INB370 / INN370 Software Development

Assignment 1: Object-Oriented Programming and Unit Testing

Semester 1, 2010

Due date: Friday, 26th of March, 2010 (end of Week 5)
Weighting: 20%
Assessment type: Individual assignment

Learning Objectives

- To demonstrate your ability to interpret object-oriented API documentation.
- To develop your skills in writing classes in an object-oriented programming language.
- To develop your skills in designing a suite of unit tests.
- To demonstrate your ability to develop program code that interfaces with code written by others.

The Scenario — Managing a Warehouse

Warehouses are used to smooth the flow of goods between manufacturers and retailers. (In this sense they serve exactly the same purpose as data buffers in ICT systems.) Managing a warehouse is challenging because it involves reconciling two contradictory requirements. On the one hand, company management wants to minimise the amount of stock on the shelves because the warehouse itself is

expensive to maintain and idle stock is an unproductive asset. On the other hand, the warehouse manager wants to keep the warehouse full to ensure that all incoming orders from retailers can be met.

In fact, managing a warehouse successfully is an example of a general class of mathematical problem known as an 'optimal stopping problem'[1]. The challenge is to optimise the values of certain variables (e.g., minimising the amount of warehouse stock on the shelves and maximising profits) despite one of the variables having an unpredictable behaviour (e.g., the number of items that will be ordered from the warehouse on a given day).

For the purposes of this assignment we will assume that you have accepted a fixed-term appointment as a warehouse manager. To do this you maintain a ledger which shows the current stock levels and cash reserve. Your overall goal is to increase the cash reserve, by buying items at a low wholesale cost and selling them to retailers at a higher price, while keeping stock levels as low as possible, to minimise overheads. Each morning you have to decide whether or not to restock the warehouse. If you choose to restock the warehouse you must pay not only the wholesale cost of the items bought, but a fixed delivery charge. Restocking the warehouse too often thus runs the risk of going bankrupt because the delivery charges may outweigh the profits made by selling items to retailers. However, if you don't restock the warehouse often enough you run the risk of not having enough items to satisfy daily orders from retailers. To avoid being fired you therefore have to find a delicate balance between these two extremes!

The Challenge — Completing a Warehouse Simulator

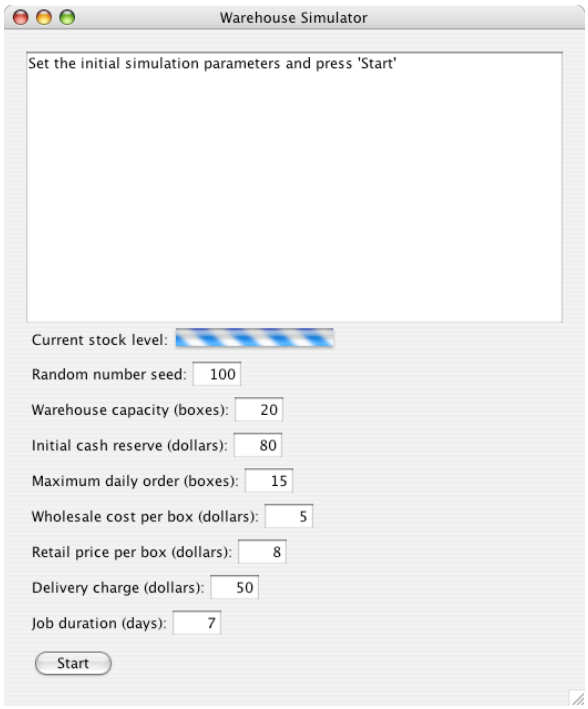
In this assignment you will complete the implementation of a simulator that allows this warehouse managment scenario to be explored with different values for the many variables involved.

- You will be supplied with a Graphical User Interface for running the simulation, and you must develop two additional classes necessary to make the simulation work. (Developing software that must interface properly with code produced by other programmers is a typical situation in large-scale software development.)
- The necessary functionality of the classes to be produced is described in a 'Javadoc' Application Programming Interface specification. (Being able to understand such documentation is an essential skill in software engineering.)
- In addition, you must develop comprehensive test suites for the two classes you produce. (In large-scale software development, test suites are essential not only for convincing yourself and others that your program works, but also for helping manage future changes to your code.)

When you have completed the tasks for the assignment, you will be able to test the complete simulator by adding your code to the provided GUI. The complete simulator provides a game-like environment which begins with a screen like the one shown overleaf, to allow you to enter initial values for the many variables involved in the simulation.

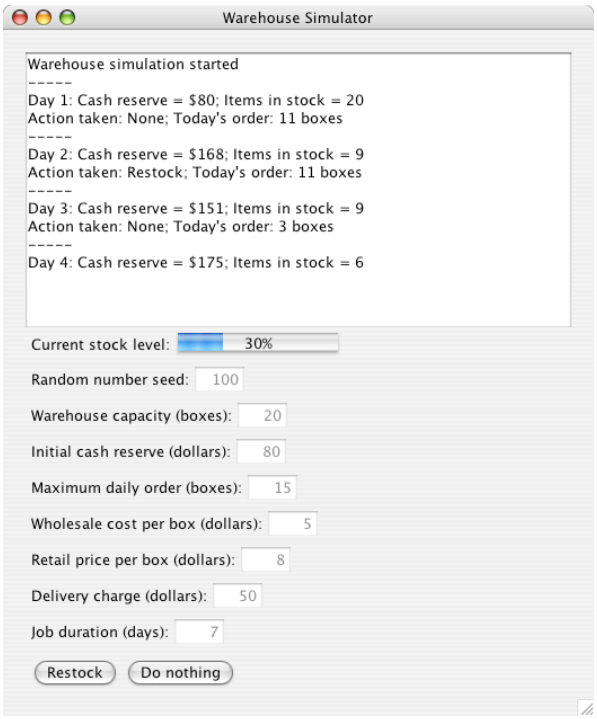
---

[1] Thomas S. Ferguson, *Optimal Stopping and Applications*, `http://www.math.ucla.edu/~tom/Stopping/Contents`, accessed 1 March 2010



The 'random number seed' is used to control the generation of daily orders from retailers. The same seed will always produce the same sequence of pseudo-random orders, allowing you to test the whole system deterministically. The other variables should be self-explanatory.

Once the `Start` button is pressed the simulation variables are all fixed and the simulation begins. The simulator displays the state of the ledger at the beginning of each day and then allows the user to select one of two actions, `Restock`, which will restock the warehouse to capacity, using the cash reserve to pay for the new stock and the delivery charge, or `Do nothing`, which does not buy any new stock. The simulation then tries to satisfy a random retail order and advances to the next day.



The simulation ends in one of three ways:

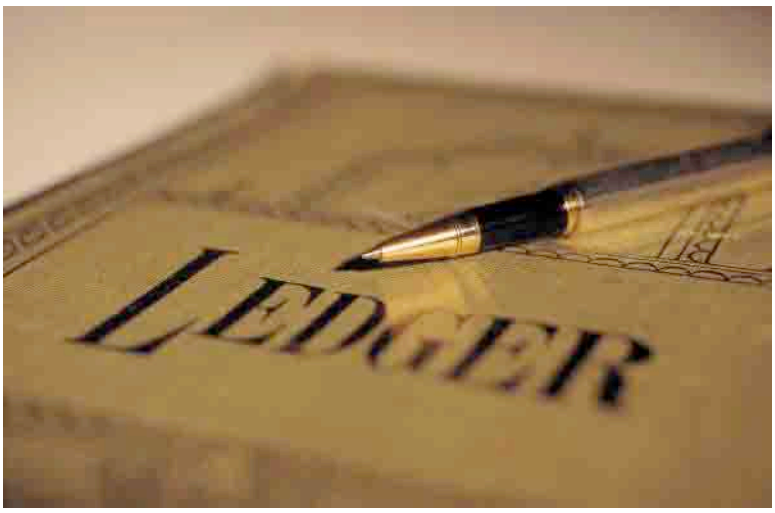
- If the cash reserve is negative at the end of a day you are fired for sending the business bankrupt. (For simplicity we ignore the potential value of the remaining stock on the shelves.)
- If an order is received that cannot be satisfied in full, because there is insufficient stock in the warehouse, you are fired for letting down the company's retail customers.
- If the end of your fixed-term appointment is reached, without the warehouse going bankrupt or failing its customers, you have succeeded and receive your payment!

Specific Tasks for Completing the Assignment

To complete the assignment you must finish four programming tasks. Begin by placing the supplied package `asgn1Question` in a new Eclipse project. This package contains the GUI simulation code, including the interfaces that you must implement and an exception class that you must use. In particular, class `SimulationComponents` relies on the two classes that you are required to write, and class `Simulation` contains the `main` method that you must run to start the simulation (once your own code is complete).

Also supplied is a Javadoc API specification that describes the required functionality of the two classes you must implement. To see it, open the `index` file in the supplied folder `doc` in an HTML browser.

All of the program and unit test code you produce must be placed in a package called `asgn1Solution`, in the same Java project as package `asgn1Question`.



Task 1: Implementing a `WarehouseLedger` class

In the materials provided with these instructions you will find program code for a Java interface called `Ledger` and its API specification in Javadoc format. This interface specifies the characteristics of a class which maintains a ledger, i.e., a record of a company's commercial transactions. The warehouse simulation GUI relies on this class to keep track of the warehouse business's state.

Your first task is to implement this interface as a concrete class called `WarehouseLedger`. In doing so you should examine the code in the provided class `SimulationComponents` to see how your `WarehouseLedger` class will be used.

You must implement the class to match the API documentation precisely, including throwing appropriate exceptions with meaningful messages for invalid inputs. Exception class `WarehouseException` is provided in package `asgn1-Question` for this purpose.

Task 2: Developing Unit Tests for the `WarehouseLedger` class

In addition to developing your `WarehouseLedger` class you must develop a set of JUnit tests for it, to be placed in a file named `LedgerTest.java`. These tests must comprehensively exercise all of the `WarehouseLedger` class's methods, including normal, exceptional and boundary cases.

Task 3: Implementing a `WarehouseTransactions` class

In the materials provided you will find program code for a Java interface called `Transactions` and its API specification in Javadoc format. This interface specifies the operations needed by the simulation GUI to simulate the daily transactions of a warehousing business, including ordering new wholesale deliveries and selling stock to retailers. In particular, pressing the buttons in the GUI causes the specified methods to be called.

Your next task is to implement this interface as a concrete class called `WarehouseTransactions`. In doing so you should examine the code in the provided class `SimulationComponents` to see how your `Warehouse-Transactions` class will be used. In particular, note that the `Warehouse-Transactions` class's constructor relies on the existence of a `Warehouse-Ledger` class.

Again you must implement the class to match the API documentation precisely, including throwing appropriate exceptions with meaningful messages.

Task 4: Developing Unit Tests for the `WarehouseTransactions` class

Finally, you must develop a set of unit tests for your `WarehouseTransactions` class, to be placed in a file named `TransactionsTest.java`. These tests must comprehensively exercise all of the `WarehouseTransactions` class's methods, including normal, exceptional and boundary cases. When doing so, keep in mind that the `WarehouseTransactions` class relies on the `WarehouseLedger` class.

Academic Integrity

This is an individual assignment. Please read and follow the guidelines in QUT's *Academic Integrity Kit*, which is available from the INB370 Blackboard site on the `Assessment` page. Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (`http://theory.stanford.edu/~aiken/moss/`).

Assessment

Submitted assignments will be tested automatically, so you must adhere precisely to the specifications in these instructions and on Blackboard. Your `WarehouseLedger` and `WarehouseTransactions` classes will be unit tested against our own test suite to ensure that they have the necessary functionality. Your `LedgerTest` and `TransactionsTest` classes will be exercised on defective programs to ensure that they adequately detect programming errors.

Apart from the quantitative results of these tests, the quality of your code will also be considered, so all four of your classes must be presented to a professional standard. Further detail about assessment criteria appears below.

Submitting Your Assignment

You must submit your completed `asgn1Solution` package, containing the Java classes `WarehouseLedger`, `WarehouseTransactions`, `LedgerTest` and `TransactionsTest`, by the deadline. Solutions will be submitted via the Online Assessment System (`http://www.scitech.qut.edu.au/study/current/oas/`). Full details of required file formats for submissions will appear on Blackboard near the deadline.

You must submit your solution before midnight on the due date to avoid incurring a late penalty. You should take into account the fact that the network might be slow or temporarily unavailable when you try to submit. **Network problems near the deadline will not be accepted as an excuse for late assignments.** To be sure of avoiding a late penalty, submit your solution well before the deadline.

The following late penalties will apply. One day late, 10% of the mark given is deducted; Two days late, 20% of the mark given is deducted; Three to four days late, 30% of the mark given is deducted; Five to seven days late, 40% of the mark given is deducted; Eight to ten days late, 50% of the mark given is deducted; Greater than ten days late, 100% of the mark given is deducted.

Assessment Criteria — Program Code (classes `WarehouseLedger` *and* `WarehouseTransactions`*)*

	Excellent (Grade 7)	**Good (Grade 6–5)**	**Average (Grade 4)**	**Unsatisfactory (Grade 3–1)**
Functionality of program code (7 marks)	The classes pass all, or nearly all, of our unit tests. The classes interface properly to the provided GUI. **7 marks**	The classes fail a significant number of our 'extreme' unit tests, although they pass most 'normal' cases. The classes interface properly to the provided GUI. **6–5 marks**	The classes fail a large number of our unit tests, including some 'normal' cases. The classes interface properly to the provided GUI. **4 marks**	The classes are largely incomplete or fail most of our unit tests. The classes do not interface properly to the provided GUI. **3–0 marks**
Presentation of program code (4 marks)	The program code is presented to a professional standard, with self-explanatory identifiers, and clear, concise commenting applied only where needed. Messages in thrown exceptions describe the problem clearly and concisely. **4 marks**	The program code is generally well-presented, but a few identifiers are cryptic, or the level of commenting is inapproprate in places (insufficient or merely repeating the code). Most exception messages are self-explanatory. **4–3 marks**	The code is reasonably well-presented, but: some identifiers are cryptic; some code is unnecessarily complex; or there is insufficient or unhelpful commenting. Some exception messages are obscure. **3 marks**	The code is poorly presented and hard to understand because: identifiers are not self-explanatory; the code is much more complex than necessary; comments are largely absent or incomprehensible; or messages for thrown exceptions are unclear. **2–0 marks**

Assessment Criteria — Unit Tests (classes `LedgerTest` *and* `TransactionsTest`*)*

	Excellent (Grade 7)	**Good (Grade 6–5)**	**Average (Grade 4)**	**Unsatisfactory (Grade 3–1)**
Effectiveness of unit tests (6 marks)	The unit tests detect all, or nearly all, of the errors in our test programs. **6 marks**	The unit tests miss several of the more obscure errors in our test programs. **5–4 marks**	The unit tests miss many of the more obscure errors in our test programs and some obvious ones. **4 marks**	The unit tests miss a large number of the errors in our test programs, including many obvious ones. **3–0 marks**
Presentation and design of unit tests (3 marks)	The unit tests are well-designed, each with a single clear purpose, which is obvious from their names and/or commenting as necessary. The tests are designed to work independently of one another. **3 marks**	The purpose of a few tests is not immediately obvious, or the test suite appears disorganised in places. Most tests are independent of one another. **3–2 marks**	The purpose of several tests is not clear from their names or comments. Some tests are dependent on previous ones, which may obscure the reason for failing a test. **2–1 marks**	The test suite's organisation is unclear. Many tests are inter-dependent. The purpose of many tests is not obvious from comments or the choice of identifiers. **1–0 marks**