Software Quality – ENGR 3980
Swift Ticket Project - Phase 2
Front End Rapid Prototype
Design Brief

Created by

Daniel Smullen

Rayan Alfaheid

Jonathan Gillett

28/02/2013

Document Revision 1.0

Contents

Programming Style Guidelines

The following section details the conventions used for programming style and format used pervasively throughout the software.

File Naming Convention

1. Header files are named using the C++ header file naming convention *.hpp* so that C++ header files are not confused with C header files.
2. Source files are named using the C++ source file naming convention *.cpp* so that C++ source files are not confused with C source files.

Variable Labelling Convention

The variable naming convention uses lowercase characters with underscores. For example, variables are labelled such as `like_this` or `a_long_variable_name`.

Class/OOP Conventions

1. Similar to Google style, class names must be camel-case and start with a capital letter. E.g. MyClass, ErrorHandling, TransactionCode.
2. Multiple inheritance should never be used, just as with Google's guidelines a pure interface should be used.
3. The dominant naming convention for C++ is not camel-case.

Code Formatting

1. The 80 character per line limit will be ignored. The convention used will aim for 80 characters, with a hard limit of 100 characters per line. For most non-C code and longer variable names, it is difficult to cleanly meet the 80 character limit without ugly multi-line statements.
2. No tab characters will be used. All indentations will be performed using spaces. Four spaces will be used in place of one tab.
3. All code blocks/curly braces must always be on the next line, no exceptions. Example follows:

```
class MyClass
{
    ...
}

if (condition)
{
    ...
}

for (...)
{
    ...
}
```

4. A curly brace indicating a code block must not have any other statements on the same line. For example, the following represents **incorrect** style:

```
if (condition)
{
    ...
} else          // BAD
{
    ...
}
```

Conditional Statements

5. Conditional statement conditionals must always have the parenthesis one space after the "if" keyword. Example follows:

```
if ( ... )
```

6. Conditions within the parenthesis must not have a space unless it enhances the readability. This applies in instances such as when you have multiple parentheses to enforce order of operations. Example follows:

```
if (variable <= variable2)

if ( (variable + 5) / (variable2 - 3) )
```

7. Every conditional, even if it is one line, must have a curly brace to identify the code block. In keeping with brace convention, it must be on the next line. Example follows:

```
if (variable == 5)
{
    return false;
}
```

8. An 'if' statement followed by an else statement must have the else statement on the next line. Example follows:

```
if (condition)
{
   ...
}
else
{
    ...
}
```

9. Switch statements must have each case indented exactly 2 space characters, and the code block for the case must be indented 4 spaces. Example follows:

```
switch (condition)
{
  case 0:    // 2 space indent
  {
    ...      // 4 space indent
    break;
  }

  case 1:
  {
    ...
    break;
  }

  default:
  {
    assert(false);
  }
}
```

Iteration

10. Similar to conditionals, the conditional for the iteration statement must contain one space before the parentheses, after the keyword. Example follows:

```
while (condition)
{
   ...
}
```

11. Conditions within parentheses must not have a space unless it enhances the readability. Example follows:

```
while ( ((y_2 - y_1) / (x_2 - x_1)) >= 3 )
{
   ...
}
```

12. Every iteration statement, even if it is one line, must have a curly brace to identify the code block. In keeping with the brace convention, it must be on the next line. Example follows:

```
while (variable < 5)
{
   ++variable;
}
```

Functions

13. Function definitions and calls must have the parentheses directly after the function name. Curly braces denoting code blocks must be on separate lines. Example follows:

```
void my_func(...)
{
   ...
}


my_func(...)
```

14. Function arguments must be separated by a space for definition and calling, except for the first and last arguments in parentheses. Example follows:

```
big_function(var_1, var_2, var_3, var_4)
```

15. Function arguments can be separated by a space for the first and last argument if it enhances readability. Example follows:

```
math_operation( (x_1 - x_2) / 2, (y_1 - y_2) / 3 )
```

Classes

16. For information hiding key words, fields must be aligned with the curly braces. The statements for each method and member must be indented four spaces. Example follows:

```
MyClass
{
private:
    variable;        // 4 space indent

public:
    get_variable();  // 4 space indent
}
```

Solution Architecture

Our design incorporates an object-oriented approach, decomposing all major transactions into objects which inherit from a main superclass. A detailed introspection into the design follows.

Overview

The central aspect of the design is the Transaction superclass, which all the transactions within the frontend inherit from. This aspect was chosen because all the transactions have main commonalities, particularly in terms of their collaboration with other objects. Transactions are aggregated within the DailyTransaction container object, which also contains methods for interaction with Transactions.

The User object exists to store and access session information as well as attributes of the current user. It has methods for evaluation of permissions in order to facilitate the privileged nature of certain transactions.

The Exception object contains methods and stores data with respect to errors that occur within the program. It is connected within the architecture to every other class, as exceptions can be thrown within all aspects of the code.

Input and Output

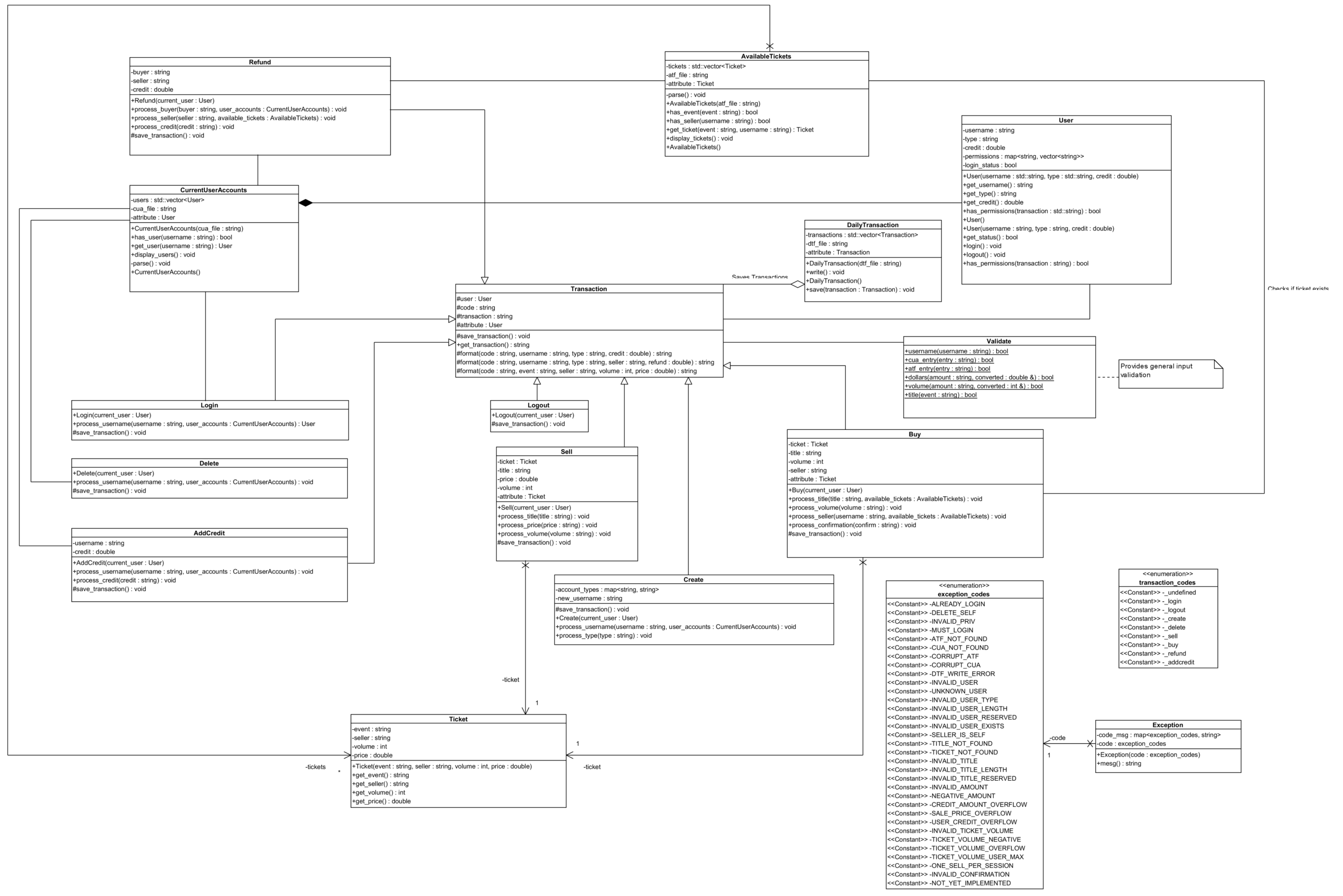
Inputs and outputs are handled by classes respective of each input and output aspect. Input from transactions is handled within a class designated for each transaction within the program. Outputs to individual files are handled by classes dedicated to each output file – the daily transaction file, current user accounts file, etc.

UML Representation

An object model diagram, complete with methods and attributes is presented below. This diagram shows the interaction between the classes that make up the program. It should be noted that while the Exception class does not depict an explicit interaction with other classes, it does explicitly interact with all other objects inside the program, as they can all throw exceptions (and therefore generate Exception objects).

In this diagram, the superclass inheritance relationship between the Transaction class and the various transactions in the program becomes evident. Also, the aggregation of transactions into the daily transaction file, as well as the aggregation of users within the current user accounts file is evident.

UML Diagram


AvailableTickets
-tickets : std::vector<Ticket>
-atf_file : string
-attribute : Ticket
-parse() : void
+AvailableTickets(atf_file : string)
+has_event(event : string) : bool
+has_seller(username : string) : bool
+get_ticket(event : string, username : string) : Ticket
+display_tickets() : void
+AvailableTickets()
Refund
-buyer : string
-seller : string
-credit : double
+Refund(current_user : User)
+process_buyer(buyer : string, user_accounts : CurrentUserAccounts) : void
+process_seller(seller : string, available_tickets : AvailableTickets) : void
+process_credit(credit : string) : void
#save_transaction() : void
User
-username : string
-type : string
-credit : double
-permissions : map<string, vector<string>>
-login_status : bool
+User(username : std::string, type : std::string, credit : double)
+get_username() : string
+get_type() : string
+get_credit() : double
+has_permissions(transaction : std::string) : bool
+User()
+User(username : string, type : string, credit : double)
+get_status() : bool
+login() : void
+logout() : void
+has_permissions(transaction : string) : bool
CurrentUserAccounts
-users : std::vector<User>
-cua_file : string
-attribute : User
+CurrentUserAccounts(cua_file : string)
+has_user(username : string) : bool
+get_user(username : string) : User
+display_users() : void
-parse() : void
+CurrentUserAccounts()
DailyTransaction
-transactions : std::vector<Transaction>
-dtf_file : string
-attribute : Transaction
+DailyTransaction(dtf_file : string)
+write() : void
+DailyTransaction()
+save(transaction : Transaction) : void
Saves Transactions
Checks if ticket exists
Transaction
#user : User
#code : string
#transaction : string
#attribute : User
#save_transaction() : void
+get_transaction() : string
#format(code : string, username : string, type : string, credit : double) : string
#format(code : string, username : string, type : string, seller : string, refund : double) : string
#format(code : string, event : string, seller : string, volume : int, price : double) : string
Validate
+username(username : string) : bool
+cua_entry(entry : string) : bool
+atf_entry(entry : string) : bool
+dollars(amount : string, converted : double &) : bool
+volume(amount : string, converted : int &) : bool
+title(event : string) : bool
Provides general input validation
Login
+Login(current_user : User)
+process_username(username : string, user_accounts : CurrentUserAccounts) : User
#save_transaction() : void
Logout
+Logout(current_user : User)
#save_transaction() : void
Delete
+Delete(current_user : User)
+process_username(username : string, user_accounts : CurrentUserAccounts) : void
#save_transaction() : void
Sell
-ticket : Ticket
-title : string
-price : double
-volume : int
-attribute : Ticket
+Sell(current_user : User)
+process_title(title : string) : void
+process_price(price : string) : void
+process_volume(volume : string) : void
#save_transaction() : void
Buy
-ticket : Ticket
-title : string
-volume : int
-seller : string
-attribute : Ticket
+Buy(current_user : User)
+process_title(title : string, available_tickets : AvailableTickets) : void
+process_volume(volume : string) : void
+process_seller(username : string, available_tickets : AvailableTickets) : void
+process_confirmation(confirm : string) : void
#save_transaction() : void
AddCredit
-username : string
-credit : double
+AddCredit(current_user : User)
+process_username(username : string, user_accounts : CurrentUserAccounts) : void
+process_credit(credit : string) : void
#save_transaction() : void
Create
-account_types : map<string, string>
-new_username : string
#save_transaction() : void
+Create(current_user : User)
+process_username(username : string, user_accounts : CurrentUserAccounts) : void
+process_type(type : string) : void
<<enumeration>>
exception_codes
<<Constant>> -ALREADY_LOGIN
<<Constant>> -DELETE_SELF
<<Constant>> -INVALID_PRIV
<<Constant>> -MUST_LOGIN
<<Constant>> -ATF_NOT_FOUND
<<Constant>> -CUA_NOT_FOUND
<<Constant>> -CORRUPT_ATF
<<Constant>> -CORRUPT_CUA
<<Constant>> -DTF_WRITE_ERROR
<<Constant>> -INVALID_USER
<<Constant>> -UNKNOWN_USER
<<Constant>> -INVALID_USER_TYPE
<<Constant>> -INVALID_USER_LENGTH
<<Constant>> -INVALID_USER_RESERVED
<<Constant>> -INVALID_USER_EXISTS
<<Constant>> -SELLER_IS_SELF
<<Constant>> -TITLE_NOT_FOUND
<<Constant>> -TICKET_NOT_FOUND
<<Constant>> -INVALID_TITLE
<<Constant>> -INVALID_TITLE_LENGTH
<<Constant>> -INVALID_TITLE_RESERVED
<<Constant>> -INVALID_AMOUNT
<<Constant>> -NEGATIVE_AMOUNT
<<Constant>> -CREDIT_AMOUNT_OVERFLOW
<<Constant>> -SALE_PRICE_OVERFLOW
<<Constant>> -USER_CREDIT_OVERFLOW
<<Constant>> -INVALID_TICKET_VOLUME
<<Constant>> -TICKET_VOLUME_NEGATIVE
<<Constant>> -TICKET_VOLUME_OVERFLOW
<<Constant>> -TICKET_VOLUME_USER_MAX
<<Constant>> -ONE_SELL_PER_SESSION
<<Constant>> -INVALID_CONFIRMATION
<<Constant>> -NOT_YET_IMPLEMENTED
<<enumeration>>
transaction_codes
<<Constant>> -_undefined
<<Constant>> -_login
<<Constant>> -_logout
<<Constant>> -_create
<<Constant>> -_delete
<<Constant>> -_sell
<<Constant>> -_buy
<<Constant>> -_refund
<<Constant>> -_addcredit
Exception
-code_msg : map<exception_codes, string>
-code : exception_codes
+Exception(code : exception_codes)
+mesg() : string
-code
1
-ticket
1
Ticket
-event : string
-seller : string
-volume : int
-price : double
+Ticket(event : string, seller : string, volume : int, price : double)
+get_event() : string
+get_seller() : string
+get_volume() : int
+get_price() : double
-tickets
*
1
-ticket

Classes

The following is a list of the classes within the project. Please refer to the Phase 1 Software Requirements Specification for specific requirement information.

AddCredit Class

Contains the methods and attributes for the addcredit transaction. This means that the actual underlying functionality for adding credit to user accounts is handled by this object. As per the original design and architecture, this class is instantiated whenever the addcredit transaction is performed.

Requirements Addressed

The requirements met are regarding processing user input for the username provided, credit amount to be added, and saving the transaction to daily transaction list.

12, 12.1, 12.1.1, 12.1.2, 12.1.3, 12.2

AvailableTickets Class

This class contains the methods and attributes related to the available tickets file, and interacting with that file on disk. This class provides methods for writing the output to the file, and also validates and stores the values in memory while the available tickets are interacted with by other operations.

Requirements Addressed

The requirements met are regarding processing the available tickets file, reading in the list of available tickets into memory, and ensuring that there is no corruption.

14, 14.1, 14.2, 14.2.1, 14.2.2, 14.3, 14.3.1, 14.3.2, 14.4, 14.4.1, 14.4.2, 14.5, 14.5.1, 14.5.2, 14.6, 14.7. 14.8, 14.9, 14.10, 21

Buy Class

Contains the methods and attributes for the buy transaction. This means that the underlying functionality for completing the purchasing of a ticket (or tickets) is handled by this object. As per the original design and architecture, this class is instantiated whenever the buy transaction is performed.

Requirements Addressed

The requirements met are with respect to processing user input for the event title, seller username, transaction confirmation, and saving the transaction to daily transaction list.

10, 10.1, 10.2, 10.2.1, 10.2.2, 10.3, 10.3.1, 10.3.2, 10.4, 10.4.1, 10.4.2, 10.5, 10.6, 10.6.1, 10.7, 10.8, 10.9,

Create Class

Contains the methods and attributes for the create transaction. This means that this object is dedicated to facilitating the process for creating users. Per the original design, this class is instantiated whenever the create transaction is performed.

Requirements Addressed
The requirements met are with respect to processing user input for the username and account type for the new user to create and saving the transaction to daily transaction list.

7, 7.1, 7.1.1, 7.1.2, 7.1.3, 7.2, 7.2.1, 7.3, 7.4

CurrentUserAccounts Class
Contains the methods and attributes for interacting with the current user accounts file. This object is designed for tracking and facilitating privilege and user management.

Requirements Addressed
The requirements met are with respect to processing the current user accounts file, reading in the list of user accounts into memory, and ensuring that there is no corruption.

13, 13.1, 13.2, 13.2.1, 13.2.2, 13.3, 13.4, 13.4.1, 13.4.2, 13.5, 13.6, 13.7, 13.8, 13.9, 13.10, 20

DailyTransaction Class
Contains the methods and attributes related to interacting with the daily transaction file. This class provides methods for writing out the daily transaction file, adding entries to it, and storing those entries in memory for processing before output to disk.

Requirements Addressed
The requirements met are with respect to writing the daily transactions stored in memory to the file specified. The daily transaction file stores a list of Transaction objects which handle the individual formatting of each type of daily transaction.

15, 16, 17

Delete Class
Contains the methods and attributes for the delete transaction. This class is instantiated in each instance that the delete transaction takes place. Methods are provided for deleting user accounts from the current user accounts file.

Requirements Addressed
The requirements met are with respect to processing the input for the username to delete, verifying the user exists and saving the transaction to the daily transaction file.

8, 8.1, 8.2, 8.2.1, 8.2.2, 8.3, 8.4, 8.5,

Exception Class
The exception class stores the strings used during exceptions that are thrown within various other classes throughout the program. The exception class provides facilities for mapping exception codes to diagnostic messages.

Requirements Addressed

The exceptions class is used by all of the classes within the project, including each transaction in order to throw exceptions in the event that a handled exception occurs. The exception class is used by every class to meet all validation requirements.

Login Class

Contains the methods and attributes for the login transaction. This class is instantiated in each instance that a user logs in. During the course of the execution of this class, it should be noted that login transactions do not leave artifacts in the daily transaction file.

Requirements Addressed

The requirements met are with respect to processing the input for the username to login, verifying the user exists.

5, 5.1, 5.2, 5.2.1, 5.3, 5.3.1, 5.4, 5.5, 5.6, 5.7

Logout Class

Contains the methods and attributes for the logout transaction. This class is instantiated and used when a user logs out.

Requirements Addressed

The requirements met are with respect to logging the user of the system and saving the transaction to the daily transaction file.

6, 6.1

Refund Class

Contains the methods and attributes for the refund transaction. This class is used in each instance that the refund transaction is used, and contains helper functions for processing refund amounts, and making changes to the available tickets and account credit for users.

Requirements Addressed

The requirements met are with respect to processing the buyer and seller usernames, verifying the accounts exist, processing the credit amount to transfer, and saving the transaction to the daily transaction file.

11, 11.1, 11.2, 11.2.1, 11.3, 11.3.1, 11.4, 11.4.1, 11.5, 11.5.1, 11.6, 11.7

Sell Class

Contains the methods and attributes for the sell transaction. This class is used for processing sales, and contains functions used for adjusting ticket volumes and account credits based on sales.

Requirements Addressed

The requirements met are with respect to processing user input for the event title, ticket price, number of tickets for sale, and saving the transaction to daily transaction list.

9, 9.1, 9.2, 9.2.1, 9.2.2, 9.3, 9.4, 9.4.1, 9.4.2, 9.5, 9.5.1, 9.5.2, 9.6, 9.7

Ticket Class

Contains methods and attributes for data related to event tickets. Each entry in the available tickets file is stored as a ticket object.

Requirements Addressed

The Ticket class is used indirectly by the AvailableTickets class to process the available tickets file and fulfill the requirements met by the AvailableTickets class.

Transaction Class

Contains methods and attributes for data related to completed transactions. Each transaction is stored within the daily transaction file, and the DailyTransaction class contains an array of Transaction objects. This class provides various methods for interacting with the attributes associated with transaction entries.

Requirements Addressed

The transaction class fulfills all of the requirements with respect to the daily transaction formatting for each of the different categories of daily transaction file entries.

15.1, 15.2, 15.2.1, 15.2.2, 15.3, 15.4, 15.4.1, 15.4.2, 16.1, 16.2, 16.2.1, 16.2.2, 16.3, 16.3.1, 16.3.2, 16.4, 16.4.1, 16.4.2, 17.1, 17.2, 17.2.1, 17.2.2, 17.3, 17.3.1, 17.3.2, 17.4, 17.4.1, 17.4.2, 17.5, 17.5.1, 17.5.2, 18, 18.1, 18.2, 18.3, 18.4, 18.5, 18.6

User Class

Contains methods and attributes for data related to users. Each user is stored within the current user accounts file, and the CurrentUserAccounts container object. This class also provides methods for interacting with user attributes.

Requirements Addressed

The User class is used indirectly by the CurrentUserAccounts class to process the current user accounts file and fulfill the requirements met by the CurrentUserAccounts class.

Validate Class

This class contains methods for validating input from users and from the various files interacted with by the program. These methods ensure that input and output is compliant with requirements.

Requirements Addressed

The Validate class is used indirectly by nearly every class to perform basic input conversion and validation in order to meet all of the data and input validation requirements.