ZILDOR, INC.

Software Design Document

SplitPay

Rick Aasen
Leland Cerauskis
Blake Matson
Ed Carlisle
Eric Jeffers
Josh Ritchey
Nick Carson
Travis Green
Phuong Vo

Version 1.0 Approved
March 4, 2011

TABLE OF CONTENTS

1. INTRODUCTION

The purpose of this software design document is to provide a low-level description of the SplitPay system, providing insight into the structure and design of each component. Topics covered include the following:

- Class hierarchies and interactions
- Data flow and design
- Processing narratives
- Algorithmic models
- Design constraints and restrictions
- User interface design
- Test cases and expected results

In short, this document is meant to equip the reader with a solid understanding of the inner workings of the SplitPay system.

1.1 GOALS AND OBJECTIVES

The purpose of SplitPay is to facilitate the process of resolving shared expenses, like those that arise when paying rent, splitting the check at dinner, or dividing travel expenses. Thus, the primary objective of the SplitPay project is to create a viable alternative to sorting out such expenses manually.

Accordingly, the final product must be quick, efficient, and extremely easy to use. It must offer useful features without overwhelming the user with options. The user interface must be intuitive and have little or no learning curve. Beyond these general design principles, the application must also provide the following concrete functionalities:

- Algorithm for determining the most efficient payment scheme
- Support for multiple groups, bills, and transactions
- Automatic data synchronization
- Group, bill, and transaction history
- Push notifications
- Support for offline users

1.2 PROJECT OVERVIEW AND SCOPE

The SplitPay system is composed of two primary components: a client-side application that receives user input and performs calculations, and a server-side application which updates and synchronizes data across devices. The system features can be broken up into two groups as well: core features, which are essential to the function of the application, and additional features, which are only meant to add extra functionality. The following list includes all of the features currently designated for inclusion in the final release of SplitPay:

CORE FEATURES

1. USER REGISTRATION & WELCOME
   - Only appears once (the first time the application is run)
   - Allows the user to register with the SplitPay server
   - Enables the user to customize his/her account settings and preferences
2. GROUP CREATION & MANAGEMENT
   - Streamlines the process of creating and organizing groups
   - Provides support for multiple groups
   - Allows the user to add group members manually or from contacts list
3. POSTING A BILL
   - Stores and monitors the bill amount, the individuals involved, etc.
   - Includes support for multiple simultaneous bills
   - Efficiently distributes debt amongst the individuals responsible for the bill
4. MEMBER-TO-MEMBER TRANSACTIONS
   - Enables group members to simulate transfers of debt, payments made, etc.
   - Adjusts member balances accordingly
   - Records relevant information (amount paid, members involved, etc.)
5. FINAL DEBT RESOLUTION
   - Calculates the most efficient method of sorting out debts
   - Notifies group members of unresolved debts, credits, etc.
   - Offers the option to disband a group once all payments are made
6. GROUP HISTORY
   - Automatically records all transactions and bills posted to each group
   - Provides users with access to a detailed history of transactions
   - Supports sorting transactions by date, amount, payer, etc.
7. SHOW ALL DEBTS
   - Enumerates all of a user's unresolved debts across each group he/she is a part of
   - Provides easy access to relevant information (past transactions, group info, etc.)
   - Offers the option to resolve a debt (or debts) immediately

8. PUSH NOTIFICATIONS
   - Appear after any significant event occurs in a group
   - Alert group members of newly incurred expenses
   - Remind users of unresolved debts

The features below are *not* guaranteed to be present in the final release of SplitPay, but will be added as time permits. Due to their tentative nature, they will not be covered in this document.

ADDITIONAL FEATURES

1. HELP MENU
   - Displays a list of topics covering the different components of SplitPay
   - Offers detailed information on each feature, menu, etc.
   - Can be accessed at any time via the Settings menu
2. SETTINGS MENU
   - Allows the user to customize his/her preferences
   - Enables the user to modify certain features and functionalities
   - Can be accessed at any time using the built-in Settings button on Android phones
3. PAYPAL INTEGRATION
   - Incorporates a mechanism for initiating real transactions
   - Facilitates secure, hassle-free transactions between members
   - Automatically updates member balances as transactions occur
4. GPS TRACKING
   - Stores location data associated with certain events
   - Utilizes Google Maps to display transaction locations
   - Creates an expense map which can be viewed by all members of a group
5. RECEIPT IMAGING
   - Utilizes the camera built into Android handsets
   - Records and stores a snapshot of receipts associated with different expenses
   - Provides a method of checking/verifying expenses posted to a group
6. E-MAIL/SMS NOTIFICATIONS
   - Extends the standard notifications service built into SplitPay
   - Automatically delivers notifications via e-mail and/or text message
   - Enables individuals without SplitPay to receive group notifications
7. SPLITPAY TUTORIAL
   - Provides an abridged version of the Help menu for first-time users
   - Offers a step-by-step run through of each feature, menu, etc.
   - Enables any user to quickly and easily take advantage of all of SplitPay's functionalities

1.3 SOFTWARE CONTEXT

SplitPay will be offered on the Android market free of charge. Development and maintenance costs are virtually nonexistent, so funding should not be an issue. If, however, this situation changes at some point, it will be possible to fund the project by incorporating on-screen advertisements into the application.

Future development plans will be based on the features (if any) that do not make it in the initial release of the application. If all of these features are included, there are several experimental features that will potentially be incorporated. These features are not covered in this document.

1.4 MAJOR CONSTRAINTS

The greatest constraint for the SplitPay project is time. There is roughly one month allocated to the development, testing, and documentation of this project, including both the Android application and the server-side application and database. Collectively, the development team has very little experience with the Android platform, so a significant portion of this time will be dedicated to learning the environment. Consequently, time is an even greater constraint. This may result in fewer features in the initial release, however the *core* functionality of the system will be unaffected.

1.5 INTENDED AUDIENCE AND READING SUGGESTIONS

While the software requirement specification (SRS) document is written for a more general audience, this document is intended for individuals directly involved in the development of SplitPay. This includes software developers, project consultants, and team managers. This document need not be read sequentially; users are encouraged to jump to any section they find relevant. Below is a brief overview of each part of the document.

- Part 1 (Introduction)
    - This section offers a summary of the SplitPay project, including goals and objectives, project scope, general system details, and some major constraints associated with the intended platform.

- Part 2 (Data Design)
    - Readers interested in how SplitPay organizes and handles data should consult this section, which covers data structures and flow patterns utilized by the system.
- Part 3 (Architectural and Component-Level Design)
    - This section describes the SplitPay system class by class, including interface details, class hierarchies, performance/design constraints, process details, and algorithmic models.
- Part 4 (User Interface Design)
    - This section covers all of the details related to the structure of the graphical user interface (GUI), including some preliminary mockups of the SplitPay Android application. Readers can view this section for a tentative glimpse of what the final product will look like.
- Part 5 (Restrictions, Limitations, and Constraints)
    - This section discusses the general constraints imposed upon the project
- Part 6 (Testing Issues)
    - Readers interested in the software testing process should consult this section, which offers a list of test cases, expected responses, and other pertinent information.
- Part 7 (Appendices)
    - This section includes any additional information which may be helpful to readers.

2. DATA DESIGN

2.1 INTERNAL SOFTWARE DATA STRUCTURE

SplitPay's internal structure is divided into two parts: server-side and client-side.

On the client side, data will reside locally in memory and will be organized based on the classes defined later in this document. Since the SplitPay program may be considered data-centric, the classes that handle the data will be isolated and will be accessed by way of a Model-View-Controller system. The data on the local Android client will be requested from the server at application initialization and refreshed as necessary based on user actions.

The data structure on the server will essentially mirror the structure of the local Android client in terms of member fields of the classes. The server will be implemented using PHP. Permanent storage of user information will be accomplished using a MySQL database. Section 3.8 (*Server Component*) covers this subject in further detail.

The server and Android client will exchange data using the JSON format. JSON is a lightweight object description language that is similar to XML. JSON is being used due to its versatility and because of the fact that an implementation is available for both PHP and JAVA.

2.2 GLOBAL DATA STRUCTURE

The global data structure of this application is best characterized by the database. The database structure shows the data involved in the application in its purest sense. The local Android client of SplitPay will never access this database directly; it will instead issue requests to the server.

2.3 TEMPORARY DATA STRUCTURE

Temporary data structures, as they relate to SplitPay, refer to the data objects that are created on the local Android client, and also to the JSON objects that interchanged between the server and the client. The data objects created on the local device will only exist for the duration of time that the application is running, and will subsequently be destroyed.

The JSON objects will only exist for the duration of the transaction between the client and server. The server will destroy the objects after sending them, and the client will destroy the JSON objects once they have been parsed.

2.4 DATABASE DESCRIPTION

```
-- Table structure for table `bill`
--
CREATE TABLE `bill` (
  `billId` int(11) NOT NULL auto_increment,
  `groupId` int(11) NOT NULL,
  `amount` int(11) NOT NULL,
  `displayName` varchar(30) NOT NULL,
  `payer` int(11) NOT NULL,
  `involved` int(11) NOT NULL,
  PRIMARY KEY  (`billId`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;

--
-- Table structure for table `billPayees`
--

CREATE TABLE `billPayees` (
  `billId` int(11) NOT NULL,
  `userId` int(11) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

--
-- Table structure for table `group`
--

CREATE TABLE `group` (
  `groupId` int(11) NOT NULL auto_increment,
  `displayName` varchar(30) NOT NULL,
  `userId` int(11) NOT NULL,
  `leaderId` int(11) NOT NULL,
  `isActive` tinyint(1) NOT NULL,
  PRIMARY KEY  (`groupId`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;

--
-- Table structure for table `member`
--

CREATE TABLE `member` (
  `userId` int(11) NOT NULL,
  `groupId` int(11) NOT NULL,
  `balance` int(11) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```
--
-- Table structure for table `prefrences`
--

CREATE TABLE `prefrences` (
  `userId` int(11) NOT NULL,
  `setting1` varchar(30) NOT NULL,
  `setting2` varchar(30) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;


--
-- Table structure for table `transactions`
--

CREATE TABLE `transactions` (
  `transactionId` int(11) NOT NULL auto_increment,
  `groupId` int(11) NOT NULL,
  `toMember` int(11) NOT NULL,
  `fromMember` int(11) NOT NULL,
  `amount` decimal(10,0) NOT NULL,
  PRIMARY KEY  (`transactionId`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;

--
-- Table structure for table `users`
--

CREATE TABLE `users` (
  `userId` int(11) NOT NULL auto_increment,
  `email` varchar(30) NOT NULL,
  `displayName` varchar(30) NOT NULL,
  `isOffline` tinyint(1) NOT NULL,
  PRIMARY KEY  (`userId`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

3. ARCHITECTURAL & COMPONENT-LEVEL DESIGN

3.1 SYSTEM STRUCTURE

The SplitPay system is broken up into two major components: a client-side Android application and a server-side PHP application and MySQL database.

The client-side application is also separated into two parts: the functional component (written in Java), and the graphical component (written in XML). The functional component forms the core of SplitPay. It receives user input and constructs groups, bills, and transactions. It performs all of the calculations required to resolve debts. The graphical component, as the name implies, is simply the graphical user interface. It provides all of the buttons, text boxes, and other on-screen elements which allow the user to access all of the features provided by the application.

The server component of SplitPay is comprised of a PHP interface, which manages incoming and outgoing messages, and a MySQL database, which provides centralized storage for synchronized data. The server application receives serialized data from Android devices and parses it into useful information. This data is then stored in the database and subsequently synchronized to other devices (if any) in the same group.

3.2 USERACCOUNT CLASS

The UserAccount class is meant to represent SplitPay user accounts and includes a unique identifier, the user's e-mail address, a display name, and any other pertinent information (covered below). User objects are meant to represent SplitPay users within the application, and consequently there is one unique UserAccount object associated with each user.

3.2.1 PROCESSING NARRATIVE (PSPEC)

When a user first creates his/her account with SplitPay, a new UserAccount object is created. This object is responsible for storing information unique to the SplitPay user. This includes the following:

- User ID
- E-mail address
- Display name
- Groups, bills, and transactions in which the user is involved

Thus, any time information related to a user is required, the UserAccount object is called upon. For instance, when a group leader adds a new member to a group, a UserAccount object is used to uniquely identify that user.

3.2.2 INTERFACE DESCRIPTION

UserAccount

```
new( displayName: String, email: String )
getDisplayName() : String
setDisplayName( newName : String )
getEmail() : String
setEmail( newEmail : String )
getID() : int
getGroups() : Group[]
getBills() : Bill[]
getTransactions(): Transaction[]
joinGroup( group: Group ) : boolean
leaveGroup( groupId: int ) : boolean
addBill( bill: Bill ) : boolean
removeBill( billId: int ) : boolean
addTransaction( transaction: Transaction ) : boolean
removeTransaction( transactionId: int ) : boolean
pullDataFromServer() : boolean
pushDataToServer() : boolean
```

3.2.3 PROCESSING DETAIL

Since UserAccount is mainly used for data storage/retrieval, there are no algorithms associated with this class. The only methods in the class are accessors, mutators, and processes that communicate with the server.

3.2.3.1 DESIGN CLASS HIERARCHY

The UserAccount class has no parent or child classes. However, each instance GroupMember has an associated UserAccount (which is not necessarily unique to that GroupMember).

3.2.3.2 RESTRICTIONS/LIMITATIONS

Since the UserAccount class is self-contained, there are no practical restrictions.

3.2.3.3 PERFORMANCE ISSUES

Considering that there is only one UserAccount object associated with each device, there are no performance-related issues associated with this class. It requires a minimal amount of information to be stored on the phone, and requires no processor-intensive calculations. The

only potential performance issue would occur when this class attempts to communicate with the SplitPay server. If the device cannot connect to the server, then the methods responsible for communicating with the server will not be carried out until connection is restored.

3.2.3.4 DESIGN CONSTRAINTS

The major constraint for this class is that there may only be one instance of UserAccount per handset. This is because there is only one user associated with each device. In addition, each user ID must be unique, or else interactions with the server will be highly problematic. Lastly, each UserAccount must have a valid e-mail address.

3.2.3.5 PROCESSING DETAIL FOR EACH OPERATION

- `new( displayName: String, email: String )`
  - When a user first registers for an account, he/she inputs a display name and a valid e-mail address. The UserAccount constructor is called and this information is passed, creating a new UserAccount object for that user.

- `joinGroup( group: Group ) : boolean`
  - Each UserAccount stores a list of all groups which the user is a member of. Thus, whenever a user creates or is added to a group, this method is called to add that group to the UserAccount.

- `leaveGroup( groupId: int ) : boolean`
  - If a group is disbanded or a member leaves the group, this method is invoked to remove the target group from the UserAccount's group list.

- `addBill( bill: Bill ) : boolean`
  - As with groups, each UserAccount stores a list of bills the user is involved in. Thus, whenever the user is involved in a new bill, this method is called to update the bills list.

- `removeBill( billId: int ) : boolean`
  - Whenever a bill is resolved and deleted, this method is invoked to update the bill list within this UserAccount.

- `addTransaction( transaction: Transaction ) : boolean`
  - As with bills, whenever a user is involved in a transaction, the associated UserAccount calls this method to add the transaction to the transactions list.

- `removeTransaction( transactionId: int ) : boolean`
    - When a transaction is resolved and deleted, this method is invoked to update this UserAccount's transactions list.

- `pullDataFromServer() : boolean`
    - If a bill or transaction is created, all the users involved must have their UserAccount objects updated to reflect the changes. When this method is called, all UserAccount information stored on the server replaces this UserAccount's information (if it has been changed).

- `pushDataToServer() : boolean`
    - This method performs the opposite function of pullDataFromServer(): it replaces all outdated UserAccount information on the server with the UserAccount information stored on the phone.

- Accessors/mutators
    - These methods are used to obtain/modify UserAccount information as needed. All fields can be modified except for the unique identifier.

3.3 GROUPMEMBER CLASS

The GroupMember class represents participants in SplitPay groups. This class includes a unique identifier, the respective group member's e-mail address, display name, a boolean value indicating if the member is the group leader, and the member's balance for the group. Each instance of the classes Group, Bill, and Transaction holds one or more instances of this class.

3.3.1 PROCESSING NARRATIVE (PSPEC)

When a user adds a member to a group, an instance of GroupMember is created. This object is responsible for storing all the information about the new member necessary for group functionality. This includes the following:

- User ID
- E-mail address
- Display name
- Whether or not the member is an "offline user"
- Whether or not the member is the group leader
- The current balance of the member for that particular group

This class is referenced in all of the following situations:

- A user wishes to create or delete a group, bill, or transaction
- A user wishes to add or remove members of a group
- A group wishes to reconcile all debt
- The user requests information about a participant in a group, bill, or transaction (e.g., when showing a group's history of bills and transactions)

3.3.2 INTERFACE DESCRIPTION

GroupMember

```
new( displayName: String, email: String )
getIsOffline(): boolean
setIsOffline( b: boolean )
getUserId(): int
getEmail(): String
getDisplayName(): String
setDisplayName( name: String )
getBalance(): double
setBalance( balance: double )
```

3.3.3 PROCESSING DETAIL

As this class only represents a member of a group, bill, or transaction, it will not perform any computations on its own and therefore requires only accessor and mutator methods.

3.3.3.1 DESIGN CLASS HIERARCHY

The GroupMember class has no parent or child classes. It does, however, hold a reference to its associated UserAccount object.

3.3.3.2 RESTRICTIONS/LIMITATIONS

Due to the simplistic nature of this class, there are no known restrictions or limitations.

3.3.3.3 PERFORMANCE ISSUES

Performance issues for this class should be minimal. It is little more than a small aggregation of information for specific members of groups, and therefore should not require significant processor or storage resources.

3.3.3.4 DESIGN CONSTRAINTS

Due to the nature of the design, each instance of GroupMember must be associated with an instance of Group, Bill, or Transaction. There will be no free-floating instances of the class. In addition, each instance must have a unique user ID and e-mail address.

3.3.3.5 PROCESSING DETAIL FOR EACH OPERATION

This class contains no operations beyond the standard accessor and mutator methods.

3.4 GROUP CLASS

The Group class represents a collection of SplitPay users that have elected to distribute amongst themselves the balances of bills and transactions. Each group has a unique ID, display name, a list of participants (instances of class GroupMember), a list of associated transactions (instances of class Transaction), a list of associated bills (instances of class Bill), as well as methods for adding and removing members, stopping or disbanding the group, and several other operations (covered in the following sections).

3.4.1 PROCESSING NARRATIVE (PSPEC)

When a user creates a new group, a new Group object is constructed. This object is responsible for storing the associations between members, bills, and transactions, as well as calculating and updating the balances of each member according to the proprietary SplitPay algorithm.

3.4.2 INTERFACE DESCRIPTION

Group

```
new( displayName: String )
updateGroup( message : String )
addMember( member: GroupMember )
removeMember( member: GroupMember )
findMember( id: int ): GroupMember
addBill( [bill parameters] )
addTransaction( [transaction parameters] )
disband()
stop()
resolveAllDebts()
getDisplayName(): String
setDisplayName( name: String )
getGroupId(): int
```

3.4.3 PROCESSING DETAIL

In addition to the standard accessor and mutator methods for attributes, this class implements the SplitPay algorithm in the method `resolveAllDebts()` to return to the user a suggested payment scenario that balances all debts and credits with the fewest number of payments.

3.4.3.1 DESIGN CLASS HIERARCHY

The Group class has no parent or child classes. However, each instance of Group holds collections of GroupMember, Bill, and Transaction objects.

3.4.3.2 RESTRICTIONS/LIMITATIONS

Although Group objects hold a substantial amount of information, there is not enough data present to cause any memory-related issues. The amount of memory space occupied by a Group object will be minimal compared to the space available.

3.4.3.3 PERFORMANCE ISSUES

Projected performance issues for this class should be minimal. The only computationally complex aspect of this class is the `resolveAllDebts` algorithm, but it should perform perfectly on all Android devices. The algorithm's execution time scales linearly with the number of group members, and only basic mathematical operations (addition/subtraction) are utilized. Consequently, the algorithm's execution time will be negligible, even for very large groups.

3.4.3.4 DESIGN CONSTRAINTS

This class has only two constraints: (1) group identifiers must be unique, as they are used by the server to identify the group, and (2) each instance of Group must have one or more members, one of which must be designated the leader.

3.4.3.5 PROCESSING DETAIL FOR EACH OPERATION

- `new( displayName: String )`
  - This constructor is called whenever a user creates a new group and enters in all of the required information (in this case, the display name).
- `addMember( member: GroupMember )`
  - This method receives an instance of class GroupMember and adds it to the `members[]` array of the group.

- `removeMember( member: GroupMember )`
  - This method receives an instance of class GroupMember and removes it from the `members[]` array of the group.
- `findMember( id: int )`
  - This method searches `members[]` for a GroupMember with a user ID matching the parameter id.
- `disband()`
  - This method dissolves the group. Only the leader may call this method.
- `stop()`
  - This method prevents new bills and transactions from being added to the group, as well as member creation/deletion. Only the leader may call this method.
- `resolveAllDebt()`
  - This method uses a mathematical algorithm to calculate the most efficient manner of balancing debts in the group, minimizing the overall number of payments. Only the leader may call this method.

3.5 BILL CLASS

Bill objects are meant to represent expenses that group members will be dividing amongst themselves. A Bill object includes a unique identifier, a display name, the bill amount, the designated payer, and a list of members involved with the bill.

3.5.1 PROCESSING NARRATIVE (PSPEC)

When a user creates a Bill object, a unique identifier is assigned to it and the following information is collected:

- Display name
- Bill amount
- GroupMembers involved
- Timestamp (automatically recorded by the application)

The `splitBill()` method will then be called upon to determine the changes for each members balance. Finally, the `pushBalanceChanges()` method will push these changes to the server which can then apply the changes to each member's balance in the database.

3.5.2 INTERFACE DESCRIPTION

Bill

```
new( displayName: String, double: amount, GroupMember: payer,
GroupMember: involved[] )
getDisplayName() : String
setDisplayName( newName : String ) : boolean
getID() : int
getInvolved() : GroupMember[]
getPayer() : GroupMember
getAmount() : int
splitBill() : boolean
pushBalanceChanges() : boolean
```

3.5.3 PROCESSING DETAIL

This class is responsible for splitting a bill into equal payments and updating involved members' balances.

3.5.3.1 DESIGN CLASS HIERARCHY

The Bill class has no parent or child classes. However, each instance of Group has an array of associated Bills.

3.5.3.2 RESTRICTIONS/LIMITATIONS

Since the Bill class is self-contained, there are no practical restrictions.

3.5.3.3 PERFORMANCE ISSUES

The only processing that this class is responsible for is splitting a bill into equal payments and then determining the changes that need to be made to each member's balance, neither of which is computationally intensive. The only potential performance issue would occur when this class attempts to communicate with the SplitPay server. If the device cannot connect to the server, then the methods responsible for communicating with the server will not function correctly. It will, instead, cache the information locally and send it once connection is restored.

3.5.3.4 DESIGN CONSTRAINTS

The major constraint for this class is that data pushed to the server should be relative as opposed to absolute. For example, if a member's balance needs to be changed from $30 to $40, the change in balance, $10, would be pushed to the server, rather than the final balance, $40. This insures that if several bills are added at the same time, there will be no conflicts.

3.5.3.5 PROCESSING DETAIL FOR EACH OPERATION

- `new( displayName: String, double: amount, GroupMember: payer, GroupMember: involved[])`
  - When a user creates a bill, he/she inputs a display name and a list of members involved in the bill. The Bill constructor is called and this information is passed, creating a new Bill object for that Group.
- `splitBill() : boolean`
  - This method is responsible for determining the balance changes for each member involved with the bill.
- `pushBalanceChanges() : boolean`
  - This method is responsible for pushing the balance changes that were determined in the splitBill() method to the server.
- Accessors/mutators
  - These methods are used to obtain/modify Bill information as needed. The only field that can be modified is the displayName.

3.6 TRANSACTION CLASS

The Transaction class is meant to represent monetary exchanges between group members. Each instance includes a unique identifier, the transaction amount, the payer, the receiver, and a confirmation boolean.

3.6.1 PROCESSING NARRATIVE (PSPEC)

When a user creates a Transaction object, a unique identifier is created and the following information is collected:

- Transaction amount
- Payer
- Receiver
- Timestamp (recorded automatically by the application)

The group leader must then confirm the transaction by invoking the `confirm()` method. This sets the Transaction object's `confirmed` flag to true, indicating that the exchange actually occurred.

3.6.2 INTERFACE DESCRIPTION

Transaction

```
new( double: amount, GroupMember: payer, GroupMember: receiver )
getID() : int
getPayer() : GroupMember
getReceiver() : GroupMember
getAmount() : int
getHasBeenPaid() : boolean
setHasBeenPaid() : boolean
confirm() : boolean
```

3.6.3 PROCESSING DETAIL

This class is responsible for handling exchanges between individual members of a group.

3.6.3.1 DESIGN CLASS HIERARCHY

The Transaction class has no parent or child classes. However, each instance of Group has an array of associated Transactions.

3.6.3.2 RESTRICTIONS/LIMITATIONS

Since the Transaction class is self-contained, there are no practical restrictions.

3.6.3.3 PERFORMANCE ISSUES

This class does not perform any computationally intensive tasks, nor does it require substantial resources. Consequently, there should be no performance problems for this class.

3.6.3.4 DESIGN CONSTRAINTS

This class encounters the same design constraints as the Bill class. Balances must be uploaded to the server as *changes*, not as absolute amounts. This prevents multiple transactions from conflicting with one another if pushed to the server simultaneously.

3.6.3.5 PROCESSING DETAIL FOR EACH OPERATION

- `new( double: amount, GroupMember: payer, GroupMember: receiver)`
    - When a user creates a transaction, he/she inputs an amount and a receiver. The Transaction constructor is called and this information is passed, creating a new Transaction object for that Group.

- `confirm() : boolean`
  - This method is responsible for determining and pushing the balance changes for both members involved in the transaction to the server.
- Accessors/mutators
  - These methods are used to obtain/modify Transaction information as needed. The only field that can be modified is hasBeenPaid.

3.7 SERVER COMPONENT

The server component of SplitPay is responsible for storing and synchronizing data across devices. The server ensures that information on all handsets is consistent, and that the central database is kept up-to-date. There will be a ServerComm class that manages communications between the SplitPay Android application and the server.

3.7.1 PROCESSING NARRATIVE (PSPEC)

The information stored in the database tables are made available to the SplitPay application (for both reading and writing) via multiple interfacing options, including:

- Push-to-Server
- Pull-from-Server
- Push-to-Phone
- Receive-from-Server
- Data will be exchanged between the client and server using the JSON format. Requests will be posted over HTTP using the REQUEST variables.

3.7.2 INTERFACE DESCRIPTION

ServerComm

```
pushToServer()
PullFromServer()
PushToPhone()
ReceiveFromServer()
```

3.7.3 PROCESSING DETAIL

The ServerComm methods defined above are relatively straightforward. There are no complex algorithms in this section as all that is taking place is a simple exchange of data. Three paradigms define the type of communication that will take place between the client and the server. These paradigms are covered in depth below.

3.7.3.1 DESIGN CLASS HIERARCHY

The classes on the server will mirror those of the local application in terms of data member fields. They will be independent classes with no external dependencies, except for communication layers for database queries and HTTP communication.

3.7.3.2 RESTRICTIONS/LIMITATIONS

Bandwidth and the presence of an internet connection are required in order to use these methods. If there is not one available, the methods will fail and the user will be notified. A caching function may be implemented in the future in order to batch updates that may need to be sent from the client until an internet connection is available.

3.7.3.3 PERFORMANCE ISSUES

Performance issues are not of concern in this section. The data being exchanged is only plaintext, so bandwidth will never be an issue. The exception to this assumption is the optional feature of receipt imaging, which would require potentially large files to be transferred to and from the server. This may create a bottleneck, or prove to be too taxing on network throughput. This feature, however, is not designated for inclusion in the first release of the application.

3.7.3.4 DESIGN CONSTRAINTS

The current design for the ServerComm component requires that each method be able to handle any object of any type. Thus, the serialization and deserialization functions will need to be incredibly robust and cannot rely on any assumptions about the data until it is decoded. Each method must contain logic to determine the type of object that it has deserialized and to call other server-side methods to handle these objects accordingly.

3.7.3.5 PROCESSING NARRATIVE (PSPEC) FOR EACH OPERATION

- `pushToServer()`
    - This method will be used by the local Android application and will be used to upload an object. The object will be serialized into JSON format and then sent over HTTP to the server. The server will then de-serialize the object, check its validity, and send a response to the client. The client will use this response to determine if the operation was successful or not.

- `PullFromServer()`
  - This method will issue a request for an object to the server, along with a unique identifier for that object (userID, groupID, etc.). The server will query the database and extract the requested object. The server will then serialize each object into a JSON object and send the JSON object as a response over HTTP. This method will gather the response, deserialize it, and return with the requested object.
- `PushToPhone()`
  - This method will be called on the server side, likely as the result of an action from another user. This method will take the object to be sent, serialize it into a JSON object, and send the object to the user determined by server side logic.
- `ReceiveFromServer()`
  - This method will be implemented as an Android service that runs in the background in order to receive notifications sent by PushToPhone() from the server. This method will deserialize the object that is being received and display the appropriate notification.

3.8 VIEW CLASS

This class is provided by the Android API. It acts as the foundation for all user interface components. SplitPay's various graphical elements will inherit from and interact with this class. These components are separate from the functional design of SplitPay, and thus they are covered outside of this section. See Part 4 (*User Interface Design*) below for more information.

4. USER INTERFACE DESIGN

The user interface consists of a set of menus through which the user can interact with data on the SplitPay server. These menus include a "Welcome" menu, a "Groups" menu, a "Group Management" menu, a "Group History" menu, a "Create a Bill" menu, and a "Transaction" menu. Each menu will contain fields for entering data required to perform a specified task. The user will interact with the menus through the device's touch screen.

4.1 DESCRIPTION OF THE USER INTERFACE

Each menu will consist of various GUI components, such as buttons, labels, text fields, and list objects. These components will be arranged in such a way that the user will be able to quickly grasp the purpose of each menu and perform whatever task it is designed for efficiently. A detailed description of these menus and their interactions with each other will be described in section 4.1.2.

4.1.1 OBJECTS AND ACTIONS

The next several pages describe and illustrate the following SplitPay menus:

For first time users…

- Welcome/Registration
- Groups
- Create Group

For returning users…

- Groups
- Create a Bill
- Member-to-Member Transaction
- View Group
- Group History
- Manage Group

FOR FIRST TIME USERS

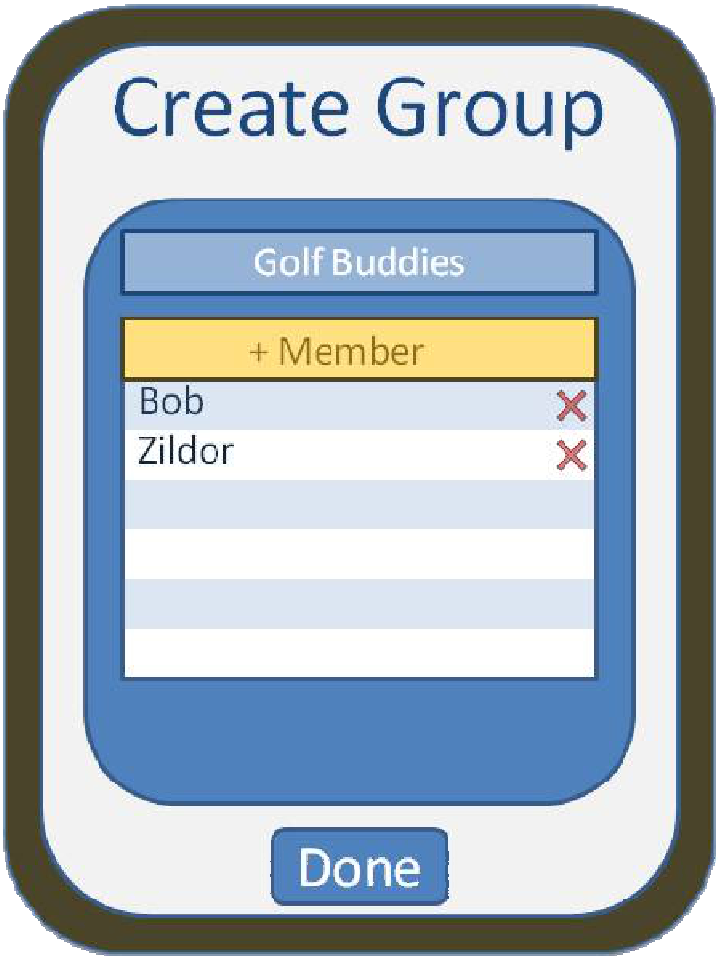
WELCOME/REGISTRATION

- The first time a user runs the SplitPay Android application on his/her phone, the "Welcome/Registration" menu will appear, which welcomes the user to the application and requests his/her email, cellphone number, and display name so a unique UserAccount can be created.

GROUPS

- Once the user has finished creating an account, he/she will be taken to the main "Groups" menu which will initially be empty. The user will have the option to create a new group. The user can create a new group by pushing the 'add group' button.

CREATE GROUP



- This will take them to the “Create Group” menu. On this menu, the user will be able to add members using the “+ Member” button. This will cause a popup menu to appear, prompting the user to enter the member’s email address so the server can determine if he/she has an existing SplitPay account. Once the user has created a group, he/she will have the option to create a bill within that group or carry out a transaction with another group member.

FOR RETURNING USERS

GROUPS (REVISITED)

- When a returning user runs SplitPay, the user will be taken directly to the groups menu. Any existing groups that the user is a member of will be displayed here. The user's current balance in each group will also be displayed here.
- From the groups menu, the user will be able to select a group and either add a bill to the selected group or simply view the details of the group.

CREATE A BILL

- When the user creates a bill, all members will be highlighted on the list to signify that they are splitting the bill. The user can remove members by touching their names on the screen. This deselects that member, and they will no longer be highlighted. Initially, the app will assume that the user creating the bill is also paying it, but the user can drag another member from the list to become the new payer.
- The user must also enter the *amount*, which will be split amongst the bill's participants.
- The user will be able to enter a name for the bill in the text field at the top of the menu.
- Once the user is finished, he/she will press the submit button, so the bill can be confirmed by the group leader through the "Group History" menu (detailed later in this section).

MEMBER-TO-MEMBER TRANSACTION

- When the user chooses to make a member-to-member transaction, he/she will be presented with a list of all group members. The user will be selected as the payer and will be expected to select a member from the list to receive the payment. The user may also drag a different member from the list to become the payer.
- The user will enter the amount being transferred between the payer and receiver in the text field at the top of the menu.
- Once the user is finished, he/she will press the submit button. As with new bills, transactions can be confirmed by the group leader through the "Group History" menu (covered later in this section).

VIEW GROUP MENU

- This menu will display all members of the selected group and their current balances. Balances will be displayed both graphically and textually.

- From this menu, users will be able to add a bill or transaction as explained in the previous sections.

- The user can access the group's history by tapping the "History" button on the screen.

- Only the group leader will have access to group management, which he/she can navigate to by tapping the "Manage" button.

GROUP HISTORY MENU

- This menu displays all of the bills and transaction that belong to the selected group.
- Selecting an item on the list will expand it, showing further details for that item.
- The group leader has the additional option of confirming or denying events. After the event has been confirmed, it will be sent to the server and synced to each device.

MANAGE GROUP MENU

- This menu is only accessible to the group leader.
- On this menu, the leader will be able to add new members by tapping "+ Member", or remove members by tapping the "X" next to the member's name.
- If a group is done adding bills, the leader can freeze the group by pushing the "Stop" button. This will prevent members from adding new bills, although they are still permitted to carry out transactions in order to resolve debts.
- The leader can choose to resolve the group's debts by pressing the button labeled "Resolve debts". This displays a list of transactions which representing the most efficient method for reconciling all debts. All group members can access this list from the "Group History" menu. The leader can then confirm these transactions once the transactions have actually occurred.
- Once all debts have been resolved, the leader can disband the group by pressing the "Disband" button.

4.2 INTERFACE DESIGN RULES

The interface design rules for SplitPay are derived from Ben Shneiderman's "Eight Golden Rules of Interface Design". The following list offers a description of each rule, as well as how the rule applies to SplitPay.

1. Strive for consistency.

- Consistent sequences of actions should be required in similar situations. Identical terminology should be used in prompts, menus, and help screens, and consistent commands should be employed throughout.
- For each menu in SplitPay: all of the major options are displayed at the bottom of the menu (i.e. "Add bill", "Manage", "Submit", etc.). All lists are displayed in the center of the menu (i.e. group members and group history). All data fields are displayed near the top of the menu.

2. Enable frequent users to use shortcuts.

- As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macrofacilities are very helpful to an expert user.
- For ease of use, users will be able to add a bill from the main groups menu by simply selecting the group and pressing the "Add bill" button.

3. Offer informative feedback.

- For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.
- Using Android's notification system, SplitPay will notify users whenever new information is successfully pushed to the server. Users will also be notified when the server is down, or they are unable to connect to the server.

4. Design dialog to yield closure.

- Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.
- Whenever a user finishes creating a bill or transaction, they will be notified that the process was completed successfully. This however, does not guarantee that the item was successfully pushed to server.

5. Offer simple error handling.

- As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.
- If fields are not properly filled out when a user presses the submit button for a menu, they will be notified, and suggestions will be made to help them fix the mistake.

6. Permit easy reversal of actions.

- This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.
- If leader adds a member incorrectly, he/she can remove the member and re-add. The leader can also deny bills and transactions that were created in error.

7. Support internal locus of control.

- Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
- Experienced operators will often be the leader of their groups. As leader, the user will have heightened control over the functionality of the group.

8. Reduce short-term memory load.

- The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.
- Each menu is oriented to a singular task, allowing menu to be quickly and easily understood by the user.

4.3 COMPONENTS AVAILABLE

Android provides a plethora of useful GUI components. The components that SplitPay will be using include the following:

- View
  - View class is what Android uses to display all of its GUI components.
  - All of SplitPay's menus will inherit from the android View class.

- AbsoluteLayout
  - The AbsoluteLayout component allows the arbitrary placement of components within its bounds.
  - SplitPay uses this component on each of its menus to arrange the buttons and text in a logical manner.
- Button
  - The Button component allows users to interact by pushing it. When a button is pushed it creates an event which can be handled by an EventHandler outside of the Button class.
  - SplitPay will handle most of its human-computer interaction through buttons. Each menu class will have its own unique EventHandlers to handle the user's inputs.
- TextView
  - The TextView component is used to display text in a linear manner.
  - SplitPay will use TextView to display menu labels and messages.
- ListView
  - The ListView component is used to display a list of text with a scroll bar. ListView allows the user to scroll and select/highlight objects displayed on the list
  - ListView generates an event whenever the user changes the selection of objects on the list.
  - SplitPay will use ListView to display the list of groups, members, and group history events.
  - Each menu of the application will define EventHandlers for when the ListViews are interacted with.

4.4 UIDS DESCRIPTION

- The User Interface Development System is provided by the ADT plug-in for Eclipse.
- Our UIDS generates XML code from a graphical drag and drop menu designer. The graphical components of the UIDS are somewhat limited, so much of the design will be done by modifying the XML code directly.
- The functional aspects of each menu will be implemented in Java. This encompasses EventHandlers and any menu class members.

5. RESTRICTIONS, LIMITATIONS, AND CONSTRAINTS

As time is a limiting factor, the optional features previously mentioned in the SRS document are not discussed at all in this document. This is due to the fact that these features will likely not be implemented within the allotted time for this projects completion. However, as a result of the highly modular design and organization of data – as well as unlimited expansion potential on the server side – implementing these optional features at a later date would be arguably easier than incorporating them into the first design.

Another limitation of the software is the lack of a web interface. While not included in the optional features (as it may be considered a product of its own), a web interface to the SpiltPay system would allow users with or without the Android application to use a web interface with all of the capabilities of the SplitPay application. Once the server for the client application has been developed, it would be possible to implement this interface with relative ease.

A constraint that is frequently mentioned in this document as well as the SRS is the requirement for the user to have internet access on their Android client. This is essential, as all data mutating actions make a call to the server in order to complete that action. A potential solution is an offline queue that stores actions to be sent to the server once an active internet connection is established. If any conflicting information has been uploaded by other users during the first users offline time, all of the first user's queued actions are discarded and the user is notified of this (and presented with the most recently changed data).

As the application frequently queries a server over the internet, care must be taken to ensure that large amounts of traffic are not being sent or received by our application, as this may dissuade users with costly data plans from using our application. Currently, we do not anticipate this being a problem, due our use of encoded JSON messages for passing data between the client and the server. However, if during testing we find our usage is too high, we will begin to investigate ways to decrease this usage. Possible ideas include requesting specific fields that have changed, rather than entire objects, and also compressing the JSON objects that are sent between client and server.

Another constraint imposed on the user of this software is that they must have an email address. This is used a unique identifier for each user and also provides an avenue for a user to re-register their account in the event that they switch phones (as the ID of the phone will provide authentication for a user).

6. TESTING ISSUES

Each class will be tested individually to make sure the functions and constructors are operating correctly. Then, once the program is assembled, it will be tested as a whole to ensure all of the components work together correctly.

6.1 TESTING CASES AND EXPECTED RESULTS

The types of tests to be conducted are specified below, including as much detail as is possible at this stage. Emphasis here is on black-box and white-box testing.

6.1.1 WHITE BOX TESTING:

While each class is being implemented, the developer assigned to that class will test to make sure each function is working. The developer is fully responsible for debugging his/her own code because the overhead of sharing code between developers has been deemed too costly. However, all code will be accessible through the provided version control system, so this rule may be violated if needed.

6.1.2 BLACK BOX TESTING:

Black Box Testing comprises a majority of the testing process. This will be done after all the components are assembled, and will consist of running through all possible situations that may occur in the use of the SplitPay application.

6.1.3 FEATURE TESTING

The following subsections provide a brief overview of the testing process for each feature.

6.1.3.1 ACCOUNT CREATION

- Description: correct data input
    - Input: Valid email, that is not already on the server
    - Output: Account is created on server, user taken to main page

- Description: incorrect data input
  - Input: invalid email
  - Output: Account is not created, user is asked for different email
- Description: incorrect data input
  - Input: email already exists on server and is an online account
  - Output: Account is not created, user is asked for different email and notified email already exist
- Description: incorrect data input
  - Input: email already exists on server and is an offline account
  - Output: Account is not created, existing account is set to online, user is sent to his main page.

6.1.3.2 CONNECTING TO SERVER

- Description: connection is established.
  - Input: Device tries to access the server, and succeeds
  - Output: Device pushes and pulls information as normal
- Description: connection cannot be established
  - Input: Device tries to access server and fails
  - Output: User is alerted that they are offline, no data is transferred to server. All changes are stored locally and temporarily.
- Description: connection is established after user has made changes while offline
  - Input: User is offline and makes changes that need to be pushed to the server; connection is established at a later time.
  - Output: device goes online; changes or either pushed to server or determined to be outdated.

6.1.3.3 CREATING GROUP

- Description: create group button is clicked
  - Input: group name is entered
  - Output: group created on server. User added to group, user set to leader, user's device goes to the new group's page.

6.1.3.4 ADDING MEMBERS TO GROUP

- Description: Add member button clicked
  - Input: Add member button clicked
  - Output: user is taken to a screen where he can enter an email address or select a person from his contacts.

- Description: Adds from contacts
  - Input: Valid email, already on server
  - Output: user is added to group
- Description: Adds from contacts
  - Input: Valid email, already on server
  - Output: user is added to group
- Description: correct data input
  - Input: Valid email, already on server
  - Output: user is added to group
- Description: correct data input
  - Input: Valid email, not on server
  - Output: offline account is created, user is added to group
- Description: incorrect data input
  - Input: invalid email
  - Output: Account is not created, user is asked for different email

6.1.3.5 ADDING BILL/ USER TO USER TRANSACTION

- Description: Add bill button clicked
  - Input: Add bill button clicked
  - Output: User is taken to the add bill screen
- Description: valid data
  - Input: positive real numbers
  - Output: Leader is notified; if he confirms then the bill is added, pushed to the server, group member balances are updated and pushed to the server. User is returned to group page. Group displays new bill.
- Description: invalid data
  - Input: negative numbers; characters
  - Output: user remains on screen and asked to enter valid data.

6.1.3.6 STOPPING GROUP

- Description: if leader, group is stopped
  - Input: leader stops group
  - Output: Group is set to stopped on server. No new bills are allowed. No new members can be added.

6.1.3.7 RESOLVE ALL DEBT

- Description: resolve all debt is selected.
    - Input: leader or receiver confirm transaction
    - Output: New transaction is created and pushed to server, all debt recomputed.

6.2 PERFORMANCE BOUNDS

Due to fact that the application is very demanding with respect to resources, execution time for all local actions should be negligible. This includes screen navigation, group management, bill/transaction creation, etc. Also, since data is exchanged with the server in small plaintext messages, interactions between the client and server should also take very little time.

In relation to the server component, it must uphold acceptable performance ability when negotiating the passing of information between server and the client. For the scope of this project, a simple PHP application implemented on a standard virtual dedicated server will suffice. As this server is only performing simple algorithms and database calls, it is not processing-intensive. The only expected issue involving the server (in terms of performance) involves the amount of groups and devices performing an interaction. This would be easily throttled by upgrading the bandwidth of the server. However, this will not be an issue for this project, as no more than 30-40 users will be utilizing the app. If the application is launched on the Android market, the server system will be revamped entirely.

6.3 CRITICAL SYSTEMS

The two most critical components of the Android application are that debt calculations must be accurate and that the server database must be updated correctly and contain no corrupted or false data.

Server interaction is of critical note. Without proper server setup and flow of information to phones and to the database, the app will be rendered useless. Extensive testing must be done to ensure the validity of the server implementation in regard to networking with the individual mobile phone apps. Since the functionality of the app depends on the ability to request and send information to the server, it is imperative that this works as expected. Testing procedures will call for verifying information is passed wholly and correctly to the app and likewise from the app to the server. PHP will drive the operation of this process by either receiving information or disseminating it (to/from the database).

6.4 TESTING CASES

The following table lists all currently scheduled test cases:

Feature	Cases
Server Communication	Connection is established. New account, New Group, new member, new bill, and new transaction are all created and sent to the server.
	Connection is not established. New account, New Group, new member, new bill, and new transaction are all created.
	Connection is not established. New account, New Group, new member, new bill, and new transaction are all created. Connection is then established.
Account creation	Correct information input.
	Incorrect information input.
Group created.	New group created.
Add member	Invalid input
	From contacts has account
	From contacts does not have account
	By email has account
	By email does not have account
Add bill/ Transaction	Correct values
	Invalid values
Stop Group	Try to add bills and members after group has been stopped.
Resolve All debts	Make transaction through this screen
	Make transaction through transaction screen
Algorithm	Create several groups, members and bills and make sure all data is accurate.
Performance	Time all actions to make sure device runs smoothly.

7. APPENDICES

7.1 PACKAGING AND INSTALLATION ISSUES

No special considerations are warranted here beyond installation of SplitPay app through Android Marketplace and/or running of the environment through the SDK emulator.

Packaging requires that the Android application be signed. The Android system will not install applications which are not signed. After this the final APK package will be zipped.

7.2 DESIGN METRICS TO BE USED

Stability will be determined by measuring and calculating the ease to change packages without affecting other packages within the application. Abstractness will be evaluated subjectively by analyzing code readability, modularization of classes and methods, and overall structure of the code.

7.3 SEQUENCE DIAGRAMS

See attached file: *sequence_diagrams.pdf*

7.4 UML DIAGRAM

See attached file: *UML_diagram.pdf*

NOTE: these files are included as attachments in order to preserve formatting.