Decorator

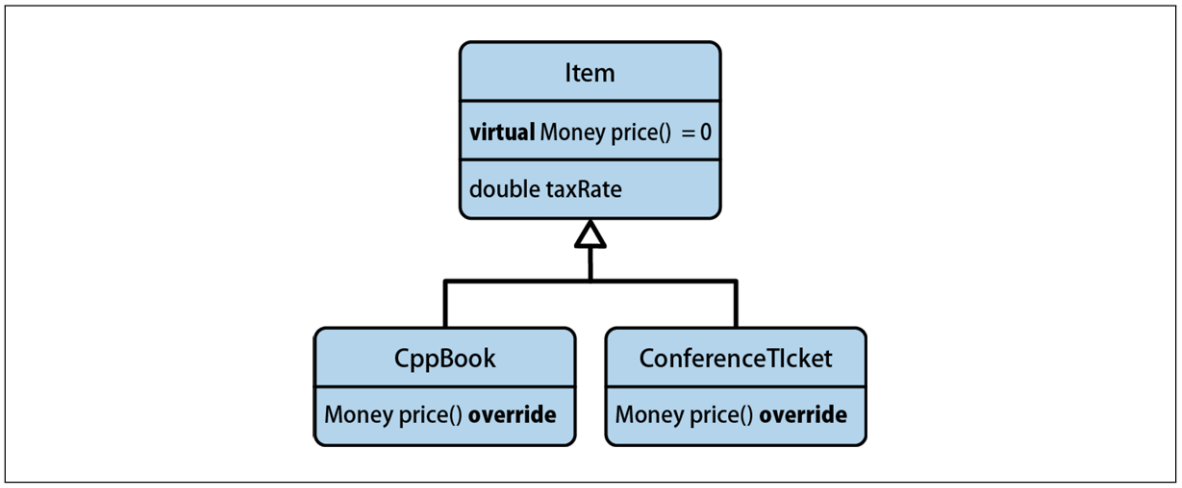

図 9-1　Item の継承階層（初期段階）

C++関係の商品を扱う商店があって売り物はItemクラスを継承するクラス構成になっている。
はじめは税金のことだけ考えればいいため、protect要素に税金の情報を入れて各具象化クラスでprice()で計算していた。

問題

- 商品が増えるごとにpriceの実装が必要
- price. つまり値段の計算に影響を与える要素が出るたびに各クラスを修正しないといけな

悪い例：値段の構成要素を継承で表現

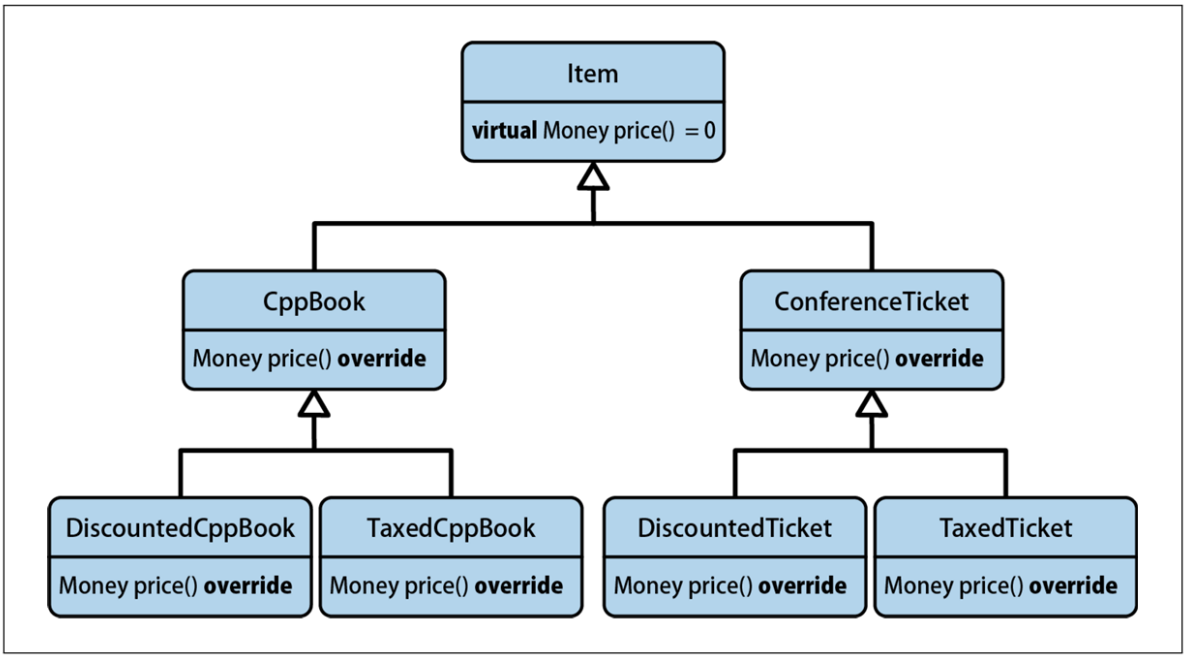

図 9-2　Item 継承階層の拡大

関心の分離：Starategyパターン

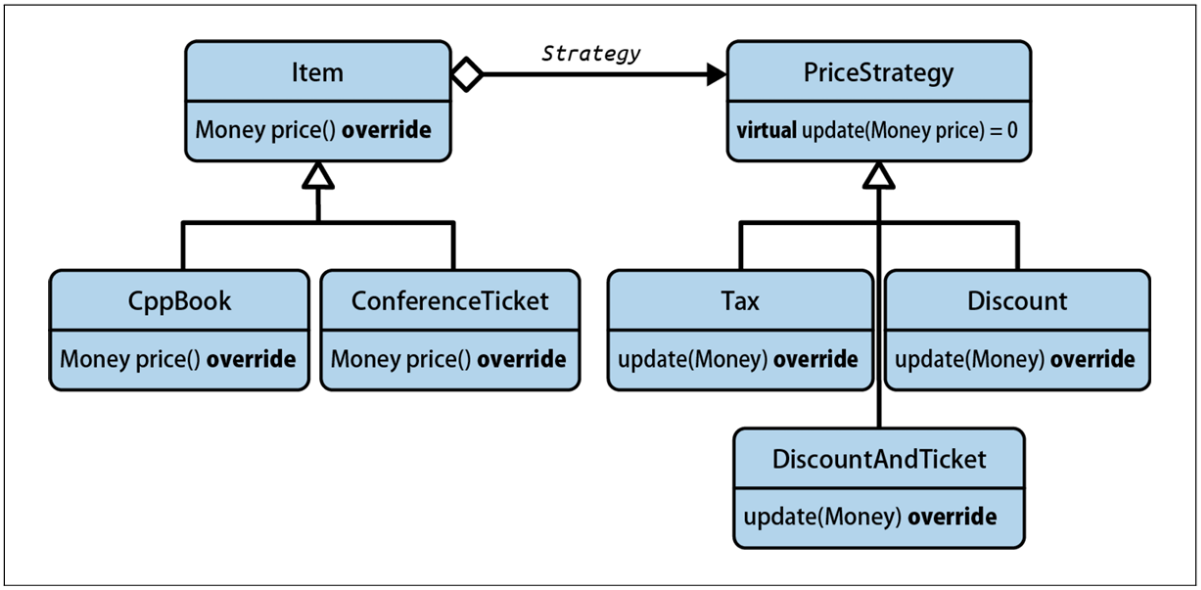

図 9-4　Strategy パターンベースの Item 継承階層

気になること

- 価格変更要因を持たない商品もPriceStrategyを持つ必要がある。nullとかで表現する？

- 価格変更要因の組み合わせには対応できていない。例えばTaxとDiscountを組み合わせはどうする

Strategyパターンの性質

- 関数の詳細にある依存関係を切り離して、問題をキレイに扱えるようにしてくれる
- 組み合わせや再利用まではサポートしない
  ここでいうとstrategyの要素になったTaxやDiscountを再利用、組み合わせるといったこと

Decoratorパターン

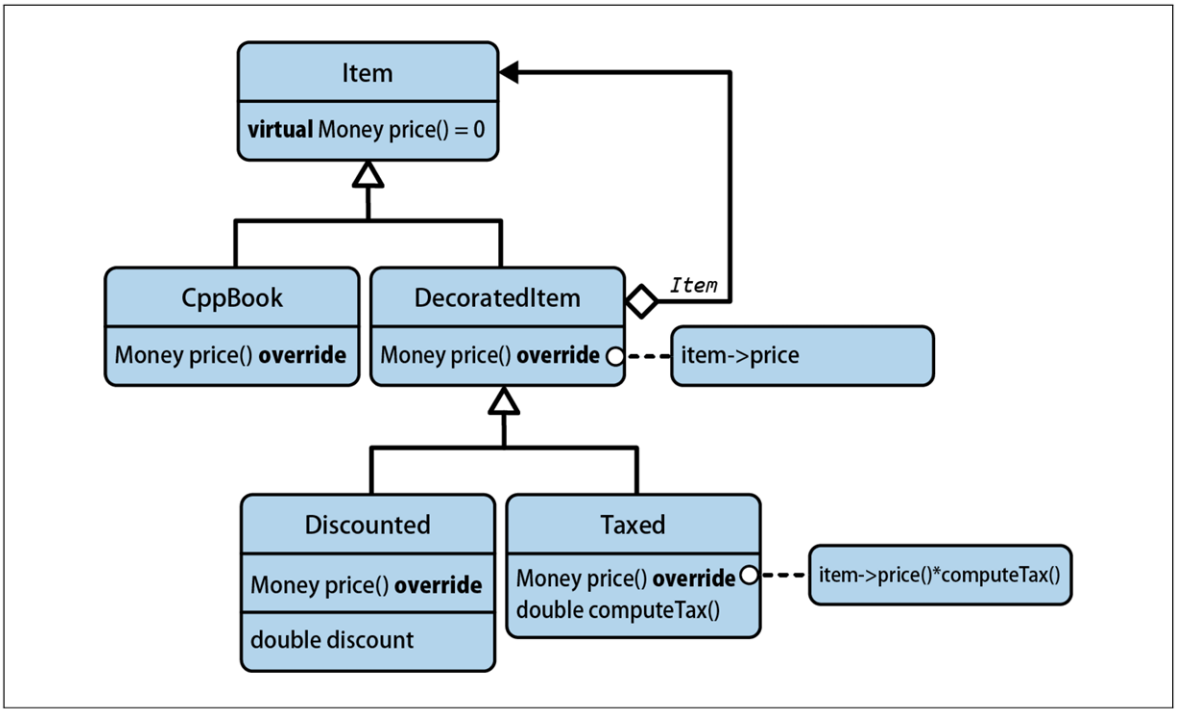

図 9-5　Decorator パターンの UML ダイアグラム

利用方法は以下のようになる最終的な商品CppBook,ConferenceTicketをDecoratorItemに渡すことでDecoratorItemで扱う価格変更要因を適用したpriceを計算できるようになっている。また、item2ではTaxedとDiscountedを組み合わせることもできている。

```
int main() {
    //税率7%: 19 * 1.07 = 20.33
```

```
    std::unique_ptr<Item> item1(
        std::make_unique<Taxed> (0.07,
            std::make_unique<CppBook> ("Effective C++", 19.0

    //税率19% 値引き 20%: 999*0.8*1.19 = 951.05
    std::unique_ptr<Item> item2(
        std::make_unique<Taxed> (0.19,
            std::make_unique<Discounted> (0.2,
                std::make_unique<ConferenceTicket> ("CppCon"

    Money const totalPrice1 = item1->price(); // 20.33
    Money const totalPrice2 = item2->price(); // 951.05
}
```

実装の紹介
Item

```
//----<Item.h>---------------
#include<Money.h>
classItem
{
    public:
         virtual ~Item() = default;
         virtual Money price() const = 0;
};
```

DecoratedItem
DecoratedItem クラスではまだprice()は純粋仮想関数のまま

```
//---- <DecoratedItem.h>---------------
#include <Item.h>
#include <memory>
#include <stdexcept>
#include <utility>
```

```
class DecoratedItem : public Item
{
    public:
    explicit DecoratedItem( std::unique_ptr<Item> item )
    : item_( std::move(item) )
    {
        if( !item_ ) {
         throw std::invalid_argument( "Invalid item" );
     }
    }
 protected:
     Item& item() { return *item_; }
     Item const& item() const { return *item_; }
 private:
     std::unique_ptr<Item> item_;
 };
```

Discounted

```
//---- <Discounted.h>---------------
#include <DecoratedItem.h>
class Discounted : public DecoratedItem
{
 public:
 Discounted( double discount, std::unique_ptr<Item> item )
     : DecoratedItem( std::move(item) )
     , factor_( 1.0- discount )
     {
         if( !std::isfinite(discount) || discount < 0.0 || di
                throw std::invalid_argument( "Invalid discoun
         }
     }

 Money price() const override
 {
     return item().price() * factor_;
 }
```

```
private:
    double factor_;
};
```

余談：Use Cases and Examples of GoF Decorator Pattern for IO
同じ抽象クラスのインスタンスを取るコンストラクタを持っている場合はデコレーター・パターンを使っている可能性がある。例えばJavaのファイルIO

First open an inputstream of it:

```
FileInputStream fis = new FileInputStream("/objects.gz");
```

We want speed, so let's buffer it in memory:

```
BufferedInputStream bis = new BufferedInputStream(fis);
```

The file is gzipped, so we need to ungzip it:

```
GzipInputStream gis = new GzipInputStream(bis);
```

We need to unserialize those Java objects:

```
ObjectInputStream ois = new ObjectInputStream(gis);
```

Now we can finally use it:

```
SomeObject someObject = (SomeObject) ois.readObject();
```