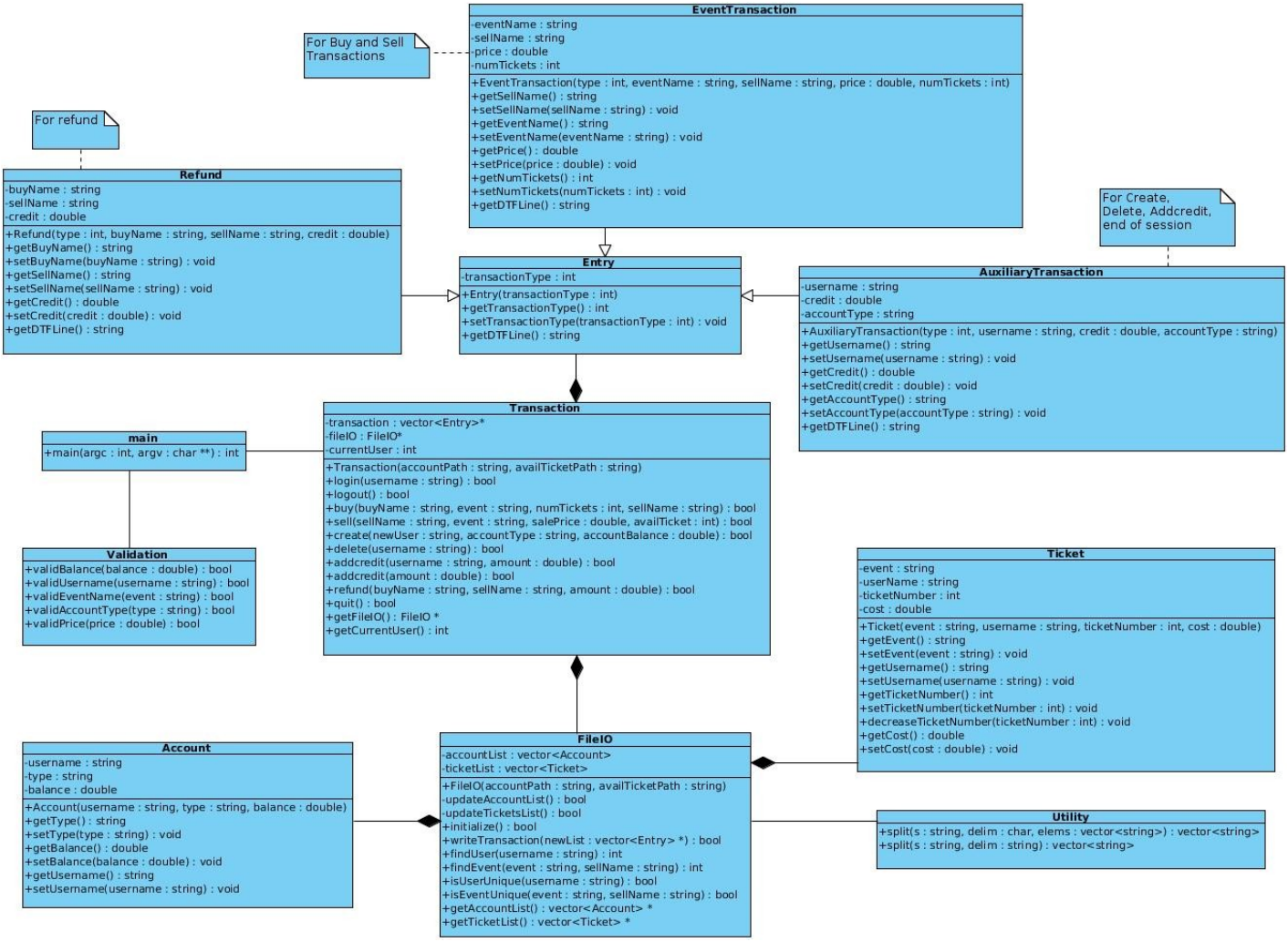
Software Quality

Assignment 2

Joseph Heron 100425488
Khalil Fazal 100425046
Carly Marshall 100426654
Ryan Crawford 100425694

UML Design Document



Ticket Class

Class	Methods	Description
Ticket		Class for holding information involving tickets for sale in the program.
	Ticket(string event, string username, int ticketNumber, double cost)	Constructor for making a ticket listing for an event.
	string getEvent()	Accessor method for getting the event name.
	void setEvent(string event)	Mutator method for the event name.
	getUsername() : string	Accessor method for getting the seller's username.
	void setUsername(string username)	Mutator method for the seller's username.
	int getTicketNumber()	Accessor method for getting the number of tickets available for the show.
	void setTicketNumber(int ticketNumber)	Mutator method for number of remaining tickets for sale.
	void decreaseTicketNumber(int ticketNumber)	Decrement the number of tickets for sale for the event.
	double getCost()	Accessor method for getting the cost of the event per ticket.
	void setCost(double cost)	Mutator method for the cost per ticket for the event.

Entry Abstract Class

Class	Methods	Description
Entry		Class that allows for entries to the daily transaction file.
	explicit Entry(int transactionType)	Constructor for creating an entry.
	int getTransactionType()	Accessor method for getting the type of transaction.
	void setTransactionType(int transactionType)	Mutator method for setting the transaction type.
	string getDTFLine()	Returns a line which will be used to write to the daily transaction file

EventTransaction Class extends Entry

Class	Methods	Description
EventTransaction		A class to hold information for a buy or sell transaction.
	EventTransaction(int type, string eventName, string sellName, double price, int numTickets)	Constructor for creating an event transaction.
	string getSellName()	Accessor method for getting the username of the seller.
	void setSellName(string sellName)	Mutator method for setting the username of the seller.
	string getEventName()	Accessor method for getting the name of the event.
	void setEventName(string eventName)	Mutator method for setting the name of the event.
	double getPrice()	Accessor method for getting the price of an event.
	void setPrice(double price)	Mutator method for setting the price of an event.
	int getNumTickets()	Accessor method for getting the number of tickets.
	void setNumTickets(int numTickets)	Mutator method for setting the number of tickets.
	string getDTFLine()	See Entry#getDTFLine()

AuxillaryTransaction Class extends Entry

Class	Methods	Description
AuxillaryTransaction		A class to hold information for a create, delete, addcredit, or end transaction.
	AuxiliaryTransaction(int type, string username, double credit, string accountType)	Constructor for creating a new transaction.
	string getUsername()	Accessor method for getting the username.
	void setUsername(string username)	Mutator method for setting the username.
	double getCredit()	Accessor method for getting the credit amount associated with the transaction.
	void setCredit(double credit)	Mutator method for setting the amount of credit.
	string getAccountType()	Accessor method for getting the user account type.
	void setAccountType(string type)	Mutator method for setting the account type.
	string getDTFLine()	See Entry#getDTFLine()

Refund Class extends Entry

Class	Methods	Description
Refund		A class to hold information for a refund transaction.
	Refund(int type, string buyName, string sellName, double credit)	Constructor to create a refund transaction.
	string getBuyName()	Accessor method for getting the username of the buyer.
	void setBuyName(string buyName)	Mutator method for setting the username of the buyer.
	string getSellName()	Accessor method for getting the username of the seller.
	void setSellName(string sellName)	Mutator method for setting the username of the seller.
	double getCredit()	Accessor method for getting the amount of credit to be refunded.
	void setCredit(double credit)	Mutator method for setting the amount of credit to be refunded.
	string getDTFLine()	See Entry#getDTFLine()

Transaction Class

Class	Methods	Description
Transaction		A class that defines all of the legal transactions for the user to make.
	Transaction(char* accountPath, char* availTicketPath, char* dailyTransactionPath)	The constructor for starting sessions of transactions.
	bool login(string username)	The initial transaction for starting a user session allowing for all other transactions to be done.
	bool logout()	The transaction that ends a session. No other transactions will be accepted until a user logs in again.
	bool buy(string event, int numTickets, string sellName)	The transaction for buy tickets for an event. The user must have buy privileges.
	bool sell(string event, double salePrice, int availTicket)	The transaction for posting new tickets to be sold for an event.
	bool create(string newUser, string accountType, double accountBalance)	The transaction for creating a new user for the system.
	bool removeUser(string username)	The transaction for deleting a user from the system.
	bool addcredit(double amount)	The transaction for adding credit to a user's account. Buy standard and full standard will use this method since they can only add credit to their own account.
	bool addcredit(string username, double amount)	The transaction for adding credit to a user's account. This method is only for use by a user with administrative privileges.
	bool refund(string buyName, string sellName, double amount)	The transaction for refunding money from one user to another user.
	bool quit()	The transaction for exiting the system. When called the system writes out to the daily transaction file.

	FileIO* getFileIO()	Accessor method for FileIO.
	int getCurrentUser()	Accessor method for the current user's id.
	bool isAdmin()	Identify if the current user has administrative privileges.
	bool isLoggedIn()	Check if a user is logged in to the system.

Account Class

Class	Methods	Description
Account		A class for storing all the relevant data for each user.
	Account(string username, string type, double balance)	Constructor for creating a new account.
	string getType()	Accessor method for retrieving the account type.
	void setType(string type)	Mutator method for setting the account type for each account.
	double getBalance()	Accessor method for retrieving the current balance of the account.
	void setBalance(double balance)	Mutator method for setting the account balance.
	string getUsername()	Accessor method for retrieving the username of the account.
	void setUsername(string username)	Mutator method for setting the username of the account.
	friend bool operator==(Account left, Account right)	Implemented an equals operator to compare to accounts.

FileIO Class

Class	Methods
FileIO	
	FileIO(char* uao, char* ato,char* dtf)
	bool initialize()
	bool updateAccountList()
	bool updateTicketList()
	bool FileIO::writeTransaction(vector<Entry*> newList)
	int findUser(string username)
	int findEvent(string event, string sellName)
	bool isUserUnique(string username)master_term_output
	bool isEventUnique(string event, string sellName)
	vector<Account>* getAccountList()
	vector<Ticket>* getTicketList()

Utility Class

Class	Methods	Description
Utility		A class that contains utility methods that are used by other classes but are unrelated enough to not be apart of those classes.
	string &rtrim(string &s);	Removes all spaces from the right side of a string
	vector<string> &split(const string &s, char delim, vector<string> &elems)	Puts the results in an already constructed vector
	vector<string> split(const string &s, char delim)	Splits a string by a delimiter

Validation Class

Class	Methods	Description
Validation		A file that contains methods for validating input.
	bool validBalance(double balance)	Validate the entered balance.
	bool validUsername(string username)	Validate the username.
	bool validEventName(string event)	Validate the event's name.
	bool validAccountType(string type)	Validate the account type.
	bool validPrice(double price)	Validate the price.

Style Guidelines

Naming

The style is that generally followed with Java development. For file, classes, methods, variables, and constants the naming convention is as follows:

Classes

Will start with an uppercase letter and each new word will start with an uppercase letter.
Example: MyClass.cpp

Methods

Will start with a lowercase letter and each new word will start with an uppercase letter.
Example: myAwesomeMethod

Variables

Will start with a lowercase letter and each new word will start with an uppercase letter.
Example: myAwesomeVariable

Constant

Will be all uppercase with underscores separating words.
Example: MY_CONSTANT_VALUE

Every file, class, method, variable, and constant should be named to allow clarity while being used. Names should be relevant to what the file, class, method, variable, or constant is used for.

When declaring methods, constructors, and deconstructors, the bracket following must be immediately after the method name.
Example:
void myMethod() // Good
void myMethod () // Bad

Comments

For commenting style, javadoc style will be used.

File Commenting

A the beginning of files that are not for classes a comment will start with an outline of the purpose of the file. Followed by the authors of that file.
Example:

```
/**
 * Really important classes that the system just cannot be without.
 *
 * @author Ryan Crawford
 * @author Khalil Fazal
 * @author Joseph Heron
 * @author Carly Marshall
```

```
 *
 */
```

Class Commenting

Commenting of classes will be prior to the class declaration. It will contain an explanation of the class and its purpose as well as the authors of the class. If a file contains a class and the class is it's sole purpose then the file will not have a comment.
Example:

```
/**
 * Really important class that basically does everything.
 *
 * @author Ryan Crawford
 * @author Khalil Fazal
 * @author Joseph Heron
 * @author Carly Marshall
 *
 */
```

Method Commenting

Commenting for methods will take two forms, the header file comments, and the source code file comments. Header file comments will focus on who the method can be used by in external methods while the source code file comments will explain who the method for those who maintain for need to fix it. The header file comment should not be copied in the source code file comment. If the method is rather simple or has sufficient inline comment an initial source code file comment is not necessary. The declaration of the method's parameters and return value with explanation is also required through the use of '@param' and '@return'.
Example:

```
/**
 * The transaction for buy tickets for an event. The user must have buy
 * privileges.
 *
 * @param event the name of the event.
 * @param numTickets the number of tickets to purchase.
 * @param sellName the seller's username.
 * @return whether the buy was successful.
 */
bool buy(string event, int numTickets, string sellName);
```

Methods will also contain inline comments which will be used to identify a point of interest or to make the code more readable. Comments should not be used on simple sections to document easy to identify aspects of the code. Finally, comments must have a space between the actual comment and the comment starting token ('//' or '/*').
Example:

```
// The following searches for a user in the accounts list    // Good
//Bad comment                                                 // Bad
```

Formating

Brackets

Curly braces will be on the same line as the method or class declaration with a space between the declaration and the curly brace. The closing curly brace will be on a new line of it's own. The else statement is on a new line, not on the same line as the closing curly brace. Also, the flow control or loop is a single statement it will use curly brackets.
Example 1:

```
if (true) {
   // Something
}
else {
   // SomethingElse
}
```

Example 2:

```
class Account {
        // Declartion
}
```

Example 3:

```
while (false) {
        // Something
}
```

Spacing

For flow control and loop tokens a single space is required after the keyword and before the bracket.
Example:

```
while (true) {
        // Good
}
```

Math operations will have spaces between them.
Example:

```
variable = 1 + 3 // Good
variable = 1+3   // Bad
```

Line

Line size limit is 80 characters. This is within reason of course. If the line happens to go over 80 by 1 to 10 it can be left on the same line, any more and it needs to be wrapped around. The wrap should attempt to be such that lines of calls are not broken up.

Variables

Variables are declared at the start of the methods and are grouped by similar type. This of course is within reason, for example one is needed for another's declaration.