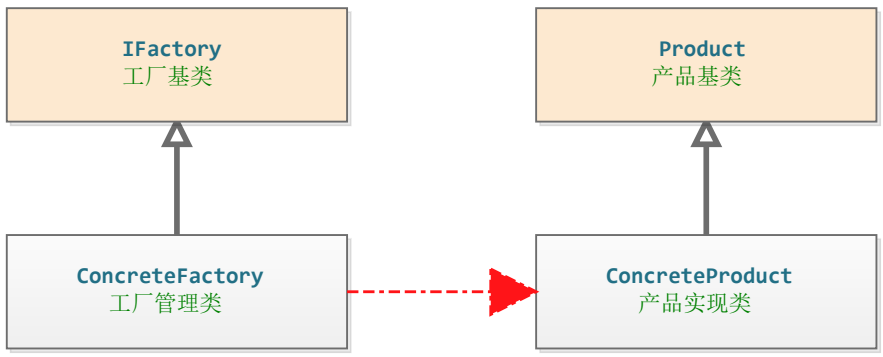
1.工厂模式-Factory

特点:将类的实例化推迟到子类中



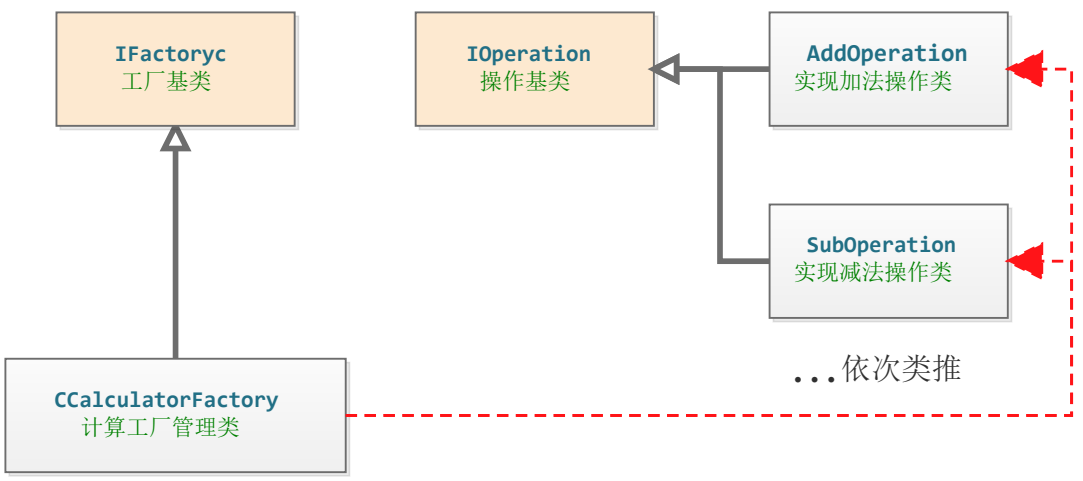

```
//工厂基类
class  IFactory
{
protected :
    IFactory (){};
public :
    virtual  ~IFactory (){};
    virtual  Product * CreateProduct ()= 0;
};
```

```
//工厂管理类
struct  ConcreteFactory  :public  IFactory
{
    ConcreteFactory ()
    {
        printf("创建工厂管理 \n");
    };
    ~ConcreteFactory (){};
    Product * CreateProduct ()
    {
        return  new  ConcreteProduct ();
    };
};
```

```
//产品基类需要继承才能创建
class  Product
{
protected :
    //封装产品类构造函数   可以在子类继承后构造
    Product (){};
public :
    virtual  ~Product (){};
};
```

```
//继承产品基类
struct  ConcreteProduct  :public  Product
{
    ConcreteProduct ()
    {
        printf("ConcreteProduct( 创建产品 )\n");
    };
    virtual  ~ConcreteProduct (){};
};
```

```
void  main(int  argc , char ** argv)
{
    //创建工厂类
    IFactory * fac = new ConcreteFactory ();
    //创建产品类
    Product * p = fac->CreateProduct ();
    delete  p;
    delete  fac;
}
```




```
class  IFactoryc
{
protected :
    IFactoryc (){};
public :
    virtual  ~IFactoryc (){};
    virtual  IOperation * CreateProduct (char  cOperator ) = 0;
};
```

```
class  IOperation
{
protected :
    int  m_nFirst ;
    int  m_nSecond ;
protected :
    //封装产品类构造函数   可以在子类继承后构造
    IOperation () : m_nFirst (0), m_nSecond (0){};
public :
    virtual  ~IOperation (){};
    void  SetKeyValue (int  a, int  b)
    {
        m_nFirst  = a;
        m_nSecond  = b;
    };
    virtual  double  GetResult ()=0;
};
```

```
struct  AddOperation  :public  IOperation
{
    virtual  double  GetResult ()
    {
        return  m_nFirst  + m_nSecond ;
    };
};
```

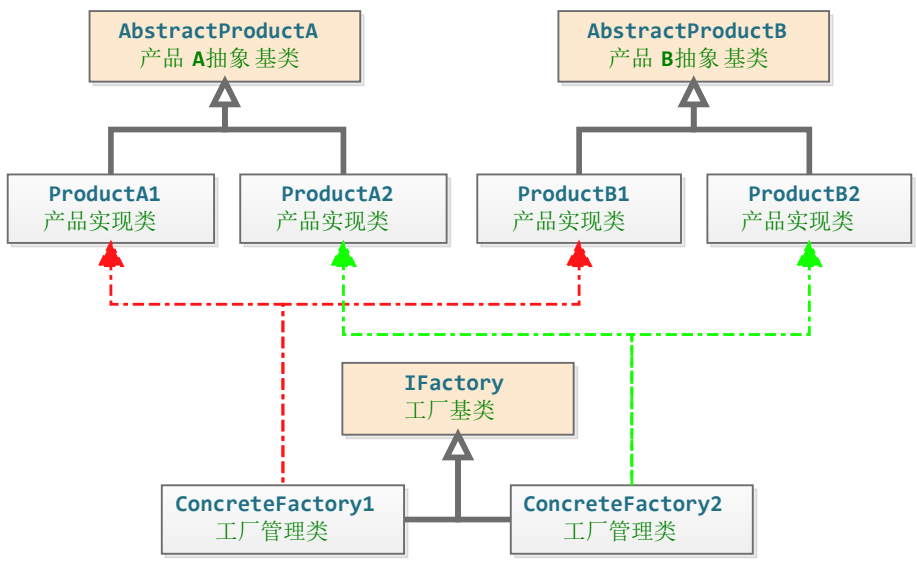
```
struct  SubOperation  :public  IOperation
{
    virtual  double  GetResult ()
    {
        return  m_nFirst  - m_nSecond ;
    };
};
```

```
struct  CCalculatorFactory  :public  IFactoryc
{
    CCalculatorFactory (){};
    ~CCalculatorFactory (){};
    IOperation * CreateProduct (char  cOperator )
    {
        IOperation * oper ;
        switch  (cOperator )
        {
            case  '+':
                oper  = new  AddOperation ();
                break ;
            case  '-':
                oper  = new  SubOperation ();
                break ;
            default :
                oper  = new  AddOperation ();
                break ;
        }
        return  oper ;
    };
};
```

```
void  Mode_Factory (int  argc , char ** argv)
{
    IFactoryc * pFac = new CCalculatorFactory ();      //创建工厂
    IOperation * pOP = NULL ;                          //产品对象指针
    pOP = pFac->CreateProduct ('+');                   //加法
    pOP->SetKeyValue (5, 3);
    std::cout  << pOP->GetResult () << std::endl ;
    pOP = pFac->CreateProduct ('-');                   //减法
    pOP->SetKeyValue (5, 3);
    std::cout  << pOP->GetResult () << std::endl ;
    delete  pOP;
    delete  pFac;
}
```

2.抽象工厂模式-AbstractFactory

特点:创建一组相关的工厂




```
//产品A的抽象
class AbstractProductA
{
protected:
    AbstractProductA () {};
public:
    virtual ~AbstractProductA ()
{};
};
```

```
//产品A1
struct ProductA1 : public AbstractProductA
{
    ProductA1 ()
    {
        printf("ProductA1( 产品A1)\n");
    };
    ~ProductA1 () {};
};
```

```
//产品A2
struct ProductA2 : public AbstractProductA
{
    ProductA2 ()
    {
        printf("ProductA2( 产品A2)\n");
    };
    ~ProductA2 () {};
};
```

```
//产品B的抽象
class AbstractProductB
{
protected:
    AbstractProductB () {};
public:
    virtual ~AbstractProductB ()
{};
};
```

```
//产品B1
struct ProductB1 : public AbstractProductB
{
    ProductB1 ()
    {
        printf("ProductB1( 产品B1)\n");
    };
    ~ProductB1 () {};
};
```

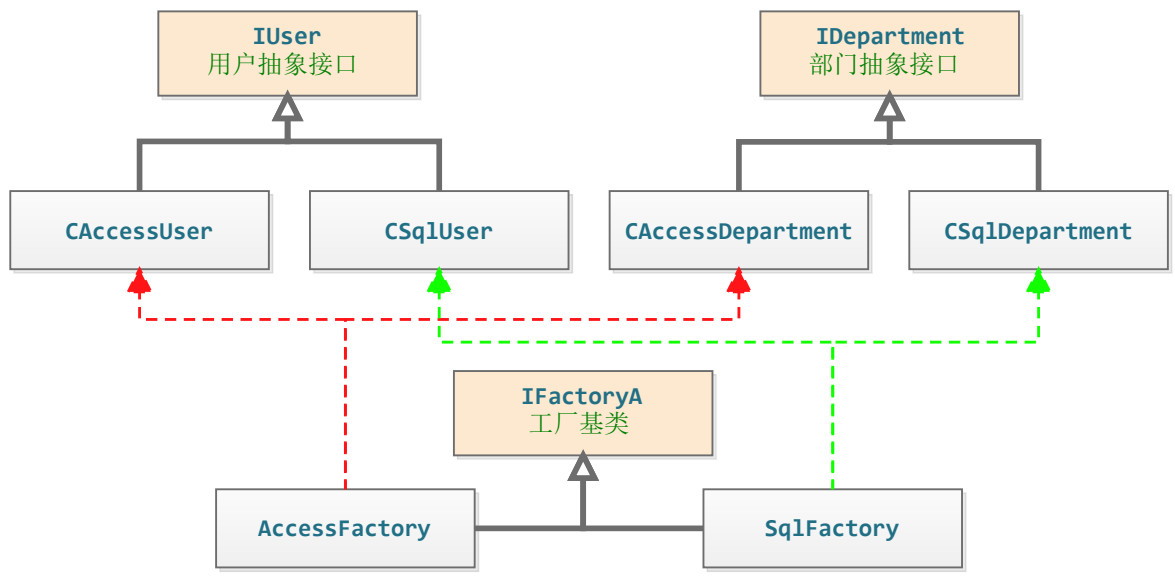
```
//产品B2
struct ProductB2 : public AbstractProductB
{
    ProductB2 ()
    {
        printf("ProductB2( 产品B2)\n");
    };
    ~ProductB2 () {};
};
```

```
//抽象工厂基类 ,生产产品A和产品B
class AbstractFactory
{
protected:
    AbstractFactory () {};
public:
    virtual ~AbstractFactory () {};
    virtual  AbstractProductA * CreateProductA () = 0;
    virtual  AbstractProductB * CreateProductB () = 0;
};
```

```
//生产产品A和产品B的第一种实现
struct ConcreteFactory1 : public AbstractFactory
{
    ConcreteFactory1 () {};
    ~ConcreteFactory1 () {};
    AbstractProductA * CreateProductA ()
    {
        return new ProductA1 ();
    };
    AbstractProductB * CreateProductB ()
    {
        return new ProductB1 ();
    };
};
```

```
//生产产品A和产品B的第二种实现
struct ConcreteFactory2 : public AbstractFactory
{
    ConcreteFactory2 () {};
    ~ConcreteFactory2 () {};
    AbstractProductA * CreateProductA ()
    {
        return new ProductA2 ();
    };
    AbstractProductB * CreateProductB ()
    {
        return new ProductB2 ();
    };
};
```

```
int main(int argc, char** argv)
{
    std::shared_ptr<AbstractFactory> cf1(new ConcreteFactory1 ());
    cf1->CreateProductA ();  //生产产品A的第一种实现
    cf1->CreateProductB ();  //生产产品B的第一种实现
    std::shared_ptr<AbstractFactory> cf2(new ConcreteFactory2 ());
    AbstractProductA * pProductA = cf2->CreateProductA ();    //生产产品A的第二种实现
    AbstractProductB * pProductB = cf2->CreateProductB ();    //生产产品B的第二种实现
    delete pProductB;    delete pProductA;
};
```




```
//用户抽象接口
struct IUser
{
    virtual void GetUser() = 0;
    virtual void InsertUser() = 0;
};
```

```
//部门抽象接口
struct IDepartment
{
    virtual void GetDepartment () = 0;
    virtual void InsertDepartment () = 0;
};
```

```
//抽象工厂
struct IFactoryA
{
    virtual IUser* CreateUser () = 0;
    virtual IDepartment * CreateDepartment () = 0;
};
```

```
//ACCESS 用户
struct CAccessUser : public IUser
{
    virtual void GetUser()
    {
        printf("GetUser( 获取Access用户)\n");
    };
    virtual void InsertUser()
    {
        printf("InsertUser( 插入Access用户)\n");
    };
};
```

```
//ACCESS 部门
struct CAccessDepartment : public IDepartment
{
    virtual void GetDepartment ()
    {
        printf("GetDepartment( 获得Access部门)\n");
    };
    virtual void InsertDepartment ()
    {
        printf("InsertDepartment( 插入Access部门)\n");
    };
};
```

```
//ACCESS 工厂
struct AccessFactory : public IFactoryA
{
    virtual IUser* CreateUser ()
    {
        return new CAccessUser ();
    };
    virtual IDepartment * CreateDepartment ()
    {
        return new CAccessDepartment ();
    };
};
```

```
//SQL 用户
struct CSqlUser : public IUser
{
    virtual void GetUser()
    {
        printf("Sql User( 获取Sql用户)\n");
    };
    virtual void InsertUser()
    {
        printf("Sql InsertUser( 插入Sql用户)\n");
    };
};
```

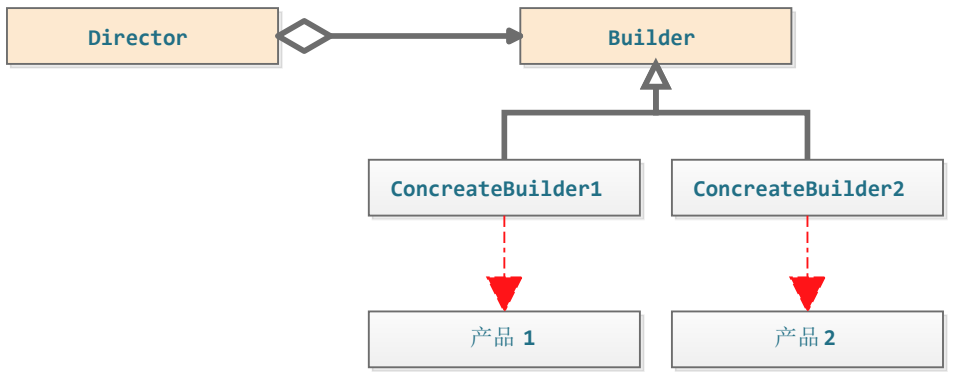
```
//SQL 部门类
struct CSqlDepartment : public IDepartment
{
    virtual void GetDepartment ()
    {
        printf("getDepartment( 获得sql部门)\n");
    };
    virtual void InsertDepartment ()
    {
        printf("insertdepartment( 插入sql部门)\n");
    };
};
```

```
//SQL 工厂
struct SqlFactory : public IFactoryA
{
    virtual IUser* CreateUser ()
    {
        return new CSqlUser ();
    };
    virtual IDepartment * CreateDepartment ()
    {
        return new CSqlDepartment ();
    };
};
```

```
int main(int argc, char** argv)
{
    //可以实现换数据库的功能  (为了增强理解加入强制类型转换语意  )
    std::shared_ptr<IFactoryA> Acc(new AccessFactory ());        //创建需要的工厂类型
    std::shared_ptr<IFactoryA> Sql(new SqlFactory ());
    //操作acc数据库
    IUser* user = (CAccessUser *)Acc->CreateUser ();
    IDepartment * depart = (CAccessDepartment *)Acc->CreateDepartment ();
    user->GetUser ();
    depart->GetDepartment ();
    //操作sql数据库
    user = (CSqlUser *)Sql->CreateUser ();
    depart = (CSqlDepartment *)Sql->CreateDepartment ();
    user->GetUser ();
    depart->GetDepartment ();
    delete depart;    delete user;
};
```

3.生成器模式-Builder

特点:将复杂对象的构造与表示分离



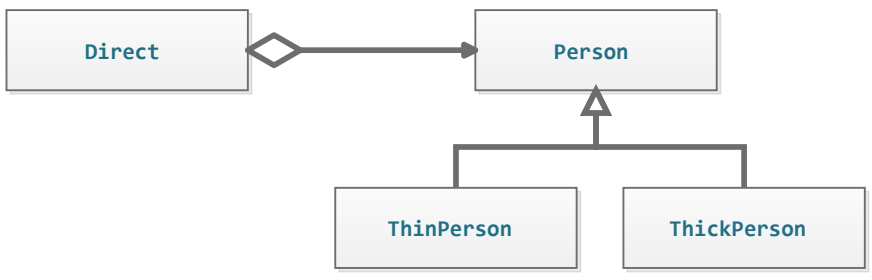

```
//使用 Builder 构建产品 ,构建产品的过程都一致 ,但是不同的 builder 有不同的实现
//这个不同的实现通过不同的 Builder 派生类来实现 ,存有一个 Builder 的指针 ,通过这个来实现多态调用
class Director
{
private:
    Builder * m_pBuilder;
public:
    Director(Builder * pBuilder) : m_pBuilder(pBuilder) {};
    ~Director()
    {
        delete m_pBuilder;
        m_pBuilder = NULL;
    };
    //Construct 函数表示一个对象的整个构建过程 ,不同的部分之间的装配方式都是一致的 ,
    //首先构建 PartA 其次是 PartB, 只是根据不同的构建者会有不同的表示
    void Construct()
    {
        m_pBuilder->BuilderPartA();
        m_pBuilder->BuilderPartB();
    };
};
```

```
//虚拟基类 ,是所有 Builder 的基类 ,提供不同部分的构建接口函数
struct Builder
{
    Builder() {};
    virtual ~Builder() {};
    virtual void BuilderPartA() = 0;
    virtual void BuilderPartB() = 0;
};
```

```
struct ConcreateBuilder1 : public Builder
{
    virtual void BuilderPartA()
    {
        printf("BuilderPartA by ConcreateBuilder1\n");
    };
    virtual void BuilderPartB()
    {
        printf("BuilderPartB by ConcreateBuilder1\n");
    };
};
```

```
int main(int argc, char** argv)
{
    std::shared_ptr<Builder> pBuilder1(new ConcreateBuilder1);
    std::shared_ptr<Director> pDirector1(new Director(pBuilder1));
    pDirector1->Construct();
    std::shared_ptr<Builder> pBuilder2(new ConcreateBuilder2);
    std::shared_ptr<Director> pDirector2(new Director(pBuilder2));
    pDirector2->Construct();
};
```

```
struct ConcreateBuilder2 : public Builder
{
    virtual void BuilderPartA()
    {
        printf("BuilderPartA by ConcreateBuilder2\n");
    };
    virtual void BuilderPartB()
    {
        printf("BuilderPartA by ConcreateBuilder2\n");
    };
};
```




```
//指挥者类
class Direct
{
private:
    Person* p;
public:
    Direct(Person* temp)
    {
        p = temp;
    };
    virtual ~Direct() {};
    void Create()
    {
        p->CreateHead();
        p->CreateBody();
        p->CreateHand();
        p->CreateFoot();
    };
};
```

```
//建造者类
struct Person
{
    virtual void CreateHead() = 0;
    virtual void CreateHand() = 0;
    virtual void CreateBody() = 0;
    virtual void CreateFoot() = 0;
};
```

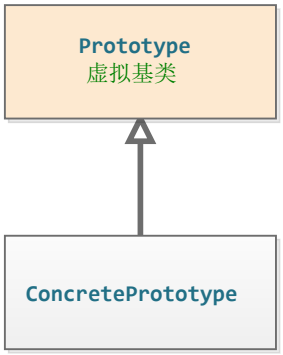
```
struct ThickPerson : public Person
{
    virtual void CreateHead()
    {
        printf("ThickPerson head\n");
    };
    virtual void CreateHand()
    {
        printf("ThickPerson hand\n");
    };
    virtual void CreateBody()
    {
        printf("ThickPerson body\n");
    };
    virtual void CreateFoot()
    {
        printf("ThickPerson foot\n");
    };
};
```

```
struct ThinPerson : public Person
{
    virtual void CreateHead()
    {
        printf("thin head\n");
    };
    virtual void CreateHand()
    {
        printf("thin hand\n");
    }
    virtual void CreateBody()
    {
        printf("thin body\n");
    };
    virtual void CreateFoot()
    {
        printf("thin foot\n");
    };
};
```

```
int main(int argc, char** argv)
{
    std::shared_ptr<Person> p(new ThickPerson());
    std::shared_ptr<Direct> d(new Direct(p));
    d->Create();
};
```

4.原型模式-Prototype

特点:指定类的原型实例，克隆该实例可以生成新的对象




```
//虚拟基类，所有原型的基类 ，提供Clone接口函数
class Prototype
{
protected:
    Prototype() {};
public:
    virtual ~Prototype() {};
    virtual Prototype* Clone() const = 0;
};
```

```
void main(int argc, char** argv)
{
    Prototype* p = new ConcretePrototype();
    Prototype* p1 = p->Clone();
    delete p;
    delete p1;
}
```

```
struct ConcretePrototype : public Prototype
{
    ConcretePrototype() {};
    ~ConcretePrototype() {};
    ConcretePrototype(const ConcretePrototype& cp)
    {
        printf("ConcretePrototype1 copy \n");
    };
    Prototype* Clone() const
    {
        return new ConcretePrototype(*this);
    };
};
```

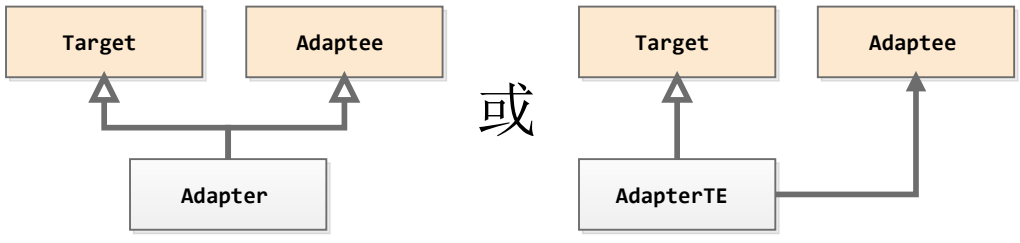
5.单例模式-Singleton

```
class Singleton
{
private:
    //这样就有唯一的对象了    维护静态变量
    static Singleton* _instance;
protected:
    //注意单键不要能实例化
    Singleton()
    {
        printf("Singleton\n");
    };
public:
    ~Singleton() {};
    static Singleton* Instance()
    {
        if (NULL == _instance)
            _instance = new Singleton();
        return _instance;
    };
};

Singleton* Singleton::_instance = 0;

int Mode_Singleton(int argc, char** argv)
{
    //单例创建过程
    Singleton* sgn = Singleton::Instance();
    return 1;
}
```

特点:确保类只有一个实例
(达到创建一个全局变量(对象)效果)

6.适配器模式-Adapter

特点:将类的接口转换为另一种接口
(转换不兼容的接口类)




```
//需要被 Adapt 的类
struct  Target
{
    Target () {};
    virtual  ~Target () {};
    virtual  void  Request ()
    {
        printf ("Target::Request\n" );
    };
};
```

```
//与被 Adapt 对象提供不兼容接口的类
struct  Adaptee
{
    Adaptee () {};
    virtual  ~Adaptee () {};
    void  SpecificRequest ()
    {
        printf ("Adaptee::SpecificRequest\n"  );
    };
};
```

```
//多重继承 (组合 )也能聚合原有接口类的方式
struct  Adapter  :public  Target ,private  Adaptee
{
    Adapter () {};
    virtual  ~Adapter () {};
    virtual  void  Request ()
    {
        this ->SpecificRequest ();
    };
};
```

```
void  main (int  argc , char** argv )
{
    Adaptee * pAdaptee  = new  Adaptee ;
    Target * pTarget  = new  AdapterTE (pAdaptee );
    pTarget ->Request ();
    // 直接将Adaptee 转换为Target 接口一致的类
    Adapter * pAdapter  = new  Adapter ();
    pAdapter ->Request ();
};
```

```
//进行 Adapt 的类,采用继承原有接口类的方式
class  AdapterTE  :public  Target
{
private :
    Adaptee * m_pAdptee ;
public :
    AdapterTE (Adaptee * pAdaptee ) : m_pAdptee (pAdaptee ){};
    virtual  ~AdapterTE ()
    {
        delete  m_pAdptee ;
        m_pAdptee  = NULL ;
    };
    virtual  void  Request ()
    {
        m_pAdptee ->SpecificRequest ();
    };
};
```

```
//为中场翻译
class  TransLater  : public  Player
{
private :
    Center*  player ;
public :
    TransLater (string  strName ) : Player( strName )
    {
        player  = new  Center( strName );
    };
    virtual  void  Attack ()
    {
        player ->Attack ();
    };
    virtual  void  Defense ()
    {
        player ->Defense ();
    };
}
```

```
class  Player
{
protected :
    string  name
public :
    Player (string  strName )
    {
        name  = strName
    };
    virtual  void  Attack () = 0;
    virtual  void  Defense () = 0;
}
```

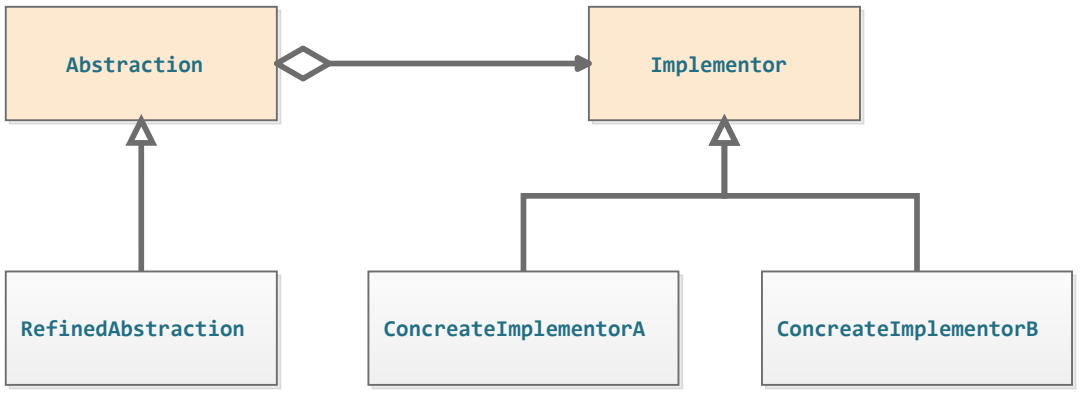
```
//前锋
class  Forwards  : public  Player
{
public :
    Forwards (string  strName ): Player( strName ) {};
public :
    virtual  void  Attack ()
    {
        cout  << name  << "前锋进攻 "  << endl ;
    };
    virtual  void  Defense ()
    {
        cout  << name  << "前锋防守 "  << endl ;
    };
}
```

```
//中场
class  Center  : public  Player
{
public :
    Center (string  strName ) : Player( strName ) {};
public :
    virtual  void  Attack ()
    {
        cout  << name  << "中场进攻 "  << endl ;
    };
    virtual  void  Defense ()
    {
        cout  << name  << "中场防守 "  << endl ;
    };
}
```

```
int  main ()
{
    Player*  p = new  TransLater( "小李 ");
    p->Attack();
    return  0;
}
```

7.桥接模式-Bridge

特点:将抽象部分与它的实现部分分离,使它们都可以独立地变化.
(系统的耦合性也得到了很好的降低　)




```
class  Abstraction
{
protected :
    Abstraction () {};
public :
    virtual  ~Abstraction () {};
    virtual  void  Operation () = 0;
};
```

```
//为实现 Abstraction 定义的抽象基类 ,定义了实现的接口函数
class  Implementor
{
protected :
    Implementor () {};
public :
    virtual  ~Implementor () {};
    virtual  void  Operation () = 0;
};
```

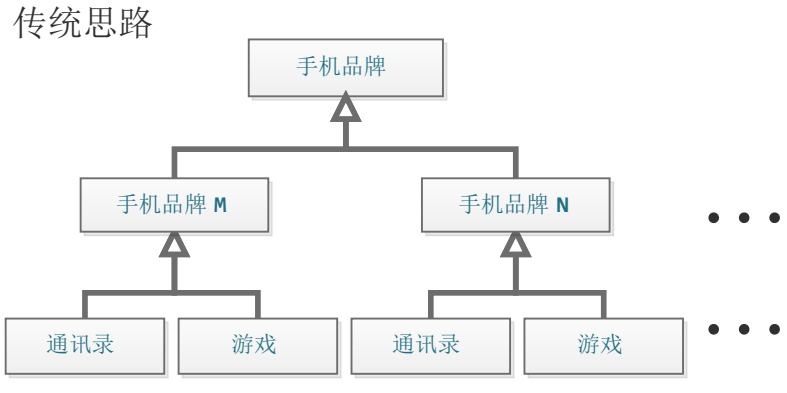
```
//维护一个 Implementor 类的指针
class  RefinedAbstraction : public  Abstraction
{
private :
    Implementor * m_pImp ;
public :
    RefinedAbstraction (Implementor * pImp ):m_pImp (pImp ){};
    ~RefinedAbstraction ()
    {
        delete  m_pImp ;
        m_pImp  = NULL ;
    };
    void  Operation ()
    {
        m_pImp ->Operation ();
    };
};
```

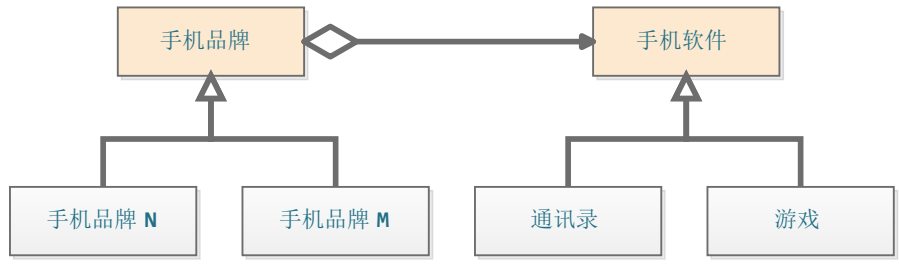
```
//继承自 Implementor, 是Implementor 的不同实现之一
struct  ConcreateImplementorA  : public  Implementor
{
    ConcreateImplementorA () {};
    virtual  ~ConcreateImplementorA () {};
    virtual  void  Operation ()
    {
        printf ("ConcreateImplementorA\n"  );
    };
};
```

```
//继承自 Implementor, 是Implementor 的不同实现之一
struct  ConcreateImplementorB  : public  Implementor
{
    ConcreateImplementorB () {};
    ~ConcreateImplementorB () {};
    virtual  void  Operation ()
    {
        printf ("ConcreateImplementorB\n"  );
    };
};
```

```
int main(int  argc , char* argv[])
{
    Implementor*  imp = new ConcreteAbstractionImpA();
    Abstraction*  abs = new RefinedAbstraction( imp);
    abs->Operation ();
    return  0;
}
```

传统思路






```
//手机品牌
class  HandsetBrand
{
protected :
    HandsetSoft*  m_soft ;
public :
    void  SetHandsetSoft (HandsetSoft*  temp )
    {
        m_soft  = temp ;
    }
    virtual  void  Run() = 0;
};
```

```
//M品牌
struct  HandsetBrandM  : public  HandsetBrand
{
    virtual  void  Run()
    {
        m_soft-> Run();
    }
};
```

```
//手机软件
struct  HandsetSoft
{
    virtual  void  Run()= 0;
};
```

```
//N品牌
struct  HandsetBrandN  : public  HandsetBrand
{
    virtual  void  Run()
    {
        m_soft-> Run();
    }
};
```

```
//游戏软件
struct  HandsetGame  : public  HandsetSoft
{
    virtual  void  Run()
    {
        cout  << "运行手机游戏  "  << endl ;
    }
};
```

```
int main()
{
    HandsetBrand * brand  = new  HandsetBrandM();
    brand ->SetHandsetSoft( new  HandsetGame ());
    brand ->Run();
    brand ->SetHandsetSoft( new  HandSetAddressList ());
    brand ->Run();
    return  0;
}
```

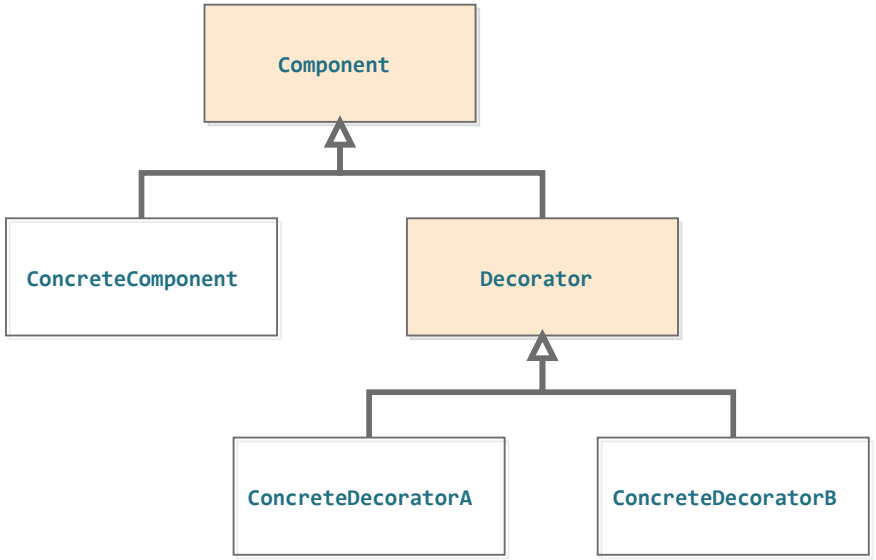
```
//通讯录软件
struct  HandSetAddressList :public  HandsetSoft
{
    virtual  void  Run()
    {
        cout  << "手机通讯录  "  << endl ;
    }
};
```

8.组合模式-Composite

特点:将对象组合成树型结构，表示“部分-整体”的层次结构

9.装饰模式-Decorator

特点:动态地给一个对象添加一些额外的行为




```
class ConcreteComponent : public Component
{
public:
    ConcreteComponent () {};
    ~ConcreteComponent () {};
    void Operation ()
    {
        cout << "ConcreteComponent operation..." <<endl;
    };
};
```

```
class Component
{
public:
    virtual ~Component () {};
    virtual void Operation () {};
protected:
    Component () {};
};
```

```
class Decorator : public Component
{
public:
    Decorator (Component * com)
    {
        this->_com = com;
    };
    virtual ~Decorator ()
    {
        delete _com;
    };
    void Operation () {};
protected:
    Component * _com;
};
```

```
int main(int argc, char** argv)
{
    Component * com =new ConcreteComponent ();
    Decorator * dec =new ConcreteDecorator (com);
    dec->Operation ();
    delete dec;
    return 1;
}
```

```
class ConcreteDecoratorA : public Decorator
{
public:
    ConcreteDecoratorA (Component * com) : Decorator (com){};
    ~ConcreteDecoratorA () {};
    void Operation ()
    {
        _com->Operation ();
        this->AddedBehavior ();
    };
    void AddedBehavior ()
    {
        cout<<"ConcreteDecoratorA::AddedBehacior...." <<endl;
    };
};
```

10.享元模式-Flyweight

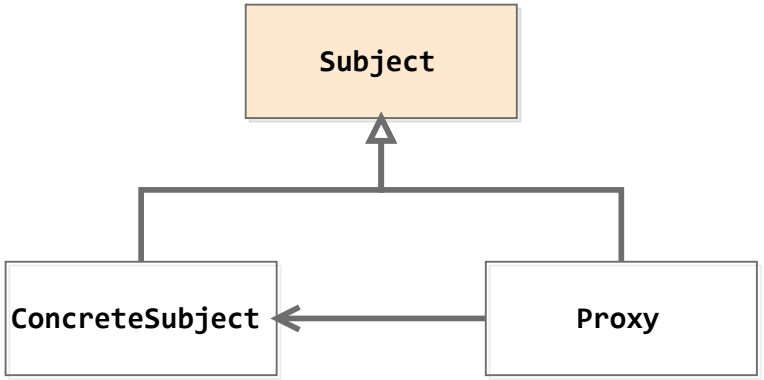
特点:利用共享技术高效的支持大量细粒度的对象

11.外观模式-Facade

特点:为子系统中的一组接口提供统一的高层次接口

12.代理模式-Proxy

特点:提供另一个对象的代替物或占位符,以便控制对该对象的访问
(实现了逻辑和实现的彻底解耦)




```cpp
//定义接口
struct Subject
{
    virtual ~Subject() {};
    virtual void Request() = 0;
protected:
    Subject() {};
};
```

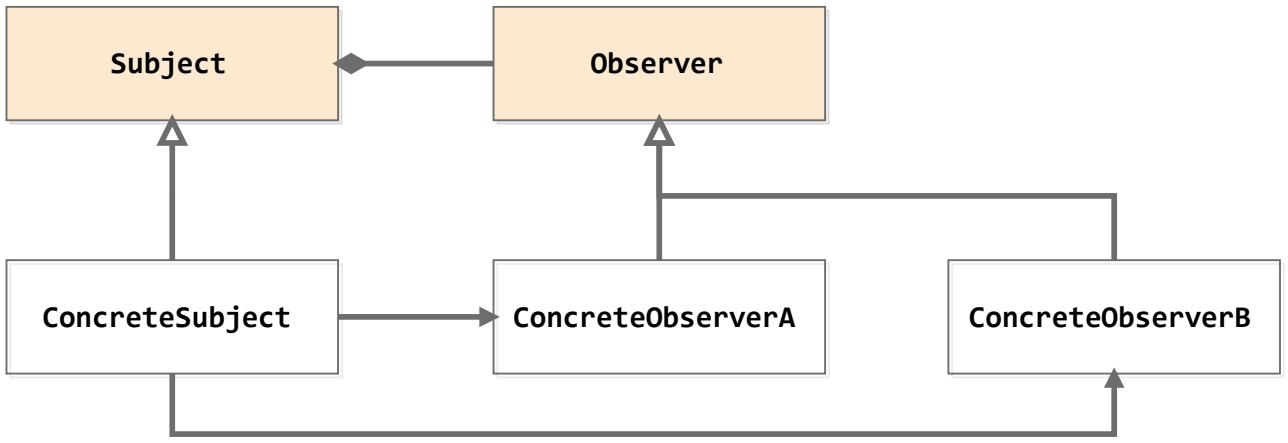
```cpp
//代理类
class Proxy
{
private:
    Subject* _sub;
public:
    Proxy() {};
    Proxy(Subject* sub)
    {
        _sub = sub;
    };
    ~Proxy()
    {
        delete _sub;
    };
    void Request()
    {
        printf("Proxy request...." );
        _sub->Request();
    };
};
```

```cpp
//真实类
struct ConcreteSubject : public Subject
{
    ConcreteSubject() {};
    ~ConcreteSubject() {};
    void Request()
    {
        printf("真实的请求 !");
    };
};
```

```cpp
int main(int argc, char** argv)
{
    Subject* sub = new ConcreteSubject();
    Proxy* p = new Proxy(sub);
    p->Request();
    //p的Request请求实际上是交给了   sub来实际执行
    //实现了逻辑和实现的彻底解耦    .
    return 1;
}
```

13.观察者模式-Observer

特点:定义对象间的一种一对多的依赖关系，当对象的状态发生改变时，所有依赖与它的对象都将得到通知




```
struct Observer
{
    virtual ~Observer() {};
    virtual void Update(Subject* sub) = 0;
    virtual void PrintInfo(void) = 0;
protected:
    Observer(){ _st = '\0'; };
    State _st;
};
```

```
struct Subject
{
    virtual ~Subject() {};
    virtual void Attach(Observer* obv)
    {
        obvs->push_front(obv);
    };
    virtual void Detach(Observer* obv)
    {
        if (obv != NULL)
            obvs->remove(obv);
    };
    virtual void Notify(void)
    {
        for (auto it=obvs->begin();it!=obvs->end();it++)
        {   //关于模板和 iterator 的用法
            (* it)->Update(this);
        }
    };
    virtual void SetState(const State& st) = 0;
    virtual State GetState(void) = 0;
protected:
    Subject()
    {   //在模板的使用之前一定要   new,创建
        obvs = new list<Observer*>;
    };
private:
    list<Observer*>* obvs;
};
```

```
struct ConcreteObserverA : public Observer
{
    ConcreteObserverA(Subject* sub)
    {
        _sub = sub;
        _sub->Attach(this);
    };
    virtual ~ConcreteObserverA()
    {
        _sub->Detach(this);
        if (_sub != 0)
            delete _sub;
    };
    virtual Subject* GetSubject()
    {
        return _sub;
    };
    //传入 Subject 作为参数 ,这样可以
    //让一个 View 属于多个的 Subject。
    void Update(Subject* sub)
    {
        _st = sub->GetState();
        PrintInfo();
    };
    void PrintInfo(void)
    {
        cout << "ConcreteObserverA "
             << _sub->GetState() << endl;
    };
private:
    Subject* _sub;
};
```

```
class ConcreteObserverB : public Observer
{
public:
    ConcreteObserverB(Subject* sub)
    {
        _sub = sub;
        _sub->Attach(this);
    }   ;
    virtual ~ConcreteObserverB()
    {
        _sub->Detach(this);
        if (_sub != 0)
            delete _sub;
    };
    virtual Subject* GetSubject()
    {
        return _sub;
    };
    //传入 Subject 作为参数 ,这样可以
    //让一个 View 属于多个的 Subject。
    void Update(Subject* sub)
    {
        _st = sub->GetState();
        PrintInfo();
    };
    void PrintInfo(void)
    {
        cout << "ConcreteObserverB "
             << _sub->GetState() << endl;
    };
private:
    Subject* _sub;
};
```

```
struct ConcreteSubject : public Subject
{
    ConcreteSubject()
    {
        _st = '\0';
    };
    ~ConcreteSubject() {};
    State GetState(void)
    {
        return _st;
    };
    void SetState(const State& st)
    {
        _st = st;
    };
private:
    State _st;
};
```

```
int main(int argc, char** argv)
{
    ConcreteSubject * sub = new ConcreteSubject();
    Observer * o1 = new ConcreteObserverA(sub);
    Observer * o2 = new ConcreteObserverB(sub);
    sub->SetState("old");
    sub->Notify();
    sub->SetState("new");  //也可以由 Observer 调用
    sub->Notify();
    return 0;
}
```

14.策略模式-Strategy

特点:定义一组算法并封装每个算法，使它们在运行时可以相互替换

15.状态模式-State

特点:当对象内部状态改变时，对象看起来好像修改了它所属的类

16.迭代器模式-Iterator

特点:提供一种顺序访问某种聚合对象元素的途径

17.备忘录模式-Memento

特点:捕获对象的内部状态，以便将来可将该对象恢复到保存的状态

18.访问者模式-Visitor

特点:表述对某对象结构的元素所执行的操作

19.解释器模式-Interpreter

特点:指定如何对某种语言的语句进行表示
和判断

20.中介者模式-Mediator

特点:定义一个中介对象，用于封装一组对象的交互

21.类行为型模式-TemplateMethod （模板方法模式）

特点:定义某操作中算法的框架，将其中一
些步骤推迟到子类中

22.命令模式-Command

特点:将请求或操作封装成对象，并支持可撤销的操作

23.职责链-ChainOfResponsibility

特点:使多个接受者对象有机会处理来自发
送者对象的请求