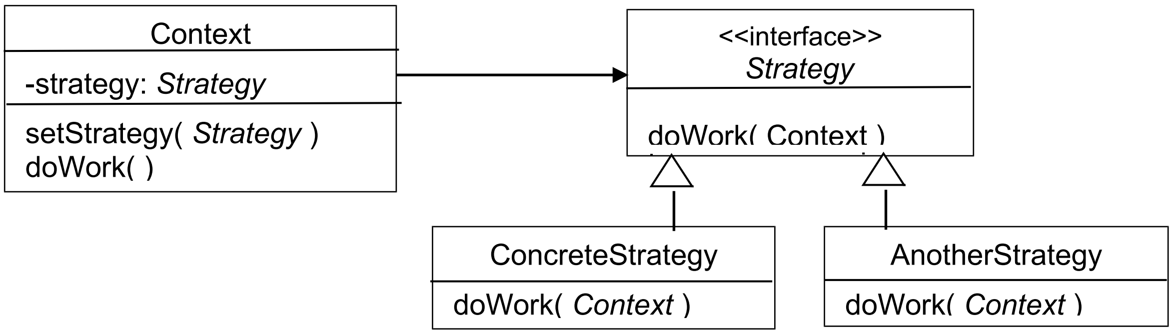
What to Submit	Commit the Coin Purse with strategies in package coinpurse.stategy to Bitbucket. Use same project as your previous labs, including a TAG for each lab (Lab2, Lab3, …).

Introduction to the 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.

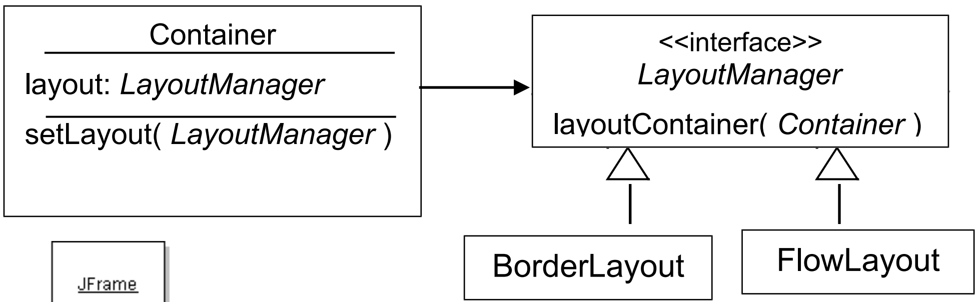
Solution: Design an interface for the method(s) that perform the algorithm (the strategy). This is the *Strategy*. Modify the *Context* class so that it calls a method of the *Strategy* to perform the work, instead of doing the work itself. Then write a concrete implementation of the *Strategy* interface.

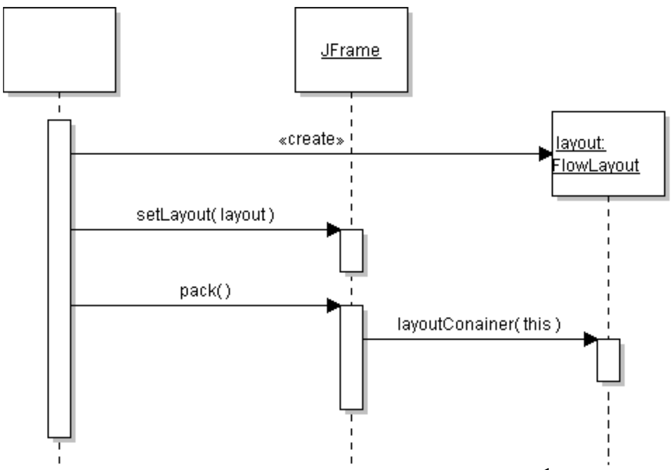
In the *Context,* provide a "setStrategy" method so you can specify the strategy at run-time.



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).

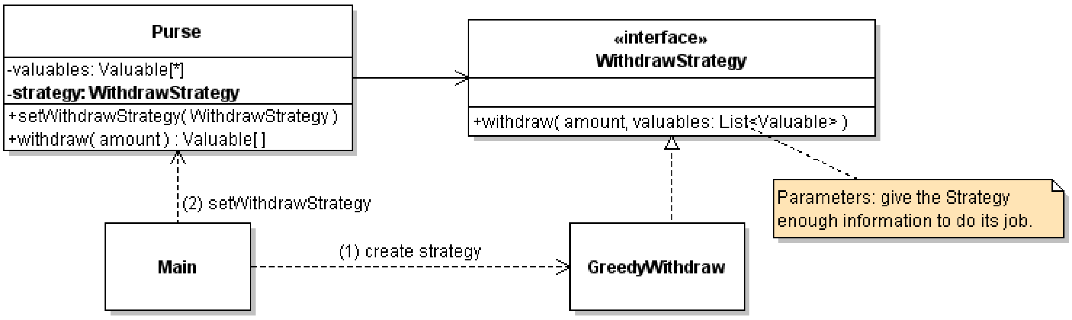




Problem 1: Define a Withdraw Strategy for Coin Purse

In the Purse, the most complex method is withdraw. The *easiest* way to withdraw is the greedy algorithm (as in previous labs) but it doesn't always work. Using *recursion* would always find a solution (if a solution exists), but recursion takes more time.

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. Notice that the withdraw method has an extra parameter(s). The Purse must give the strategy enough information to do its job.

3. Write good Javadoc for the withdraw( ) method of WithdrawStrategy. What *exactly* does it <u>do</u> and what will it <u>*return*</u>?

Does the WithdrawStrategy actually remove items from the List (Valuables in the Purse) or simply *suggest* what the purse should withdraw? Your Javadoc should clearly state this.

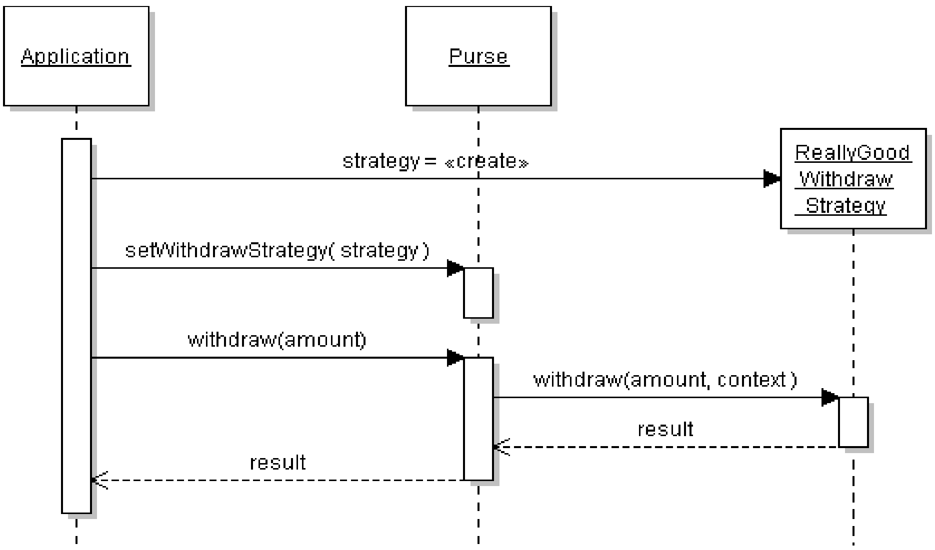
4. Create a concrete class that *implements* WithdrawStrategy. For the strategy that we have used before, a good name is **`GreedyWithdraw`** (our withdraw uses the greedy algorithm).

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

Note that some code should stay in `Purse.withdraw`. Use the guide: "*Separate the part that varies from the part that stays the same. Then encapsulate the part that varies.*" (as in PA1)

For example, Purse.withdraw can verify amount <= balance, and Purse.withdraw is responsible for actually removing the items from its Valuables list.

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



Problem 2: Recursion Practice

Recursion means that a method calls itself. Java (and most languages) allow this, but you have to be careful how you use it. You must ensure that *recursion* will eventually stop. There are 2 criteria that generally ensure recursion will stop:

- each time the method calls itself (recursion) the problem is made smaller
- the recursive method always checks a *stopping criterion* which will eventually be true

2.1 Sum elements in an array by recursion

To sum elements in an array using a loop (*iteration*) we would write:

```
public double sum(double [] x) {
   double sum = 0;
   for(int k=0; k < x.length; k++ ) sum = sum + x[k];
   return sum;
}
```

To sum by recursion, create a second method, called a ***helper method***, with an extra parameter to make it easier to use recursion. The parameter is the last array index we want to sum:

```
/**
  * Sum all elements of an array up to lastIndex.
  * @param x  array of elements to sum.
  * @param lastIndex  is index of the last element to sum.
  * The caller must ensure that lastIndex < array.length
  */
private double sumTo( double [] x, int lastIndex)
   if (lastIndex  < 0) return 0.0; // (1) the stopping criterion
   double sum = x[lastIndex]
        + sumTo(x, lastIndex - 1); // (2) recursive step
   return sum;
}
```

In this code:

(2) *Recursive Step:* the method calls itself, but it makes **`lastIndex`** smaller each time, so the problem is getting smaller.

(1) checks a stopping criterion, so the method won't call itself forever. Since **`lastIndex`** is decreased by 1 at each step, this test must eventually be true.

`private` - sumTo is a *helper method* for use by the public `sum( )` method, so we declare it private.

To use the recursive sum, the public **`sum`** method calls the *helper method*:

```
public double sum(double [] array) {
   return sumTo( array, array.length - 1);
}
```

2.2 Write a Recursive "print" for a List

Create a class named **`ListUtil`** and write a static method **`printList(List<?> list)`** to print all the elements in a **`List`** on one line to System.out, with a comma between elements but **no comma after the last element**. Use recursion (not a loop). Example using BlueJ Codepad:

```
> import java.util.Arrays;
> List food = Arrays.asList( "apple", "banana", "grape", "fig" );
> ListUtil.printList( food );
// output appears in console window:
 apple, banana, grape, fig
```

2.3 Write a Recursive "max" for a List of Strings

Write a method that returns the "max" element of a *List* of *String* objects, using recursion. For easy testing, make this method static. Design your own *helper method*.

```
public class ListUtil {
    private static void printList(List<?> list) {
    //TODO Problem 2.2
    }

    /**
     * Find the largest element in a List of Strings,
     * using the String compareTo method.
     * @return the lexically largest element in the List
     */
    private static String max( List<String> list) {
    //TODO complete this
    }

    /** Test the max method. */
    public static void main(String [] args) {
        List<String> list;
        // if any command line args, then use them as the list!
        if (args.length > 0) list = Arrays.asList( args );
        else list = Arrays.asList("bird", "zebra", "cat", "pig");

        System.out.print("List contains: ");
        printList( list );

        String max = max(list);
        System.out.println("Lexically greatest element is "+max);
    }
}
```

Problem 3: Implement a Recursive Withdraw Strategy

The greedy withdraw strategy (problem 1) can fail, even when a withdraw is possible. Can you give an example of this?

You can fix this using recursion. Recursion requires more time and memory, but a coin purse doesn't usually have many items and you can quickly eliminate some possibilities (for example, if we need to withdraw 12 Baht, we can skip over items with a value greater than 12).

At each level of recursion you pick one item from the Purse and try two cases:

Case 1: Select this item for withdraw and (by recursion) try to withdraw the *remaining* amount = amount - value of this item, using only the remaining items.

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

At each step of the recursion, you don't want to modify the list or create a new list, because that uses more memory and time. Instead, follow the example of Problem 2.1 and write a *helper method* with a parameter to indicate the last element (or first element) in the list that can be used as candidates for withdraw.

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

3.2 In the Purse, add a setWithdrawStrategy method (if you didn't already).

3.3 In the Main class, set the strategy to RecursiveWithdraw. Test your code.

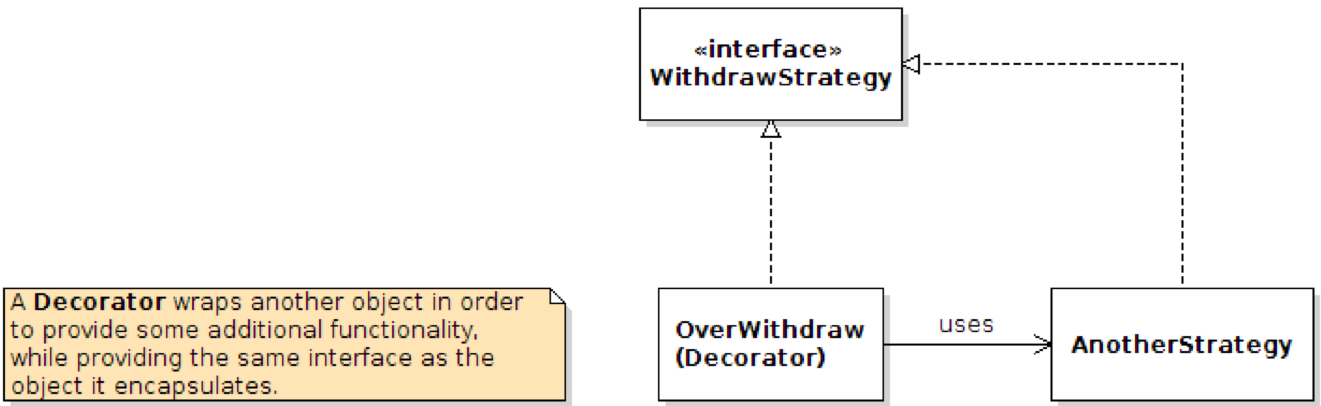
3.4 Construct an example (in the Main class) where GreedyWithdraw fails but RecursiveWithdraw succeeds.

Challenge Problem: Create an Over-withdraw Strategy (not required)

What if lunch costs 15 Baht but your purse only has 20-Baht banknotes? Would you rather go hungry or *over*-withdraw 20-Baht?

Write an Overwithdraw strategy that withdraws the smallest amount *greater than or equal to* the requested amount.

One simple way to implement this is as a Composite or Decorator strategy: Overwithdraw is a Withdraw Strategy that calls some other strategy (like RecursiveWithdraw) with increasing amounts, until it finds one that succeeds.

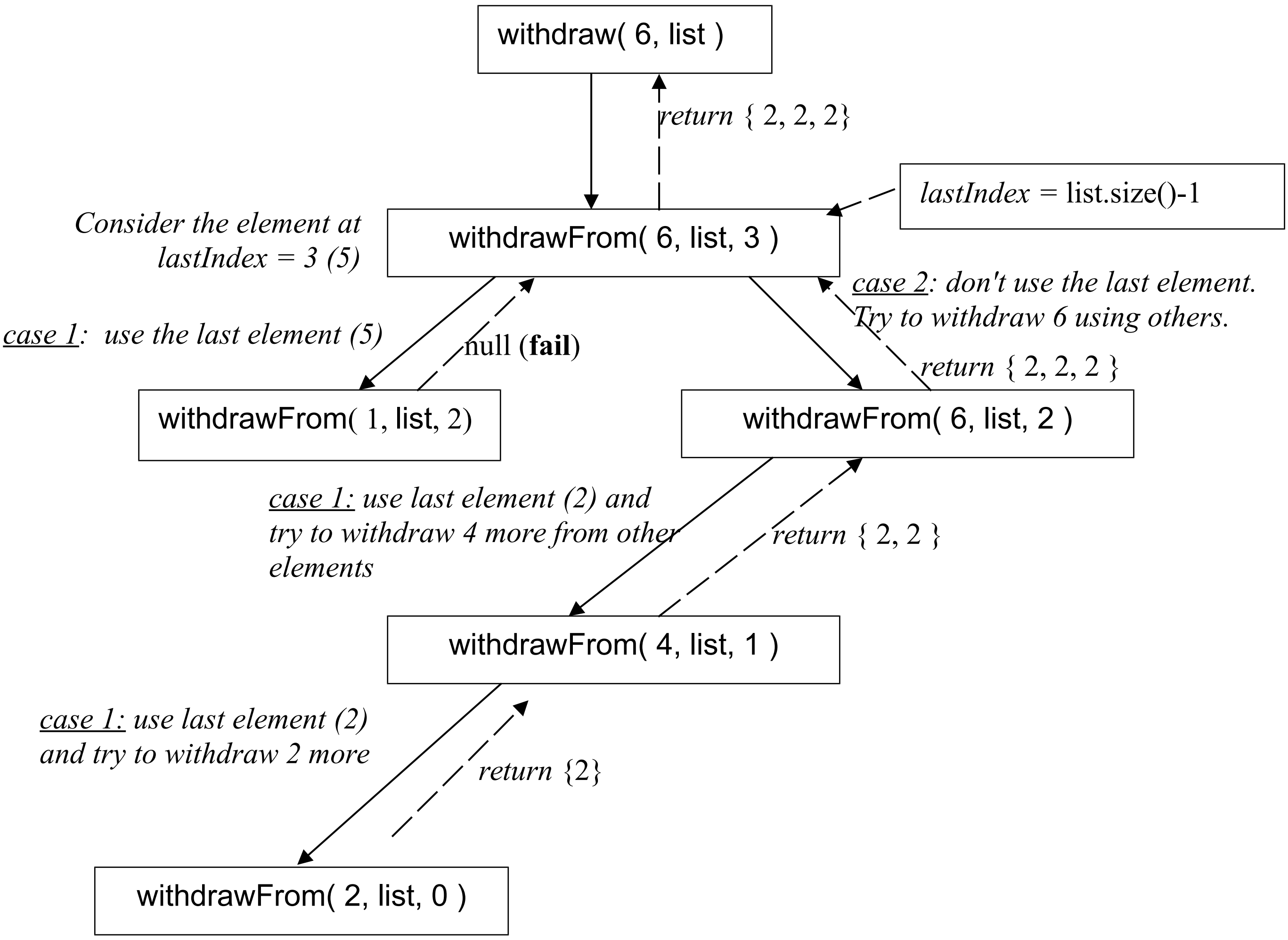


Example:

Here is an example recursive withdraw method that uses a *helper method*. The example uses a list of integers, for simplicity. Your method should use Valuable, of course.

withdrawFrom(amount, list, lastIndex) - try to withdraw amount from list of Integer, using only elements 0 through lastIndex. Returns a List of elements to withdraw. **Does not** modify the list parameter.

Example: Recursive withdraw for list = {2, 2, 2, 5}, amount to withdraw = 6



Try to withdraw 2 using only element 0 of the list.
Fortunately, 2 exactly equals the value of list.get(0), so we return that element

Does Recursion Eventually Stop?

In this example, the `lastIndex` of the helper method is decreased at each step of recursion, so eventually there won't be any candidate elements (`lastIndex` < 0). And at each step there are only 2 choices: use the last list element or don't use it.

Reference

Big Java chapter 13 covers Recursion.