Sedan Tipinde Araba Seçildi.";
    }
};

class ArabaFactory {
public:
    static Araba* getAraba(string tip) {
        if (tip == "Sedan")
            return new Sedan();
        else if (tip == "Hatchback")
            return new Hatchback();

        return nullptr;
    }
};
```

Code listing – Factory Method (part 2 of 2)

```cpp
int main() {
    Araba* hatchbackAraba = ArabaFactory::getAraba("Hatchback");
    Araba* sedanAraba = ArabaFactory::getAraba("Sedan");

    cout << hatchbackAraba->getTip() << endl;
    cout << sedanAraba->getTip() << endl;

    delete hatchbackAraba;
    delete sedanAraba;

    return 0;
}
```

2.1.2 Abstract Factory Pattern

Because it looks similar to the Factory Method, we explain the difference through an example. The Factory Method leaves the object creation function to subclasses, whereas the Abstract Factory allows families of related objects to be created through an abstract class. In this way, users interact only with abstract classes and do not depend on concrete classes. The most important advantage is flexibility, because the dependencies in the code are reduced.

Example: A customized version of the previous example is used for the Abstract Factory: an abstract car class is used, with a different factory class for each type. The Abstract Factory is usually used when we create products that consist of several related objects, while the Factory Method is applied when we create one kind of object in several variants.

Code listing – Abstract Factory (part 1 of 2)

[*] Untitled2

```
1   #include <iostream>
2   using namespace std;
3
4   class Araba {
5   public:
6       virtual string getTip() = 0;
7   };
8
9   class Hatchback : public Araba {
10  public:
11      string getTip() {
12          return "Hatchback Tipinde Araba Seçildi.";
13      }
14  };
15
16  class Sedan : public Araba {
17  public:
18      string getTip() {
19          return "Sedan Tipinde Araba Seçildi.";
20      }
21  };
22
23  class ArabaFactory {
24  public:
25      virtual Araba* getAraba() = 0;
26  };
27
28  class HatchbackFactory : public ArabaFactory {
29  public:
30      Araba* getAraba() {
31          return new Hatchback();
32      }
33  };
34
35  class SedanFactory : public ArabaFactory {
36  public:
37      Araba* getAraba() {
38          return new Sedan();
39      }
40  };
```

Code listing – Abstract Factory (part 2 of 2)

```
41
42  int main() {
43      ArabaFactory* factory = new HatchbackFactory();
44      Araba* hatchbackAraba = factory->getAraba();
45      cout << hatchbackAraba->getTip() << endl;
46
47      delete factory;
48      delete hatchbackAraba;
49
50      factory = new SedanFactory();
51      Araba* sedanAraba = factory->getAraba();
52      cout << sedanAraba->getTip() << endl;
53
54      delete factory;
55      delete sedanAraba;
56
57      return 0;
58  }
59
```

2.1.3 Builder Pattern

When objects of the same class must be created with small differences, constructors can cause problems. The Builder lets us specify the properties of an object separately and create it step by step, which is its main advantage. On the other hand, the complexity of the code increases because several additional classes are needed.

Example: A good everyday example is a waiter taking orders in a restaurant: everyone wants their toast prepared differently, for example with different fillings such as sausage. We solve this problem with a builder, and the characteristics of the toast to be created are separated into the toast builder (TostBuilder) classes.

Code listing – Builder (part 1 of 4)

```
#include <iostream>
#include <string>

using namespace std;

class Tost {
public:
    void setEkmek(const string& ekmek) {
        ekmek_ = ekmek;
    }

    void setSos(const string& sos) {
        sos_ = sos;
    }

    void setIcMalzeme(const string& icMalzeme) {
        icMalzeme_ = icMalzeme;
    }

    void gosterTost() {
        cout << "Tost: " << ekmek_ << " ekmek, " << sos_ << " sos, " << icMalzeme_ << " iç malzeme\n";
    }

private:
    string ekmek_;
    string sos_;
    string icMalzeme_;
};
```

Code listing – Builder (part 2 of 4)

```
30  class TostBuilder {
31  public:
32      virtual ~TostBuilder() {}
33
34      Tost* getTost() {
35          return tost_;
36      }
37
38      void yeniTostOlustur() {
39          tost_ = new Tost();
40      }
41
42      virtual void ekmekKoy() = 0;
43      virtual void sosKoy() = 0;
44      virtual void icMalzemeKoy() = 0;
45
46  protected:
47      Tost* tost_;
48  };
49
50  class KlasikTostBuilder : public TostBuilder {
51  public:
52      void ekmekKoy() {
53          tost_->setEkmek("beyaz");
54      }
55
56      void sosKoy() {
57          tost_->setSos("mayonez");
58      }
59
60      void icMalzemeKoy() {
61          tost_->setIcMalzeme("ksar, sucuk, sosis");
62      }
63  };
```

Code listing – Builder (part 3 of 4)

```
65  class AciliTostBuilder : public TostBuilder {
66  public:
67      void ekmekKoy() {
68          tost_->setEkmek("kepekli");
69      }
70
71      void sosKoy() {
72          tost_->setSos("aci sos");
73      }
74
75      void icMalzemeKoy() {
76          tost_->setIcMalzeme("kasar, acili sucuk, yesil biber");
77      }
78  };
79
80  class TostUstasi {
81  public:
82      void setTostBuilder(TostBuilder* builder) {
83          tostBuilder_ = builder;
84      }
85
86      Tost* getTost() {
87          return tostBuilder_->getTost();
88      }
89
90      void tostHazirla() {
91          tostBuilder_->yeniTostOlustur();
92          tostBuilder_->ekmekKoy();
93          tostBuilder_->sosKoy();
94          tostBuilder_->icMalzemeKoy();
95      }
96
97  private:
98      TostBuilder* tostBuilder_;
99  };
```

Code listing – Builder (part 4 of 4)

```
101 int main() {
102     TostUstasi tostUstasi;
103     KlasikTostBuilder klasikTostBuilder;
104     AciliTostBuilder aciliTostBuilder;
105
106     tostUstasi.setTostBuilder(&klasikTostBuilder);
107     tostUstasi.tostHazirla();
108     Tost* klasikTost = tostUstasi.getTost
109
```

2.1.4 Singleton Pattern

It allows only one object of a class to be created and the same object to be used again and again in the rest of the code. The advantage is that no memory is wasted on unnecessary objects. The disadvantage is that it reduces the testability of the code.

Example: We block external access to the Singleton class by making its constructor private, and the object is created only through a function inside the class. Inside this function, the object is created only if it does not exist yet (that is, if the pointer is NULL). The instance pointer is a static member, so it must be defined outside the class, and the function that returns it must also be static.

Code listing – Singleton

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

class Singleton {
    private:
    static Singleton* instance;
    int data;

    Singleton() {
        data = 0;
    }

public:
    static Singleton* getInstance() {
        if (instance == NULL) {
            instance = new Singleton();
        }
        return instance;
    }

    int getData() {
        return data;
    }

    void setData(int value) {
        data = value;
    }
};

Singleton* Singleton::instance = NULL;
```

2.1.5 Prototype Pattern

Instead of creating new objects from scratch, it asks us to copy existing ones. Copying objects in the usual way may cause errors with some private members, so a copy constructor is used. The advantage is that complex objects can be produced quickly by copying, without depending on their classes. On the other hand, copying objects can sometimes be difficult.

Example: To keep the example as simple and understandable as possible, we use the soldiers of a game. After using the new operator only twice in the code, we can create 100 soldiers in the same way by copying them.

Code listing – Prototype

```cpp
#include <iostream>
#include <string>

using namespace std;

class Asker {
private:
    string isim;

public:
    Asker(string isim) : isim(isim) {}

    string getIsim() {
        return isim;
    }

    Asker* kopyaOlustur() {
        return new Asker(*this);
    }
};

int main() {
    Asker* orijinalAsker = new Asker("Ahmet");
    cout << "Orijinal Asker: " << orijinalAsker->getIsim() << endl;

    Asker* kopyaAsker1 = orijinalAsker->kopyaOlustur();
    cout << "Kopya Asker 1: " << kopyaAsker1->getIsim() << endl;

    Asker* kopyaAsker2 = orijinalAsker->kopyaOlustur();
    cout << "Kopya Asker 2: " << kopyaAsker2->getIsim() << endl;

    delete orijinalAsker;
    delete kopyaAsker1;
    delete kopyaAsker2;

    return 0;
}
```

2.1.6 Object Pool Pattern

As the name suggests, it works like a pool: the desired number of objects is created and kept in the pool, an object is handed out when it is requested and is returned to the pool when it is released. The advantage is that it saves us from excessive memory use and repeated object creation. The size of the pool must be set well; otherwise there is no point in using an object pool, because producing objects again takes time when the pool is empty.

Example: A bullet in a game is an object, and as many bullets as desired are created in advance and stored in a pool (a vector). A bullet is taken from the pool with the al() function, using the add and remove functions of the vector class. When the bullet is no longer needed, it is returned to the pool with the geriVer() function.

Code listing – Object Pool (part 1 of 2)

```
#include <iostream>
#include <vector>

using namespace std;
class Mermi {
public:
    void atesle() {
        cout << "Mermi ateslendi" << endl;
    }
};

class ObjectPool {
private:
    vector<Mermi*> havuz;
    int mks;

public:
    ObjectPool(int boyut) : mks(boyut) {
        for (int i = 0; i < mks; i++) {
            havuz.push_back(new Mermi());
        }
    }

    Mermi* al() {
        if (havuz.empty()) {
            return new Mermi();
        } else {
            Mermi* mermi = havuz.back();
            havuz.pop_back();
            return mermi;
        }
    }

    void geriVer(Mermi* mermi) {
        havuz.push_back(mermi);
    }
};
```

Code listing – Object Pool (part 2 of 2)

```
};

int main() {
    ObjectPool pool(10);

    Mermi* mermi1 = pool.al();
    mermi1->atesle();

    Mermi* mermi2 = pool.al();
    mermi2->atesle();

    pool.geriVer(mermi1);
    pool.geriVer(mermi2);

    return 0;
}
```

2.2. Structural Patterns

These patterns deal with the relationships and structures between classes and objects.

2.2.1 Adapter Pattern

As the name suggests, it allows two classes that are not compatible with each other to work together. It resolves problems that could otherwise break the application by adding an adapter between our code and a class that has a different interface.

Advantage: it ensures that classes that are different and incompatible, but need to work together, operate smoothly.

Disadvantage: converting between dissimilar classes may increase the processing time.

Example: There are two classes, one from our own code and one external class, and both must be used in the code. We used the Adapter pattern so that they do not conflict and cause an error: in main, we run the

adapter object created from the adapter class, which calls the function of the external class through its own function.

Code listing – Adapter

```cpp
#include <iostream>

class Kodunsinifi {
public:
    virtual void adaptr() = 0;
};

class Disardakisinif {
public:
    void eklenen() {
      cout << "Mevcut operasyon calisti." << endl;
    }
};

class Adaptor : public kodunsinifi {
private:
    Disardakisinif d_sinif;

public:
    void adaptr() {
        d_sinif.eklenen();
    }
};

int main() {
    Kodunsinifi* adaptor = new Adaptor();
    adaptor->adaptr();

    delete adaptor;

    return 0;
}
```

2.2.2 Bridge Pattern

Unlike the Adapter, it decouples an abstraction from its implementation. The Bridge pattern reduces the dependency between the abstraction and the implementation classes that comes from an "is-a" (inheritance) relationship, by connecting them through a "has-a" (composition) relationship, which provides flexibility. Its advantage is that the abstraction and the implementation can be separated and become independent of each other. However, since two separate abstractions are created, the code complexity increases.

Example: The shape (Sekil) is an abstract class from which the square and circle classes inherit. In the same way, the implementation (application interface) is an abstract class with its own square and circle classes.

The Bridge pattern connects the shape and the implementation, so a new shape or a new implementation can be added easily and independently.

Code listing – Bridge (part 1 of 3)

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



class UygulamaArayuzu {
public:
    virtual void ciz() = 0;
};

class Kare : public UygulamaArayuzu {
public:
    void ciz() {
        cout << "Kare çizildi." << endl;
    }
};

class Daire : public UygulamaArayuzu {
public:
    void ciz() {
        cout << "Daire çizildi." << endl;
    }
};
```

Code listing – Bridge (part 2 of 3)

```
25  class Sekil {
26  protected:
27      UygulamaArayuzu* uygulamaArayuzu;
28
29  public:
30      Sekil(UygulamaArayuzu* arayuz) : uygulamaArayuzu(arayuz) {}
31
32      virtual void ciz() = 0;
33  };
34
35  class KareSekli : public Sekil {
36  public:
37      KareSekli(UygulamaArayuzu* arayuz) : Sekil(arayuz) {}
38
39      void ciz() {
40          uygulamaArayuzu->ciz();
41      }
42  };
43
44  class DaireSekli : public Sekil {
45  public:
46      DaireSekli(UygulamaArayuzu* arayuz) : Sekil(arayuz) {}
47
48      void ciz() {
49          uygulamaArayuzu->ciz();
50      }
51  };
```

Code listing – Bridge (part 3 of 3)

```
53  int main() {
54      UygulamaArayuzu* kareArayuzu = new Kare();
55      Sekil* kareSekli = new KareSekli(kareArayuzu);
56      kareSekli->ciz();
57
58      UygulamaArayuzu* daireArayuzu = new Daire();
59      Sekil* daireSekli = new DaireSekli(daireArayuzu);
60      daireSekli->ciz();
61
62
63      return 0;
64  }
65
```

2.2.3 Composite Pattern

It allows objects to be combined into a hierarchy (a tree structure). The advantage of this pattern is that, because they implement the same interface, the client can manage derived classes and container objects in a single way. In addition, using leaf objects and container objects together makes it easier to create complex structures and provides structural flexibility. The individual objects are usually called "leaves".

Example: A shape group (SekilGrubu) is a composite structure that combines classes with different names that perform the same function into a single class and presents them as a whole. A recursive function with a for loop allows access to all the leaves of the structure.

Code listing – Composite (part 1 of 2)

```
using namespace std;
class Sekil {
public:
    virtual void ciz() = 0;
};

class Kare : public Sekil {
public:
    void ciz() {
        cout << "Kare çizildi." << endl;
    }
};

class Daire : public Sekil {
public:
    void ciz() {
    cout < < "Daire çizildi." << endl;
    }
};
```

Code listing – Composite (part 2 of 2)

```
23
24  class SekilGrubu : public Sekil {
25  private:
26      string ad;
27  vector<Sekil*> sekiller;
28
29  public:
30      SekilGrubu( string& isim) : ad(isim) {}
31
32      void ciz() {
33          cout  << "Sekil Grubu: " << ad << stndl;
34
35          for (Sekil* sekil : sekiller) {
36              sekil->ciz();
37          }
38      }
39
40      void ekle(Sekil* sekil) {
41          sekiller.push_back(sekil);
42      }
43
44      void sil(Sekil* sekil) {
45      }
46  };
47
48  int main() {
49      Sekil* kare = new Kare();
50      Sekil* daire = new Daire();
51
52      SekilGrubu* grup = new SekilGrubu("Ana Grup");
53      grup->ekle(kare);
54      grup->ekle(daire);
55
56      grup->ciz();
57
58      delete kare;
59      delete daire;
60      delete grup;
61
62      return 0;
```

2.2.4 Decorator Pattern

Without changing the structure of the class, it can add and remove features of an existing object while the program is running, and it stores the existing object within itself. Each decorator class has a single responsibility, which makes the code clearer and easier to understand, and it allows new features to be added to an object dynamically at runtime. The disadvantage is that a decorator class must be created for each new feature, and because behavior can be changed dynamically, the code can become harder to understand and follow.

Example: There is an abstract car class and a standard car class with normal features. After a car is first created as standard, it can be made luxury or fast: if the customer wants to add a feature, the car can be customized through the decorator classes of the car customizer.

Code listing – Decorator (part 1 of 2)

```
#include <iostream>
#include <string>

using namespace std;

class Araba {
public:
    virtual string getModel() const = 0;
    virtual double getPrice() const = 0;
};

class StandartAraba : public Araba {
public:
    string getModel() {
        return "Standart Araba";
    }

    double getPrice() {
        return 50000.0;
    }
};

class ArabaOzellikEkleyici : public Araba {
protected:
    Araba* araba;

public:
    ArabaOzellikEkleyici(Araba* araba) : araba(araba) {}

    string getModel()  {
        return araba->getModel();
    }

    double getPrice() {
        return araba->getPrice();
    }
};
```

Code listing – Decorator (part 2 of 2)

```
class HizliAraba : public ArabaOzellikEkleyici {
public:
    HizliAraba(Araba* araba) : ArabaOzellikEkleyici(araba) {}

    string getModel() {
        return araba->getModel() + ", Hizli";
    }

    double getPrice() {
        return araba->getPrice() + 10000.0;
    }
};

class LuksAraba : public ArabaOzellikEkleyici {
public:
    LuksAraba(Araba* araba) : ArabaOzellikEkleyici(araba) {}

    string getModel() {
        return araba->getModel() + ", Lüks";
    }

    double getPrice(){
        return araba->getPrice() + 20000.0;
    }
};
```

2.2.5 Facade Pattern

It allows us to use several complex classes through a single, simple structure.

Advantage: it reduces the amount of code and repetition for the client, and thus reduces the cost.

Disadvantage: changes in the underlying classes can affect the Facade, and it can limit the flexibility of the code.

Example: We turned a shopping center into a Facade by combining complex operations (security, payment, inventory) under one class, which makes things convenient for the client.

Code listing – Facade (part 1 of 2)

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

class GuvenlikSistemi {
public:
    void kontrolEt() {
        cout << "Güvenlik sistemi kontrol ediliyor." << endl;
    }
};

class OdemeSistemi {
public:
    void odemeYap() {
        cout << "Ödeme sistemi ile ödeme yapýlýyor." << endl;
    }
};

class EnvanterSistemi {
public:
    void stokKontrolu() {
        cout << "Envanter yönetim sistemi stoklarý kontrol ediyor." << endl;
    }
};
```

Code listing – Facade (part 2 of 2)

```
26  class AlisverisMerkezi {
27  private:
28      GuvenlikSistemi* guvenlikSistemi;
29      OdemeSistemi* odemeSistemi;
30      EnvanterSistemi* envanterSistemi;
31
32  public:
33      AlisverisMerkezi() {
34          guvenlikSistemi = new GuvenlikSistemi();
35          odemeSistemi = new OdemeSistemi();
36          envanterSistemi = new EnvanterSistemi();
37      }
38
39        ~AlisverisMerkezi() {
40          delete guvenlikSistemi;
41          delete odemeSistemi;
42          delete envanterSistemi;
43      }
44
45
46      void alisverisYap() {
47          cout << "alisveris yapiliyor:" << endl;
48          guvenlikSistemi->kontrolEt();
49          odemeSistemi->odemeYap();
50          envanterYonetimSistemi->stokKontrolu();
51      }
52  };
53
54
55  int main() {
56      AlisverisMerkezi alisverisMerkezi;
57
58      alisverisMerkezi.alisverisYap();
59
60      return 0;
61  }
62
```

2.2.6 Flyweight Pattern

When similar objects are needed, it does not create new objects; it lets existing objects be shared and reused. There are two important parts: the intrinsic state is the part shared by the objects, and the extrinsic state is the part that varies and is stored separately.

Advantage: it saves memory.

Disadvantage: sharing objects can increase the complexity of the application.

Related patterns: Because it is similar to the Object Pool, the most important difference should be kept in mind: the Object Pool only stores and hands out objects, whereas the Flyweight manages the shared and the variable properties of objects. Another difference is that in an object pool the desired number of objects is created at the beginning and then used, whereas in the Flyweight individual objects are not created for each user; all users share a common object.

Example: The bullet factory in the Flyweight example should not be confused with the pool we wrote earlier. Although it is similar to the Object Pool, the working logic is different: if an object of the requested type already exists, it is taken and returned; if it does not exist, it is created and stored in the requested type while the program is running.

Code listing – Flyweight (part 1 of 2)

```
4   using namespace std;
5
6   class Mermi {
7   public:
8       virtual void atesle() = 0;
9   };
10
11  class GercekMermi : public Mermi {
12  public:
13      void atesle() {
14          cout << "Gerçek mermi ateslendi" << endl;
15      }
16  };
17
18  class HavaMermisi : public Mermi {
19  public:
20      void atesle() {
21          cout << "Hava mermi ateslendi" << endl;
22      }
23  };
24
25  class MermiFabrikasi {
26  private:
27      vector<Mermi*> havuz;
28
29  public:
30      Mermi* al(string tür) {
31          for (Mermi* mermi : havuz) {
32              if (mermi->getType() == tür) {
33                  havuz.erase(mermi);
34                  return mermi;
35              }
36          }
37
38          Mermi* mermi;
39          if (tür == "gerçek") {
40              mermi = new GercekMermi();
41          } else if (tür == "hava") {
42              mermi = new HavaMermisi();
43          } else {
44              cout << "Geçersiz mermi türü" << endl;
45              return NULL;
46          }
47
48          return mermi;
49      }
50
51      void geriVer(Mermi* mermi) {
52          havuz.push_back(mermi);
53      }
54  };
```

Code listing – Flyweight (part 2 of 2)

```
int main() {
    MermiFabrikasi mermiFabrikasi;

    Mermi* mermi1 = mermiFabrikasi.al("gerçek");
    mermi1->atesle();

    Mermi* mermi2 = mermiFabrikasi.al("hava");
    mermi2->atesle();

    mermiFabrikasi.geriVer(mermi1);
    mermiFabrikasi.geriVer(mermi2);

    return 0;
}
```

2.2.7 Proxy Pattern

It acts as an intermediary for the real object and allows us to see or use only the features we want, instead of accessing an object that has many features. The Decorator and the Proxy can be confused: the Decorator is used to add new behavior to an object dynamically, whereas the Proxy controls access to the real object. Its advantage is that it controls access to the object, so resource usage can be optimized.

Disadvantage: it may affect performance, because extra code is used to access the object.

Example: When we want to drive a car, we may first want to check whether the car exists. We use a proxy car for this: after the request is sent to the proxy car, it creates the real car and performs the driving operation only when necessary. In this way, we increase performance by preventing unnecessary creation and

operations, and we use resources effectively. To avoid confusion: the proxy only controls access to the real car, and it is the real car that is actually driven.

Code listing – Proxy

```
public:
    virtual void sur() = 0;
};

class GercekAraba : public Araba {
private:
    string model;

public:
    GercekAraba(string model) : model(model) {}

    void sur() {
        cout << model << " arabas sürülüyor." << endl;
    }
};

class ProxyAraba : public Araba {
private:
    GercekAraba* gercekAraba;
    string model;

public:
    ProxyAraba(string model) : model(model), gercekAraba(NULL) {}

    void sur() {
        if (gercekAraba == NULL) {
            gercekAraba = new GercekAraba(model);
        }
        cout << "Proxy " << model << " araba denetleniyor." << endl;
        gercekAraba->sur();
    }
};

int main() {
    ProxyAraba proxy("BMW");
    proxy.sur();

    return 0;
}
```

2.3. Behavioral Patterns

These patterns deal with the communication between classes and objects and allow this communication to be organized in more flexible structures.

2.3.1 Chain of Responsibility Pattern

It decouples the classes from each other. In short, a request is passed to the connected classes one after another: each class performs its operation and then passes the request on to the next linked class. In other words, it works like a control mechanism, and the classes are loosely coupled. It is usually used where there would otherwise be many if-else blocks.

> **Advantages:** each link of the chain is independent and can be changed; it is easy to add or remove a processing step; and since each class performs a single function, the code is easy to maintain and test.

Disadvantages: each processor in the chain is checked in order to handle the request, which can affect performance when the request has to travel to the end of the chain; and if the operation did not take place in any of the classes, it may be difficult to find out why.

Example: The bank transaction classes (for example withdrawal, deposit and balance inquiry) are linked to each other through a request function. The next pointer being NULL works as an end-of-chain flag: it tells the program that the chain of transactions has ended.

Code listing – Chain of Responsibility (part 1 of 2)

```
class BankaIslemi {
protected:
    BankaIslemi* sonrakiIslem;

public:
    BankaIslemi() : sonrakiIslem(NULL) {}

      void setSonrakiIslem(BankaIslemi* islem) {
        sonrakiIslem = islem;
    }

   virtual void talebiIsle(string talep) = 0;
};

class ParaCekme : public BankaIslemi {
public:
    void talebiIsle(string talep) {
        if (talep == "paraCekme ") {
            cout << "Para çekme islemi gerçeklestiriliyor." << endl;
        }
         else if (sonrakiIslem != NULL)  {
            sonrakiIslem->talebiIsle(talep);
         }
    }
};

 class ParaYatirma : public BankaIslemi {
public:
    void talebiIsle(string talep) {
        if (talep == "paraYatirma") {
            cout << "Para yatirma islemi gerçeklestiriliyor." << endl;
        }
         else if (sonrakiIslem != NULL) {
            sonrakiIslem->talebiIsle(talep);
        }
    }
};
```

Code listing – Chain of Responsibility (part 2 of 2)

```
42
43  class BakiyeSorgulama : public BankaIslemi {
44  public:
45      void talebiIsle(string talep) {
46          if (talep == "bakiyeSorgulama") {
47              cout << "Bakiye sorgulama islemi gerçeklestiriliyor." << endl;
48          }
49          else if (sonrakiIslem != NULL)   {
50              sonrakiIslem->talebiIsle(talep);
51           }
52      }
53  };
54
55  int main() {
56      BankaIslemi* paraCekme = new ParaCekme();
57      BankaIslemi* paraYatirma = new ParaYatirma();
58      BankaIslemi* bakiyeSorgulama = new BakiyeSorgulama();
59
60      paraCekme->setSonrakiIslem(paraYatirma);
61      paraYatirma->setSonrakiIslem(bakiyeSorgulama);
62
63      paraCekme->talebiIsle("paraYatirma");
64      paraCekme->talebiIsle("bakiyeSorgulama");
65      paraCekme->talebiIsle("paraCekme");
66      return 0;
67  }
68
```

2.3.2 Command Pattern

It turns each operation into a separate class.

Advantages: it separates the classes from each other and makes operations independent, and new operations are easy to add.

Disadvantage: a class is created for each operation, which can increase the cost for the program.

Related patterns: In the Chain of Responsibility pattern, requests move along the chain and each object checks whether it can process the request; in the Command pattern, the client sends the request directly to the class that handles it.

Example: Unlike Chain of Responsibility, every branch of the if-else logic is created as a class (for example food order and drink order). The operations are passed by the client to the required class, and the necessary work is performed inside that class.

Code listing – Command (part 1 of 2)

```
class Islem {
public:
    virtual void gerceklestir() = 0;
};

class YemekSiparisi : public Islem {
private:
    string yemekAdi;

public:
    YemekSiparisi(string yemekAdi) : yemekAdi(yemekAdi) {}

    void gerceklestir() {
        cout << "Yemek  alindi: " << yemekAdi << endl;
    }
};

 class IcecekSiparisi : public Islem {
private:
    string icecekAdi;

public:
    IcecekSiparisi(string icecekAdi) : icecekAdi(icecekAdi) {}

    void gerceklestir() {
        cout << "icecek   alindi: " << icecekAdi << endl;
     }
};
```

Code listing – Command (part 2 of 2)

```
35
36    class KomutYurutucu {
37  private:
38      vector<Islem*> komutlar;
39
40  public:
41      void komutEkle(Islem* komut) {
42          komutlar.push_back(komut);
43      }
44
45      void komutlariCalistir() {
46          for (Islem* komut : komutlar) {
47              komut->gerceklestir();
48          }
49      }
50  };
51
52  int main() {
53
54      Islem* yemekSiparisi = new YemekSiparisi("Pizza");
55      Islem* icecekSiparisi = new IcecekSiparisi("Kola");
56
57      KomutYurutucu yurutucu;
58      yurutucu.komutEkle(yemekSiparisi);
59      yurutucu.komutEkle(icecekSiparisi);
60
61      yurutucu.komutlariCalistir();
62
63      return 0;
64  }
65
```

2.3.3 Template Method Pattern

A template of an algorithm is created in the base class, and derived classes can change certain steps of this algorithm without changing its structure.

Advantages: it saves code repetition, and individual steps can be changed smoothly and easily.

Disadvantage: the code is less flexible, because the derived classes are tied to the base class and can only customize certain steps.

Example: A pizza template is defined in the base Pizza class, with steps such as preparing the dough, adding ingredients, baking and serving. The ItalianPizza and TurkPizza classes are derived from this template. Since

both classes inherit from Pizza, they can customize the functions of the base class, here the step that adds the ingredients.

Code listing – Template Method (part 1 of 2)

```
class Pizza {
public:
    void hazirla() {
        hamurHazirla();
        malzemeleriEkle();
        pisir();
        servisEt();
    }

    virtual void hamurHazirla() {
        cout << "Hamur hazilanir "<<endl;
    }

    virtual void malzemeleriEkle() {
        cout << "Malzemeler ekleniyor." << endl;
    }

    virtual void pisir() {
        cout << "Pisiriliyor." << endl;
    }

    void servisEt() {
        cout << "Pizza servis ediliyor." << endl;
    }
};
```

Code listing – Template Method (part 2 of 2)

```
30
31  class ItalyanPizza : public Pizza {
32  public:
33      void malzemeleriEkle() {
34          cout << "Italyan usulu malzemeler ekleniyor." << endl;
35      }
36  };
37
38  class TurkPizza : public Pizza {
39  public:
40      void malzemeleriEkle() {
41          cout << "Turk usulu malzemeler ekleniyor." << endl;
42      }
43  };
44
45  int main() {
46      Pizza* italyanPizza = new ItalyanPizza();
47      italyanPizza->hazirla();
48
49      cout << endl;
50
51      Pizza* turkPizza = new TurkPizza();
52      turkPizza->hazirla();
53
54      return 0;
55  }
56
```

2.3.4 Iterator Pattern

It is used to navigate through a collection and access the values in it. The Iterator hides the internal structure of the collection and provides an interface for accessing its elements.

Advantages: it gives flexibility to the code by abstracting the internal structure and simplifying data access.

Disadvantages: when the structure of the collection changes, the iterator must be updated, and it uses more memory because extra objects are used.

Example: A list is created using the ConcreteAggregate class and items are added. An Iterator object is created by calling CreateIterator(). Then, using the iterator in a while loop, the collection is traversed and

each item is printed on the screen. The most important benefit is that we performed this operation with the iterator class, without depending on the main class.

Code listing – Iterator (part 1 of 3)

```
 8  class Aggregate {
 9  public:
10      virtual Iterator* CreateIterator() = 0;
11  };
12
13  class ConcreteAggregate : public Aggregate {
14  private:
15      vector<string> _items;
16
17  public:
18      Iterator* CreateIterator();
19
20      int Count() {
21          return _items.size();
22      }
23
24      string& operator[](int index) {
25          return _items[index];
26      }
27  };
28
29  class Iterator {
30  public:
31      virtual string First() = 0;
32      virtual string Next() = 0;
33      virtual bool IsDone() = 0;
34      virtual string CurrentItem() = 0;
35  };
```

Code listing – Iterator (part 2 of 3)

```
class Iterator {
public:
    virtual string First() = 0;
    virtual string Next() = 0;
    virtual bool IsDone() = 0;
    virtual string CurrentItem() = 0;
};

class ConcreteIterator : public Iterator {
private:
    ConcreteAggregate* _aggregate;
    int _current;

public:
    ConcreteIterator(ConcreteAggregate* aggregate)
        : _aggregate(aggregate), _current(0) {}

    string First() {
        return (*_aggregate)[0];
    }

    string Next() {
        string ret;
        if (_current < _aggregate->Count() - 1) {
            ret = (*_aggregate)[++_current];
        }
        return ret;
    }

    string CurrentItem() {
        return (*_aggregate)[_current];
    }

    bool IsDone() {
        return _current >= _aggregate->Count();
    }
};
```

Code listing – Iterator (part 3 of 3)

```
66
67  Iterator* ConcreteAggregate::CreateIterator() {
68      return new ConcreteIterator(this);
69  }
70
71  int main() {
72      ConcreteAggregate a;
73      a[0] = "Item A";
74      a[1] = "Item B";
75      a[2] = "Item C";
76      a[3] = "Item D";
77
78      Iterator* i = a.CreateIterator();
79
80      cout << "Iterating over collection:" << endl;
81
82      string item = i->First();
83      while (!i->IsDone()) {
84          cout << item << endl;
85          item = i->Next();
86      }
87
88      delete i;
89
90      return 0;
91  }
```

2.3.5 Mediator Pattern

It limits direct communication between objects and forces them to communicate only through a mediator object. The advantage is that it provides flexibility by establishing loose coupling between objects.

Disadvantage: a single class (the mediator) can end up containing all of the logic.

Related patterns: It is similar to the Observer pattern; however, the Observer pattern provides communication between objects that are independent of each other, while the Mediator pattern manages the interaction between objects at a central point.

Example: A bank account object is created. Then customer objects are created and added to the bank account. The customers make deposits and withdrawals, and these transactions update the balance after the Mediator class has checked them.

Code listing – Mediator (part 1 of 2)

```
class BankaHesabi;

class Musteri {
protected:
    BankaHesabi* bankaHesabi;

public:
    Musteri(BankaHesabi* bankaHesabi) : bankaHesabi(bankaHesabi) {}

    virtual void paraYatir(double miktar) = 0;
    virtual void paraCek(double miktar) = 0;
    virtual void hesapBakiyesi(double bakiye) = 0;
};

class BireyselMusteri : public Musteri {
public:
    BireyselMusteri(BankaHesabi* bankaHesabi) : Musteri(bankaHesabi) {}

    void paraYatir(double miktar) {
        cout << "Bireysel  para yatiriyor: " << miktar << " TL" << endl;
        bankaHesabi->bildirimGonder(this, miktar);
    }

    void paraCek(double miktar) {
        cout << "Bireysel  para çekiyor: " << miktar << " TL" << endl;
        bankaHesabi->bildirimGonder(this, -miktar);
    }

    void hesapBakiyesi(double bakiye) {
        cout << "Bireysel  hesap bakiyesi güncellendi: " << bakiye << " TL" << endl;
    }
};

class KurumsalMusteri : public Musteri {
public:
    KurumsalMusteri(BankaHesabi* bankaHesabi) : Musteri(bankaHesabi) {}

    void paraYatir(double miktar) {
        cout << "Kurumsal  para yatiriyor: " << miktar << " TL" << endl;
        bankaHesabi->bildirimGonder(this, miktar);
    }

    void paraCek(double miktar) {
        cout << "Kurumsal  para çekiyor: " << miktar << " TL" << endl;
        bankaHesabi->bildirimGonder(this, -miktar);
    }

    void hesapBakiyesi(double bakiye) {
        cout << "Kurumsal  hesap bakiyesi güncellendi: " << bakiye << " TL" << endl;
    }
};
```

Code listing – Mediator (part 2 of 2)

```
58
59  class mediator {
60  private:
61      vector<Musteri*> musteriler;
62
63  public:
64      void musteriEkle(Musteri* musteri) {
65          musteriler.push_back(musteri)
66      }
67
68      void bildirimGonder(Musteri* gonderen, double miktar) {
69          for (Musteri* alici : musteriler) {
70              if (alici != gonderen) {
71                  alici->hesapBakiyesi(miktar);
72              }
73          }
74      }
75  };
76
77  int main() {
78      mediator mediatorr;
79
80      BireyselMusteri bireyselMusteri1(&bankaHesabi);
81      BireyselMusteri bireyselMusteri2(&bankaHesabi);
82      KurumsalMusteri kurumsalMusteri(&bankaHesabi);
83
84      mediatorr.musteriEkle(&bireyselMusteri1);
85      mediatorr.musteriEkle(&bireyselMusteri2);
86      mediatorr.musteriEkle(&kurumsalMusteri);
87
88      bireyselMusteri1.paraYatir(1000);
89      bireyselMusteri2.paraCek(500);
90      kurumsalMusteri.paraYatir(2000);
91
92      return 0;
93  }
```

2.3.6 Memento Pattern

A snapshot of an object is saved so that its existing information can be restored later.

Advantage: it allows the object to return to its past states.

Disadvantage: memory consumption can be high, because an exact copy of each state is kept.

Example: The restore function of the EditorMomento class restores the text to its previous state. The TextEditor class holds the current state, and the EditorMomento class holds the previous state, so the editor can return to the previous state when requested.

Code listing – Memento (part 1 of 2)

```
class EditorMomento {
private:
    string text;

public:
    EditorMomento(const string& text) : text(text) {}

    string getText() const {
        return text;
    }
};

class TextEditor {
private:
    string text;

public:
    void setText(const string& text) {
        this->text = text;
    }

    string getText() const {
        return text;
    }

    EditorMomento createMomento() const {
        return EditorMomento(text);
    }

    void restoreromMomento(const EditorMomento& momento) {
        text = momento.getText();
    }
};
```

Code listing – Memento (part 2 of 2)

```
40
41  int main() {
42      TextEditor editor;
43
44      editor.setText("Bu bir metin");
45
46      EditorMomento momento = editor.createMomento();
47
48      editor.setText("Bu  metin günceldir");
49
50      editor.restorefromMomento(momento);
51
52      cout << "Geri al metin: " << editor.getText() << endl;
53
54      return 0;
55  }
56
```

2.3.7 Null Object Pattern

It is used to represent the "null" state of a class in code and reduces the number of null checks in the application: it returns a specific object that does nothing when the object reference would otherwise be null.

Advantages: it removes null errors and repeated null checks, so the code works without such errors.

Disadvantage: it can increase memory usage when null objects are created.

Example: A log is a computing term that can be thought of as a diary that records what happens in a program. Here, the NullLogger does not produce any output when it is called, so even when no real logger is available, the code runs without null errors.

Code listing – Null Object

```
class ILogger {
public:
    virtual void Log(const string& message) = 0;
};

class ConsoleLogger : public ILogger {
public:
    void Log(const string& message) {
        cout << "Console Log: " << message << endl;
    }
};

class NullLogger : public ILogger {
public:
    void Log(const sring& message) {
        ////
        ////
        ////
        ////
        }

};

int main() {
    ILogger* logger = new ConsoleLogger();
    logger->Log("Log message 1");

    ILogger* nullLogger = new NullLogger();
    nullLogger->Log("Log message 2");

    delete nullLogger;
    delete logger;
    return 0;
}
```

2.3.8 Observer Pattern

It provides the connection between classes that are decoupled from each other. It can be remembered with the phrase "one to many".

Advantage: it reduces the dependency between objects, which makes the code more loosely coupled.

Disadvantage: constantly checking for state changes can affect the performance of the application.

Related patterns: It is similar to the Mediator pattern.

Example: The observer class receives changes from the Subject class through its update() function. When the data changes, the observer can show the new value immediately thanks to the update function.

Code listing – Observer (part 1 of 2)

```
class Observer {
public:
    virtual void update(int data) = 0;
};

class Subject {
private:
    int data;
    vector<Observer*> observers;

public:
    void attach(Observer* observer) {
        observers.push_back(observer);
    }

    void detach(Observer* observer) {
        for (int i = 0; i < observers.size(); ++i) {
            if (observers[i] == observer) {
                observers.erase(observers.begin() + i);
                break;
            }
        }
    }

    void notify() {

        for (int i = 0; i < observers.size(); ++i) {
            observers[i]->update(data);
        }
    }

    void setData(int value) {
        data = value;
        notify();
    }
};

class ConcreteObserver : public Observer {
public:
    void update(int data) {
        cout << "Güncel veri: " << data << endl;
    }
};
```

Code listing – Observer (part 2 of 2)

```
50
51  int main() {
52      Subject subject;
53
54      ConcreteObserver observer1;
55      ConcreteObserver observer2;
56
57      subject.attach(&observer1);
58      subject.attach(&observer2);
59
60      subject.setData(10);
61
62      subject.detach(&observer1);
63
64      subject.setData(20);
65
66      return 0;
67  }
68
```

2.3.9 State Pattern

It allows an object to change its behavior at runtime when its state changes.

Advantages: state-specific behaviors can be added and changed easily, and it applies the single responsibility principle by reducing dependencies.

Disadvantage: managing shared data can become difficult.

Example: It seems that there are four different situations, but there are actually only two states: the lamp is on or off. When the button is pressed while the lamp is on, it turns off; when it is pressed while the lamp is

off, it turns on. When the same function is run twice, the lamp does not switch again, because it is already in that state.

Code listing – State (part 1 of 2)

```
class State {
public:
    virtual void stateDegistir(class Lamba* lamba) = 0;
};

class AcikState : public State {
public:
    void stateDegistir(class Lamba* lamba);
};

class KapaliState : public State {
public:
    void stateDegistir(class Lamba* lamba);
};

class Lamba {
private:
    State* suankiState;

public:
    Lamba() {
        suankiState = new KapaliState();
    }

    void stateDegistir(State* newState) {
        suankiState = newState;
    }

    void ac() {
        suankiState->stateDegistir(this);
    }

    void kapat() {
        suankiState->stateDegistir(this);
    }
};

void AcikState::stateDegistir(Lamba* lamba) {
    cout << "Lamba kapatiliyor." << endl;
    lamba->stateDegistir(new KapaliState());
}

void KapaliState::stateDegistir(Lamba* lamba) {
    cout << "Lamba aciliyor." << endl;
    lamba->stateDegistir(new AcikState());
}
```

Code listing – State (part 2 of 2)

```
int main() {
    Lamba lamba;

    lamba.ac();
    lamba.ac();
    lamba.kapat();
    lamba.kapat();

    return 0;
}
```

2.3.10 Strategy Pattern

Multiple algorithms that can do the same job are placed in different classes, and we can choose the algorithm we want at runtime, so the code does not need to be recompiled.

Disadvantage: extra classes are required to represent the different strategies, which can slightly increase the code base.

Example: By changing the calculator code a little and adding strategy classes, the calculation can be performed with a method chosen from three different calculation strategies (addition, subtraction and multiplication) when the software is started.

Code listing – Strategy

```
class CalculationStrategy {
public:
    virtual int calculate(int a, int b) = 0;
};

class AddStrategy : public CalculationStrategy {
public:
    int calculate(int a, int b) {
        return a + b;
    }
};

class SubtractStrategy : public CalculationStrategy {
public:
    int calculate(int a, int b) {
        return a - b;
    }
};

class MultiplyStrategy : public CalculationStrategy {
public:
    int calculate(int a, int b) {
        return a * b;
    }
};

class Calculator {
private:
    CalculationStrategy* strategy;

public:
    void setStrategy(CalculationStrategy* strategy) {
        this->strategy = strategy;
    }

    int calculate(int a, int b) {
        return strategy->calculate(a, b);
    }
};
```

2.3.11 Visitor Pattern

It is used only to add new functions to classes, without making changes inside them. It is usually called double dispatch.

Advantages: it allows new functions to be added without changing the object structure, and the classes do not depend on each other's internal structure.

Disadvantage: if the classes change continuously, the visitor class may need to be updated constantly, which increases the maintenance cost.

Related patterns: The Strategy pattern is similar, but it adds different algorithms through separate classes, whereas the Visitor pattern adds functions to classes.

Example: In main, the accept function is called with a visitor object and a car object. In this function, the car passes itself to the visitor, and the desired functions are performed each time the visitor class is called.

Code listing – Visitor

```
8
9  class ArabaVisitor {
0  public:
1      virtual void visit(Araba* araba) = 0;
2  };
3
4  class Araba {
5  public:
6      virtual void accept(ArabaVisitor* visitor) = 0;
7  };
8
9  class Sedan : public Araba {
0  public:
1      void accept(ArabaVisitor* visitor) {
2          visitor->visit(this);
3      }
4  };
5
6  class SUV : public Araba {
7  public:
8      void accept(ArabaVisitor* visitor) {
9          visitor->visit(this);
0      }
1  };
2
3  class ArabaVisitorImpl : public ArabaVisitor {
4  public:
5      void visit(Araba* araba) {
6          cout << "Araba ziyaret edildi." << endl;
7      }
8  };
9
```

2.3.12 Interpreter Pattern

It allows certain expressions to be interpreted. This pattern uses an expression tree structure that represents the grammatical rules of an expression, and it interprets and processes the expression by traversing this tree.

Advantages: it provides dynamism and flexibility because it allows expressions to be interpreted, and it is easy to add new rules.

Disadvantages: creating the structure of the expression is complex, and since a tree is created, it must be traversed at every evaluation, which can be costly.

Example: A mathematical expression given as a string is parsed into expression objects using a stack, converted to an integer value and evaluated.

Code listing – Interpreter (part 1 of 2)

```
class Expression {
public:
    virtual int interpret() = 0;
};

class NumberExpression : public Expression {
private:
    int number;

public:
    NumberExpression(int num) : number(num) {}

    int interpret() {
        return number;
    }
};

class AdditionExpression : public Expression {
private:
    Expression* left;
    Expression* right;

public:
    AdditionExpression(Expression* l, Expression* r) : left(l), right(r) {}

    int interpret() {
        return left->interpret() + right->interpret();
    }
};

class Calculator {
private:
    std::stack<Expression*> expressionStack;
```

Code listing – Interpreter (part 2 of 2)

```
39  public:
40      void parse(std::string input) {
41          for (int i = 0; i < input.length(); ++i) {
42              if (isdigit(input[i])) {
43                  int num = input[i] - '0';
44                  Expression* numberExpression = new NumberExpression(num);
45                  expressionStack.push(numberExpression);
46              } else if (input[i] == '+') {
47                  Expression* right = expressionStack.top();
48                  expressionStack.pop();
49                  Expression* left = expressionStack.top();
50                  expressionStack.pop();
51                  Expression* additionExpression = new AdditionExpression(left, right);
52                  expressionStack.push(additionExpression);
53              }
54          }
55      }
56
57      int calculate() {
58          if (!expressionStack.empty()) {
59              Expression* expression = expressionStack.top();
60              expressionStack.pop();
61              return expression->interpret();
62          }
63          return 0;
64      }
65  };
66
67  int main() {
68      Calculator calculator;
69      calculator.parse("5+2+1");
70      int result = calculator.calculate();
71      std::cout << "Result: " << result << std::endl;
72
73      return 0;
74  }
75
```

3. Anti-Patterns

Unlike design patterns, anti-patterns do more harm than good in code. They are common mistakes that can lead to negative consequences. At first there seems to be no problem, but later they may cause serious errors in the code. These mistakes are usually caused by inexperience, time pressure, misunderstanding, bad design or bad management. The advantage of knowing them is that, if they are detected early, the application benefits greatly in terms of code health and sustainability. The disadvantage is that, if they are not known beforehand and not corrected, they can cause serious trouble. They are divided into three types:

- **Architectural Anti-patterns:** These are anti-patterns that occur at the scale of whole systems and are usually caused by design errors.
- **Programming Anti-patterns:** These refer to errors made during the software coding process.
- **Project Management Anti-patterns:** These refer to errors made during the project management process.

4. Conclusion

When creating software, the SOLID principles mentioned at the beginning should definitely be applied, and design patterns and knowledge of anti-patterns should be used to write better, more flexible and more sustainable code. When we use design patterns in the right places, our code becomes more readable and

easier to maintain. In addition, if we know the anti-patterns and avoid writing structures similar to them, our code will be better. Finally, if we know both concepts well and use them in our code, we will always be one step ahead in software development.