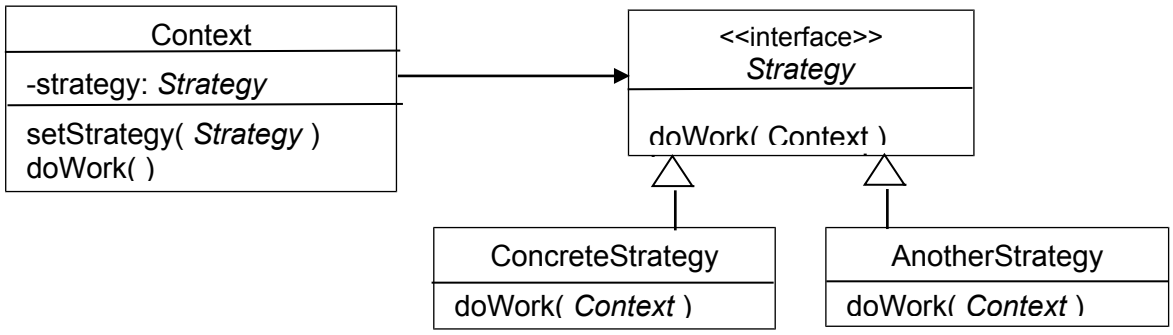
Instructions	1. Define a WithdrawStrategy for withdrawing money from the purse. 2. Write Unit Tests for the WithdrawStrategy. Do this before implementing it. 3. Implement 2 concrete strategies: GreedyWithdraw and RecursiveWithdraw. Use the package **`coinpurse.strategy`** for your code. 4. Modify the Purse to use a WithdrawStrategy. Make GreedyWithdraw be the default (in case the user never calls setWithdrawStrategy).
What to Submit	Add the code to your coinpurse project and push it to Github.

Introduction to Strategy Pattern

Context: An object (called the *Context*) has some behavior that you can implement using several different algorithms, and you'd like to be able to change the algorithm independent of the *Context*.

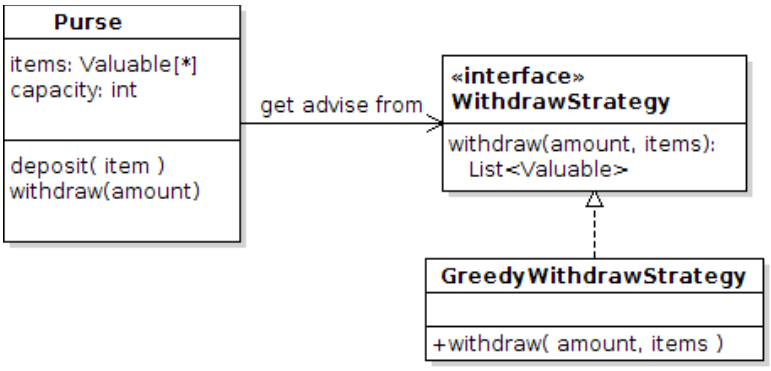
Solution: Design an interface for the method(s) that perform the algorithm (the strategy). This is the *Strategy*. Modify the *Context* class so that it calls a method of the *Strategy* to perform the work, instead of doing the work itself. Then write a concrete implementation of the *Strategy* interface.

In the *Context,* provide a "setStrategy" method so you can specify the strategy at run-time.



Example:

The withdraw( ) method in Purse uses an algorithm to decide which values it should withdraw. There is more than one possible algorithm for choosing what to withdraw, and we might want to change the algorithm. So, we define a WithdrawStrategy for Purse, and write a concrete implementation for each algorithm.

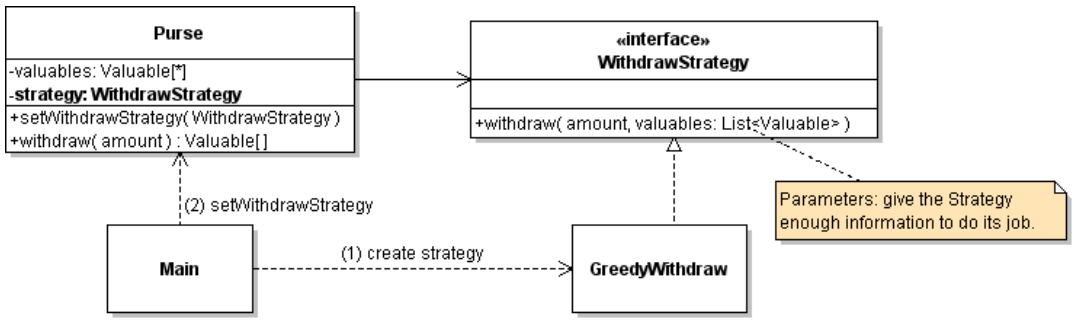


In order for WithdrawStrategy to tell the Purse what to withdraw, the strategy needs a reference to the items in the Purse. Hence withdraw( ) has an extra parameter for the List of items in the purse.

A WithdrawStrategy acts like an advisor. It suggests what to withdraw, but does not actually remove anything from the Purse. The Purse's own withdraw() method should do that -- which also eliminates duplicate code -- only the Purse's withdraw needs code to remove money from its list and create an array.

Problem 1: Define a Withdraw Strategy for Coin Purse

Define a WithdrawStrategy that decides what items the Purse should withdraw. Then we can change the withdraw algorithm anytime without changing the Purse class.



1. Create a new package to hold your strategies, named **`coinpurse.strategy`**.

2. Create a WithdrawStrategy interface that has a withdraw method.

(a) The withdraw method needs two parameters, since it needs to know *what* is in the Purse.

(b) withdraw returns a "recommended" solution, but it does **not change the contents** of the Purse.

3. Write **good, complete Javadoc** for the WithdrawStrategy interface and the withdraw( ) method. The interface Javadoc should clearly tell the developer what the purpose of the interface is, what the withdraw method is expected to do, and how to invoke it. Javadoc should explain:

- what are the *preconditions* for calling withdraw? Does the withdraw method require that valuables (in List) be sorted? Sort in what order?
- what does the method return if withdraw is not possible? Empty list? Null?

```
/**
 * Find and return items from a collection whose total value equals
 * the requested amount.
 * @param amount is the amount of money to withdraw, with currency
 * @param money the contents that are available for possible withdraw.
 *        Must not be null, but may be an empty list.
 *        This list is not modified.
 * @return if a solution is found, return a List containing references
 *        from the money parameter (List) whose sum equals the amount.
 *        If a solution is not found, returns (WHAT?)
 */
public List<Valuable> withdraw(Valuable amount, List<Valuable> money);
```

Problem 2: Write a JUnit Test Suite for WithdrawStrategy

Write a **`WithdrawTest`** class containing JUnit tests for WithdrawStrategy. These tests should test the WithdrawStrategy directly. They don't require a Purse!

2.1 Write at least **10 test methods**, with each method testing a distinct situation. Use descriptive methods names, such as testWithdrawEverything. ZERO credit for names like "testWitthdraw1". Note that one method usually contains more than one "assert" or "assertThat" case.

2.2 Your WithdrawTest should be able to test *any* WithdrawStrategy class. Don't make assumptions about the code or try to access anything not in the WithdrawStrategy interface.

To enable your test suite to use different WithdrawStrategy classes, create the concrete strategy object in a JUnit "setUp" method. This is a special method that is run before each test, so you don't have repeat code in each test. Your can use any method name, but it must have annotation `@Before`.

```
import org.junit.Before;

public class WithdrawTest {
    private WithdrawStrategy strategy;
    /**
     * Code to run before each test. Setup the "test fixture".
     */
    @Before
    public void setUp( ) {
        strategy = new NeverWithdrawStrategy( );
        // other initialization code for your tests
    }
```

2.3 Write some "*borderline*" test cases. For example: withdraw everything from Purse; withdraw everything *except* one item; withdraw slightly *more* than the total value (should fail); withdraw amount where more than one solution is possible.

2.4 Test some simple cases to find obvious errors. For example, a List contains 5-Baht coin and 5-Ringgit coin. What happens if you withdraw 5 Baht or 10 Baht? What about 5 Rupees? This tests for code not checking the currency carefully.

2.5 Write test cases where greedy withdraw succeeds, some where it fails (but withdraw is possible using a different algorithm), and some cases where withdraw is clearly impossible.

2.6 Verify that WithdrawStrategy does not change the items in the Purse. (To do this, you might make a copy of the List of money, and compare with the original List.)

2.7 Verify that the return value from WithdrawStrategy is correct (of course): a) amount is correct, b) the returned value contains items from the purse (not creating new money).

For computations that you do repeatedly, write a separate method.

For example, you need to prepare the List of money items that you will pass as parameter.

Problem 3: Apply Test-Driven-Development to Create GreedyWithdraw class

Write a GreedyWithdraw that implements WithdrawStrategy. Apply Test-Driven-Development (TDD), which means you write code and continuously test it, until it passes your unit tests. Initially GreedyWithdraw doesn't do anything:

```
package coinpurse.strategy;
import java.util.List;
import coinpurse.Valuable;

/**
 * TODO Write good Javadoc.
 */
public class GreedyWithdraw implements WithdrawStrategy {
    public List<Valuable> withdraw(Valuable amount,
                        List<Valuable> items) {
        // whatever your strategy returns for "can't withdraw".
        // May be empty list or null.
        return null;
    }
}
```

3.1 Run the JUnit tests. Tests will mostly fail, of course. A few tests may succeed (if the expected result is "can't withdraw") by accident.

3.2 Copy (or move) the code for withdraw from the Purse class to GreedyWithdraw.withdraw(). Only move the code that decides what to withdraw.

In Purse.withdraw() there should be some code at the beginning (validate withdraw amount) and end (actually remove items from Purse and create an array) which is not part of the WithdrawStrategy, so leave that code in the Purse.withdraw() method. That is code that is always needed.

3.3 Run the JUnit tests again. Some more tests should pass. Repeat it until the code passes all tests, except those where GreedyWithdraw does not work.

Problem 4: Modify Purse to Use WithdrawStrategy

4.1 In the Purse, add an *attribute* for WithdrawStrategy and write a setWithdrawStrategy( ) method to set it. To make sure that the attribute is not null, use the constructor to assign it to a GreedyWithdraw object.

```
/** The strategy for for withdrawing items. */
private WithdrawStrategy strategy;
public Purse(int capacity) {
    // set strategy = new GreedyWithdraw()
}
//TODO write setWithdrawStrategy()
```

4.2 Modify withdraw( ) so that it calls WithdrawStrategy to decide what money items to withdraw. There is still some code needed at the beginning and end of the method.

```
public Valuable[ ] withdraw(Valuable amount) {
    // 1. verify that amount is valid
    // 2. call withdrawStrategy(amount, items ) to decide what to withdraw
    // 3. remove items from purse and return them as an array.
    //    Or return null if withdraw fails.
}
```

Use the principle:

> "*Separate the part that varies from the part that stays the same. Encapsulate the part that varies.*"

The code that is always needed for withdraw (like putting items into an array) stays in the Purse withdraw method. The part that depends on the *algorithm* for withdraw goes in the WithdrawStrategy.

4.3. Test the Purse to verify withdraw works the same as before.

Problem 5: Implement a Recursive Withdraw Strategy

The greedy withdraw strategy can fail, even when a withdraw is possible. For example:

insert: 5 Baht, 2 Baht, 2 Baht, 2 Baht

withdraw: 6 Baht

Write a RecursiveWithdraw strategy to find a solution using *recursion*. The algorithm is similar to the **`groupSum`** problem on **`codingbat.com`**.

5.1 Write a RecursiveWithdraw strategy class in the coinpurse.strategy package.

5.2 Implement the recursive withdraw using TDD. Keep testing and coding until all tests pass.

5.3 When your code passes all tests, take a short break and then perform *Code Review*. Read your RecursiveWithdraw code line-by-line and explain it to yourself or another student. *Code Review* often finds bugs that testing misses. If you find any "missed" bugs, create a new JUnit test that would detect it.

Programming Hints

1. At each step of recursion, select *the first item* (or last item) in the money List and consider 2 cases:

Case 1: Choose this item for withdraw. By recursion, try to withdraw the *remaining* amount = amount - value of this item, using only the remaining items in the list.

Case 2: *Don't* use this item for withdraw. By recursion, try to withdraw the *entire* amount using only the remaining items in the list.

2. For the recursive step, create a *sublist* of the current list that excludes element 0.

For example:

```
// select the first item in the list, for possible withdraw
Valuable first = money.get(0);
// Case 1: select this item for withdraw -- currency must match,
// and try to withdraw the remaining amount using rest of the list.
// Some code is missing (you don't want it to be too easy, do you?)

Valuable remaining = new Money(amount.getValue()-first.getValue(),
                               currency);
List<Valuable> result = withdraw( remaining,
                                  money.subList(1,money.size()) );
// test if the recursion succeeded.
// Case 2: don't use this item for withdraw.
// Try to withdraw amount using the rest of the list.
```

`list.subList( start, end )` creates a *view* of the list starting at index `start`, and up to (but not including) index `end`. It is a *view,* not a copy. This is efficient (no copying of the List).

3. During recursion, at some point it will either fail or succeed. If it succeeds, create a new list for the return result and add whatever element you want to return. Higher level callers will just *append* their item on this return list -- **don't create a new list at each level**!

Does Recursion Eventually Stop?

You should ensure that recursion eventually stops.

How to See What RecursiveWithdraw is doing?

The withdraw strategy object doesn't print anything, so it is hard to debug.

You could add code at the beginning and end (before every return) to print something. To keep you code clean, write methods named enter() and leave() to print whatever is useful to you. For example:

```
public List<Valuable> withdraw(Valuable amount, List<Valuable> money) {
    enter(amount, money);
    // do something
    List<Valuable> result = ...
    // call leave before returning.  leave() will return its own parameter
    return leave(result);
}
//TODO: write enter() and leave() to print whatever you like.
```

Reference

Big Java chapter 13 covers Recursion.

codingbat.com "Recursion-2" problem set covers recursion with backtracking. The `groupSum` problem is exactly like this one.

Example of Recursive Withdraw:

Here is an example recursive withdraw using a list of numbers.

withdraw(amount, list) - try to withdraw amount from list of Number. Returns a List of elements to withdraw.

Example: Recursive withdraw for amount = 4, list = {1, 2, 2, 5},