INB370 / INN370 Software Development
Assignment 2: Test-Driven Development and Graphical User Interface Programming

Part 1: Test-Driven Development

Semester 1, 2010

Due date: Friday, 21st of May, 2010 (end of Week 12)
Weighting: Part 1: 19%; Part 2: 11%
Assessment type: Group assignment, working in pairs (either both INB370 or both INN370 students)

Learning Objectives for the Assignment

- To experience team-based program development using an approach similar to test-driven development (covered in this part of the assignment).
- To practice simple Graphical User Interface programming (covered in Part 2 of the assignment).

The Scenario — Managing Cargo on a Container Ship

Intermodal freight containers are used to transport manufactured goods all around the world. In this assignment your team will play the role of a small software company which has been commissioned by a major shipping firm to develop a program to help them manage the mixed cargo carried on the decks of their container ships.

The shipping containers are loaded and unloaded via an overhead crane. Before a voyage begins, the ship's captain decides how much weight the ship can carry, how many stacks of containers there will be on deck, how high the stacks may be, and any other constraints on the loading and unloading of containers. Each container has a unique code to identify it, consistent with international standards.

The ship's captain needs to monitor loading and unloading activities from the bridge, but the view of activities on the deck is restricted from this vantage point. The program you are required to develop will form part of an automated system which will help the captain see the state of the ship's cargo, without the need to go down on deck.

The Challenge — Implementing a Cargo Manifest

In this first part of the assignment you will complete the implementation of some Java packages that allow you to maintain a ship's cargo manifest, i.e., a record of the cargo containers currently on board. (In Part 2 of the assignment you will provide a Graphical User Interface to allow the ship's captain to manipulate the manifest.)

There are three main parts to the manifest's implementation.

1. Valid container codes are defined by an international standard, ISO 6346, so a class is needed to represent valid codes.
2. Freight containers come in different types, each with different characteristics, so a class hierarchy is needed to model the different kinds of containers.
3. The manifest itself is a record of the containers currently on board, so a class is needed which allows containers to be loaded, unloaded and located.

An outline of the tasks required to complete this part of the assignment is given below. Details of the technical requirements for each of the classes is given in the Javadoc API specification accompanying these instructions.

Working as a Team

Professional large-scale software development inevitably involves working in a team. For the purposes of this assignment you are required to work in pairs and follow the 'agile' programming practices discussed in the lectures so far.

1. When developing the classes for container codes and freight containers, one team member must take the role of the *tester*, and the other must take the role of the *coder*. The tester will develop new unit test cases, while the coder must respond with program code that passes the tests.
2. When developing the class for the ship's manifest, these roles must be reversed, so that both team members get to play both roles.

NB: You are required to used Javadoc '`@author`' comments before each of the program code and unit test classes to identify who played which role. Include both your name and student number.

Producing Professional Quality Code

The provided Javadoc API specification clearly describes the necessary packages, classes, methods, parameters and return types needed to complete this part of the assignment. As a professional programmer, you must follow these specfications *precisely*. (In computer programming 'near enough' is *not* 'good enough'!)

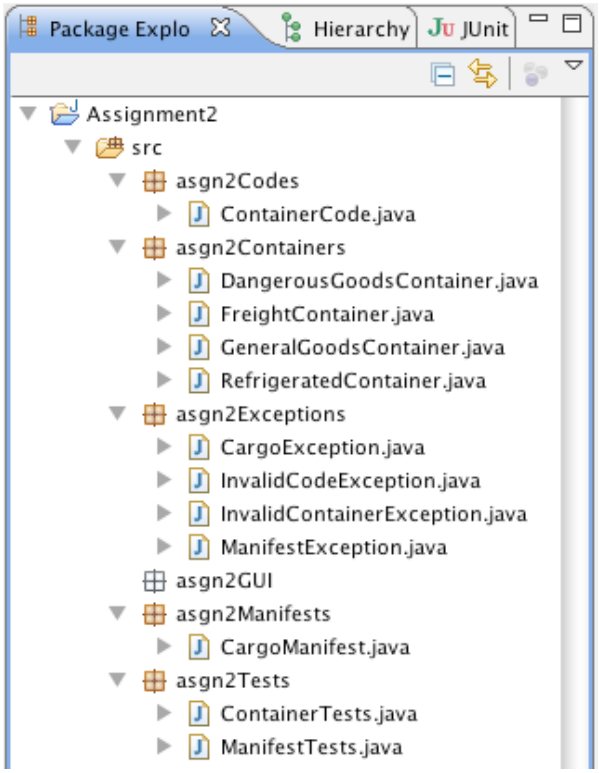
In addition, you must submit your assignment in the specified zip archive format. More detail on the required format will be released close to the deadline.

Since we have already provided Javadoc-style documentation for this part of the assignment you are not required to duplicate these comments in your code. However, you must properly comment any code of your own, especially private utility methods that are not part of the specified API.

Your code should also be presented in a professional style. Following a recognised coding convention, such as that described in the *Code Conventions for the Java Programming Language*[1], is recommended.

Specific Tasks for Completing Part 1 of the Assignment

To complete this part of the assignment your team must finish the following programming tasks. Full technical specifications for all the program code that must be developed are given in the Javadoc API description accompanying these instructions. Details of the process to be followed and the unit tests to be produced appear in the sections below.

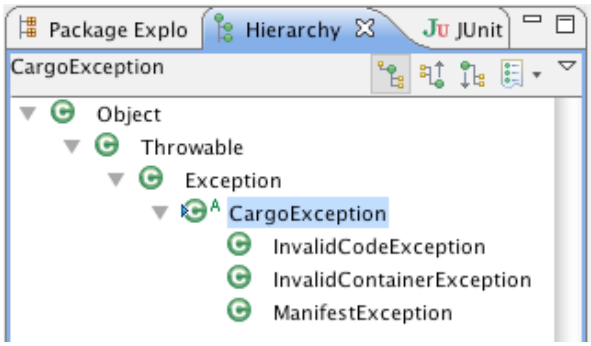


[1] http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

The overall Java project comprises several packages as shown in the folder hierarchy on the previous page. Ensure that you follow these package and class names precisely (and do not introduce any classes other than those shown here).

Task 1: Implementing an exception class hierarchy

There are three main program code packages to be developed in this part of the assignment, for container codes, freight containers, and ship's manifests, respectively. In Part 2 of the assignment, when you are developing the GUI, it will be helpful to know which of these parts of the program generated an exception. Therefore, we begin by implementing an exception class *hierarchy*, with distinct exception types for the three main parts of the program. In this first task your team must implement package `asgn2Exceptions` which contains a type hierarchy rooted at abstract class `CargoException` as follows.



See the Javadoc API specification for details of each class. This is a simple task—each of the exception classes just consists of a trivial constructor—so it can be completed by either team member.

Task 2a: Implementing a class for container codes

Valid codes used to identify freight containers are defined by international standard ISO 6346, so in this task you will implement package `asgn2Codes`, containing class `ContainerCode` for constructing valid container code objects.

In practice each container code is represented by an 11-character string, following a specific format. Full details can be found in the Javadoc API specification. For instance, the following container has valid code `KOCU8090115`:

Similarly, code `MSCU6639871` on this container is valid in our variant of the standard:

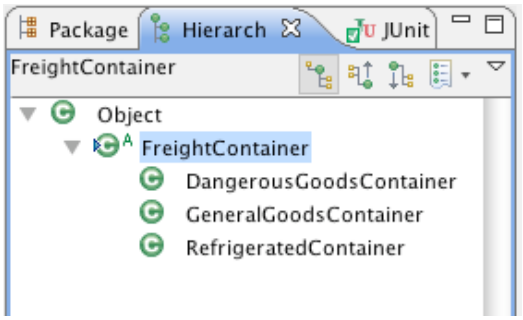
In this task you must adopt a test-driven development-based programming style, with one team member acting as tester and one team member acting as coder. At each step:

- The *tester* must add new unit tests to class `ContainerTests` in package `asgn2Tests`. (Although test-driven development is usually depicted as adding tests one at a time, it is often more practical to add small groups of related tests, especially when trying to cover boundary cases. However, avoid the temptation to add several *unrelated* tests in one step, as this makes the job very difficult for the coder, and violates the TDD philosophy.)
- The *coder* must add code to class `ContainerCode` in package `asgn2Codes` to pass the tests, and must refactor the code appropriately to generalise the solution.

Ensure that both classes contain '`@author`' comments to identify who played which role. (Include both your name and student number.)

Task 2b: Implementing a class hierarchy for freight containers

In practice there are many different types of freight container, so in this task your team will implement a small class hierarchy for different kinds of freight container object:



Each of the three concrete classes at the bottom of the hierarchy defines a different type of freight container. For instance, dangerous goods containers must display a category label identifying the kind of hazard posed by their contents. Full details can be found in the Javadoc API specification.

To complete this hierarchy your team must maintain the same roles as in Task 2a. (Although Task 2b appears complicated, it is actually easier than Task 2a, because each of the individual classes in the hierarchy is trivial.)

- The *tester* must add new unit tests to class `ContainerTests` in package `asgn2Tests` (the same class as used in Task 2a).
- The *coder* must create and add code to classes `FreightContainer`, `GeneralGoodsContainer`, `RefrigeratedContainer` and `DangerousGoodsContainer` in package `asgn2Containers` to pass the tests, and must refactor the code appropriately to generalise the solution.

Ensure that all classes contain '`@author`' comments to identify who played which role. (Include both your name and student number.)

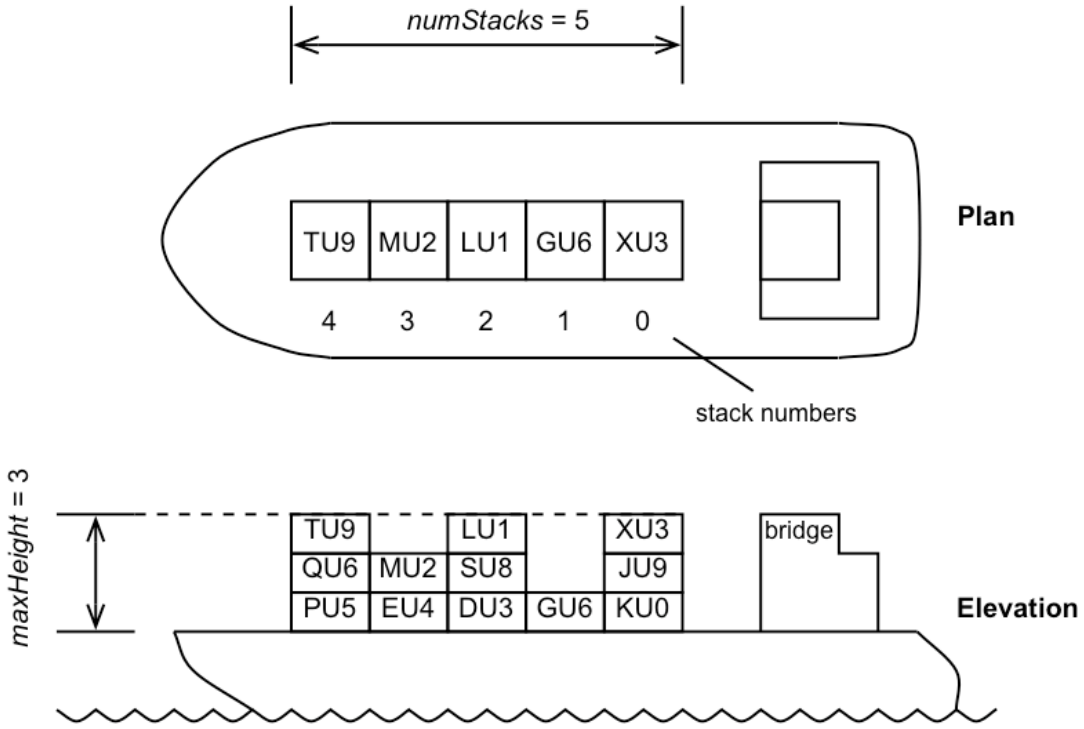
Task 3: Implementing a class for cargo manifests

In this task your team will implement the class for managing cargo manifests. This is the largest and most complex class in this part of the assignment. To complete it you must swap the *tester* and *coder* roles played so far.

For the purposes of the assignment we will use a fairly simple model of a container ship's cargo, in which containers are kept in stacks, in a single file, as shown in the figure on the following page. The stacks are individually numbered, starting with stack zero being the one nearest the bridge.

NB: The container codes shown in this figure are *not* valid codes.

Each voyage is different, so the number of stack spaces on the deck, the maximum allowed height of the stacks, and the total allowed weight of freight containers are all parameters to the ship's manifest. Full details can be found in the Javadoc API specification.



Since cargo containers are loaded and unloaded by an overhead crane, a container can be unloaded only if it is accessible, i.e., on top of a stack. In the figure above, container `MU2` can be unloaded immediately, but container `JU9` cannot.

In addition, the way containers can be loaded is dictated by the ship's captain. Here we assume that the captain has the following requirements, expressed as 'stories' in the agile software development style:

- As a ship's captain I want containers to be loaded as close to the bridge as possible, where I can see them.
- As a ship's captain I want all stacks to hold containers of the same type, because they are likely to all be going to the same destination.
- As a ship's captain I want all stacks to be no higher than the maximum limit for this voyage, for safety.
- As a ship's captain I want a container to be loaded only if doing so will not exceed the maximum cargo weight for this voyage.
- As a ship's captain I want all containers to be stacked in the allotted places on deck only.
- As a ship's captain I want to ensure that all the container codes for freight containers on board are distinct.

For instance, the particular configuration in the figure above is valid if the containers in stacks 0 and 1 are all refrigerated containers, if the containers in stacks 2 and 3 are all general goods containers, and if the containers in stack 4 are all dangerous goods containers. In particular, note that if a new refrigerated container arrives it can be added to stack 1. However, if a new dangerous goods container arrives it cannot be loaded because (a) there are no dangerous goods stacks that are not already at their

maximum height, and (b) there are no further spaces on deck to start a new dangerous goods stack.

To complete this task your team must undertake the following software development roles, remembering that you must swap the roles you had in Tasks 2a and 2b.

- The *tester* must add new unit tests to class `ManifestTests` in package `asgn2Tests`.
- The *coder* must add code to class `CargoManifest` in package `asgn2Manifests` to pass the tests, and must refactor the code appropriately to generalise the solution.

Ensure that both classes contain '`@author`' comments to identify who played which role. (Include both your name and student number.)

Academic Integrity

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 program code classes will be unit tested against our own test suite to ensure that they have the necessary functionality. Your unit test 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 of your classes must be presented to a professional standard. Further detail about assessment criteria appears below.

Submitting Your Assignment

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, Assignment 2, Part 1 — Program Code (packages* `asgn2Exceptions`, `asgn2Codes`, `asgn2Containers` *and* `asgn2Manifests`*)

	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 code precisely matches the specified API. The zip archive matches the specified format. **7 marks**	The classes fail a significant number of our 'extreme' unit tests, although they pass most 'normal' cases. The code precisely matches the specified API. The zip archive matches the specified format. **6–5 marks**	The classes fail a large number of our unit tests, including some 'normal' cases. The code precisely matches the specified API. The zip archive matches the specified format. **4 marks**	The classes are largely incomplete or fail most of our unit tests. The code does not match the specified API, or the zip archive does not match the specified format. **3–0 marks**
Presentation of program code (3 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. Authors of each class are identified. **3 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. Authors of each class are identified. **3–2 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. Authors of each class are identified. **2 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. Authors of the classes are not clearly identified. **1–0 marks**

Assessment Criteria, Assignment 2, Part 1 — Unit Tests (package `asgn2Tests`*)*

	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. Authors of each class are identified. **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. Authors of each class are identified. **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. Authors of each class are identified. **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. Authors of the classes are not identified. **1–0 marks**