

a project in
object-oriented programming 2

HOG

Multi-Purpose Coin Organizer Machine

A Java Implementation

Presented By:

BS Computer Science 4

Lee Alexis Bermejo. *Team leader and programmer*

Jan Fredrick Nietes. *Graphic designer*

Mienard Vidas. *System analyst*

Neil Valezuela. *System analyst*

Presented to:

Jason Sermenio. *Instructor*

October 15, 2010

The General Objective

Design a program in Java that simulates the operation of the Multi-Purpose Coin Organizer machine.

The Specific Objectives

1. The program should be able to demonstrate following concepts: polymorphism, encapsulation, and inheritance.
2. The program should be able to simulate the **depositing of coins** into the virtual MPCO machine.
3. The program should be able to simulate the **Coin Inquiry transaction** of the MPCO machine.
4. The program should be able to simulate the **Coin Withdrawal** transaction of the MPCO machine.
5. The program should be able to simulate the **Change Security PIN** transaction of the MPCO machine.
6. The program should be as similar to its real-life MPCO counterpart as possible.

The Analysis

Java implementation and function

To meet our objective of simulating the MPCO machine as Java program, we created a class we called `Hog`. We used “`Hog`” to associate this class with the piggy bank— but more advanced and secure (because if it weren’t, we could just have named it “`Pig`”, creative folks that we are.) The `Hog` class is an extension of the class `JFrame` from `java.awt` library. So we now we have a window — in the real world, this is now the Multi-Purpose Coin Organizer machine’s box that is “built with an indestructible material”.

To simulate the components of an MPCO machine (openings, number pad, display panel) we extended the classes `JLabel` and `JButton` from the `java.awt` package to create new ones that use images. This helped us create a Look-and-Feel that is totally customizable and simulates what the MPCO controls would have looked like in the real world.

Now with the “physical” blueprint complete, it’s time to proceed to simulating the operations.

A graphical interface means that the flow of the program would be controlled by actions — in `Hog`’s case, by mouse clicks. Clicks in Java do nothing unless you implement the `ActionListener` interface.

The virtual machine `Hog` would just sit there and wait for clicks on the menu or touchpad buttons. Simply adding a method (on what to do) on each button would not be good enough.

For example, during Coin Withdrawal, the number pad buttons must cause the display panel to display numbers, but during PIN entry, the number pad buttons must cause the display panel to display a “mask” by using asterisks, but must still save the value somewhere. Another example would be the ENTER button. Its function also varies on every transaction. To solve this problem of button multi-use, we used some variables to act as **flags** – indicating which operation `Hog` is currently running. We also need to break the methods for the transactions (Inquiry, Withdrawal, Change PIN, PIN Entry) into sections. These methods are broken at the portions where they need to wait for user input.

Action Listening

Depositing coins would be simple because `Hog` does not prompt for PIN 1 and 2. The remaining transactions (Inquiry, Withdrawal, and Change Pin) however, requires `Hog` to prompt for PIN 1 and 2.

Since PIN entry is such a common action between those three methods, we could just create a method `PINentry()` and add it as the first instruction in each of the three transactions, right?

Wrong. For some reason the transactions skip `PINentry()` and proceed to the next instruction.

Preferably, the flow should be like this. Click the Inquire button, then the code of the module (which includes a call to PIN entry) executes.

PIN Entry module

```
Statement1;  
Statement2 ;  
...  
StatementN;
```

Inquiry module

```
PINEntry(); //this skips  
Statement1;  
Statement2;  
...  
StatementN;
```

and the same for the other transactions.

However, Hog only has one method that can listen for actions and execute code accordingly — the `actionPerformed()` method of the `ActionListener` interface. So how can we tell Hog to stop and listen for PIN input?

What we did is we split the modules into sections and then added the portions where Hog needs to listen for input into the `actionPerformed()` method. But now we have a cocktail mix of code of various modules inside `actionPerformed()`. How can we select which piece of code inside `actionPerformed()` to execute that corresponds to their module?

This is solved by using flag variables and a simple `if-else` block inside `actionPerformed()` that will select which piece of code to run depending on what transaction Hog is currently doing.

The Flags

Since we need to track which operation or transaction the user has executed, we use flag variables. Flag variables determine which piece of code the `actionPerformed()` method should fire.

The `int opMode` determines which module is being executed —

- 0 means no transaction (this is used to indicate that Hog is displaying the start screen, usually after a “refresh”);
- 1 tells `actionPerformed()` that the user is currently entering PINs;
- 2 means the user is inquiring his balance;
- 3 means the user wants to withdraw, and
- 4 means the user wants to change his PINs.

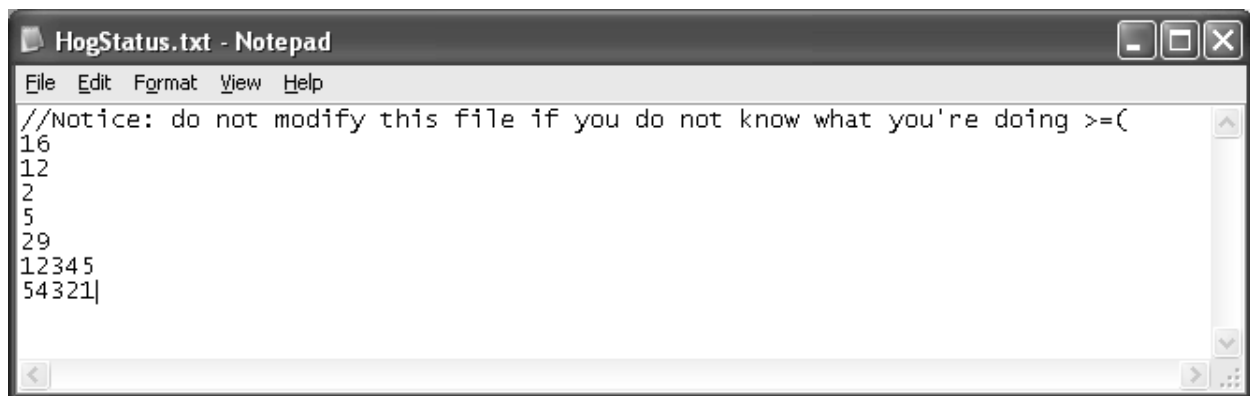
The `boolean apr` (`apr` means “approval”) determines if the user has entered correct PINs and Hog should proceed with the transaction. If this is `true`, it means that the PIN entry portion of the current module has ended with correct PIN's, and the number pad buttons may now change behavior from displaying a password mask to displaying plain numbers, depending on what transaction.

The variable `confirm`, also `boolean`, is used for the ENTER button to change behavior during a PIN change. A `false` means it is the first textual entry for the PIN currently being changed, and a `true` means that the textual entry occurring at the moment is a confirmation check for that new PIN. If it matches, the new PIN is saved to `HogStatus.txt`.

There are other flags, but their purpose are quite trivial.

Hog Status

The amount stored in the MPCO does not change or reset to zero when it is switched off, and the MPCO knows the amount it is storing when it is switched on. To simulate that ability, `Hog` uses a text file called `HogStatus.txt` to save the status of the coins, with each line representing a value. It also saves the PINs.



```
//Notice: do not modify this file if you do not know what you're doing >=(
16
12
2
5
29
12345
54321|
```

The first line is a warning against tampering.

The second line indicates the amount of 10-peso coins.

The third line indicates the amount of 5-peso coins.

The fourth line indicates the amount of 1-peso coins.

The fifth line indicates the amount of 25-centavo coins.

The sixth line indicates the amount of 10-centavo coins.

The seventh line indicates the PIN1.

The third line indicates the PIN2.

We created two methods for `Hog` to interact with the text file.

- `readFileLine(int iLine)` returns a `String` value that comes from the line number indicated by the `int iLine`.
- `writeFileLine(String strData ,int iLine)` writes the value of `strData` on the line indicated by `int iLine`. It overwrites the current value.

Using the text file, `Hog`'s coins does not reset to zero when `Hog` is exited.

Coin Deposit

Depositing coins is the simplest thing a user can do on Hog, and receiving them is also the simplest operation Hog can do.

Like the MPCO machine, Hog has five openings on the top – one for each coin. Technically these openings are buttons, but they look like coin-holes because of the images they hold.

Clicking one of them would execute the `addCoin(int whichCoin)` method where `whichCoin` is the line number (for the coin being added) on the `HogStatus.txt` file. `addCoin()` uses `readFileLine()` to retrieve the current value, increment it by 1, then uses `writeFileLine()` to save the result at once.

The Inquire Module

Clicking the Inquire button changes the `opMode` value into 2. The behavior of the number pad changes accordingly.

After a successful PIN entry, Hog shows a menu of inquiry choices. `opMode = 2` tells `actionPerformed()` —which fires when a button is clicked— to listen for clicks on the number buttons 1 to 6, which then execute a corresponding piece of code. Hog retrieves coin values from the `HogStatus.txt` file and displays them as output.

Clicking the Dot button will “refresh” Hog —reassign variables and flags to default values and return it to its start screen with `opMode == 0`.

The Withdraw Module

Clicking the Withdraw button changes `opMode` to 3, then Hog asks for PINs.

Upon success, Hog prompts for an amount to withdraw, An `opMode` with the value of 3 tells `actionPerformed()` to listen to all number pad buttons.

- Clicking a number will append that number to the display.
- Clicking the Dot will append a decimal point if there is none.
- Clicking the C button will clear a digit from the number displayed.
- Clicking ENTER will process the amount.

If all calculations go well, the user is notified of the successful withdrawal. The withdrawn value is subtracted from the values saved in `HogStatus.txt` and `opMode` is reset to 0.

The Change PIN Module

Clicking this button will change `opMode` to 4. After a successful PIN entry, the user faces a menu to choose which PIN to change. A flag (an integer named `pinChangeChoice`) tells `actionPerformed()` to listen to number pad buttons 1 and 2. When the user clicks one of those two buttons, the flag changes. Now `actionPerformed()` listens on all number pad buttons. The display acts similarly as the way it does during PIN entry.

If the confirmation of the new PIN matches, the machine refreshes. If not, the user can clear his input and try again.

The PIN entry module

The PIN module works differently than the other modules. It is started by a call on the method `PINEntryThenSet()`, so called because it starts PIN entry then sets the prompt for whatever transaction follows next.

First, the current `opMode` value is saved to the `int next` flag. Then `opMode` is set to 1, which tells the number pad buttons to behave during a PIN entry —telling the panel to display asterisks while keeping the original characters somewhere, in this case inside a private variable (a `String` named `text`) inside the `lblNumDisplay` object.

After checking the PIN1 and PIN2 entries if they match the ones saved on the `HogStatus.txt` file, the module then reassigns the value in `next` to `opMode`. Then the rest of the current transaction module (Inquire, Withdraw, Change PIN) begins. The reassignment of `opMode` tells the number pad buttons to change behavior again, aside from the PIN entry behavior.

Inside Withdraw: Breaking Down the Amount

We have difficulties in breaking down the amount into the corresponding coin denominations. We needed to break the amount into a combination of the values of 10-peso, 5-peso, 1-peso, 25-cents, and 10-cents. The MPCO breaks down amounts by optimization —10-peso down to 10-cents, and depending on the current stored number of each coin.

First, we converted the withdraw amount into cents (multiply it by 100) before passing it to the recursive `breakDown(int amt, int whatDenom)` method. We converted to cents to make handling the amount easier — so that we can eliminate the decimal point and use integer variables because float variables in Java have some ridiculous issues concerning precision.

The `breakDown()` method is recursive — it calls itself again when the `int amt` is still not zero.

First `breakdown()` divides the amount by the first denomination —10-peso— which indicate the number of 10-peso coins the amount can be broken down to. Then we compare that amount to the one

saved in `HogStatus.txt`. If the saved amount is less, we use the saved amount. The difference in amount is added back to the amount, then we call `breakDown()` again, this time with 5-peso, and so on until the amount is zero. The values for the number of coins to dispense is stored in an array named `int tmpAmtPerCoin[]` of size 5.

The 25 centavo is tricky, because some amounts require 10-cents even if 25-cents can be used, as portrayed in the example in the MPCO instructions (40 cents). We beat the problem by checking if the amount requires a 25 cent denomination (by checking its remainder, if divided by 10, is equal to 5). If it does, we reserve a 25-cent coin from the `HogStatus.txt` at once. If there no 25-cent coins in the text file, `Hog` will tell the user straight out that it cannot dispense the amount.

If the amount does not look like it requires a 25-cent, but a 5-cent or 15-cent remains after being broken down optimally (an example of this kind of amount is 40 cents), we have a simple solution: we remove one 25-cent coin from the `tmpAmtPerCoin[]` array, adds it back to the amount, then removes all 10-cent coins from the array and also add them back to the amount, then we breakdown the amount again, this time using only a 10-cent denomination.

If there are any "coins-not enough" issues detected by the `if`-blocks inside `breakDown()`, the transaction is stopped at once and the user is shown an error message at once.

The values in `HogStatus.txt` are safe from accidental erasure arising from errors in calculation. `Hog` uses the `commitTransaction()` method to change the values in the text file. `commitTransaction()` is called by `breakDown()` only when it has finished calculating and no errors arose.

The Designed Model

Hog in Containment Hierarchy Model

(*Instance: Class*)

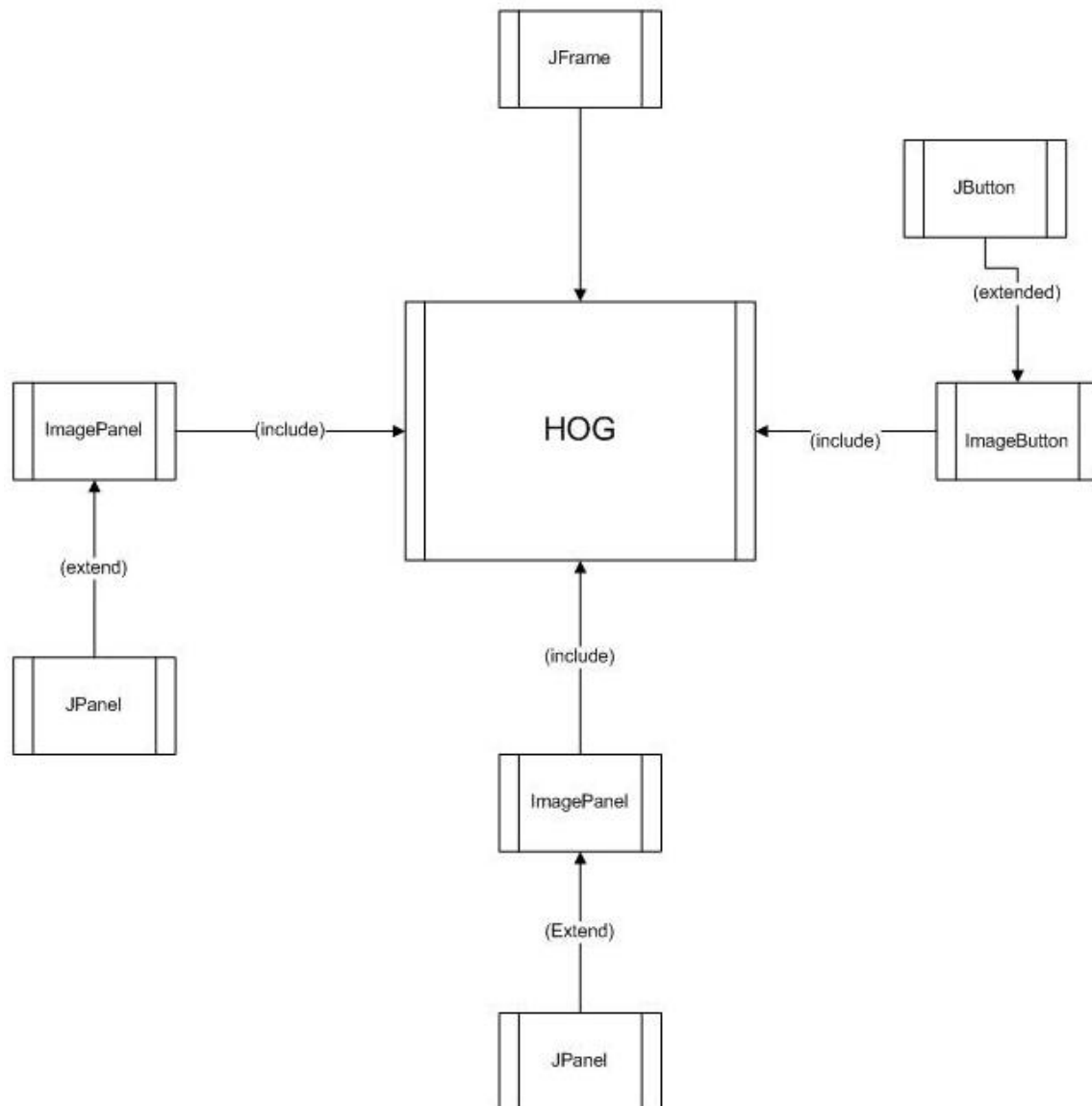
Hog: JFrame

- Wrap: ImagePanel
 - btnOpening[]: ImageButton
 - btnInquire: ImageButton
 - btnWithdraw: ImageButton
 - btnChangePIN: ImageButton
 - lblDescDisplay: ImageLabel
 - lblNumDisplay: ImageLabel
 - btnNumPad[13]: ImageButton
 - lblDispenser: ImageLabel

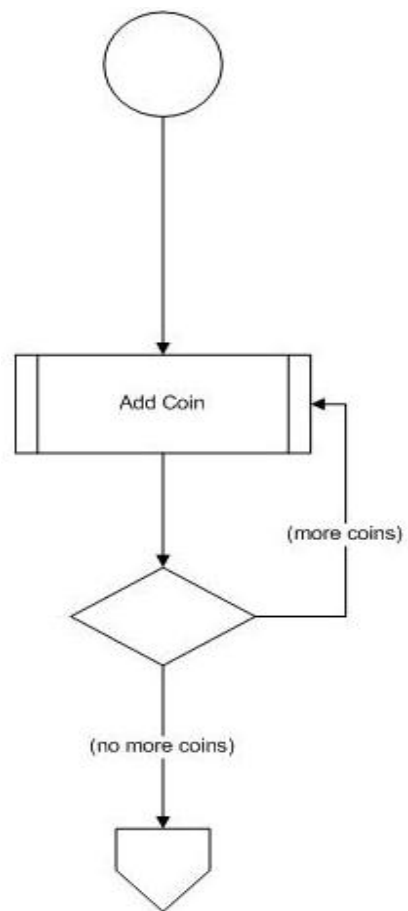
Hog is a JFrame subclass that contains wrap, an instance of the ImagePanel class.

wrap contains an assortment of instances of the classes ImageLabels and ImageButtons.

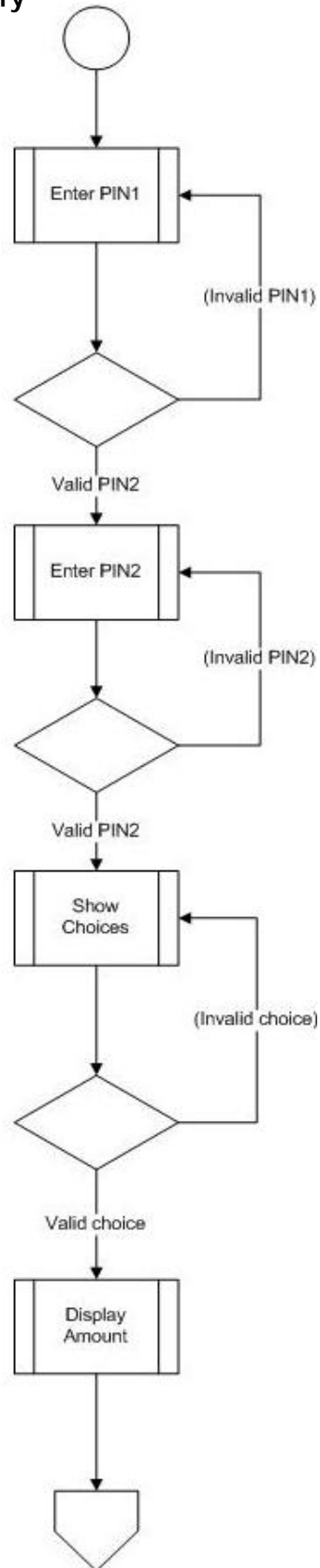
The Hog in Use-Case Model



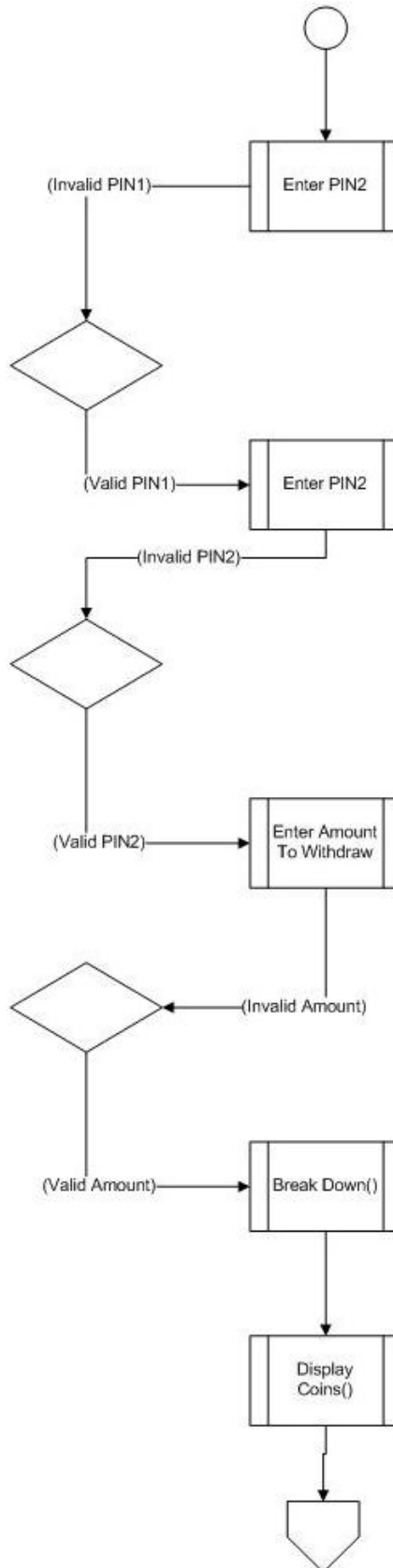
The Hog in Sequence Model: Adding a coin



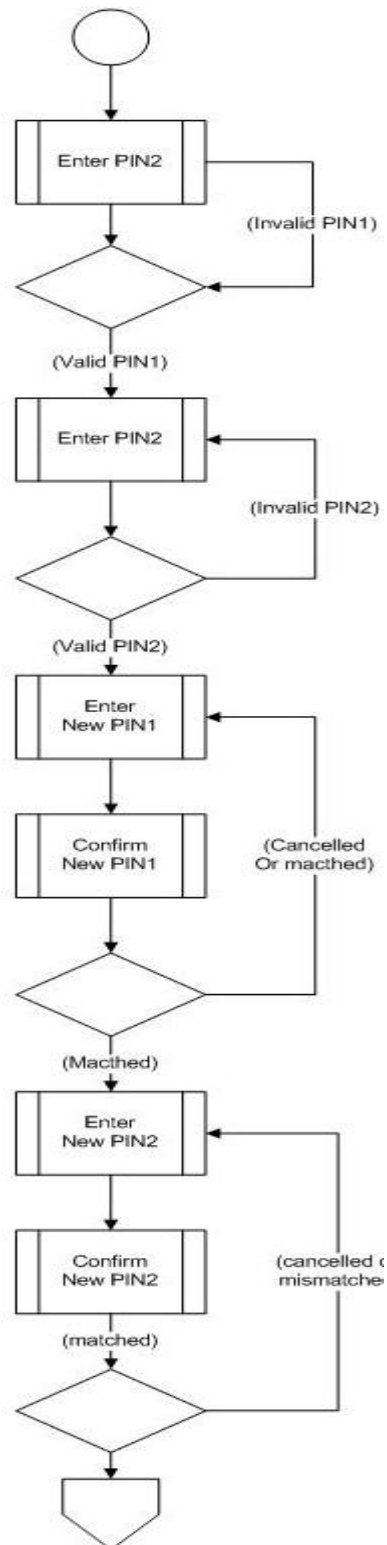
The Hog in Sequence Model: Inquiry



The Hog in Sequence Model: Withdraw



The Hog in Sequence Model: Changing PIN's



Status Report

This is a brief timeline on our work on Hog.

September 3, 2010	Members start to work on their assigned tasks and Lee Alexis Bermejo the team leader conceptualized objects to use in the Java program.
September 6-7, 2010	We formulated general and specific objectives based on the problems presented by the Multi-Purpose Coin Organizer.
September 9, 2010	Our group helped to formulate a basic GUI for the HOG.
September 10, 2010	The designer started designing the GUI in Photoshop.
September 14-18, 2010	System analysts started to encode the general and specific objectives. They also formulated class and sequence diagrams, and the programmer started coding custom classes for the GUI.
September 16, 2010	The programmer started coding the main class Hog and included the finished Photoshop images for the GUI.
September 18, 2010	System analysts created the model diagrams.
September 21-28, 2010	The programmer continued coding the main class Hog. The system analysts revised the design model as they fixed problems. The designer revised some flawed GUI images.
October 4-8, 2010	The designer finalized the GUI design and programmer is still working on Hog.
October 13-14, 2010	Our group finalized the all submitting requirements.
October 15, 2010	Our group e-mailed the documentation, the source code, and the working program to the instructor.

Bug Report

As far as we know, $\mathbb{H}\text{og}$ has no bugs. Most possible errors that can result from the MPCO design can be trapped by Hog.

Maybe it's because we have little time to test it thoroughly.

Conclusion

We haven't tested the Hog design thoroughly, but we believe that errors would be minimal.

Input is safely controlled — the user can only enter input by clicking Hog's buttons, which in turn only work if the `opMode` variable requires Hog to listen to them for input. This is best demonstrated by the menu choices. If Hog displays a menu of six choices indicated by numbers 1 to 6, Hog would only listen for clicks on buttons for 1 to 6.

The only errors possible on input in entering a withdrawal amount whose centavo portion cannot be broken into 25 or 10 cents. Example: 10.777 pesos. In this case, Hog would still trap the error and show an error.

Calculations (when breaking down a withdrawal amount) do not continue or save false coin values to `HogStatus.txt` when errors occur. Errors include:

- having amount that has a centavo portion that is indispensable in 25 or 10 centavo; not having enough coins in the `HogStatus.txt` file;
- withdrawal amount exceeds total stored amount,
- or a value —any value— suddenly results in a negative integer (this has never happened, anyway.)

There is a chance, however that the user may forget his PIN1 or PIN2, and Hog does not offer a feature for retrieving or resetting them.

We also believe that using a windowed GUI lets us fulfill our objective of making a simulation of an MPCO machine as similar as how it would look or feel in the real world.

Recommendations

We recommend that the use of Hog be limited to demonstration purposes. Its contents are make-believe: the coins, the openings, and even the dispenser, which shows a pile of coins even if you only withdraw one peso.

However, every component of Hog has an equivalent in the real life MPCO: the opening buttons are the opening, the labels for the display are the display panel, the HogStatus.txt is the vault, and so on. The Hog design can be implemented in real-life hardware.

The image-themed classes –ImageLabel, ImagePanel, ImageButton – are not our original work. They are based originally from an e-book titled “Swing Hacks”. We modified them slightly to meet Hog’s needs. These classes can be used in other Java projects that implement a GUI.