Homework 2, 4354 Summer 13

Requirements

Develop a software system for simple account management using "push" Model View Controller architecture (i.e. a view receives updates from a model by events), exception mechanism and JUnit framework in Java with Swing (cf. JUnit example at http://junit.sourceforge.net/doc/testinfected/testing.htm , http://junit.sourceforge.net/doc/cookbook/cookbook.htm http://www.elsevierdirect.com/companion.jsp?ISBN=9781558608689).

On start up the system should load a list of accounts from a file specified in a command line and present the user with a frame that contains

- a drop down list populated with account IDs with names appended (in increasing order of IDs); the first account ID should be pre-selected; the items in the drop-down list immutable
- buttons "Edit account in $"; "Edit account in Euros"; "Edit account in Yen"
- button Save
- button Exit

An entry in a file with a list of accounts should contain a string with a name (only letters allowed); a string with an ID (only digits allowed); a string with an amount in $ (digits and decimal point allowed; thus amount >=0.0). IDs are unique; names can repeat.

On pressing "Save" the system should write the current state of accounts to the file whose name appeared in the command line.

On pressing "Exit" the system should write the current state of accounts to the file whose name appeared in the command line if they have been modified since last save. After that the system should exit.

On pressing one of the buttons "Edit account in .." the system should open another window that contains:

- window's title <NameOfAccountHolder ID; Operations in {$ or Euros or Yen}>
- immutable textfield that shows the current amount in the account titled "Available funds" (i.e. Available funds: <textfield>)
- editable textfield initialized to 0.0 that only allows digits and decimal point to be entered; titled "Enter amount in {$ or Euros or Yen}"
- buttons "Deposit", 'Withdraw", "Dismiss"

On pressing "Dismiss" the window should close

On pressing "Deposit" if the editable textfield contains a positive number greater than 1 then that amount is added to the account amount;

The result of the operation must be seen in all other windows open for this account with appropriate exchange rate modifications so that all open windows show consistent account state.

On pressing "Withdraw" if the editable textfield contains a positive number greater than 1 and there are sufficient funds then that amount is subtracted from the account amount; the result of the operation must be seen in all other windows open for this account with appropriate exchange rate modifications. If funds are insufficient then an exception must be raised by some method from a class in the model package that will eventually is caught by the controller and the controller must request the view to open a pop-up window that contains message "Insufficient funds: amount to withdraw is x, it is greater than available

funds: y". The pop-up window contains button "Dismiss" on pressing which the pop-up window should close.

After pressing the "Deposit" or "Withdraw" button the content of the Enter amount textfield should be reset to 0.0

The system should have other exceptions as needed (e.g. in response to corrupted or inconsistent content of the file with accounts; e.g. inadmissible characters in the fields of an account entry; broken format of the file). Exceptions should result in useful messages via dismissable pop-up windows. In case of file inconsistencies messages should mention name of the file, line number where inconsistency occurred, nature of inconsistency, suggestion to fix (e.g. "Name must have only letters" or "Amount must not be negative"). If an error is unrecoverable the system should exit on dismissing the pop up window.

Add buttons "Create deposit agent" and "Create withdraw agent" to the main frame (window) of the application.

On pressing one of these buttons a window "Start deposit/withdraw agent for account: <accountID>" should appear. The accountID should correspond to the account selected in the drop-down list of the main window when a "Create … agent" button was pressed.

A "Start … agent …" window should contain:

- Editable textfield "Agent ID"
- Editable textfield "Amount in $"
- Editable textfield "Operations per second" initialized to 0.0, it accepts only digits and a decimal point
- Button "Start agent"
- Button "Dismiss"

"Agent ID" textfield should allow unique IDs for the set of currently running agents.

If the button "Start agent" is pressed while the input is insufficient or improper, an informative dismissable pop-up window should appear. The "Start … agent …" window should stay on.

If the button "Start agent" is pressed while all the input is properly entered then the "Start … agent …" window should be replaced with a corresponding "Deposit agent <agentID> for account <accountID>" window or "Withdraw agent <agentID> for account <accountID>"and an agent should start running and the "State" textfield should show "Running".

A "<Deposit/Withdraw> agent <agentID> for account <accountID>" window should contain:

- Immutable textfields "Amount in $" and "Operations per second" with the proper info
- Uneditable textfield "State" (proper values: Running, Blocked, Stopped)
- Uneditable textfield "Amount in $ transferred" that reflects the amount transferred by this agent (either deposited or withdrawn"
- Uneditable textfield "Operations completed" with a proper count
- Button "Stop agent"
- Button "Dismiss agent"

Button "Dismiss agent" should be grayed out before button "Stop agent" is pressed.
On pressing button "Stop agent" the agent should be stopped with the window staying on. The "State" textfield should show "Stopped".
If button "Dismiss agent" is pressed after button "Stop agent" has been pressed then this window should close.

A withdraw agent should block if the amount in the account would become negative as a result of the agent's next operation. A "Withdraw agent <agentID> for account <accountID>" window should show "Blocked" in its "State" textfield.

In this manner it should be possible to keep multiple windows open for the same or different accounts in the same or different currencies.

The exchange rates should be hardwired as constants (public final static double) as:
1 $ = 0.79 Euro
1 $ = 94.1 Yen

The software system ought to be designed according to the object oriented approach.

Comments should be provided for each class, important methods and important code portions inside the methods. Comments for classes and methods should be in javadoc format so that to produce a simple API documentation automatically.

JUnit testcases should be provided for each method in classes in model and controller packages (as needed for classes in the view package).

Process

1. Create a UML class diagram with classes, abstract classes and interfaces related by generalization, association and other relationships as needed and packaged (model, view, controller). The classes should contain your first take at attributes and operations (that should include constructors and at least one main method). The diagram should contain interfaces, abstract classes and classes that define an MVC architecture (i.e. interfaces Model, View, Controller, ModelListener; class ModelEvent (or other Events); abstract classes AbstractModel, AbstractController) in addition to your model specific classes. You can update your classes and interfaces in the code later and reverse engineer a class diagram consistent with your code. The classes should include some initial set of exceptions and operation definitions should mention if they throw them.
2. Use the class diagram to create a skeleton of the source code (start from the model package including interfaces and classes that define an MVC then the rest of the controller and view interfaces and classes; your first iteration of the model package should be relatively complete before proceeding to the rest of the controller and view)
3. Incrementally edit the source code to include the rest of attribute and operation declarations (NOT implementations yet) with proper javadoc comments for classes, attributes and operations. Focus on usefulness of API (the kinds of operations and their formal parameter lists, return types, exceptions). As you create additional operation declarations think about contingencies that may happen when executing an operation and create corresponding exception classes and specify that an operation throws them. Also, add stubs into operations that throw exceptions so that this code (without implementations) would compile.
4. Make sure your code compiles while doing the increments (nothing to run yet because it only contains declarations and stubs).
5. Produce an initial javadoc documentation (with all scopes of visibility allowed)
6. Reverse engineer the code in the current state into a UML class diagram
7. Analyze the consistent class diagram and javadoc docs and make changes to the code if necessary. Iterate steps 4,5,6,7 until satisfied with your initial model
8. Identify an initial set of methods to implement; write JUnit testcases for them; then implementations; test them and iterate until satisfied
9. Proceed in increments (write test; write implementation; test the implementation; change javadoc comments if necessary) until you implement the functionality of the

model package first. You should be able to test all the functionality of the classes in the model package without controller and view classes
10. Proceed in increments to cover the rest of the functionality
11. Conduct acceptance tests; fix faults if found.
12. Reverse engineer the code to produce a consistent UML class diagram
13. Create a final javadoc

Submission

Turn in the assignment electronically to the TRACS drop box.

The files of the problem should be archived into one archive file named accountManager<your initials>. The archive should preserve the directory structure starting from the root directory of the software system (directory named accountManager).

Classes should be in packages accountManager.model, accountManager.view, accountManager.controller (placed according to the MVC architecture). If needed there can be a package accountManager.util in addition to the ones already mentioned.

The archive file should contain:

1. Description of acceptance testcases
2. Description of execution of acceptance testcases illustrated with screenshots of all the windows and pop-up windows of the system along an acceptance testcase
3. UML class diagram for the software system
4. Textual description of data structures in the model package.
5. Source code (archive of directory structure starting from accountManager dir)
6. API docs produced by javadoc
7. JUnit testcases in the code (for the Model and Controller classes and their methods)