哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 5: Reusability-Oriented Software Construction Approaches

5.3 Design Patterns for Reuse

Ming Liu

March 27, 2019

Outline

- **Structural patterns**
  - Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
  - Decorator dynamically adds/overrides behavior in an existing method of an object.
  - Facade provides a simplified interface to a large body of code.
- **Behavioral patterns**
  - Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
  - Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
  - Iterator accesses the elements of an object sequentially without exposing its underlying representation.

Recall: Why reusable Designs?

A design…

…enables flexibility to change (reusability)

…minimizes the introduction of new problems when fixing old ones (maintainability)

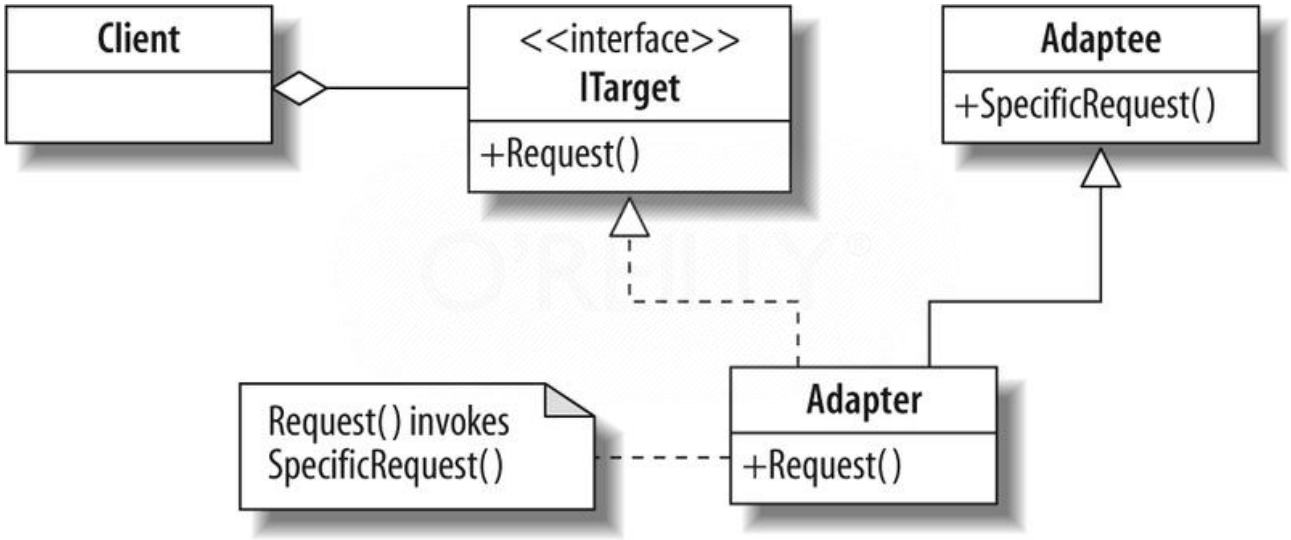
…allows the delivery of more functionality after an initial delivery (extensibility).

1 Structural patterns

(1) Adapter

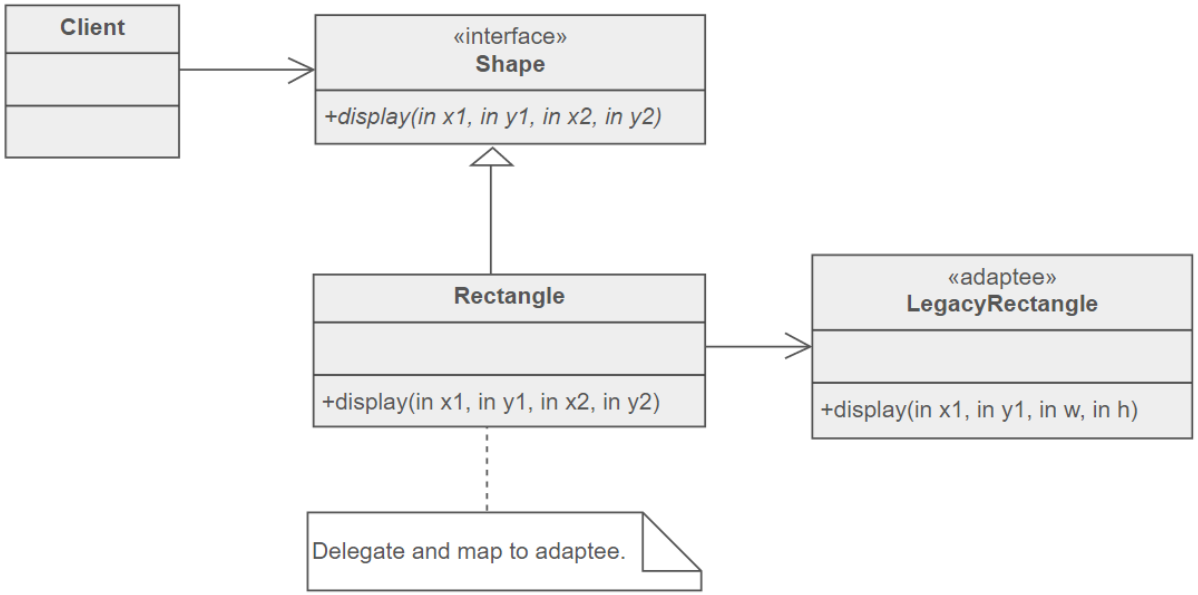
Adapter Pattern

- **Intent: Convert the interface of a class into another interface clients expect.**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
- **Object: to reuse an old component to a new system (also called "wrapper")**



Example

- **A legacy `Rectangle` component's `display()` method expects to receive "`x, y, w, h`" parameters.**
- **But the client wants to pass "upper left `x` and `y`" and "lower right `x` and `y`".**
- **This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.**



(2) Decorator

Motivating example of Decorator pattern

- **Suppose you want various extensions of a `Stack` data structure…**
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses

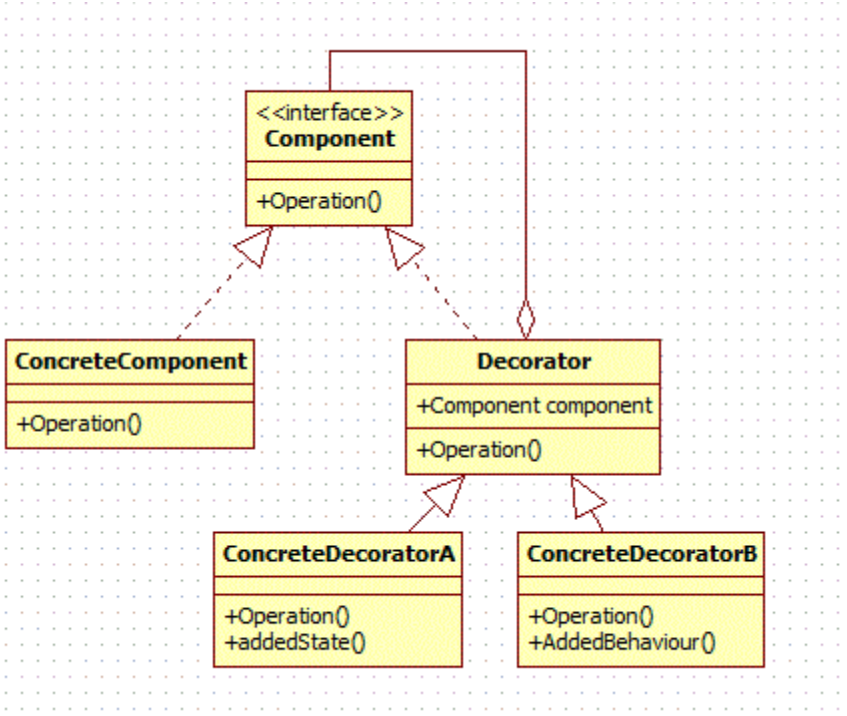
Inheritance

- **And arbitrarily composable extensions:**
  - `SecureUndoStack`: A stack that requires a password, and also lets you undo previous operations
  - `SynchronizedUndoStack`: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - `SecureSynchronizedStack`: …
  - `SecureSynchronizedUndoStack`: …

Inheritance hierarchies? Multi-Inheritance?

Decorator

- **Problem: You need arbitrary or dynamically composable extensions to individual objects.**
- **Solution: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.**
- **Consequences:**
  - More flexible than static inheritance
  - Customizable, cohesive extensions
- **Decorators use both subtyping and delegation**



The AbstractStackDecorator Class

```
public abstract class AbstractStackDecorator implements Stack {
   private final Stack stack;
   public AbstractStackDecorator(Stack stack) {
      this.stack = stack;
   }
   public void push(Item e) {
      stack.push(e);
   }
   public Item pop() {
      return stack.pop();
   }
   …
}
```

The concrete decorator classes

```
public class UndoStack
        extends AbstractStackDecorator
        implements Stack {

   private final UndoLog log = new UndoLog();
   public UndoStack(Stack stack) {
      super(stack);
   }
   public void push(Item e) {
      log.append(UndoLog.PUSH, e);
      super.push(e);
   }
   …
}
```

Using the decorator classes

- To construct a plain stack:
  - `Stack s = new ArrayStack();`
- To construct an undo stack:
  - `UndoStack s = new UndoStack(new ArrayStack());`
- To construct a secure synchronized undo stack:
  -
```
SecureStack s = new SecureStack(
                    new SynchronizedStack(
                       new UndoStack(s))
```

- **Flexibly Composible!**

Decorator vs. Inheritance

- **Decorator composes features at run time**
  - Inheritance composes features at compile time
- **Decorator consists of multiple collaborating objects**
  - Inheritance produces a single, clearly-typed object
- **Can mix and match multiple decorations**
  - Multiple inheritance is conceptually difficult

Decorators from `java.util.Collections`

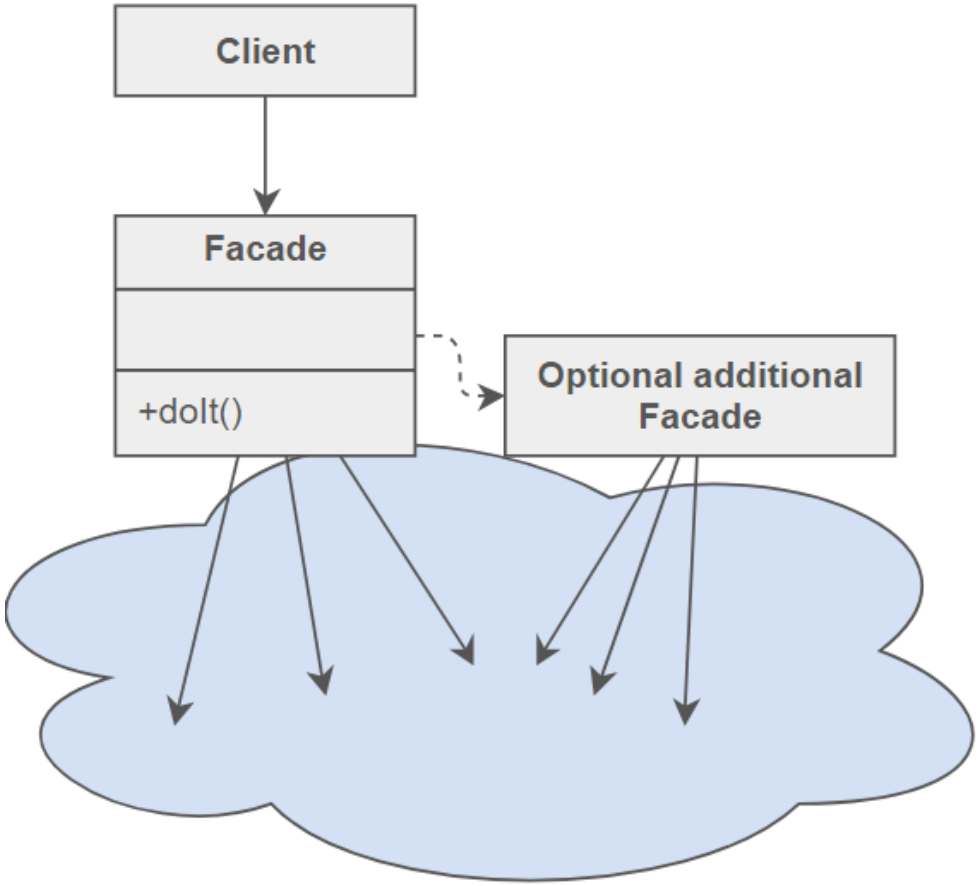
- **Turn a mutable list into an immutable list:**
  - `static List<T>  unmodifiableList(List<T> lst);`
  - `static Set<T>   unmodifiableSet( Set<T> set);`
  - `static Map<K,V> unmodifiableMap( Map<K,V> map);`
- **Similar for synchronization:**
  - `static List<T>  synchronizedList(List<T> lst);`
  - `static Set<T>   synchronizedSet( Set<T> set);`
  - `static Map<K,V> synchronizedMap( Map<K,V> map);`

(3) Facade

Facade

- **Problem**
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.
- **This reduces the learning curve necessary to successfully leverage the subsystem.**
- **It also promotes decoupling the subsystem from its potentially many clients.**

Facade



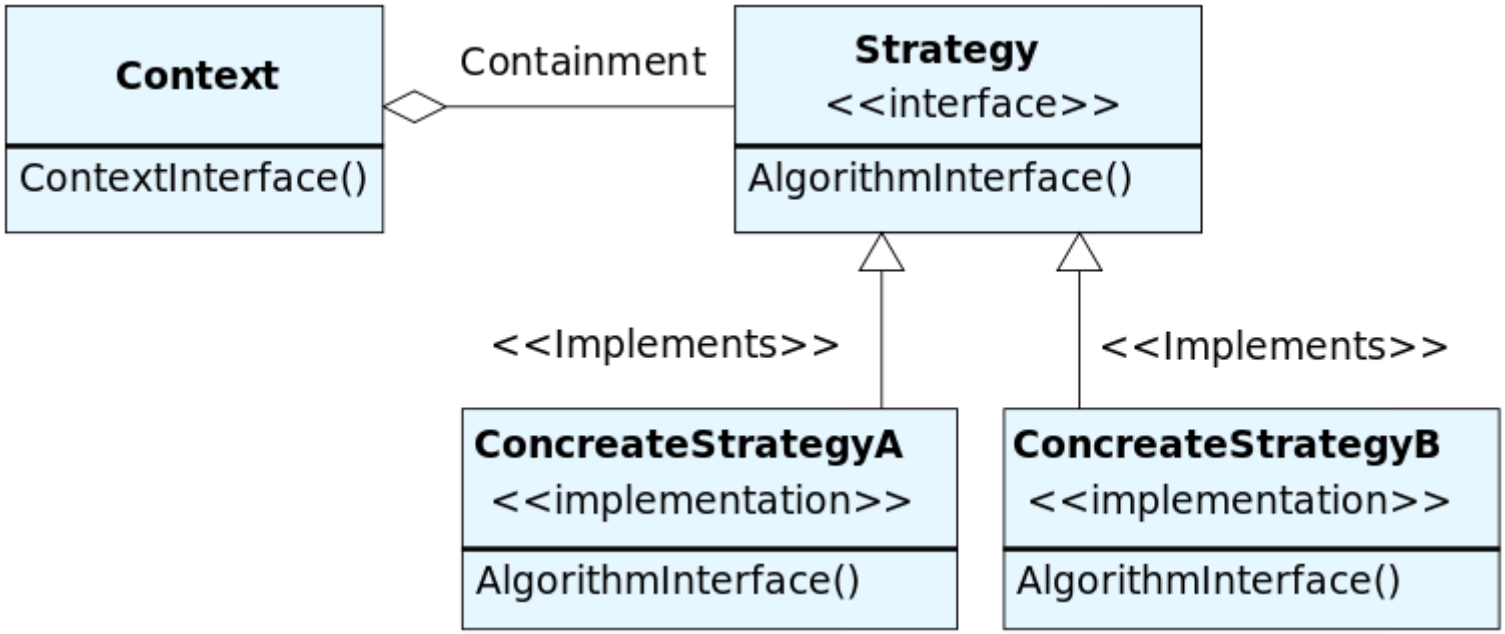
2 Behavioral patterns

(1) Strategy

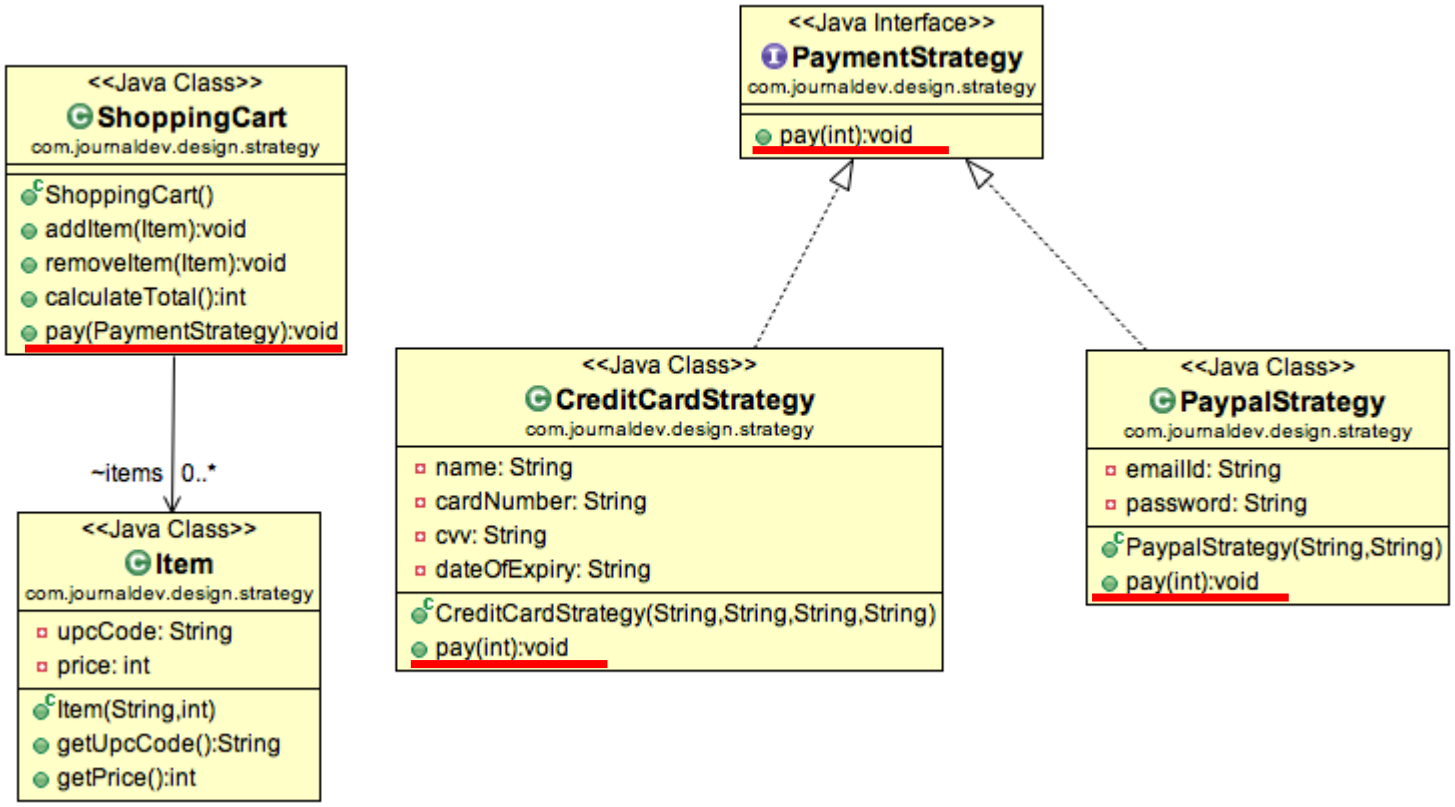
Strategy Pattern

- **Problem:** Different algorithms exists for a specific task, but client can switch between the algorithms at run time in terms of dynamic context.
- **Example:** Sorting a list of customers (Bubble sort, mergesort, quicksort)
- **Solution:** Create an interface for the algorithm, with an implementing class for each variant of the algorithm.
- **Advantage:**
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context

Strategy Pattern

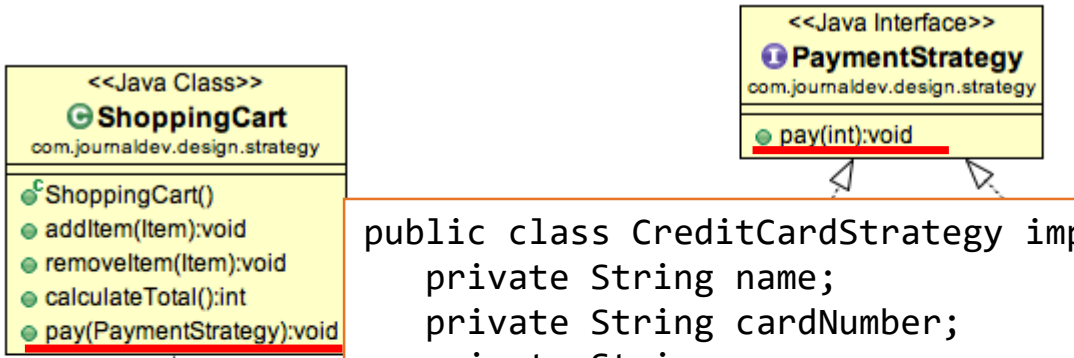


Code example



Code example

```
public interface PaymentStrategy {
      public void pay(int amount);
}
```

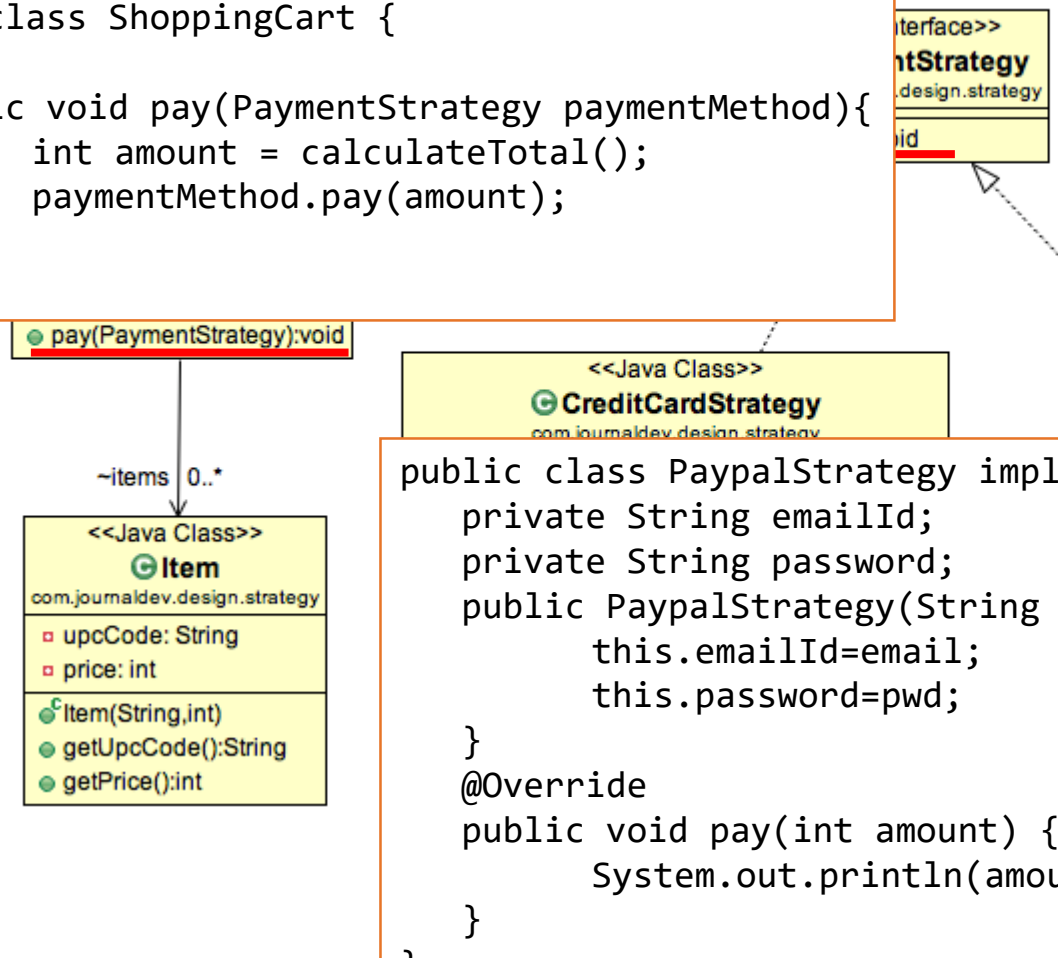



```
public class CreditCardStrategy implements PaymentStrategy {
   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;
   public CreditCardStrategy(String nm, String ccNum,
               String cvv, String expiryDate){
         this.name=nm;
         this.cardNumber=ccNum;
         this.cvv=cvv;
         this.dateOfExpiry=expiryDate;
   }
   @Override
   public void pay(int amount) {
         System.out.println(amount +" paid with credit card");
   }
}
```

Code example

```
public interface PaymentStrategy {
      public void pay(int amount);
}
```

```
public class ShoppingCart {
    ...
    public void pay(PaymentStrategy paymentMethod){
          int amount = calculateTotal();
          paymentMethod.pay(amount);
    }
}
```

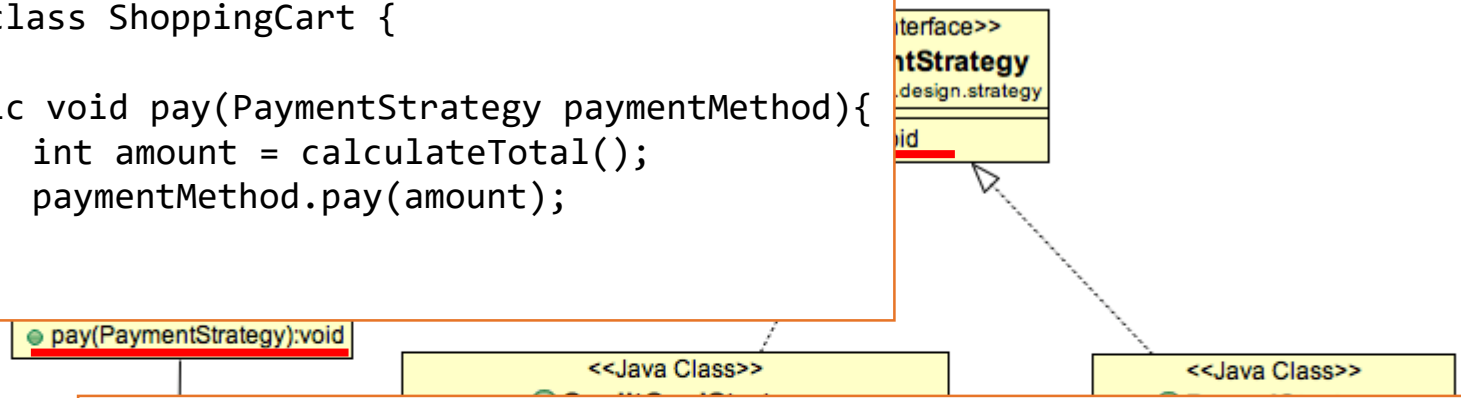



```
public class PaypalStrategy implements PaymentStrategy {
    private String emailId;
    private String password;
    public PaypalStrategy(String email, String pwd){
          this.emailId=email;
          this.password=pwd;
    }
    @Override
    public void pay(int amount) {
          System.out.println(amount + " paid using Paypal.");
    }
}
```

Code example

```
public interface PaymentStrategy {
     public void pay(int amount);
}
```

```
public class ShoppingCart {
   ...
   public void pay(PaymentStrategy paymentMethod){
         int amount = calculateTotal();
         paymentMethod.pay(amount);
   }
}
```




```
public class ShoppingCartTest {
   public static void main(String[] args) {
         ShoppingCart cart = new ShoppingCart();
         Item item1 = new Item("1234",10);
         Item item2 = new Item("5678",40);
         cart.addItem(item1);
         cart.addItem(item2);
         //pay by paypal
         cart.pay(new PaypalStrategy("myemail@exp.com", "mypwd"));
         //pay by credit card
         cart.pay(new CreditCardStrategy(“Alice", "1234", "786", "12/18"));
   }
}
```

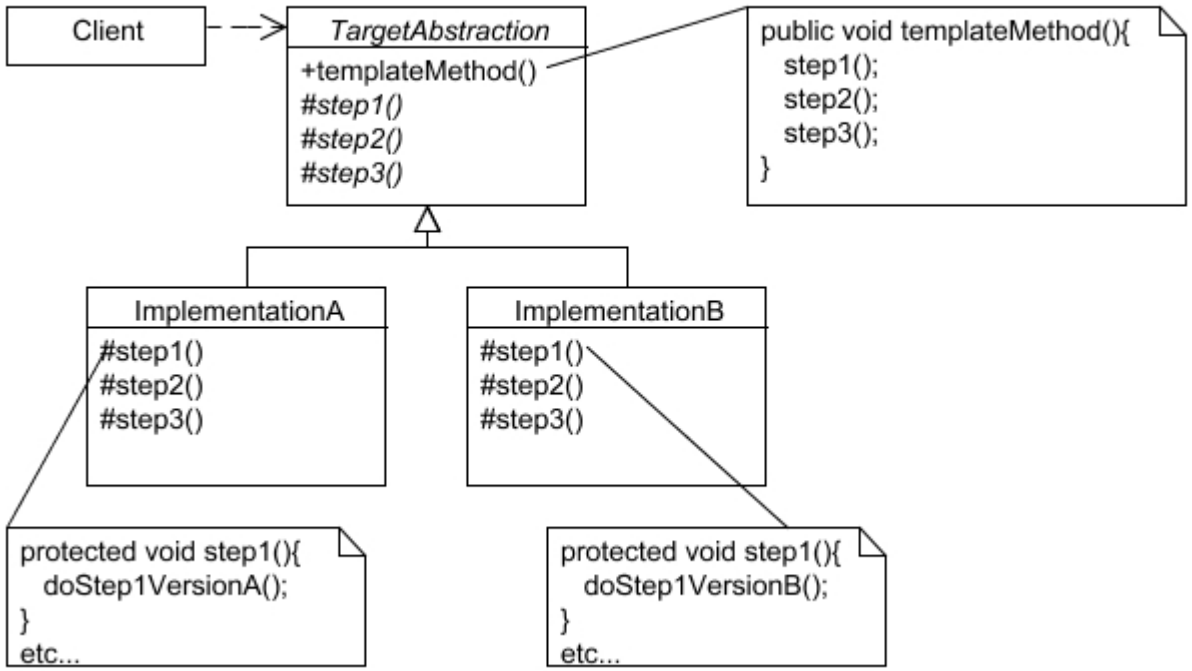
(2) Template Method

Template Method Motivation

- **Problem:** Several clients share the same algorithm but differ on the specifics, i.e., an algorithm consists of customizable parts and invariant parts. **Common steps should not be duplicated in the subclasses but need to be reused.**
- **Examples:**
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types
- **Solution:**
  - The common steps of the algorithm are factored out into an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm.
  - Subclasses provide different realizations for each of these steps.

```
step1();
…
step2();
…
step3();
```

Template Method pattern



Template Method Pattern Applicability

- **Template method** pattern uses inheritance + overridable methods **to vary part of an algorithm**
  - While strategy pattern uses delegation to vary the entire algorithm (interface and ad-hoc polymorphism).
- **Template Method is widely used in frameworks**
  - The framework implements the invariants of the algorithm
  - The client customizations provide specialized steps for the algorithm
  - Principle: “Don’t call us, we’ll call you”.

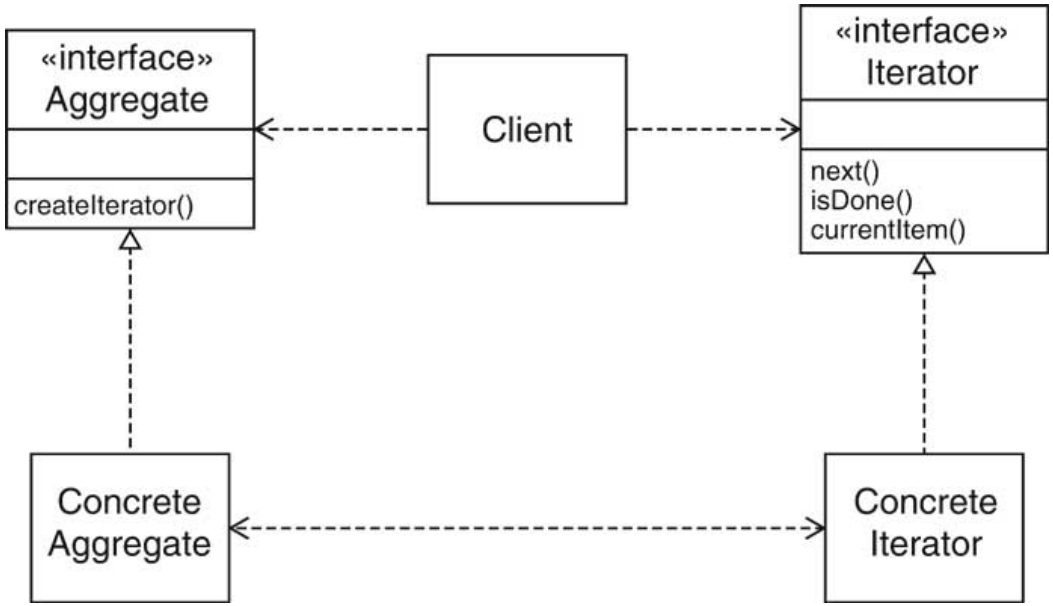
(3) Iterator

Iterator Pattern

- **Problem:** Clients need uniform strategy to access all elements in a container, independent of the container type
- **Solution:** A strategy pattern for iteration
- **Consequences:**
  - Hides internal implementation of underlying container
  - Support multiple traversal strategies with uniform interface
  - Easy to change container type
  - Facilitates communication between parts of the program

Iterator Pattern

- **Pattern structure**
  - Abstract Iterator class defines traversal protocol
  - Concrete Iterator subclasses for each aggregate class
  - Aggregate instance creates instances of Iterator objects
  - Aggregate instance keeps reference to Iterator object



Getting an Iterator

```
public interface Collection<E> {
   boolean add(E e);
   boolean addAll(Collection<? extends E> c);
   boolean remove(Object e);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
   boolean contains(Object e);
   boolean containsAll(Collection<?> c);
   void clear();
   int size();
   boolean isEmpty();
   Iterator<E> iterator();
   Object[] toArray()
   <T> T[] toArray(T[] a);
   …
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.

An example of Iterator pattern

```
public class Pair<E>{
   private final E first, second;
   public Pair(E f, E s) { first = f; second = s; }
   public Iterator<E> iterator() {
      return new PairIterator(this);
   }

   private class PairIterator implements Iterator<E> {
      private boolean seenFirst = false, seenSecond = false;
      public boolean hasNext() { return !seenSecond; }
      public E next() {
         if (!seenFirst) { seenFirst = true; return first; }
         if (!seenSecond) { seenSecond = true; return second; }
             throw new NoSuchElementException();
      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
   }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

The end

March 27, 2019