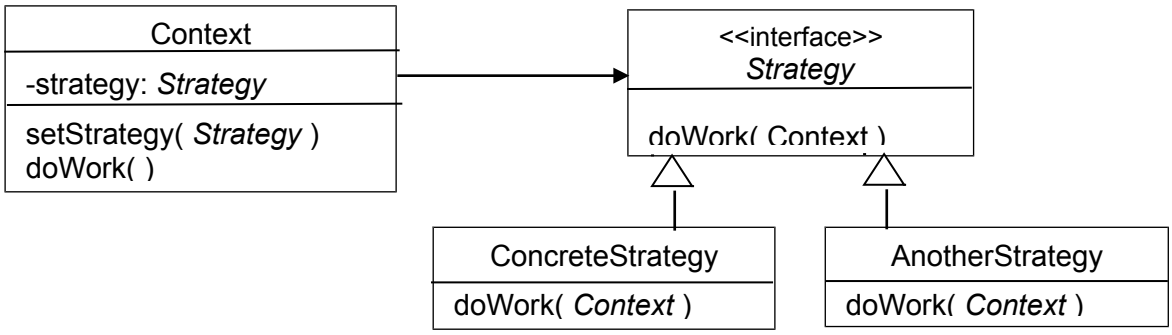
Instructions	Define a WithdrawStrategy for withdrawing money from the purse, and implement 2 concrete strategies: GreedyWithdraw and RecursiveWithdraw. Modify the Purse to use the WithdrawStrategy, and make GreedyWithdraw be the default (in case the user never calls setWithdrawStrategy). Use the package **`coinpurse.strategy`** for your code.
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 object that *uses* the behavior.

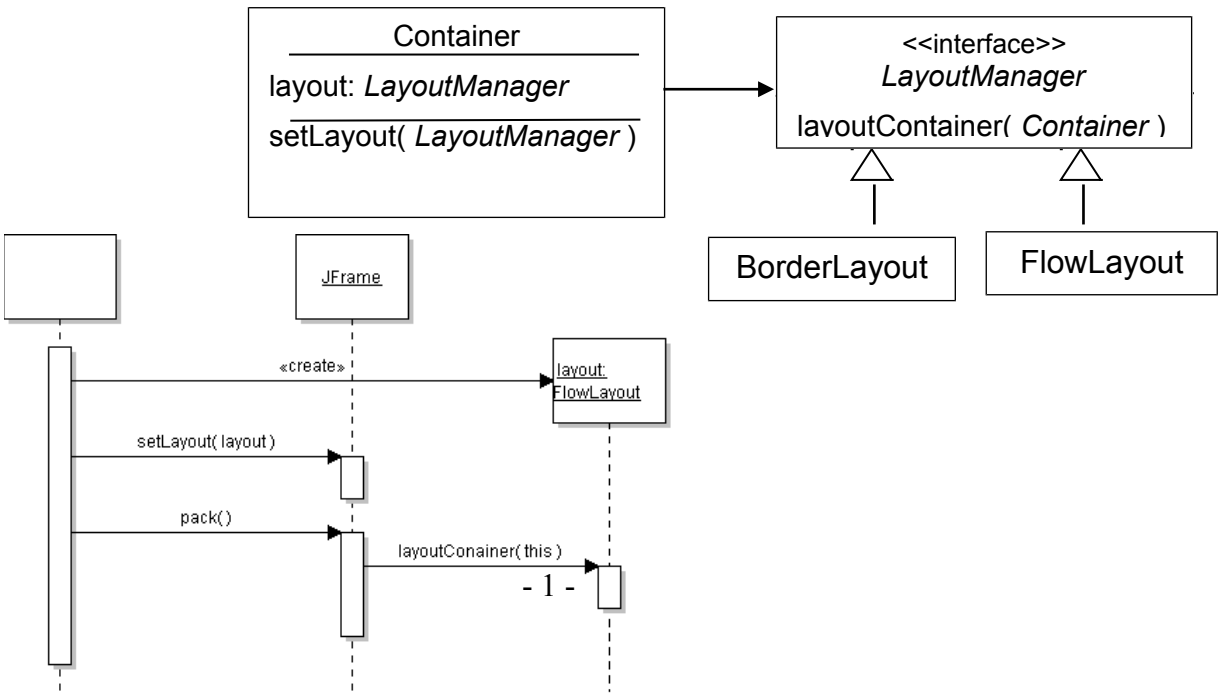
Solution: Design an interface for the method(s) that perform the algorithm (the strategy). This is the *Strategy*. Modify the *Context* class so that it calls a method of the *Strategy* to perform the work, instead of doing the work itself. Then write a concrete implementation of the *Strategy* interface.

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



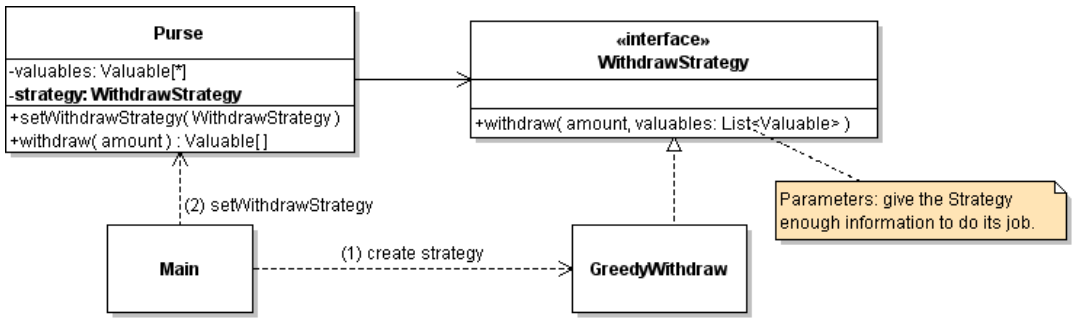
Example: The Swing containers (JFrame, JPanel, JWindow) need to layout components in the container. There are many ways (*algorithms*) to layout components, and we want to be able to *change* the way we layout a container. We also want to *reuse* the same layout algorithm in different containers without duplicating the code. If layout code is written as part of the Container class, we cannot change the layout and cannot reuse the layout code.

The solution is for containers use the Strategy Pattern. The *LayoutManager* interface is the Strategy; it defines the methods needed to lay out components in a container. Each Container has a *setLayout*( ) method and a reference to an instance of a concrete *LayoutManager*. When a container needs to layout its components it calls layout.layoutContainer(this).



Problem 1: Define a Withdraw Strategy for Coin Purse

Define a WithdrawStrategy to perform 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) Notice that the withdraw method needs two parameters: the amount and the money in the Purse.

(b) withdraw returns a List of its "recommended" solution, but it does not change the contents of the Purse. Let the Purse withdraw money itself.

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

```
/**
 * Find and return a collection of money that will enable the purse to
 * withdraw the requested amount.
 * @param amount is the amount of money to withdraw
 * @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(double amount, List<Valuable> money);
```

3b. Does your strategy *require* that the money be sorted? In what order (smallest to largest)? You must document this in the Javadoc!

4. Create a concrete class that *implements* WithdrawStrategy. The strategy that we have used in the Purse is the *greedy algorithm*, so a good name for the strategy class is **`GreedyWithdraw`** .

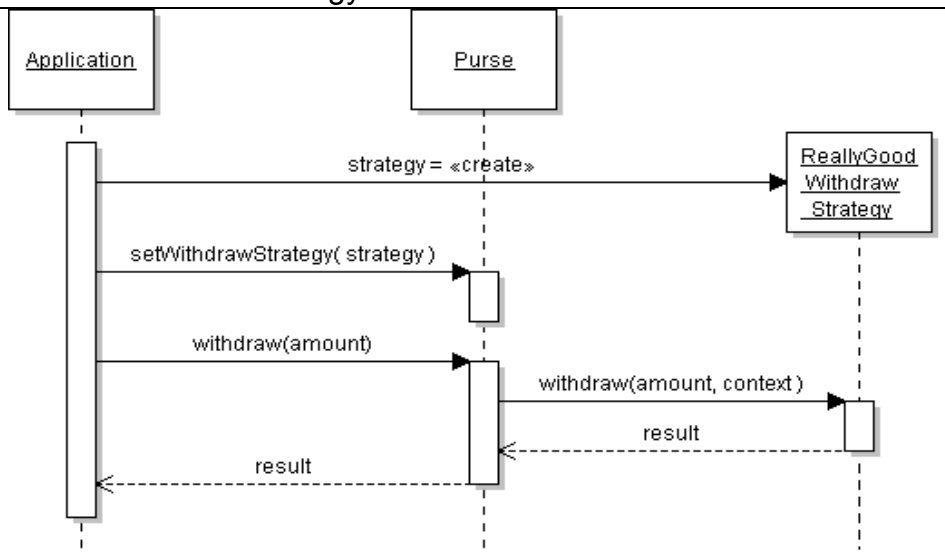
5. Move *most* of the code from **`Purse.withdraw`** into the concrete **`GreedyWithdraw`** class, and instead invoke `strategy.withdraw(amount, valuables)`.

The WithdrawStrategy *suggests a solution*, but the Purse is responsible for actually removing the money and returning it as an array. (This also avoids duplicate code.)

Use the principle: "*Separate the part that varies from the part that stays the same. Then encapsulate the part that varies.*"

For example, Purse.withdraw can verify amount <= balance before calling the strategy, and Purse.withdraw is responsible for actually removing the money from its valuables list.

6. Test the Purse to verify it works the same as before.



Problem 2: Write a JUnit Test for Withdraw

Write test cases where greedy withdraw succeeds, some where it fails (but should be possible), and some cases where withdraw is impossible.

Problem 3: Implement a Recursive Withdraw Strategy

The greedy withdraw strategy (problem 1) can fail, even when a withdraw is possible. For example:

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

withdraw: 6 Baht

You can fix this using recursion and trying all possibilities. Recursion requires more time and memory, but a coin purse doesn't usually have many items and you can quickly eliminate some possibilities.

Example: if you need to withdraw 12 Baht you can skip all items with value > 12 Baht.

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

3.2 Implement the recursive withdraw and test that it works.

3.4 Construct an example (in a separate class, using a main method) where GreedyWithdraw fails but RecursiveWithdraw succeeds. **Demonstrate this to the TAs.** You can use JUnit if you like.

Programming Hints:

1. At each level of recursion, select *the first 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:

```
// save reference to first item in list, in case withdraw succeeds
```

```
Valuable first = money.get(0);
List<Valuable> result = withdraw( amount - first.getValue(),
                          money.subList(1,money.size()) );
```

`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 values).

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.

In the lab, I'll show you howto define a *decorator* for WithdrawStrategy to print each method call and each return. You could also use the Eclipse debug feature for this.

A *decorator* is a class that looks like another class but adds some additional functionality. A decorator encapsulates another object (from the base class) and invokes methods of the encapsulated object for most functions, but "decorates" some part.

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},