Design Pattern

bqrm

a bqrm's work

BQRM

a bqrm's work

目 录

第1章 设计模式的六大原则 1
1.1 单一职责原则 1
1.2 里氏替换原则 1
1.3 依赖倒转原则 2
1.4 接口隔离原则 2
1.5 迪米特法则 2
1.6 合成复用原则 2

第一部分 创建型模式 3

第2章 单例模式 4
2.1 UML图 4
2.2 优点 4
2.3 缺点 4
2.4 应用场景 5
2.5 实例 5
2.5.1 C++ 5
2.5.2 Python 7
第3章 工厂模式 9
3.1 简单工厂模式 9
3.1.1 UML图 9
3.1.2 优点 10
3.1.3 缺点 10
3.1.4 应用场景 10
3.1.5 实例 11
3.2 工厂方法 16
3.2.1 UML图 16
3.2.2 优点 17

3.2.3 缺点 17
3.2.4 应用场景 17
3.2.5 实例 18
3.3 抽象工厂 23
3.3.1 UML图 23
3.3.2 优点 24
3.3.3 缺点 24
3.3.4 应用场景 24
3.3.5 实例 24
第4章 建造者模式 30
4.1 UML图 30
4.2 优点 31
4.3 缺点 31
4.4 应用场景 31
4.5 实例 31
4.6 与抽象工厂模式的异同 35
第5章 原型模式 36
5.1 UML图 36
5.2 优点 37
5.3 缺点 37
5.4 应用场景 37
5.5 实例 37

第二部分 结构型模式 41

第6章 适配器模式 42
6.1 UML图 42
6.2 优点 43
6.3 缺点 43
6.4 应用场景 43
6.5 实例 44

第7章 桥接模式 47
7.1 UML图 47
7.2 优点 48
7.3 缺点 48
7.4 应用场景 48
7.5 实例 48
第8章 外观模式 53
8.1 UML图 53
8.2 优点 54
8.3 缺点 54
8.4 应用场景 54
8.5 实例 54
第9章 复合模式 58
9.1 UML图 58
9.2 优点 59
9.3 缺点 59
9.4 应用场景 59
9.5 实例 59
第10章 装饰模式 63
10.1 UML图 63
10.2 优点 64
10.3 缺点 64
10.4 应用场景 64
10.5 实例 64
10.6 与复合模式的比较 68
第11章 享元模式 69
11.1 UML图 69
11.2 优点 70
11.3 缺点 70
11.4 应用场景 70
11.5 实例 70

11.6 与工厂模式的区别 73

第12章 代理模式 74

12.1 UML图 74

12.2 优点 75

12.3 缺点 75

12.4 应用场景 75

12.5 实例 75

12.6 与装饰模式的区别 77

第三部分 行为型模式 78

第13章 观察者模式 79

13.1 UML图 79

13.2 优点 80

13.3 缺点 80

13.4 应用场景 80

13.5 实例 80

第14章 命令模式 85

14.1 UML图 85

14.2 优点 86

14.3 缺点 86

14.4 应用场景 86

14.5 实例 87

第15章 责任链模式 91

15.1 UML图 91

15.2 优点 92

15.3 缺点 92

15.4 应用场景 92

15.5 实例 92

第16章 解释器模式 96

16.1 UML图 96

16.2 优点 97

16.3 缺点 97
16.4 应用场景 97
16.5 实例 97
16.6 和复合模式的区别 103
第17章 迭代器 104
17.1 UML图 104
17.2 优点 105
17.3 缺点 105
17.4 应用场景 105
17.5 实例 105
第18章 中介者模式 109
18.1 UML图 109
18.2 优点 110
18.3 缺点 110
18.4 应用场景 110
18.5 实例 111
第19章 备忘录模式 116
19.1 UML图 116
19.2 优点 117
19.3 缺点 117
19.4 应用场景 117
19.5 实例 117
第20章 状态模式 121
20.1 UML图 121
20.2 优点 121
20.3 缺点 122
20.4 应用场景 122
20.5 实例 122
20.6 和其它模式的比较 127
20.6.1 Abstract Fatory 127
20.6.2 Strategy 127

第21章 策略模式 128
21.1 UML图 128
21.2 优点 128
21.3 缺点 129
21.4 应用场景 129
21.5 实例 129
第22章 模板模式 133
22.1 UML图 133
22.2 优点 133
22.3 缺点 134
22.4 应用场景 134
22.5 实例 134
第23章 访问者模式 137
23.1 UML图 137
23.2 优点 138
23.3 缺点 138
23.4 应用场景 138
23.5 实例 139

第四部分 附录 143

附录 A UML类图关系的总结 144
A.1 泛化 144
A.2 实现 144
A.3 聚合 144
A.4 组合 145
A.5 关联 146
A.6 依赖 147
A.7 一个例子 147
附录 B 异同 150

第1章　设计模式的六大原则

设计模式（Design Pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

设计模式总原则：开闭原则（Open Close Principle）。软件设计要对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。

所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，需要使用接口和抽象类等。

1.1　单一职责原则

不要存在多于一个导致类变更的原因。

每个类应该实现单一的职责，如若不然，就应该把类拆分。

1.2　里氏替换原则（Liskov Substitution Principle）

任何基类可以出现的地方，子类一定可以出现。

LSP 是面向对象设计的基本原则之一，是继承复用的基石。当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

里氏代换原则是对“开-闭”原则的补充，实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

里氏替换原则中，子类对父类的方法尽量不要重写和重载。因为子类不

应该随便破坏父类定义好的结构。

1.3 依赖倒转原则（Dependence Inversion Principle）

面向接口编程，依赖于抽象而不依赖于具体。

写代码用到具体类时，不与具体类交互，而与其上层接口交互。

1.4 接口隔离原则（Interface Segregation Principle）

每个接口中不存在子类用不到却必须实现的方法，否则拆分接口。

使用多个隔离的接口，比使用混杂的单个接口要好。

1.5 迪米特法则（最少知道原则）（Demeter Principle）

一个类对自己依赖的类，知道的越少越好。

无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过 public 方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。

耦合分为依赖、关联、聚合、组合。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

1.6 合成复用原则（Composite Reuse Principle）

尽量使用合成或聚合的方式，而不是使用继承。

第一部分

创建型模式

第2章　单例模式

单例模式（Singleton）保证一个类仅有一个实例，并提供一个访问它的全局访问点。

2.1　UML图

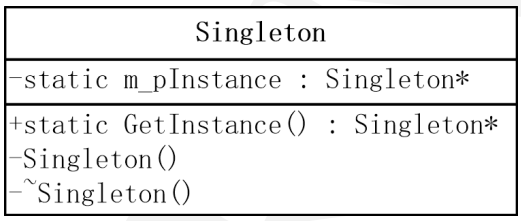

图 2–1

图 2–1 中，类和对象的关系为：

1. 单例（Singleton）：提供一个Instance的方法，使客户可以调用由内部实现生成的唯一一个实例。

2.2　优点

1. 为一个面向对象的应用程序提供了一个唯一的对象访问点。

2.3　缺点

1. 派生子类比较困难，只有在父类没有被实例化时才可以实现。

2.4 应用场景

1. 系统只需要一个实例的对象。
2. 客户调用类的单个实例只允许使用一个公共访问点。

2.5 实例

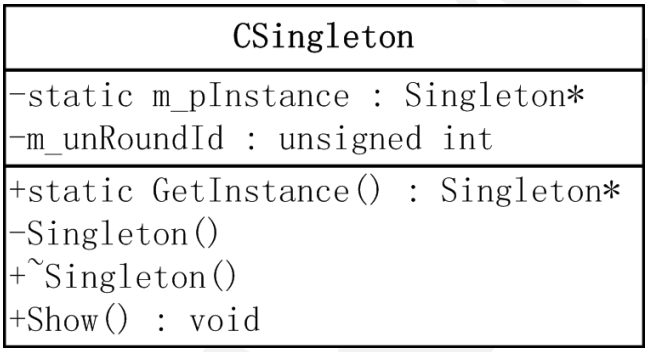

图 2–2

2.5.1 C++

代码 2.1: Singleton.h

```
#include <mutex>
#include <stdio.h>
#include <string>

std::mutex m_oSingletonLock;

class CSingleton
{
public:
    static CSingleton* GetInstance()
    {
        if (nullptr == m_pInstance)
        {
            m_oSingletonLock.lock();
            if (nullptr == m_pInstance)
            {
                m_pInstance = new CSingleton();
            }
            m_oSingletonLock.unlock();
```

```
20          }
21
22          return m_pInstance;
23      }
24
25      void Show()
26      {
27          printf("show %2u times.\n", ++m_unRoundId);
28      }
29
30  private:
31      // private constructor, unable to instantiated outside the class
32      CSingleton()
33      {
34          printf("singleton....\n");
35          m_unRoundId = 0;
36      }
37      ~CSingleton() {}
38
39      // to destruct singleton
40      class Garbo
41      {
42      public:
43          ~Garbo()
44          {
45              if (nullptr != m_pInstance)
46              {
47                  delete m_pInstance;
48                  m_pInstance = nullptr;
49                  printf("press any key to destroy Singleton\n");
50                  getchar();
51              }
52          }
53      };
54
55      // static member, the only instance of the class
56      static CSingleton* m_pInstance;
57
58      // static member, to destruct singleton
59      static Garbo oGarbo;
60
61      unsigned m_unRoundId;
62  };
63
64  CSingleton *CSingleton::m_pInstance = nullptr;
65  CSingleton::Garbo CSingleton::oGarbo;
```

代码 2.2: MainCaller.cpp

```
1  #include "Singleton.h"
```

```
2
3  int main(int argc, char** argv)
4  {
5      for (int rIdx = 0; rIdx < 10; rIdx++)
6      {
7          CSingleton::GetInstance()->Show();
8      }
9
10     printf("press any key to exit...\n");
11     getchar();
12 }
```

2.5.2 Python

Python实例

代码 2.3: Singleton.py

```
1  import threading
2
3
4  class Singleton(object):
5      vars = {}
6      single_lock = threading.Lock()
7      round_idx = 0
8
9      def __new__(cls, *args, **kwargs):
10         if cls in cls.vars:
11             return cls.vars[cls]
12         cls.single_lock.acquire()
13         try:
14             if cls in cls.vars:
15                 return cls.vars[cls]
16             cls.vars[cls] = super().__new__(cls, *args, **kwargs)
17             return cls.vars[cls]
18         finally:
19             cls.single_lock.release()
20
21     def show(self):
22         self.single_lock.acquire()
23         try:
24             print("%d" % self.round_idx)
25             self.round_idx += 1
26         finally:
27             self.single_lock.release()
28
29
30 if "__main__" == __name__:
```

```
31      t1 = Singleton()
32      t2 = Singleton()
33      print(id(t1))
34      print(id(t2))
35      t1.show()
36      t2.show()
```

第3章　工厂模式

工厂模式的特点是解耦合，使得系统变得容易维护、可扩展、可利用、灵活性好。依照其抽象层次不同，可分为三种工厂模式：

- 简单工厂模式
- 工厂方法
- 抽象工厂

3.1　简单工厂模式

简单工厂模式专门定义一个类来负责创建其它类型的实例，被创建的实例通常具有共同的父类。

3.1.1　UML图

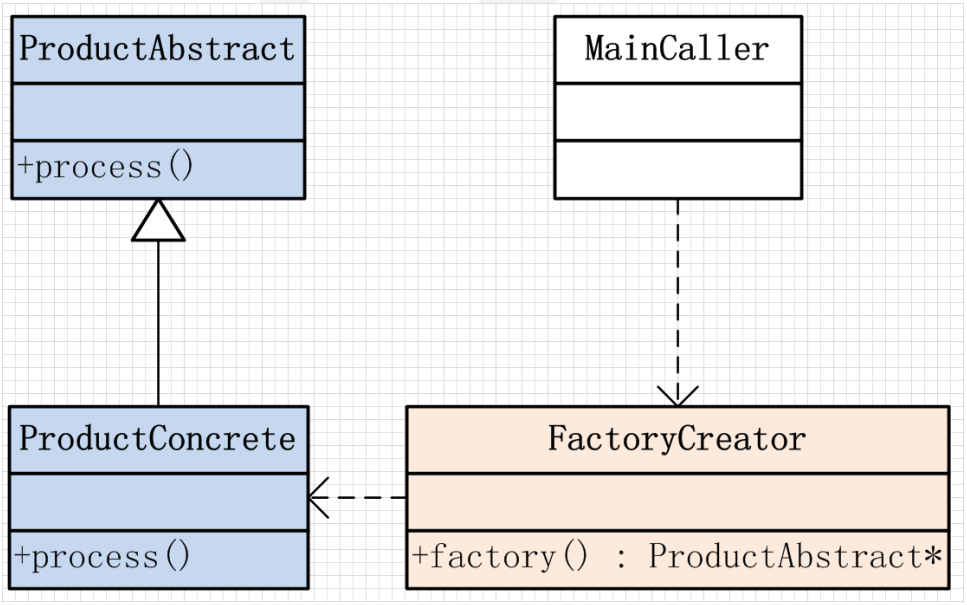

图 3–1: 简单工厂模式模型

图 3–1 中，类和对象的关系为：

1. 工厂（Factory Creator）：简单工厂模式的核心，负责实现创建所有实例的内部逻辑。此类被外界直接调用，创建所需要的产品对象。
2. 抽象产品（Abstract Product）：被创建的所有对象的父类，用以描述所有实例所共有的公共接口。
3. 具体产品（Concrete Product）：被创建的目标。所有被创建的对象，都充当这个角色的某个具体类的实例。

3.1.2 优点

1. 能够根据外界给定的信息，返回创建的具体的对象（switch）

3.1.3 缺点

1. 工厂（Factory Creator）集中了所有实例的创建逻辑，很容易违反GRASP（General Responsibility Assignment Software Patterns）的高内聚的责任分配原则。
2. 工厂类需要知道每一个产品类的细节，当系统中的具体产品类较多时，会出现要求工厂类根据不同条件创建不同实例的需求。这种对条件的判断和对产品的判断，对系统扩展不利。
3. 在一个类中，根据 if 或 switch 分支来决定创建哪一具体类型的对象，一旦需要扩展，就需要修改类的代码，增加分支，破坏了软件实体可扩展但是不可修改的“开-闭”（OCP，Open Closed Principle）原则。

3.1.4 应用场景

1. 工厂类负责创建的对象比较少。
2. 客户只知道传入工厂类的参数，对如何创建对象不关心。

3.1.5 实例

设计一个收银系统，平时按照正常价格计费，遇到商场促销活动时，有的商品打折，有的商品满减。

代码的结构如图 3–2 所示：

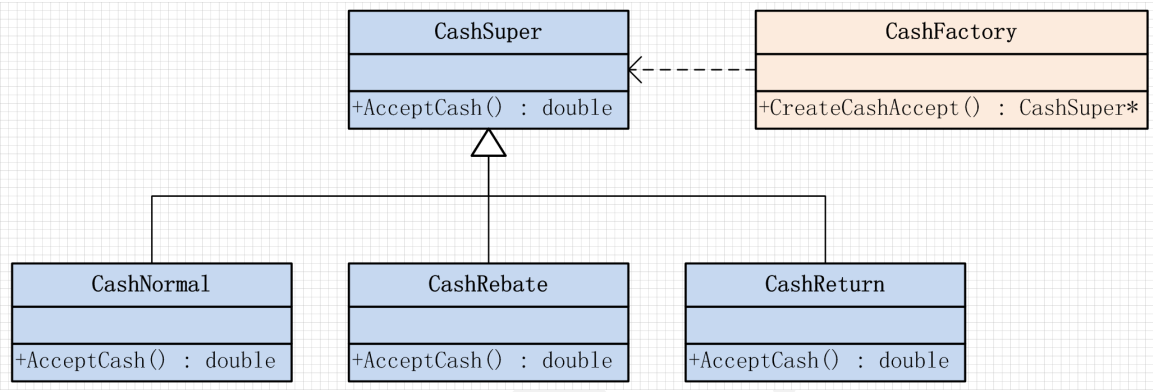

图 3–2: 简单工厂模式实例

3.1.5.1 C++

代码 3.1: SuperCash.h

```
#include <stdio.h>

enum CashAcceptMethod
{
    Normal,
    Rebate,
    Return
};

class ICashSuper
{
public:
    virtual double AcceptCash() = 0;
};
```

代码 3.2: CashNormal.h

```
#include "CashSuper.h"

class CCashNormal : public ICashSuper
{
```

```
5  public:
6      CCashNormal() {}
7      ~CCashNormal() {}
8
9      double AcceptCash()
10     {
11         printf ("accetp cash in normal method.\n");
12     }
13 };
```

代码 3.3: CashRebate.h

```
1  #include "CashSuper.h"
2
3  class CCashRebate : public ICashSuper
4  {
5  public:
6      CCashRebate() {}
7      ~CCashRebate() {}
8
9      double AcceptCash()
10     {
11         printf ("accetp cash in rebate method.\n");
12     }
13 };
```

代码 3.4: CashReturn.h

```
1  #include "CashSuper.h"
2
3  class CCashReturn : public ICashSuper
4  {
5  public:
6      CCashReturn() {}
7      ~CCashReturn() {}
8
9      double AcceptCash()
10     {
11         printf ("accetp cash in return method.\n");
12     }
13 };
```

代码 3.5: CashFactory.h

```
1  #include "CashNormal.h"
2  #include "CashRebate.h"
3  #include "CashReturn.h"
4
5  class CCashFactory
6  {
```

```
7  public:
8      CCashFactory() {}
9      ~CCashFactory() {}
10
11     ICashSuper* CreateCashAccept(CashAcceptMethod eAcceptMethod)
12     {
13         switch (eAcceptMethod)
14         {
15         case Normal:
16             return new CCashNormal();
17         case Rebate:
18             return new CCashRebate();
19         case Return:
20             return new CCashReturn();
21         default:
22             printf("this branch should never be hit!\n");
23             return nullptr;
24         }
25     }
26 };
```

代码 3.6: MainCaller.cpp

```
1  #include "CashFactory.h"
2
3  int main(int argc, char **argv)
4  {
5      CCashFactory *pCashFactory = new CCashFactory();
6
7      ICashSuper *pCashNormalAccepter = pCashFactory->CreateCashAccept(Normal);
8      pCashNormalAccepter->AcceptCash();
9      delete pCashNormalAccepter;
10     pCashNormalAccepter = nullptr;
11
12     ICashSuper *pCashRebateAccepter = pCashFactory->CreateCashAccept(Rebate);
13     pCashRebateAccepter->AcceptCash();
14     delete pCashRebateAccepter;
15     pCashRebateAccepter = nullptr;
16
17     ICashSuper *pCashReturnAccepter = pCashFactory->CreateCashAccept(Return);
18     pCashReturnAccepter->AcceptCash();
19     delete pCashReturnAccepter;
20     pCashReturnAccepter = nullptr;
21
22     delete pCashFactory;
23     pCashFactory = nullptr;
24
25     printf("press any key to continue...\n");
26     getchar();
27 }
```

3.1.5.2 Python

代码 3.7: cash_super.py

```
import enum

class CashAcceptMethod(enum.Enum):
    price_normal = 1,
    price_rebate = 2,
    price_return = 3


class CashSuper:
    """
    define interface
    """

    def __init__(self):
        pass

    def accept_cash(self):
        """pure virtual function"""
        raise NotImplementedError()
```

代码 3.8: cash_normal.py

```
import cash_super

class CashNormal(cash_super.CashSuper):
    """
    accept cash in normal method
    """

    def __init__(self):
        pass

    def accept_cash(self):
        print("accept cash in normal method")
```

代码 3.9: cash_rebate.py

```
import cash_super

class CashRebate(cash_super.CashSuper):
    """
    accept cash in rebate method
    """

    def __init__(self):
        pass
```

```
10
11      def accept_cash(self):
12          print("accept cash in rebate method")
```

代码 3.10: cash_return.py

```
1  import cash_super
2
3
4  class CashReturn(cash_super.CashSuper):
5      """
6      accept cash in return method
7      """
8
9      def __init__(self):
10         pass
11
12     def accept_cash(self):
13         print("accept cash in return method")
```

代码 3.11: cash_factory.py

```
1  import cash_normal
2  import cash_rebate
3  import cash_return
4  import cash_super
5
6  class CashFactory:
7      """
8      cash factory
9      """
10
11     def __init__(self):
12         pass
13
14     def create_cash_accept(self, cash_accept_method):
15         if cash_super.CashAcceptMethod.price_normal == cash_accept_method:
16             return cash_normal.CashNormal()
17         elif cash_super.CashAcceptMethod.price_rebate == cash_accept_method:
18             return cash_rebate.CashRebate()
19         elif cash_super.CashAcceptMethod.price_return == cash_accept_method:
20             return cash_return.CashReturn()
21         else:
22             return None
```

代码 3.12: factory.py

```
1  import cash_factory
2  import cash_super
3
```

```
4  if "__main__" == __name__:
5      factory = cash_factory.CashFactory()
6
7      cash_acceptor = factory.create_cash_accept(cash_super.CashAcceptMethod.
           price_normal)
8      cash_acceptor.accept_cash()
9
10     cash_acceptor = factory.create_cash_accept(cash_super.CashAcceptMethod.
           price_rebate)
11     cash_acceptor.accept_cash()
12
13     cash_acceptor = factory.create_cash_accept(cash_super.CashAcceptMethod.
           price_return)
14     cash_acceptor.accept_cash()
```

3.2 工厂方法

由 3.1.3 节可知，简单工厂模式每添加一个产品子类，都需要在工厂类中添加一个判断分支，违背了软件实体可扩展但是不可修改的“开-闭”原则，工厂模式就是为了解决这个问题而产生的。

将每次选择执行时的判断分析，都生成一个工厂子类，每次添加产品的时候，只需要添加一个工厂子类就可以了。这样，就遵循了“开-闭”原则。

3.2.1 UML图

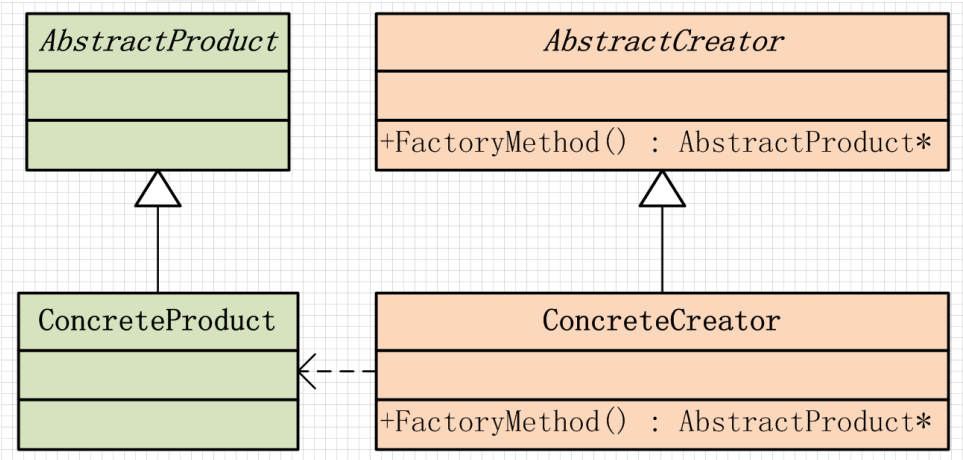

图 3–3: 工厂模式模型

图 3–3 中，类和对象的关系为：

1. 抽象工厂（Abstract Factory）：声明工厂方法（Factory Method），返回一个产品。
2. 具体工厂（Concrete Factory）：实现工厂方法，由客户调用，返回一个产品的实例。
3. 抽象产品（Abstract Product）：被创建的所有对象的父类，用以描述所有实例所共有的公共接口。
4. 具体产品（Concrete Product）：被创建的目标。所有被创建的对象，都充当这个角色的某个具体类的实例。

3.2.2 优点

1. 基于工厂角色和产品角色的多态，能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则封装在工厂内部。
2. 加入新产品时，无需修改抽象工厂和抽象产品，无需修改客户端，也无需修改其它的具体工厂和具体产品，符合“开-闭”原则。

3.2.3 缺点

1. 添加新产品时，不仅需要添加具体产品类，还需要提供与之对应的具体工厂类，增加了系统开销。

3.2.4 应用场景

同 3.1.4 节简单工厂的应用场景。

1. 工厂类负责创建的对象比较少。
2. 客户只知道传入工厂类的参数，对如何创建对象不关心。

3.2.5 实例

设计一个手机工厂，Motorola 工厂生产 Motorola 手机，Nokia 工厂生产 Nokia 手机。

代码的结构如图 3–4 所示：

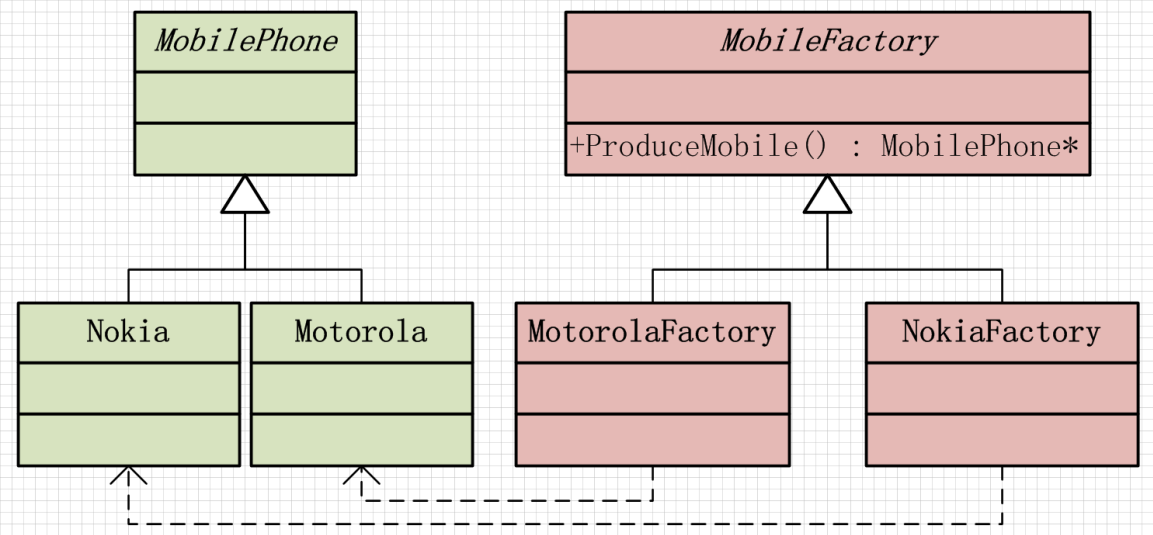

图 3–4: 工厂模式实例

3.2.5.1 C++

代码 3.13: MobilePhone.h

```
1 #include <stdio.h>
2
3 class IMobilePhone
4 {
5 public:
6     void virtual MakeCall() = 0;
7 };
```

代码 3.14: MobileFactory.h

```
1 #include "MobilePhone.h"
2
3 class IMobileFactory
4 {
5 public:
6     virtual IMobilePhone* ProduceMobile() = 0;
7 };
```

代码 3.15: MotorolaFactory.h

```
#include "MobileFactory.h"
#include "MotorolaPhone.h"

class CMotorolaFactory : public IMobileFactory
{
public:
    CMotorolaFactory();
    ~CMotorolaFactory();

    IMobilePhone* ProduceMobile()
    {
        printf("motorola's factory produces ");
        return new CMotorolaPhone();
    }
};
```

代码 3.16: MotorolaPhone.h

```
#include "MobilePhone.h"

class CMotorolaPhone : public IMobilePhone
{
public:
    CMotorolaPhone();
    ~CMotorolaPhone();

    void MakeCall()
    {
        printf("motorola cellphone.\n");
    }
};
```

代码 3.17: NokiaFactory.h

```
#include "MobileFactory.h"
#include "NokiaPhone.h"

class CNokiaFactory : public IMobileFactory
{
public:
    CNokiaFactory();
    ~CNokiaFactory();

    IMobilePhone* ProduceMobile()
    {
        printf("nokia's factory produces ");
        return new CNokiaPhone();
    }
```

```
15 };
```

代码 3.18: NokiaPhone.h

```
1  #include "MobilePhone.h"
2
3  class CNokiaPhone : public IMobilePhone
4  {
5  public:
6      CNokiaPhone();
7      ~CNokiaPhone();
8
9      void MakeCall()
10     {
11         printf("nokia cellphone.\n");
12     }
13 };
```

代码 3.19: MainCaller.cpp

```
1  #include "NokiaFactory.h"
2  #include "MotorolaFactory.h"
3
4  int main(int argc, char**argv)
5  {
6      IMobileFactory *pMobileFactory;
7      IMobilePhone *pMobilePhone;
8
9      pMobileFactory = new CMotorolaFactory();
10     pMobilePhone = pMobileFactory->ProduceMobile();
11     pMobilePhone->MakeCall();
12     delete pMobilePhone;
13     pMobilePhone = nullptr;
14     delete pMobileFactory;
15     pMobileFactory = nullptr;
16
17     pMobileFactory = new CNokiaFactory();
18     pMobilePhone = pMobileFactory->ProduceMobile();
19     pMobilePhone->MakeCall();
20     delete pMobilePhone;
21     pMobilePhone = nullptr;
22     delete pMobileFactory;
23     pMobileFactory = nullptr;
24
25     printf("press any key to exit...\n");
26     getchar();
```

3.2.5.2 Python

代码 3.20: mobile_phone.py

```
class MobilePhone:
    """
    define interface of product
    """

    def make_call(self):
        """pure virtual function"""
        raise NotImplementedError()
```

代码 3.21: mobile_factory.py

```
class MobileFactory:
    """
    define factory
    """

    def produce_mobile(self):
        """pure virtual function"""
        raise NotImplementedError()
```

代码 3.22: motorola_phone.py

```
import mobile_phone

class MotorolaPhone(mobile_phone.MobilePhone):
    """
    motorola phone
    """

    def __init__(self):
        pass

    def make_call(self):
        print("motorola phone")
```

代码 3.23: motorola_factory.py

```
import mobile_factory
import motorola_phone

class MotorolaFactory(mobile_factory.MobileFactory):
    """
    motorola factory
    """

```

```
9       def __init__(self):
10          pass
11
12      def produce_mobile(self):
13          return motorola_phone.MotorolaPhone()
```

代码 3.24: nokia_phone.py

```
1   import mobile_phone
2
3   class NokiaPhone(mobile_phone.MobilePhone):
4       """
5       motorola
6       """
7
8       def __init__(self):
9           pass
10
11      def make_call(self):
12          print("nokia phone")
```

代码 3.25: nokia_factory.py

```
1   import mobile_factory
2   import nokia_phone
3
4   class NokiaFactory(mobile_factory.MobileFactory):
5       """
6       motorola factory
7       """
8
9       def __init__(self):
10          pass
11
12      def produce_mobile(self):
13          return nokia_phone.NokiaPhone()
```

代码 3.26: factory.py

```
1   import motorola_factory
2   import nokia_factory
3
4   if "__main__" == __name__:
5       mobile_factory = motorola_factory.MotorolaFactory()
6       mobile_phone = mobile_factory.produce_mobile()
7       mobile_phone.make_call()
8
9       mobile_factory = nokia_factory.NokiaFactory()
10      mobile_phone = mobile_factory.produce_mobile()
11      mobile_phone.make_call()
```

3.3 抽象工厂

简单工厂模式和工厂模式要求产品子类必须是同一类型的，拥有共同的方法，限制了产品子类的扩展。

抽象工厂将不同的抽象产品的具体子类，组合成一个产品族，当客户调用一个具体工厂时，可以获得一整套抽象产品的具体子类。

3.3.1 UML图

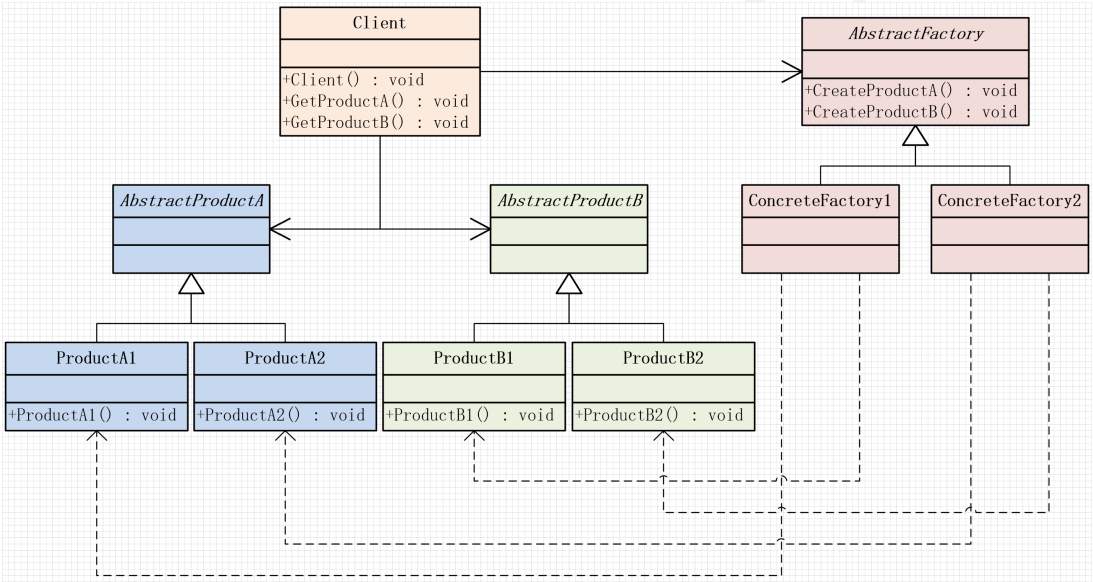

图 3–5: 抽象工厂模式模型

图 3–5 中，类和对象的关系为：

1. 抽象工厂（Abstract Factory）：声明生成抽象产品的方法。
2. 具体工厂（Concrete Factory）：执行生成抽象产品的方法，生成一个具体的产品。
3. 抽象产品（Abstract Product）：被创建的所有对象的父类，用以描述所有实例所共有的公共接口。
4. 具体产品（Concrete Product）：被创建的目标。所有被创建的对象，都充当这个角色的某个具体类的实例。

3.3.2 优点

1. 隔离具体类的生成，使客户不需要知道工厂和具体产品的类型，只能操作工厂接口和产品接口。
2. 可以保证客户使用同一个产品族中的对象。

3.3.3 缺点

1. 添加新产品时，需要对所有的具体工厂添加新的产品。

3.3.4 应用场景

1. 系统需要屏蔽有关对象如何创建、如何组织和如何表示。
2. 系统需要由关联的多个对象来构成。
3. 有关联的多个对象需要一起应用，对象之间有强约束，不可分离。
4. 提供一组对象而不显示它们的实现过程，只显示它们的接口。

3.3.5 实例

代码的结构如图 3–6 所示：

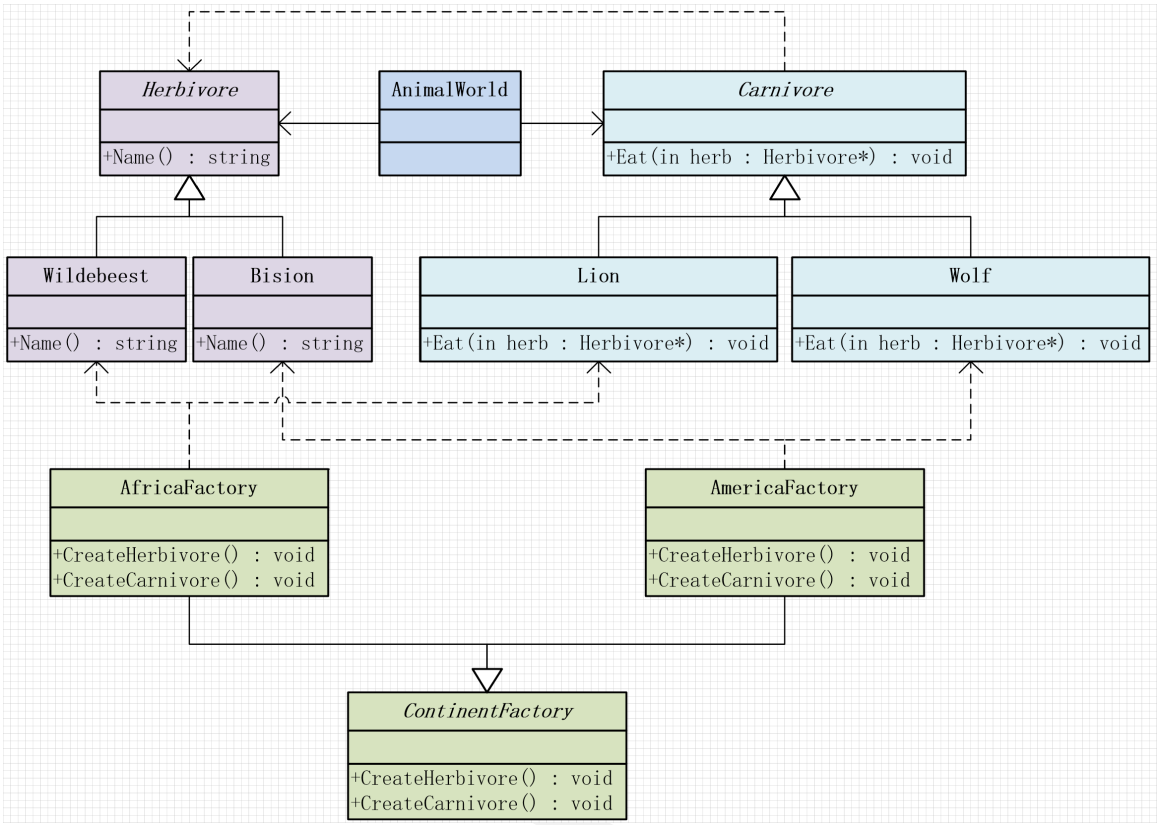

图 3–6: 简单工厂模式实例

3.3.5.1 C++

代码 3.27: Herbivore.h

```
#include <string>

class IHerbivore
{
public:
    virtual std::string Name() = 0;
};
```

代码 3.28: Carnivore.h

```
#include "Herbivore.h"

class ICarnivore
{
public:
    virtual void Eat(IHerbivore *pHerb) = 0;
    virtual std::string Name() = 0;
};
```

代码 3.29: ContinentFactory.h

```
#include "Herbivore.h"
#include "Carnivore.h"

class IContinentFactory
{
public:
    virtual IHerbivore* CreateHerbivore() = 0;
    virtual ICarnivore* CreateCarnivore() = 0;
};
```

代码 3.30: Bision.h

```
#include "Herbivore.h"

class CBision : public IHerbivore
{
public:
    CBision() {}
    ~CBision() {}

    std::string Name()
    {
        return "Bison";
    }
};
```

代码 3.31: Wolf.h

```
#include "Carnivore.h"

class CWolf : public ICarnivore
{
public:
    CWolf() {}
    ~CWolf() {}

    void Eat(IHerbivore *pHerb)
    {
        printf("%s eats %s.\n", Name().c_str(), pHerb->Name().c_str());
    }

    std::string Name()
    {
        return "Wolf";
    }
};
```

代码 3.32: AmericaFactory.h

```
#include "ContinentFactory.h"
#include "Bision.h"
#include "Wolf.h"

class CAmericaFactory : public IContinentFactory
{
public:
    CAmericaFactory() {}
    ~CAmericaFactory() {}

    IHerbivore* CreateHerbivore()
    {
        return new CBision();
    }

    ICarnivore* CreateCarnivore()
    {
        return new CWolf();
    }
};
```

代码 3.33: Wildebeest.h

```
#include "Herbivore.h"

class CWildebeest : public IHerbivore
{
public:
    CWildebeest() {}
    ~CWildebeest() {}

    std::string Name()
    {
        return "WildeBeest";
    }
};
```

代码 3.34: Lion.h

```
#include "Carnivore.h"

class CLion : public ICarnivore
{
public:
    CLion() {}
    ~CLion() {}

    void Eat(IHerbivore *pHerb)
```

```
10      {
11          printf ("%s eats %s\n", Name().c_str(), pHerb->Name().c_str());
12      }
13
14      std :: string  Name()
15      {
16          return  "Lion";
17      }
18  };
```

代码 3.35: AfricaFactory.h

```
1   #include "ContinentFactory.h"
2   #include "Lion.h"
3   #include "Wildebeest.h"
4
5   class CAfricaFactory : public IContinentFactory
6   {
7   public:
8       CAfricaFactory() {}
9       ~CAfricaFactory() {}
10
11      IHerbivore* CreateHerbivore()
12      {
13          return new CWildebeest();
14      }
15
16      ICarnivore* CreateCarnivore()
17      {
18          return new CLion();
19      }
20  };
```

代码 3.36: AnimalWorld.h

```
1   #include "Carnivore.h"
2   #include "Herbivore.h"
3   #include "ContinentFactory.h"
4
5   class CAnimalWorld
6   {
7   public:
8       CAnimalWorld(IContinentFactory *pFactory)
9       {
10          m_pHerb = pFactory->CreateHerbivore();
11          m_pCarn = pFactory->CreateCarnivore();
12      }
13
14      ~CAnimalWorld()
15      {
```

```
16          delete m_pHerb;
17          m_pHerb = nullptr;
18
19          delete m_pCarn;
20          m_pCarn = nullptr;
21      }
22
23      void RunFoodChain()
24      {
25          m_pCarn->Eat(m_pHerb);
26      }
27
28  private:
29      IHerbivore *m_pHerb;
30      ICarnivore *m_pCarn;
31  };
```

代码 3.37: MainCaller.cpp

```
1   #include "AfricaFactory.h"
2   #include "AmericaFactory.h"
3   #include "AnimalWorld.h"
4
5   int main(int argc, char** argv)
6   {
7       IContinentFactory *pAfrica = new CAfricaFactory();
8
9       CAnimalWorld *pAanimalWorld = new CAnimalWorld(pAfrica);
10      pAanimalWorld->RunFoodChain();
11      delete pAanimalWorld;
12      pAanimalWorld = nullptr;
13      delete pAfrica;
14      pAfrica = nullptr;
15
16      IContinentFactory *pAmerica = new CAmericaFactory();
17
18      pAanimalWorld = new CAnimalWorld(pAmerica);
19      pAanimalWorld->RunFoodChain();
20      delete pAanimalWorld;
21      pAanimalWorld = nullptr;
22      delete pAmerica;
23      pAmerica = nullptr;
24
25      printf("press any key to exit...\n");
26      getchar();
27  }
```

3.3.5.2 Python

第4章　建造者模式

建造者模式（Builder）将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示，可以只通过指定复杂对象各组成部分的类型和内容就可以构建它们，用户不知道内部的具体构建细节。

4.1　UML图

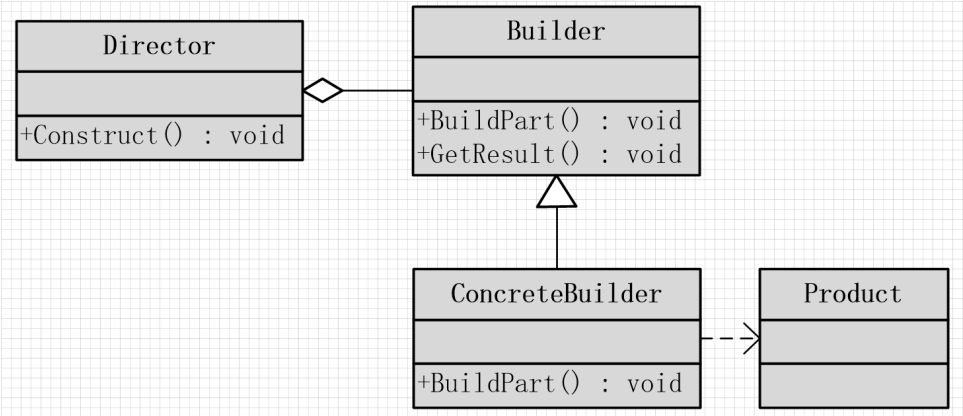

图 4–1

图 4–1 中，类和对象的关系为：

1. 抽象建造者（Abstract Builder）：为创建一个产品（Product）对象的各个部分指定抽象接口，用来规范产品对象和各个组成成份。
2. 具体建造者（Concrete Builder）：在指导者的调用下创建产品实例，拥有产品类的具体知识。
3. 指导者（Director）：调用具体建造者以创建产品对象，指导者没有产品类的具体知识。
4. 产品（Product）：待创建的复杂对象，需要包含那些定义组件的类，包括将这些组件装配成产品的接口。

4.2 优点

1. 职责分离，使结构更加清晰。
2. 需要新产品时，只需要从建造者基类（AbstractBuilder）派生新的建造者子类（ConcreteBuilder），在其中实现新产品的创建工作。
3. 如果需要调整原有产品的创建过程，只需要对调用 AbstractBuilder 类接口时，对产品构建自制代码作简要修改即可。
4. 产品表示由产品类自身完成，而产品的构建顺序由外部的建造者来完成，产品的表示与构建分离了。

4.3 缺点

1. 产品的构造步骤是固定的，不允许修改，只允许变动顺序。
2. 产品的构建模式变化较多时，容易导致具体建造者子类膨胀。

4.4 应用场景

1. 创建复杂对象的算法，独立于该对象的组成部分和其装配方式。
2. 构造过程允许被构造的对象有不同的表示。

4.5 实例

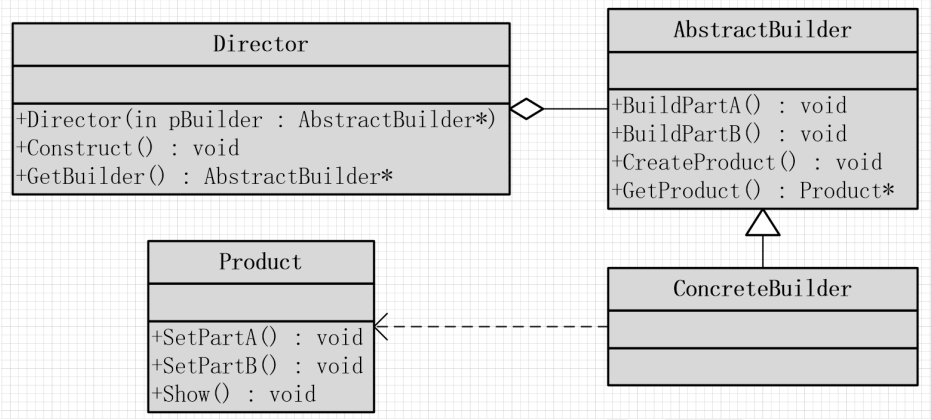

图 4–2

代码 4.1: Product.h

```
#include <stdio.h>

class CProduct
{
public:
    CProduct() {}
    ~CProduct() {}

    void SetPartA(int nParam)
    {
        m_nPartA = nParam;
    }

    void SetPartB(int nParam)
    {
        m_nPartB = nParam;
    }

    void Show()
    {
        printf("partA = \%d, partB = \%d.\n", m_nPartA, m_nPartB);
    }

private:
    int m_nPartA;
    int m_nPartB;
};
```

代码 4.2: AbstractBuilder.h

```
#include "Product.h"

class IAbstractBuilder
```

```
4  {
5  public:
6      virtual void BuildPartA(int nParam) = 0;
7      virtual void BuildPartB(int nParam) = 0;
8
9      virtual void CreateProduct() = 0;
10
11     virtual CProduct* GetProduct() = 0;
12 };
```

代码 4.3: ConcreteBuilder.h

```
1  #include "AbstractBuilder.h"
2
3  class CConcreteBuilder : public IAbstractBuilder
4  {
5  public:
6      CConcreteBuilder() {}
7      ~CConcreteBuilder() {}
8
9      void BuildPartA(int nParam)
10     {
11         printf("constructing part A.\n");
12         m_pCurProduct->SetPartA(nParam);
13     }
14
15     void BuildPartB(int nParam)
16     {
17         printf("constructing part B.\n");
18         m_pCurProduct->SetPartB(nParam);
19     }
20
21     void CreateProduct()
22     {
23         printf("constructing an empty product.\n");
24         m_pCurProduct = new CProduct();
25     }
26
27     CProduct* GetProduct()
28     {
29         return m_pCurProduct;
30     }
31
32 private:
33     CProduct *m_pCurProduct;
34 };
```

代码 4.4: Director.h

```
#include "ConcreteBuilder.h"

class CDirector
{
public:
    CDirector( IAbstractBuilder *pBuilder)
    {
        m_pCurBuilder = pBuilder;
    }

    ~CDirector()
    {
        m_pCurBuilder = nullptr;
    }

    void Construct()
    {
        if ( nullptr == m_pCurBuilder)
        {
            printf("no builder in director.\n");
            return;
        }

        m_pCurBuilder->CreateProduct();
        m_pCurBuilder->BuildPartA(1);
        m_pCurBuilder->BuildPartB(2);
    }

    IAbstractBuilder* GetBuilder()
    {
        return m_pCurBuilder;
    }

private:
    IAbstractBuilder *m_pCurBuilder;
};
```

代码 4.5: MainCaller.cpp

```
#include <stdlib.h>

#include "Director.h"

int main(int argc, char** argv)
{
    IAbstractBuilder *pBuilder = new CConcreteBuilder();

    CDirector *pDirector = new CDirector(pBuilder);
```

```
10
11      pDirector->Construct();
12      pBuilder->GetProduct()->Show();
13      pDirector->GetBuilder()->GetProduct()->Show();
14
15      delete pDirector;
16      pDirector = nullptr;
17      delete pBuilder;
18      pBuilder = nullptr;
19
20      printf("press any key to exit...\n");
21      getchar();
22  }
```

4.6 与抽象工厂模式的异同

Builder 模式强调的是一步步创建对象，通过相同的创建过程可以获得不同的结果对象，创建的对象不是直接返回的。

AbstractFactory 模式中对象是直接返回的，强调的是为创建多个相互依赖的对象提供一个同一的接口。

第5章　原型模式

原型模式（Proto Type）用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。动态抽取当前对象运行时的状态，从自身构造一个新的对象，即自身的拷贝。

5.1　UML图

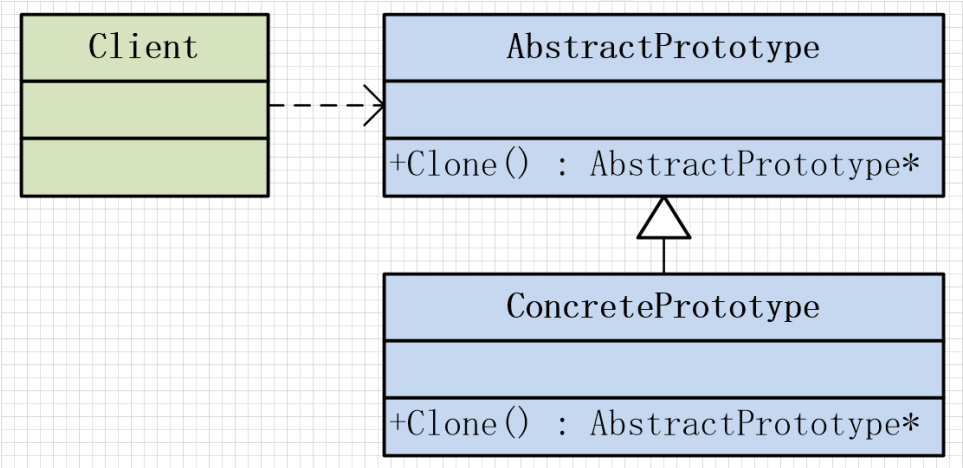

图 5–1

图 5–1 中，类和对象的关系为：

1. 抽象原型（Abstract Prototype）：一个抽象角色，给出所有具体原型类所需的接口。
2. 具体原型（Concrete Prototype）：被复制的对象，需要实现抽象原型角色所要求的接口。
3. 客户（Client）：客户类提出创建对象的请求

5.2 优点

1. 通过复制自身来创建新产品，客户不需要知道对象的实例类型，只需要知道它的抽象基类即可。
2. 扩展产品类型比较容易，不会影响到其它产品甚至是客户代码。
3. 当需要拷贝一个包含多种元素的子类时，需要做的工作将十分简单。
4. 新生成的对象克隆了被复制对象运行时的状态。

5.3 缺点

1. 必须先有一个对象实例（即原型）才能复制

5.4 应用场景

1. 类的实例化是动态的。
2. 需要避免使用分层次的工厂类来创建分层次的对象。
3. 类的实例对象只有一个或很少的几个组合状态。

5.5 实例

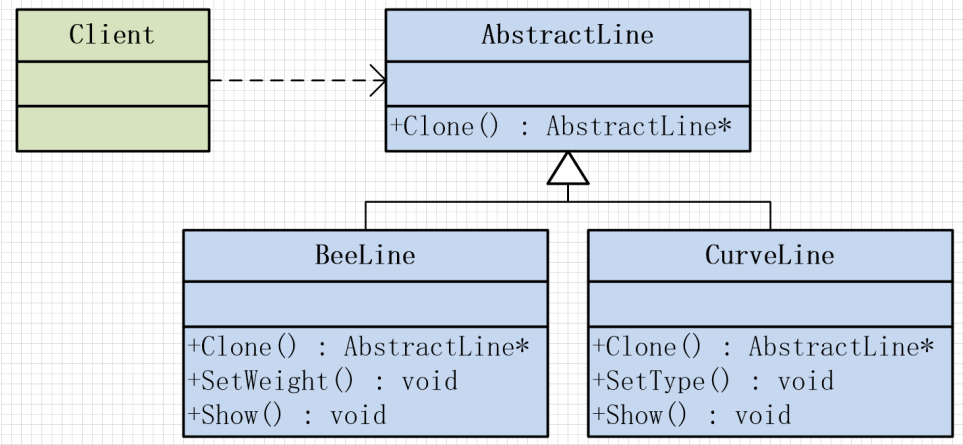

图 5–2

代码 5.1: AbstractLine.h

```
1  #include <stdio.h>
2
3  class IAbstractLine
4  {
5  public:
6      virtual IAbstractLine* Clone() = 0;
7  };
```

代码 5.2: BeeLine.h

```
1  #include "AbstractLine.h"
2
3  class CBeeLine: public IAbstractLine
4  {
5  public:
6      CBeeLine() {}
7      ~CBeeLine() {}
8
9      IAbstractLine* Clone()
10     {
11         printf("Bee line clone a line.\n");
12         return new CBeeLine(*this);
13     }
14
15     void SetWeight(const int nWeight)
16     {
17         m_nWeight = nWeight;
18     }
19
20     void Show()
21     {
22         printf("beeline weight = %d.\n", m_nWeight);
```

```
23      }
24
25  protected :
26      int  m_nWeight;
27  };
```

代码 5.3: CurveLine.h

```
1   #include "AbstractLine.h"
2
3   class  CCurveLine : public  IAbstractLine
4   {
5   public :
6       CCurveLine()
7       {
8           m_strLineType. reserve (10) ;
9       }
10      ~CCurveLine() {}
11
12      IAbstractLine * Clone()
13      {
14          printf ("Curve line clone a curve.\n");
15          return  new CCurveLine(*this);
16      }
17
18      void  SetType(const  std :: string  &strLineType)
19      {
20          m_strLineType = strLineType ;
21      }
22
23      void  Show()
24      {
25          printf ("line type = %s.\n", m_strLineType.c_str());
26      }
27
28  protected :
29      std :: string  m_strLineType;
30  };
```

代码 5.4: MainCaller.cpp

```
1   #include "BeeLine.h"
2   #include "CurveLine.h"
3
4   int  main(int  argc , char**argv)
5   {
6       CBeeLine *pBeeLine = new CBeeLine();
7       pBeeLine->SetWeight(10);
8       CBeeLine *pBeeClone = (CBeeLine*)(pBeeLine->Clone());
9       pBeeClone->Show();
```

```
10      delete pBeeClone;
11      pBeeClone = nullptr ;
12      delete pBeeLine;
13      pBeeLine = nullptr ;
14
15      CCurveLine *pCurveLine = new CCurveLine();
16      pCurveLine->SetType("DOT");
17      CCurveLine *pCurveClone = (CCurveLine*)(pCurveLine->Clone());
18      pCurveClone->Show();
19      delete pCurveClone;
20      pCurveClone = nullptr ;
21      delete pCurveLine;
22      pCurveLine = nullptr ;
23
24      printf ("press any key to exit...\n");
25      getchar () ;
26  }
```

第二部分

结构型模式

第6章　适配器模式

适配器模式（Adapter）将一个类的接口转换成客户希望的另外一个接口，为一个功能正确但接口不合的对象创建一个新接口。 Adapter 模式经常被描述成第三方函数库或旧的程序库与现有系统接口或需求不一致时的救星。

6.1　UML图

适配器模式一共有两种形式，分别为继承模式和对象模式。

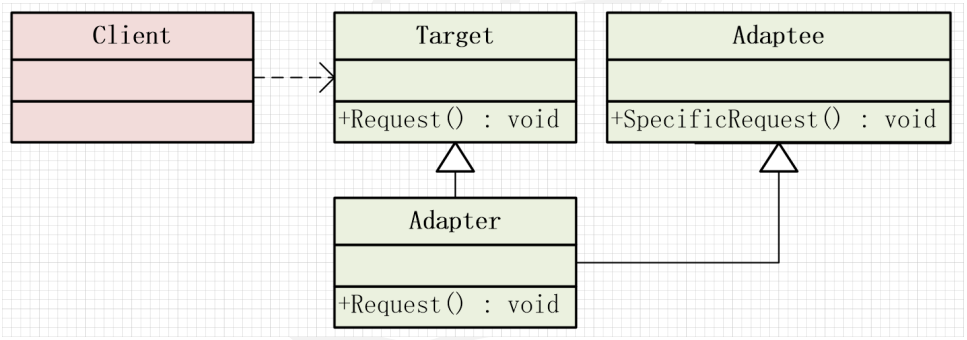

图 6–1: 继承模式

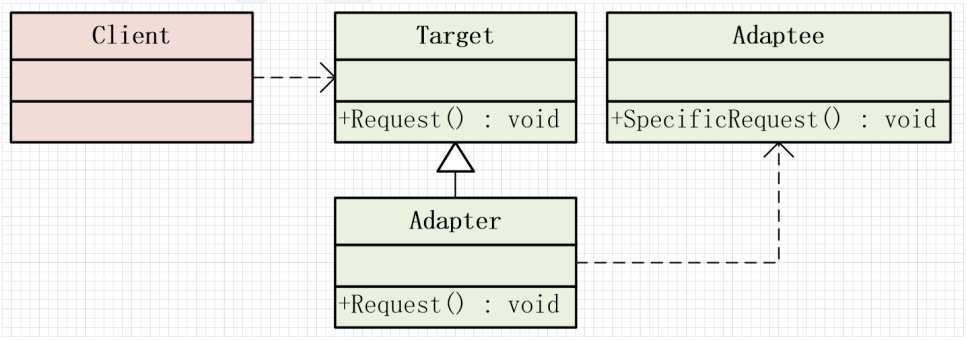

图 6–2: 对象模式

图 6–1 和 6–2 中，类和对象的关系为：

1. 目标（Target）：客户期待的接口，目标可以是具体的或者抽象的类，也可以是接口。
2. 原接口（Adaptee）：需要适配的类。
3. 继承模式的适配器（Adapter）：负责包装原接口，以把原接口转换成目标接口，此角色必须是类。
4. 对象模式的适配器（Adapter）：通过在内部包装一个Adaptee对象，把原接口转换成目标接口。

6.2 优点

1. 委托实现，提供不同接口
2. 提供接口默认实现，简化客户代码，解决需求与实现不一致的问题。
3. 接口转换，在维持原有接口的同时，达到支持新接口的目的。
4. 简化接口，分离实现。

6.3 缺点

1. 代码难以维护，随着系统的升级，接口类将变得越来越臃肿。
2. 底层对客户过于透明，客户程序员要很好地使用接口。

6.4 应用场景

1. 系统需要使用现有的类，而此类的接口不符合系统的需要。
2. 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关系的一些类，包括一些可能在接来引进的类一起工作，这些源类不一定有一致的接口。
3. 通过接口转换，将一个类插入另一个类系中。

6.5 实例

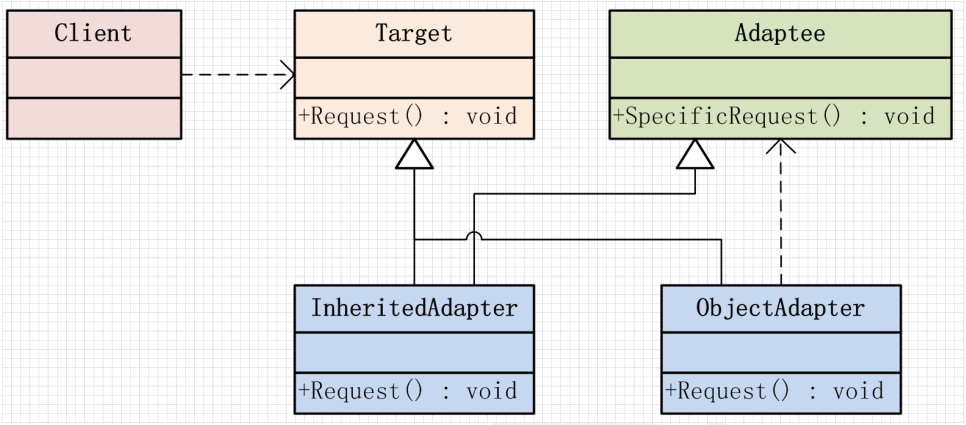

图 6–3

代码 6.1: Target.h

```
1  class ITarget
2  {
3  public:
4      virtual void Request() = 0;
5  };
```

代码 6.2: Adaptee.h

```
1  #include <stdio.h>
2
3  class CAdaptee
4  {
5  public:
6      CAdaptee() {}
7      virtual ~CAdaptee() {}
8
9      void SpecificRequest ()
10     {
11         printf ("specific request in adaptee.\n");
12     }
13 };
```

代码 6.3: ObjectAdapter.h

```
1  #include "Target.h"
2  #include "Adaptee.h"
3
4  class CObjectAdapter : public ITarget
5  {
```

```
6  public:
7      CObjectAdapter()
8      {
9          m_pAdapter = new CAdaptee();
10     }
11     ~CObjectAdapter()
12     {
13         delete m_pAdapter;
14         m_pAdapter = nullptr;
15     }
16
17     void Request()
18     {
19         printf("object adapter calls ");
20         m_pAdapter->SpecificRequest();
21     }
22
23 private:
24     CAdaptee *m_pAdapter;
25 };
```

代码 6.4: InheritedAdapter.h

```
1  #include "Adaptee.h"
2  #include "Target.h"
3
4  class CInheritedAdapter : public ITarget, public CAdaptee
5  {
6  public:
7      CInheritedAdapter() {}
8      ~CInheritedAdapter() {}
9
10     void Request()
11     {
12         printf("inherited adapter calls ");
13         SpecificRequest();
14     }
15 };
```

代码 6.5: MainCaller.cpp

```
1  #include "InheritedAdapter.h"
2  #include "ObjectAdapter.h"
3
4  int main(int argc, char** argv)
5  {
6      ITarget *pTargetFunc = new CObjectAdapter();
7      pTargetFunc->Request();
8      delete pTargetFunc;
9      pTargetFunc = nullptr;
```

```
10
11      pTargetFunc = new CInheritedAdapter();
12      pTargetFunc->Request();
13      delete pTargetFunc;
14      pTargetFunc = nullptr;
15
16      printf("press any key to exit...\n");
17      getchar();
18  }
```

第7章　桥接模式

桥接模式（Bridge）将事物的抽象部分与其实现部分解耦，使它们都可以独立地变化。对事件本身的抽象为抽象部分，对事物功能的实现为实现部分，使这两部分分离。这里的实现部分并不是指从抽象类派生的具体类。

7.1　UML图

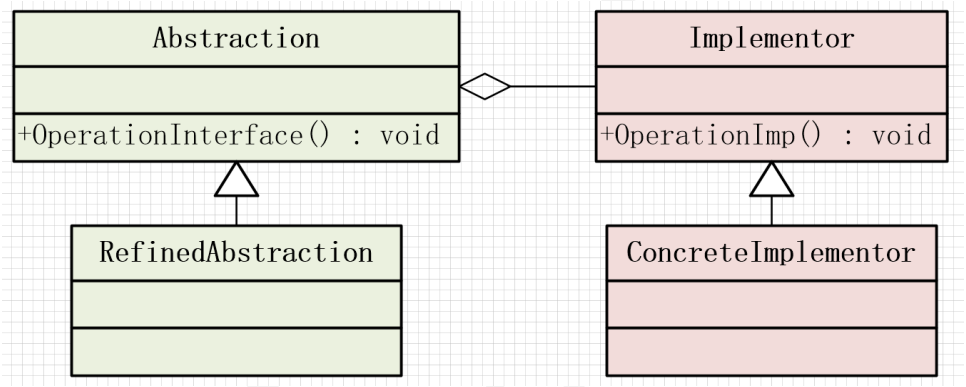

图 7–1

图 7–1 中，类和对象的关系为：

1. Abstraction（抽象类）：定义抽象部分的接口，维护一个 Implementor（实现抽象类）的对象。
2. RefinedAbstraction（扩充抽象类）：扩充由Abstraction定义的接口。
3. Implementor（实现类接口）：定义实现部分的接口，不一定要与 Abstraction 的接口完全一致，一般 Implementor 只提供基本操作，而 Abstraction 会做更多复杂的操作。
4. ConcreteImplementor（具体实现类）：实现Implementor的接口。

7.2 优点

1. 分离抽象接口及其实现部分，提供了比继承更好的解决方案。
2. 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。
3. 实现细节对客户透明。

7.3 缺点

1. 桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。
2. 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

7.4 应用场景

1. 一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。
2. 适用于不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统。
3. 一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

7.5 实例

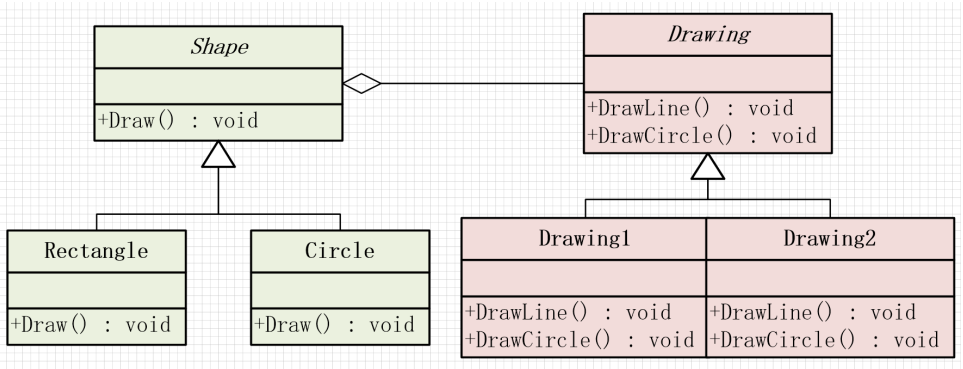

图 7–2

代码 7.1: Drawing.h

```
#include <stdio.h>

class IDrawing
{
public:
    void virtual DrawLine() = 0;
    void virtual DrawCircle() = 0;
};
```

代码 7.2: Drawing1.h

```
#include "Drawing.h"

class CConcreteDrawing1 : public IDrawing
{
public:
    CConcreteDrawing1() {}
    ~CConcreteDrawing1() {}

    void DrawLine()
    {
        printf ("draw a line called in Drawing1.\n");
    }

    void DrawCircle()
    {
        printf ("draw a circle called in Drawing1.\n");
    }
};
```

代码 7.3: Drawing2.h

```
#include "Drawing.h"

class CConcreteDrawing2 : public IDrawing
{
public:
    CConcreteDrawing2() {}
    ~CConcreteDrawing2() {}

    void DrawLine()
    {
        printf ("draw a line called in Drawing2.\n");
    }

    void DrawCircle()
    {
        printf ("draw a circle called in Drawing2.\n");
    }
};
```

代码 7.4: Shape.h

```
#include "Drawing.h"

class IShape
{
public:
    virtual void Draw() = 0;

protected:
    IDrawing *m_pDraw;
};
```

代码 7.5: Circle.h

```
#include "Shape.h"

class CCircle : public IShape
{
public:
    CCircle(IDrawing *pDraw)
    {
        m_pDraw = pDraw;
    }
    ~CCircle()
    {
        m_pDraw = nullptr;
    }

```

```
15      void Draw()
16      {
17          m_pDraw->DrawCircle();
18      }
19  };
```

代码 7.6: Rectangle.h

```
1
2   #include "Shape.h"
3
4   class CRectangle : public IShape
5   {
6   public:
7       CRectangle(IDrawing *pDraw)
8       {
9           m_pDraw = pDraw;
10      }
11
12      ~CRectangle()
13      {
14          m_pDraw = nullptr;
15      }
16
17      void Draw()
18      {
19          m_pDraw->DrawLine();
20          m_pDraw->DrawLine();
21          m_pDraw->DrawLine();
22          m_pDraw->DrawLine();
23      }
24  };
```

代码 7.7: MainCaller.cpp

```
1   #include "Drawing1.h"
2   #include "Drawing2.h"
3   #include "Circle.h"
4   #include "Rectangle.h"
5
6   int main(int argc, char** argv)
7   {
8       IDrawing *pDraw = new CConcreteDrawing1();
9       IShape *pShape = new CRectangle(pDraw);
10      pShape->Draw();
11      delete pShape;
12      pShape = nullptr;
13      delete pDraw;
14      pDraw = nullptr;
15
```

```
16      pDraw = new CConcreteDrawing2();
17      pShape = new CCircle(pDraw);
18      pShape->Draw();
19      delete pShape;
20      pShape = nullptr ;
21      delete pDraw;
22      pDraw = nullptr ;
23
24      printf ("press any key to exit...\n");
25      getchar () ;
26  }
```

第8章　外观模式

外观模式（Facade）对一些老旧或者涉及多个子系统的代码，定义一个高层接口，为系统中的一组接口提供一个统一的界面，隐藏子系统的复杂性，使子系统更加容易使用。

8.1　UML图

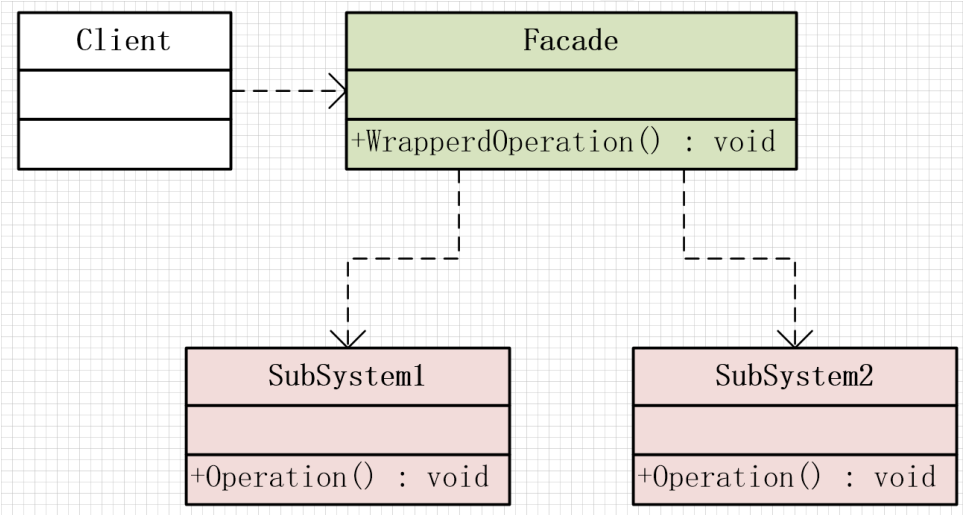

图 8–1

图 8–1 中，类和对象的关系为：

1. Facade：为用户调用而定义的统一接口，将客户的请求传递给相应的子系统处理。
2. 子系统（SubSys）：具体功能的提供者。
3. 调用者（Clients）：通过Facade接口调用提供的功能。

8.2 优点

1. 对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统更加方便使用。
2. 实现了子系统与客户之前的松耦合关系，子系统内部的功能组件往往是紧耦合的。

8.3 缺点

1. 过多的或者是不太合理的 Facade 不如直接调用模块。

8.4 应用场景

1. 为一个复杂子系统提供一个简单接口，子系统的修改不会影响接口。
2. 将子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。
3. 如果子系统之间是相互依赖的，让它们仅通过 Facade 进行通讯，从而简化了它们之间的依赖关系。

8.5 实例

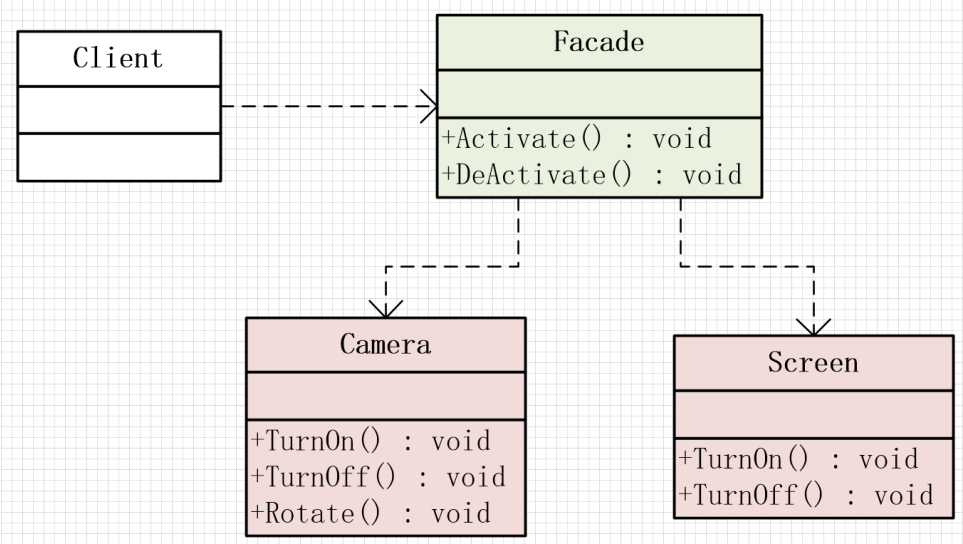

图 8–2

代码 8.1: Screen.h

```
1  #include <stdio.h>
2
3  class CScreen
4  {
5  public:
6      CScreen() {}
7      ~CScreen() {}
8
9      void TrunOn()
10     {
11         printf("Screens on.\n");
12     }
13
14     void TrunOff()
15     {
16         printf("Screens off.\n");
17     }
18 };
```

代码 8.2: Camera.h

```
1  #include <stdio.h>
2
3  class CCamera
4  {
5  public:
6      CCamera() {}
7      ~CCamera() {}
8
```

```
    void TurnOn()
    {
        printf("Camera on.\n");
    }

    void TurnOff()
    {
        printf("Camera off.\n");
    }

    void Rotate(int nDegree)
    {
        printf("Rotate the camera by %d degree.\n", nDegree);
    }
};
```

代码 8.3: Facade.h

```
#include "Camera.h"
#include "Screen.h"

class CFacade
{
public:
    CFacade()
    {
        m_pCamera = new CCamera();
        m_pScreen = new CScreen();
    }
    ~CFacade()
    {
        delete m_pCamera;
        m_pCamera = nullptr;
        delete m_pScreen;
        m_pScreen = nullptr;
    }

    void Activate()
    {
        printf("activate all equipments.\n");
        m_pCamera->TurnOn();
        m_pCamera->Rotate(90);
        m_pScreen->TrunOn();
    }

    void Deactivate()
    {
        m_pScreen->TrunOff();
        m_pCamera->Rotate(0);
        m_pCamera->TurnOff();
```

```
33          printf ("all equipments deactivated.\n");
34      }
35
36  private :
37      CCamera *m_pCamera;
38      CScreen *m_pScreen;
39  };
```

代码 8.4: MainCaller.cpp

```
1   #include "Facade.h"
2
3   int main(int argc, char** argv)
4   {
5       CFacade *pFacade = new CFacade();
6
7       pFacade->Activate();
8
9       printf ("\nlive show.\n\n");
10
11      pFacade->Deactivate();
12
13      printf ("press any key to exit...\n");
14      getchar ();
15  }
```

第9章　复合模式

复合模式（Composite）将对象组合成树形结构以表示“部分-整体”的层次结构，使得客户对单个对象和复合对象的使用具有一致性。复合对象是很多单个对象的“组合”，复合对象与单个对象有共同的特征和操作。

9.1　UML图

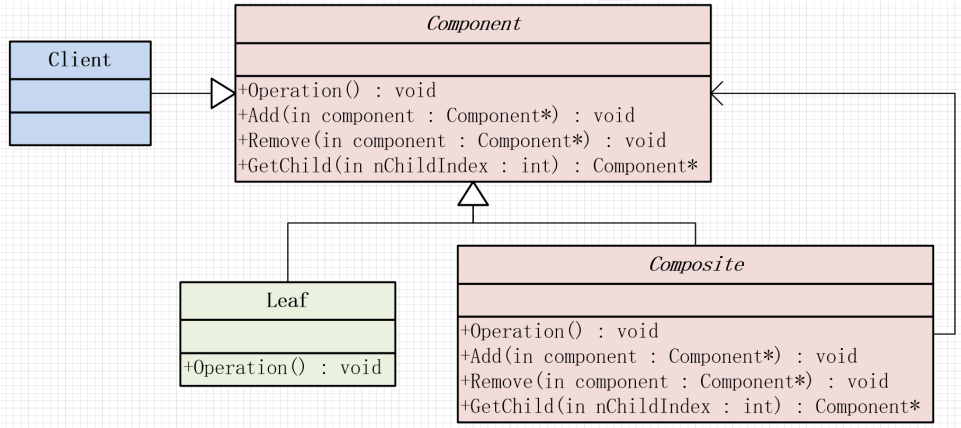

图 9–1

图 9–1 中，类和对象的关系为：

1. 树（Component）：复合对象和单个对象共有特征的抽象，是一个抽象类，有一个对象列表，定义了一些增删对象的操作。
2. 子树（Composite）：代表复合对象，树上有很多子树，子树也是树的一种。
3. 树叶（Leaf）：单个对象， Component中的操作的单个对象，是只有一个结点的树。

9.2 优点

1. 定义了包含基本对象和组合对象的类层次结构，基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，形成递归。
2. 简化客户代码，客户通常不知道处理的是一个叶节点还是一个组合组件，这样简化了客户代码。
3. 更容易增加新类型的组件，客户程序不需因新的 Component 而改变。

9.3 缺点

1. 很难限制组合中的组件，使用 Composite 时不能依赖类型系统施加这些约束，而必须在运行时进行检查。

9.4 应用场景

1. 需要表示对象的“部分-整体”层次结构
2. 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象，在这些对象上执行某个操作。

9.5 实例

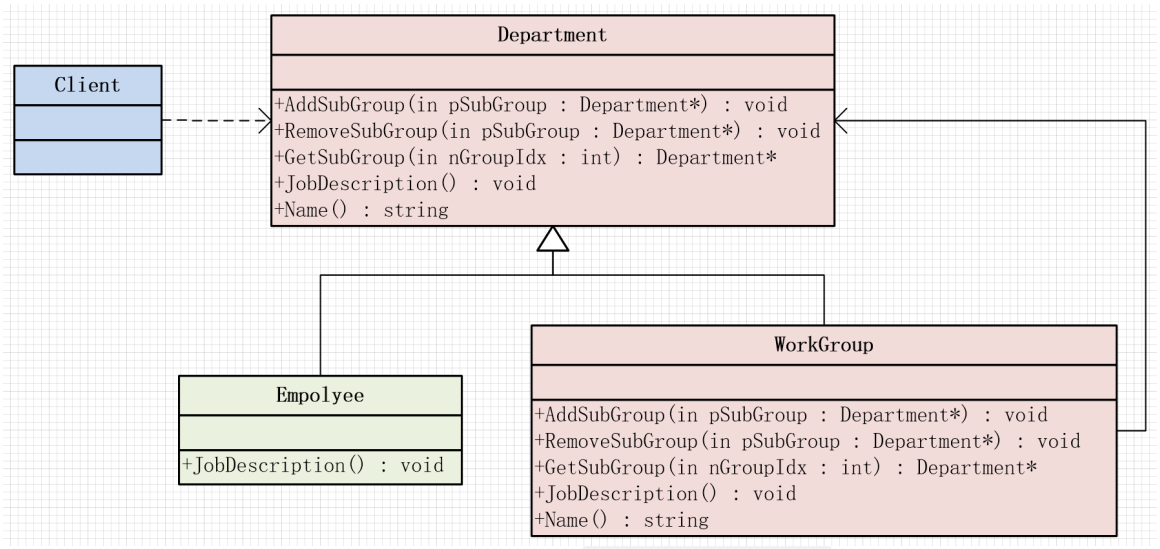

图 9–2

代码 9.1: Department.h

```
1  #include <stdio.h>
2
3  class IDepartment
4  {
5  public:
6      // pure virtual function, all inherited nodes should implement this function
7      virtual void JobDescription () = 0;
8
9      //virtual function, leaf nodes does not override these functions
10     virtual void AddSubGroup(IDepartment*) {}
11     virtual void RemoveSubGroup(IDepartment*) {}
12     virtual IDepartment* GetSubGroup(int nChildIdx) { return nullptr; }
13 };
```

代码 9.2: Employee.h

```
1  #include "Department.h"
2
3  class CEmployee : public IDepartment
4  {
5  public:
6      CEmployee() {}
7      ~CEmployee() {}
8
9      virtual void JobDescription ()
10     {
11         printf ("CEmployee::JobDescription.\n");
12     }
13 };
```

代码 9.3: WorkGroup.h

```
#include <vector>

#include "Department.h"

class CWorkGroup : public IDepartment
{
public:
    CWorkGroup() {}
    ~CWorkGroup() {}

    void JobDescription()
    {
        printf("WorkGroup::JobDescription.\n");

        for (auto iterIdx = m_vecDepartment.begin(); iterIdx != m_vecDepartment.end(); iterIdx++)
        {
            (*iterIdx)->JobDescription();
        }
    }
    void AddSubGroup(IDepartment* pDepartment)
    {
        std::vector<IDepartment*>::iterator iter = find(m_vecDepartment.begin(), m_vecDepartment.end(), pDepartment);
        if (m_vecDepartment.end() == iter)
        {
            m_vecDepartment.push_back(pDepartment);
        }
    }
    void RemoveSubGroup(IDepartment* pDepartment)
    {
        std::vector<IDepartment*>::iterator iter = find(m_vecDepartment.begin(), m_vecDepartment.end(), pDepartment);
        if (m_vecDepartment.end() != iter)
        {
            m_vecDepartment.erase(iter);
        }
    }
    IDepartment* GetSubGroup(int nChildIdx)
    {
        if (nChildIdx < 0 || nChildIdx > m_vecDepartment.size())
        {
            return nullptr;
        }

        return m_vecDepartment[nChildIdx];
    }

private:
```

```
47      std :: vector<IDepartment*> m_vecDepartment;
48  };
```

代码 9.4: MainCaller.cpp

```
1   #include "Employee.h"
2   #include "WorkGroup.h"
3
4   int main(int argc, char** argv)
5   {
6       CWorkGroup *pCompany = new CWorkGroup();
7
8       pCompany->AddSubGroup(new CEmployee());
9
10      CEmployee* pLeaf1 = new CEmployee();
11      CEmployee* pLeaf2 = new CEmployee();
12
13      // meaningless operation
14      pLeaf1->AddSubGroup(pLeaf2);
15      pLeaf1->RemoveSubGroup(pLeaf2);
16
17      // leaf node doing job
18      pLeaf1->JobDescription();
19      printf ("\n\n");
20
21      CWorkGroup* pDepartment = new CWorkGroup();
22      pDepartment->AddSubGroup(pLeaf1);
23      pDepartment->AddSubGroup(pLeaf2);
24      pDepartment->JobDescription();
25      printf ("\n\n");
26
27      pCompany->AddSubGroup(pDepartment);
28      pCompany->JobDescription();
29      printf ("\n\n");
30
31      pDepartment->GetSubGroup(0)->JobDescription();
32      printf ("\n\n");
33
34      pDepartment->RemoveSubGroup(pLeaf1);
35      pDepartment->GetSubGroup(0)->JobDescription();
36
37      printf ("press any key to exit...\n");
38      getchar ();
39  }
```

第10章　装饰模式

装饰模式（Decorator）以对客户透明的方式动态地给对象添加一些额外的功能，以实现在不创造更多子类的情况下，扩展对象的功能。

10.1　UML图

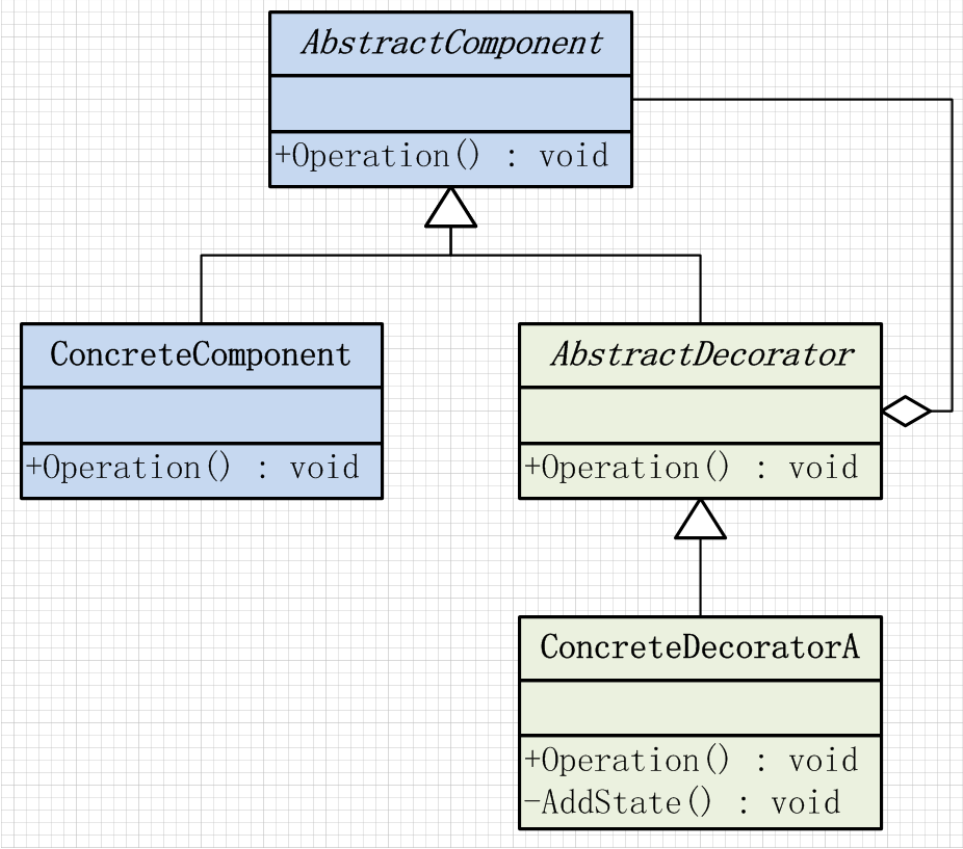

图 10–1

图 10–1 中，类和对象的关系为：

1. 抽象构件（AbstractComponent）：抽象接口，规范要接收附加责任的对象。

2. 具体构件（ConcreteComponent）：定义一个将要接收附加责任的类。
3. 抽象装饰（AbstractDecorator）：持有一个 AbstractComponent 的实例，并定义一个与 AbstractComponent 接口一致的接口
4. 具体装饰（ConcreteDecorator）：给构件对象添加附加的责任。

10.2 优点

1. 在扩展对象功能时，可以比继承更加灵活，同时也需要更少的类。
2. 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以设计出不同的行为。

10.3 缺点

1. 会产生一些为少量特殊功能而定制的极为类似的小型对象。

10.4 应用场景

1. 在不影响其他对象的情况下，以动态和透明的方式给单个对象添加职责。
2. 处理可以撤消的职责。
3. 当不能采用生成子类方法进行扩充时采用此方式。

10.5 实例

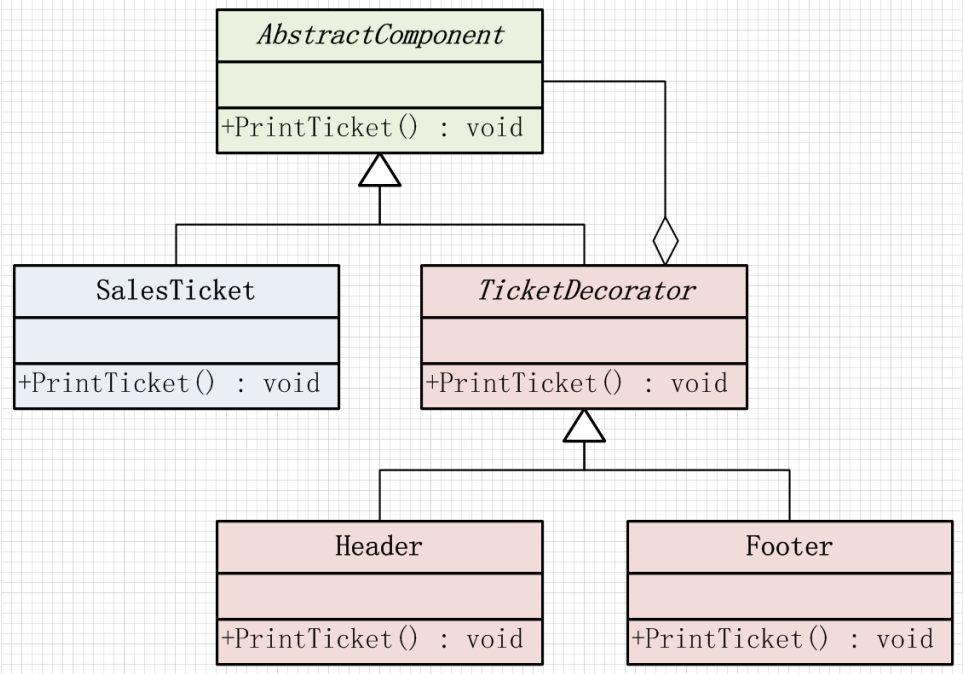

图 10–2

代码 10.1: Component.h

```
#include <stdio.h>

class IComponent
{
public:
    virtual void PrintTicket () = 0;
};
```

代码 10.2: SalesTicket.h

```
#include "Component.h"

class CSalesTicket : public IComponent
{
public:
    CSalesTicket() {}
    ~CSalesTicket() {}

    void PrintTicket ()
    {
        printf ("SalesTicket.\n");
    }
};
```

代码 10.3: TicketDecorator.h

```
#include "Component.h"

class ITicketDecorator : public IComponent
{
public:
    ITicketDecorator()
    {
        m_pDecorateComponent = nullptr;
    }
    ITicketDecorator(IComponent *pDecorateComponent)
    {
        m_pDecorateComponent = pDecorateComponent;
    }
    ~ITicketDecorator()
    {
        m_pDecorateComponent = nullptr;
    }

    virtual void PrintTicket()
    {
        printf("PrintTicket called in ITicketDecorator\n");
        m_pDecorateComponent->PrintTicket();
    }

protected:
    IComponent *m_pDecorateComponent;
};
```

代码 10.4: TicketHeader.h

```
#include "TicketDecorator.h"

class CTicketHeader : public ITicketDecorator
{
public:
    CTicketHeader(IComponent *pDecorateComponent)
    {
        m_pDecorateComponent = pDecorateComponent;
    }
    ~CTicketHeader()
    {
        m_pDecorateComponent = nullptr;
    }

    void PrintTicket()
    {
        printf("Header.\n");
        m_pDecorateComponent->PrintTicket();
    }
```

```
};
```

代码 10.5: TicketFooter.h

```
#include "TicketDecorator.h"

class CTicketFooter : public ITicketDecorator
{
public:
    CTicketFooter(IComponent *pDecorateComponent)
    {
        m_pDecorateComponent = pDecorateComponent;
    }
    ~CTicketFooter()
    {
        m_pDecorateComponent = nullptr;
    }

    void PrintTicket ()
    {
        m_pDecorateComponent->PrintTicket();
        printf ("Footer.\n");
    }
};
```

代码 10.6: MainCaller.cpp

```
#include "Footer.h"
#include "Header.h"
#include "SalesTicket.h"

int main(int argc, char** argv)
{
    IComponent *pTicketInfo = new CSalesTicket();
    IComponent *pTicketFooter = new CTicketFooter( pTicketInfo );
    IComponent *pTicketHeader = new CTicketHeader(pTicketFooter);

    pTicketHeader->PrintTicket();

    delete pTicketHeader;
    pTicketHeader = nullptr;
    delete pTicketFooter;
    pTicketFooter = nullptr;
    delete pTicketInfo;
    pTicketInfo = nullptr;

    printf ("press any key to exit...\n");
    getchar();
}
```

10.6 与复合模式的比较

第11章　享元模式

享元模式（FlyWeight）以共享的方式高效地支持大量的细粒度对象。享元对象能做到共享的关键是区分内部状态（Internal State）和外部状态（External State）：

1. 内部状态：存储在享元对象内部，不会随环境改变而改变的、可以共享的状态。
2. 外部状态：随环境改变而改变的、不可以共享的状态。

享元对象的外部状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。

外部状态与内部状态是相互独立的。

11.1　UML图

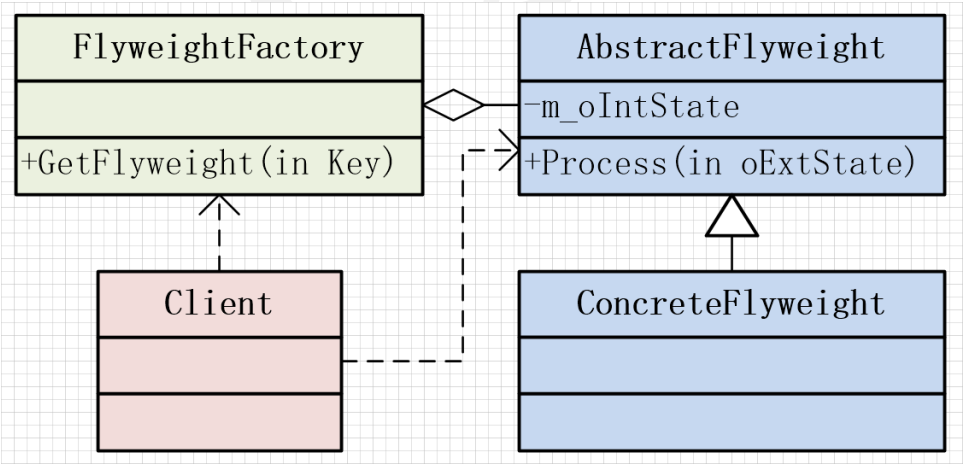

图 11–1

图 11–1 中，类和对象的关系为：

1. 抽象享元（Abstract Flyweight）：定义享元类的公共接口。需要外部状

态的操作可以参数形式传入外部状态。

2. 具体享元（Concrete Flyweight）：实现抽象享元规定的接口。
3. 享元工厂（FlyweightFactory）：负责创建和管理享元。

11.2 优点

1. 大幅度地降低内存中对象的数量。

11.3 缺点

1. 为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
2. 享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

11.4 应用场景

1. 一个系统有大量的，状态可以外部化的，耗费大量的内存的对象。
2. 对象可以按照内部状态分成很多组，当把外部状态从对象中剔除时，每一个组都可以仅用一个对象代替。

11.5 实例

在这个咖啡店（Coffee Shop）所使用的系统里，有一系列的咖啡“风味”（Flavor），屋子里有很多的桌子供客人坐。客人到店里购买咖啡时，选好座位点咖啡，店员做好的咖啡送到座位上。咖啡有内部状态（风味），外部状态（座位）。如果系统为每一杯咖啡和每一个座位都创建一个独立的对象的话，那么就需要创建出很多的细小对象来。享元模式将咖啡按照种类“风味”划分，每一种风味的咖啡只创建一个对象，并实行共享。将座位信息作为外部状态，传送到执行函数内。

使用咖啡摊主的语言来讲，所有的咖啡都可按“风味”划分成如 Capucino、Espresso 等，每一种风味的咖啡不论卖出多少杯，都是全同、不可分辨的。所谓共享，就是咖啡风味的共享，制造方法的共享。

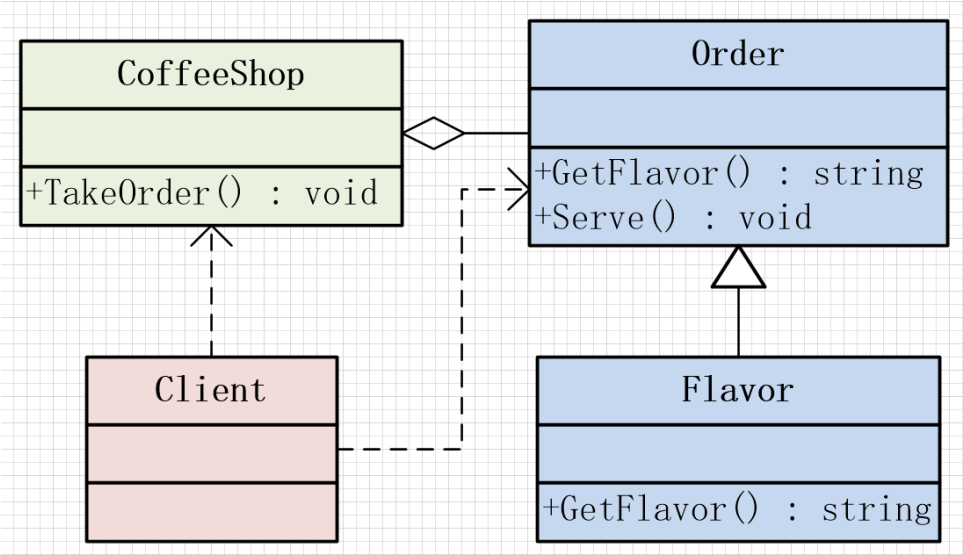

图 11–2

代码 11.1: Order.h

```
1  #include <stdio.h>
2  #include <string>
3
4  class COrder
5  {
6  public:
7      COrder() {}
8      virtual ~COrder() {}
9
10     void Serve(int nTable)
11     {
12         printf("Serve table %d a cup of %s\n", nTable, GetFlavor().c_str());
13     }
14
15     virtual std::string GetFlavor() = 0;
16
17 protected:
18     std::string m_strFlavor;
19 };
```

代码 11.2: Flavor.h

```
#include "Order.h"

class CFlavor : public COrder
{
public:
    CFlavor(std::string strFlavor)
    {
        m_strFlavor = strFlavor;
    }
    ~CFlavor() {}

    std::string GetFlavor() { return m_strFlavor; }
};
```

代码 11.3: CoffeeShop.h

```
#include <map>

#include "Flavor.h"

class CCoffeeShop
{
public:
    CCoffeeShop() { m_setOrder.clear(); }
    ~CCoffeeShop()
    {
        for (auto iter : m_setOrder)
        {
            delete iter.second;
            iter.second = nullptr;
        }

        m_setOrder.clear();
    }

    COrder* TakeOrder(std::string strFlavor)
    {
        if (m_setOrder.find(strFlavor) == m_setOrder.end())
        {
            m_setOrder.insert(std::make_pair(strFlavor, new CFlavor(strFlavor)));
        }

        return m_setOrder[strFlavor];
    }

private:
    std::map<std::string, COrder*> m_setOrder;
};
```

代码 11.4: MainCaller.cpp

```
#include "CoffeeShop.h"
#include "Flavor.h"

using namespace std;

int main(int argc, char** argv)
{
    CCoffeeShop *pShop = new CCoffeeShop;

    COrder *pOrder = pShop->TakeOrder("BlackCoffee");
    pOrder->Serve(1);

    pOrder = pShop->TakeOrder("Capucino");
    pOrder->Serve(5);

    delete pShop;
    pShop = nullptr;

    printf("press any key to exit...\n");
    getchar();
}
```

11.6 与工厂模式的区别

第12章　代理模式

代理模式（Proxy）为其他对象提供一个代理以控制对这个对象的访问，代理可用于解决在直接访问对象不方便或不符合要求时，为这个对象提供一种代理，以控制对该对象的访问。

12.1　UML图

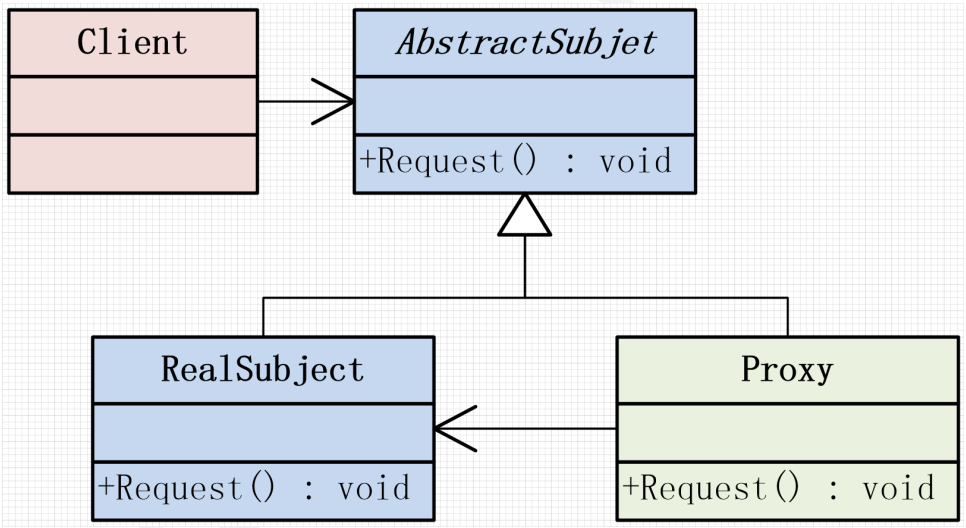

图 12–1

图 12–1 中，类和对象的关系为：

1. AbstractSubject：声明了真实主题和代理主题的共同接口。
2. Proxy：代理主题角色，内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象。代理提供一个与真实角色相同的接口。
3. RealSubject：定义了代理角色所代表的真实对象。

12.2 优点

1. 职责清晰，真实的角色就是实现实际的业务逻辑，不关心其他非本职的事务，通过代理完成一件完整事务。
2. 高扩展性，真实角色可以发现变化，代理类完全就可以在不做任何修改的情况下使用。
3. 协调调用者和被调用者，降低了系统的耦合度。
4. 代理对象作为客户端和目标对象之间的中介，起到了保护目标对象的作用。

12.3 缺点

1. 由于在客户端和真实主题之间增加了代理对象，因此会造成请求的处理速度变慢。
2. 实现代理模式需要额外的工作（有些代理模式的实现非常复杂），从而增加了系统实现的复杂度。

12.4 应用场景

1. 为一个对象再不同的地址空间提供局部的代表。
2. 创建新的对象时开销非常大，用代理减少系统的开销。
3. 访问对象时附加额外操作：在不影响对象类的情况下，在访问对象时进行更多的操作。

12.5 实例

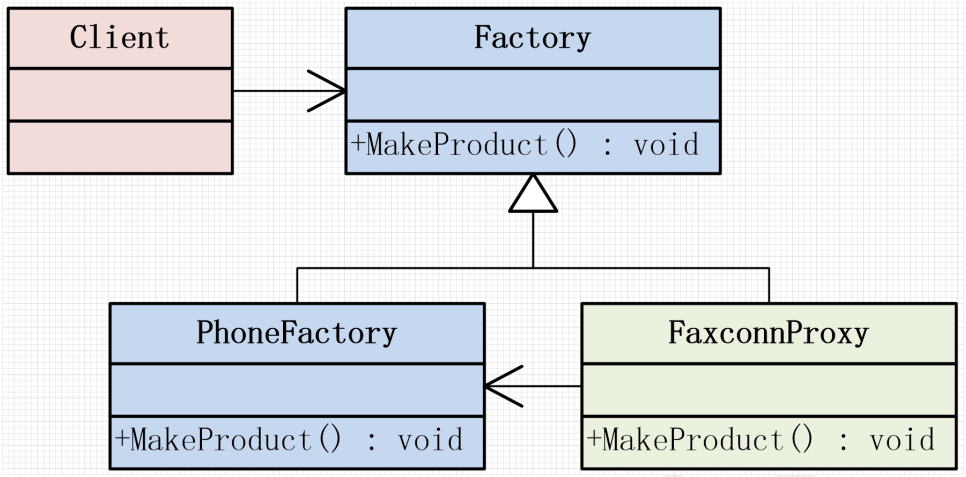

图 12–2

代码 12.1: Factory.h

```
1  class IFactory
2  {
3  public:
4      virtual void MakeProduct() = 0;
5  };
```

代码 12.2: PhoneFactory.h

```
1  #include <stdio.h>
2
3  #include "Factory.h"
4
5  class CPhoneFactory : public IFactory
6  {
7  public:
8      CPhoneFactory() {}
9      ~CPhoneFactory() {}
10
11     void MakeProduct()
12     {
13         printf("Phone Factory.\n");
14     }
15 };
```

代码 12.3: FoxconnProxy.h

```
1  #include "Factory.h"
2
3  class CFoxconnProxy : public IFactory
4  {
5  public:
```

```
6       CFoxconnProxy()
7       {
8           m_pFactory = nullptr;
9       }
10      CFoxconnProxy(IFactory *pFactory)
11      {
12          m_pFactory = pFactory;
13      }
14      ~CFoxconnProxy()
15      {
16          m_pFactory = nullptr;
17      }
18
19      void MakeProduct()
20      {
21          m_pFactory->MakeProduct();
22      }
23
24  private:
25      IFactory *m_pFactory;
26  };
```

代码 12.4: MainCaller.cpp

```
1   #include "FaxconnProxy.h"
2   #include "PhoneFactory.h"
3
4   int main(int argc, char** argv)
5   {
6       IFactory *pFactory = new CPhoneFactory();
7       CFoxconnProxy *pProxy = new CFoxconnProxy(pFactory);
8       pProxy->MakeProduct();
9
10      printf("press any key to exit...\n");
11      getchar();
12  }
```

12.6 与装饰模式的区别

第三部分

行为型模式

第13章　观察者模式

观察者模式（Observer）定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知并自动更新。观察者模式又被称作发布-订阅（Publish-Subscribe）模式，是一种对象的行为模式。

13.1　UML图

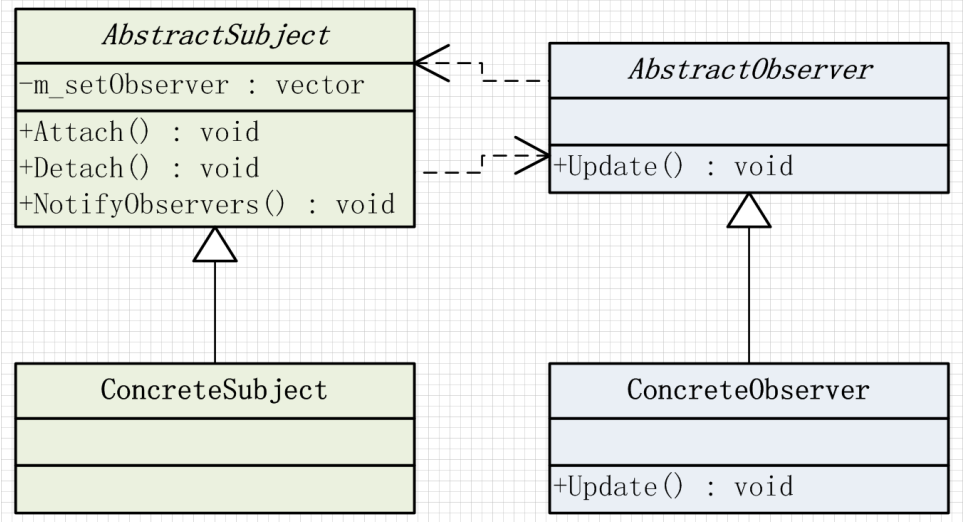

图 13–1

图 13–1 中，类和对象的关系为：

1. 抽象主题（AbstractSubject）：把所有观察者对象的引用保存在一个列表里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。
2. 具体主题（ConcreteSubject）：将有关状态存入具体观察者对象，在具体主题的内部状态改变时，给所有登记过的观察者发出通知。

3. 抽象观察者（AbstractObserver）：为所有具体观察者定义一个接口，在得到主题的通知时更新自己。
4. 具体观察者（ConcreteObserver）：存储与主题的状态自恰的状态。

13.2 优点

1. Subject 和 Observer 之间的耦合是抽象的和最小的。
2. Subject发送的通知不需要指定接收者，被自动广播给所有已向该目标对象登记的有关对象。

13.3 缺点

1. 一个 Observer 不知道其它 Observer 的存在，可能不知道改变 Subject 的最终代价。
2. Subject 上一个操作可能会引起一系列对 Observer 的更新。

13.4 应用场景

1. 一个抽象模式有两个方面，其中一个方面依赖于另一个方面。
2. 当对一个对象的改变需要同时改变其它对象，并且不知道具体有多少对象有待改变。
3. 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

13.5 实例

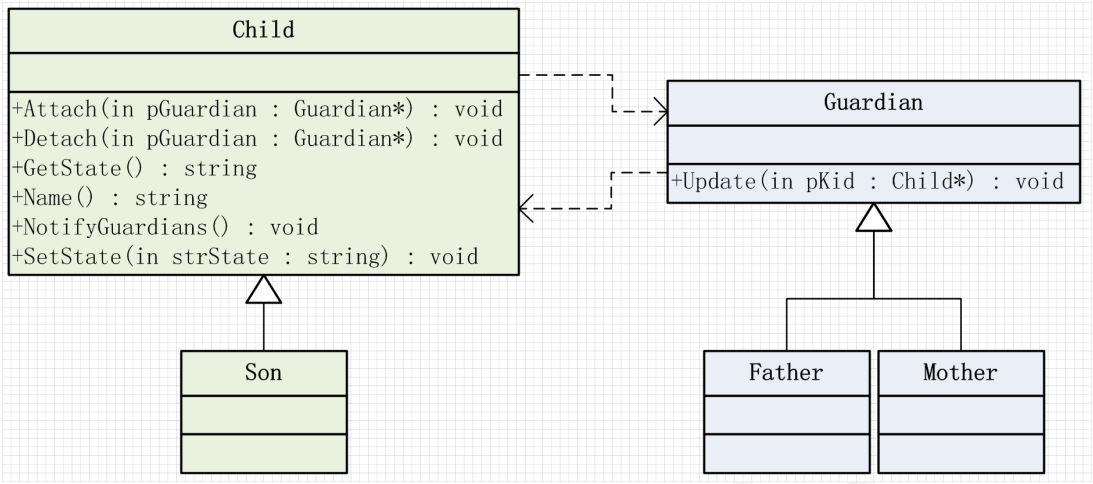

图 13–2

代码 13.1: Guardian.h

```
1  #include "Child.h"
2
3  /* important */
4  class IChild;
5
6  class IGuardian
7  {
8  public:
9      virtual void Update(IChild *pKid) = 0;
10 };
```

代码 13.2: Child.h

```
1  #include <stdio.h>
2  #include <string>
3  #include <set>
4
5  #include "Guardian.h"
6
7  /* important */
8  class IGuardian;
9
10 class IChild
11 {
12 public:
13     virtual void Attach(IGuardian* pGuardian) = 0;
14     virtual void Detach(IGuardian* pGuardian) = 0;
15     virtual std::string GetState() = 0;
16     virtual std::string Name() = 0;
17     virtual void NotifyGuardians() = 0;
18     virtual void SetState(std::string strState) = 0;
```

```
19
20 protected :
21     std :: set <IGuardian*> m_setGuardians;
22     std :: string   m_strState ;
23 };
```

代码 13.3: Son.h

```
1  #include "Child.h"
2
3  class CSon : public IChild
4  {
5  public :
6      CSon() {}
7      ~CSon() {}
8
9      void Attach(IGuardian* pGuardian)
10     {
11         if (m_setGuardians.find(pGuardian) == m_setGuardians.end())
12         {
13             m_setGuardians.insert(pGuardian);
14         }
15     }
16
17     void Detach(IGuardian* pGuardian)
18     {
19         if (m_setGuardians.find(pGuardian) != m_setGuardians.end())
20         {
21             m_setGuardians.erase(pGuardian);
22         }
23     }
24
25     std :: string   GetState()
26     {
27         return   m_strState ;
28     }
29
30     std :: string   Name(){ return "Tom"; }
31
32     void NotifyGuardians()
33     {
34         for (auto iter = m_setGuardians.begin(); iter != m_setGuardians.end(); iter
           ++)
35         {
36             (* iter )->Update((IChild*)this);
37         }
38     }
39
40     void SetState ( std :: string   strState )
41     {
```

```
42          m_strState = strState;
43      }
44  };
```

代码 13.4: Father.h

```
1
2   #include "Guardian.h"
3
4   class CFather : public IGuardian
5   {
6   public:
7       CFather() {}
8       ~CFather() {}
9
10      void Update(IChild *pKid)
11      {
12          printf("father notices that %s %s.\n", pKid->Name().c_str(), pKid
            ->GetState().c_str());
13      }
14  };
```

代码 13.5: Mother.h

```
1
2   #include "Guardian.h"
3
4   class CMother: public IGuardian
5   {
6   public:
7       CMother() {}
8       ~CMother() {}
9
10      void Update(IChild *pKid)
11      {
12          printf("mother notices that %s %s.\n", pKid->Name().c_str(), pKid
            ->GetState().c_str());
13      }
14  };
```

代码 13.6: MainCaller.cpp

```
1   #include "Father.h"
2   #include "Mother.h"
3   #include "Son.h"
4
5   int main(int argc, char** argv)
6   {
7       IChild *pKid = new CSon();
8       IGuardian *pFather = new CFather();
```

```
9       IGuardian *pMother = new CMother();
10
11      pKid->Attach(pFather);
12      pKid->Attach(pMother);
13
14      printf("\n");
15      pKid->SetState("fight with others");
16      pKid->NotifyGuardians();
17
18      printf("\n");
19      pKid->Detach(pMother);
20      delete pMother;
21      pMother = nullptr;
22
23      pKid->SetState("got a scholarship");
24      pKid->NotifyGuardians();
25
26      printf("\n");
27      pKid->Detach(pFather);
28      delete pFather;
29      pFather = nullptr;
30
31      delete pKid;
32      pKid = nullptr;
33
34      printf("press any key to exit...\n");
35      getchar();
36  }
```

第14章　命令模式

命令模式（Command）将一个请求封闭为一个对象，从而可以用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可取消的操作。命令模式是一种对象行为模式，可以对发送者和接收者完全解耦。

14.1　UML图

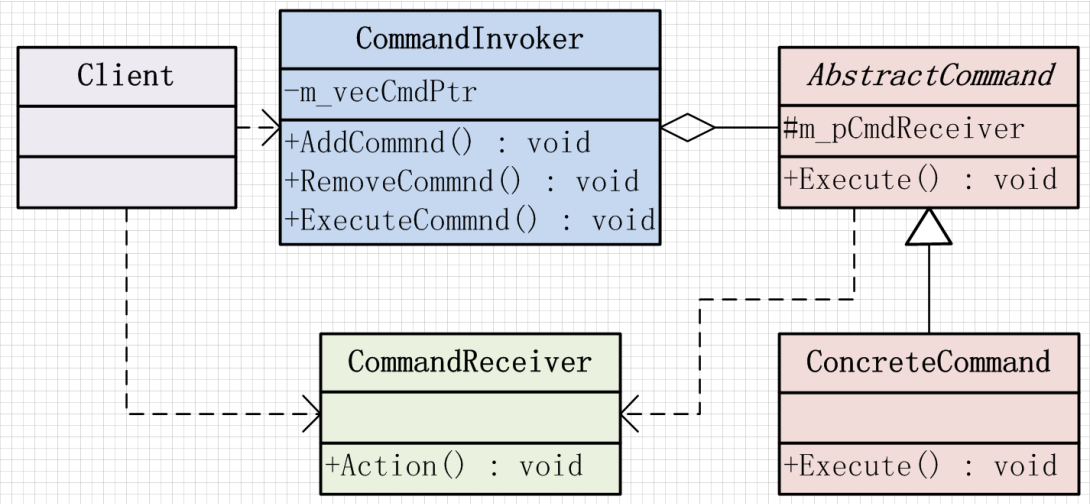

图 14–1

图 14–1 中，类和对象的关系为：

1. 抽象命令（Abstract Command）：声明了一个给所有具体命令类的抽象接口。
2. 具体命令（Concrete Command）：定义一个接收者和行为之间的弱耦合，实现 execute 方法，负责调用接收者的相应操作。
3. 请求者（Invoker）：负责调用命令对象执行请求。
4. 接收者（Receiver）：负责具体实施和执行一个请求。

14.2 优点

1. 新的命令很容易地被加入到系统中，并且不影响其它类。
2. 允许接收请求的一方决定是否要否决请求。
3. 能较容易地设计一个命令队列。
4. 可以容易地实现对请求的“撤消”和“重做”。
5. 在需要的情况下，可以较容易地将命令记入日志。
6. 把请求一个操作的对象与知道怎么执行一个操作的对象分割开。

14.3 缺点

1. 系统有过多的具体 Command 类。
2. CommandReceiver 需要实现所有的终端执行操作。

14.4 应用场景

1. 使用命令模式作为“CallBack”在面向对象系统中的替代。
2. 需要在不同的时间指定请求、将请求排队。
3. 一个命令对象和原先的请求发出者可以有不同的生命期：原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。
4. 系统需要支持命令的撤消（undo）：命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo 方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo 方法，以供客户端在需要时，再重新实施命令效果。
5. 一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用 execute 方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

14.5 实例

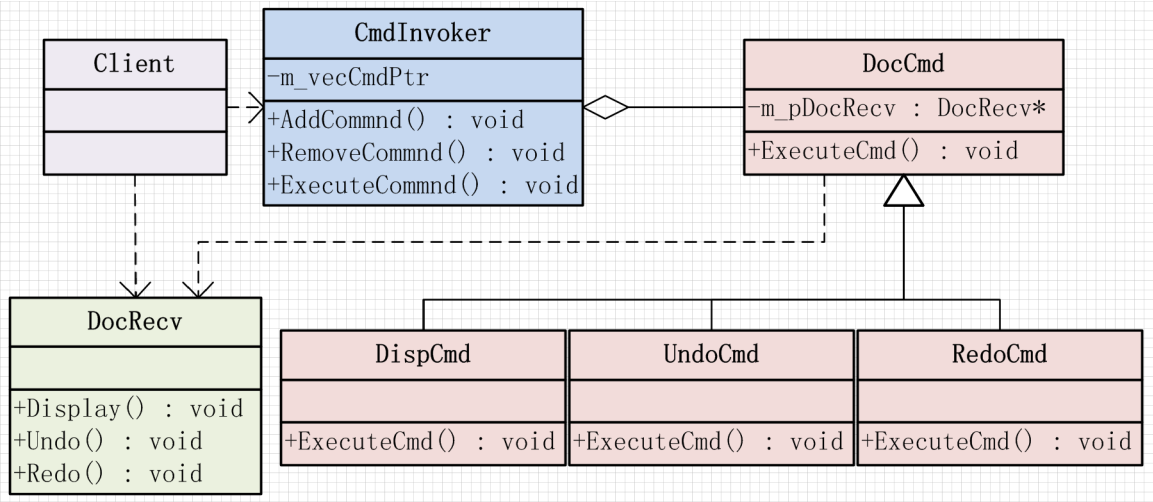

图 14–2

代码 14.1: DocReceiver.h

```
1  #include <stdio.h>
2
3  class CDocRecv
4  {
5  public:
6      CDocRecv() {}
7      ~CDocRecv() {}
8
9      void Display()
10     {
11         printf("show the context of the doc.\n");
12     }
13     void Undo()
14     {
15         printf("cancel the context of the doc.\n");
16     }
17     void Redo()
18     {
19         printf("redo the context of the doc.\n");
20     }
21 };
```

代码 14.2: DocCmd.h

```
#include "DocReceiver.h"

class IDocCmd
{
public:
    virtual void ExecuteCmd() = 0;

protected:
    CDocRecv* m_pDocRecv;
};
```

```
#include <set>

#include "DocCmd.h"

class CCmdInvoker
{
public:
    CCmdInvoker() {}
    ~CCmdInvoker() {}

    void AddCmd(IDocCmd *pDocCmd)
    {
        if (m_setCmd.find(pDocCmd) == m_setCmd.end())
        {
            m_setCmd.insert(pDocCmd);
        }
    }
    void RemoveCmd(IDocCmd *pDocCmd)
    {
        if (m_setCmd.find(pDocCmd) != m_setCmd.end())
        {
            m_setCmd.erase(pDocCmd);
        }
    }
    void ExecuteCmd()
    {
        for (auto pCmd : m_setCmd)
        {
            pCmd->ExecuteCmd();
        }
    }

private:
    std::set<IDocCmd*> m_setCmd;
};
```

代码 14.3: DispCmd.h

```
#include "DocCmd.h"

class CDispCmd : public IDocCmd
{
public:
    CDispCmd(CDocRecv* pDoc)
    {
        m_pDocRecv = pDoc;
    }
    ~CDispCmd()
    {
        m_pDocRecv = nullptr;
    }

    void ExecuteCmd()
    {
        m_pDocRecv->Display();
    }
};
```

代码 14.4: RedoCmd.h

```
#include "DocCmd.h"

class CRedoCmd : public IDocCmd
{
public:
    CRedoCmd(CDocRecv* pDoc)
    {
        m_pDocRecv = pDoc;
    }
    ~CRedoCmd()
    {
        m_pDocRecv = nullptr;
    }

    void ExecuteCmd()
    {
        m_pDocRecv->Redo();
    }
};
```

代码 14.5: UndoCmd.h

```
#include "DocCmd.h"

class CUndoCmd : public IDocCmd
{
public:
    CUndoCmd(CDocRecv* pDoc)
    {
        m_pDocRecv = pDoc;
    }
    ~CUndoCmd()
    {
        m_pDocRecv = nullptr;
    }

    void ExecuteCmd()
    {
        m_pDocRecv->Undo();
    }
};
```

代码 14.6: MainCaller.cpp

```
#include "DispCmd.h"
#include "RedoCmd.h"
#include "UndoCmd.h"
#include "CmdInvoker.h"

int main(int argc, char **argv)
{
    CDocRecv *pDoc = new CDocRecv();

    IDocCmd *pDisp = new CDispCmd(pDoc);
    IDocCmd *pUndo = new CUndoCmd(pDoc);
    IDocCmd *pRedo = new CRedoCmd(pDoc);

    CCmdInvoker *pCmdInvoker = new CCmdInvoker();
    pCmdInvoker->AddCmd(pDisp);
    pCmdInvoker->AddCmd(pUndo);
    pCmdInvoker->AddCmd(pRedo);

    pCmdInvoker->ExecuteCmd();

    delete pCmdInvoker;
    pCmdInvoker = nullptr;

    printf("press any key to exit...\n");
    getchar();
}
```

第15章　责任链模式

责任链模式（Responsibility Chain）避免了请求的发送者和接收者之间的耦合关系，使多个接受对象都有机会处理请求。在不止一个对象可以处理客户端请求的时候，为了使每个对象都有机会处理请求，将这些对象连成一条链（每个被串连的对象都有一个指向下一个对象的指针），当有请求时，当前对象处理该请求或者向下一个对象传递该请求，直到有一个对象处理它为止，并且该请求必须被处理。

责任链模式并不创建责任链，责任链的创建必须由系统的其它部分创建出来。

15.1　UML图

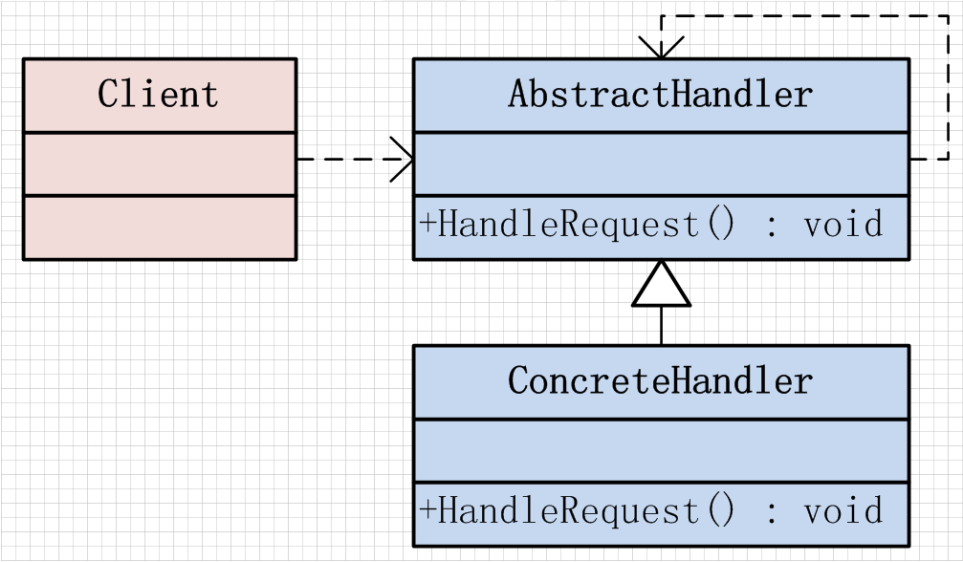

图 15–1

图 15–1 中，类和对象的关系为：

1. 抽象处理者（AbstractHandler）：定义出一个处理请求的接口，如果需

要，定义一个方法，以设定和返回对下家的引用。

2. 具体处理者（ConcreteHandler）：接到请求后，可以选择将请求处理掉或者将请求转发给下家。

15.2 优点

1. 发出这个请求的客户端并不知道链上的哪一个具体对象最终处理这个请求，使得系统可以在不影响客户端的情况下动态地重新组合链和分配责任，以降低处理请求与处理对象以及处理对象之间的耦合关系。

15.3 缺点

1. 责任链需要由系统的其它部分创建出来，需要上层应用的维护。
2. 构建链应遵循一定的规则：由特殊到普通。

15.4 应用场景

1. 多个的对象可以处理一个请求，哪个对象处理该请求运行时刻确定。
2. 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
3. 处理一个请求的对象集合需要动态指定。

15.5 实例

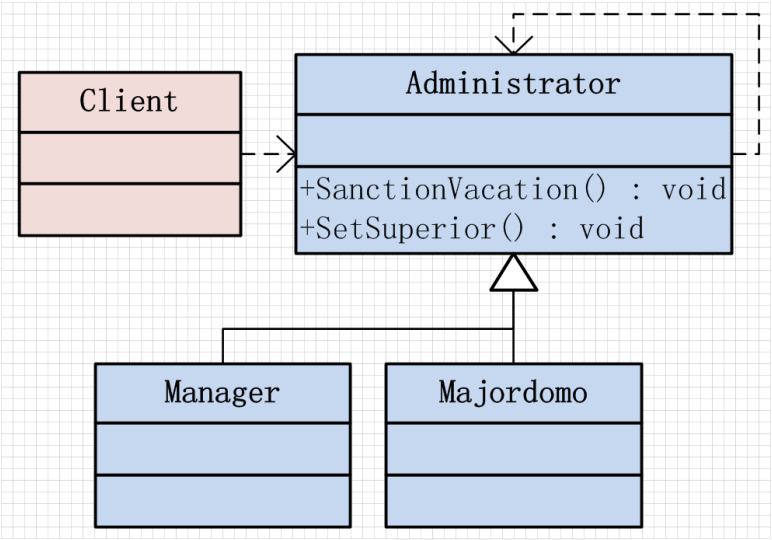

图 15–2

代码 15.1: Administrator.h

```
#include <stdio.h>

class IAdministrator
{
public:
    virtual void SanctionVacation(int nDays) = 0;
    virtual void SetSuperior(IAdministrator *pSuperiorPtr) = 0;

protected:
    IAdministrator *m_pSuperiorPtr;
};
```

代码 15.2: Majordomo.h

```
#include "Administrator.h"

class CMajordomo : public IAdministrator
{
public:
    CMajordomo()
    {
        m_pSuperiorPtr = nullptr;
    }
    ~CMajordomo()
    {
        m_pSuperiorPtr = nullptr;
    }

    void SanctionVacation(int nDays)
```

```
16      {
17          if (nDays <= 10)
18          {
19              printf ("%d days vacation approved\n", nDays);
20          }
21          else
22          {
23              printf ("%d days vacation denied\n", nDays);
24          }
25      }
26
27      void SetSuperior( IAdministrator *pSuperiorPtr ) {}
28  };
```

代码 15.3: Manager.h

```
1   #include "Administrator.h"
2
3   class CManager : public IAdministrator
4   {
5   public:
6       CManager()
7       {
8           m_pSuperiorPtr = nullptr;
9       }
10      ~CManager()
11      {
12          m_pSuperiorPtr = nullptr;
13      }
14
15      void SanctionVacation( int nDays)
16      {
17          if (nDays <= 3)
18          {
19              printf ("manager sanction %d day(s) vacation\n", nDays);
20          }
21          else
22          {
23              printf ("%d days vacation is beyond manager's pay grade,
                ask for majordomo's approval\n", nDays);
24              m_pSuperiorPtr->SanctionVacation(nDays);
25          }
26      }
27
28      void SetSuperior( IAdministrator *pSuperiorPtr )
29      {
30          m_pSuperiorPtr = pSuperiorPtr;
31      }
32  };
```

代码 15.4: MainCaller.cpp

```
#include "Majordomo.h"
#include "Manager.h"

int main(int argc, char **argv)
{
    CMajordomo *pMajordomo = new CMajordomo();
    CManager *pManager = new CManager();
    pManager->SetSuperior(pMajordomo);

    pManager->SanctionVacation(2);
    pManager->SanctionVacation(8);
    pManager->SanctionVacation(25);

    delete pManager;
    pManager = nullptr;
    delete pMajordomo;
    pMajordomo = nullptr;

    printf("press any key to exit...\n");
    getchar();
}
```

第16章　解释器模式

解释器模式（Interpreter）：给定一个语言，定义它的方法的一种表示并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

16.1　UML图

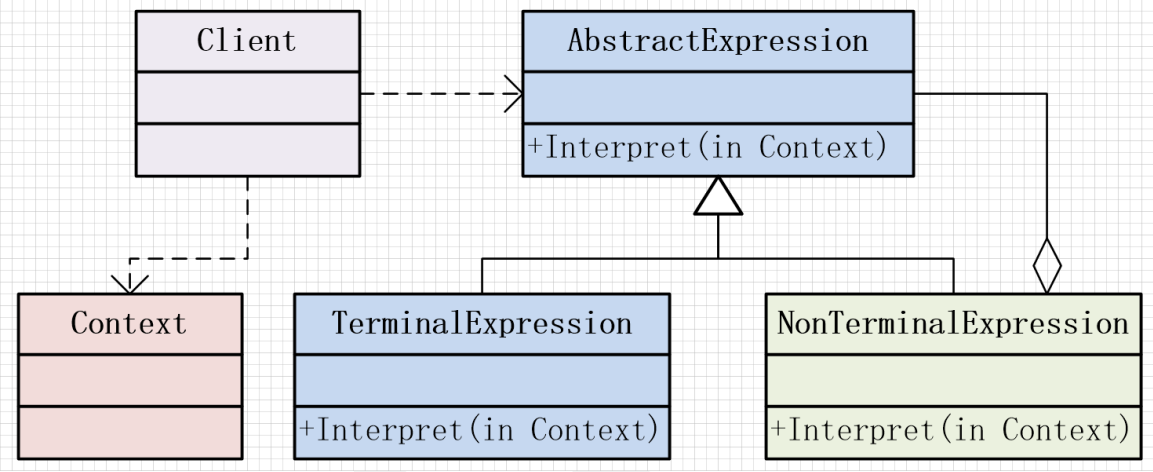

图 16–1

图 16–1 中，类和对象的关系为：

1. 抽象表达式（Abstract Expression）：声明一个抽象的解释操作，这个接口为所有具体表达角色都要实现的。
2. 终结符表达式（TerminalExpression）：实现与方法中的终结符相关联的解释操作。每个终结符需要该类的一个实例与之对应。
3. 非终结符表达式（NonterminalExpression）：递归地调用方法中的每条规则的解释。
4. 上下文环境（Context）：包含解释器之外的一些全局信息，用来存储解释器的上下文环境。

5. 客户角色（Client）：该方法定义的语言中的一个特定的句子的抽象语法树，调用解释操作。

16.2 优点

1. 提供了一个简单的方法来执行语法，而且容易修改或者扩展。

16.3 缺点

1. 不同的规则是由不同的类来实现的，使得添加一个新的语法规则变得简单。

16.4 应用场景

1. 方法简单，无需构建抽象语法树即可解释表达式，可以节省空间还可以节省时间。
2. 效率不是一个关键问题的情况。

16.5 实例

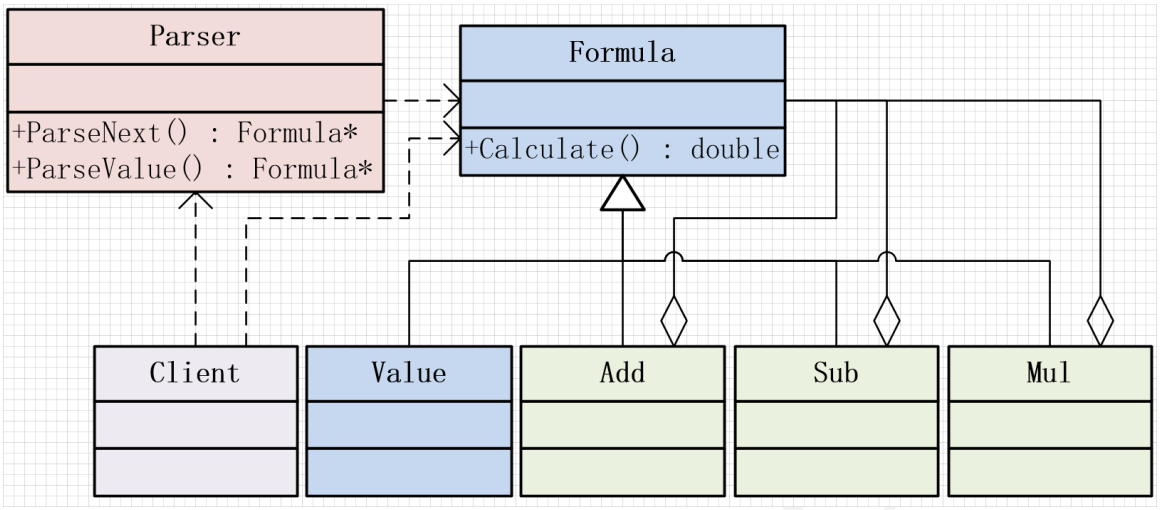

图 16–2

代码 16.1: Formula.h

```
1  class IFormula
2  {
3  public:
4      virtual double Calculate() = 0;
5  };
```

代码 16.2: Add.h

```
1  #include "Formula.h"
2
3  class CAdd : public IFormula
4  {
5  public:
6      CAdd(IFormula *pExp1, IFormula *pExp2)
7      {
8          m_pExp1 = pExp1;
9          m_pExp2 = pExp2;
10     }
11     ~CAdd()
12     {
13         m_pExp1 = nullptr;
14         m_pExp2 = nullptr;
15     }
16
17     double Calculate()
18     {
19         return m_pExp1->Calculate() + m_pExp2->Calculate();
20     }
21
22 private:
```

```
23      IFormula *m_pExp1;
24      IFormula *m_pExp2;
25  };
```

代码 16.3: Mul.h

```
1   #include "Formula.h"
2
3   class CMul : public IFormula
4   {
5   public:
6       CMul(IFormula *pExp1, IFormula *pExp2)
7       {
8           m_pExp1 = pExp1;
9           m_pExp2 = pExp2;
10      }
11      ~CMul()
12      {
13          m_pExp1 = nullptr;
14          m_pExp2 = nullptr;
15      }
16
17      double Calculate ()
18      {
19          return m_pExp1->Calculate() * m_pExp2->Calculate();
20      }
21
22  private:
23      IFormula *m_pExp1;
24      IFormula *m_pExp2;
25  };
```

代码 16.4: Sub.h

```
1   #include "Formula.h"
2
3   class CSub : public IFormula
4   {
5   public:
6       CSub(IFormula *pExp1, IFormula *pExp2)
7       {
8           m_pExp1 = pExp1;
9           m_pExp2 = pExp2;
10      }
11      ~CSub()
12      {
13          m_pExp1 = nullptr;
14          m_pExp2 = nullptr;
15      }
16
```

```
17      double Calculate ()
18      {
19          return m_pExp1->Calculate() - m_pExp2->Calculate();
20      }
21
22  private :
23      IFormula *m_pExp1;
24      IFormula *m_pExp2;
25  };
```

代码 16.5: Value.h

```
1   #include "Formula.h"
2
3   class CValue : public IFormula
4   {
5   public :
6       CValue(double dValue)
7       {
8           m_dValue = dValue;
9       }
10      ~CValue()
11      {
12      }
13
14      double Calculate () { return m_dValue; }
15
16  private :
17      double m_dValue;
18  };
```

代码 16.6: Parser.h

```
1   #include <list>
2   #include <string>
3
4   #include "Add.h"
5   #include "Mul.h"
6   #include "Sub.h"
7   #include "Value.h"
8
9   class CParser
10  {
11  public :
12      CParser() {}
13      ~CParser() {}
14
15      IFormula* ParseNext(std :: list <std :: string >& listFormulaElement)
16      {
17          try
```

```
18          {
19              m_dValue = stof(listFormulaElement.front(), nullptr);
20              listFormulaElement.pop_front();
21              return new CValue(m_dValue);
22          }
23          catch (...)
24          {
25              return ParseSymbol(listFormulaElement);
26          }
27      }
28
29      IFormula* ParseSymbol(std::list<std::string>& listFormulaElement)
30      {
31          std::string strSymbol = listFormulaElement.front();
32          listFormulaElement.pop_front();
33
34          IFormula *pVal1 = ParseNext(listFormulaElement);
35          IFormula *pVal2 = ParseNext(listFormulaElement);
36
37          switch (strSymbol[0])
38          {
39          case '+':
40              return new CAdd(pVal1, pVal2);
41          case '-':
42              return new CSub(pVal1, pVal2);
43          case '*':
44              return new CMul(pVal1, pVal2);
45          default:
46              return nullptr;
47          }
48      }
49
50  private:
51      double m_dValue;
52  };
```

代码 16.7: MainCaller.cpp

```
1   #include "Formula.h"
2   #include "Parser.h"
3
4   using namespace std;
5
6   void SplitString(string& strSrc, string& strSplitter, list<string>& listSplittedStr
        )
7   {
8       // index of character in string
9       string::size_type nSubStrIdxStart = 0;
10      string::size_type nSubStrIdxStop = 0;
11      string::size_type nSubStrIdxTemp = 0;
```

```
12
13          listSplittedStr .clear();
14
15      do
16      {
17          // find first matched splitter from
18          nSubStrIdxStop = strSrc.find_first_of(strSplitter, nSubStrIdxStart);
19          if (nSubStrIdxStop != string::npos)
20          {
21              nSubStrIdxTemp = nSubStrIdxStop;
22              while (' ' == strSrc[nSubStrIdxTemp - 1])
23              {
24                  nSubStrIdxTemp--;
25              }
26              listSplittedStr.push_back(strSrc.substr(nSubStrIdxStart,
                nSubStrIdxTemp - nSubStrIdxStart));
27
28              do
29              {
30                  nSubStrIdxStart = strSrc.find_first_not_of(strSplitter, ++
                nSubStrIdxStop);
31              } while (string::npos != nSubStrIdxStart && ' ' == strSrc[
                nSubStrIdxStart]);
32
33              if (string::npos == nSubStrIdxStart)
34              {
35                  return;
36              }
37          }
38          else
39          {
40              listSplittedStr.push_back(strSrc.substr(nSubStrIdxStart));
41          }
42      } while (nSubStrIdxStop != string::npos);
43  }
44
45  int main(int argc, char** argv)
46  {
47      CParser *pParser = new CParser();
48
49      string strFormula = "+ - + - - 2 3 4 + - -5 6 + -7 8 9 0";
50      list<string> listFormula;
51      SplitString(strFormula, " ", listFormula);
52      IFormula *pFormula = pParser->ParseNext(listFormula);
53
54      printf("Polish Notation \"%s\" = %1.2f\n", strFormula.c_str(), pFormula
            ->Calculate());
55
56      delete pFormula;
57      pFormula = nullptr;
58      delete pParser;
```

```
59      pParser = nullptr;
60
61      printf("press any key to exit...\n");
62      getchar();
63  }
```

16.6 和复合模式的区别

第17章　迭代器

迭代器模式（Iterator）提供一个方法，顺序访问一个聚合对象的各个元素，而不暴露该对象的内部表示。

17.1　UML图

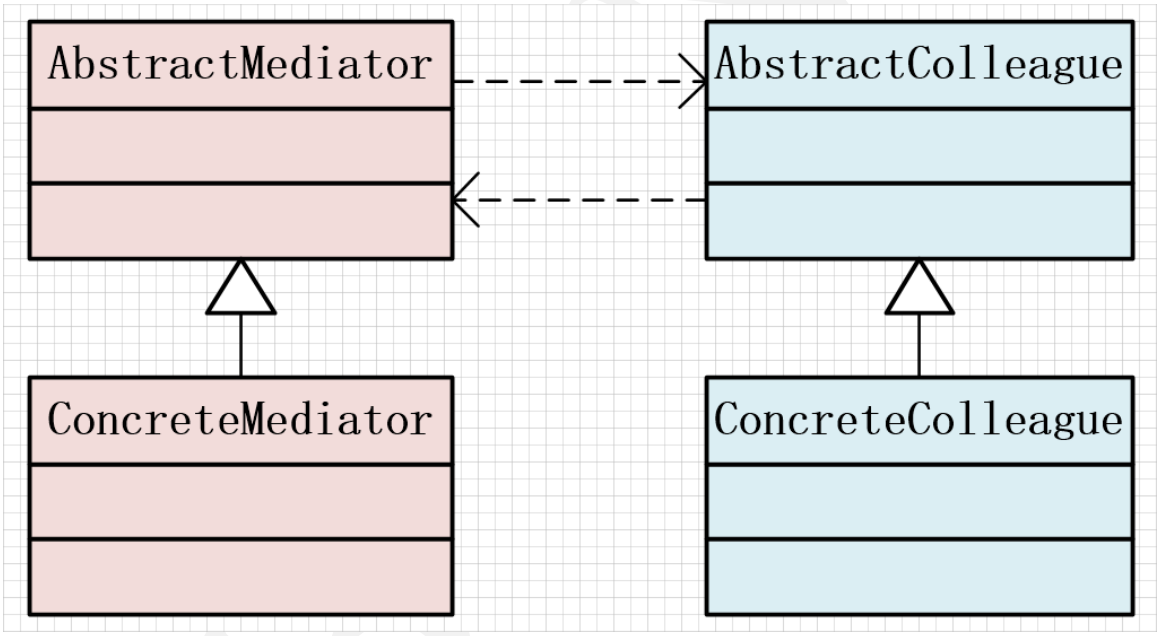

图 17–1

图 17–1 中，类和对象的关系为：

1. 抽象迭代器（Abstract Iterator）：定义实现迭代功能的最小定义方法集.
2. 具体迭代器（Concrete Iterator）
3. 容器接口（Abstract Aggregate）：定义基本功能以及提供类似Iterator的方法。
4. 容器实现（Concrete Aggregate）

17.2 优点

1. 简化遍历方式。
2. 提供多种遍历方式。
3. 封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

17.3 缺点

1. 对于比较简单的遍历，使用迭代器方式遍历较为繁琐。

17.4 应用场景

1. 实现一个集合，就需要同时提供这个集合的迭代器。

17.5 实例

UML类图参考 17–1。

代码 17.1: AbstractIterator.h

```
1  #include <string>
2
3  class IAbstractIterator
4  {
5  public:
6      virtual std::string First() = 0;
7      virtual std::string Next() = 0;
8      virtual std::string Current() = 0;
9      virtual bool IsEnd() = 0;
10 };
```

代码 17.2: AbstractAggregate.h

```
1  #include "AbstractIterator.h"
2
3  class IAbstractAggregate
4  {
```

```
5  public:
6      virtual unsigned int Count() = 0;
7      virtual void Push(const std::string& strValue) = 0;
8      virtual std::string Pop(const unsigned int nIdx) = 0;
9      virtual IAbstractIterator* CreateIterator() = 0;
10 };
```

代码 17.3: ConcreteIterator.h

```
1  #include "AbstractAggregate.h"
2  #include "AbstractIterator.h"
3
4  class CConcreteIterator : public IAbstractIterator
5  {
6  public:
7      CConcreteIterator(IAbstractAggregate *pAggregate)
8      {
9          m_unCurrentIdx = 0;
10         m_pAggregate = pAggregate;
11     }
12     ~CConcreteIterator();
13
14     std::string First()
15     {
16         return m_pAggregate->Pop(m_unCurrentIdx);
17     }
18     std::string Current()
19     {
20         return m_pAggregate->Pop(m_unCurrentIdx);
21     }
22     std::string Next()
23     {
24         if (m_unCurrentIdx++ < m_pAggregate->Count())
25         {
26             return m_pAggregate->Pop(m_unCurrentIdx);
27         }
28
29         return "";
30     }
31     bool IsEnd()
32     {
33         return m_unCurrentIdx >= m_pAggregate->Count() ? true : false;
34     }
35
36 private:
37     unsigned int m_unCurrentIdx;
38
39     IAbstractAggregate *m_pAggregate;
40 };
```

代码 17.4: ConcreteAggregate.h

```
#include <vector>

#include "AbstractAggregate.h"
#include "ConcreteIterator.h"

class CConcreteAggregate : public IAbstractAggregate
{
public:
    CConcreteAggregate()
    {
        m_vecItems.clear();
        m_pIterator = nullptr;
    }
    ~CConcreteAggregate()
    {
        if (nullptr != m_pIterator)
        {
            delete m_pIterator;
            m_pIterator = nullptr;
        }

        m_vecItems.clear();
    }

    IAbstractIterator * CreateIterator()
    {
        if (nullptr == m_pIterator)
        {
            m_pIterator = new CConcreteIterator(this);
        }
        return m_pIterator;
    }

    unsigned int Count()
    {
        return m_vecItems.size();
    }
    void Push(const std::string & strValue)
    {
        m_vecItems.push_back(strValue);
    }
    std::string Pop(const unsigned int nIdx)
    {
        if (nIdx < m_vecItems.size())
        {
            return m_vecItems[nIdx];
        }
        return "";
    }
```

```
50
51 private :
52     std :: vector<std:: string > m_vecItems;
53      IAbstractIterator   * m_pIterator ;
54 };
```

代码 17.5: MainCaller.cpp

```
1  #include <stdio.h>
2
3  #include "ConcreteAggregate.h"
4  #include "ConcreteIterator.h"
5
6  using namespace std;
7
8  int main(int argc, char**argv)
9  {
10     IAbstractAggregate *pAggreator = new CConcreteAggregate();
11     pAggreator->Push("hello");
12     pAggreator->Push("world");
13     pAggreator->Push("cxue");
14
15      IAbstractIterator   * pIterator  = pAggreator->CreateIterator() ;
16     if ( nullptr  !=  pIterator )
17     {
18         printf ("%s is ok\n", pIterator->Next().c_str());
19         printf ("%s is ok\n", pIterator->Current().c_str());
20         printf ("%s is ok\n", pIterator->First().c_str()) ;
21     }
22
23     printf ("press any key to exit...\n");
24     getchar () ;
25 }
```

第18章　中介者模式

中介者模式（Mediator）用一个中介者对象来封装一系列的对象交互。中介者使各对象不需要显式的相互引用，从而使其耦合松散，而且可以独立的改变他们之间的交互。

18.1　UML图

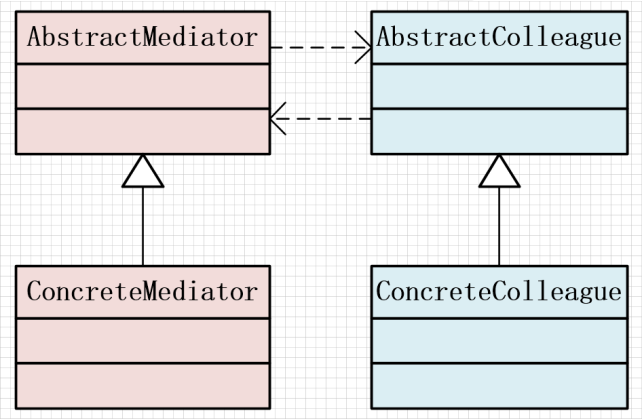

图 18–1

图 18–1 中，类和对象的关系为：

1. 抽象中介者（Abstract Mediator）：中介者定义一个接口用于与各同事（Colleague）对象通信。
2. 具体中介者（Concrete Mediator）：通过协调各同事对象实现协作行为，并了解和维护它的各个同事。
3. 同事类（Colleague）：每一个同事类都知道它的中介者对象，每一个同事对象在需与其它的同事通信的时候，与它的中介者通信。

18.2 优点

1. 将原本分布于多个对象间的行为集中在一起，改变这些行为只需生成 Mediator 的子类，这样各个 Colleague 类可以被重用，减少子类生成。
2. 将各 Colleague 解耦，可以独立的改变和利用各 Colleague 类和 Mediator 类。
3. 简化了对象协议，用 Mediator 和各 Colleague 间的一对多的交互来代替多对多的交互。
4. 对对象如何协作进行了抽象，将中介作为一个独立的概念并将其封装在一个对象中，使注意力从对象各自本身的行为转移到它们之间的交互上来，有助于弄清楚一个系统中的对象是如何交互的。

18.3 缺点

1. 使控制集中化，将交互的复杂性变为中介者的复杂性，封装了协议的中介者，可能变得比任一个 Colleague 都复杂。

18.4 应用场景

1. 一组对象以定义良好但复杂的方式进行通信时，产生的相互依赖关系结构混乱且难以理解。
2. 一个对象引用其它很多对象并且直接与这些对象通信，导致难以利用该对象。
3. 想定制一个分布在多个类中的行为，而又不想生成大多的子类。

18.5 实例

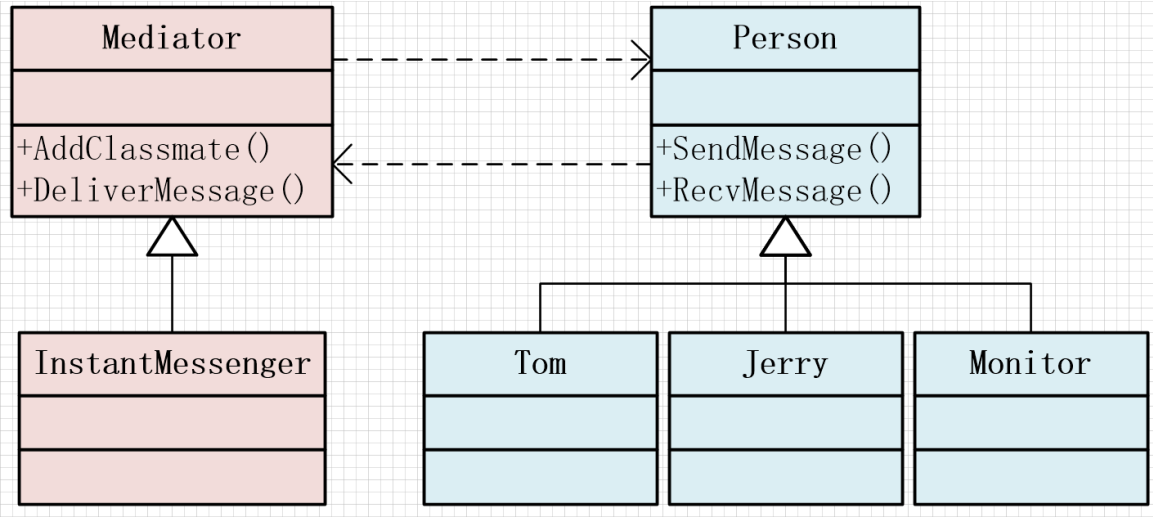

图 18–2

代码 18.1: Person.h

```
1  #include <string>
2
3  #include "Mediator.h"
4
5  class IMediator;
6
7  class IPerson
8  {
9  public:
10     virtual void SendMessage(std:: string strName, std :: string strMsg) = 0;
11     virtual void RecvMessage(std:: string strName, std :: string strMsg) = 0;
12
13 protected:
14     IMediator *m_pMediator;
15 };
```

代码 18.2: Mediator.h

```
1  #include <map>
2  #include <string>
3
4  #include "Person.h"
5
6  class IPerson;
7
8  class IMediator
9  {
```

```
10 public:
11     virtual void AddClassmate(std::string strName, IPerson *pClassmate) = 0;
12     virtual void DeliverMessage(std::string strFromName, std::string strToName, std
           ::string strMsg) = 0;
13
14 protected:
15     std::map<std::string, IPerson*> m_mapRoster;
16 };
```

代码 18.3: Jerry.h

```
1  #include "Person.h"
2
3  class CJerry : public IPerson
4  {
5  public:
6      CJerry(IMediator *pMediator)
7      {
8          m_pMediator = pMediator;
9      }
10     ~CJerry()
11     {
12         m_pMediator = nullptr;
13     }
14
15     void SendMessage(std::string strName, std::string strMsg)
16     {
17         printf("Jerry send message \"%s\" to %s\n", strMsg.c_str(), strName
               .c_str());
18         m_pMediator->DeliverMessage("Jerry", strName, strMsg);
19     }
20
21     void RecvMessage(std::string strName, std::string strMsg)
22     {
23         printf("Jerry gets message \"%s\" from %s\n", strMsg.c_str(),
               strName.c_str());
24     }
25 };
```

代码 18.4: Monitor.h

```
1  #include "Person.h"
2
3  class CMonitor: public IPerson
4  {
5  public:
6      CMonitor(IMediator *pMediator)
7      {
8          m_pMediator = pMediator;
9      }
```

```
10      ~CMonitor()
11      {
12          m_pMediator = nullptr;
13      }
14
15      void SendMessage(std::string strName, std::string strMsg)
16      {
17          printf("Monitor send message \"%s\" to %s\n", strMsg.c_str(),
             strName.c_str());
18          m_pMediator->DeliverMessage("Monitor", strName, strMsg);
19      }
20
21      void RecvMessage(std::string strName, std::string strMsg)
22      {
23          printf("Monitor gets message \"%s\" from %s\n", strMsg.c_str(),
             strName.c_str());
24      }
25  };
```

代码 18.5: Tom.h

```
1   #include "Person.h"
2
3   class CTom : public IPerson
4   {
5   public:
6       CTom(IMediator *pMediator)
7       {
8           m_pMediator = pMediator;
9       }
10      ~CTom()
11      {
12          m_pMediator = nullptr;
13      }
14
15      void SendMessage(std::string strName, std::string strMsg)
16      {
17          printf("Tom send message \"%s\" to %s\n", strMsg.c_str(), strName.
             c_str());
18          m_pMediator->DeliverMessage("Tom", strName, strMsg);
19      }
20
21      void RecvMessage(std::string strName, std::string strMsg)
22      {
23          printf("Tom gets message \"%s\" from %s\n", strMsg.c_str(), strName
             .c_str());
24      }
25  };
```

代码 18.6: InstantMessager.h

```
#include "Mediator.h"
#include "Monitor.h"
#include "Jerry.h"
#include "Tom.h"

class CInstantMessager : public IMediator
{
public:
    CInstantMessager() {}
    ~CInstantMessager() {}

    void AddClassmate(std::string strName, IPerson *pClassmate)
    {
        if (m_mapRoster.find(strName) == m_mapRoster.end())
        {
            m_mapRoster.insert(std::make_pair(strName, pClassmate));
        }
    }

    void DeliverMessage(std::string strFromName, std::string strToName, std::string
        strMsg)
    {
        if (strToName == "All")
        {
            for (auto mapIter = m_mapRoster.begin(); mapIter != m_mapRoster.end();
            mapIter++)
            {
                ((*mapIter).second)->RecvMessage(strFromName, strMsg);
            }
        }
        else
        {
            m_mapRoster[strToName]->RecvMessage(strFromName, strMsg);
        }
    }
};
```

代码 18.7: MainCaller.cpp

```
#include "InstantMessager.h"
#include "Monitor.h"
#include "Jerry.h"
#include "Tom.h"

int main(int argc, char** argv)
{
    CInstantMessager *pIM = new CInstantMessager();
    CJerry *pJerry = new CJerry(pIM);
```

```
10      CTom *pTom = new CTom(pIM);
11      CMonitor *pMonitor = new CMonitor(pIM);
12
13      pIM->AddClassmate("Jerry", pJerry);
14      pIM->AddClassmate("Tom", pTom);
15      pIM->AddClassmate("Monitor", pMonitor);
16
17      pTom->SendMessage("Jerry", "I want to eat you");
18      pJerry->SendMessage("Tom", "stop joking");
19      pMonitor->SendMessage("All", "be quiet");
20      pTom->SendMessage("Monitor", "OK");
21      pJerry->SendMessage("Monitor", "OK");
22
23      delete pMonitor;
24      pMonitor = nullptr;
25      delete pTom;
26      pTom = nullptr;
27      delete pJerry;
28      pJerry = nullptr;
29      delete pIM;
30      pIM = nullptr;
31
32      printf("press any key to exit...\n");
33      getchar();
34  }
```

第19章　备忘录模式

备忘录模式（Memento）在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象外部保存这个状态。

19.1　UML图

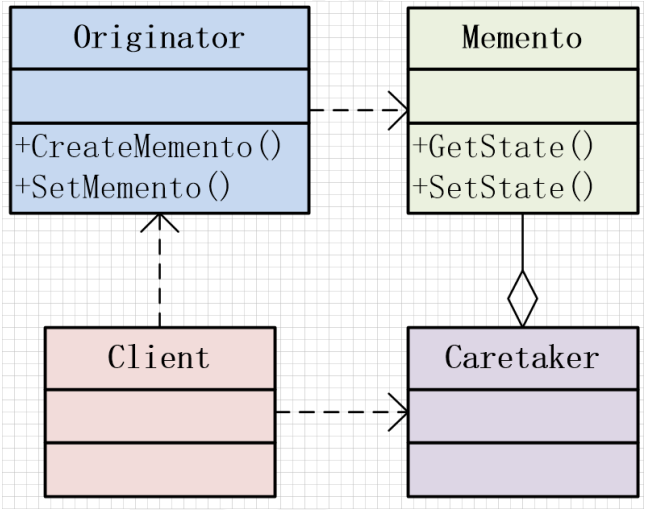

图 19–1

图 19–1 中，类和对象的关系为：

1. 原发器（Originator）：要被备份的成员，它提供一创建备忘录的方法，其实就是将它自身的某些信息拷贝一份到一个备忘录对象中，并提供另外一个方法将备忘录中的信息覆盖自身的信息。
2. 备忘录（Memento）：备忘录对象中包含存储发起人状态的成员变量，它提供 set 和 get 方法保存及获取发起人状态。
3. 管理角色（Caretaker）：用于管理备忘录对象的实现类。

19.2 优点

1. 保持封装边界：避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。
2. 简化了原发器。

19.3 缺点

1. 可能代价很高：生成备忘录时或者频繁创建备忘录和恢复原发器状态，可能会导致非常大的开销。
2. 维护备忘录的潜在代价。

19.4 应用场景

1. 必须保存一个对象在某一时刻的状态，这样可以在需要的时候恢复。
2. 保护对象的实现细节以及封装性不被暴露。

19.5 实例

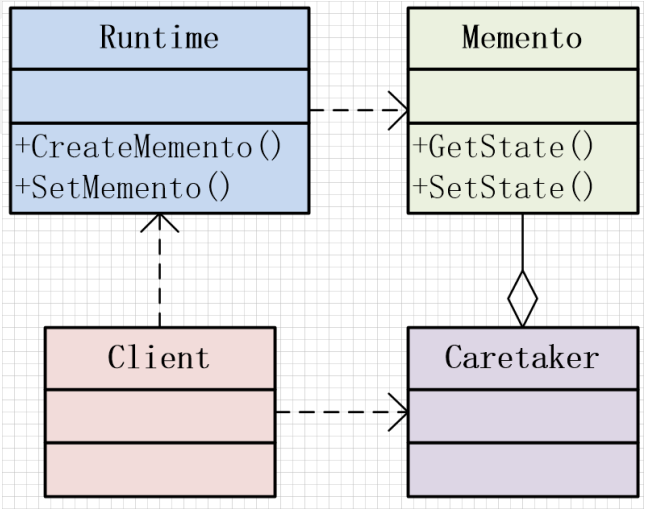

图 19–2

代码 19.1: Memento.h

```
#include <stdio.h>
#include <string>

class CMemento
{
public:
    CMemento(std::string strState)
    {
        m_strState = strState;
    }
    ~CMemento() {}

    std::string GetState()
    {
        return m_strState;
    }
    void SetState(std::string strState)
    {
        m_strState = strState;
    }

private:
    std::string m_strState;
};
```

代码 19.2: Runtime.h

```
#include "Memento.h"

class CRuntime
{
public:
    CRuntime() {}
    ~CRuntime() {}

    CMemento* CreateMemento()
    {
        return new CMemento(m_strState);
    }
    void RestoreMemento(CMemento *pM)
    {
        m_strState = pM->GetState();
    }
    std::string GetState()
    {
        return m_strState;
    }
    void SetState(std::string strState)
```

```
22      {
23          m_strState = strState ;
24          printf ("the system is run at %s.\n", m_strState.c_str());
25      }
26
27  private :
28      std :: string   m_strState ;
29  };
```

代码 19.3: UserMemento.h

```
1   #include "Memento.h"
2
3   class CUserMemento
4   {
5   public :
6       CUserMemento()
7       {
8           m_pM = nullptr;
9       }
10      ~CUserMemento()
11      {
12          m_pM = nullptr;
13      }
14
15      CMemento* RetrieveMemento()
16      {
17          return m_pM;
18      }
19      void SaveMemento(CMemento *pM)
20      {
21          m_pM = pM;
22      }
23
24  private :
25      CMemento *m_pM;
26  };
```

代码 19.4: MainCaller.cpp

```
1   #include "Runtime.h"
2   #include "UserTaker.h"
3
4   int main(int argc, char** argv)
5   {
6       CRuntime *winSys = new CRuntime();
7       CUserMemento* pCarer = new CUserMemento();
8
9       winSys->SetState("good");
10      pCarer->SaveMemento(winSys->CreateMemento());
```

```
11
12      winSys->SetState("bad");
13      winSys->RestoreMemento(pCarer->RetrieveMemento());
14
15      printf("state of winSys is %s.\n", winSys->GetState().c_str());
16
17      delete pCarer;
18      pCarer = nullptr;
19      delete winSys;
20      winSys = nullptr;
21
22      printf("press any key to exit...\n");
23      getchar();
24  }
```

第20章　状态模式

状态模式（State）允许一个对象在它的内部状态改变时改变它的行为。

20.1　UML图

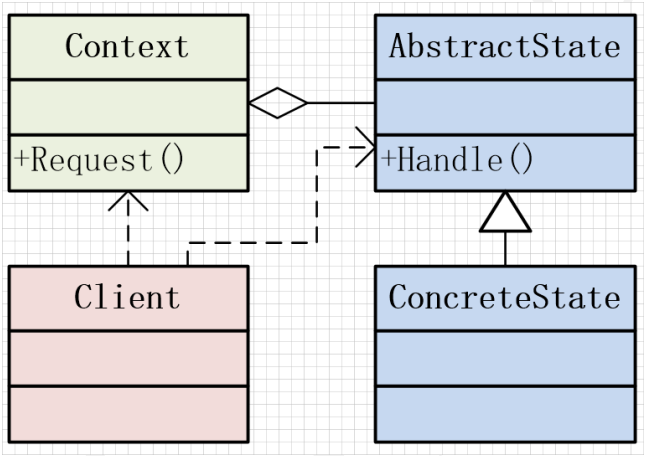

图 20–1

图 20–1 中，类和对象的关系为：

1. 上下文（Context）：通常用来定义客户感兴趣的接口，同时维护一个来具体处理当前状态的实例对象。
2. 状态接口（State）：用来封装与上下文的一个特定状态所对应的行为。
3. 具体状态（ConcreteState）：具体实现状态处理的类，每个类实现一个跟上下文相关的状态的具体处理。

20.2　优点

1. 封装了转换规则。

2. 枚举可能的状态，在枚举状态之前需要确定状态种类。
3. 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
4. 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
5. 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

20.3 缺点

1. 状态模式的使用必然会增加系统类和对象的个数。
2. 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
3. 状态模式对”开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

20.4 应用场景

1. 行为随状态改变而改变的场景。
2. 条件、分支语句的代替者。

20.5 实例

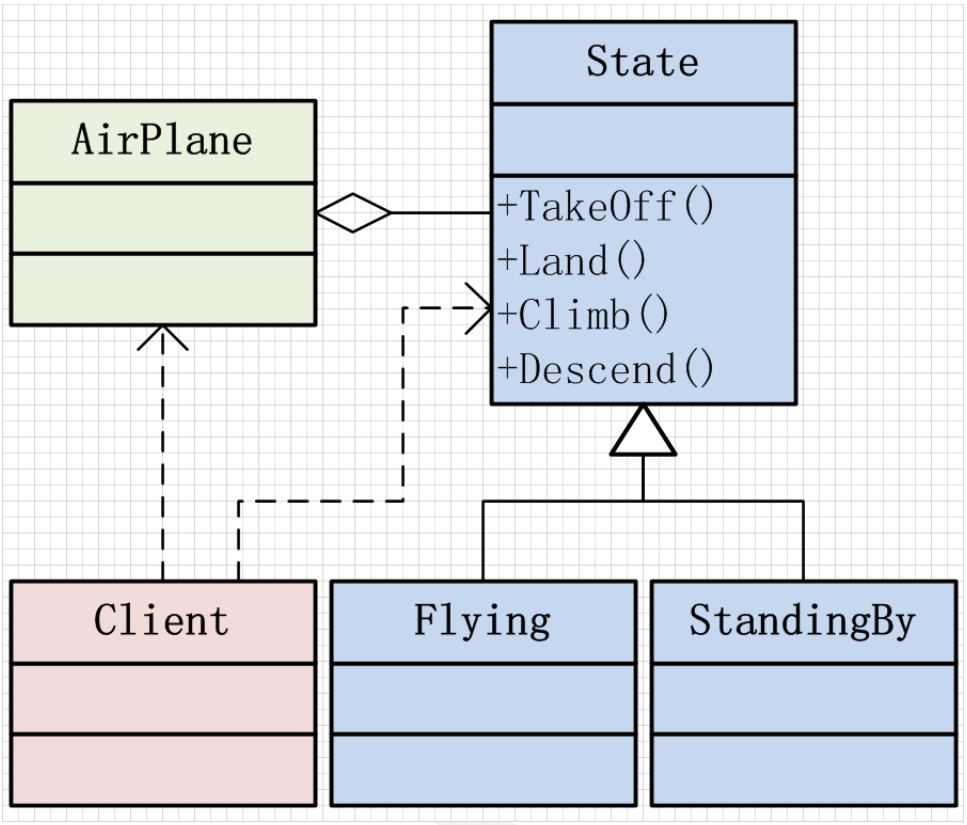

图 20–2

代码 20.1: State.h

```
1  #include <stdio.h>
2
3  class IState
4  {
5  public:
6      virtual void Board() = 0;
7      virtual void Deplane() = 0;
8      virtual void TakeOff() = 0;
9      virtual void Land() = 0;
10     virtual void Climb() = 0;
11     virtual void Descend() = 0;
12 };
```

代码 20.2: StandingBy.h

```
1  #include "State.h"
2
3  class CStandingBy : public IState
4  {
5  public:
6      CStandingBy() {}
```

```
7       ~CStandingBy() {}
8
9       void Board()
10      {
11          printf ("passengers getting board\n");
12      }
13      void Deplane()
14      {
15          printf ("passengers getting off the plane\n");
16      }
17      void TakeOff()
18      {
19          printf ("prepare to take off\n");
20      }
21      void Land()
22      {
23          printf ("the plane has already been on the land\n");
24      }
25      void Climb()
26      {
27          printf ("unable to climb, the plane is on the land\n");
28      }
29      void Descend()
30      {
31          printf ("unable to descend, the plane is on the land\n");
32      }
33  };
```

代码 20.3: Flying.h

```
1   #include "State.h"
2
3   class CFlying : public IState
4   {
5   public:
6       CFlying() {}
7       ~CFlying() {}
8
9       void Board()
10      {
11          printf ("unable to get on board\n");
12      }
13      void Deplane()
14      {
15          printf ("unable to get off the plane\n");
16      }
17      void TakeOff()
18      {
19          printf ("the plane is in the air\n");
20      }
```

```
21      void Land()
22      {
23          printf("landing\n");
24      }
25      void Climb()
26      {
27          printf("climbing\n");
28      }
29      void Descend()
30      {
31          printf("descend\n");
32      }
33  };
```

代码 20.4: AirPlane.h

```
1   #include "State.h"
2
3   class CAirPlane
4   {
5   public:
6       CAirPlane()
7       {
8           m_pState = nullptr;
9       }
10      ~CAirPlane()
11      {
12          m_pState = nullptr;
13      }
14
15      void SetState(IState *pState)
16      {
17          m_pState = pState;
18      }
19
20      void Board()
21      {
22          if (nullptr != m_pState)
23          {
24              m_pState->Board();
25          }
26      }
27      void Deplane()
28      {
29          if (nullptr != m_pState)
30          {
31              m_pState->Deplane();
32          }
33      }
34      void TakeOff()
```

```
35      {
36          if ( nullptr != m_pState)
37          {
38              m_pState->TakeOff();
39          }
40      }
41      void Land()
42      {
43          if ( nullptr != m_pState)
44          {
45              m_pState->Land();
46          }
47      }
48      void Climb()
49      {
50          if ( nullptr != m_pState)
51          {
52              m_pState->Climb();
53          }
54      }
55      void Descend()
56      {
57          if ( nullptr != m_pState)
58          {
59              m_pState->Descend();
60          }
61      }
62
63  private :
64      IState *m_pState;
65  };
```

代码 20.5: MainCaller.cpp

```
1   #include "AirPlane.h"
2   #include "Flying.h"
3   #include "StandingBy.h"
4
5   using namespace std;
6
7   int main(int argc, char** argv)
8   {
9       CAirPlane *pAirPlane = new CAirPlane();
10      IState *pState = new CStandingBy();
11      pAirPlane->SetState(pState);
12      pAirPlane->Board();
13      pAirPlane->Climb();
14      pAirPlane->TakeOff();
15
16      delete pState;
```

```
17      pState = new CFlying();
18
19      pAirPlane->SetState(pState);
20      pAirPlane->Climb();
21      pAirPlane->Deplane();
22      pAirPlane->Descend();
23      pAirPlane->Land();
24
25      delete pState;
26      pState = new CStandingBy();
27
28      pAirPlane->SetState(pState);
29      pAirPlane->Deplane();
30
31      delete pAirPlane;
32      pAirPlane = nullptr;
33      delete pState;
34      pState = nullptr;
35
36      printf("press any key to exit...\n");
37      getchar();
38  }
```

20.6 和其它模式的比较

20.6.1 Abstract Fatory

20.6.2 Strategy

第21章　策略模式

策略模式（Strategy）：定义一系列算法类，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化

21.1　UML图

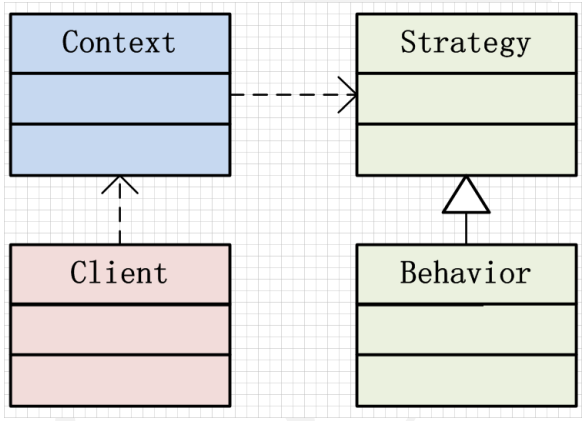

图 21–1

图 21–1 中，类和对象的关系为：

1. 上下文（Context）：通常用来定义客户感兴趣的接口，同时维护一个来具体处理当前状态的实例对象。
2. 策略接口（Strategy）：用来封装与上下文的一个特定状态所对应的行为。
3. 具体状态（ConcreteState）：具体实现状态处理的类，每个类实现一个跟上下文相关的状态的具体处理。

21.2　优点

1. 提供了对开闭原则的完美支持，用户可以在不修改原有系统的基础上

选择具体算法或行为，也可以灵活地增加新的算法或行为。

2. 避免了多重的 if-else 条件选择语句，利于系统的维护。
3. 提供了一种算法的复用机制，不同的环境类可以方便地复用这些策略类。

21.3 缺点

1. 客户端需要知道所有的策略类，并自行决定使用哪一个策略。
2. 将造成系统产生很多的具体策略类，任何细小的变化都将导致系统要增加一个具体策略类。
3. 无法在客户端同时使用多个策略类。

21.4 应用场景

1. 如果一个系统要动态地在几种算法之间选择其中一种。
2. 如果有难以维护的多重 if-else 条件选择语句是为了实现对象的行为。
3. 不希望客户知道复杂的与算法有关的数据结构，可以将其封装到策略中。

21.5 实例

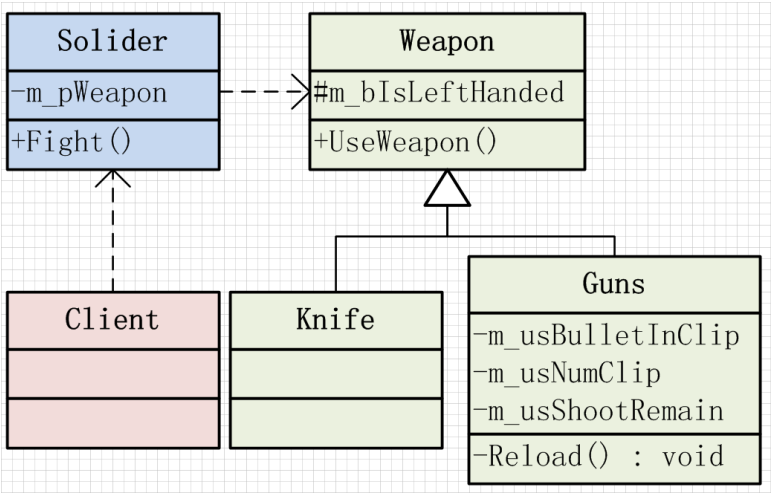

图 21–2

代码 21.1: Weapon.h

```
#include <stdio.h>

enum WeaponType
{
    Gun,
    Knife
};

class IWeapon
{
public:
    IWeapon(bool bIsLeftHanded)
    {
        m_bIsLeftHanded = bIsLeftHanded;
    }
    virtual ~IWeapon() {}

    void virtual UseWeapon(const unsigned short usTryTime) = 0;

protected:
    bool m_bIsLeftHanded;
};
```

代码 21.2: Gun.h

```
#include "Weapon.h"

class CGun : public IWeapon
{
public:
```

```
6     CGun(bool bIsLeftHanded, unsigned short usInitClips ) : IWeapon(bIsLeftHanded)
7     {
8         m_usNumClip = usInitClips;
9
10        m_unBulletInClip = 0;
11
12        m_bIsLeftHanded = false;
13
14        Reload();
15    }
16    ~CGun() {}
17
18    // override the virtual function in base class.
19    void UseWeapon(unsigned short usBulletsShot )
20    {
21        m_usShootRemain = usBulletsShot;
22
23        while (m_usShootRemain > 0)
24        {
25            if (m_unBulletInClip > m_usShootRemain)
26            {
27                if (m_bIsLeftHanded)
28                {
29                    printf ("shoot %u times with left hand.\n",
      m_usShootRemain);
30                }
31                else
32                {
33                    printf ("shoot %u times with right hand.\n",
      m_usShootRemain);
34                }
35
36                m_unBulletInClip -= m_usShootRemain;
37                m_usShootRemain = 0;
38            }
39            else
40            {
41                if (m_bIsLeftHanded)
42                {
43                    printf ("shoot %u times with left hand.\n",
      m_unBulletInClip);
44                }
45                else
46                {
47                    printf ("shoot %u times with right hand.\n",
      m_unBulletInClip);
48                }
49
50                m_usShootRemain -= m_unBulletInClip;
51
52                if (!Reload())
```

```
53                  {
54                      return ;
55                  }
56              }
57          }
58      }
59
60  private :
61      bool Reload()
62      {
63          if (m_usNumClip > 0)
64          {
65              printf ("bullets reload.\n");
66              m_unBulletInClip = 20;
67              m_usNumClip--;
68
69              return true ;
70          }
71          else
72          {
73              printf ("out of ammunition\n");
74              return false ;
75          }
76      }
77
78      unsigned short m_unBulletInClip;
79      unsigned short m_usNumClip;
80      unsigned short m_usShootRemain;
81  };
```

第22章　模板模式

模板模式（Template）定义一个操作中的算法骨架，将一些步骤延迟到子类中。使子类可以不改变算法的结构而重新定义算法的某些特定步骤。

22.1　UML图

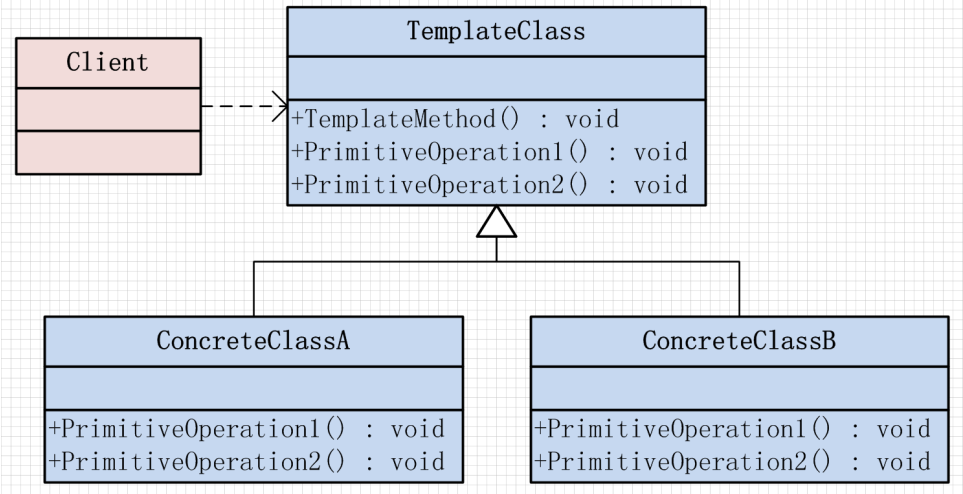

图 22–1

图 22–1 中，类和对象的关系为：

1. 模板方法（Template Class）：一般是一个具体方法的框架，实现对基本方法的调度，完成固定的逻辑。
2. 具体类（Concrete Class）：由子类实现的方法，被模板方法调用。

22.2　优点

1. 逻辑框架是封装的不变部分，具体细节是供扩展可变部分。
2. 提取公共部分代码，便于维护。

3. 行为由父类控制，子类实现基本方法，因此子类可以通过扩展的方式增加相应的功能，符合开闭原则。

22.3 缺点

1. 模板方法将依赖颠倒，抽象中的框架实现依赖具体类的细节实现，所以是子类行为影响了父类。在复杂项目中，会造成代码阅读困难。

22.4 应用场景

1. 多个子类有共有的方法，并且逻辑相同，细节有差异.
2. 对复杂的算法，将核心算法设计为模板方法，周边细节由子类实现。
3. 重构时，将相同的代码抽象到父类，通过钩子函数约束行为。

22.5 实例

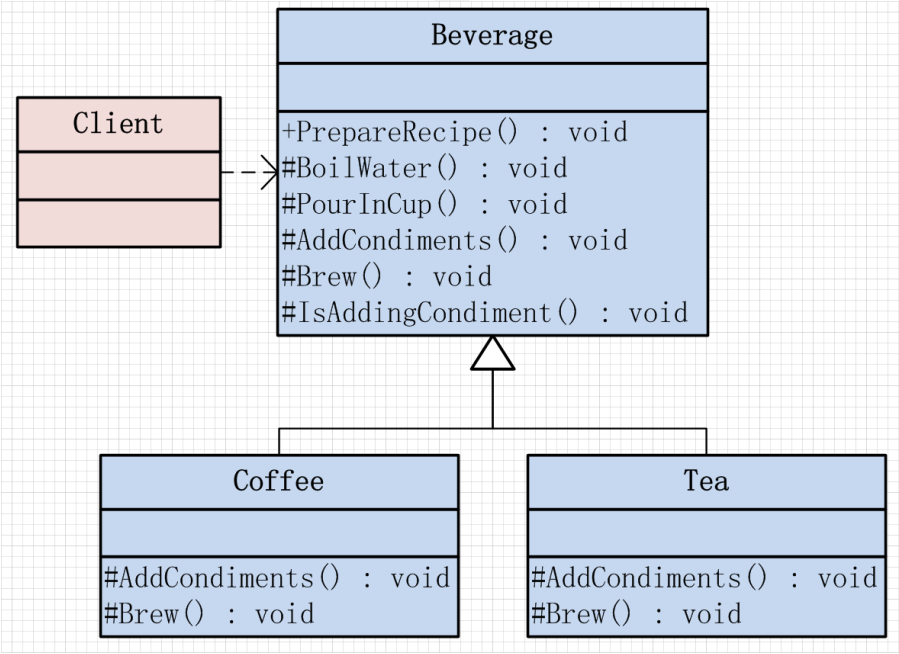

图 22–2

代码 22.1: Beverage.h

```
#include <stdio.h>

class CBeverage
{
public:
    CBeverage() { m_bIsAddingCondiment = false; }
    ~CBeverage() {}
    void PrepareRecipe()
    {
        BoilWater();
        Brew();
        PourInCup();
        AddCondiment();
    }
protected:
    void BoilWater()
    {
        printf("boil the water\n");
    }
    void PourInCup()
    {
        printf("pour water into cup\n");
    }
    virtual void AddCondiment() = 0;
    virtual void Brew() = 0;

    bool m_bIsAddingCondiment;
};
```

代码 22.2: Coffee.h

```
#include "Beverage.h"

class CCoffee : public CBeverage
{
public:
    CCoffee(bool bIsAddingCondiment = true) { m_bIsAddingCondiment =
        bIsAddingCondiment; }
    ~CCoffee() {}
private:
    void AddCondiment()
    {
        printf("add milk and sugar\n");
    }
    void Brew()
    {
        printf("brew coffee pounder with water\n");
    }
```

```
};
```

代码 22.3: Tee.h

```
#include "Beverage.h"

class CTee : public CBeverage
{
public:
    CTee(bool bIsAddingCondiment = true) { m_bIsAddingCondiment =
        bIsAddingCondiment; }
    ~CTee() {}

private:
    void AddCondiment()
    {
        printf ("add lemon\n");
    }
    void Brew()
    {
        printf ("brew tea leaves with water\n");
    }
};
```

代码 22.4: MainCaller.cpp

```
#include "Coffee.h"
#include "Tee.h"

int main(int argc, char** argv)
{
    CCoffee *pCoffee = new CCoffee();
    pCoffee->PrepareRecipe();
    delete pCoffee;
    pCoffee = nullptr;

    printf ("\n");

    CTee *pTee = new CTee();
    pTee->PrepareRecipe();
    delete pTee;
    pTee = nullptr;

    printf ("press any key to exit...\n");
    getchar();
}
```

第23章　访问者模式

访问者模式（Visitor）表示一个作用于某对象结构中各个元素的操作。可以使用户不修改各元素类的前提下定义作用于这些元素的新操作，也就是动态的增加新的方法。

23.1　UML图

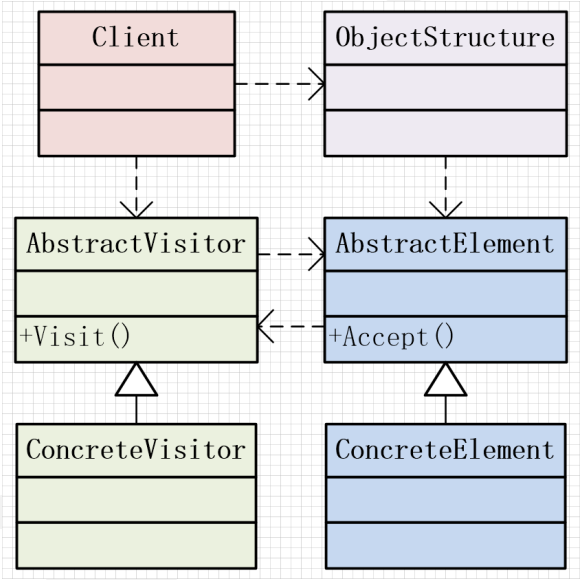

图 23–1

图 23–1 中，类和对象的关系为：

1. 抽象访问者（Abstract Visitor）：抽象出访问元素的动作。
2. 具体访问者（Concrete Visitor）：实现访问元素的动作。
3. 抽象元素（Abstract Element）：定义一个接受访问的操作，其参数为访问者。
4. 具体元素(ConcreteElement)：实现接受访问操作。
5. 对象结构类(ObjectStructure)：枚举并且管理元素。

23.2 优点

1. 访问者模式使得增加新的操作变得很容易，增加新的操作就意味着增加一个新的访问者类。
2. 访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。
3. 访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。
4. 每一个单独的访问者对象都集中了相关的行为，从而也就可以在访问的过程中将执行操作的状态积累在自己内部，而不是分散到很多的节点对象中。

23.3 缺点

1. 增加新的节点类变得很困难。每增加一个新的节点都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中增加相应的具体操作。
2. 破坏封装。访问者模式要求访问者对象访问并调用每一个节点对象的操作，要求所有节点的对象必须暴露一些自己的操作和内部状态。

23.4 应用场景

1. 一个对象结构包含很多类操作，它们有不同的接口，对这些对象实施一些依赖于其具体类的操作。
2. 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作污染这些对象的类。
3. 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。

23.5 实例

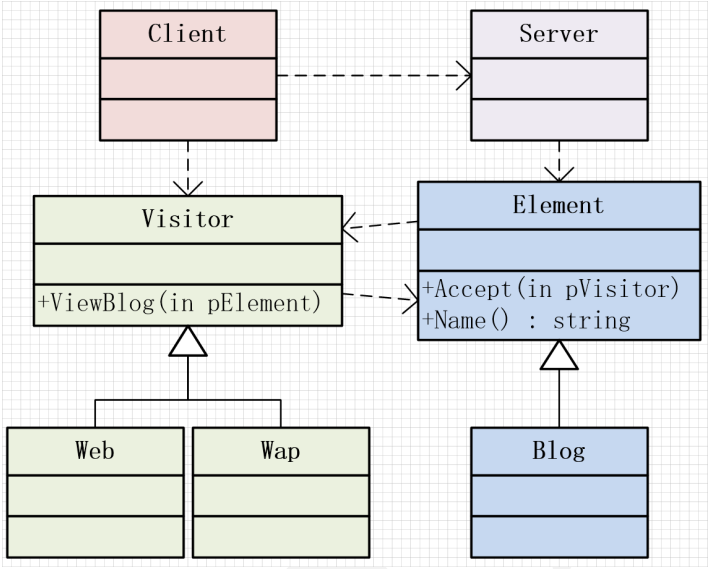

图 23–2

代码 23.1: Element.h

```
1  #include <stdio.h>
2  #include <string>
3
4  #include "Visitor.h"
5
6  class IVisitor ;
7
8  class IElement
9  {
10 public:
11     virtual void Accept( IVisitor *pVisitor ) = 0;
12     virtual std::string Name() = 0;
13
14 protected:
15     std::string m_strName;
16 };
```

代码 23.2: Cellphone.h

```
1  #include "Visitor.h"
2
3  class CCellphone : public IVisitor
4  {
5  public:
6      CCellphone() {}
```

```
7       ~CCellphone() {}
8
9       void ViewBlog(IElement *pElement)
10      {
11          printf ("view %s on cellphone\n", pElement->Name().c_str());
12      }
13  };
```

代码 23.3: Computer.h

```
1   #include "Visitor.h"
2
3   class CComputer : public IVisitor
4   {
5   public:
6       CComputer() {}
7       ~CComputer() {}
8
9       void ViewBlog(IElement *pElement)
10      {
11          printf ("view %s on computer\n", pElement->Name().c_str());
12      }
13  };
```

代码 23.4: Visitor.h

```
1   #include <stdio.h>
2
3   #include "Element.h"
4
5   class IElement;
6
7   class IVisitor
8   {
9   public:
10      virtual void ViewBlog(IElement *pElement) = 0;
11  };
```

代码 23.5: Blog.h

```
1   #include "Element.h"
2
3   class CBlog : public IElement
4   {
5   public:
6       CBlog(std::string strName)
7       {
8           m_strName = strName;
9       }
10      ~CBlog() {}
```

```
11
12      void Accept( IVisitor *pVisitor )
13      {
14          pVisitor->ViewBlog(this);
15      }
16
17      std::string Name()
18      {
19          return m_strName;
20      }
21  };
```

代码 23.6: Server.h

```
1   #include <map>
2
3   #include "Blog.h"
4   #include "Visitor.h"
5
6   class CServer
7   {
8   public:
9       CServer()
10      {
11          InitBlogs();
12      }
13      ~CServer()
14      {
15          RemoveBlogs();
16      }
17
18      void Accept( IVisitor *pVisitor )
19      {
20          for (auto iterElement : m_mapBlogs)
21          {
22              iterElement.second->Accept(pVisitor);
23          }
24      }
25
26  private:
27      void InitBlogs()
28      {
29          m_mapBlogs.insert(std::make_pair("art", new CBlog("art")));
30          m_mapBlogs.insert(std::make_pair("physics", new CBlog("physics")));
31          m_mapBlogs.insert(std::make_pair("mathematics", new CBlog("
            mathematics")));
32      }
33      void RemoveBlogs()
34      {
35          for (auto iterElement : m_mapBlogs)
```

```
36          {
37              delete iterElement.second;
38          }
39
40          m_mapBlogs.clear();
41      }
42
43      std::map<std::string, IElement*> m_mapBlogs;
44  };
```

代码 23.7: MainCaller.cpp

```
1   #include "Cellphone.h"
2   #include "Computer.h"
3   #include "Server.h"
4
5   int main(int argc, char** argv)
6   {
7       CComputer *pComputer = new CComputer();
8       CCellphone *pCellphone = new CCellphone();
9
10      CServer *pBlogServer = new CServer();
11
12      pBlogServer->Accept(pCellphone);
13      printf("\n");
14      pBlogServer->Accept(pComputer);
15
16      delete pBlogServer;
17      pBlogServer = nullptr;
18      delete pCellphone;
19      pCellphone = nullptr;
20      delete pComputer;
21      pComputer = nullptr;
22
23      printf("\npress any key to exit...\n");
24      getchar();
25  }
```

第四部分

附录

附录 A　UML类图关系的总结

在UML类图中，常见的有以下几种关系：

- 泛化（Generalization）
- 实现（Realization）
- 关联（Association）
- 聚合（Aggregation）
- 组合（Composition）
- 依赖（Dependency）

上述各种关系的*强弱顺序*为：

泛化=实现>组合>聚合>关联>依赖

A.1　泛化

*泛化关系*是一种继承关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种。

A.2　实现

*实现关系*是一种*类与接口*的关系，表示类是接口所有特征和行为的实现。

A.3　聚合

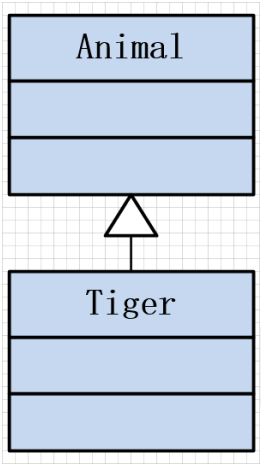

图 A-1: 泛化

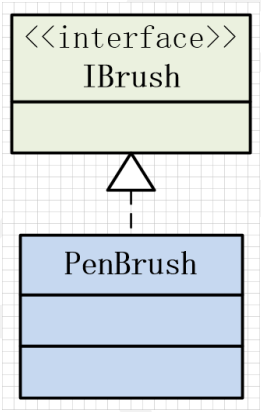

图 A-2: 实现

聚合关系是整体与部分的关系。如车和轮胎是可分割的整体和部分的关系。聚合关系是关联关系的一种，是强的关联关系。关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

A.4 组合

组合关系是不可分割的整体与部分的关系。没有公司就不存在部门。组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

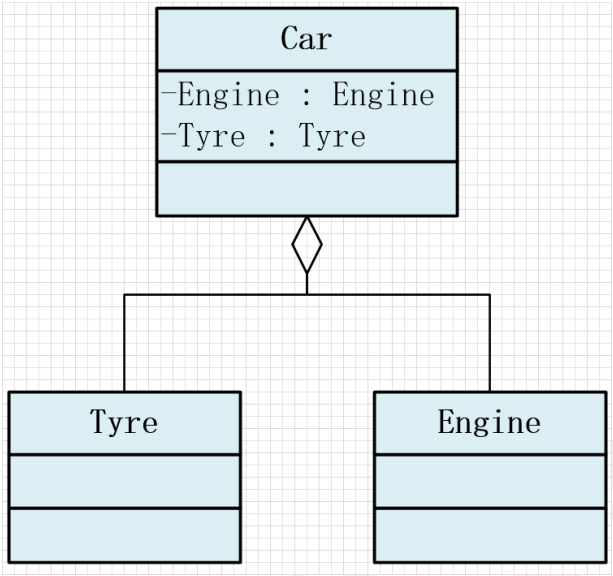

图 A-3: 聚合

A.5　关联

关联关系是一种拥有的关系，一般是长期性的，有内在联系的。例如：老师与学生，老师的职业需要有学生才有意义。关联可以是双向的，也可以是单向的。 双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

- **代码体现：**成员变量。

图 A-5 中，老师与学生是双向关联，老师有多名学生，学生也可能有多名老师。学生与某课程间的关系为单向关联，一名学生可能要上多门课程，课程是个抽象的东西，它不拥有学生。

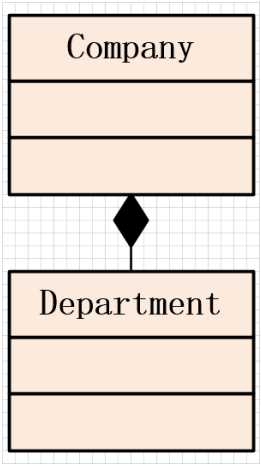

图 A-4: 组合

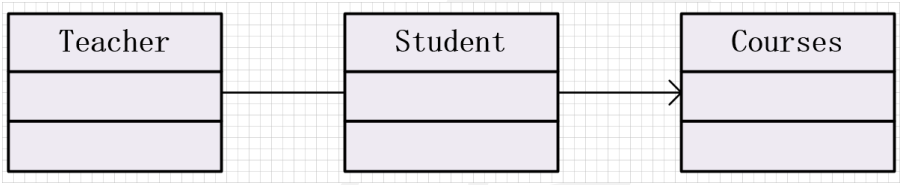

图 A-5: 关联

A.6 依赖

依赖关系是一种使用的关系，一个类 A 使用到了另一个类 B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是 B 类的变化会影响到 A。例如：人的本身属性并没有手机，而是通过 Buy(Product *) 的方法传入的。

- **代码体现**：局部变量、方法的参数或者对静态方法的调用。

A.7 一个例子

图 A-7 中，鸟有两个翅膀，但是翅膀不能脱离鸟单独存在，是不可分割的整体与个体关系，使用组合（Composition）关系。大雁可以属于某个雁群，也可以脱离雁群以个体单独存在，使用聚合（Aggregation）关系。动物需要呼吸和喝水来获取氧气和淡水，是引入的临时变量，使用依赖（Dependency）关系。环境是动物活动的一个影响因素，伴随动物的一生，需要作

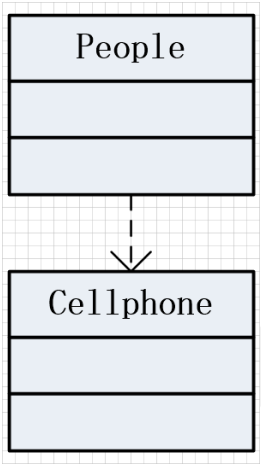

图 A-6: 依赖

为成员变量而不是临时变量存在，使用关联（Association）关系。

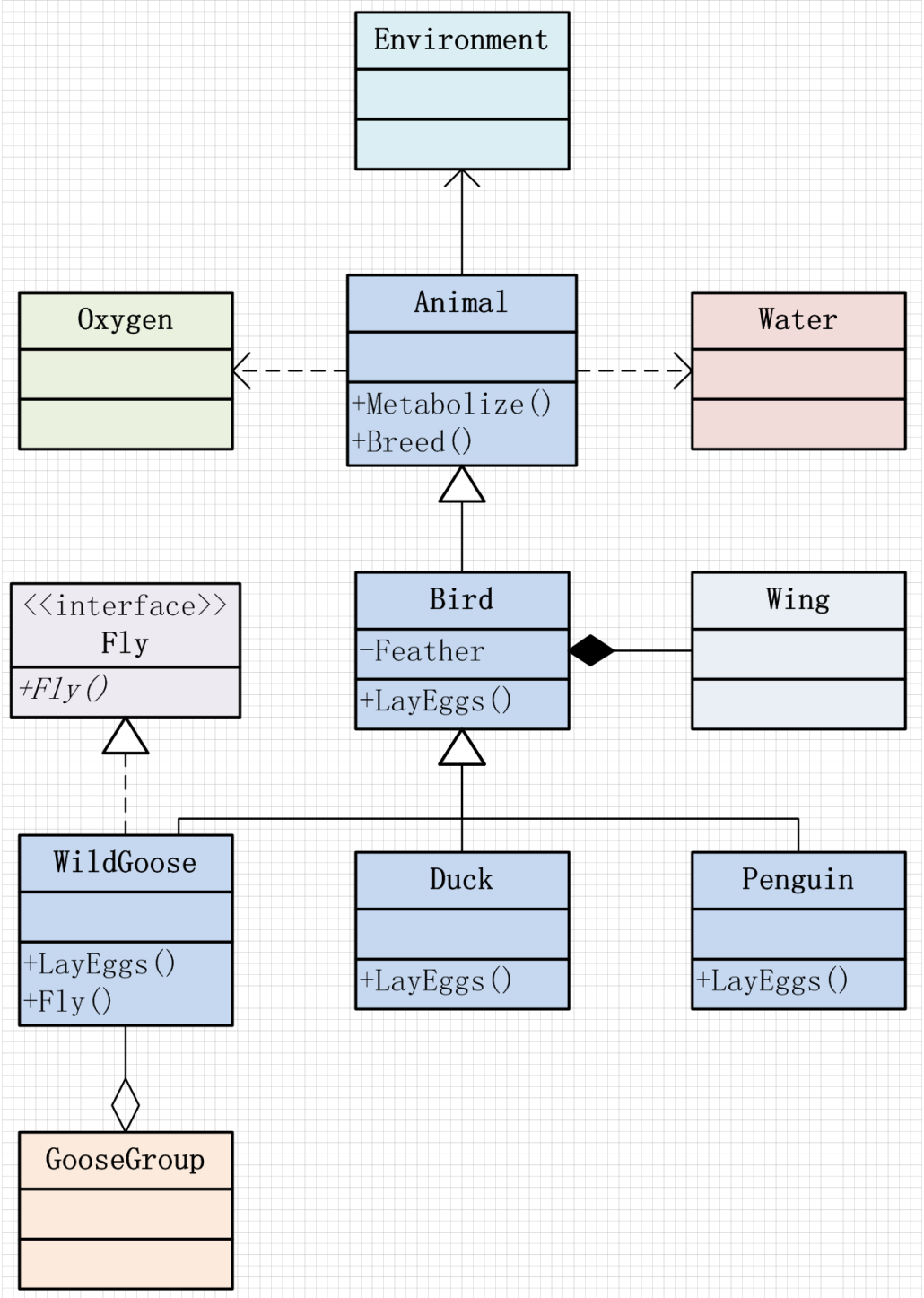
Environment
Oxygen
Animal
+Metabolize()
+Breed()
Water
<<interface>>
Fly
+Fly()
Bird
-Feather
+LayEggs()
Wing
WildGoose
+LayEggs()
+Fly()
Duck
+LayEggs()
Penguin
+LayEggs()
GooseGroup

图 A-7: 例子

附录 B　异同

图 B-1

1.

2.

3.