Instructions	Modfy the Purse to use a WithdrawStrategy for deciding which items to withdraw from the Purse. The Purse's own withdraw method should use the WithdrawStrategy object to decide ("advise") what to withdraw. The Purse's own withdraw method then removes the items from its money list and returns them as an array (same as your original code). Steps are described in detail below.
What to Submit	Add the code to your coinpurse project and push it to Github. The WithdrawStrategy interface and implementation should be in package coinpurse.strategy.

Introduction to the Strategy Pattern

The Strategy Pattern is useful in an application that uses an *algorithm* to perform some *task*, and there is more than one algorithm that can be used. You would like a way to select which algorithm to use, or even add new algorithms later (without changing the original code).

In the Coin Purse, the withdraw method uses an algorithm to decide what items to withdraw, and there is more than one algorithm we could use. That's a good place to apply the Strategy Pattern.

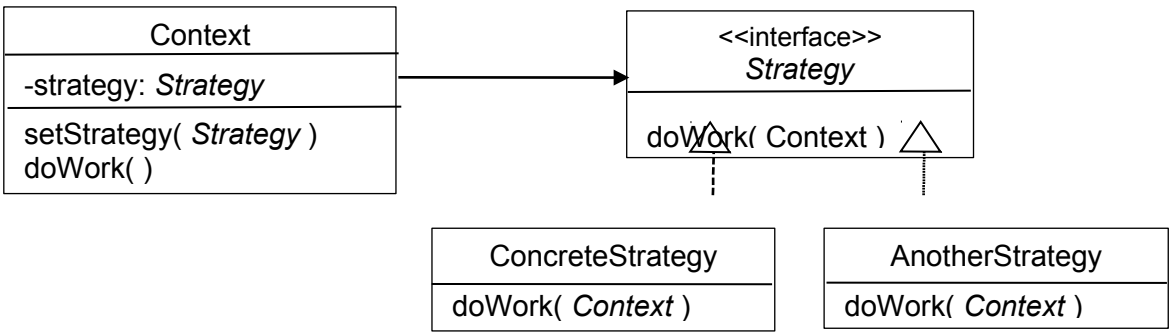
Here's a description of how to use Strategy:

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 performs the algorithm. This is the *Strategy*. Modify the *Context* class so that it calls a method of the *Strategy* to perform the task, instead of doing the task itself. Then write a concrete implementation of the *Strategy* interface.

Notice that the *Strategy* needs a reference to some data from the *Context* so it can perform the task for the Context. In the Coin Purse, the Strategy needs to know what items are in the Purse in order to decide what the Purse should withdraw. So, in the *Strategy* interface, the method(s) need a parameter that refers to the *Context* or some data from the *Context (e.g. money in the Purse)*.

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



Coin Purse Withdraw Method

The withdraw() method in Purse uses an algorithm to decide which items it should withdraw.

There is more than one algorithm for this, and we might want to change which algorithm we use.

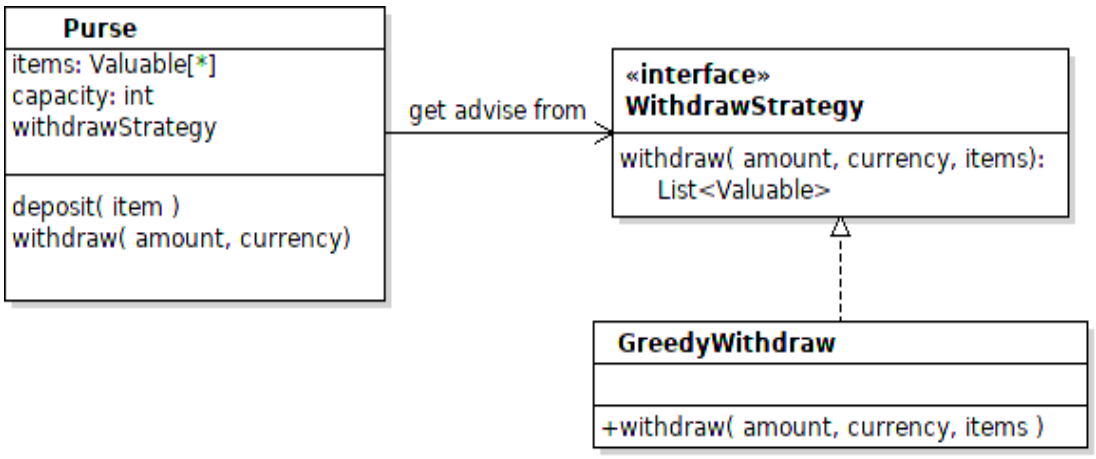
So, define a WithdrawStrategy for Purse, and modify the Purse code to use WithdrawStrategy.

The WithdrawStrategy decides which items to withdraw, but does not modify the contents of the Purse. The strategy only "recommends" what the Purse should do, and let's the Purse perform the actual withdraw. That's good encapsulation and separation of responsibility. Purse is responsible for managing money in the Purse.

In order for WithdrawStrategy to "recommend" what to withdraw, the WithdrawStrategy needs to know what money is in the Purse and how much we want to withdraw.

So, it needs parameters for (a) items in the purse, (b) how much to withdraw, (c) the currency.

WithdrawStrategy returns a *List* of items to withdraw (not an array).



Problem 1: Define a Withdraw Strategy for Coin Purse and Implement It

Here are steps to implement this:

1. Write some unit tests for the current withdraw method (if you don't have them already!). This is to verify that it works before and after you change the code. JUnit Purse tests were given in an earlier lab. Those are a good start. Does your Purse pass all the tests?

2. Create a new package to hold withdraw strategies named **`coinpurse.strategy`**.

3. Define a **`WithdrawStrategy`** interface for withdrawing money from the purse. It has only 1 method, as shown in the UML diagram.

Write good Javadoc for WithdrawStrategy! An interface is a *specification of some behavior*, so it needs clear, complete **documentation** that tells others how to implement the behavior. Javadoc should explain:

- what do the parameters mean? Can WithdrawStrategy modify the parameters?
- what does it return? What does it return if can't withdraw the requested amount?
- what are the *preconditions* for calling withdraw? Does the withdraw method require that valuables (in List) be sorted? Can the List of valuables be empty? Can the amount parameter be zero?

4. Create a concrete class named **`GreedyWithdraw`** that implements WithdrawStrategy. Copy the code from your Purse.withdraw method into GreedyWithdraw -- except the last part that removes items from the Purse and returns an array. That part is still performed by the Purse's withdraw() code.

- GreedyWithdraw does not need any attributes! All the code is in the withdraw() method.

5. Add a WithdrawStrategy attribute to the Purse, and initialize it in the Purse constructor to a GreedyWithdraw object. That's the strategy your Purse will use.

6. Modify the Purse withdraw method to call withdrawStrategy.withdraw(). If the WithdrawStrategy returns a non-null result (success) then use the list to withdraw money from the purse and return an array (same as original code).

7. Test the modified code. It should work the same as the original code.

```
/**
 * Select and return items from a collection whose total value equals
 * the requested amount and having the given currency.
 *
 * @param amount is the amount of money to withdraw.
 * @param currency is the currency to use for withdraw. Must not be null
 * @param items the contents that are available for 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 List whose sum equals the amount.
 *        If a solution is not found, returns ????
 */
public List<Valuable> withdraw(double amount, String currency,
                               List<Valuable> items);
```

Problem 2: Modify Purse so we Can Set the Withdraw Strategy

To make the WithdrawStrategy useful, we need a way to change which strategy the Purse uses -- without modifying the Purse code.

Add a setWithdrawStrategy() and getWithdrawStrategy() method to the Purse.

For Hackers: to verify that setWithdrawStrategy() is working, define another WithdrawStrategy that does something different. Be creative. For example, a NoWithdraw strategy that never withdraws anything!

```
public class NoWithdraw implements WithdrawStrategy {
    public List<Valuable> withdraw(...) {
        // Hoard everything! Never withdraw.
        return null;
    }
}
// In the Main class:
purse.setWithdrawStrategy( new NoWithdraw() );
```

Design Principle

Do you remember this design principle from the StopWatch lab?

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

It the StopWatch lab, the part that varies are the tasks that we want to compute the run time. The part that stays the same is the timeAndPrint code.

In the Purse, what is the part (of code) that varies? How are we encapsulating it?