# Module 7 Labs

These labs are centered on the topic of doing refactoring with respect to two different situations covered in the lectures.

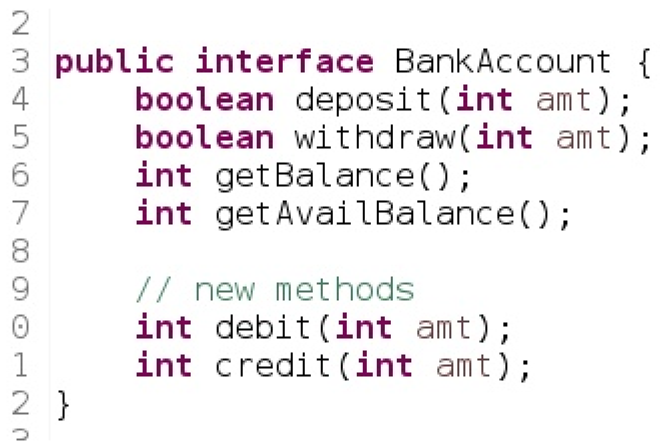
## *Lab 7-1: Refactoring a Requirements Change*

For this lab, to ensure that everyone is starting with the same implementation of the problem, use the starter workspace lab7-1_starter. The application is in the state that it should have been at the end of module 5 with the addition of the two new methods in the interface and stubs for those methods in MyAct class. However, you should feel free to use the AssertJ matchers as you work through this and the other labs, although they will not be used in the solutions provided.

The first refactoring comes from a change in requirements that will force a change in the actual interface, in this case adding two new interface methods. Recall that the only two things that should cause a change in the test cases are a change in requirements or change in the design or architecture of the application.

### *Part 1: Specification revision*

It has been decided that the bank account application that we are creating is going to be shared by an internal system as well as a customer facing system. The new system accesses the bank accounts using a couple of different methods then the existing ones:



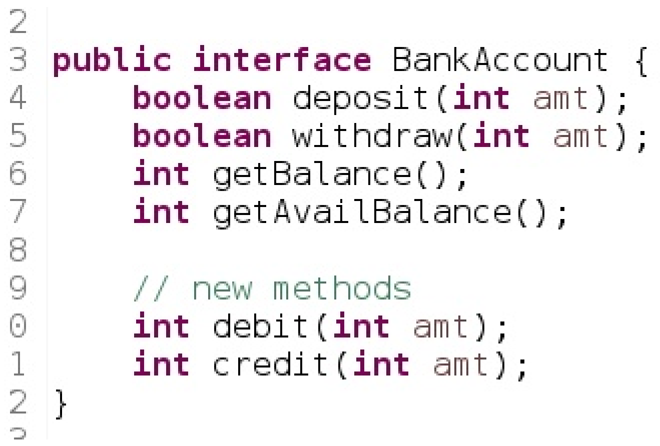
```
public interface BankAccount {
    boolean deposit(int amt);
    boolean withdraw(int amt);
    int getBalance();
    int getAvailBalance();

    // new methods
    int debit(int amt);
    int credit(int amt);
}
```

The getBalance() and getAvailBalance() methods work exactly the same as the ones in the existing interface. The debit() method works exactly the same as the withdraw() method in our existing interface and the the credit() method works exactly the same as the deposit() method. There is no change in the functionality of the underlying application, only a change in how the client wants to request the functionality.

There are two ways to implement this change, one that requires significantly more work than the other. The harder way, which is also the one that a lot of programmers just jump to right away, is to add the two new debit() and credit() methods to the interface and then to cut and paste the existing code from the deposit() and withdraw() methods.

But if we do this, there is a code smell – duplicated code. The easier refactoring approach is to create a new private method with the common code and call the private method from each of the public interface methods that require the same functionality. Refactoring from duplicate code in two public methods takes a lot more work because we are actually be writing the production code twice in different places, then deleting what some of what we have just written. This constant cutting, pasting and deleting introduces the risk of adding incidental errors during the process.

The refactoring approach this lab will take is more efficient in terms of involving less work for us and reducing the risk of accidental errors. What we will do is refactor the existing code then add the new interface methods.



```java
public interface BankAccount {
    boolean deposit(int amt);
    boolean withdraw(int amt);
    int getBalance();
    int getAvailBalance();

    // new methods
    int debit(int amt);
    int credit(int amt);
}
```

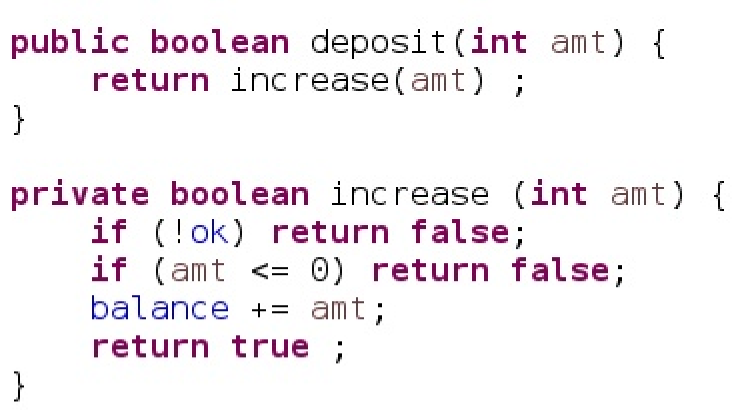
### *Part 2: Refactoring the deposit() method.*

Refactorings are done by making some changes, then running all of our tests to ensure that we didn't introduce any bugs. In the solutions for this lab, the state of the code at each point we run the tests is provided so you can compare how you are doing with a sample solution.

1. Add a new private method

   ```
   private boolean increase(int) {}
   ```

   or whatever else you might want to call it. Since it is a private method, it doesn't really matter what the name is we choose.



```java
public boolean deposit(int amt) {
    return increase(amt) ;
}

private boolean increase (int amt) {
    if (!ok) return false;
    if (amt <= 0) return false;
    balance += amt;
    return true ;
}
```

2. Move the code from the deposit method() to the increase() method.
3. In the body of the deposit() method, add a method call to the private increase() method.

4. Run all of the tests. If you moved the code correctly, all the tests should pass. If they don't, then recheck your work and debug based on the failing test.

Notice that we didn't change the existing interface. Any client that called deposit() before will still call deposit() and get the same response as before. We did not add any code that is not being exercised by the existing unit tests.

## *Part 3: Refactoring the withdraw() method*

Repeat the same process as for deposit() but this time use a private method called decrease(). Run the tests and if they all pass, then you have finished the refactoring.

The solution up to this point in the lab is in the solution workspace lab7-1_solution_step1 in the solutions folder for Module07.

## *Part 4: Adding the new methods.*

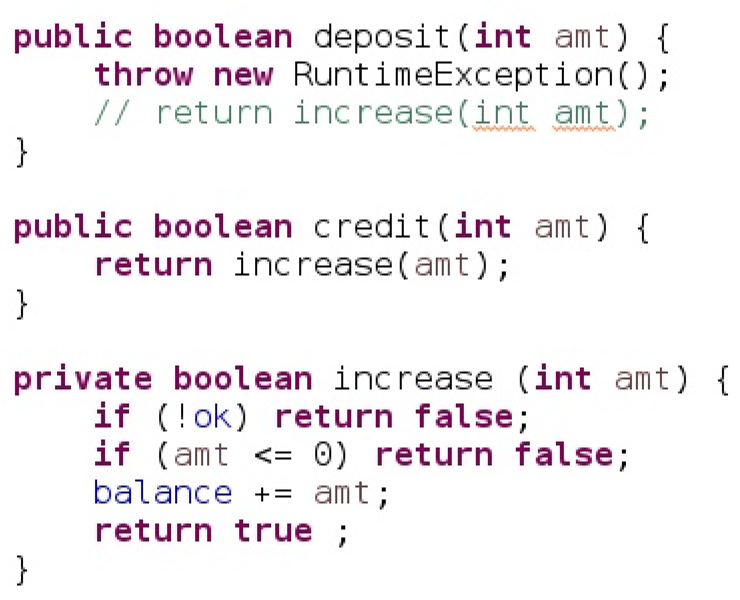
Since the functionality of the two new methods is exactly the same we can use copies of the existing unit tests. Just keep in mind that we are not going to be adding new code, we are going to modify the TDD process just a bit. Step 1 has already been done if you are using the starter workspace.

1. Add the new interface to the project and implement the interface in the My Acct class that we have been using and add the stubs for the interface methods.
2. In the body of debit() method, make a function call to the decrease() private method.
3. In the body of the credit() method, make a function call to the increase() private method.
4. Run all the tests. At this point if one of the tests fails, it means that we somehow broke the connection between the previous interface and the private methods because these tests all passed earlier.

## *Part 5: Updating the tests*

Now we have interface methods that are not covered by any tests, which is not a desirable situation. Remember Boris Beizer's advice "If you don't test it, rip it out." There are two ways we can update the tests. The quick and dirty way is to just take a copy of all the deposit() tests, past them into our test class and rename them to credit() tests. The problem with doing this is that we may not modify the tests correctly and when we think we are testing the credit() method, the test is in fact calling the deposit() method because we forgot to make the change in the test.

1. First we disable the deposit() and withdraw() methods so we will know if we call them by mistake. A common way to do this is to add an exception as the first line in the method body as shown on the next page. We want to use an unchecked exception so we don't have to worry about try blocks.
2. Put an @Ignore annotation on each of the deposit() test methods.
3. Now we copy each of the deposit() test in turn. For each test, we make changes so that it runs on the credit() method and not the deposit() method.



```java
public boolean deposit(int amt) {
    throw new RuntimeException();
    // return increase(int amt);
}

public boolean credit(int amt) {
    return increase(amt);
}

private boolean increase (int amt) {
    if (!ok) return false;
    if (amt <= 0) return false;
    balance += amt;
    return true ;
}
```

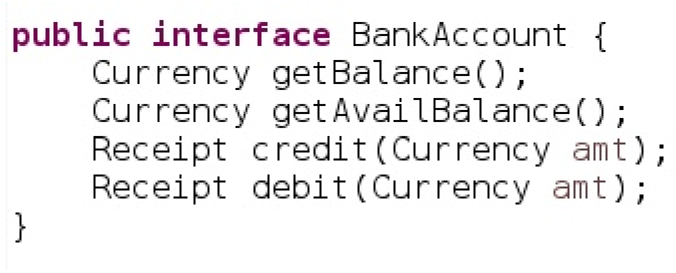
4. After we each the test, we run all the tests. None of the deposit() tests should run since they are all ignored, and if we copied the test without errors, then all the tests should pass.
5. Once all the tests have been copied and verified, we remove line in the deposit() method we added that threw the Exception and we remove all of the @Ignore annotation from all the deposit tests.
6. Now we run all of the tests and if they all pass, a fully tested credit() method has been added.

The important part is to run all of the tests after each test method we copy. This way we catch any errors that occur and can fix them right then rather than running all the tests only at the end of the process.

Now repeat the process for the debit() method. A full solution is available in the solutions as lab7-1_solution_final.

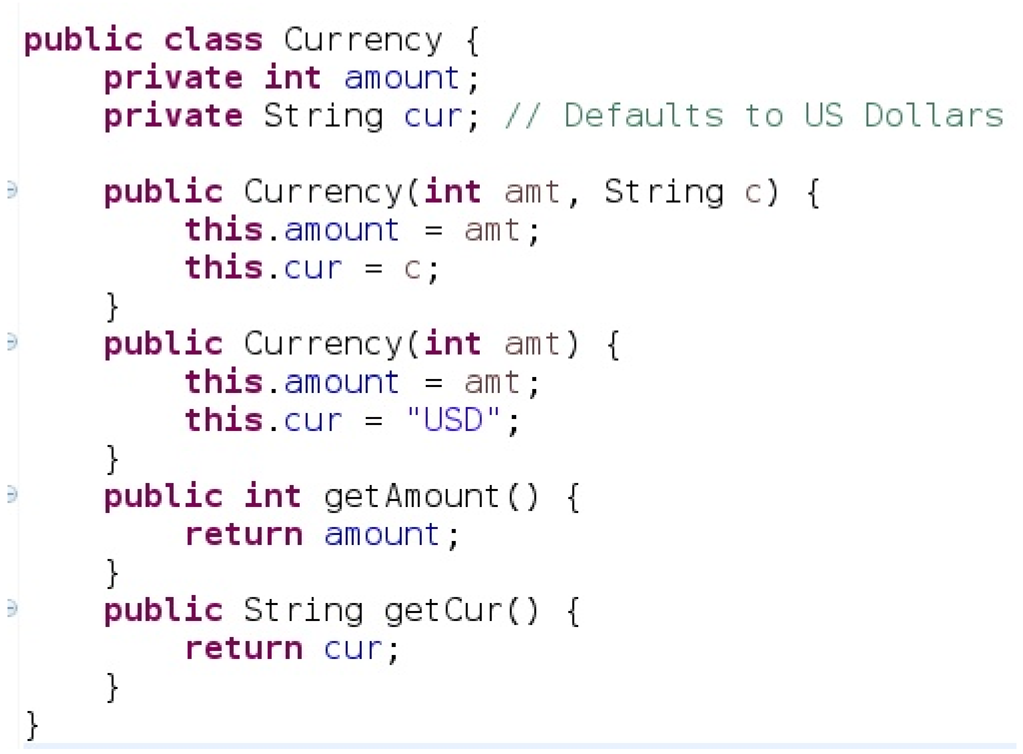
## *Lab 7-2: Refactoring to a design change*

One of the obvious code smells that the current design has is primitive obsession. The methods in the BnakAccount interface all take primitive data types as parameters and return values. The design change that will be made is to redesign the interface to look like this:

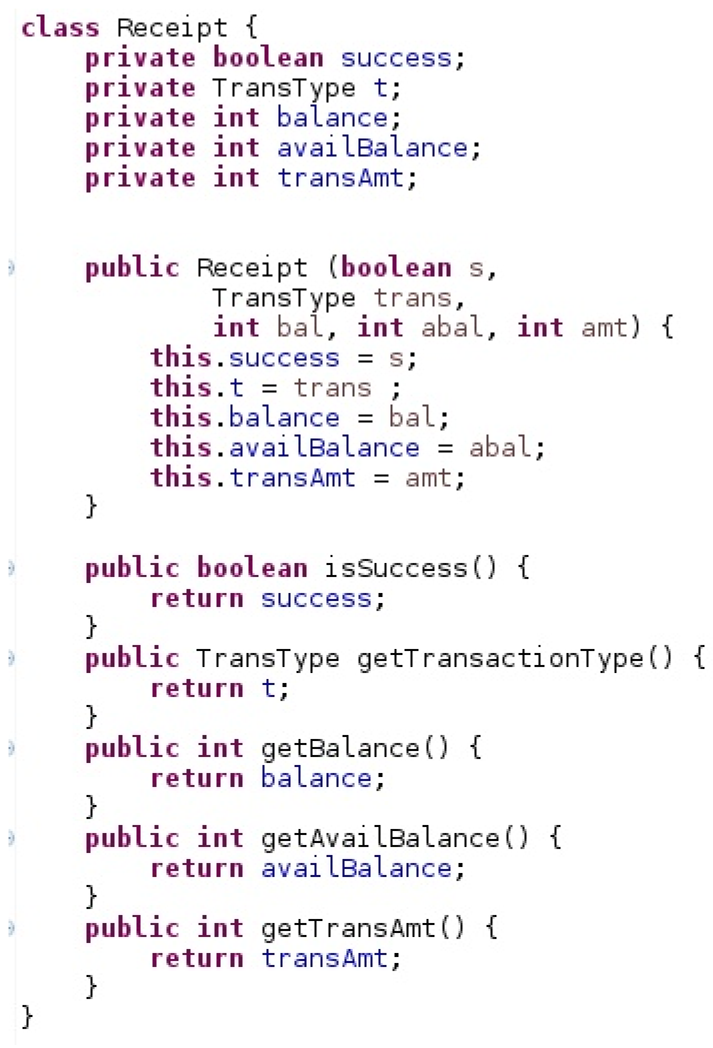


```
public interface BankAccount {
    Currency getBalance();
    Currency getAvailBalance();
    Receipt credit(Currency amt);
    Receipt debit(Currency amt);
}
```

Where the Receipt and Currency classes are defined as shown. These two classes are supplied and fully documented in the starter workspace for this lab.



```
public class Currency {
    private int amount;
    private String cur; // Defaults to US Dollars

    public Currency(int amt, String c) {
        this.amount = amt;
        this.cur = c;
    }
    public Currency(int amt) {
        this.amount = amt;
        this.cur = "USD";
    }
    public int getAmount() {
        return amount;
    }
    public String getCur() {
        return cur;
    }
}
```



```
class Receipt {
    private boolean success;
    private TransType t;
    private int balance;
    private int availBalance;
    private int transAmt;


    public Receipt (boolean s,
            TransType trans,
            int bal, int abal, int amt) {
        this.success = s;
        this.t = trans ;
        this.balance = bal;
        this.availBalance = abal;
        this.transAmt = amt;
    }

    public boolean isSuccess() {
        return success;
    }
    public TransType getTransactionType() {
        return t;
    }
    public int getBalance() {
        return balance;
    }
    public int getAvailBalance() {
        return availBalance;
    }
    public int getTransAmt() {
        return transAmt;
    }
}
```

## *Part 1: Analysis of the changes*

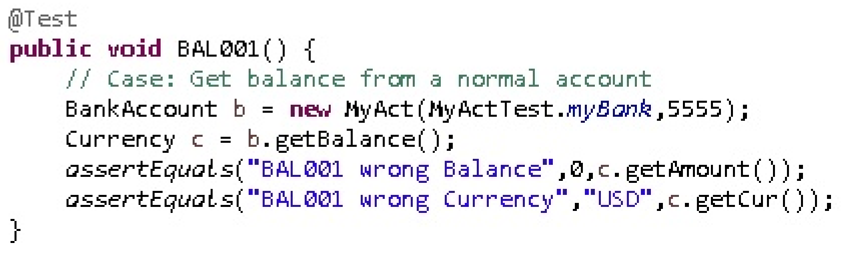
Before any code change are made, we have to ensure we understand exactly how the existing code, as in the supplied started workspace, needs to change.

1. None of the behaviors of the test cases will change, just the form the return values and parameters take for the interface methods. Since there are two changes required, we will implement each of them independently.
2. The simplest change will be to the parameter. For the sake of time, we will assume that the Receipt and Currency classes have been fully tested.
3. We do have queries and commands, however since all the functional code is already written and tested, but we can make our work easier by refactoring the queries first.
4. The changes in the application interface and helper classes are all provided in the Lab7-2_starter workspace in the resources directory. Read through starter file so you can understand what we are starting with.
5. We have commented out all of the test methods that relied on the old interface. As we implement the new interface, we will convert each test to drive and test the interface redesign.

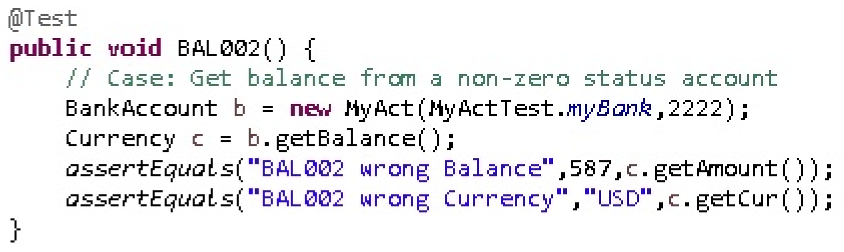
## *Part 2: Adding the new getBalance() and getAvailBalance() method.*

We will do this incrementally/

1. First we rewrite the BAL001 and BAL002 tests so that they work with a Currency object instead of an int. Notice that the test cases are still logically the same, meaning they test the same condition, they just access the data differently.

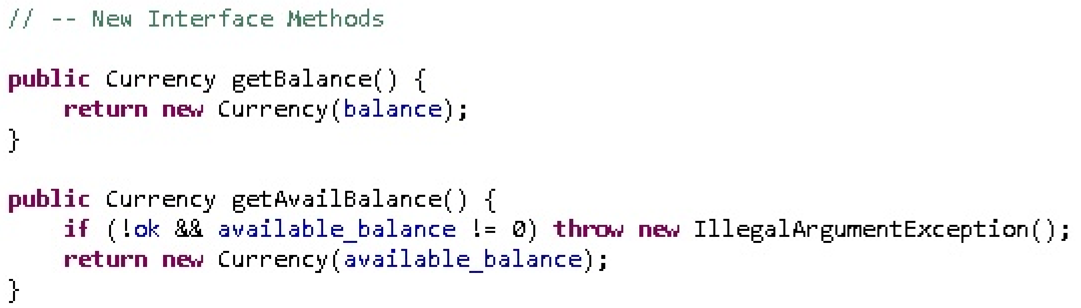


```
@Test
public void BAL001() {
    // Case: Get balance from a normal account
    BankAccount b = new MyAct(MyActTest.myBank,5555);
    Currency c = b.getBalance();
    assertEquals("BAL001 wrong Balance",0,c.getAmount());
    assertEquals("BAL001 wrong Currency","USD",c.getCur());
}
```



```
@Test
public void BAL002() {
    // Case: Get balance from a non-zero status account
    BankAccount b = new MyAct(MyActTest.myBank,2222);
    Currency c = b.getBalance();
    assertEquals("BAL002 wrong Balance",587,c.getAmount());
    assertEquals("BAL002 wrong Currency","USD",c.getCur());
}
```

2. Run the two new tests and they should fail.
3. Add the production code to make them pass



```
// -- New Interface Methods

public Currency getBalance() {
    return new Currency(balance);
}

public Currency getAvailBalance() {
    if (!ok && available_balance != 0) throw new IllegalArgumentException();
    return new Currency(available_balance);
}
```

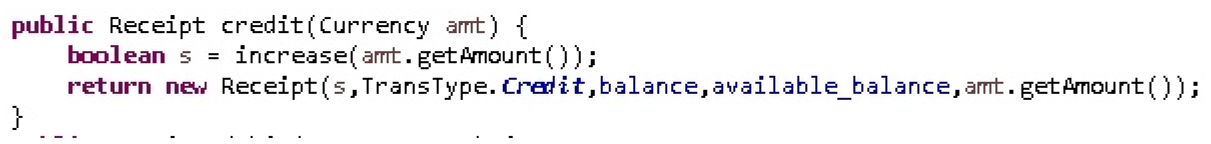
4. Now do the same for the getAvailBalance() tests and method
5. After you copy each test, run all the tests to ensure both of the getBalance() and get-AvailBalance() methods work..

A full solution of the lab up until this point is available in the solutions folder as lab7-2_solution_step1.

## *Part 3: Adding the new credit() method.*

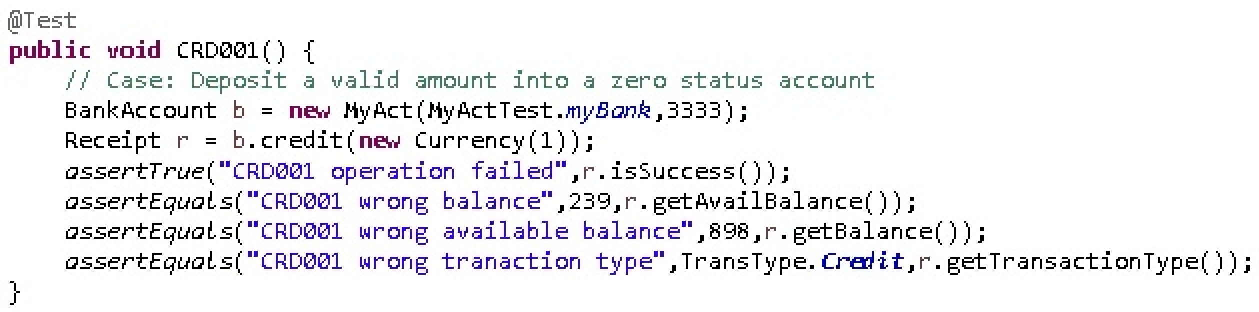
In some sense this is a bit easier since we only have to convert the credit() test methods and not with deposit() methods.

1. In this section, we can make the code change to the credit() method first. Notice that we are not changing any actual functionality for any of the test cases in terms of logic, only how the inputs and return values are prepared. The code change that we want to make is illustrated below.



```java
public Receipt credit(Currency amt) {
    boolean s = increase(amt.getAmount());
    return new Receipt(s,TransType.Credit,balance,available_balance,amt.getAmount());
}
```

2. For each of the credit() test methods, make the same sort of modifications that are shown below. Notice that we get a Receipt object as the result of the credit() operation, and then we need to check that each of the fields in the Receipt object is set correctly.



```java
@Test
public void CRD001() {
    // Case: Deposit a valid amount into a zero status account
    BankAccount b = new MyAct(MyActTest.myBank,3333);
    Receipt r = b.credit(new Currency(1));
    assertTrue("CRD001 operation failed",r.isSuccess());
    assertEquals("CRD001 wrong balance",239,r.getAvailBalance());
    assertEquals("CRD001 wrong available balance",898,r.getBalance());
    assertEquals("CRD001 wrong tranaction type",TransType.Credit,r.getTransactionType());
}
```

A complete solution for this part is in in the solutions folder as lab7-2_solution_step2.

## *Part 4: Adding the new debit() method*

Exactly the same process as for the credit() method. A complete solution for this part and the whole lab is in in the solutions folder as lab7-2_solution_final.

## *Lab 7-3: Refactoring to Patterns*

This is not really a course on design patterns however two design patterns that often used to refactor code smells are the state and strategy patterns. One of the testable anti-patterns we looked at in module 4 was the hidden state pattern. In this lab we will refactor our bank account example to a state pattern, and for good measure, we will throw in the singleton pattern as well

### *Part 1: Additions to the Specification*

In true Agile style, we are going to change the requirements and specification to reflect a change in bank policies regarding accounts. The changes revolve around how accounts with different status codes are managed in terms of deposits and withdrawals. The new status codes are:

1. Code 0 “Normal”: All transactions are processed normally
2. Code 1 “New”: Regardless of any limits initially allocated to the account, during the first 5 business days after an account is opened, the user is limited to a transaction and session limit of $100
3. Code 2: “Suspended”: No transactions may be performed on accounts that are suspended. Available balances on suspended accounts must be reported as $0. Balances are reported as normal.
4. Code 34: “Garnishment”: Only deposits are allowed on garnishments and the available balance must always be reported as $0. Balances are reported normally.
5. Code 99: “Closed”: No transactions of any type are allowed on closed accounts, including queries. Balances and available balances are always reported as $0.
6. All other listed codes have are now used for internal tracking and should not affect the normal processing of accounts. These new rules now supersede all of the previous rules for how accounts with different codes have been processed in the past. For all other administrative codes (5,7,10,23), the accounts are processed as if they were in a normal state.
7. These status codes and processing rules are expected to change in the future as more of the internal banking status codes(around two dozen) are introduced into the automated banking system.

In our existing code, we only have been concerned about one issue, whether or not an account has a zero balance. Now we have the behavior of our methods dependent upon the state of the account. This means that we have to get rid of the previous binary distinction between “ok” and not ok.

In addition, we have also been provided with a set of rules that will describe the state transition logic for the account statuses.

1. No account may be placed into a New (1) state. This state is assigned only when the account is first opened by a bank employee.
2. All accounts may be placed into a Normal (0) state except Closed (99) accounts. In fact, Closed (99) accounts can never change status.
3. Suspended (2) and Closed (99) accounts may not be place into a Garnishment (34) state.

4. Accounts that are under Garnishment (34) cannot be Suspended (2) or Closed (99).
5. Only Normal (0) accounts can be assigned any of the other administrative status codes (5,7,10,23). The only status that can be assigned to an account in one the administrative statuses is the Normal (0) status.

We will also assume that we have been asked to provide two new interface methods for a BankAccount of the form:

boolean changeStatus(int newStatus);

int getStatus();

These methods perform the obvious functionality with the changeStatus() method returning true if the state transition was successful. Idempotent state changes (where the new and old statuses are the same) are considered to be successes.

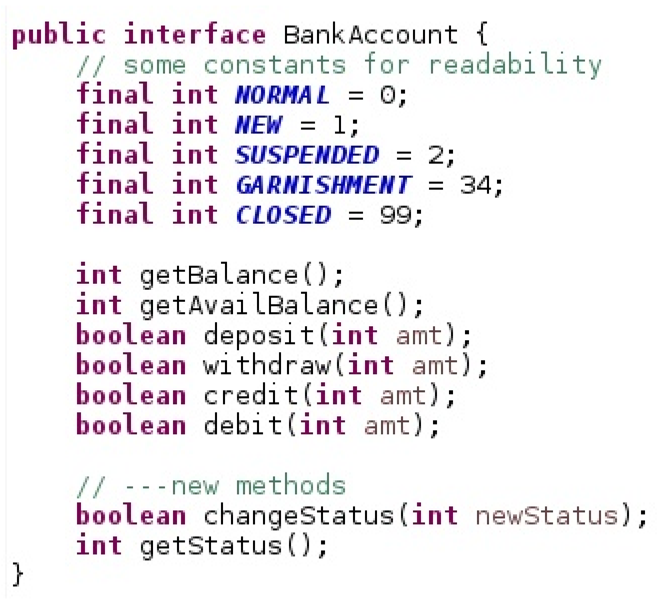
## *Part 2: Create a State Diagram and Matrix*

To make sure that you have an understanding of the state transitions, draw a statechart or state transition diagram. One is also provided in the module 7 lab resources folder. This will help you visualize the state transitions for the account status. You will also find it useful to create a finite state automata table. One is provided in the "transitions" tab in the spreadsheet in the lab 7-3 resources folder.

## *Part 3: Refactor to add status codes.*

To keep things simple, we are going to start with the lab as it was at the end of lab7-1. However, that code is a bit sloppy and will get sloppier as we add the new functionality. We are going to take the opportunity when we refactor the code later to clean it up as well. A good TDD practice is to make the code "cleaner" as we refactor.

1. Add the new interface methods to the Bank Account interface and stub them out in the MyAct class.



```java
public interface BankAccount {
    // some constants for readability
    final int NORMAL = 0;
    final int NEW = 1;
    final int SUSPENDED = 2;
    final int GARNISHMENT = 34;
    final int CLOSED = 99;

    int getBalance();
    int getAvailBalance();
    boolean deposit(int amt);
    boolean withdraw(int amt);
    boolean credit(int amt);
    boolean debit(int amt);

    // ---new methods
    boolean changeStatus(int newStatus);
    int getStatus();
}
```

2. You will also find it useful to create some final constants in the interface for the status codes 0,1,2,34 and 99 so that you can refer to them as NORMAL, NEW, SUSPENDED,etc. This is good coding practice. The administrative codes will be left as numeric since we don't really have meaningful names for them.
3. Add some new accounts so that we have accounts with the appropriate status codes to work with. There are some added to the existing accounts we have used so far in the spreadsheet in the lab 7 resources folder in the "accounts" tab. Add them to the MockDB so that they will be ready to use.
4. Add a new private integer instance variable "status". Do not remove the boolean "ok" instance variable yet.
5. Once all this is done, run the tests and everything should now pass.
6. Look for all the tests that you have been running that work on accounts with a "not-ok" status. Since we are refactoring the status codes, these tests are obsolete and should be removed. They are just commented out in the solution so you can see what tests should have been removed.

## *Implementing getStatus()*

By now this should be a known process. It is a good idea to try your hand and developing a set of test cases yourself. There is also a set provided in the spreadsheet under the tab "statustests". For this first method, we only need one test case since this really is a straight getter with no internal logic.

## *Implementing changeStatus()*

When developing the change status tests, we have to make sure that we test each transition in the statechart and also test to make sure that we cannot perform an illegal state transition. This can be quite a lot of tests. To make life easier, the statechart has been rewritten as finite state machine transition matrix in the "transitions" tab.

Theoretically, we would want to have a test for each of the transitions or for each cell in the matrix. A general rule of good design is to keep the number of states as minimal as possible because the bigger the table, the more likely we are to make some sort of state transition error. For the purposes of this class, we will not require you to implement tests for all of them, just for a few key cases, which provide fairly good coverage of the state transition.

There are 36 possible transitions, not all of which are legal. Now the decision is how many of the possible test cases we should use. Some considerations:

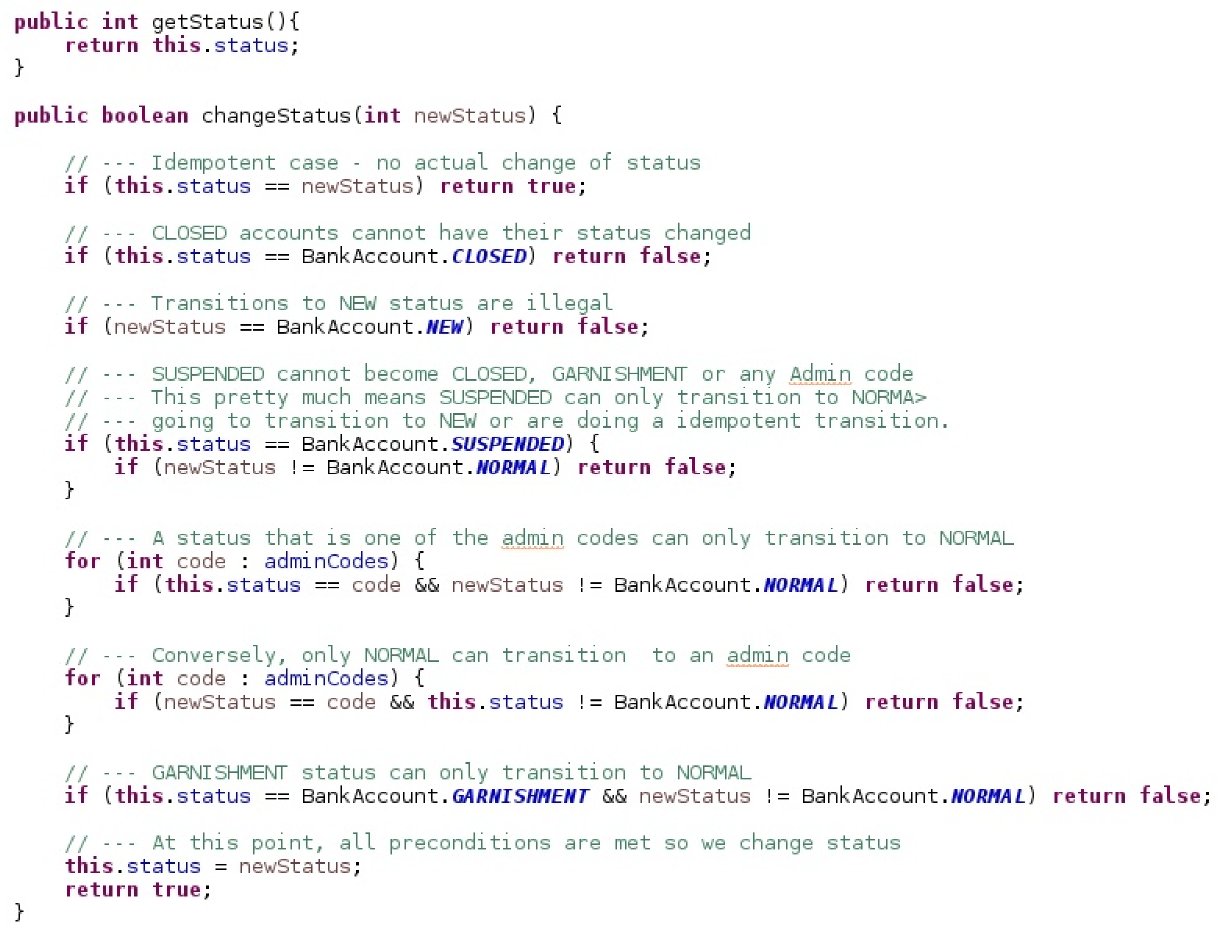
1. One of the factors that will affect our decision is a risk analysis. Getting some of the transitions wrong may introduce more risk in terms of negative outcomes for the application. We may want to rank the transitions by risk and add the tests for the most risky ones first. For example, suspending or unsuspending an account may be done in response to a legal action so getting that one wrong may have introduce some real world liability. On the other hand, the idempotent transitions may be relatively low risk.
2. Another factor we may want to consider is the frequency of the transactions. The higher frequency transitions (like new to normal) may be a priority to test more thoroughly than the lower frequency transitions (new to closed).

3. Ideally, we would like to test all the transitions so we try to ensure all of the state transition logic is exercised as much as we can where we often prioritize in terms of risk and frequency.

The strategy taken is described below:

1. We will test one valid transition for each status and verify that we cannot do any of the invalid transitions.
2. All of the idempotent transitions form an equivalence class because they are all handled by the same code. So we will only add one idempotent test case.
2. All non-idempotent transitions out of CLOSED are illegal so they also form an equivalence class so we will add only one of those tests.
3. All non-idempotent transitions into the NEW status are illegal so again, one test case for this.
4. And so we continue on only starting with a subset of all possible test cases. Notice that the order in which we add the production code is very important and also may suggest that we probably need to refactor that code.

The spreadsheet shows what would be a minimal set of tests that we would want to develop to, however there is nothing to prevent us from adding more.



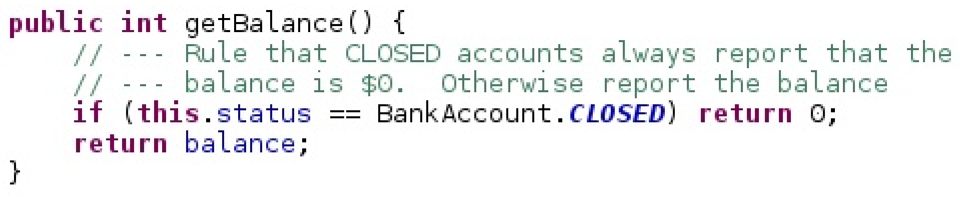
```java
public int getStatus(){
    return this.status;
}

public boolean changeStatus(int newStatus) {

    // --- Idempotent case - no actual change of status
    if (this.status == newStatus) return true;

    // --- CLOSED accounts cannot have their status changed
    if (this.status == BankAccount.CLOSED) return false;

    // --- Transitions to NEW status are illegal
    if (newStatus == BankAccount.NEW) return false;

    // --- SUSPENDED cannot become CLOSED, GARNISHMENT or any Admin code
    // --- This pretty much means SUSPENDED can only transition to NORMA>
    // --- going to transition to NEW or are doing a idempotent transition.
    if (this.status == BankAccount.SUSPENDED) {
        if (newStatus != BankAccount.NORMAL) return false;
    }

    // --- A status that is one of the admin codes can only transition to NORMAL
    for (int code : adminCodes) {
        if (this.status == code && newStatus != BankAccount.NORMAL) return false;
    }

    // --- Conversely, only NORMAL can transition  to an admin code
    for (int code : adminCodes) {
        if (newStatus == code && this.status != BankAccount.NORMAL) return false;
    }

    // --- GARNISHMENT status can only transition to NORMAL
    if (this.status == BankAccount.GARNISHMENT && newStatus != BankAccount.NORMAL) return false;

    // --- At this point, all preconditions are met so we change status
    this.status = newStatus;
    return true;
}
```

### *Implementing the changeStatus() method*

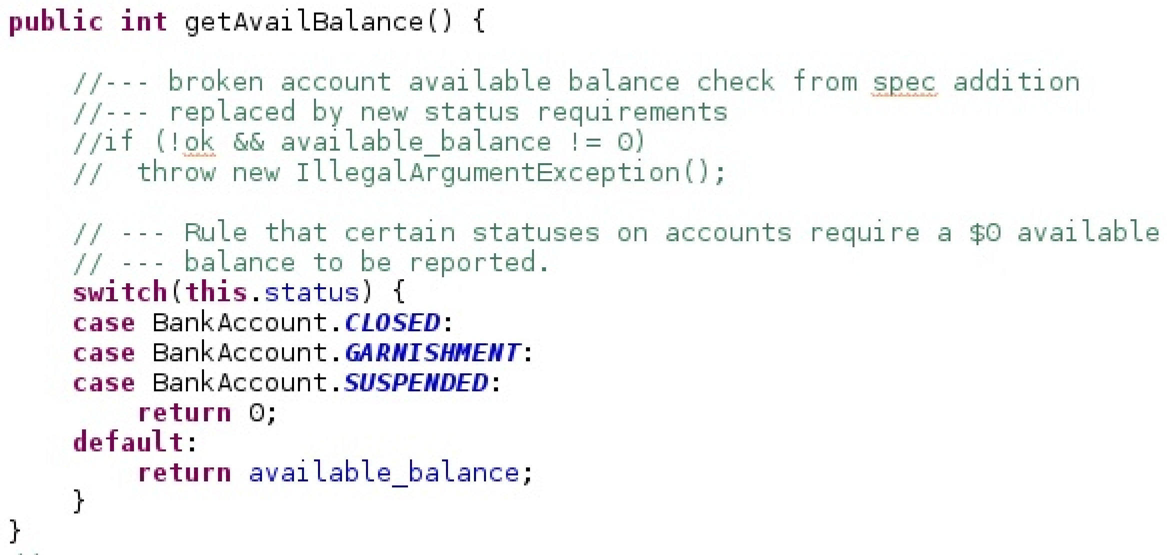
Adding the test, implement enough code, as simple as you can make it, that results in the tests passing. This can start to get somewhat poorly structured. Notice that the original requirements were a bit sloppy and that seems to be reflected in the code.  You code may be a lot cleaner, but the example counts on refactoring things in the next part of the lab.

### *Modifying the getAvailBalance() and getBalance() methods*

Notice that there are some rules on how different statuses report the balance and available balance.  You should implement a couple of tests (refer to the spreadsheet) and make the modifications.  Remember to ensure that all of your tests still pass.



```java
public int getBalance() {
    // --- Rule that CLOSED accounts always report that the
    // --- balance is $0.  Otherwise report the balance
    if (this.status == BankAccount.CLOSED) return 0;
    return balance;
}
```

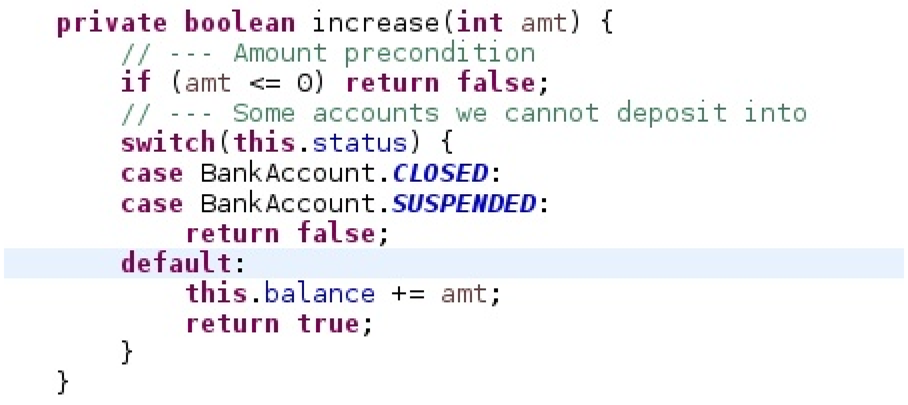


```java
public int getAvailBalance() {

    //--- broken account available balance check from spec addition
    //--- replaced by new status requirements
    //if (!ok && available_balance != 0)
    //  throw new IllegalArgumentException();

    // --- Rule that certain statuses on accounts require a $0 available
    // --- balance to be reported.
    switch(this.status) {
    case BankAccount.CLOSED:
    case BankAccount.GARNISHMENT:
    case BankAccount.SUSPENDED:
        return 0;
    default:
        return available_balance;
    }
}
```

### *Modifying the deposit() and credit() methods*

We only need to do one of these since they both call the actual underlying private increase() method where the code modification takes place.

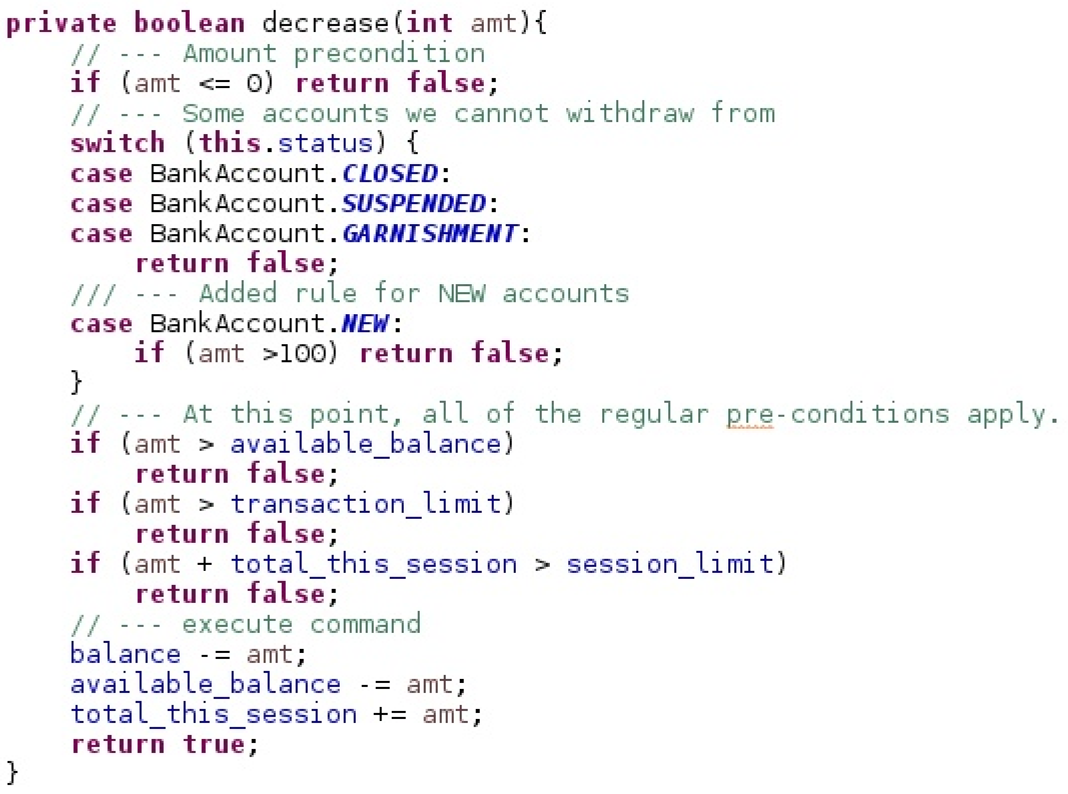
1. Add the new deposit tests.  Notice the ones we already have should be kept except for those that talk about “non-ok” status accounts.  Those can be replaced by the new test cases in the spreadsheet.  The old tests are just commented out in the solution.
2. Modify the code to make the tests pass



```java
private boolean increase(int amt) {
    // --- Amount precondition
    if (amt <= 0) return false;
    // --- Some accounts we cannot deposit into
    switch(this.status) {
    case BankAccount.CLOSED:
    case BankAccount.SUSPENDED:
        return false;
    default:
        this.balance += amt;
        return true;
    }
}
```

### *Modifying the withdraw() and debit() methods*

Repeat the process you did in the previous section



```java
private boolean decrease(int amt){
    // --- Amount precondition
    if (amt <= 0) return false;
    // --- Some accounts we cannot withdraw from
    switch (this.status) {
    case BankAccount.CLOSED:
    case BankAccount.SUSPENDED:
    case BankAccount.GARNISHMENT:
        return false;
    /// --- Added rule for NEW accounts
    case BankAccount.NEW:
        if (amt >100) return false;
    }
    // --- At this point, all of the regular pre-conditions apply.
    if (amt > available_balance)
        return false;
    if (amt > transaction_limit)
        return false;
    if (amt + total_this_session > session_limit)
        return false;
    // --- execute command
    balance -= amt;
    available_balance -= amt;
    total_this_session += amt;
    return true;
}
```

At this point, you should be able to remove the definition of the “ok” instance variable.

A full solution up to this point is available in workspace lab7-3_part1

## *Lab 7-3: Refactoring to Patterns*

This is not really a course on design patterns however two design patterns that often used to refactor code smells are the state and strategy patterns. One of the testable anti-patterns we looked at in module 4 was the hidden state pattern. In this lab we will refactor our bank account example to a state pattern, and for good measure, we will throw in the singleton pattern as well

### *Part 1: Additions to the Specification*

In true Agile style, we are going to change the requirements and specification to reflect a change in bank policies regarding accounts. The changes revolve around how accounts with different status codes are managed in terms of deposits and withdrawals. The new status codes are:

1. Code 0 “Normal”: All transactions are processed normally
2. Code 1 “New”: Regardless of any limits initially allocated to the account, during the first 5 business days after an account is opened, the user is limited to a transaction and session limit of $100
3. Code 2: “Suspended”: No transactions may be performed on accounts that are suspended. Available balances on suspended accounts must be reported as $0. Balances are reported as normal.
4. Code 34: “Garnishment”: Only deposits are allowed on garnishments and the available balance must always be reported as $0. Balances are reported normally.
5. Code 99: “Closed”: No transactions of any type are allowed on closed accounts, including queries. Balances and available balances are always reported as $0.
6. All other listed codes have are now used for internal tracking and should not affect the normal processing of accounts. These new rules now supersede all of the previous rules for how accounts with different codes have been processed in the past. For all other administrative codes (5,7,10,23), the accounts are processed as if they were in a normal state.
7. These status codes and processing rules are expected to change in the future as more of the internal banking status codes(around two dozen) are introduced into the automated banking system.

In our existing code, we only have been concerned about one issue, whether or not an account has a zero balance. Now we have the behavior of our methods dependent upon the state of the account. This means that we have to get rid of the previous binary distinction between “ok” and not ok.

In addition, we have also been provided with a set of rules that will describe the state transition logic for the account statuses.

1. No account may be placed into a New (1) state. This state is assigned only when the account is first opened by a bank employee.
2. All accounts may be placed into a Normal (0) state except Closed (99) accounts. In fact, Closed (99) accounts can never change status.
3. Suspended (2) and Closed (99) accounts may not be place into a Garnishment (34) state.

4. Accounts that are under Garnishment (34) cannot be Suspended (2) or Closed (99).
5. Only Normal (0) accounts can be assigned any of the other administrative status codes (5,7,10,23). The only status that can be assigned to an account in one the administrative statuses is the Normal (0) status.

We will also assume that we have been asked to provide two new interface methods for a BankAccount of the form:

```
boolean changeStatus(int newStatus);

int getStatus();
```

These methods perform the obvious functionality with the changeStatus() method returning true if the state transition was successful. Idempotent state changes (where the new and old statuses are the same) are considered to be successes.

## *Part 2: Create a State Diagram and Matrix*

To make sure that you have an understanding of the state transitions, draw a statechart or state transition diagram. One is also provided in the module 7 lab resources folder. This will help you visualize the state transitions for the account status. You will also find it useful to create a finite state automata table. One is provided in the "transitions" tab in the spreadsheet in the lab 7-3 resources folder.

## *Part 3: Refactor to add status codes.*

To keep things simple, we are going to start with the lab as it was at the end of lab7-1. However, that code is a bit sloppy and will get sloppier as we add the new functionality. We are going to take the opportunity when we refactor the code later to clean it up as well. A good TDD practice is to make the code "cleaner" as we refactor.

1. Add the new interface methods to the Bank Account interface and stub them out in the MyAct class.
2. You will also find it useful to create some final constants in the interface for the status codes 0,1,2,34 and 99 so that you can refer to them as NORMAL, NEW, SUSPENDED,etc. This is good coding practice. The administrative codes will be left as numeric since we don't really have meaningful names for them.
3. Add some new accounts so that we have accounts with the appropriate status codes to work with. There are some added to the existing accounts we have used so far in the spreadsheet in the lab 7 resources folder in the "accounts" tab. Add them to the MockDB so that they will be ready to use.
4. Add a new private integer instance variable "status". Do not remove the boolean "ok" instance variable yet.
5. Once all this is done, run the tests and everything should now pass.
6. Look for all the tests that you have been running that work on accounts with a "not-ok" status. Since we are refactoring the status codes, these tests are obsolete and should be removed. They are just commented out in the solution so you can see what tests should have been removed.

## *Implementing getStatus()*

By now this should be a known process. It is a good idea to try your hand and developing a set of test cases yourself. There is also a set provided in the spreadsheet under the tab “statustests”. For this first method, we only need one test case since this really is a straight getter with no internal logic.

## *Implementing changeStatus()*

When developing the change status tests, we have to make sure that we test each transition in the statechart and also test to make sure that we cannot perform an illegal state transition. This can be quite a lot of tests. To make life easier, the statechart has been rewritten as finite state machine transition matrix in the “transitions” tab.

Theoretically, we would want to have a test for each of the transitions or for each cell in the matrix. A general rule of good design is to keep the number of states as minimal as possible because the bigger the table, the more likely we are to make some sort of state transition error. For the purposes of this class, we will not require you to implement tests for all of them, just for a few key cases, which provide fairly good coverage of the state transition.

There are 36 possible transitions, not all of which are legal. Now the decision is how many of the possible test cases we should use. Some considerations:

1. One of the factors that will affect our decision is a risk analysis. Getting some of the transitions wrong may introduce more risk in terms of negative outcomes for the application. We may want to rank the transitions by risk and add the tests for the most risky ones first. For example, suspending or unsuspending an account may be done in response to a legal action so getting that one wrong may have introduce some real world liability. On the other hand, the idempotent transitions may be relatively low risk.
2. Another factor we may want to consider is the frequency of the transactions. The higher frequency transitions (like new to normal) may be a priority to test more thoroughly than the lower frequency transitions (new to closed).
3. Ideally, we would like to test all the transitions so we try to ensure all of the state transition logic is exercised as much as we can where we often prioritize in terms of risk and frequency.

The strategy taken is described below:

1. We will test one valid transition for each status and verify that we cannot do any of the invalid transitions.
2. All of the idempotent transitions form an equivalence class because they are all handled by the same code. So we will only add one idempotent test case.
2. All non-idempotent transitions out of CLOSED are illegal so they also form an equivalence class so we will add only one of those tests.
3. All non-idempotent transitions into the NEW status are illegal so again, one test case for this.
4. And so we continue on only starting with a subset of all possible test cases. Notice that the order in which we add the production code is very important and also may suggest that we probably need to refactor that code.

The spreadsheet shows what would be a minimal set of tests that we would want to develop to, however there is nothing to prevent us from adding more.

### *Implementing the changeStatus() method*

Adding the test, implement enough code, as simple as you can make it, that results in the tests passing. This can start to get somewhat poorly structured. Notice that the original requirements were a bit sloppy and that seems to be reflected in the code. You code may be a lot cleaner, but the example counts on refactoring things in the next part of the lab.

### *Modifying the getAvailBalance() and getBalance() methods*

Notice that there are some rules on how different statuses report the balance and available balance. You should implement a couple of tests (refer to the spreadsheet) and make the modifications. Remember to ensure that all of your tests still pass.

### *Modifying the deposit() and credit() methods*

We only need to do one of these since they both call the actual underlying private increase() method where the code modification takes place.

1. Add the new deposit tests. Notice the ones we already have should be kept except for those that talk about “non-ok” status accounts. Those can be replaced by the new test cases in the spreadsheet. The old tests are just commented out in the solution.
2. Modify the code to make the tests pass

### *Modifying the withdraw() and debit() methods*

Repeat the process you did in the previous section

At this point, you should be able to remove the definition of the “ok” instance variable.

A full solution up to this point is the solutiosn workspace lab7-3_part1

## *Lab 7-3: Refactoring to Patterns Continued*

If you are unfamiliar with the State Design pattern, then you should consult one of the standard references online for design patterns or ask the instructor to explain it to you. The state pattern is where we want to refactor messy state logic (like we have at the end of the last section) into a more OO type of structure.

The essence of the state pattern is to encapsulate state specific logic into state objects. For example, the withdraw() method has different logic depending on the status of the account. In this case we have a Garnishment class that contains only the logic that is used for a withdrawal() while a bank account is in that state. The client calling the withdraw() method on the bank account does not need to do anything different to use the refactored bank account object. When the withdraw method is sent to Bank Account object, it just forwards or delegates the method to the Garnishment class. Delegation is one of the basic OO object design techniques.

However remember that we have to think of refactoring as changes in the implementation, not the interface, which means that the state specific logic is in the private increase() and decrease() methods, not the withdraw() or debit() or other public methods. So the methods we want in our state objects are state specific increase() and decrease() methods.

To refactor our code, we will do a couple of preliminary steps to start the restructuring process.

1. We create a “status” interface
2. We implement that interface in a set of classes that correspond to the states. Each state implements its own version of the private increase() and decrease() methods where the state specific version of each method handles the logic only for that state.
3. When we call deposit() or getBalance() or other method on a BankAccount object, when the increase() or decrease() method is called it delegates to the specific state object verison

### *Part 1: Creating the State Interface and classes.*

The first step is to create the interface as shown on the next page. Notice there are some final constants defined in the interface – we will explain those in Part 2.

```
//-------------------------------------
// ---- The Status Interface
//-------------------------------------

interface Status {
    // -- Meaningful names for data[] indices
    final int status = 0;
    final int balance = 1;
    final int available_balance = 2;
    final int transaction_limit = 3;
    final int session_limit = 4;
    final int total = 5;

    // --- the query methods are told what the current balance or available balance are
    // --- for the account at that time, then use the state specific logic to alter the
    // --- reported value, if necessary.
    int getBalance(int bal);
    int getAvailBalance(int avl);

    // --- The increase method takes the current data for the account, and the amount,
    // --- Then applies state specific logic to update the account information.
    boolean increase(int amt, int[] data);
    boolean decrease(int amt, int[] data);
    // --- Changes the status, updates the data and returns an object of the new status type
    // --- If the status doesn't change it just returns a reference to itself.
    Status changeStatus(int newStatus, int[] data);
}
```

We also stub out the methods for each of the state classes as show below. Don't worry about the changeStatus() method for now, we will deal with that in Part 3.

We are now face with the design issue of how we manage these various status objects. The following is a standard approach.

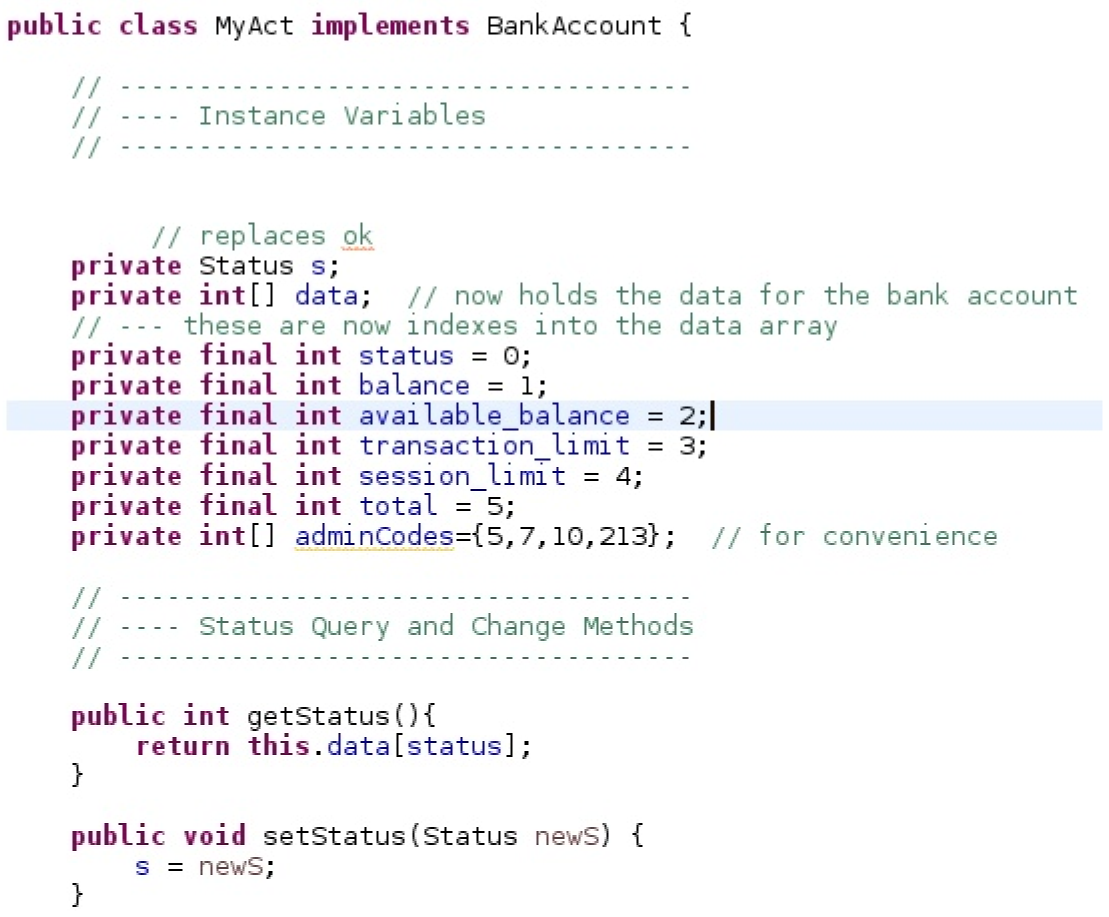
1. We have a Status reference variable in the BankAccount class that points to the current status. Changing the status is equivalent to swapping in a different status object.
2. To maintain a set of status objects to use, we use a factory method and a singleton pattern. Every time a status object is needed, it is requested from the factory. If one already exists, a reference to that one is returned; if not, then one is created and the new reference returned. Since the status objects themselves have no internal state, we can re-use and share them without problems.
3. Notice that we have also defined a number of constants to make the code more read-able by giving the status descriptive names.

```
//-----------------------------------
//---- The Normal Class
//-----------------------------------

class Normal implements Status {

    public int getBalance(int bal) {
        return -1;
    }

    public int getAvailBalance(int avl) {
        return avl;
    }

    public boolean increase(int amt, int[] data) {
        return false;
    }

    public boolean decrease(int amt, int[] data) {
        return false;
    }

    public Status changeStatus(int newStatus, int[] data) {
        return null;
    }

}
```

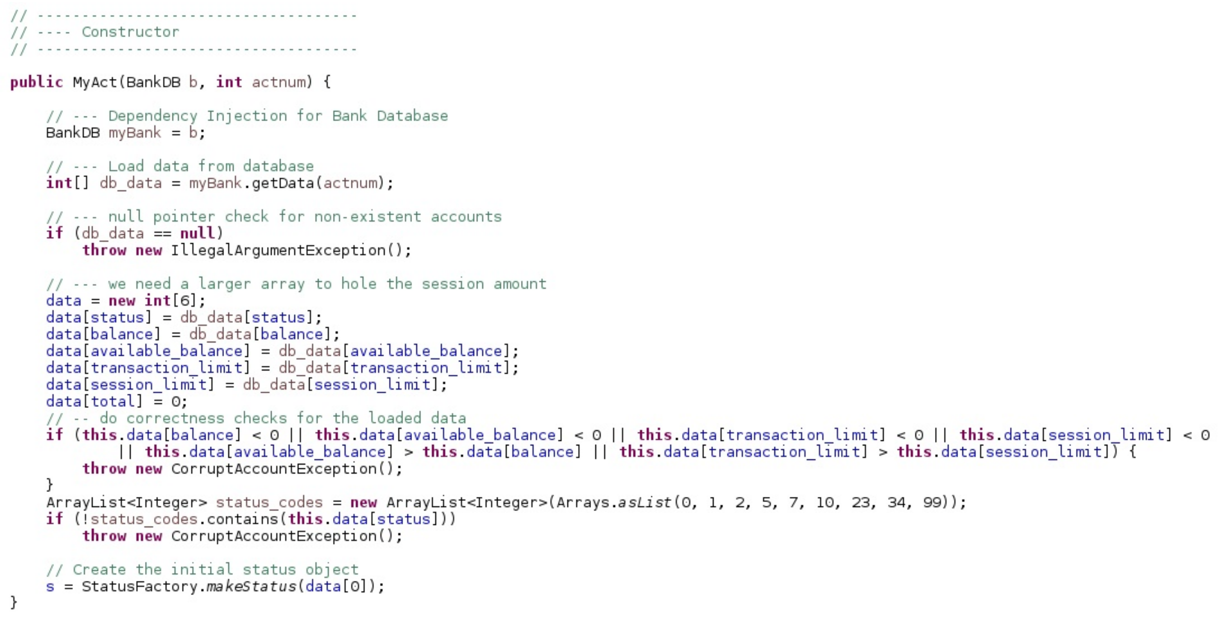
One of the problems we have introduced is that by moving the code to the delegated status object, it can no longer update any of the instance variables in the actual bank account because the code and the variables are in different classes.

There are a number of not so good solutions to this (the worst of which would be to make all those bank account instance variables public), but there is one solution that is often used which is sort of based on the memento pattern. It works like this:

1. The account object wants another object, in this case a status object, to perform some calculations that will affect its internal data but it also does not need the coupling of creating a code smell (inappropriate intimacy). If we were programming in C++ we could call the status "Friend" classes to the bank account class, but there is a more efficient way to do this.
2. Instead, we eliminate all of the instance variables and just pass an object around that holds the current internal data for the bank account. We just use the data[] array with a couple of additions – and each entry in the array is not referenced by numerical index by a constant to make the code more readable.
3. Notice that we are correcting what could have been another code smell – a data clump – by packaging all the data up into this array. When we pass this data clump as a parameter, recall that we are passing a reference to the data array and the status object can update the contents without problem.
4. The images below illustrate this.

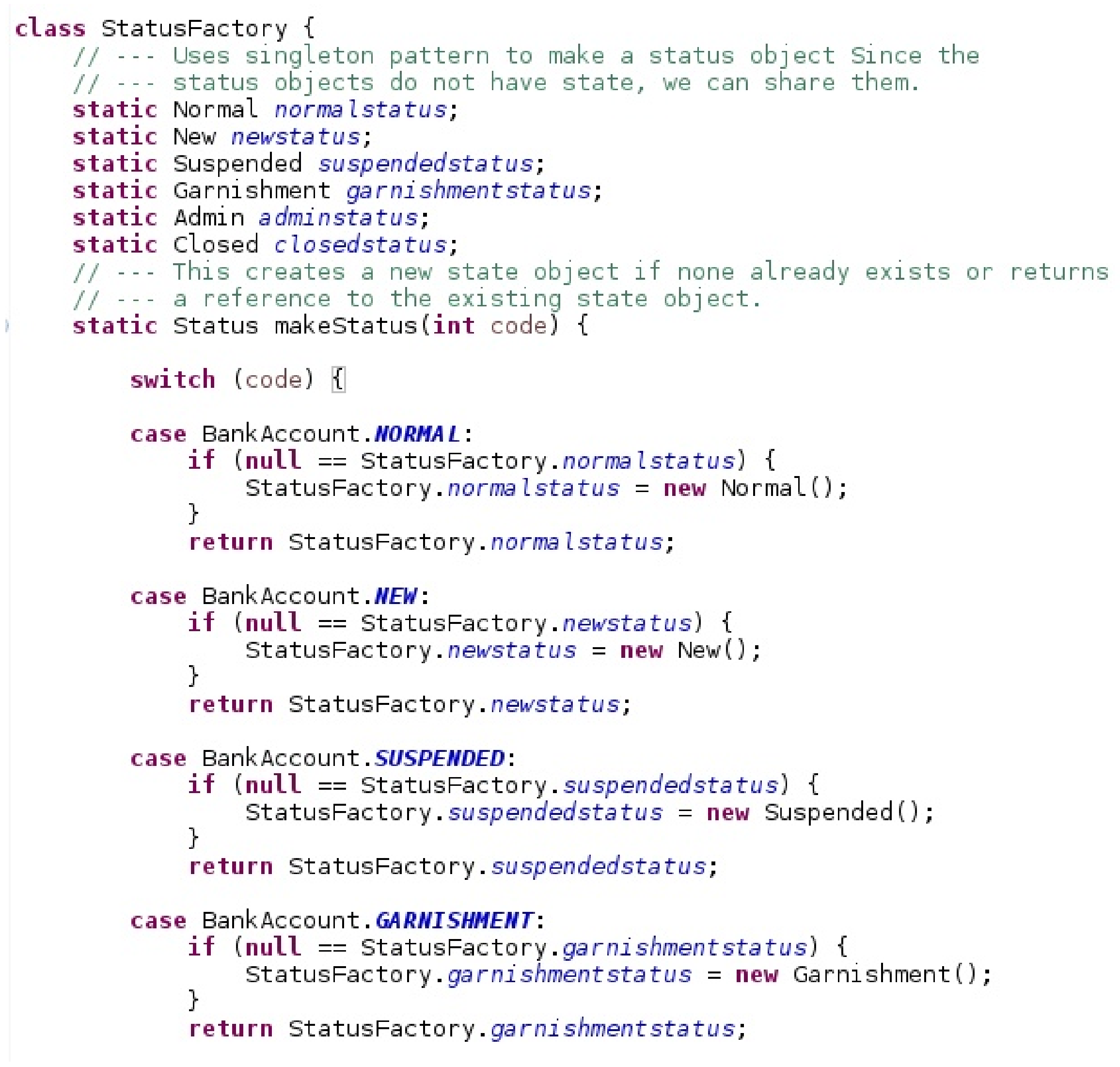


```
public class MyAct implements BankAccount {

    // -----------------------------------
    // ---- Instance Variables
    // -----------------------------------


        // replaces ok
    private Status s;
    private int[] data;  // now holds the data for the bank account
    // --- these are now indexes into the data array
    private final int status = 0;
    private final int balance = 1;
    private final int available_balance = 2;
    private final int transaction_limit = 3;
    private final int session_limit = 4;
    private final int total = 5;
    private int[] adminCodes={5,7,10,213};  // for convenience

    // -----------------------------------
    // ---- Status Query and Change Methods
    // -----------------------------------

    public int getStatus(){
        return this.data[status];
    }

    public void setStatus(Status newS) {
        s = newS;
    }
```



```java
// ------------------------------------
// ---- Constructor
// ------------------------------------

public MyAct(BankDB b, int actnum) {

    // --- Dependency Injection for Bank Database
    BankDB myBank = b;

    // --- Load data from database
    int[] db_data = myBank.getData(actnum);

    // --- null pointer check for non-existent accounts
    if (db_data == null)
        throw new IllegalArgumentException();

    // --- we need a larger array to hole the session amount
    data = new int[6];
    data[status] = db_data[status];
    data[balance] = db_data[balance];
    data[available_balance] = db_data[available_balance];
    data[transaction_limit] = db_data[transaction_limit];
    data[session_limit] = db_data[session_limit];
    data[total] = 0;
    // -- do correctness checks for the loaded data
    if (this.data[balance] < 0 || this.data[available_balance] < 0 || this.data[transaction_limit] < 0 || this.data[session_limit] < 0
            || this.data[available_balance] > this.data[balance] || this.data[transaction_limit] > this.data[session_limit]) {
        throw new CorruptAccountException();
    }
    ArrayList<Integer> status_codes = new ArrayList<Integer>(Arrays.asList(0, 1, 2, 5, 7, 10, 23, 34, 99));
    if (!status_codes.contains(this.data[status]))
        throw new CorruptAccountException();

    // Create the initial status object
    s = StatusFactory.makeStatus(data[0]);
}
```

As mentioned in the previous section, we are using a factory to produce the status objects. The code below illustrates how this is done.



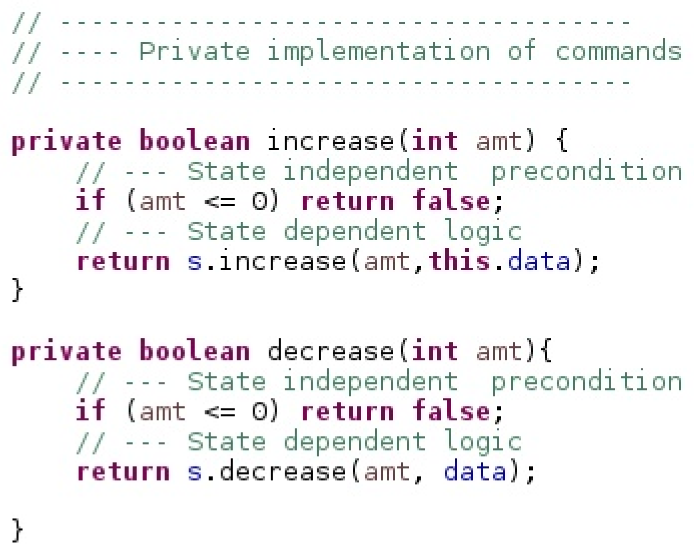
```java
class StatusFactory {
    // --- Uses singleton pattern to make a status object Since the
    // --- status objects do not have state, we can share them.
    static Normal normalstatus;
    static New newstatus;
    static Suspended suspendedstatus;
    static Garnishment garnishmentstatus;
    static Admin adminstatus;
    static Closed closedstatus;
    // --- This creates a new state object if none already exists or returns
    // --- a reference to the existing state object.
    static Status makeStatus(int code) {

        switch (code) {

        case BankAccount.NORMAL:
            if (null == StatusFactory.normalstatus) {
                StatusFactory.normalstatus = new Normal();
            }
            return StatusFactory.normalstatus;

        case BankAccount.NEW:
            if (null == StatusFactory.newstatus) {
                StatusFactory.newstatus = new New();
            }
            return StatusFactory.newstatus;

        case BankAccount.SUSPENDED:
            if (null == StatusFactory.suspendedstatus) {
                StatusFactory.suspendedstatus = new Suspended();
            }
            return StatusFactory.suspendedstatus;

        case BankAccount.GARNISHMENT:
            if (null == StatusFactory.garnishmentstatus) {
                StatusFactory.garnishmentstatus = new Garnishment();
            }
            return StatusFactory.garnishmentstatus;
```

## *Part 3: Making the Changes*

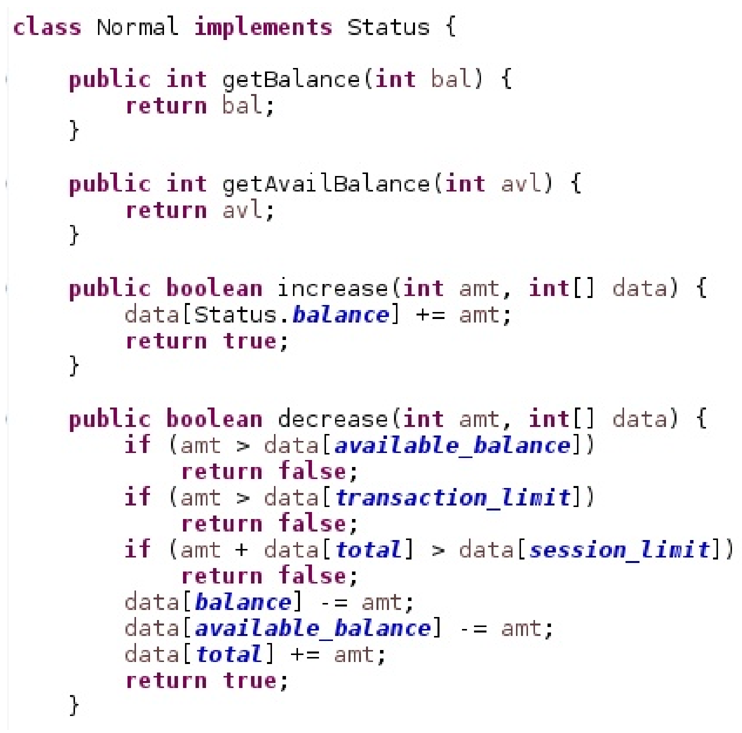
Since we haven't actually implemented the connections between our new code and our interface methods, all of the tests we have should still pass.

The process we need to follow is like this:

1. @Ignore out all of the tests except for those that function only in the Normal state, including all of the state transitions tests (i.e. we are not running the state transitions tests... yet).
2. Move the logic for the Normal status for the increase() and decrease() methods into the status object and add the delegation code to the bank account increase() and decrease() methods.



```
// ------------------------------------
// ---- Private implementation of commands
// ------------------------------------

private boolean increase(int amt) {
    // --- State independent  precondition
    if (amt <= 0) return false;
    // --- State dependent logic
    return s.increase(amt,this.data);
}

private boolean decrease(int amt){
    // --- State independent  precondition
    if (amt <= 0) return false;
    // --- State dependent logic
    return s.decrease(amt, data);

}
```



```
class Normal implements Status {

    public int getBalance(int bal) {
        return bal;
    }

    public int getAvailBalance(int avl) {
        return avl;
    }

    public boolean increase(int amt, int[] data) {
        data[Status.balance] += amt;
        return true;
    }

    public boolean decrease(int amt, int[] data) {
        if (amt > data[available_balance])
            return false;
        if (amt > data[transaction_limit])
            return false;
        if (amt + data[total] > data[session_limit])
            return false;
        data[balance] -= amt;
        data[available_balance] -= amt;
        data[total] += amt;
        return true;
    }
```

3. Run all non-@Ignored tests and ensure they run.
4. We can now do this for each of the other statuses – un-@Ignore each set of tests in turn and move the code.

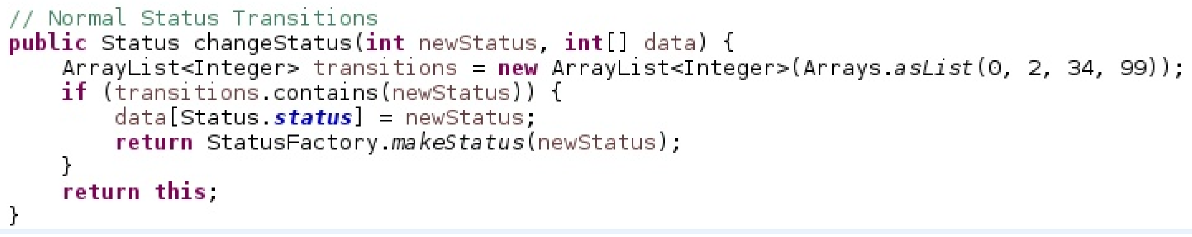
### *Part 4: Change of Status*

A change of status is a change of state.

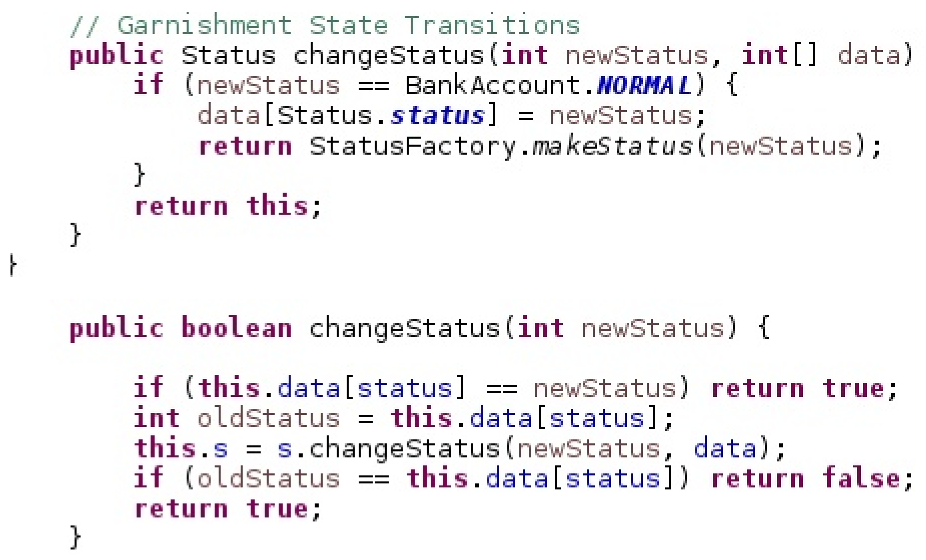
1. Rather than have some master set of state transition logic, what we will do is have each state know what transitions it can make.
2. When we ask a status object to do status change, and if it is a valid status change, then we return a reference to the new status object.
3. If the status change is not valid, then a reference to itself is returned (i.e. the status does not change).

We can refactor this in the following way:

1. Add the state transition logic into each state.
2. Have the bank account changeStatus() method delegate the request to the changeSt-atus() method in the current status object.



```java
// Normal Status Transitions
public Status changeStatus(int newStatus, int[] data) {
    ArrayList<Integer> transitions = new ArrayList<Integer>(Arrays.asList(0, 2, 34, 99));
    if (transitions.contains(newStatus)) {
        data[Status.status] = newStatus;
        return StatusFactory.makeStatus(newStatus);
    }
    return this;
}
```



```java
    // Garnishment State Transitions
    public Status changeStatus(int newStatus, int[] data) {
        if (newStatus == BankAccount.NORMAL) {
            data[Status.status] = newStatus;
            return StatusFactory.makeStatus(newStatus);
        }
        return this;
    }
}

    public boolean changeStatus(int newStatus) {

        if (this.data[status] == newStatus) return true;
        int oldStatus = this.data[status];
        this.s = s.changeStatus(newStatus, data);
        if (oldStatus == this.data[status]) return false;
        return true;
    }
```

3. Run the tests to ensure they work. You should be able to run the full suite of tests from the last section now and see all of them pass.

A full solution is available in the lab solutions folder for this lab.