CMPT 275 Group T

Momentum Software Engineering's

Flight the Freights Manager
Architectural Design Document
Revision 1

July 5, 2002

Approved by: (By alphabetical order)

Vincent Chu _____
Steven Essen _____
Sung Hwang _____
David Jeong _____

Revision History Page

Revision 1	-Written jointly by David Jeong, Sung Hwang, Steven Essen and Vincent Chu. Compiled together into one document.

Table of Contents

Section 1.0 – Introduction

Air Kanata has requested Momentum Software Engineering develop a system that will manage the air cargo items which are assigned to certain flights. Various operations on the system include adding new customers, new cargo items, and new flights; deleting cargo items and flights; assigning cargo items to flights; de-assigning cargo items from flights; inquiring about the capacity on a flight; and printing flight and its cargo items in a report. As background information to the system, please refer to the Requirements Specification Document and the User Manual for more insight.

The architectural design document (the document you are now reading) talks about how the system will be implemented. This document is necessary to instruct the programmers to program the customer requirements in an efficient, orderly, and accurate manner. Two designs will be presented; the first one (discussed in Section 3) will be used for implementation. Both designs will be discussed and compared. The document also will layout the function calls for each classes and modules as well as the parameters and functionalities. Note that much of the interface design is not located in this document but is available by referring to the actual source code modules in the back of this binder.

Section 2.0 – Refinement of the Analysis Model

2.1 2NF vs 3NF calculation

In this section, 2NF and 3NF calculations are performed. 3NF uses the Aircraft Model class but the 2NF does not. However, 2NF puts the attributes Maximum Weight and Maximum Length from the Aircraft Model into the Flight class.

3NF Case:

Flight:

	Format	Size
<PK> Flight Number	7-10 characters	10 characters x 2 bytes/character = 20 bytes
Aircraft Model Number	1-8 characters	8 characters x 2 bytes/character = 16 bytes
Total per flight record:		36 bytes
31·10 flight records in the system:		310 record x 36 bytes per record = 11160 bytes

Aircraft:

	Format	Size
<PK> Aircraft Model Number	1-8 characters	8 characters x 2 bytes/character = 16 bytes
Maximum Weight	000000.0-999999.9 {digit}^{5}digit{“.”digit}	8 bytes / double
Maximum Length	000.00-199.9 {digit}^{5}digit{“.”digit}	8 bytes / double
Total per model:		32 bytes
6 models in the system:		6 record x 32 bytes per record = 192

In total, assuming there are 31·10 records of flights and 6 models, 3NF system would need 11160 + 192 =11352 bytes of storage.

2NF Case:

	Format	Size
<PK> Flight Number	7-10 characters	10 charactes x 2 bytes/character = 20 bytes
Maximum Weight	000000.0-999999.9 {digit}^{5}digit{“.”digit}	8 bytes / double
Maximum Length	000.00-199.9 {digit}^{5}digit{“.”digit}	8 bytes / double
Total per flight:		36 bytes
31·10 flight records in the system:		310 record x 36 bytes per record = 11160 bytes

In total, assuming there are 31·10 records of flights, 2NF system would need 11160 bytes of storage.

Based on the above results, 2NF takes up less space than 3NF. Since time and trouble of entering/maintaining redundant data is not a problem, 2NF will be used in the implementation of the system.

2.2 Derived Attributes

For each flight instance, the following derived attributes will be added:

- footprints free
- weight remaining

The system will just automatically show values stored in these attributes as results of a flight query. Thus, time is saved in not calculating them every time a user wants to inquire about a flight (especially when assigning a bunch of cargo items to flights).

Section 3.0 – Architecture Presentation and Discussion

There are thirteen collaboration diagrams in total, each of which describes one of the following use case scenarios:

- Adding a new customer
- Adding a standard container
- Adding a pallet
- Creating a new flight
- Assigning cargo items to a flight
- Deleting a cargo item
- Querying the space remaining
- Deleting the flight & its cargo items
- Re-assigning all cargo items
- De-assigning a cargo from a flight
- Printing flight load report

3.1 Introduction

The system employs a "principle object-based scenario" design. As discussed later, the user will first select a desired operation from the user menu. The user interface (UI) module will initiate the appropriate functions that are stored in different classes depending on the selected operation. Those functions, in turn, will call the appropriate classes that are directly below them.

3.2 Name and Purpose of Each Module

The overall object communication diagram is presented on the next page (Figure 3.1). The 8 modules in *Flight the Freights* Manager are:

Modules:	**Purpose:**	**How Cohesion is Exhibited:**
Main Module	**Main module** is called when the program first starts. The main module would initiate the user interface class, the customer class, the flight class, and the cargo class by calling init( ). In turn, the customer, flight, and cargo classes would initialize (i.e. open) the record files by calling open() in the File I/O Class. Main module also contains the procedure for shutting down. When the user quits our system, the shutdown function is called from the main module.	The initialize and shut down procedure are all stored inside the main module.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

User Interface Class	**User interface module** contains the entire user menu. This is where the user state machine code is stored. However, the individual prompting (e.g. "Please enter the customer name.") would be done in the customer, flight, and cargo classes. The user interface module initiates the "scenario-starting" module. It starts each scenario (when the user has selected an option) by calling the appropriate functions that are stored in one of the three classes – customer class, flight class, or cargo class.	The entire user menu state machine code is stored in this single class. This allows easy maintenance. Also, all the scenarios are started by this one module. However, the individual procedure (program code) for each scenario is stored in different classes according to the nature of the operation. This makes the role of the user interface class conceptually clear, as it does not contain detailed code for all the operation.
Customer Class	**Customer class,** as its name implies, stores all the operations required in the "Cargo Maintenance" sub-menu in the user menu. It also contains the attributes that a customer object would store. It has access the functions provided by the File I/O Class for customer Ccass.	All the required functions for maintaining a customer are put into one place.
Flight Class	**Flight class** would store all the operations required in the "Flight Maintenance" sub-menu. It contains all the attributes that a flight object would store. It can access the functions provided by the File I/O Class for flight class.	All the required functions for maintaining flights are put into one place.
Cargo Class	**Cargo Class**, similarly, would store all the operations required for maintaining cargos. It can also access the functions provided by the File I/O Class for cargo class.	All the required functions for maintaining cargos are put into one place.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

File I/O class for Customer Class	**File I/O Class for customer class** uses the appropriate functions provided by the system to allow easy random access to manipulate the customer records. This class only allows the customer class to access its functions.	All the required functions for creating/deleting/modifying customer records are put into one place.
File I/O class for Flight Class	**File I/O Class for flight class** has the purpose similar to the File I/O Class for customer class, but this time it is for the flight class	All the required functions for creating/deleting/modifying flight records are put into one place.
File I/O class for Cargo Class	**File I/O Class for flight class** has the purpose similar to the File I/O class for customer class, but this time it is for the cargo class	All the required functions for creating/deleting/modifying cargo records are put into one place.

3.2 Functions provided by each module and the abstraction employed

Please note that the empty brackets used in the table below are there merely to indicate that those are functions. This does not imply that the functions do not have any parameters.

Modules:	**Functions Provided:**	**Abstraction:**
Main Module	init() shutdown()	
User Interface Class	init() shutdown()	Control Abstraction: The User Interface class calls the appropriate class(es) to complete an operation. It does not concern itself with how that operation is completed.
Customer Class	addNewCust() init() isExist() shutdown()	Data Abstraction: For classes that use this class, the way the attributes are stored are hidden. Also, this class only exports functions that will safely manipulate a customer object.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Flight Class	calculate() createFlight() decCapRemain() deletFlight() getLargestPallet() incCapRemain() init() inquiry() isExist() listFlight() printReport() reassignCargo() shutdown()	Data Abstraction: Same as above. This class handles the flight objects however.
Cargo Class	addContainer() addPallet() assignCargo() deassignCargo() deleteAllCargosOnFlight() deleteCargoItem() getCargoOnFlight() init() isExist() isFlightNull() listCargoForFlight() listUnassigned() shutdown()	Data Abstraction: Same as above. This class handles the cargo objects however.
File I/O Class for Customer Class	close() getCurrentID() getNextFlight() open() reset() wriFlight()	Control Abstraction: Only this class knows the exact binary format of the file that stores the customer records. The customer class (the only class that has access to this class's functions) only calls the appropriate functions exported by this class.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

File I/O Class for Flight Class	close() getCurrentID() getNextCustomer() open() reset() wriCustomer()	Control Abstraction: Same as above. This class handles the file for the flight records however.
File I/O Class for Cargo Class	close() getCurrentID() getNextCargo() open() reset() wriPrevCargo()	Control Abstraction: Same as above. This class handles the file for the cargo records however.

3.3 General Discussion of the Design

As stated previously, *Momentum Software Engineering* has decided to use "Principle Object-based Scenario" Design for the *Flight the Freights Manager*. In this section, the justification of this overall design is made. This will subsequently also justify the rejection of other detailed design Momentum has also prepared.

The user menu is organized into submenu – each submenu provides operations to perform "maintenance" of different types of objects. Without the loss of generality, the cargo class will be discussed in terms of what Momentum has done – these design decisions would also apply to the flight class and customer class.

3.3.1 Comparison with other possible main types of design

Naturally, it would be conceptually clear to have the cargo class know about all the maintenance functions for a cargo item. More importantly, in order to perform a specific function on cargo, most of its attributes (e.g. cargo size, who the cargo belongs to, etc) would be needed. A good design should encapsulate its data attributes and declare them private unless there is reason to do otherwise. This is the main justification for using "Principle Object-based Scenario" design.

Momentum has considered the centralized scenario design. However, if Momentum were to use such strategy, it would face some overriding disadvantages that we will discuss in the next section.

Momentum has also considered round-about design. The major advantage of round-about design is that it can use asynchronous one-way messages – something that is very suitable for distributed programming. However, Momentum is not access different functions that are stored in different computers, and there is no real standing point for trying to reduce the amount of time taken to return values to callers. On the other hand, using this design strategy would make the "user-prompting" cumbersome, and some additional functions need to be made so that the control would travel in a "roundabout"

manner. Since the advantage of using this approach is not significant in the context of the system, Momentum has chosen not to employ this strategy.

3.3.2 Weak Coupling

As a consequence of the choice of design strategy, the User Interface Module does not have to control the sequencing of every single use case scenario. This means that the User Interface Module would not need and therefore would not have access to the retained data in the system. This achieves an important aspect of abstraction, and there are weak couplings between modules.

3.3.3 Fanning in and classes have access to only functions they need

In the design, Momentum separates out the file handling classes not only to allow easy maintenance, but also to achieve data abstraction as mentioned previously (Refer to section 3.2).

3.3.4 Menu State Machine Code

Momentum has chosen to put the menu state machine code in the user interface module. This is the main purpose of the module. Although Momentum has chosen a way to organize the menu that is best suited for the purpose, Momentum can still change it easily (achieve the purpose of maintenance). It is extremely easy if the code is put all in once place. There is little justification for putting the submenus into each separate class. That would be done if the submenu needs to know about something that depends heavily on the individual class. (i.e. if the submenu were to change its look depending on the data records already stored in the system.) Furthermore, as mentioned previously, Momentum wants to make the User Interface Module the source of initiating every use case scenario.

3.3.5 Initialization and Shut Down

The main module will handle the initialization and shut down. When the user runs *Flight the Freight Manager*, main() would be called. main() contains the code for initializing the customer, flight and cargo Class. Because those are the only classes that have access to File I/O Class for Customer, File I/O Class for Flight and File I/O Class for Cargo respectively, they will be responsible for calling initialization functions. (i.e. open()).

Similarly, when the user chooses the shut down function from the user menu, the user interface class would call the shut down function of the main module. The cargo class, for example, would close the File I/O Class for Cargo (i.e. closing the record file). The same propagation applies to the flight and customer class. Finally, the main module will shut down the User Interface Module, and the user will completely quit the system.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Section 4.0 – Comparison with Alternate Design

The alternate OCD diagram ("centralized scenario diagram") is shown on the next page, illustrating a different design from which Momentum is going to adopt but which actually could have been used.

The rationale behind the alternate design is that the user interface (UI) module does all the work in getting the input needed for all the operations/scenarios to be performed instead in the customer, flight, and cargo modules. Thus, the UI is like the scenario orchestrator that knows what functions are to be called and in what order as well as how to handle low-level exceptions. So, this type of design centralizes control of the scenario in the UI module (the event detector).

4.1 Advantages and Disadvantages of Alternate Design

The advantages and disadvantages of the alternate design (compared to the final design) are shown in this section.

Advantages:

- highly cohesive – control and sequencing of internal calls needed to do the processing are encapsulated in one function of one module
- maintenance is easier without affecting other modules. If a control or scenario needs future changes, only one function in one module needs to be updated.
- user I/O is concentrated in one module. Therefore, less work is required in the customer, flight, and cargo modules. Operations are performed in a "centralized" module/class (i.e. flight, customer, or cargo) other than the UI
- less function calls are made from one customer instance to another, flight to flight, and cargo to cargo.
- same types of operation are used in this design as in the final design – the difference is who is making the function calls.
- UI will still contain the main menu but also maintain the submenus
- Initialization and shut down are still handled by the main module

Disadvantages:

- more function calls are made from the UI to the customer, flight, and cargo modules
- centralized UI is strongly coupled with the customer, flight, and cargo modules. So the UI shares many detailed assumptions about the other modules and these modules must rely on the services of mainly the UI.
- encapsulation – since they are not closely related to the UI, some functions should be done in customer, flight, or cargo modules; with the alternative design, note that the UI may have to access the attributes in the other modules.

- information hiding is not really preserved – the internal details and complexities of the module are not necessarily shown to other modules; only the function prototype and parameters are known as well as certain assumptions about the other modules are needed to carry out the scenarios.

4.2 The Use of the Final Design and Not the Alternate Design

Momentum feels that the final design's advantages outweigh the alternate design's advantages. There is no need to group function calls to the other modules in the UI since each class/module knows it functions better than the UI. The final design encapsulates its data attributes and declares them private. Therefore, weakly coupling is considered good as it will promote easy re-arrangement and code re-use; the UI does not have to control the sequencing of every single use-case scenario. Basically, the decentralized scenario initiator (UI) first informs the principal application object involved in the scenario and then this object does whatever is necessary to fulfil the requests (such as checking for the existence of a customer, flight, or cargo item). A return value is required to sent back to the principal object; this is good since the principal object would have information needed to do the next task. Information hiding is present as one module does not know much about what goes on in other modules and there are few shared data types and structures; thus, each module knows what to do on its own.

Section 5.0 – Initial Interface Design

The only constants that will be used are the length and width of the standard (cargo) containers. These will be stored in the cargo class. They can not be avoided because the containers do not have customized sizes like the pallets and therefore, since the container's length and width are already known beforehand, the two constants must be declared to avoid putting magic numbers (numbers that pop up without any explanation) in the code. Besides the addition of these constants, no other types and global variables are needed at this time.

Section 6.0 – Detailed Interface Design

The detailed interface design is not part of this document. Refer to the main and class modules located in the back of this binder.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Section 7.0 – Test Cases

Three different classes of tests, which will be performed, are described in the next three sub-sections.

Note: It is assumed that after typing in any input, the ENTER/Carriage Return Key is pressed. Anything in Courier 10pt font (e.g. `Cargo Menu`) is screen output and anything in Courier 10pt bold font (e.g. **`Thomas`**) is user input.

7.1 Performance Test
Type: Response time

Description: Check whether the program is able to add a new customer data according to the response time as specified in the requirement specification document.

Preconditions:
1. The program is active and set at the main menu.
2. The customer data that is entered in this test is the new data that does not exist in the system.
3. The given customer data is following:

```
Customer name: Chris McNeal

Phone number: (555) 123-1234
```

Test Procedure:
1) Enter 1 from the main menu.

2) Enter 1 to add a new customer, as shown below.

```
======Customer Maintenance======
1) Add a new customer
2) Delete a customer [MAINTENANCE]
3) View all customers [MAINTENANCE]
0) Exit
Choose an option [0-4] and press ENTER:
1
```

3) Enter the name of the customer to be added.

```
Enter the Customer Name (Length: 30 max.):
Chris McNeal
```

4) Enter the phone number of the customer.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

```
Enter the phone number of the customer (Length: 13 max.):
(555)123-1234
```

5) After successfully finishing step 4, the program will ask as following:

```
Are you sure you want to add the customer Chris McNeal(Y/N)?
```

If you see this, enter Y to confirm.

```
Y
```

6) The program will print the message saying,

```
The new customer is successfully added.
```

Then, the program will ask,

```
Do you wish to add another customer(Y/N)?
```

Answer by entering:

```
N
```

Expected result:

1. After you input the customer name data and press ENTER, the process adding the customer name to the system will take within 3 seconds. For the telephone number data, it will take the same amount of time.
2. Since new customer data is being added, the program should ASK us to input the name and phone number of the customer.
3. To see if the newly inputted customer data is actually in the system, follow preceding steps:

    1) Enter 1 from the main menu.
    2) You should see following screen:

```
   ======Customer Maintenance======
   1) Add a new customer
2) Delete a customer [MAINTENANCE]
3) View all customers [MAINTENANCE]
0) Exit
Choose an option [0-4] and press ENTER:
3
```

    3) The list of all customers will be displayed on the screen as following:

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

```
            ======Customer List======
====Name==========================Phone N.=====
 Chris McNeal                      (555)123-1234
================================================
```

This test is now finished.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

7.2 Stress Test
Type: Invalid data

Description: The weight of a cargo item must be typed in as an integer. This test makes sure that if a weight is typed in as a string of nonnumeric (possibly mixed with numeric) characters, then the program will give a proper error message.

Preconditions:
1. The program is active and set at the main menu.
2. Other environment such as disk space and hardware status must be at a stable condition.
3. The customer **`Isaac Newton`** is already in the system.
4. The input data is **`Thomas`**.

Test procedure:
1) Enter 2 from the main menu.
2) Enter 1 to add a new cargo item.

```
======Cargo Maintenance======
1) Add a new cargo item
2) Assign a cargo item to a flight
3) De-assign a cargo item from a flight
4) Delete a cargo item
5) Reassign all cargo on a flight to a different flight.
6) List out all the unassigned cargo [MAINTENANCE]
0) Exit
Choose an option [0-4] and press ENTER:
1
```

3) Enter 2 to add a pallet to the system.

```
=======Add a new cargo========
1) Standard Container
2) Pallet
0) Cancel
Choose an option [0-2] and press ENTER:
2
```

4) Enter the name of the customer.

```
Enter the Customer Name(Length: 30 max.):
Issac Newton
```

5) Enter the cargo weight as shown below:

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Enter the cargo weight(0.1-999999.9 kg):
Thomas

Expected result:
The program should display the following error message:

```
Invalid data.  Please re-enter a numerical cargo weight.
```

Since the wrong style/format of data for the weight was entered, the program should recognize that the data is invalid and warn the user about this through an error message. The message will ask the user to re-enter the weight in the correct format.

Further test inputs:
Repeat the test with the following inputs for the weight:

```
Enter the cargo weight(0.1-999999.9 kg):
90dave
```

(integer followed by a string of alphabet) and also:

```
Enter the cargo weight(0.1-999999.9 kg):
k103
```

(an alphabet followed by a string of integer).

The results for data above are same as the previous expected result.

This test is now finished.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

7.3 Functional Test

Type: Boundary inputs

Description: To check whether the program will process the customer name whose length is the legal maximum range for the name.

Preconditions:

1. The program is active and set at the main menu.
2. Add a new customer that has a name of 30 characters (make sure that this customer is not already in the system).
3. The customer name to be entered is **`Johnthisisalongname Idontknowi`**.

Test procedure:

1) Enter 1 from the main menu.
2) Enter 1 to add a new customer.

```
======Customer Maintenance======
1) Add a new customer
2) Delete a customer [MAINTENANCE]
3) View all customers [MAINTENANCE]
0) Exit
Choose an option [0-4] and press ENTER:
1
```

3) Enter the name of customer.

```
Enter the Customer Name (Length: 30 max.):
Johnthisisalongname Idontknowi
```

Expected result:

The program should accept the input data and go to the next step.
You should see the following screen:

```
Enter the cargo weight(0.1-999999.9 kg):
```

This test is now finished.

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Appendix A – Source Code Convention

Spelling

- The first letter of a function is lower case, and the first letter of every embedded word in a function name is capitalized

    Eg. deleteAllCargosOnFlight()

- The first letter of a class is capitalized.

    Eg. class Customer

- The first letter of a variable is lower case, and the first letter of every embedded word in a variable name is capitalized.

    Eg. int maxUnit;

-Constants are completely capitalized.

    Eg. const CUSTOMERSIZE = 30;

Punctuation

- "{" is placed on the same line as function header (put a space before the "{"), and "}" is placed on the next line from where function ended. The same rule applies to loops and switches.

```
Eg. function ( int a, int b ) {
        ……
          }
```

-Every operator (mathematical, logical, relational, equal sign, and so on) is surrounded by spaces. This does not apply to splitters such as comma, colon, or semi-colon.

```
Eg. x = y + z;
        if ( a || b )
```

- A space is follows after every comma, colon, and ")".

```
Eg. function (int d, double e) {
```

Indentation

-Function statements and statements inside loops are indented 1 tab space (which equals 4 normal spaces) from each "{".

```
Eg. class Customer {
            for () {
                    ……..
        }
  }
```

School of Computing Science
Simon Fraser University
8888 University Drive
Burnaby, B.C.
Canada. V5A 1S6

Comments

- Each major section of a listing should be separated by a dividing comment line and comments between dividing comment lines start with "*". The dividing line goes before such things as the beginning of constants and types, beginning of each static function, and beginning of each instance function.

Eg. /*----------*/
* ……
/*----------*/

-Each module starts with a comment stating that file name, then has a Revision History section in order of the oldest to most recent.

Eg. /*----------*/
* file name: customer.java
* Jul. 02, 2002 Revision 1 – Original by David Jeong
*
* Purpose:
* ……..
/*----------*/

-A comment about a function is right above the function, and has the following format: For example:

```
/**--------------------
*-The procedure for returning a vector of cargo contained on a flight
@param flightID (in): any cargo that is assigned to the flight with this
*flight ID will be added to the vector and returned
@return Vector: A vector containing all the cargo that is assigned to the
*flight with the specified flightID
==========================================================*/
```

-The short bar at the top indicates the beginning of comments.
-The description of the function follows immediately after the short bar
-The parameters are commented after the comments about the function. Each parameter would occupy a single line. The (in/out) nature of the parameter is also indicated, follow by a brief comment about the parameter. It is labeled with @param to generate javadoc later.
-The return value is commented after the comments about the parameters. The return value would occupy a separate line. It is labeled with @return to generate javadoc later.
-The commenting about a function is ended with a long bar. Immediately after the long bar is the signature of the function.