TRAP

Travel Reimbursement Application Processing

Dylan Bettermann Bett0159@umn.edu
Andrew Helgeson Helge206@umn.edu
Brian Maurer Maure113@umn.edu
Ethan Waytas Wayt0012@umn.edu

11/8/2012

Design Document

1. Introduction

1.1 Purpose

This document is intended to give the design of the travel reimbursement application processing system for the development team. It will describe the system interface, provided database interfaces, internal form validation and processing, and the classes associated with these processes.
The intent of this specification is also to allow the development team to refine their understanding of the tool and to compare that understanding with the customer's requirements as stated in [1].

1.2 Definitions, acronyms, and abbreviations

This section gives definitions for terms used later in the document as well as the abbreviations or acronyms for those elements.

Application Program Interface (API) - provides a means for requesting program services through an interface [2].

Objected Oriented (OO) - a programming model that is viewed as a collection of interacting objects. An object stores related data and the methods (operations) that can be performed on that data.

ReimbursementApp (RApp) - A class in the design of TRAP. For brevity it may be referred to as a RApp.

RestrictedLineItem (RLI) - A class in the design of TRAP. For brevity it may be referred to as an RLI.

Travel Reimbursement Application Processing (TRAP) - the tool that will be created to process reimbursement applications as well as save in-progress and completed forms.

Unified Modeling Language (UML) - graphical language used to describe the functionality and components of logical systems.

1.3 System Overview

The University of Minnesota (UMN) accounting department needs a more efficient way to process expense reimbursement forms due to the time required to manually process the forms by hand.

The UMN accounting department had previously contracted a development team that created a front-end graphical user interface (GUI). This team left after implementing the GUI. The accounting department now needs the back-end program to actually process the reimbursement forms.

The **T**ravel **R**eimbursement **A**pplication **P**rocessing (TRAP) program will need to take a reimbursement form from a user, audit it and determine if there are errors (and alert the user). If the form has been filled out correctly (e.g. date formats) with allowed expenses, it will then be sent to the accountants for final approval. This document will outline use cases and requirements that will help the users, accountants and developers to understand what the TRAP system needs to achieve with regards to increasing the accuracy, reliability and efficiency of auditing reimbursement forms.

The TRAP system is intended for users who have access to grants and have eligible expenses that can be reimbursed through a grant. A sampling of potential users could be:

- A professor
- A graduate student
- A person eligible to use a grant from the University of Minnesota
    - This person may not necessarily be an employee of the University of Minnesota

In general, a user would be considered a person who has been awarded a grant and is able to claim expense reimbursement. Any potential claims for reimbursement will be viewed by an accountant who will have the final say on the release of money.

1.4 Design Objectives

The design of TRAP, as you will find in the following sections(2-5), is designed first and foremost for the easy modification of input validation and business logic rules. By easy we mean that it shall take little time to add, remove or modify rules within the system and it poses little risk to the rest of the system apart from the damage of not correctly implementing the given rule.

For usability, TRAP's design supports saving and loading of forms. TRAP does not try to persist these saved forms across more than one running instance of the program. This means that if the system crashes no previously saved forms will be available. TRAP automatically saves completed forms which can be loaded later using the same formId. TRAP also allows the removal of all saved forms as a way of maintaining the running system when there gets to be many saved forms.

2. Design Overview

2.1 Introduction

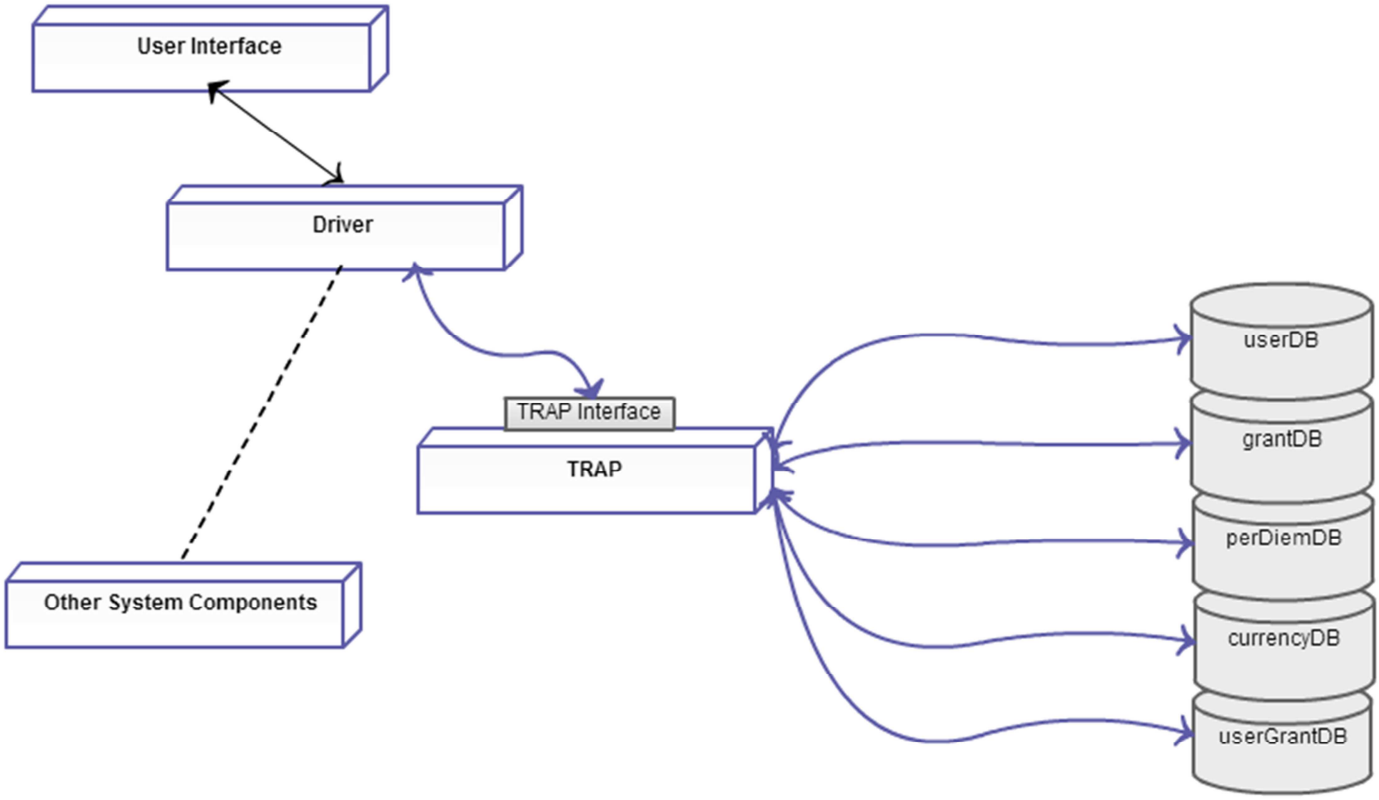
The TRAP system is designed using an object-oriented design. The architecture of the system is a call-return model. TRAP defines a simple interface defined in section 3.1.1 which provides this call-return functionality to the external environment.

TRAP is also a call return model internally. To load and save forms you call upon a storage unit (AllUserForms) which calls down a chain to find the appropriate form data or metadata to return. Processing forms calls upon a registry of rules that uses a modified observer pattern to check all rules. The rules being called are the input validation and the business logic modules. In turn, these modules may call their own methods or other interfaces to complete their role before returning.

In developing this design document Google Documents was used for the collaborative creation of this document. For UML diagramming an online service called Creately was used to easily collaborate on diagrams between group members. These online collaborative tools are essential to efficiently work as a geographically distributed team.

2.2 Environment Overview

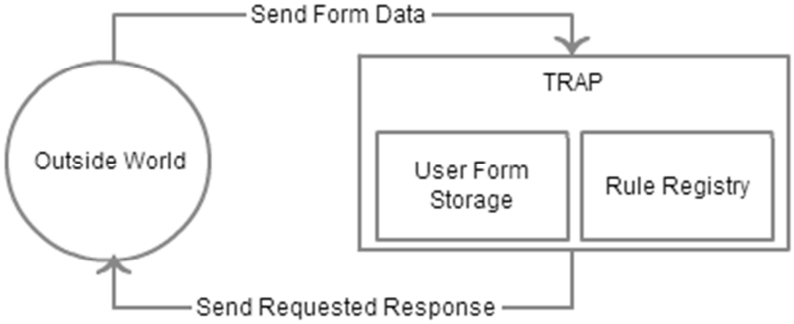
2.2.1 High-level view of TRAP's role in larger system



TRAP is being developed as a system to drop into a partially written existing system. TRAP has a very narrow communications scope with the rest of the system which is defined through the *TRAP Interface (*section 3.1.1). This is the only avenue through which TRAP can be invoked. TRAP does not make any assumption nor deals with any system component beyond the driver.

TRAP must also communicate with external databases which have their own api. For the purpose of information hiding, the database interfaces are being wrapped in a stable api.

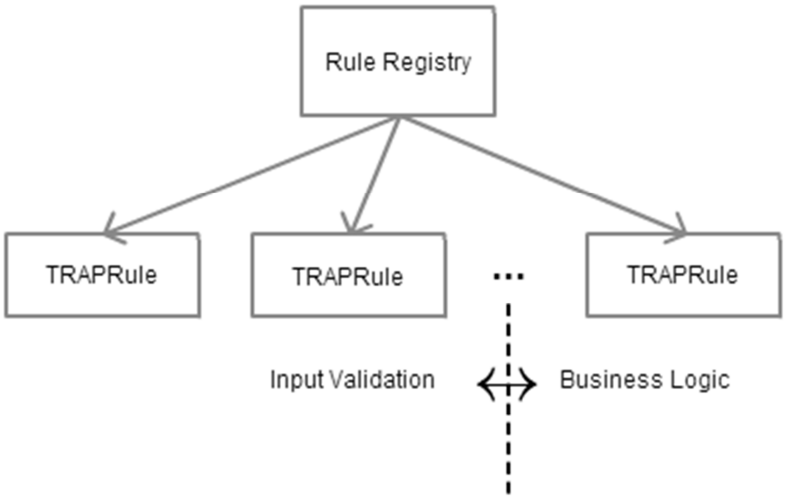
2.3 System Architecture



The system is comprised of three sub components: *TRAP*, the *Rule Registry*, and *User Form Storage*. These three components, combined with the Reimbursement Application (which is just our way of storing form data), make up the high level system architecture.

TRAP
TRAP is the subsystem that communicates with the outside world and the other two subsystems. It also facilitates the process to transform the data from the outside world into a Reimbursement Application that can be used by the *Rule Registry*.

Rule Registry
The rule registry is used to validate and check the rules which need to be enforced in the Reimbursement Application.

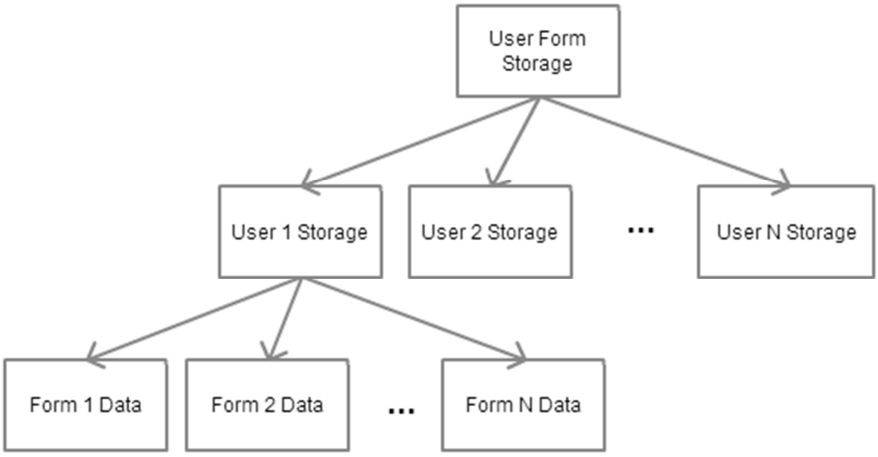


Check the constraints in section 2.4.3 for more information on the split between the input validation and business logic rules.

When TRAP receives a request to submit a form it will create a *ReimbursementApplication* object from the form's data and pass it to the RuleRegistry. The RuleRegistry will in turn process

the application using each of the registered rules. The term “register” is used in reference to the modified observer model being used for processing rules. The rules “register” to an observable which in this case is the single *ReimbursementApplication* object.

User Form Storage
The *User Form Storage* subsystem is used to load, save, and store user forms.



2.4 Constraints and Assumptions

2.4.1 Customer Constraints

- The implementation language for TRAP shall be Java. The interface provided to the rest of the system (the driver) is a Java interface.

2.4.2 Interface Constraints

- The interface provided with the driver is a simple one which only allows for a single user environment. The customer is aware of this but it is a point that could be looked at for improvement in the future. See related assumption (2.4.4.4) before making any changes.

2.4.3 Design Induced Constraints

- *Business logic* will change throughout the life of *TRAP* so it shall be efficient with respect to time and money to add, remove or modify *business logic.*
    - This is addressed through the use of a modified observer pattern. Input validation and business logic are hidden behind a rule class that is equivalent to the observer abstract type. The single observable object is the reimbursement application object that gets created when the form is submitted. It is created by converting the raw map data into the appropriate object hierarchy. This

"observable" application is passed into each rule for checking and/or processing. This model decouples the presence of a rule from how it is implemented. As long as rules are kept mutually exclusive, they can be added and removed at will. The only distinction we make is that "validation" rules that check input must run before business logic rules.

    - Other design models to support this non-functional requirement did not meet the same modularity and ease of maintenance that is required. Some of these other potential models and their problems are as follows:
        - A single or very few classes that implement rules; Essentially a massive logic block. This makes it tough to add or remove rules since they are coupled so tightly and anytime a method is added or removed it needs to be added or removed from the object calling the check/s.
        - The visitor pattern. This is a close second to the chosen design but it is too much engineering for the problem. Upon first glance it was not clear what should be the element and the visitor. While it could work, the chosen observer pattern is much more intuitive and its fit with this problem was much clearer.
- Because of the unordered observer pattern in use, TRAPRules must be independent and without a causal dependency between one another. The only exception to this is input validation checks which are always processed before business logic checks. Within these categories, input validation and business logic, there should be no assumption of the ordering that rules are checked against the *ReimbursementApplication*.
- Since the *ReimbursementApplication* (see the system UML) is created before rules are checked, it is up to some of the input validation to parse certain aggregate types. As an example, requirement 1.b specifies the format of dates entered into a form. In constructing the *ReimbursementApplication* we could add the logic to parse this format but then we would have split requirements logic (input and business) into multiple locations. Since we have specified that input validation ***must*** come first, it is possible to have the input validation check implementing requirement 1.b to parse the date itself and set the corresponding day/month/year fields on the date. If someone tries to access the specific fields of the date (ie day) before it has been set by an input rule and exception will be thrown.

2.4.4 Assumptions

1 In developing TRAP it is assumed the databases are filled with proper data and any value returned is correct. Since querying of the database is beyond the scope of the TRAP system control, this must be assumed (as the databases are behind an interface). However, this does not assume that the database is always available. Review non-functional requirement 6.a for database timeouts.

2 It is assumed that the *current user* has been previously authenticated. TRAP will accept any username to set as the current user under this assumption. The username (x500) must still appear in the user database.

3 It is assumed on good faith that *reimbursement forms* are being submitted for a single qualifying traveller. This must be assumed because short of blacklisting more terms in justification fields, *TRAP* cannot reliably enforce this from the form input. With the other business policies placed on *reimbursement forms* it would make it very tough for an individual to claim the full expenses for multiple people in one form.

4 It is assumed that the TRAP system will not be called concurrently. No plans to provision for the safety of concurrent actions within TRAP have been made. This will simplify the logic and reduce errors. If the single *current user* constraint (2.4.2.1) is lifted through a modification of the driver interface this assumption will need to be reconsidered.

3. Interfaces and Data Stores

3.1 System Interfaces

3.1.1 TravelFormProcessorIntf (Driver Interface)

This is the interface that the TRAP system implements and which allows the external world, the driver, to interact with it. This interface includes methods to set user, save form, load form, submit form and clear forms to name a few. For more in-depth information about this interface check the class diagram in section 4.1.5 and the class description for *TRAPImpl* in section 4.2.2.1.

3.1.2 userDB Interface

This interface is used to get the user's information from the TRAP system. Given the user's x500, a method in the interface will return the user's real name (in Last, First (MI.) form), the user's email address, and the user's employee id number.

3.1.3 grantDB Interface

This interface is used to communicate with the grant database in the TRAP system. It provides a method to get information about a grant as well as a method to update the balance of an account.

3.1.4 perDiemDB Interface

This interface is used to communicate with the perDiem database retrieves all meals or incidentals per diem. It provides methods to get the meals or incidentals per diem based on the location.

3.1.5 currencyDB Interface

This interface is used to communicate with the currency database to convert currency in the TRAP system. The interface provides a method to convert multiple currencies to USD based on the exchange rate of the day of the transaction.

3.1.6 userGrantDB Interface

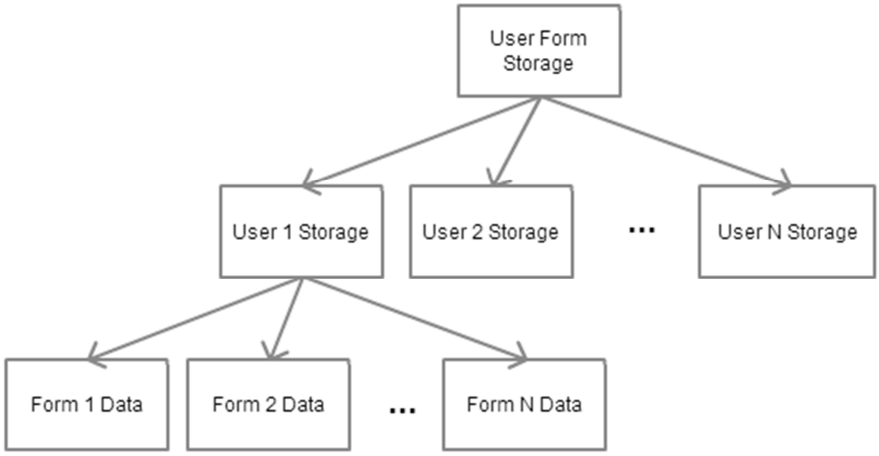
This interface provides a way to communicate with the userGrant database. This database contains associations between the users and the grants. This interfaces allows us to see this associations.

3.2 Internal Interfaces

3.2.1 TRAPRule Interface

This interface define how all rules within TRAP are called. They are called on a *checkRule* method which takes in a ReimbursementApp object. This object contains all the information necessary for a rule to do its checks and processing. This interface does not return any values. Any state change as a result of a rule occurs within the ReimbursementApp object. If there is a problem and an exception is thrown, it it propagated past this call to higher levels for processing.

3.3 Data Stores



The forms for any user of the TRAP system are saved internally. The above figure shows a large overview of the structure used to save forms. The description of this structure is as follows:

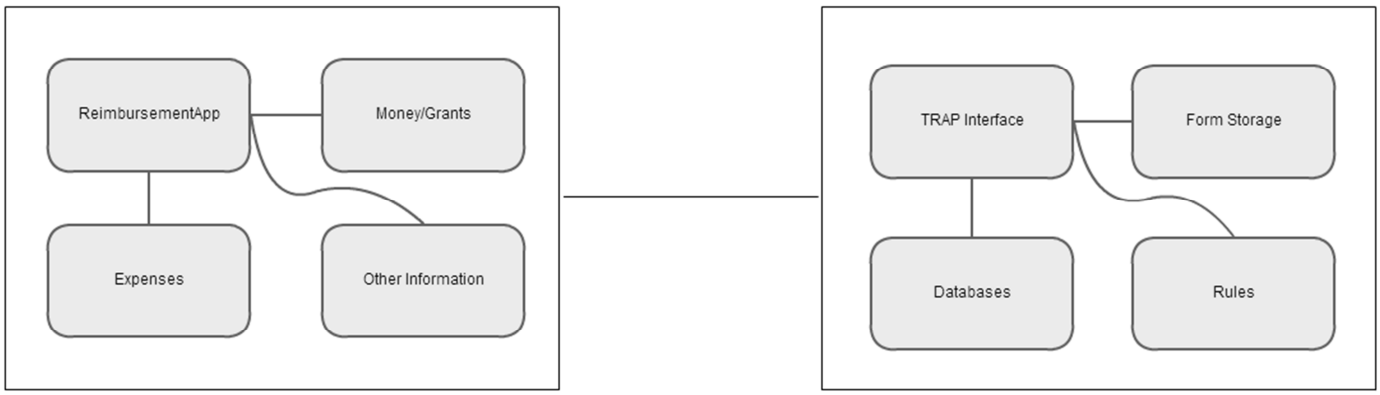
- User Form Storage (AllUserForms class)
  - This class holds a mapping of users to their form containers.
- User *N* Storage (SavedForms class)
  - This class holds a mapping of integers (form ids) to form containers
- Form *N* Data (FormContainer class)
  - This class holds a collection of items related to a form. Specifically the form data, a description of the form, and the form's status.

4. Structural Design

4.1 Class Diagram

Because of space limitations our class diagram must be broken into units in order to present. The full unabridged version of our class diagram is included with this document for further reference (see section 7.1 for more information on the extra materials).

Although it will be presented in pieces, here is a higher level abstract view of our class orginization.



The two main components of our TRAP design are the ReimbursementApp (RApp) data object and the TRAP interface itself. The RApp, as will be described more later, is a data object that serves as an alternate, stable interface to submitted data and is constructed from the raw form data. The RApp data object contains expenses, grant information, reimbursement amounts, and other information. The goal is that the RApp will contain all information necessary to process an application while separating rules from the raw format the data is received in which is currently a map of key,value pairs.

The TRAP interface is the interface to the external environment. This is the interface that the driver uses to request form operations (ie saving and loading), and submitting of forms. To carry out these operations there are other submodules for form storage and retrieval, database querying and the rules.

4.1.1 ReimubrsementApp

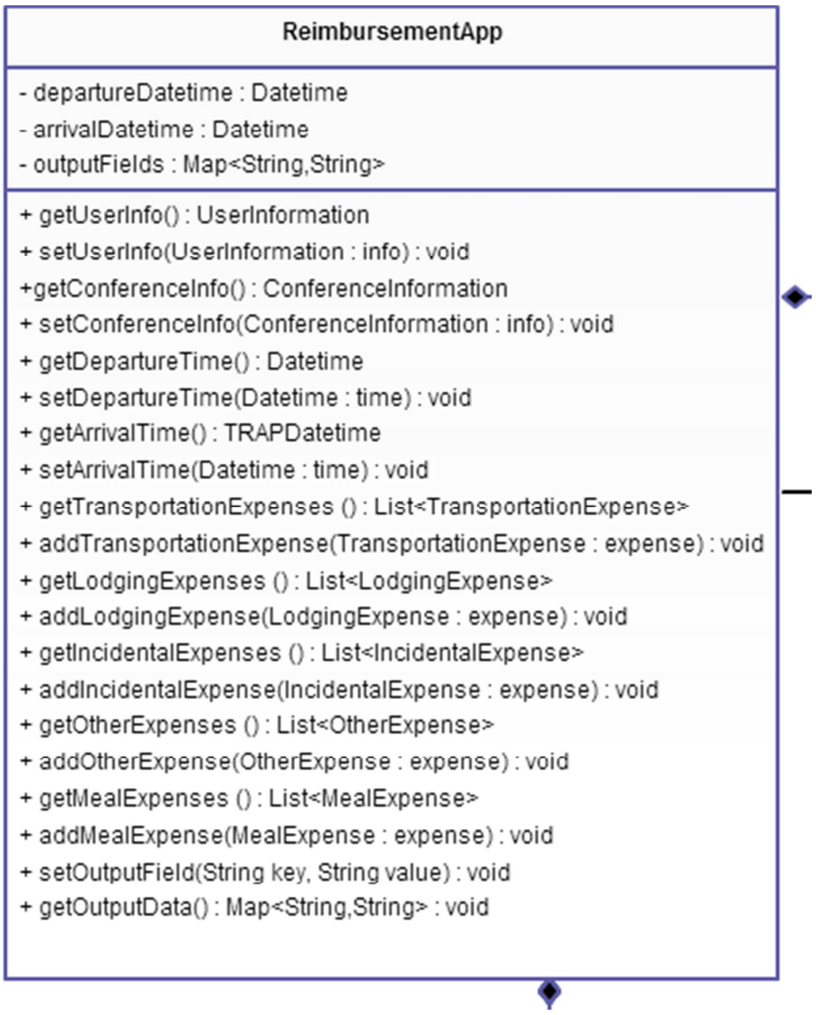


Figure 4.1.1 - ReimbursementApp

The ReimbursementApp (RApp) is the main class for our data object. This is passed to the TRAPRuleRegistry when it is time to check rules. This class and its contained attributes contain all the information required to process an application. The purpose for this, as described above is to hide the format of the raw input data (currently a map) from the rules that need to use it. This stable interface will reduce necessary work when the input method changes or the keys for input change.

Class Description: Section 4.2.1.1

4.1.2 Expenses

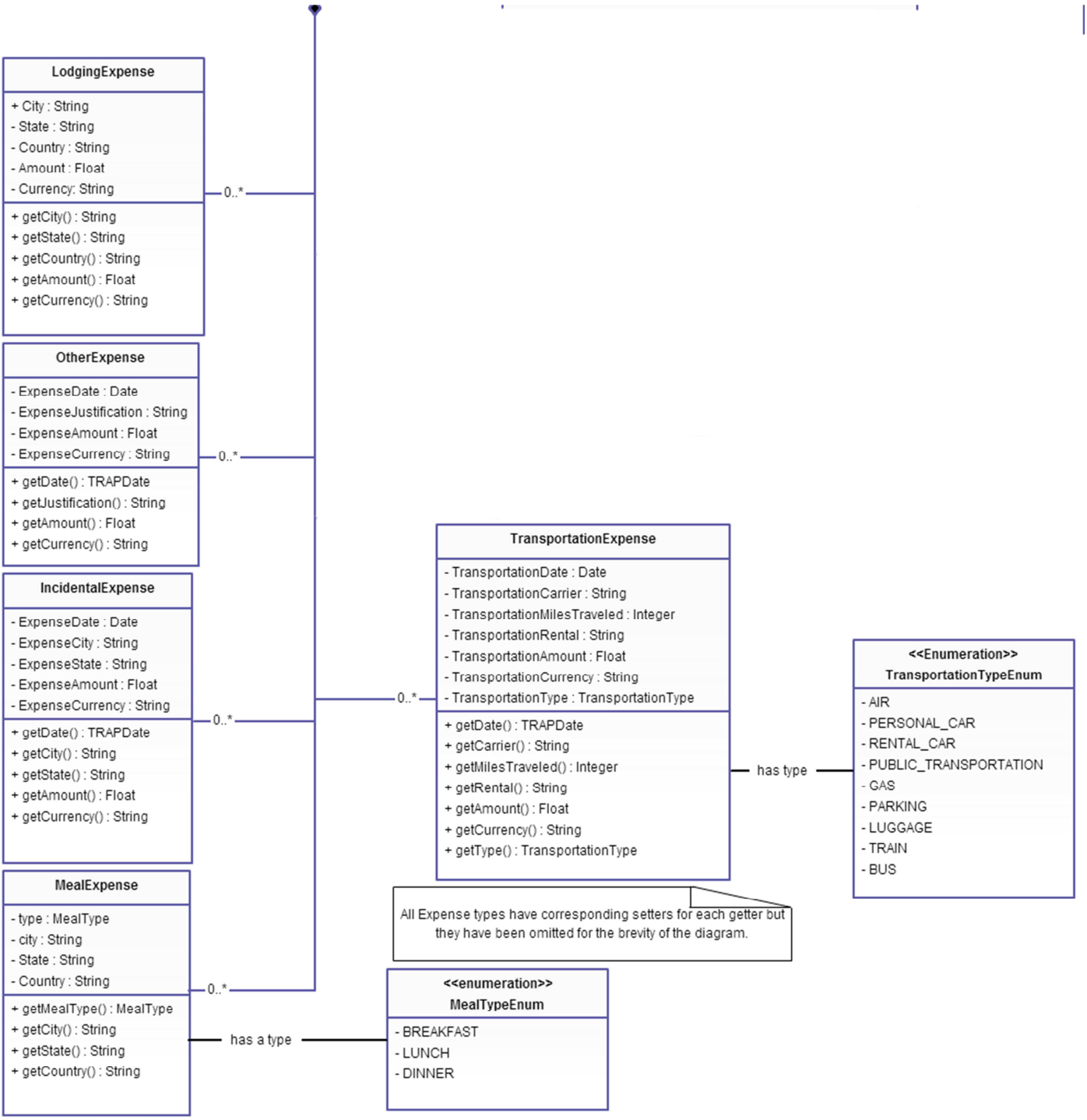


Figure 4.1.2 - Expenses

There is an expense class for each expense type. These are contained within the RApp and are created at the time the RApp is built. There can be zero or more of each of these inside a RApp.

The expenses do not inherit from a base class or implement a common interface because there is no cross-cutting similarity that we could abstract. It is also likely that rules will want to operate on specific types of expenses so the usefulness of a common interface is not necessary.

Class descriptions:

- Expenses are in section 4.2.4

4.1.3 Money and Grant Info

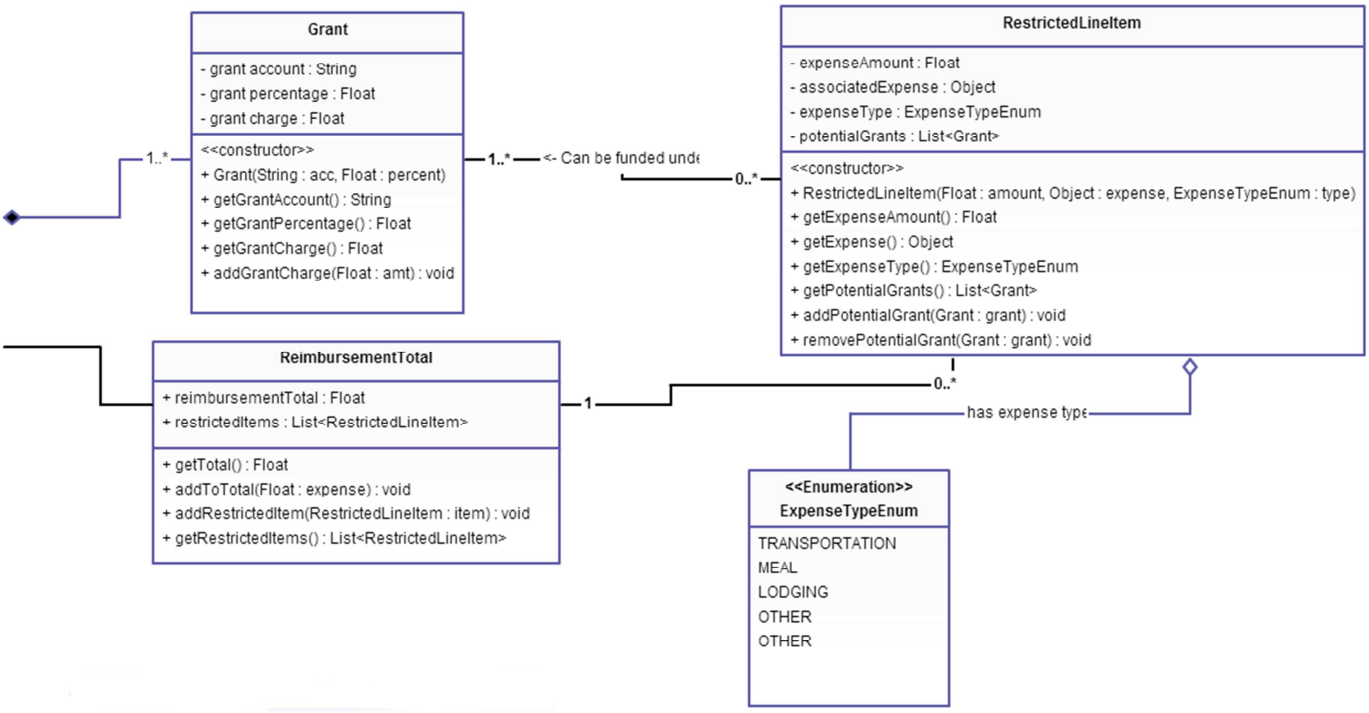


Figure 4.1.3 - Money and Grant Information

The RApp not only contains information converted from the raw form data but also has containers for outputs of processing. The ReimbursementTotal holds the running total for the reimbursement as well as any RestrictedLineItems (RLI). The RLI is a necessary container given the way this TRAP design splits expenses at the end of processing. For more detail on how the RLI works and why it is necessary, refer to section 4.3.

The Grant class is a container for grant information including the account name, the percentage of the reimbursement to fund through the grant and how much we have charged to it. Section 4.3 on the usage of RLI's also discusses how the grants are used.

The ExpenseTypeEnum, as its name implies is an emueration of the potential expense types and it is used to determine the type of a RestrictedLineItem. This makes it easier for rules to selectively check restricted items of a specific expense type.

Class descriptions:

- Grant - 4.2.1.6
- ReimbursementTotal - 4.2.1.4
- RestrictedLineItem - 4.2.1.5

4.1.4 Other App Information

Figure 4.1.4 - Other RApp Information

Despite being called “other”, these data classes are just as important to the processing of an application. The user information class is instantiated at the time the RApp is created. All fields of the UserInformation class are filled in from the userDB except for the username which we use to lookup the other fields. ConferenceInformation is another data class which is mostly a copy of the corresponding input fields.

Only one object of each of these classes is held by a RApp and they persist for the lifetime of the RApp and are destroyed thereafter.

Class Descriptions:

- UserInformation - 4.2.1.2
- ConferenceInformation - 4.2.1.3

4.1.5 TRAP Interface

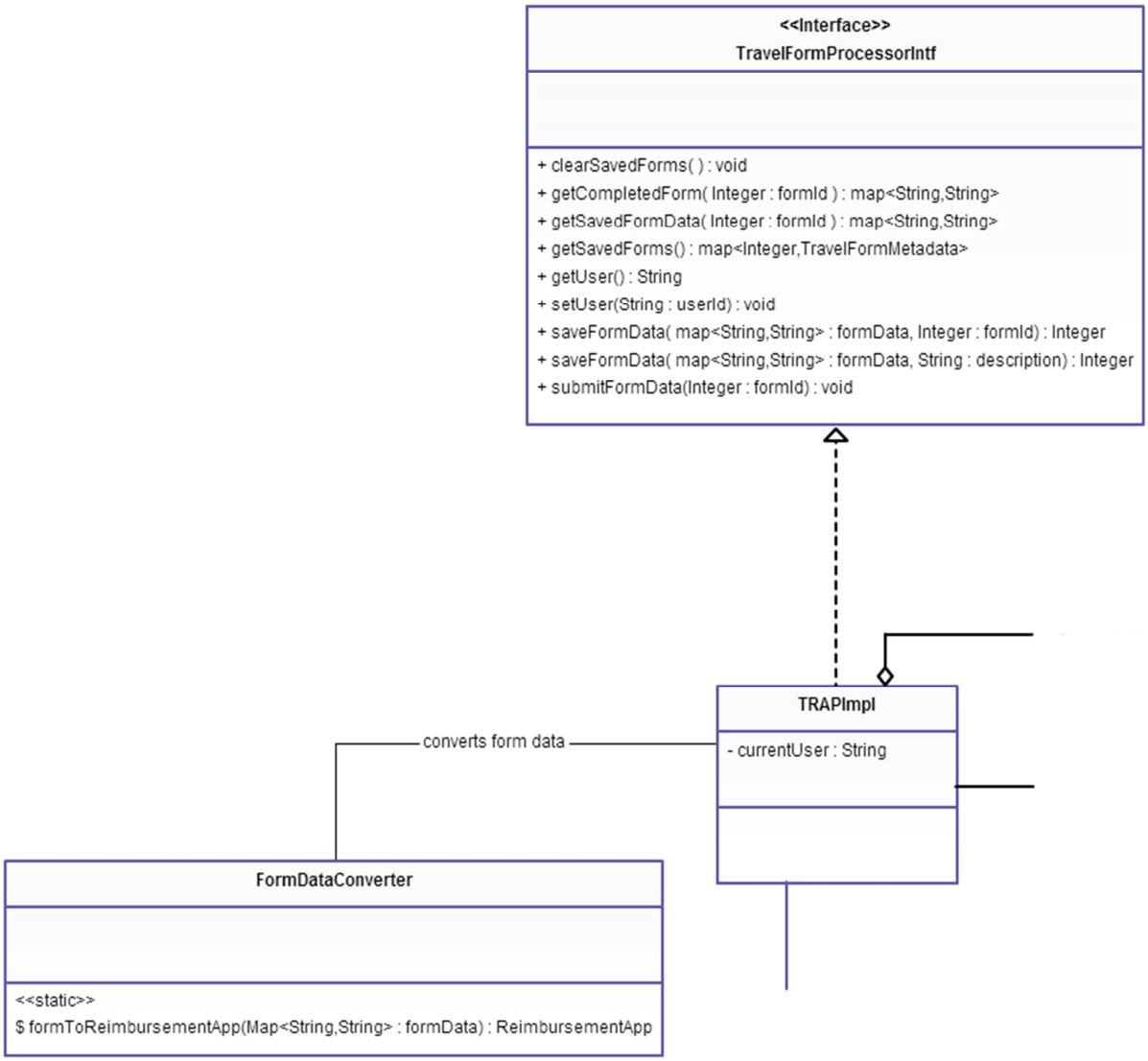


Figure 4.1.5 - TRAP Interface Class

The *TRAPImpl* class is the implementatin of the *TravelFormProcessorIntf* interface which provides the external interface for the driver to communicate with TRAP. Since *TRAPImpl* acts as the point of communication, all other functionality branches out from here including the form storage sub-module, the database wrappers, and the rule repository.

Class Description :

- TRAPImpl - 4.2.2.1
- FormDataConverter - 4.2.8.1

4.1.6 Form Storage

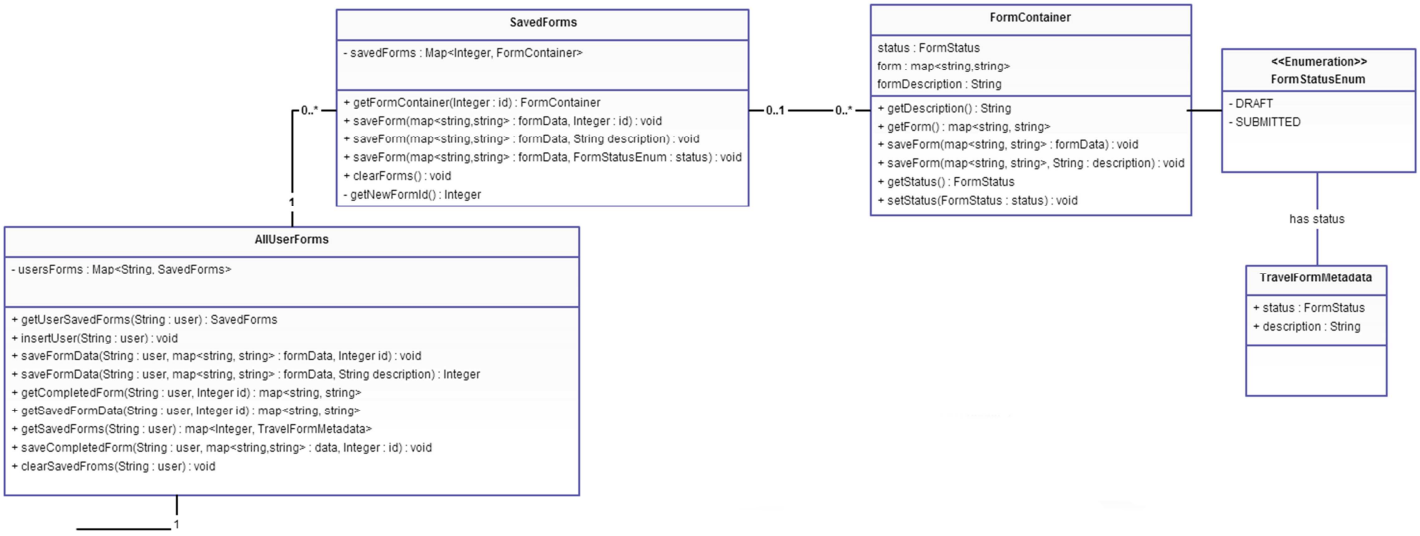


Figure 4.1.6 - Form Storage Classes

The form storage classes create the hierarchy for storing forms. Forms are first indexed by user in the *AllUserForms* class, then they are grouped by formId in the *SavedForms* class and finally we have a data container called *FormContainer*.

The *FormContainer* is used to group 3 things together for a user's saved form. That is the status of the form, DRAFT or COMPLETED as represented by the *FormStatusEnum*. It also holds the form's actual data which is the most critical piece. And finally it holds the description of the form. The *FormContainer* groups the data and metadata of a saved form in an easy to manage object.

Class Descriptions:

- Section 4.2.5.1 - 5

4.1.7 Databases

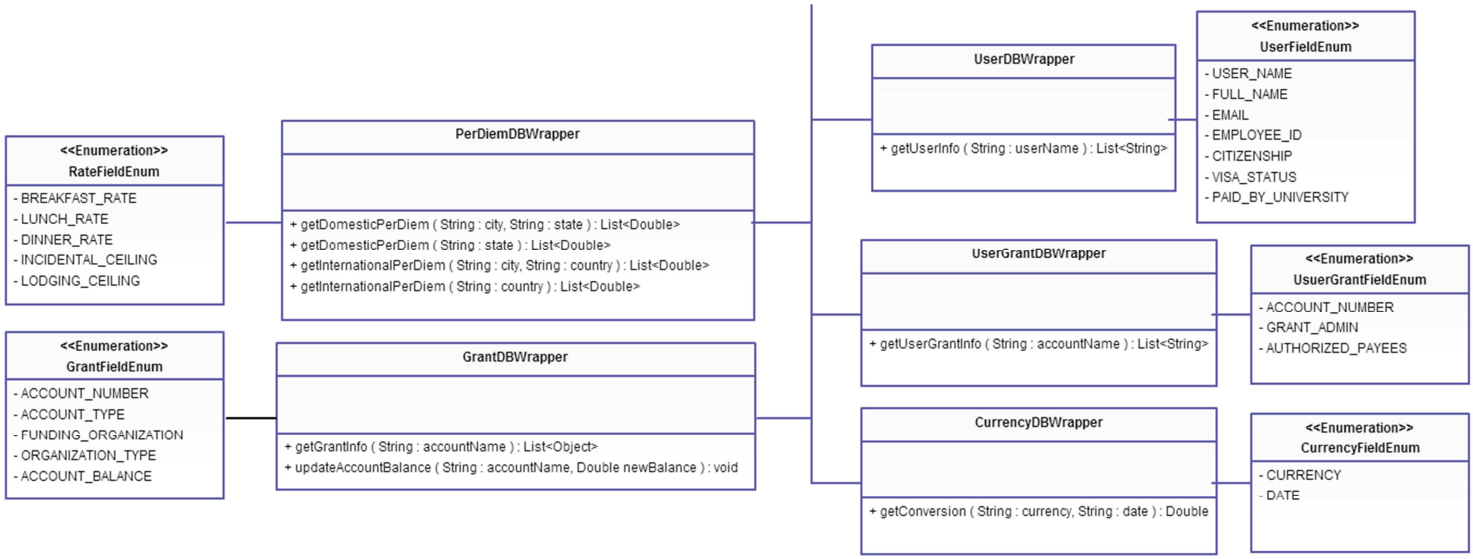


Figure 4.1.7 - Database Wrappers

The databases are another case of a design choice to protect the rest of the system from change. If the provided calls to the DB are changed, these wrappers can implement the necessary logic to give the same result or add new methods for new data.

Class Descriptions:

- Section 4.2.3.1 - 4.2.3.10

4.1.8 Rules

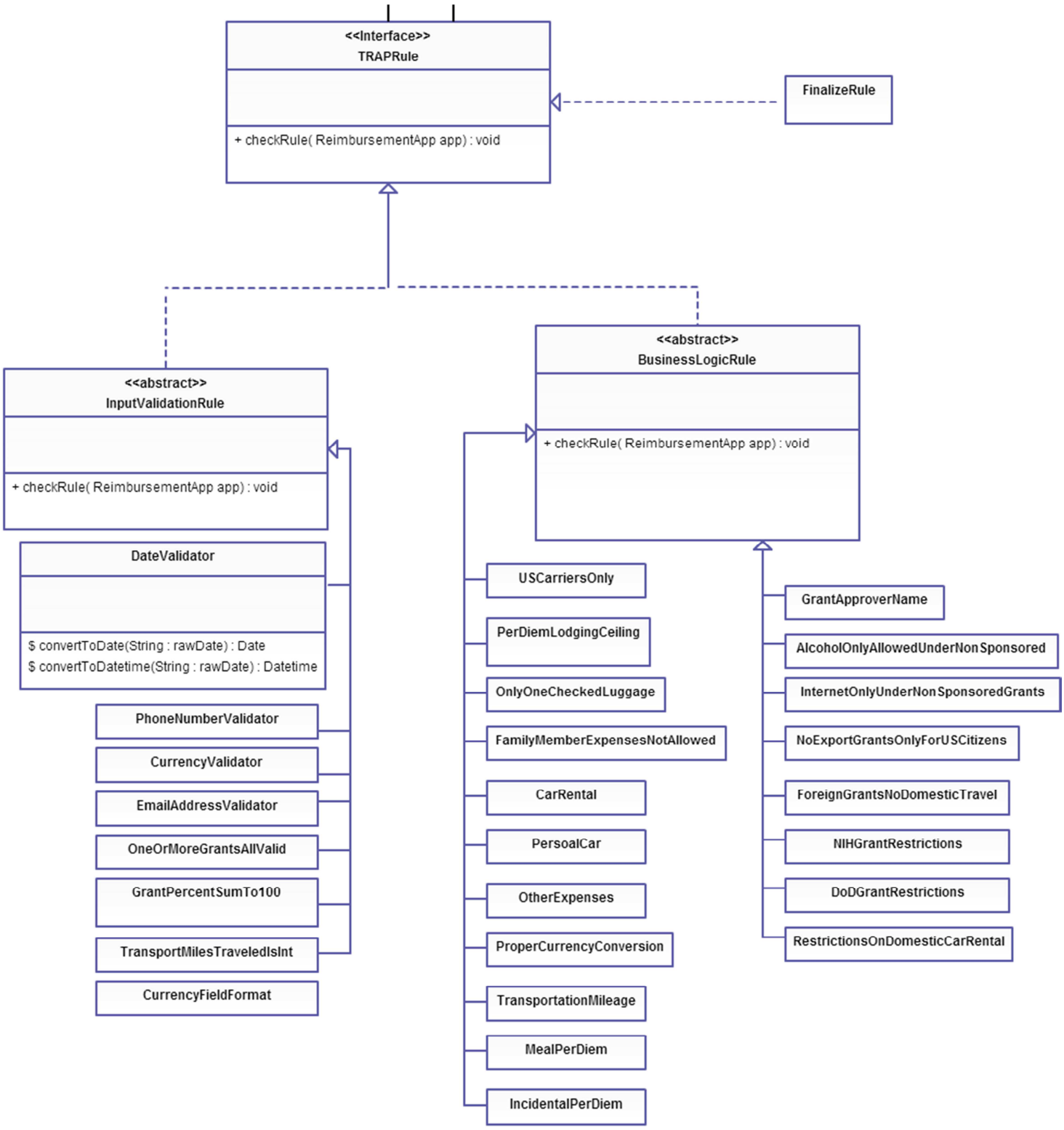


Figure 4.1.8 - Rules

To support the non-functional requirement on the ease of maintenance with respect to modifying rules, a design choice has been made to split the rules into modules. For more information on this requirement reference section 6.b in the requirements document or section 6.2 in this document.

If properly constructed, the modular nature of the rules makes them completely independent so they may come and go as needed with little change to the system.

Class Descriptions

- Section 4.2.7.1 - 4.2.7.30

4.1.9 Exceptions

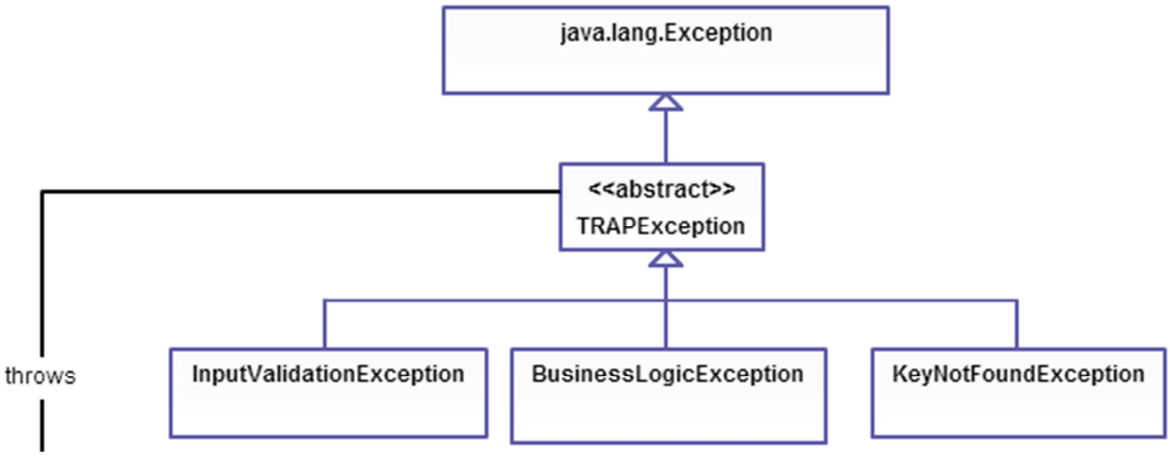


Figure 4.1.9 - Exception Types

The abstract exception type TRAPException extends the standard exception type. This abstract type is used for locations that want to be able to capture all TRAPExceptions. For rules and other methods that can raise exceptions they must be more specific of their error type by creating one of the concrete TRAPException types.

Each of these exceptions inherits the same methods from the abstract exception and in turn the standard exception. The purpose of the different types then is for TRAP to be able to selectively capture only certain errors for special processing and let other propagate.

Class Descriptions:

- Section 4.2.6.1 - 4.2.6.4

4.2 TRAP System Classes and Descriptions

For this document, a persistent object or some related data is an element that is present across calls to the TRAP interface. For example, an object that is instantiated for the purpose of a submission (ie *ReimbursementApplication)* would not exist when the submission interface call returns and therefore is not persistent.

4.2.1 Reimbursement Processing Data

4.2.1.1 Class: ReimbursementApp

- ***Purpose:*** *Contains all information necessary to process a submitted form.*
- ***Constraints:*** *Only one instance of this per audit.*
- ***Persistent:*** *Only for the duration auditing. Destroyed once auditing is done.*

4.2.1.1 Attribute Descriptions

1. ***Attribute:*** *departureDatetime*
   Type: *Datetime*
   Description: *Holds the departure datetime for when the trip began.*
   Constraints: *Must be set.*

2. ***Attribute:*** *arrivalDatetime*
   Type: *Datetime*
   Description: *Holds the arrival datetime for when the trip ended.*
   Constraints: *Must be set.*

3. ***Attribute:*** *outputFields*
   Type: *map<string, string>*
   Description: *Stores the output from auditing a form.*
   Constraints: *Filled once a form has passed auditing.*

4. ***Attribute:*** *lodgingExpenseList*
   Type: *List<LodgingExpense>*
   Description: *Stores lodging expenses*
   Constraints: *Size varies as the amount of lodging expenses.*

5. ***Attribute:*** *otherExpenseList*
   Type: *List<OtherExpense>*
   Description: *Stores other expenses*
   Constraints: *Size varies as the amount of others expenses.*

6. ***Attribute:*** *incidentalExpenseList*
   Type: *List<IncidentalExpense>*
   Description: *Stores incidental expenses*
   Constraints: *Size varies as the amount of incidental expenses.*

7. ***Attribute:*** *mealExpenseList*
   Type: *List<MealExpense>*
   Description: *Stores meal expenses*
   Constraints: *Size varies as the amount of meal expenses.*

8. ***Attribute:*** *transportationExpenseList*
   Type: *List<TransportationExpense>*
   Description: *Stores transportation expenses.*
   Constraints: *Size varies as the amount of transportation expenses.*

9. ***Attribute:*** *conferenceInformation*
   Type: *ConferenceInformatioin*
   Description: *Stores information about a conference attended.*
   Constraints: *Only one per ReimbursementApp*

10. ***Attribute:*** *userInformation*
    Type: *UserInformatioin*
    Description: *Stores information about a user.*
    Constraints: *Only one per ReimbursementApp*

11. ***Attribute:*** *grantList*

Type: *List<Grant>*
Description: *Stores grant information.*
Constraints: *Size varies as the amount of grants.*

12. ***Attribute:*** *reimbursementTotal*
Type: *ReimbursementTotal*
Description: *Stores the total amount to be reimbursed.*
Constraints: *Only one per ReimbursementApp*

4.2.1.2 Method Descriptions

1. ***Method:*** getUserInfo
Return Type: *UserInformation*
Parameters: *None*
Return Values: *Object containing information about the user*
Pre-Condition: *The ReimbursementApp has been initialized and userInformation has been set.*
Post-Condition: *The user information is returned.*
Attributes read/used: *userInformation*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getUserInfo
Processing Logic: *Call getUserInfo to return the user information.*
Test Case 1: *Initialize the ReimbursementApp. Set the userInformation attribute by calling setUserInfo. Call getUserInfo and check that user information is returned.*
2. ***Method:*** setUserInfo
Return Type: *void*
Parameters: *UserInformation : info*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The user information is set.*
Attributes read/used: *userInformation*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.setUserInfo
Processing Logic: *Call setUserInfo to set the user information.*
Test Case 1: *Initialize the ReimbursementApp. Set the userInformation attribute by calling setUserInfo. Call getUserInfo and check that user information is returned.*

3. ***Method:*** getConferenceInfo
Return Type: *ConferenceInformation*
Parameters: *None*
Return Values: *ConferenceInformation (returns information about a conference the user attended)*

Pre-Condition: *The ReimbursementApp has been initialized and conferenceInformation has been set.*
Post-Condition: *The conference information is returned.*
Attributes read/used: *conferenceInformation*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getConferenceInfo
Processing Logic: *Call getConferenceInfo to return the conference information.*
Test Case 1: *Initialize the ReimbursementApp. Set the conferenceInformation attribute by calling setConferenceInfo. Call getConferenceInfo and check that conference information is returned.*

4. ***Method:*** setConferenceInfo
Return Type: *void*
Parameters: *ConferenceInformation : info*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The conference information is set.*
Attributes read/used: *conferenceInformation*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.setConferenceInfo
Processing Logic: *Call setConferenceInfo to set the conference information.*
Test Case 1: *Initialize the ReimbursementApp. Set the conferenceInformation attribute by calling setConferenceInfo. Call getConferenceInfo and check that conference information is returned.*

5. ***Method:*** getDepartureTime
Return Type: *Datetime*
Parameters: *None*
Return Values: *Datetime (when the trip began)*
Pre-Condition: *The ReimbursementApp has been initialized and departureDateTime is set.*
Post-Condition: *The departure datetime is returned.*
Attributes read/used: *departureDateTime*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getDepartureTime
Processing Logic: *Call getDepartureTime to get the departure datetime of the trip.*
Test Case 1: *Initialize the ReimbursementApp. Set the departureDateTime attribute by calling setDepartureTime. Call getDepartureTime and check that the departure time is returned.*

6. ***Method:*** setDepartureTime
Return Type: *void*
Parameters: *Datetime : time*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The departure datetime is set.*
Attributes read/used: *departureDateTime*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.setDepartureTime
Processing Logic: *Call setDepartureTime to set the departure datetime of the trip.*
Test Case 1: *Initialize the ReimbursementApp. Set the departureDateTime attribute by calling setDepartureTime. Call getDepartureTime and check that the departure time is returned.*
7. ***Method:*** getArrivalTime
Return Type: *Datetime*
Parameters: *None*
Return Values: *Datetime (when the trip ended)*
Pre-Condition: *The ReimbursementApp has been initialized and arrivalDateTime is set.*
Post-Condition: *The arrival datetime is returned.*
Attributes read/used: *arrivalDateTime*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getArrivalTime
Processing Logic: *Call getArrivalTime to get the arrival datetime of the trip.*
Test Case 1: *Initialize the ReimbursementApp. Set the arrivalDateTime attribute by calling setArrivalTime. Call getArrivalTime and check that the departure time is returned.*

8. ***Method:*** setArrivalTime
Return Type: *void*
Parameters: *Datetime : time*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The arrival datetime is returned.*
Attributes read/used: *arrivalDateTime*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.setArrivalTime
Processing Logic: *Call setArrivalTime to set the arrival datetime of the trip.*
Test Case 1: *Initialize the ReimbursementApp. Set the arrivalDateTime attribute by calling setArrivalTime. Call getArrivalTime and check that the departure time is returned.*

9. ***Method:*** getTransportationExpenses
Return Type: *List<TransportationExpense>*
Parameters: *None*
Return Values: *List<TransportationExpense>*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of transportation expenses is returned (if any).*
Attributes read/used: *transportationExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getTransportationExpenses
Processing Logic: *Call getTransportationExpenses to get a list of transportation expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the transportation expenses by calling addTransportationExpenses. Call getTransportationExpenses and check that a list of transportation expenses are returned (if any).*

10. ***Method:*** addTransportationExpenses
Return Type: *void*
Parameters: *TransportationExpense : expense*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of transportation expenses is returned (if any).*
Attributes read/used: *transportationExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.addTransportationExpenses
Processing Logic: *Call addTransportationExpenses to get a list of transportation expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the transportation expenses by calling addTransportationExpenses. Call getTransportationExpenses and check that a list of transportation expenses are returned (if any).*

11. ***Method:*** getLodgingExpenses
Return Type: *List<LodgingExpense>*
Parameters: *None*
Return Values: *List<LodgingExpense>*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of lodging expenses is returned (if any).*
Attributes read/used: *lodgingExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getLodgingExpenses
Processing Logic: *Call getLodgingExpenses to get a list of lodging expenses.*

Test Case 1: *Initialize the ReimbursementApp. Set the transportation expenses by calling addLodgingExpenses. Call getLodgingExpenses and check that a list of transportation expenses are returned (if any).*

12. ***Method:*** addLodgingExpenses
Return Type: *void*
Parameters: *LodgingExpense : expense*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of lodging expenses is returned (if any).*
Attributes read/used: *lodgingExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.addLodgingExpenses
Processing Logic: *Call addLodgingExpenses to get a list of lodging expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the lodging expenses by calling addLodgingExpenses. Call getLodgingExpenses and check that a list of lodging expenses are returned (if any).*

13. ***Method:*** getIncidentalExpenses
Return Type: *List<IncidentalExpense>*
Parameters: *None*
Return Values: *List<IncidentalExpense>*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of incidental expenses is returned (if any).*
Attributes read/used: *incidentalExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getIncidentalExpenses
Processing Logic: *Call getIncidentalExpenses to get a list of incidental expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the incidental expenses by calling addIncidentalExpenses. Call getIncidentalExpenses and check that a list of incidental expenses are returned (if any).*

14. ***Method:*** addIncidentalExpenses
Return Type: *void*
Parameters: *IncidentalExpense : expense*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of incidental expenses is returned (if any).*
Attributes read/used: *incidentalExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.addIncidentalExpenses

Processing Logic: *Call addIncidentalExpenses to get a list of incidental expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the incidental expenses by calling addIncidentalExpenses. Call getIncidentalExpenses and check that a list of incidental expenses are returned (if any).*

15. ***Method:*** getOtherExpenses
Return Type: *List<OtherExpense>*
Parameters: *None*
Return Values: *List<OtherExpense>*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of other expenses is returned (if any).*
Attributes read/used: *otherExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getOtherExpenses
Processing Logic: *Call getOtherExpenses to get a list of other expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the incidental expenses by calling addOtherExpenses. Call getOtherExpenses and check that a list of other expenses are returned (if any).*

16. ***Method:*** addOtherExpenses
Return Type: *void*
Parameters: *OtherExpense : expense*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of other expenses is returned (if any).*
Attributes read/used: *otherExpenseList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.addOtherExpenses
Processing Logic: *Call addOtherExpenses to get a list of other expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the other expenses by calling addOtherExpenses. Call getOtherExpenses and check that a list of other expenses are returned (if any).*

17. ***Method:*** getMealExpenses
Return Type: *List<MealExpense>*
Parameters: *None*
Return Values: *List<MealExpense>*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of meal expenses is returned (if any).*
Attributes read/used: *mealExpenseList*
Methods Called: *Class names will be used in place of local variables.*

ReimbursementApp.getMealExpenses
Processing Logic: *Call getMealExpenses to get a list of meal expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the meal expenses by calling addMealExpenses. Call getMealExpenses and check that a list of meal expenses are returned (if any).*

18. ***Method:*** addMealExpenses
Return Type: *void*
Parameters: *MealExpense : expense*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The list of meal expenses is returned (if any).*
Attributes read/used: *otherMealList*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.addMealExpenses
Processing Logic: *Call addMealExpenses to get a list of meal expenses.*
Test Case 1: *Initialize the ReimbursementApp. Set the meal expenses by calling addMealExpenses. Call getMealExpenses and check that a list of meal expenses are returned (if any).*

19. ***Method:*** setOutputExpense
Return Type: *void*
Parameters: *String : key, String : value*
Return Values: *void*
Pre-Condition: *The ReimbursementApp has been initialized.*
Post-Condition: *The key:value pair has been set in the output field map.*
Attributes read/used: *outputFields*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.setOutputField
Processing Logic: *Call setOutputField to set a a key:value pair in outputFields.*
Test Case 1: *Initialize the ReimbursementApp. Set the output fields by calling setOutputField. Call getOutputData and check that a map of output key:value pairs are returned.*
20. ***Method:*** getOuputData
Return Type: *map<string,string>*
Parameters: *None*
Return Values: *map<string,string> (output fields for a form)*
Pre-Condition: *The ReimbursementApp has been initialized. There are key:value pairs in outputFields.*
Post-Condition: *A mapping of the output fields is returned.*
Attributes read/used: *outputFields*
Methods Called: *Class names will be used in place of local variables.*
ReimbursementApp.getOutputData

Processing Logic: *Call getOutputData return a mapping of the output fields.*
Test Case 1: *Initialize the ReimbursementApp. Set the output fields by calling setOutputField. Call getOutputData and check that a map of output key:value pairs are returned.*

4.2.1.2 Class: UserInformation

- ***Purpose:*** *Holds information related to the user.*
- ***Constraints:*** *Only one instance of this during form auditing*
- ***Persistent:*** *No, created when a form is submitted (and destroyed after auditing is done)*

4.2.1.2.1 Attribute Descriptions

1. ***Attribute:*** *Username*
   Type: *String*
   Description: *Stores the x500 of a user*
   Constraints: *Should not be the empty string (always set)*

2. ***Attribute:*** *Email Address*
   Type: *String*
   Description: *Stores the email address of a user*
   Constraints: *Should not be the empty string (always set)*

3. ***Attribute:*** *Emergency Contact Name*
   Type: *String*
   Description: *Stores the name of the person to contact in an emergency*
   Constraints: *Should not be the empty string (always set)*

4. ***Attribute:*** *Emergency Contact Phone*
   Type: *String*
   Description: *Stores the phone number of the person to contact in an emergency*
   Constraints: *Should not be the empty string (always set)*

4.2.1.2.2 Method Descriptions

1. ***Method:*** *getUsername*
   Return Type: *String*
   Parameters: *None*
   Return Values: *String - the user's name*
   Pre-Condition: *Username is set*
   Post-Condition: *Username is set and unchanged*
   Attributes read/used: *Username*
   Methods Called: *None*

Processing Logic: *Provides access to the user name.*
Test Case 1: *Call getUsername() and see if the result is a valid username (not the empty string) and in the userDB.*

2. ***Method:*** *getEmailAddress*
Return Type: *String*
Parameters: *None*
Return Values: *String*
Pre-Condition: *Email Address is set*
Post-Condition: *Email Address is set and unchanged*
Attributes read/used: *Email Address*
Methods Called: *None*
Processing Logic: *Provides access to the email address.*
Test Case 1: *Call getEmailAddress() and see if the result is a string (not the empty string). The email address will have previously been validated for being in the correct format.*

3. ***Method:*** *getEmergencyName*
Return Type: *String*
Parameters: *None*
Return Values: *String*
Pre-Condition: *Emergency Contact Name is set*
Post-Condition: *Emergency Contact Name is set and unchanged*
Attributes read/used: *Emergency Contact Name*
Methods Called: *None*
Processing Logic: *Provides access to the emergency contact name.*
Test Case 1: *Call getEmergencyName() and see if the result is a string (not the empy string).*

4. ***Method:*** *getEmergencyPhone*
Return Type: *String*
Parameters: *None*
Return Values: *String*
Pre-Condition: *Emergency Contact Phone Number is set*
Post-Condition: *Emergency Contact Phone Number is set and unchanged*
Attributes read/used: *Emergency Contact Phone Number*
Methods Called: *None*
Processing Logic: *Provides access to the emergency contact phone number*
Test Case 1: *Call getEmergencyPhone and see if the result is a string (not the empty string). The phone number will have previously been validated for being in the correct format.*

4.2.1.4 *Class:* ConferenceInformation

- ***Purpose:*** *Contains information related to the conference a user attended.*
- ***Constraints:*** *Only one instance of this per audit.*
- ***Persistent:*** *Only for the duration of the audit. Destroyed once auditing is done.*

4.2.1.4.1 Attribute Descriptions

1. ***Attribute:*** *JustificationForTravel*
   Type: *String*
   Description: *Holds justification for the trip*
   Constraints: *Must be set.*

2. ***Attribute:*** *JustificationConferenceTitle*
   Type: *String*
   Description: *Title of the conference attended on the trip.*
   Constraints: *Must be set.*

3. ***Attribute:*** *JustificationPresented*
   Type: *String*
   Description: *Is the traveler presenting?*
   Constraints: *Must be set if the user presented at a conference.*

4. ***Attribute:*** *JustificationPresentationAbstract*
   Type: *String*
   Description: *Abstract of presented work.*
   Constraints: *Must be set if JustificationPresented is set.*

5. ***Attribute:*** *JustificationPresentationAcknowledge*
   Type: *String*
   Description: *Acknowledge a grant during the presentation.*
   Constraints: *Must be set if JustificationPresented is set.*

6. ***Attribute:*** *JustificationNonSponsored*
   Type: *String*
   Description: *Justification for use of non-sponsored funds.*
   Constraints: *Must be set if using non-sponsored funds.*

7. ***Attribute:*** *JustificationSponsored*
   Type: *String*
   Description: *Justification for use of non-sponsored funds.*
   Constraints: *Must be set if using non-sponsored funds.*

4.2.1.4.1 Method Descriptions

A list of all the methods for this class are not listed due to redundancy. For each attribute, there is a getter method, named **get<attribute name>** and a setter method, named **set<attribute name>**. Every get method has exactly one parameter, a String type. Every set method returns a string.

4.2.1.5 *Class:* ReimbursementTotal

- ***Purpose:*** *Holds information about the money to be reimbursed for the current application. This object is updated throughout the process of rule checking and the total is split up in the end by the FinalizeRule.*
- ***Constraints:*** *Only one instance of this during form auditing.*
- ***Persistent:*** *No, created when a form is submitted (and destroyed after auditing is done). See our note at the top of 4.2 for our definition of persistent for these class descriptions.*

4.2.1.5.1 Attribute Descriptions

1. ***Attribute:*** *reimbursementTotal*
   Type: *Float*
   Description: *Stores the total amount of money to be reimbursed for this application.*
   Constraints: *Must be greater than or equal to 0.*

2. ***Attribute:*** *restrictedItems*
   Type: *List<RestrictedLineItem>*
   Description: *Stores the list of restricted expenses. These are the expenses that are restricted to being reimbursed under a certain subset of the provided grants.*
   Constraints: *The list should be present (instantiated) and not null but it may not have any items.*

3. ***Attribute:*** *Emergency Contact Name*
   Type: *String*
   Description: *Stores the name of the person to contact in an emergency*
   Constraints: *Should not be the empty string (always set)*

4.2.1.5.2 Method Descriptions

1. ***Method:*** *getTotal*
   Return Type: *Float*
   Parameters: *None*
   Return Values: *Float - the total reimbursement amount*
   Pre-Condition: *Class has been instantiated*
   Post-Condition: *Total has been returned and is unchanged in the object.*
   Attributes read/used: *reimbursementTotal*

Methods Called: *None*
Processing Logic: *Provides access to the reimbursementTotal value at the time it is called. This may not be the actual final total for the application if rules are still being checked.*
Test Case 1: *Create a ReimbursementTotal object, optionally add to the total, and then ask to get the total. The value you receive should be equal to the amount added or zero if nothing was added.*

2. ***Method:*** *addToTotal*
Return Type: *None*
Parameters:
   - Float : amount

Return Values: *None*
Pre-Condition: *Class has been instantiated and addToTotal may have been called zero or more times so far.*
Post-Condition: *The reimbursementTotal in this object has been incremented by the amount parameter.*
Attributes read/used: *reimbursementTotal*
Methods Called: *None*
Processing Logic: *If the amount parameter is less than zero and exception shall be raised. Otherwise the amount parameter shall be added to the reimbursementTotal attribute.*
Test Case 1: *Create a ReimbursementTotal object. Call getTotal and verify that the total is zero. Call addToTotal with a positive number x. Call getTotal and verify that the returned total is equal to what was just added, x.*
Test Case 2: *Create a ReimbursementTotal object. Call addToTotal with a negative amount. The expected result is that an exception is thrown.*

3. ***Method:*** *addRestrictedItem*
Return Type: *None*
Parameters:
   - RestrictedLineItem : item

Return Values: *None*
Pre-Condition: *Class has been instantiated.*
Post-Condition: *The list of RLI's in this object has had the provided item added to it.*
Attributes read/used: *restrictedItems*
Methods Called: *None*
Processing Logic: *If the provided item is null an exception shall be raised. Otherwise the provided item shall be added to the restrictedItems list.*

Test Case 1: *Create a ReimbursementTotal object. Call addRestrictedItem with a valid RLI r. Call getRestrictedItems and verify that r appears in the list.*
Test Case 2: *Create a ReimbursementTotal object. Call addRestrictedItem with a null RLI r. It is expected that an exception will be raised.*

4. ***Method:*** *getRestrictedItems*
Return Type: *List<RestrictedLineItems>*
Parameters: *None*
Return Values: *The list of RLI's that have been added to this ReimbursementTotal object.*
Pre-Condition: *Class has been instantiated.*
Post-Condition: *The list of RLI's in this object has been returned to the caller.*
Attributes read/used: *restrictedItems*
Methods Called: *None*
Processing Logic: *Return the restrictedItems attribute of this object.*
Test Case 1: *Create a ReimbursementTotal object. Call getRestrictedItems and verify that the result list is empty. Call addRestrictedItem with a valid RLI r. Call getRestrictedItems and verify that the RLI r is in the resulting list.*

4.2.1.6 *Class:* RestrictedLineItem

- ***Purpose:*** *Holds information pertaining to an expense that is restricted to being funded under a particular subset of the available grants.*
- ***Constraints:*** *Must be associated with an expense. An expense does not have to be restricted but if it is it must be associated with one and only one RestrictedLineItem.*
- ***Persistent:*** *No, created when a form is submitted (and destroyed after auditing is done).*

4.2.1.6.1 Attribute Descriptions

1. ***Attribute:*** *expenseAmount*
Type: *Float*
Description: *Stores the reimbursement amount for this restricted expense. This is different from any amount that may be stored in the expense itself. This amount is after any rules have been applied such as a per diem amount.*
Constraints: *Must be greater than or equal to 0.*

2. ***Attribute:*** *associatedExpense*
Type: *Object*

Description: *The object for the associated expense. Since there is no cross cutting interface for all expenses they are each their own type so to store them here we must store an Object. The expenseType described next is used to differentiate the actual type of the expense for retrieval later.*
Constraints: *Must not be null (a valid Object).*

3. ***Attribute:*** *expenseType*
Type: *ExpenseTypeEnum*
Description: *Enumerates the various expense object types. This is used to identify the type of the object that is stored in a RestrictedLineItem.*
Constraints: *Must be set to a value with the associatedExpenseAttribute.*

4. ***Attribute:*** *potentialGrants*
Type: *List<Grant>*
Description: *Stores the set of grants that this restricted expense can be funded under.*
Constraints: *None*

4.2.1.6.2 Method Descriptions

1. ***Method:*** *RestrictedLineItem*
Return Type: *None*
Parameters:
    - Float : expense amount
    - Object : expense object
    - ExpenseTypeEnum : type

Return Values: *None*
Pre-Condition: *None (it is the constructor)*
Post-Condition: *The new RestrictedLineItem object has been properly initiazed with the given data. This means that the amount, expense object, and expense type have been set with the provided values. The potentialGrants list shall also be initialzed to zero items.*
Attributes read/used: *All parameters. The following attributes: expenseAmount, associatedExpense, expenseType and potentialGrants.*
Methods Called: *None*
Processing Logic: *Sets internal state according to the constructor parameters to properly initialize the object.*
Test Case 1: *Call RestritedLineItem with appropriate values for the constructor and verify that you get back an object.*
Expected Output: A RestrictedLineItem object.

2. ***Method:*** *getExpenseAmount*
Return Type: *Float*
Parameters: *None*

Return Values: *expenseAmount*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*
Post-Condition: *The expenseAmount for this RestrictedLineItem has been returned. No state has been modified.*
Attributes read/used: *expenseAmount*
Methods Called: *None*
Processing Logic: *Returns the value of the expenseAmount attribute..*
Test Case 1: *Construct a RestrictedLineItem with a particular amount. Call getExpenseAmount on the same object and verify that the result is the same as the amount it was constructed with.*

3. ***Method:*** *getExpense*
Return Type: *Object*
Parameters: *None*
Return Values: *The expense object, associatedExpense.*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*
Post-Condition: *The associatedExpense for this RestrictedLineItem has been returned. No state has been modified.*
Attributes read/used: *associatedExpense*
Methods Called: *None*
Processing Logic: *Returns the associatedExpense attribute.*
Test Case 1: *Construct a RestrictedLineItem. Call getExpense and verify that the returned object is the same as the one the RestrictedLineItem was constructed with.*

4. ***Method:*** *getExpenseType*
Return Type: *ExpenseTypeEnum*
Parameters: *None*
Return Values: *expenseType - The type of the expense object attribute.*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*
Post-Condition: *The expenseType for this RestrictedLineItem has been returned. No state has been modified.*
Attributes read/used: *expenseType*
Methods Called: *None*
Processing Logic: *Returns the value of the expenseType attribute.*
Test Case 1: *Construct a RestrictedLineItem. Call getExpenseType on the same object and verify that the result is the same as the type it was constructed with.*

5. ***Method:*** *getPotentialGrants*
Return Type: *List<ExpenseTypeEnum>*

Parameters: *None*
Return Values: *potentialGrants - The list of grants which this RestrictedLine Item can be funded under.*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized*
Post-Condition: *The potentialGrants attribute for this RestrictedLineItem has been returned. No state has been modified.*
Attributes read/used: *potentialGrants*
Methods Called: *None*
Processing Logic: *Returns the potentialGrants attribute.*
Test Case 1: *Construct a RestrictedLineItem. Call getPotentialGrants on the same object and verify that the result list is empty. Call addPotentialGrant and verify that the added grants shows up in the result of calling getPotentialGrants. Do the same for removing a grant.*

6. ***Method:*** *addPotentialGrant*
Return Type: *None*
Parameters:
   - Grant : grant to add

Return Values: *None*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*
Post-Condition: *The Grant provided as a parameter has been added to the potentialGrants list inside the RestrictedLineItem.*
Attributes read/used: *potentialGrants*
Methods Called: *None*
Processing Logic: *Adds the provided Grant to the internal list of grants. If the provided Grant is already present in potentialGrants it shall not be added again.*
Test Case 1: *Construct a RestrictedLineItem. Call addPotentialGrant and verify that the grant was added by calling getPotentialGrants and checking if the Grant is in the list.*
Test Case 2: *Construct a RestrictedLineItem. Call addPotentialGrant to add a grant. Call addPotentialGrant again with the same grant. Call getPotentialGrants and verify that the grant only appears once.*

6. ***Method:*** *addPotentialGrant*
Return Type: *None*
Parameters:
   - Grant : grant to add

Return Values: *None*
Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*

Post-Condition: *The Grant provided as a parameter has been added to the potentialGrants list inside the RestrictedLineItem.*
Attributes read/used: *potentialGrants*
Methods Called: *None*
Processing Logic: *Adds the provided Grant to the internal list of grants. If the provided Grant is already present in potentialGrants it shall not be added again.*
Test Case 1: *Construct a RestrictedLineItem. Call addPotentialGrant and verify that the grant was added by calling getPotentialGrants and checking if the Grant is in the list.*
Test Case 2: *Construct a RestrictedLineItem. Call addPotentialGrant to add a grant. Call addPotentialGrant again with the same grant. Call getPotentialGrants and verify that the grant only appears once.*

7. ***Method:*** *removePotentialGrant*
   Return Type: *None*
   Parameters:
   - Grant : grant to remove

   Return Values: *None*
   Pre-Condition: *The RestrictedLineItem object has been constructed/initialized.*
   Post-Condition: *The Grant provided as a parameter has been removed from the potentialGrants list inside the RestrictedLineItem.*
   Attributes read/used: *potentialGrants*
   Methods Called: *None*
   Processing Logic: *Removes the provided Grant from the internal list of grants. If the grant is not found, a TRAPException shall be thrown.*
   Test Case 1: *Construct a RestrictedLineItem. Call removePotentialGrant before any have been added and check that a TRAPException is thrown.*
   Test Case 2: Construct a RestrictedLineItem. Call addPotentialGrant and verify that it was added. Call removePotentialGrant and verify that it was removed by calling getPotentialGrants and checking for an empty list.

4.2.1.7 *Class:* Grant

- ***Purpose:*** *Holds information related to a grant.*
- ***Constraints:*** *None*
- ***Persistent:*** *No, created when a form is submitted (and destroyed after auditing is done)*

4.2.1.7.1 Attribute Descriptions

1. ***Attribute:*** *grantAccount*
   Type: *String*
   Description: *Stores the name of the grant to be looked up in the database.*

Constraints: *Must not be null*

2. ***Attribute:*** *grantPercentage*
   Type: *Float*
   Description: *Represents the percentage of the reimbursement total that shall be funded from this grant.*
   Constraints: *Must be greater than or equal to 0 but no more than 100.*

3. ***Attribute:*** *grantCharge*
   Type: *Float*
   Description: *Stores the amount of money charged to this grant. This is set during the application finalization.*
   Constraints: *Must be greater than or equal to 0.*

4.2.1.7.2 Method Descriptions

1. ***Method:*** *Grant*
   Return Type: *None*
   Parameters:
   - *String : Account name*
   - *Float : Grant percentage*

   Return Values: *None (Constructor)*
   Pre-Condition: *None*
   Post-Condition: *The new Grant object has been properly initialized with the constructor parameters. The grantCharge shall also be initialized to 0.*
   Attributes read/used: *grantAccount, grantPercentage, grantCharge*
   Methods Called: *None*
   Processing Logic: *Takes the constructor parameters and initializes grantAccount and grantPercentage. It must also initialize grantCharge to 0.*
   Test Case 1: *Call the constructor on the object with the correct parameters and ensure that a valid object is returned.*

1. ***Method:*** *getGrantAccount*
   Return Type: *String*
   Parameters: *None*
   Return Values: *String - The account name for this grant.*
   Pre-Condition: *This grant object has been properly constructed/initialized.*
   Post-Condition: *The grantAccount attribute of this grant has been returned and no state has been modified.*
   Attributes read/used: *grantAccount*
   Methods Called: *None*
   Processing Logic: *Returns the grantAccount attribute of this object.*

Test Case 1: *Construct a Grant object. Call getGrantAccount and verify that the string returned is the same as the one the grant was constructed with.*

2. ***Method:*** *getGrantPercentage*
Return Type: *Float*
Parameters: *None*
Return Values: *Float - The percentage of the reimbursement total that this grant is responsible for.*
Pre-Condition: *This grant object has been properly constructed/initialized.*
Post-Condition: *The grantPercentage attribute of this grant has been returned and no state has been modified.*
Attributes read/used: *grantPercentage*
Methods Called: *None*
Processing Logic: *Returns the grantPercentage attribute of this object.*
Test Case 1: *Construct a Grant object. Call getGrantPercentage and verify that the value returned is the same as the percentage that the grant was constructed with.*

3. ***Method:*** *getGrantCharge*
Return Type: *Float*
Parameters: *None*
Return Values: *Float - The amount of money that has been charged to this grant at this point in the application processing.*
Pre-Condition: *This grant object has been properly constructed/initialized.*
Post-Condition: *The grantCharge attribute of this grant has been returned and no state has been modified.*
Attributes read/used: *grantCharge*
Methods Called: *None*
Processing Logic: *Returns the grantCharge attribute of this object.*
Test Case 1: *Construct a Grant object. Call getGrantCharge and verify that the value returned is the same as the expected amount at this point in the application. The amount returned shall be equal to the sum of all values that addGrantCharge has been called with. Initially this attribute will be 0.*

4. ***Method:*** *addGrantCharge*
Return Type: *Float*
Parameters:
    - Float : amount

Return Values: *None*

Pre-Condition: *This grant object has been properly constructed/initialized.*
Post-Condition: *The grantCharge attribute of this grant has been incremented by the value in the amount parameter. Nothing is returned.*
Attributes read/used: *grantCharge*
Methods Called: *None*
Processing Logic: *Increment the grantCharge attribute of this grant object by the amount given in the parameter. If the amount given is less than 0 a TRAPException shall be raised.*
Test Case 1: *Construct a Grant object. Call getGrantCharge and verify that the charge amount is zero. Call addGrantCharge with some amount x. Call getGrantCharge again and verify that the charge amount is equal to x.*

4.2.2 TRAP

4.2.2.1 *Class:* TRAPImpl

- ***Purpose:*** *The class that implements the TravelFormProcessorIntf for communication externally.*
- ***Constraints:*** *Only one instance of this class.*
- ***Persistent:*** *Yes. This is persistent for the uptime of the TRAP system.*

4.2.2.1.1 Attribute Descriptions

1. ***Attribute:*** *currentUser*
   Type: *String*
   Description: *Stores the x500 of a current user*
   Constraints: *Must be set before any form operations can take place. This includes form loading, saving or submitting.*

2. ***Attribute:*** *ruleRegistry*
   Type: *TRAPRuleRegistry*
   Description: *The registry of all rules which TRAP must check against a ReimbursementApp.*
   Constraints: *Must not be null. Must be a valid constructed object.*

3. ***Attribute:*** *formStorage*
   Type: *AllUserForms*
   Description: *Storage interface for form data across all users.*
   Constraints: *Must not be null. Must be a valid constructed object.*

4.2.2.1.2 Method Descriptions

1. ***Method:*** clearSavedForms
   Return Type: *None*
   Parameters: *None*
   Return Values: *None*

Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set.*
Post-Condition: *All form data and metadata for the current user in TRAPImpl have been removed.*
Attributes read/used: *currentUser, formStorage*
Methods Called: *formStorage.clearAllForms*
Processing Logic: *Call the formStorage attribute with the currentUser to clear all the user's saved forms and metadata.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling saveFormData with some data and a description. Verify that this form is saved by calling getSavedForms and checking that the form is present. Call clearSavedForms and check the output of getSavedForms again, it should now be empty.*

2. ***Method:*** getCompletedForm
Return Type: *Map<String,String>*
Parameters:
    - Integer : formId

Return Values: *The map of key,value pairs of the completed form's data who has an id equal to the given parameter.*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A form has been submitted and passed processing so that it is now marked complete.*
Post-Condition: *The completed form data whose formId is equal to the parameter has been returned to the caller.*
Attributes read/used: *currentUser,formStorage*
Methods Called: *formStorage.getCompletedForm*
Processing Logic: *Call the formStorage attribute's getCompletedForm method with the currentUser and formId. Return the completed form's saved data to the caller. In the case of an error an Exception shall be raised and propagated back to the caller.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling saveFormData with some data and a description. Verify that this form is saved by calling getSavedForms and checking that the form is present. Call clearSavedForms and check the output of getSavedForms again, it should now be empty.*

3. ***Method:*** getSavedFormData
Return Type: *Map<String,String>*
Parameters:
    - Integer : formId

Return Values: *The map of key,value pairs of the saved form's data who has the same id as the given formId parameter.*

Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A form with an id of formId has been saved in the system.*
Post-Condition: *The saved form data whose formId is equal to the parameter has been returned to the caller.*
Attributes read/used: *currentUser,formStorage*
Methods Called: *formStorage.getSavedFormData*
Processing Logic: *Call the formStorage attribute's getSavedFormData method with the currentUser and formId param. Return the saved form data to the caller. In the case of an error an Exception shall be raised and propagated back to the caller.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling saveFormData with some data and a description. Verify that this form is saved by calling getSavedForms and checking that the form is present.*
Test Case 2: *Initialize the TRAP system. Set a current user by calling setUser. Call getSavedForms when no forms have been added or using an invalid formId. Verify that an exception is raised.*

4. ***Method:*** getSavedForms
Return Type: *Map<Integer,TravelFormMetadata>*
Parameters: None
Return Values: *The map of key,value of form id's to their metadata object for the current user.*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set.*
Post-Condition: *All (formId,TravelFormMetadata) pairs of saved forms for the current user have been returned as a map of formId -> metadata.*
Attributes read/used: *currentUser,formStorage*
Methods Called: *formStorage.getSavedForms*
Processing Logic: *Call the formStorage attribute's getSavedForms method with the currentUser. Return the map of form ids to form metadata to the caller. In the case of an error an Exception shall be raised and propagated back to the caller.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call getSavedForms and verify that an empty map is returned.*
Test Case 2: *Initialize the TRAP system. Set a current user by calling setUser. Save a new form by calling saveFormData with data and a description. Call getSavedForms and verify that the saved form shows up in the map.*

5. ***Method:*** getUser
Return Type: *String*

Parameters: None
Return Values: *The currentUser attribute of the TRAPImpl object.*
Pre-Condition: *TRAPImpl has been constructed and initialized.*
Post-Condition: *The value of the currentUser attribute has been returned to the caller. If the system has not had a user set yet this will be the empty string.*
Attributes read/used: *currentUser*
Methods Called: *None*
Processing Logic: *Return the currentUser attribute of this object.*
Test Case 1: *Initialize the TRAP system. Call getUser and verify that the returned string is empty. It should not be null nor have any characters.*
Test Case 2: *Initialize the TRAP system. Set a current user by calling setUser. Call getUser and verify that the current user returned is the one that was just set.*

6. ***Method:*** setUser
Return Type: *None*
Parameters:
    - String : username

Return Values: *None*
Pre-Condition: *TRAPImpl has been constructed and initialized.*
Post-Condition: *If the new username is valid, the value of the currentUser attribute has been set to the provided username parameter. Nothing is done with the previous value of currentUser and it is not checked before setting to the new user.*
Attributes read/used: *currentUser*
Methods Called: *UserDB.getUserInfo*
Processing Logic: *Check the username parameter in the userDB to verify its validity. If it is not valid raise a TRAPException, otherwise set the value of the currentUser attribute to the given username parameter.*
Test Case 1: *Initialize the TRAP system. Call setUser with a valid username x. Call getUser and verify that the returned user is equal to x.*
Test Case 2: *Initialize the TRAP system. Call setUser with an invalid username x. Call getUser and verify that the returned user is equal to the empty string.*
Test Case 3: *Initialize the TRAP system. Call setUser with an valid username x. Call setUser again with an invalid username y. Call getUser and verify that the returned user is equal to the x, the original valid username.*

7. ***Method:*** saveFormData
Return Type: *Integer*
Parameters:
    - *Map<String,String> : formData*

- *Integer : formId*

Return Values: *The formId the formData was saved under. This will be the same as the given formId parameter.*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A form has been previously saved with a description so that it is assigned an id.*
Post-Condition: *The provided formData has been saved under the provided formId for the currentUser. The formId has been returned to the caller.*
Attributes read/used: *currentUser,formStorage*
Methods Called: *formStorage.saveFormData*
Processing Logic: *Call the formStorage attribute's saveFormData method with the currentUser, formId and formData. Return to the caller the formId that the formData was saved under. In the case of an error an Exception shall be raised and propagated back to the caller.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with data and a description to save a new form. Call saveFormData with new data and the assigned formId with. Call getSavedFormData with the assigned formId and verify that last version of the form data is returned, not what was initially set with the description.*
Test Case 2: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with data and an id and verify that an exception is thrown.*

8. ***Method:*** saveFormData
Return Type: *Integer*
Parameters:
   - *Map<String,String> : formData*
   - *String : description*

Return Values: *The formId the formData was saved under. This formId will be assigned by the formStorage object attribute.*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set.*
Post-Condition: *The provided formData has been saved under a new formId with the given description attached to its metadata. The status of this saved form in the metadata shall be set to DRAFT.*
Attributes read/used: *currentUser,formStorage*
Methods Called: *formStorage.saveFormData*
Processing Logic: *Call the formStorage attribute's saveFormData method with the currentUser, formId and description. Return to the caller the formId that the formData was saved under. This formId will be assigned and returned by the formStorage object attribute. In the case of an error an Exception shall be raised and propagated back to the caller.*

Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with data and a description to save a new form. Call getSavedFormData with the assigned formId and verify that the same form data is received.*

9. ***Method:*** submitFormData
Return Type: *None*
Parameters:
   - *Integer : formId*

Return Values: *None*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A form has been previously saved under the given formId or else an exception will be thrown.*
Post-Condition: *The form data associated with formId will be converted to a ReimbursementApplication and put through the set of checks. If all processing completes, the data associated with formId will be overwritten by the output data and marked as completed. If there are any problems, a TRAPException will be raised and the original form data will remain unchanged.*
Attributes read/used: *currentUser,formStorage, ruleRegistry*
Methods Called: *formStorage.getSavedFormData, FormDataConverter.formToReimbursementApp, ruleRegisty.startProcessing*

Processing Logic:
For any of these steps, it an exception is thrown it will be propagated to the caller for handling.

1. *Retrieve saved form data for the provided formId.*
2. *Create a ReimbursementApp from this form data by calling the static method FormDataConverter.formToReimbursementApp.*
3. *Pass this ReimbursementApp to the ruleRegistry (attribute) method startProcessing.*
4. *If the startProcessing method returns and no exceptions were thrown then the form processing completed successfully.*
5. *Grab the output from the processing by calling the ReimbursementApp.getOutputData method and save it back to the same formId in the formStorage using the formStorage.setFormCompleted method.*

Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with complete data and a description to save a new form. Call submitFormData to process the application. Call getCompletedForm with the assigned formId and you shall receive an output map that is different from your input.*

Test Case 2: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with incomplete or invalid data and a description to save a new form. Call submitFormData to process the application. Verify that an exception was thrown and that the original form data is untouched by calling getSavedFormData.*

4.2.3 Database

4.2.3.1 *Class*: UserDBWrapper

- ***Purpose****: This class contains additional information about the user that is necessary for the TRAP system.*
- ***Constraints****: A username (X500) must be supplied before the additional information can be retrieved.*
- ***Persistent:*** *Yes. This is persistent for the uptime of the TRAP system.*

4.2.3.1.1 Attribute Descriptions

1 ***Attribute:*** *userInfo*
Type*: Map<String, List<String>>*
Description*: This object will eventually contain information about the user.*
Constraints*: Must not be null.*

4.2.3.1.2 Method Descriptions

1 ***Method:*** *UserDBWrapper*
Return Type*: None*
Parameters: *None*
Return Values: *None*
Pre-Condition: *None (it is the constructor)*
Post-Condition: *The new UserDBWrapper object has been properly initialized*
with the given data. This means that the user name, full name, email, employee id, citizenship, visa status, and paid by university flag have been set with the provided values.
Attributes read/used: *All attributes*
Methods Called: *None*
Processing Logic: *Sets internal state according to the constructor parameters to properly initialize the object.*
Test Case 1: *Call the constructor on the object with valid parameters and ensure that a valid object is returned.*

2 ***Method:*** *getUserInfo*
Return Type*: List<String>*

Parameters:

- *String : userName*

Return Values: *userInfo*
Pre-Condition: *The UserDBWrapper object has been constructed/initialized.*
Post-Condition: *The userName for this UserDBWrapper has been returned. No state has been modified.*
Attributes read/used: userName
Methods Called: *None*
Processing Logic: *Returns the value of the userInfo attribute.*
Test Case 1: *Construct a UserDBWrapper. Pass an invalid username and check that a KeyNotFoundException is thrown.*

4.2.3.2 Class: UserGrantDBWrapper

- ***Purpose:*** *This class contains information contains information about the current user's grants.*
- ***Constraints:*** *User must have an account number.*
- ***Persistent****: Yes. This is persistent for the uptime of the TRAP system.*

4.2.3.1.1 Attribute Descriptions

1 ***Attribute:*** *userInfo*
Type: *Map<String, List<String>>*
Description: *This object will eventually contain information about the user's grants.*
Constraints: *Must not be null.*

4.2.3.1.2 Method Descriptions

1 ***Method:*** *UserGrantDBWrapper*
Return Type: *None*
Parameters: *None*
Return Values: *None*
Pre-Condition: *None (it is the constructor)*
Post-Condition: *The new UserGrantDBWrapper object has been properly initialized with the given data. This means that the account number, grant admin, and authorized payees have been set with the provided values*
Attributes read/used: *userInfo*
Methods Called: *None*
Processing Logic: *Sets internal state according to the constructor parameters to properly initialize the object.*
Test Case 1*: Call the constructor on the object with valid parameters and ensure that a valid object is returned.*

2 ***Method:*** *getUserGrantInfo*
Return Type: *List<String>*
Parameters:
- *String : accountName*

Return Values*: userInfo*
Pre-Condition: *The UserGrantInfo object has been constructed/initialized.*
Post-Condition: *The userInfo for this UserGrantInfo has been returned. No state has been modified.*
Attributes read/used: *userInfo*
Methods Called: *None*
Processing Logic: *Returns the value of the userInfo attribute.*
Test Case 1: *Construct a UserGrantDBWrapper. Pass an invalid account name and check that a KeyNotFoundException is thrown.*

4.2.3.3 Class: PerDiemDBWrapper

- ***Purpose:*** *Provides a way to access the PerDiemDBWrapper, which contains the meal and incidental per diem data. Initially, the methods in this class are pass-through methods.*
- ***Constraints:*** *City, country, or state need to be in the database.*
- ***Persistent:*** *Yes. This is persistent for the uptime of the TRAP system.*

4.2.3.1.1 Attribute Descriptions

1 ***Attribute:*** *perDiemInfo*
Type: *Map<List<String>, List<Double>>*
Description: *This object will eventually contain information about the domestic or international per diem rates.*
Constraints: *Must not be null.*

4.2.3.1.2 Method Descriptions

1 ***Method:*** *PerDiemDBWrapper*
Return Type*: None*
Parameters: *None*
Return Values: *None*
Pre-Condition: *None (it is the construtor)*
Post-Condition: *The new PerDiemDBWrapper object has been properly initialized with the given data. This means that the breakfast rate, lunch rate, dinner rate, incidental ceiling, and lodging ceiling have been set with the provided values.*
Attributes read/used: *All attributes.*

Methods Called: *None*
Processing Logic: *Sets internal state according to the constructor parameters to properly initialize the object.*
Test Case 1: *Call the constructor on the object with valid parameters and ensure that a valid object is returned.*

2 ***Method:*** *getDomesitcPerDiem*
Return Type: *List<Double>*
Parameters:
- *String : city*
- *String : state*

Return Values: *rateInfo*
Pre-Condition: *PerDiemDBWrapper has been constructed and initialized.*
Post-Condition: *If the city and state are valid, the per diem amount is returned.*
Attributes read/used: *perDiemInfo*
Methods Called: *None*
Processing Logic: *Check that the city and state are valid. If they are, return the per diem amount, and if not, return a KeyNotFoundException.*
Test Case 1: *Construct a PerDiemDBWrapper object. Pass a valid city and state and ensure that the per diem info is returned.*

3 ***Method:*** *getDomesitcPerDiem*
Return Type: *List<Double>*
Parameters:
- *String : state*

Return Values: *rateInfo*
Pre-Condition: *PerDiemDBWrapper has been constructed and initialized.*
Post-Condition: *If the state is valid, the per diem amount is returned.*
Attributes read/used: *perDiemInfo*
Methods Called: *None*
Processing Logic: *Check that the state is valid. If it is, return the per diem amount, and if not, return a KeyNotFoundException.*
Test Case 1: *Construct a PerDiemDBWrapper object. Pass a valid state and ensure that the per diem info is returned.*

4 ***Method:*** *getInternationalPerDiem*
Return Type: *List<Double>*
Parameters:
- *String : city*
- *String : country*

Return Values: *rateInfo*
Pre-Condition: *PerDiemDBWrapper has been constructed and initialized.*
Post-Condition: *If the city and country are valid, the per diem amount is returned.*
Attributes read/used: *perDiemInfo*
Methods Called: *None*
Processing Logic: *Check that the city and country are valid. If they are, return the per diem amount, and if not, return a KeyNotFoundException.*
Test Case 1: *Construct a PerDiemDBWrapper object. Pass a valid city and country and ensure that the per diem info is returned.*

5 ***Method:*** *getInternationalPerDiem*
Return Type: *List<Double>*
Parameters:
- *String : country*

Return Values: *rateInfo*
Pre-Condition: *PerDiemDBWrapper has been constructed and initialized.*
Post-Condition: *If the country is valid, the per diem amount is returned.*
Attributes read/used: *perDiemInfo*
Methods Called: *None*
Processing Logic: *Check that the state is valid. If it is, return the per diem amount, and if not, return a KeyNotFoundException.*
Test Case 1: *Construct a PerDiemDBWrapper object. Pass a valid country and ensure that the per diem info is returned.*

4.2.3.4 Class: GrantDBWrapper

- ***Purpose:*** *Provides a way to access the GrantDBWrapper, which contains information about all the grants. Initially, the methods in this class are pass-through methods.*
- ***Constraints:*** *The user needs to have an account name.*
- ***Persistent:*** *Yes. This is persistent for the uptime of the TRAP system.*

4.2.3.4.1 Attribute Descriptions

1 ***Attribute:*** *grantInfo*
Type: *Map<String, List<Object>>*
Description: *This attribute will eventually contain the information about the user's grant.*
Constraints: *Must not be null.*

4.2.3.4.2 Method Descriptions

1 ***Method:*** *GrantDBWrapper*
Return Type: *None*
Parameters: *None*
Return Values: *None*
Pre-Condition: *None (it is the construtor)*
Post-Condition: *The new GrantDBWrapper object has been properly initialized with the given data. This means that the account number, account type, funding organization, organization type, and account balance have been set with the provided values.*
Attributes read/used: *All attributes.*
Methods Called: *None*
Processing Logic: *Sets internal state according to the constructor parameters to properly initialize the object.*
Test Case 1: *Call the constructor on the object with valid parameters and ensure that a valid object is returned.*

2 ***Method:*** *getGrantInfo*
Return Type: *List<Object>*
Parameters:
- *String : accountName*

Return Values: *grantInfo*
Pre-Condition: *GrantDBWrapper has been constructed and initialized.*
Post-Condition: *If the account name is valid, it will return the grant info for the user.*
Attributes read/used: *grantInfo*
Methods Called: *None*
Processing Logic: *Try to get the grant info for the current user. If nothing is returned for the account number that is given, throw a KeyNotFoundException. Otherwise return the grant info.*
Test Case 1: *Construct a GrantDBWrapper object. Pass a valid account number and ensure that the grant info is returned to the user.*

3 ***Method:*** *updateAccountBalance*
Return Type: *None*
Parameters:
- *String : accountName*
- *Double : newBalance*

Return Values: *None*
Pre-Condition: *GrantDBWrapper has been constructed and initialized.*
Post-Condition: I*f the account name is valid, it will return the updated grant info for the user.*
Attributes read/used: *grantInfo*
Methods Called: *None*

Processing Logic: *Test Case 1: Construct a GrantDBWrapper object. Pass a valid account number and ensure that the grant info is updated.*

4.2.3.5 Class: CurrencyDBWrapper

- *Purpose: Provides a way to access the CurrencyDB, which contains information about currency conversion. Initially, the methods in this class are pass-through methods.*
- *Constraints: The dates and currencies must be in a specific format as specified in requirements 1.b, 1.c, and 1.l*
- *Persistent: Yes. This is persistent for the uptime of the TRAP system.*

4.2.3.4.1 Attribute Descriptions

1 ***Attribute:*** *currencyInfo*
Type: *Map<List<String>, Double>*
Description: *This will eventually contain the date and currency pair that will be used to convert the currency.*
Constraints: *The dates and currencies must be in a specific format as specified in requirements 1.b, 1.c, and 1.l*

4.2.3.4.2 Method Descriptions

1 ***Method:*** *CurrencyDBWrapper*
Return Type: *None*
Parameters: *None*
Return Values: *None*
Pre-Condition: *None (it is the constructor)*
Post-Condition: *The new CurrencyDBWrapper object has been properly initialized with the given data. This means that the currency and date have been set with the provided values.*
Attributes read/used: *currencyInfo*
Methods Called: *None*
Processing Logic: *The dates and currencies must be in a specific format as specified in requirements 1.b, 1.c, and 1.l*
Test Case 1: *Call the constructor on the object with valid parameters and ensure that a valid object is returned.*

2 ***Method:*** *getConversion*
Return Type: *Double*
Parameters:
- *String : currency*
- *String : date*

Return Values: *value*
Pre-Condition: *CurrencyDBWrapper has been constructed and initialized.*

Post-Condition: *If the date and currency are valid, it will return the currency conversion for the user.*
Attributes read/used: *currencyInfo*
Methods Called: *None*
Processing Logic: *Try to get the currency conversion based on the currency amount and date. If the date or currency is not formatted correctly or invalid, a KeyNotFoundException will be thrown. Otherwise the currency conversion will be returned.*

Test Case 1: *Construct a CurrencyDBWrapper object. Pass a valid date and currency, and manually verify that the currency conversion is correct.*

4.2.3.6 RateFieldEnum

- ***Purpose:*** *Contains necessary information on the per diem rates and ceilings.*
  - ***Enum Descriptions***
    - *BREAKFAST_RATE : Breakfast rate in USD*
    - *LUNCH_RATE : Lunch rate in USD*
    - *DINNER_RATE : Dinner rate in USD*
    - *INCIDENTAL_CEILING : Incidental ceiling in USD*
    - *LODGING_CEILING : Lodging ceiling in USD*

4.2.3.7 GrantFieldEnum

- ***Purpose:*** *Contains necessary information about all of the grants.*
  - ***Enum Descriptions***
    - *ACCOUNT_NUMBER : Account number*
    - *ACCOUNT_TYPE : Account (Sponsored vs Non-sponsored)*
    - *FUNDING_ORGANIZATION : Funding organization*
    - *ORGANIZATION_TYPE : Organization type (i.e. government, industry, noExport, ngo, foreign)*
    - *ACCOUNT_BALANCE : Account balance*

4.2.3.8 UserFieldEnum

- ***Purpose:*** *Contains necessary information about the user.*
  - ***Enum Descriptions***
    - *USER_NAME : X500 user name*
    - *FULL_NAME : Full name of user*
    - *EMAIL : Email address of user*
    - *EMPLOYEE_ID : ID number*
    - *CITIZENSHIP : Citizenship of the user*
    - *VISA_STATUS : US visa status if not a US citizenship*

4.2.3.9 UserGrantFieldEnum

- ***Purpose:*** *Contains the associations between all grants and the current user.*
  - ***Enum Descriptions***
    - *ACCOUNT_NUMBER : Grant account number*
    - *GRANT_ADMIN : Users (X500 numbers) who can be reimbursed under the grant, without permission, and approves other's requests*
    - *AUTHORIZED_PAYEES : Lists of users (X500 numbers) who can be reimbursed under a grant, with permission from an account admin*

4.2.3.10 CurrencyFieldEnum

- ***Purpose:*** *Contains the currency conversion information.*
  - ***Enum Descriptions***
    - *CURRENCY : Currency to be converted*
    - *DATE : Date of conversion*

4.2.4 Expenses

4.2.2.1 TRAPImpl

- ***Purpose:*** *The class that implements the TravelFormProcessorIntf for communication externally.*
- ***Constraints:*** *Only one instance of this class.*
- ***Persistent:*** *Yes. This is persistent for the uptime of the TRAP system.*

4.2.2.1.1 Attribute Descriptions

1. ***Attribute:*** *currentUser*
   Type: *String*
   Description: *Stores the x500 of a current user*
   Constraints: *Must be set before any form operations can take place. This includes form loading, saving or submitting*

4.2.2.1.2 Method Descriptions

1. ***Method:*** clearSavedForms
   Return Type: *None*
   Parameters: *None*
   Return Values: *None*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set.*
   Post-Condition: *All form data and metadata for the current user in TRAPImpl have been removed.*

Attributes read/used: *currentUser, formStorage*
Methods Called: *formStorage.clearAllForms*
Processing Logic: *Call the formStorage attribute with the currentUser to clear all the user's saved forms and metadata.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling saveFormData with some data and a description. Verify that this form is saved by calling getSavedForms and checking that the form is present. Call clearSavedForms and check the output of getSavedForms again, it should now be empty.*

4.2.4.2 *Class:* TransportationExpense

- ***Purpose:*** *Holds information about a given transportation expense incurred by the User. Also provides access to this expense's information for auditing.*
- ***Constraints:*** *None*
- ***Persistent:*** *No, only created if the user had lodging expenses (create one for each lodging expense) on their trip.*

4.2.4.2.1 Attribute Descriptions

Attribute: TransportationDate
Type: *Date*
Description: *Stores the* TransportationDate *that the transportation expense occurred in.*
Constraints: *Must not be empty.*

Attribute: TransportationCarrier
Type: *String*
Description: *Stores the carrier that the user used during this expense.*
Constraints: *Must not be empty.*

Attribute: TransportationMilesTraveled
Type: *Integer*
Description: *Stores the distance that the user traveled during this expense*
Constraints: *Must not be empty, must be in miles.*

Attribute: TransportationRental
Type: String
Description: *Stores information about a rental vehicle*
Constraints: *must be filled in if TransportationType is RENTAL_*, otherwise, this must not be filled in*

Attribute: *TransportationAmount*
Type: *Float*

Description: *Stores the amount of money this expense cost*
Constraints: *Must not be empty.*

Attribute: *TransportationCurrency*
Type: *String*
Description: *Stores the type of currency that the TransportationAmount is.*
Constraints: *Must not be empty.*

Attribute: *TransportationType*
Type: *TransportationType*
Description: *Stores the type of transportation that was taken during this expense*
Constraints: *Must not be empty.*

4.2.4.2.2 Method Descriptions

Method: *getDate()*
Return Type: *TRAPDate*
Parameters: *None*
Return Values: *The Date of the current object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the date has been returned.*
Attributes read/used: *ExpenseDate*
Methods Called: *None*
Processing Logic: *Provides access to the Date*
Test Case 1: *Create a new object of this type, set the Date and try to get it.*

Method: getCarrier*()*
Return Type: *String*
Parameters: *None*
Return Values: *the carrier that was used during this objects expense*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* TransportationCarrier *has been returned.*
Attributes read/used: TransportationCarrier
Methods Called: *None*
Processing Logic: *Provides access to the* TransportationCarrier of this expense object
Test Case 1: *Create a new object of this type, set the* TransportationCarrier*, try to get it.*

Method: getMilesTraveled*()*
Return Type: *Integer*

Parameters: *None*
Return Values: *the* miles *that were traveled during this objects expense*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* TransportationMilesTraveled *has been returned.*
Attributes read/used: TransportationMilesTraveled
Methods Called: *None*
Processing Logic: *Provides access to the* TransportationMilesTraveled of this expense object
Test Case 1: *Create a new object of this type, set the* TransportationMilesTraveled*, try to get it.*

Method: getRental*()*
Return Type: *String*
Parameters: *None*
Return Values: *the* rental information for this object
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* TransportationRental *has been returned.*
Attributes read/used: TransportationRental
Methods Called: *None*
Processing Logic: *Provides access to the* TransportationRental of this expense object
Test Case 1: *Create a new object of this type, set the rental type to RENTAL_CAR, set the* TransportationRental info*, try to get it.*

Method: getAmount*()*
Return Type: *Float*
Parameters: *None*
Return Values: *the* amount of money spent during this expense
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* TransportationAmount *has been returned.*
Attributes read/used: TransportationAmount
Methods Called: *None*
Processing Logic: *Provides access to the* TransportationAmount of this expense object
Test Case 1: *Create a new object of this type, set the* TransportationAmount*, try to get it.*

Method: *getAmount()*
Return Type: *Float*
Parameters: *None*
Return Values: *The ExpenseAmount of the this object*

Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseAmount has been returned.*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseAmount in this object*
Test Case 1: *Create a new OtherExpense object, set the ExpenseAmount, try to get it.*

Method: getCurrency*()*
Return Type: *String*
Parameters: *None*
Return Values: *The* ExpenseCurrency *of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* ExpenseCurrency *has been returned.*
Attributes read/used: ExpenseCurrency
Methods Called: *None*
Processing Logic: *Provides access to the* ExpenseCurrency
Test Case 1: *Create a new OtherExpense object, set the* ExpenseCurrency*, try to get it.*

Method: getType*()*
Return Type: *TransportationType*
Parameters: *None*
Return Values: *The* transportation type *of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the* TransportationType *has been returned.*
Attributes read/used: TransportationType
Methods Called: *None*
Processing Logic: *Provides access to the* transportation type of this expense object
Test Case 1: *Create a new object of this type, set the TransportationType, try to get it.*

Method: *setDate(date)*
Return Type: *Void*
Parameters: *date of type Date*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseDate has been set to the new date*
Attributes read/used: *ExpenseDate*
Methods Called: *None*

Processing Logic: *Provides a way to set the TransportationDate*
Test Case 1: *Create a new object of this type, set the TransportationDate and try to get it.*

Method: *setCarrier(carrier)*
Return Type: *Void*
Parameters: *carrier of type string*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationCarrier has been set to the new carrier*
Attributes read/used: *TransportationCarrier*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationCarrier*
Test Case 1: *Create a new object of this type, set the TransportationCarrier and try to get it.*

Method: *setMilesTraveled(miles)*
Return Type: *Void*
Parameters: *miles of type Integer*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationMilesTraveled has been set to the new miles*
Attributes read/used: *TransportationMilesTraveled*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationMilesTraveled*
Test Case 1: *Create a new object of this type, set the TransportationMilesTraveled and try to get it.*

Method: *setRental(info)*
Return Type: *Void*
Parameters: *info of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationRental has been set to the new info*
Attributes read/used: *TransportationRental*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationRental*
Test Case 1: *Create a new object of this type, set the TransportationRental and try to get it. This should not work unless the Transportation type is RENTAL_**

Method: *setAmount(amount)*
Return Type: *Void*
Parameters: *amount of type Float*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationAmount has been set to the new amount*
Attributes read/used: *TransportationAmount*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationAmount*
Test Case 1: *Create a new object of this type, set the TransportationAmount and try to get it.*

Method: *setCurrency(currency)*
Return Type: *Void*
Parameters: *currency of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationCurrency has been set to the new currency*
Attributes read/used: *TransportationCurrency*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationCurrency*
Test Case 1: *Create a new object of this type, set the TransportationCurrency and try to get it.*

Method: *setType(type)*
Return Type: *Void*
Parameters: *type of type TransportationType*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The TransportationType has been set to the new type*
Attributes read/used: *TransportationType*
Methods Called: *None*
Processing Logic: *Provides a way to set the TransportationType*
Test Case 1: *Create a new object of this type, set the TransportationType and try to get it.*

4.2.4.3 Class: LodgingExpenses

- ***Purpose:*** *Holds information about a given lodging expense incurred by the User. Also provides access to this expense's information for auditing.*
- ***Constraints:*** *None*

○ ***Persistent:*** *No, only created if the user had lodging expenses (create one for each lodging expense) on their trip.*

4.2.4.3.1 Attribute Descriptions

Attribute: *city*
Type: *String*
Description: *Stores the City that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *state*
Type: *String*
Description: *Stores the State that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *country*
Type: *String*
Description: *Stores the Country that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *ExpenseAmount*
Type: *Float*
Description: *Stores the total amount that was spent on this expense*
Constraints: *Must not be empty. Should be recorded as a number relative to the ExpenseCurrency.*

Attribute: *ExpenseCurrency*
Type: *String*
Description: *Stores the currency that the ExpenseAmount is in.*
Constraints: *Must not be empty.*

4.2.4.3.2 Method Descriptions

Method: *getCity()*
Return Type: *String*
Parameters: *None*
Return Values: *The City of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*

Test Case 1:

Method: *getCity()*
Return Type: *String*
Parameters: *None*
Return Values: *The City of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the city, try to get it.*

Method: *getState()*
Return Type: *String*
Parameters: *None*
Return Values: *The State of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the state, try to get it.*

Method: *getCountry()*
Return Type: *String*
Parameters: *None*
Return Values: *The Country of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *country*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the country and try to get it.*

Method: *getAmount()*
Return Type: *Float*
Parameters: *None*
Return Values: *The ExpenseAmount of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseAmount has been returned.*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseAmount in this object*
Test Case 1: *Create a new OtherExpense object, set the ExpenseAmount, try to get it.*

Method: *getCurrency()*
Return Type: *String*
Parameters: *None*
Return Values: *The ExpenseCurrency of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseCurrency has been returned.*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseCurrency*
Test Case 1: *Create a new OtherExpense object, set the ExpenseCurrency, try to get it.*

Method: *setCity(city)*
Return Type: *None*
Parameters: *city – the name of the city which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of the object is updated to the new city*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides a way to set the city of the object*
Test Case 1: *create a new object of this type, set the city, try to get it.*

Method: *setState(state)*
Return Type: *None*
Parameters: *state – the name of the state city which is of type String*
Return Values: *None*

Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of this object is updated to the new state*
Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides a way to set the state of this object*
Test Case 1: *create a new object of this type, set the state, try to get it.*

Method: *setCountry(country)*
Return Type: *None*
Parameters: *country – the name of the state country which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The country of the object is updated to the new country*
Attributes read/used: *country*
Methods Called: *None*
Processing Logic: *Provides a way to set the country of the object*
Test Case 1: *Create a new object of this type, set the country and try to get it.*

Method: *setAmount(amount)*
Return Type: *Void*
Parameters: *amount : Float that has the expense amount*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseAmount is set to the new amount*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseAmount for objects of this type*
Test Case 1: *Create a new object of this type, set the ExpenseAmount, try to get it.*

Method: *setCurrency(currency)*
Return Type: *Void*
Parameters: *currency : String that has the expense currency*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseCurrency is set to the new currency*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*

Processing Logic: *Provides a way to set the ExpenseCurrency for objects of this type*
Test Case 1: *Create a new object of this type, set the ExpenseCurrency, try to get it.*

4.2.4.4 Class: IncidentalExpense

- ***Purpose:*** *Holds information about a given incidental expense incurred by the User. Also provides access to this expense's information for auditing.*
- ***Constraints:*** *None*
- ***Persistent:*** *No, only created if the user had incidental expenses (create one for each incidental expense) on their trip.*

4.2.4.4.1 Attribute Descriptions

Attribute: *ExpenseDate*
Type: *Date*
Description: *Stores the date of the expense.*
Constraints: *Must be a real date.*

Attribute: *city*
Type: *String*
Description: *Stores the City that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *state*
Type: *String*
Description: *Stores the State that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *ExpenseAmount*
Type: *Float*
Description: *Stores the total amount that was spent on this expense*
Constraints: *Must not be empty. Should be recorded as a number relative to the ExpenseCurrency.*

Attribute: *ExpenseCurrency*
Type: *String*
Description: *Stores the currency that the ExpenseAmount is in.*
Constraints: *Must not be empty.*

4.2.4.4.2 Method Descriptions

Method: *getDate()*
Return Type: *TRAPDate*
Parameters: *None*
Return Values: *The Date of the current object*

Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the date has been returned.*
Attributes read/used: *ExpenseDate*
Methods Called: *None*
Processing Logic: *Provides access to the Date*
Test Case 1: *Create a new object of this type, set the Date and try to get it.*

Method: *getCity()*
Return Type: *String*
Parameters: *None*
Return Values: *The City of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the City and try to get it.*

Method: *getState()*
Return Type: *String*
Parameters: *None*
Return Values: *The State of this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the state and try to get it.*

Method: *getAmount()*
Return Type: *Float*
Parameters: *None*
Return Values: *The ExpenseAmount of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseAmount has been returned.*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseAmount in this object*

Test Case 1: *Create a new OtherExpense object, set the ExpenseAmount, try to get it.*
Method: *getCurrency()*

Return Type: *String*
Parameters: *None*
Return Values: *The ExpenseCurrency of the this object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseCurrency has been returned.*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseCurrency*
Test Case 1: *Create a new OtherExpense object, set the ExpenseCurrency, try to get it.*

Method: *SetDate(date)*
Return Type: *Void*
Parameters: *date : Date*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The Date of this object is set to the new date*
Attributes read/used: *Date*
Methods Called: *None*
Processing Logic: *Provides a way to set the Date for objects of this type*
Test Case 1: *Create a new object of this type, set the Date, try to get it.*

Method: *setCity(city)*
Return Type: *None*
Parameters: *city – the name of the city which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of the object is updated to the new city*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides a way to set the city of the object*
Test Case 1: *create a new object of this type, set the city, try to get it.*

Method: *setState(state)*
Return Type: *None*
Parameters: *state – the name of the state city which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of this object is updated to the new state*

Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides a way to set the state of this object*
Test Case 1: *create a new object of this type, set the state, try to get it.*

Method: *setAmount(amount)*
Return Type: *Void*
Parameters: *amount : Float that has the expense amount*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseAmount is set to the new amount*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseAmount for objects of this type*
Test Case 1: *Create a new object of this type, set the ExpenseAmount, try to get it.*

Method: *setCurrency(currency)*
Return Type: *Void*
Parameters: *currency : String that has the expense currency*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseCurrency is set to the new currency*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseCurrency for objects of this type*
Test Case 1: *Create a new object of this type, set the ExpenseCurrency, try to get it.*

4.2.4.5 Class: OtherExpense

- ***Purpose:*** *Holds information about a given other expense incurred by the User. Also provides access to said information for auditing.*
- ***Constraints:*** *None*
- ***Persistent:*** *No, only created if the user had other expenses (create one for each other expense) on their trip.*

4.2.4.5.1 Attribute Descriptions

Attribute: *ExpenseDate*
Type: *Date*
Description: *Stores the date of the expense.*
Constraints: *Must be a real date.*

Attribute: *ExpenseJustification*
Type: *String*
Description: *Stores the description and reasoning for the expense.*
Constraints: *Must not be empty.*

Attribute: *ExpenseAmount*
Type: *Float*
Description: *Stores the total amount that was spent on this other expense*
Constraints: *Must not be empty. Should be recorded as a number relative to the ExpenseCurrency.*

Attribute: *ExpenseCurrency*
Type: *String*
Description: *Stores the currency that the ExpenseAmount is in.*
Constraints: *Must not be empty.*

4.2.4.5.2 Method Descriptions

Method: *getDate()*
Return Type: *TRAPDate*
Parameters: *None*
Return Values: *The Date of the current OtherExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the date has been returned.*
Attributes read/used: *ExpenseDate*
Methods Called: *None*
Processing Logic: *Provides access to the Date*
Test Case 1: *Create a new OtherExpense object, set the Date and try to get it.*

Method: *getJustification()*
Return Type: *String*
Parameters: *None*
Return Values: *The ExpenseJustification of the current OtherExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseJustification has been returned.*
Attributes read/used: *ExpenseJustification*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseJustification*
Test Case 1: *Create a new OtherExpense object, set the ExpenseJustification, try to get it.*

Method: *getAmount()*
Return Type: *Float*
Parameters: *None*
Return Values: *The ExpenseAmount of the current OtherExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseAmount has been returned.*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseAmount*
Test Case 1: *Create a new OtherExpense object, set the ExpenseAmount, try to get it.*

Method: *getCurrency()*
Return Type: *String*
Parameters: *None*
Return Values: *The ExpenseCurrency of the current OtherExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the ExpenseCurrency has been returned.*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*
Processing Logic: *Provides access to the ExpenseCurrency*
Test Case 1: *Create a new OtherExpense object, set the ExpenseCurrency, try to get it.*

Method: *SetDate(date)*
Return Type: *Void*
Parameters: *date : Date*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The Date is set to the new date*
Attributes read/used: *Date*
Methods Called: *None*
Processing Logic: *Provides a way to set the Date for OtherExpense objects*
Test Case 1: *Create a new OtherExpense object, set the Date, try to get it.*

Method: *setJustification(justification)*
Return Type: *Void*
Parameters: *justification : String that has the expense justification*
Return Values: *None*

Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseJustification is set to the new justification*
Attributes read/used: *ExpenseJustification*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseJustification for OtherExpense objects*
Test Case 1: *Create a new OtherExpense object, set the ExpenseJustification, try to get it.*

Method: *setAmount(amount)*
Return Type: *Void*
Parameters: *amount : Float that has the expense amount*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseAmount is set to the new amount*
Attributes read/used: *ExpenseAmount*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseAmount for OtherExpense objects*
Test Case 1: *Create a new OtherExpense object, set the ExpenseAmount, try to get it.*

Method: *setCurrency(currency)*
Return Type: *Void*
Parameters: *currency : String that has the expense currency*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The ExpenseCurrency is set to the new date*
Attributes read/used: *ExpenseCurrency*
Methods Called: *None*
Processing Logic: *Provides a way to set the ExpenseCurrency for OtherExpense objects*
Test Case 1: *Create a new OtherExpense object, set the ExpenseCurrency, try to get it.*

4.2.4.6 Class: MealExpense

- ***Purpose:*** *Holds information about a given meal expense incurred by the User. Also provides access to said information for auditing.*
- ***Constraints:*** *None*
- ***Persistent:*** *No, only created if the user had meal expenses (create one for each meal expense) on their trip.*

4.2.4.6.1 Attribute Descriptions

Attribute: *type*

Type: *MealType*
Description: *Stores the type of meal type that the MealExpense was.*
Constraints: *Bust be one of the enumerated type options in MealTypeEnum*

Attribute: *city*
Type: *String*
Description: *Stores the City that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *state*
Type: *String*
Description: *Stores the State that the meal expense occurred in.*
Constraints: *Must not be empty.*

Attribute: *country*
Type: *String*
Description: *Stores the Country that the meal expense occurred in.*
Constraints: *Must not be empty.*

4.2.4.6.1 Method Descriptions

Method: *getMealType()*
Return Type: *MealType*
Parameters: *None*
Return Values: *The MealType of MealExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the MealType has been returned.*
Attributes read/used: *type*
Methods Called: *None*
Processing Logic: *Provides access to the MealExpense object type.*
Test Case 1: *Create a new object of this type, set the MealType and try to get it.*

Method: *getCity()*
Return Type: *String*
Parameters: *None*
Return Values: *The City of MealExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*

Test Case 1: *Create a new object of this type, set the city and try to get it.*

Method: *getState()*
Return Type: *String*
Parameters: *None*
Return Values: *The State of MealExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the state and try to get it.*

Method: *getCountry()*
Return Type: *String*
Parameters: *None*
Return Values: *The Country of MealExpense object*
Pre-Condition: *Class has been instantiated*
Post-Condition: *Nothing has changed and the requested value has been returned.*
Attributes read/used: *country*
Methods Called: *None*
Processing Logic: *Provides access to the requested value of this object.*
Test Case 1: *Create a new object of this type, set the country and try to get it.*

Method: *setType()*
Return Type: *None*
Parameters: *type – which determines which MealTypeEnum type this object is*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The type of the object is updated to the MealEnumType*
Attributes read/used: *type*
Methods Called: *None*
Processing Logic: *Provides a way to set the type of the object*
Test Case 1: *Create a new object of this type, set the type and try to get it.*

Method: *setCity(city)*
Return Type: *None*
Parameters: *city – the name of the city which is of type String*

Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of the object is updated to the new city*
Attributes read/used: *city*
Methods Called: *None*
Processing Logic: *Provides a way to set the city of the object*
Test Case 1: *Create a new object of this type, set the city and try to get it.*

Method: *setState(state)*
Return Type: *None*
Parameters: *state – the name of the state city which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The city of the object is updated to the new state*
Attributes read/used: *state*
Methods Called: *None*
Processing Logic: *Provides a way to set the state of the object*
Test Case 1: *Create a new object of this type, set the state and try to get it.*

Method: *setCountry(country)*
Return Type: *None*
Parameters: *country – the name of the state country which is of type String*
Return Values: *None*
Pre-Condition: *Class has been instantiated*
Post-Condition: *The country of the object is updated to the new country*
Attributes read/used: *country*
Methods Called: *None*
Processing Logic: *Provides a way to set the country of the object*
Test Case 1: *Create a new object of this type, set the country and try to get it.*

4.2.4.7 Class: TransportationTypeEnum

- ***Purpose:*** *Identifies the type of a given TransportationExpense object.*
- ***Constraints:*** *Shall be used with instantiated TransportationExpense objects.*
- ***Persistent:*** *No, only used when TransportationExpense objects are instantiated.*

4.2.4.8 Class: MealTypeEnum

- ***Purpose:*** *Identifies the type of a given MealExpense object.*
- ***Constraints:*** *Shall be used with instantiated MealExpense objects.*
- ***Persistent:*** *No, only used when MealExpense objects are instantiated.*

4.2.5 User Form Management

4.2.5.1 *Class:* FormStatusEnum

- ***Purpose:*** *Identifies the current status of a form.*
  - ***Enum Descriptions***
    - *DRAFT : The form has not passed auditing*
    - *SUBMITTED : The form has passed auditing*
- ***Constraints:*** *Only one instance of this during form auditing.*
- ***Persistent:*** *Yes, when the TRAP system is started up.*

4.2.5.2 *Class:* AllUserForms

- ***Purpose:*** *Container holding a list of users and their forms*
- ***Constraints:*** *Only one instance of this.*
- ***Persistent:*** *Yes, when the TRAP system is started up.*

4.2.5.2.1 Attribute Descriptions

1. ***Attribute:*** *usersForms*
   Type: *map<string, SavedForms>*
   Description: *Stores a mapping between the users and their SavedForms*
   Constraints: *Only one instance, size varies as the amount of users.*

4.2.5.2.1 Method Descriptions

1. ***Method:*** getUserSavedForms
   Return Type: *SavedForms*
   Parameters: *String : user*
   Return Values: *SavedForms*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
   Post-Condition: *The SavedForms of a user have either been returned (if any).*
   Attributes read/used: *usersForms*
   Methods Called: *Class names will be used in place of local variables.*
   AllUsersForms.getUserSavedForms
   Processing Logic: *Call getuserSavedForms to return all the saved forms of a particular user.*
   Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling saveFormData with some data and a description. Verify that this form is saved by calling getSavedForms and checking that the form is present.*

2. ***Method:*** insertUser
   Return Type: *void*
   Parameters: *String : user*

Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set.*
Post-Condition: *A user has been inserted into the list of user saved forms.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUsersForms.insertUser(String : user)
Processing Logic: *Call insertUser to insert a user into the list of user saved forms.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Insert the user by calling insertUser. Call containsKey (from the Java standard library) to check if the key (user) is present.*

3. ***Method:*** saveFormData
Return Type: *void*
Parameters: *String : user, map<string,string> : formData, Integer : id*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *Form data has been saved.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUsersForms.saveFormData(String : user, map<string,string> : formData, Integer : id)
SavedForms.saveForm(map<string,string> : formData, Integer : id)
FormContainer.saveForm(map<string,string> : formData)

Processing Logic: *Call AllUserForms.saveFormData to save a form. Subsequent method calls will be made to insert the form data into the correct user's forms. If a form exists with the form id, it shall be overwritten with new data.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with the appropriate parameters. Call getSavedFormData to see if the form is returned (indicates the save was correct).*

4. ***Method:*** saveFormData
Return Type: *void*
Parameters: *String : user, map<string,string> : formData, String : description)*
Return Values: *void*

Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *Form data has been saved along with a form description.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUsersForms.saveFormData(String : user, map<string,string> : formData, String : description)
SavedForms.saveForm(map<string,string> : formData, Integer : id)
FormContainer.saveForm(map<string,string> : formData)
SavedForms.getNewFormId
Processing Logic: *Call AllUserForms.saveFormData to save a form. Subsequent method calls will be made to insert the form data into the correct user's forms.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData with the appropriate parameters. Call getSavedForms to see if the form is returned (indicates the save was correct).*

5. ***Method:*** getCompletedForm
Return Type: *map<string, string>*
Parameters: *String : user, Integer : id*
Return Values: *A completed form specified by the id*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *A completed form has been returned (or null if it does not exist)*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUserForms.getCompletedForm(String : user, Integer : id)
SavedForms.getFormContainer(Integer : id)
FormContainer.getStatus() (must be SUBMITTED)
FormContainer.getForm()
Processing Logic: *Call AllUserForms.getCompletedForm to retrieve a form that has passed auditing. Subsequent method calls will be made to extract the form.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Submit the form. Call saveCompletedForm on the output on a successful audit. Call getCompletedForm and see if it is returned.*

6. ***Method:*** getSavedFormData
Return Type: *map<string, string>*
Parameters: *String : user, Integer : id*

Return Values: *A form*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *A form has been returned (or null if it does not exist)*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUserForms.getSavedFormData(String : user, Integer : id)
SavedForms.getFormContainer(Integer : id)
FormContainer.getForm()
Processing Logic: *Call AllUserForms.getSavedFormData to retrieve a form. Subsequent method calls will be made to extract the form.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call saveFormData. Call getSavedFormData to see if the form is returned.*

7. ***Method:*** getSavedForms
Return Type: *map<Integer, TravelFormMetaData>*
Parameters: *String : user*
Return Values: *A map of integers (form ids) and TravelFormMetaData, which contains a form's status and description.*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has some saved forms.*
Post-Condition: *A mapping is returned of all a user's forms and appropriate TravelFormMetaData.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUserForms.getSavedForms(String : user)
SavedForms.getFormContainer(Integer : id)
FormContainer.getStatus()
FormContainer.getDescription()
Processing Logic: *Call AllUserForms.getSavedForms to retrieve information about a user's forms. Subsequent method calls will be made to extract the form.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save some forms using SaveFormData. Call getSavedForms and check that all forms that were saved are present.*

8. ***Method:*** saveCompletedForm
Return Type: *void*
Parameters: *String : user, map<string,string> : data, Integer : id*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has submitted a form that passed auditing.*

Post-Condition: *The completed form data is saved.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUserForms.saveCompletedForm(String : user, map<string,string> : data, Integer : id)
SavedForms.saveForm(map<string,string> : data, FormStatusEnum : status) (note this status will be SUBMITTED)
FormContainer.setStatus(FormStatusEnum : status)
FormContainer.saveForm(map<string,string> : data)
Processing Logic: *Call AllUserForms.saveCompletedForm to save an audited form. Subsequent method calls will be made to save the form.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save a forms using SaveFormData. Submit the form. Call saveCompletedForm once the form has passed auditing. Call getComletedForm to see if it is present.*

9. ***Method:*** clearSavedForms
Return Type: *void*
Parameters: *None*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has (likely) saved forms.*
Post-Condition: *All of a user's forms have been cleared.*
Attributes read/used: *usersForms*
Methods Called: *Class names will be used in place of local variables.*
AllUserForms.clearSavedForms()
SavedForms.clearForms()
Processing Logic: *Call AllUserForms.clearSavedForms to remove all of a user's saved forms. Subsequent method calls will be made to remove the forms.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save some forms using SaveFormData. Call clearSavedForms. Try to retrieve a form using getSavedFormData or getSavedForms.*

4.2.5.3 *Class:* SavedForms

- ***Purpose:*** *Container holding a list of a user's forms*
- ***Constraints:*** *Only one instance of this per user.*
- ***Persistent:*** *Yes, created as users added.*

4.2.5.3.1 Attribute Descriptions

1. ***Attribute:*** *savedForms*
Type: *map<Integer, FormContainer>*
Description: *Stores a mapping between a user and their forms*

Constraints: *Size varies as the number of forms.*

4.2.5.3.1 Method Descriptions

1. ***Method:*** getFormContainer
   Return Type: *FormContainer*
   Parameters: *Integer : id*
   Return Values: *FormContainer*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has (likely) saved forms.*
   Post-Condition: *A FormContainer is returned.*
   Attributes read/used: *savedForms*
   Methods Called: *Class names will be used in place of local variables.*
   SavedForms.getFormContainer(Integer : id)
   Processing Logic: *Call getFormContainer to get the container holding the user's form and additional form data.*
   Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getFormContainer to see if the form container for that form is returned.*

2. ***Method:*** saveForm
   Return Type: *void*
   Parameters: *map<string,string> formData, Integer : id*
   Return Values: *void*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of saved use forms.*
   Post-Condition: *A form has been saved*
   Attributes read/used: *savedForms*
   Methods Called: *Class names will be used in place of local variables.*
   SavedForms.saveForm(map<string,string> : formData, Integer : id)
   FormContainer.saveForm(map<string,string>)
   Processing Logic: *Call saveForm to save a form*
   Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save a form by calling saveForm. Call getFormContainer to see if the form is returned.*

3. ***Method:*** saveForm
   Return Type: *void*
   Parameters: *map<string,string> formData, String : description*
   Return Values: *void*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of saved use forms.*
   Post-Condition: *A form has been saved with a description*

Attributes read/used: *savedForms*
Methods Called: *Class names will be used in place of local variables.*
SavedForms.saveForm(map<string,string> : formData, String : description)
SavedForms.getNewFormID()
FormContainer.saveForm(map<string,string>)
Processing Logic: *Call saveForm to save a form with a description.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save a form by calling saveForm. Call getFormContainer to see if the form is returned.*

4. ***Method:*** saveForm
Return Type: *void*
Parameters: *map<string,string> formData, FormStatusEnum : status*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of saved use forms.*
Post-Condition: *A form has been saved with a new status*
Attributes read/used: *savedForms*
Methods Called: *Class names will be used in place of local variables.*
SavedForms.saveForm(map<string,string> : formData, FormStatusEnum : status)
SavedForms.getFormContainer
FormContainer.setStatus
FormContainer.saveForm(map<string,string>)
Processing Logic: *Call saveForm to save a form with a new status.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save a form by calling saveForm. Call getFormContainer to see if the form is returned.*

5. ***Method:*** clearForms
Return Type: *void*
Parameters: *None*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of saved use forms.*
Post-Condition: *A form has been saved with a new status*
Attributes read/used: *savedForms*
Methods Called: *Class names will be used in place of local variables.*
SavedForms.clearForms()
Processing Logic: *Call clearForms to remove all of a user's saved forms.*

Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Save a form(s) by calling saveForm. Call clearForms. Call getFormContainer to see if a form is returned.*

6. ***Method:*** getNewFormId
   Return Type: *Integer*
   Parameters: *None*
   Return Values: *Integer (a new form id)*
   Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of saved use forms. A user is saving a form.*
   Post-Condition: *A form has been saved with a new form id.*
   Attributes read/used: *savedForms*
   Methods Called: *Class names will be used in place of local variables.*
   SavedForms.getNewFormId
   Processing Logic: *Call getNewFormId to create a new entry in savedForms to save forms.*
   Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Call getNewFormId. See if a new key is created in savedForms.*

4.2.5.4 *Class:* FormContainer

- ***Purpose:*** *Container holding the form and related information*
- ***Constraints:*** *Only one instance of this per form.*
- ***Persistent:*** *Yes, created as users save forms.*

4.2.5.4.1 Attribute Descriptions

1. ***Attribute:*** *status*
   Type: *FormStatusEnum*
   Description: *Enum value for the status of a form*
   Constraints: *Must be set*

2. ***Attribute:*** *form*
   Type: *map<string,string>*
   Description: *Holds form data*
   Constraints: *Must be set*

3. ***Attribute:*** *formDescription*
   Type: *String*
   Description: *Description of a form*
   Constraints: *May be the empty string*

4.2.5.4.1 Method Descriptions

1. ***Method:*** getDescription
   Return Type: *String*

Parameters: *None*
Return Values: *String (the form description)*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has a saved form.*
Post-Condition: *The form description is returned.*
Attributes read/used: *formDescription*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.getDescription
Processing Logic: *Call getDescription to get the formDescription attribute.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getDescription to see if a form description is returned.*

2. ***Method:*** setDescription
Return Type: *void*
Parameters: *String : description*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has a saved form.*
Post-Condition: *The form description is set.*
Attributes read/used: *formDescription*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.setDescription
Processing Logic: *Call setDescription to set the formDescription attribute.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call setDescription. Call getDescription to see if a form description is returned.*

3. ***Method:*** getForm
Return Type: *map<string,string>*
Parameters: *None*
Return Values: *map<string,string> (the user's form)*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has a saved form.*
Post-Condition: *The form is returned.*
Attributes read/used: *form*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.getForm
Processing Logic: *Call getForm to get the form attribute.*

Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getForm to see if a form is returned.*

4. ***Method:*** saveForm
Return Type: *void*
Parameters: *map<string,string> : formData*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *The form is saved.*
Attributes read/used: *form*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.saveForm
Processing Logic: *Call saveForm to set the form attribute.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getForm to see if a form is returned.*

5. ***Method:*** saveForm
Return Type: *void*
Parameters: *map<string,string> : formData, String : description*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms.*
Post-Condition: *The form is saved with a description.*
Attributes read/used: *form, formDescription*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.saveForm
Processing Logic: *Call saveForm to set the form and formDescription attributes.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getForm to see if a form is returned and call getDescription to see if a description is returned.*

6. ***Method:*** getStatus
Return Type: *FormStatusEnum*
Parameters: *None*
Return Values: *Enum*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has saved a form.*

Post-Condition: *The form status is returned.*
Attributes read/used: *status*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.getStatus
Processing Logic: *Call getStatus to get the status attribute.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call getStatus to see if a form status is returned.*

7. ***Method:*** setStatus
Return Type: void
Parameters: *FormStatusEnum : status*
Return Values: *void*
Pre-Condition: *TRAPImpl has been constructed and initialized. A current user has been set. A user has been added to the list of user saved forms. A user has saved a form.*
Post-Condition: *The form status is set.*
Attributes read/used: *status*
Methods Called: *Class names will be used in place of local variables.*
FormContainer.setStatus
Processing Logic: *Call setStatus to set the status attribute.*
Test Case 1: *Initialize the TRAP system. Set a current user by calling setUser. Add a new form by calling AllUserForms.saveFormData. Call setStatus. Call getStatus to see if a form status is returned.*

4.2.5.5 *Class:* TravelFormMetaData

- ***Purpose:*** *Container a form status and a description.*
- ***Constraints:*** *Created on the fly, as often as AllUserForms.getSavedForms is called.*
- ***Persistent:*** *Lifetime length varies until garbage collected. TRAP and the Reimbursement Application do not depend on this.*

4.2.5.5.1 Attribute Descriptions

1. ***Attribute:*** *status*
Type: *FormStatusEnum*
Description: *Enum value for the status of a form. Public attribute, no getters/setters needed.*
Constraints: *Must be set*

2. ***Attribute:*** *description*
Type: *String*
Description: *Holds form description. Public attribute, no getters/setters needed.*

Constraints: *May be the empty string*

4.2.6 Exceptions

4.2.6.1 *Class:* TRAPException

- ***Purpose:*** *Abstract type representing an exception. Implements java.lang.Exception*
- ***Constraints:*** *Thrown when an exception is detected. Used to force the exception thrower to use a more specific exception. Catching allows a reference to a TRAPException is needed.*
- ***Persistent:*** *This class is abstract and cannot be instantiated but those classes that extend it, the actual rules, will persist as long as TRAP is running.*

4.2.7.1.1 Attribute Descriptions
None

4.2.7.1.2 Method Descriptions
None

4.2.6.2 *Class:* InputValidationException

- ***Purpose:*** *Throws exceptions when input has been entered incorrectly.*
- ***Constraints:*** *Thrown when an exception is detected. Exceptions for input validation are determined by the various instantiated InputValidationsRules. As the input is checked, the various instantiated rules will determine if an InputValidationException needs to be thrown.*
- ***Persistent:*** *Created when an error is detected.*

4.2.6.2.1 Attribute Descriptions
None

4.2.6.2.2 Method Descriptions
Inherits all methods from the java.lang.Exception class.

4.2.6.3 *Class:* BusinessLogicException

- ***Purpose:*** *Throws exceptions when business logic is violated.*
- ***Constraints:*** *Thrown when an exception is detected. Exceptions for business rules are determined by the various instantiated BusinessLogicRules. As the rules are checked, the various instantiated rules will determine if a BusinessLogicException needs to be thrown.*
- ***Persistent:*** *Created when an error is detected.*

4.2.6.3.1 Attribute Descriptions
None

4.2.6.3.2 Method Descriptions
Inherits all methods from the java.lang.Exception class.

4.2.6.4 *Class:* KeyNotFoundException

- ***Purpose:*** *Throws exceptions when a key from the input map is not present*
- ***Constraints:*** *Thrown when an exception is detected. This exception will be determined while checking the input map.Specifically, if one of the various instantiated InputValidationsRules finds a key to be missing from the input map.*
- ***Persistent:*** *Created when an error is detected.*

4.2.6.4.1 Attribute Descriptions
None

4.2.6.4.2 Method Descriptions
Inherits all methods from the java.lang.Exception class.

4.2.7 Rules

With the exception of the abstract rule types, it is assumed (for brevity) that each of these classes has a checkRule method according to the TRAPRule interface which is defined in section 3.2. For brevity, each rule will describe its goal in language similar to the requirement/s that it implements and some details on the state it shall set.

4.2.7.1 *Class:* InputValidationRule

- ***Purpose:*** *Abstract type representing all input validation rules.*
- ***Constraints:*** *InputValidation rules shall not add to the reimbursement total or add RestrictedLineItems.*
- ***Persistent:*** *This class is abstract and cannot be instantiated but those classes that extend it, the actual rules, will persist as long as TRAP is running.*

4.2.7.1.1 Attribute Descriptions
None

4.2.7.1.2 Method Descriptions
1. ***Method:*** *checkRule*
   Return Type: *None*
   Parameters:
   - *ReimbursementApp : app*

   Return Values: *None*
   Pre-Condition: *A ReimbursementApp has been constructed from some formData.*
   Post-Condition: *The rule has been checked and if any problems were found with the input an Exception will be thrown. If no problems are found the method shall return normally to continue processing.*
   Attributes read/used: *None*
   Methods Called: *None*
   Processing Logic: *Checks the logic for the rule. This is dependent on the particular rule that is extending this abstract class.*

4.2.7.2 Class: BusinessLogicRule

- ***Purpose:*** *Abstract type representing all business logic rules.*
- ***Constraints:*** *BusinessLogicRules are allowed to add to the reimbursement total and create RestrictedLineItems. BusinessLogicRules whose only purpose is to check a restriction on some item shall only add RestrictedLineItems, not add to the reimbursement total.*
- ***Persistent:*** *This class is abstract and cannot be instantiated but those classes that extend it, the actual rules, will persist as long as TRAP is running.*

4.2.7.2.1 Method Descriptions
1. ***Method:*** *checkRule*
   Return Type: *None*
   Parameters:
   - *ReimbursementApp : app*

   Return Values: *None*
   Pre-Condition: *A ReimbursementApp has been constructed from some formData.*
   Post-Condition: *The rule has been checked and if any fatal problems were found a TRAPException shall have been raised. Otherwise the rule succeeded and there shall be a state change in the reimbursement total and/or the set of RestrictedLineItems.*
   Attributes read/used: *None*
   Methods Called: *None*

Processing Logic: *Checks the logic for the rule. This is dependent on the particular rule that is extending this abstract class.*

4.2.7.3 *Class:* FinalizeRule

- ***Purpose:*** *This rule processes the ReimbursementApp after all other rules have been checked and passed. In particular it manages splitting up the reimbursement total of the application according to the grant percentages defined and any restricted line items.*
- ***Constraints:*** *The ReimbursementApp passed to this rule must have already completed all other rule checks.*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

Rule Logic: *The goal of this rule is to properly split funding across all provided grants and if this cannot be done an Exception shall be thrown. Given the percentages for grants it must first determine how much needs to be funded from each grant. For each RestrictedLineItem it must find adequate funding under one of the potential grants or an exception shall be thrown.*

The expense amount for all RestrictedLineItems is already counted in the reimbursement total so no changes must be made to it. It must be ensured that given the grant percentages provided, each RestrictedLineItem can be funded within the amount allocated (by the percentage) to one of its potential grants.

4.2.7.4 *Class:* DateValidator

- ***Purpose:*** *Checks that all date and datetime formats are valid and that appropriate dates and datetime ranges are valid.*
- ***Constraints:*** *A valid ReimbursementApp must have been created from some formData.*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

Rule Logic: *There are several things that this rule must check including the following:*

- *The arrival and departure datetime on the RApp must be a valid range. This means that the arrival datetime comes after the departure datetime.*
- *The dates for all expenses must be validated to fall within the arrival and departure datetimes.*

4.2.7.4.1 Method Descriptions

1. ***Method:*** [static] *convertToDatetime*
   Return Type: *Datetime*
   Parameters:
   - *String : rawDatetime*

Return Values: *Datetime – The Datetime object constructed from the raw datetime string. This is interpreted according to the standard TRAP datetime format defined in requirement 1.c.*
Pre-Condition: *None*
Post-Condition: *If the raw datetime string is in the proper format a Datetime object shall be returned, otherwise an exception will be thrown.*
Methods Called: *None*
Processing Logic: *Given a string of a raw datetime, this method shall parse it according to the TRAP datetime format defined in requirement #(). If the format is valid, a Datetime object shall be returned with values initialized according to the parsed datetime. Otherwise an exception shall be thrown.*
Test Case 1: *Call convertToDatetime with a valid raw datetime string and verify that you receive a corresponding Datetime object with the correctly set fields.*
Test Case 2: *Call convertToDatetime with an invalid raw datetime string and verify that you receive an exception.*

2. ***Method:*** [static] *convertToDate*
Return Type: *Date*
Parameters:
    - ***String :*** *rawDate*

Return Values: *Date – The Date object constructed from the raw date string. This is interpreted according to the standard TRAP date format defined in requirement 1.b.*
Pre-Condition: *None*
Post-Condition: *If the raw date string is in the proper format a Date object shall be returned, otherwise an exception will be thrown.*
Methods Called: *None*
Processing Logic: *Given a string of a raw date, this method shall parse it according to the TRAP date format defined in requirement 1.c. If the format is valid, a Date object shall be returned with values initialized according to the parsed date. Otherwise an exception shall be thrown.*
Test Case 1: *Call convertToDate with a valid raw date string and verify that you receive a corresponding Date object with the correctly set fields.*
Test Case 2: *Call convertToDate with an invalid raw date string and verify that you receive an exception.*

4.2.7.5 *Class:* PhoneNumberValidator

- ***Purpose:*** Validates that phone numbers are in the correct format.
- ***Constraints:*** *None*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

Rule Logic: *This validator shall check that all phone numbers are in the correct format. The format for phone numbers is ddddddddddd where d is a number 0-9. If this isn't met a TRAPException shall be thrown. This is defined in requirement 1.f for further reference.*

4.2.7.6 *Class:* CurrencyValidator

- ***Purpose:*** *Validates the value entered for a currency field.*
- ***Constraints:*** *None*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

Rule Logic: *The first part of this rule must check that the a currency field is a three character string as specified in requirement 1.l. Then this rule shall check in the currencyDB for this currency code to make sure it is valid.*

4.2.7.7 *Class*: EmailAddressValidator

- **Purpose:** Validates that email addresses are in the correct format.
- **Constraints:** None
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the TRAP.

Rule Logic: This validator shall check that all phone numbers are in the correct format. The format for email addresses is [local part]@[domain]. This is defined in requirement 1.h for further reference.

4.2.7.8 *Class:* OneOrMoreGrantsallValid

- **Purpose:** Checks that at least one grant is included in the form, and that all grants included are valid (i.e. grants that exist and are accessible to the user)
- **Constraints:** None
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP.*

Rule Logic: This validator shall check that at least one valid grant is used when processing the form. Since the expenses are reimbursed through grants, there is no reason to process the form if a grant does not exist in the form. If a grant is submitted with the form, it will need to be checked for validity (i.e. grant exists and can be used by the user).

4.2.7.9 *Class:* GrantPercentSumTo100

- **Purpose:** Validates that the grant percentages sum up to exactly 100%.
- **Constraints:** None
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP.*

Rule Logic: This validator shall make sure that *TRAP* only calculates reimbursement to the amount of the expenses. This is to ensure that users do

not specify funding that exceeds the total of their reimbursement. Refer to requirement 1.j for additional information.

4.2.7.10 *Class:* TransportMilesTraveledIsInt

- **Purpose:** Verifies that all transportation miles traveled are in the form of an integer that is greater than or equal to zero.
- **Constraints**: None.
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP.*

Rule Logic: Firstly, since partial miles are not counted, *TRAP* must ensure that the number is an integer. This rule also checks that the transport miles traveled is greater than or equal to zero because *TRAP* can only reimburse this expense if positive miles were travelled.

4.2.7.11 *Class:* CurrencyFieldFormat

- **Purpose:** Checks that any fields containing a currency amount and currency type are in the correct format.
- **Constraints:** None
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP.*

Rule Logic: There are multiple currencies supported by TRAP, but it is only made possible by creating a standard for different currencies. This rule checks that the currency abbreviation follows the specified format of CCC, where C is any letter A-Z. It also checks if the three-letter combination represents a currency that is recognized by the system. For further reference, refer to requirement 1.l in the requirements document.

4.2.7.12 *Class:* USCarriersOnly

- **Purpose:** Validates that the flight(s) claimed for travel expense were taken on a US based carrier.
- **Constraints:** None
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP*.

Rule Logic: The policy for reimbursement only allows US air carriers to be reimbursed by grants. If the air carrier is not US-based, the reimbursement request for the flight will be rejected. Refer to requirement 2.a for further reference.

4.2.7.13 *Class:* PerDiemLodgingCeiling

- **Purpose:** Checks that the per diem lodging expenses do not exceed the per diem limit for reimbursement.
- **Constraints:** None.

- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP*.

  Rule Logic: TRAP users submitting reimbursements for lodging are capped for reimbursement according to the per diem amount. Any expenses that go over this limit will not be reimbursed. Refer to requirement 2.b for further reference.

4.2.7.16 *Class:* OnlyOneCheckedLuggage

- **Purpose:** Checks that only one piece of luggage is claimed for reimbursement when traveling by air.
- **Constraints:** None.
- **Persistent:** Yes. An instance of this rule shall persist for the lifetime of the *TRAP*.

  Rule Logic: TRAP users are only reimbursed for one piece of luggage when traveling. Any additional luggage will not be reimbursed. Refer to requirement 2.e for further reference.

4.2.7.15 *Class:* FamilyMemberExpensesNotAllowed

- ***Purpose:*** Validates that no family members' expenses are allowed.
- ***Constraints:*** *Only checks on incidental and other expenses.*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This validator shall check the fields of other and incidental expenses for strings such as family, wife, husband, child, son, daughter, father, mother, dad, mom, etc. and throw an exception if any of those are found. See requirement 2.f for further reference.*

4.2.7.16 *Class:* CarRental

- ***Purpose:*** Checks to make sure car rental requirements are met.
- ***Constraints:*** *Only used if there is TransportationExpense with the type RENTAL_**
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This rule shall check that all of the required fields are filled out, that the car rental dates are between arrival and departure, and that car rental expenses and personal car expenses are NOT claimed on the same day. See requirement 2.g for reference.*

4.2.7.17 *Class:* Personal Car

- ***Purpose:*** Checks to make sure that personal car expense requirements are met,

- ***Constraints:*** *Only used if there is TransportationExpense with the type PERSONAL_CAR*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This rule shall check that all of the required fields are filled out, that the personal car expense dates are between arrival and departure, and that car rental expenses and personal car expenses are NOT claimed on the same day. See requirement 2.h for reference.*

4.2.7.18 *Class:* OtherExpenses

- ***Purpose:*** Checks to make sure that other expense requirements are met.
- ***Constraints:*** Only should be used if if there is an other expense.
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This rule shall check that all of the required fields are filled out. It shall also check to see if the expense is in a valid date range as defined by requirement 2.h.*

4.2.7.19 *Class:* ProperCurrencyConversion

- ***Purpose:*** Checks to make sure that any currency changes are properly made.
- ***Constraints:*** Should be used on all expenses objects that have an expense amount field and have a currency other than us dollars.
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This rule shall run through all expense objects and check to see if an expense amount is specified. If there is an expense amount than there shall be an associated currency field. This class shall take those two fields and convert the given currency and amount to us dollars using the currencyDB conversion rates. See requirement 2.k for details.*

4.2.7.20 *Class:* TransportationMileage

- ***Purpose:*** Checks to make sure Transportation Mileage calculations are done properly.
- ***Constraints:*** Should be used on all expense objects that have a mileage expense
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

  Rule Logic: *This requirement checks to see that transportation miles traveled is an Integer in all cases and it also calculates the total transportation reimbursement at $0.55 per mile. This should then update the transportation total*

expense claim with the calculated reimbursement addition. See requirement 2.k for details.

4.2.7.21 *Class:* MealPerDiem

- ***Purpose:*** **Daily** Enforces meal rules and first and last day meal rules.
- ***Constraints:*** *Should be applied to all meal expenses.*
- ***Persistent:*** *Yes. An instance of this rule shall persist for the lifetime of TRAP.*

Rule Logic:
Checks to make sure all the required fields are there. Then calculates the reimbursement by first we calculating the per diem rate for all meals on the trip by looking at the time of the meal occurrence and referencing the perDiemDB for reimbursement rates. After each day is calculated we must deduct the first and last day to 75% of what would reimbursed normally. See requirement 2.c for more information.

4.2.7.22 *Class:* IncidentalPerDiem

- Purpose: Checks that the user does not exceed the per diem limit for various incidentals.
- Constraints:
- Persistent: Yes. An instance of this rule shall persist for the lifetime of the *TRAP*.

Rule Logic: TRAP users will only be reimbursed for incidental expenses up to a certain daily limit. Any incidental that exceeds this amount will not be reimbursed. Also, only 75% of the per diem for incidentals can be reimbursed on the first and last day of travel. For further information refer to requirements 2.d and 2.n.

4.2.7.23 *Class:* GrantApproverName

- ***Purpose:*** *The grant approver's name is necessary for accounting to get approval to release reimbursement funds.*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: The grant approver's name is necessary for accounting to get approval to release reimbursement funds. Every grant that is charged an expense must have the grant approver name set. This name is looked up in the grantDB.

As a convenience to users, if the current user is the grant approver, it is left off.

4.2.7.24 *Class:* AlcoholOnlyAllowedUnderNonSponsored

- ***Purpose:*** *Sponsored grants are not allowed to reimburse alcohol expenses.*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: Alcohol is only reimbursable under a non-sponsored grant. An alcohol expense is determined by the user writing the word “alcohol” in the justification field (it is up to the user to be honest). If the expense is not related to alcohol, return from this rule and continue processing.

If a user has a non-sponsored grant available with enough funds, a restricted line item shall be created and the expense added to it.

If a user does not have a non-sponsored fund available (either no non-sponsored grants or not enough money), auditing shall stop and a BusinessLogicException shall be thrown.

4.2.7.25 *Class:* InternetOnlyUnderNonSponsoredGrants

- ***Purpose:*** *Internet costs are only reimbursable under non-sponsored grants*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: In order to claim reimbursement for internet costs, the user must provide an “Other Expense” justification, the word “internet” is searched in this field. It is assumed that the user is honest in reporting internet costs. If an expense does not contain the word “internet” (or other words deemed appropriate), this rule shall return and processing shall continue.

If the expense is internet related, a user must have a non-sponsored grant available. If a non-sponsored grant is available to the user, a restricted line item shall be created and the expense added to it.

If a user does not have a non-sponsored grant available, or there are not funds in a non-sponsored grant, a BusinessLogicException shall be thrown.

4.2.7.26 *Class:* NoExportGrantsOnlyForUSCitizens

- ***Purpose:*** *US citizenship is required for grants that specify “noExport”.*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: If a grant has the type “noExport” (looked up in the grantDB), the user must be a US citizen (looked up in the userDB). If the user is not a US citizen, they cannot be reimbursed funds under a “noExport” grant type. Processing should continue if the user has other available grants. If there is only one grant, a BusinessLogicException should be thrown.

4.2.7.27 *Class:* ForeignGrantsNoDomesticTravel

- ***Purpose:*** *Foreign grants do not pay for domestic travel (with U.S.).*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: Any transportation expense for domestic travel within the U.S. shall not be reimbursed under a foreign grant. For any domestic transportation expens a RLI shall be added for that expense.

4.2.7.28 *Class:* NIHGrantRestrictions

- ***Purpose:*** *NIH grants have specific restrictions that must be enforced and this rule is for that purpose.*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: This is a restriction rule that may only add RLI’s. Here are the rules it must enforce and which it will create RLI’s for.

- NIH grants shall not reimburse for any meal expense
- NIH grants shall not reimburse for any transportation expense except air travel and public transit.

4.2.7.29 *Class:* DoDGrantRestrictions

- ***Purpose:*** *DoD grants have specific restrictions that must be enforced and this rule is for that purpose.*
- ***Constraints:*** *Only one instance of this rule.*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: This is a restriction rule that may only add RLI’s. Here are the rules it must enforce and which it will create RLI’s for.

- DoD grants shall not reimburse for breakfast meal expenses.

- DoD grants shall only reimburse for rental car expenses through the Hertz rental car provider.
- DoD grants shall not reimburse for any non-domestic travel expenses.

4.2.7.30 *Class:* RestrictionsOnDomesticCarRental

- ***Purpose:*** *For domestic (US) car rental, there is only one approved carrier, "National Traveler"*
- ***Constraints:*** *Only one instance of this rule. This rule is superseded by DoDGrantRestrictions (4.2.7.29).*
- ***Persistent:*** *Yes, created when TRAP is started. Exists for the whole lifetime of TRAP.*

Rule Logic: Due to policy, the only allowed carrier is "National Traveler" for domestic car rental. If the carrier is not "National Traveler", an empty restricted line item shall be created. As mentioned above, the DoD grant may be able to be charged if the carrier is "Hertz". This, however, will not be determined in this business rule.

4.2.8 Data Conversion

Not enough time to complete. This class converts raw form data, a map of key value pairs, into a ReimbursementApp to be processed.

4.3 More on RestrictedLineItems (RLI's)

This section discusses how expenses will be split between grants in the face of restricted expenses. To begin to discuss this there are a set of datatypes that must be known. These types are listed below and more information on them can be found above in the class descriptions (section 4.2).

- *reimbursementTotal* which is an internal attribute of the *ReimbursementTotals* class (4.2.1.4). This holds the total amount to be reimbursed for this application. It really only represents the total after all rules have been checked (excluding the Finalize rule).

- *RestrictedLineItem - RLI (4.2.1.5)*. A *ReimbursementTotals* can have one or more of these. These represent expenses which are restricted by some rule (requirement) and can only be funded under a subset of the provided grants.
- *Grant*. A *Grant* object holds the account name and reimbursement percentage for a grant. After all rules have been processed, the Finalize will use the *reimbursementTotal* along with the percentages to determine how much each grant shall be charged.

Even if all grants have the appropriate funds for the reimbursement application, we may not be certain that all restricted items can be funded under the grants which they are eligible for. We do not want even part of restricted expenses being funded from a grant which doesn't accept the expense. This is why we track the *RestrictedLineItems (RLI)*.

Regardless of expense restrictions, all expense reimbursement amounts shall be added to the total but they may have a RLI added as well if they are restricted. In the finalization of the application we know how much we can fund from each grant. With this we go through RLI and deduct its expense amount from one of the grants it can be funded from. We deduct from the amount we can potentially fund from this grant which was calculated with the percentage for the grant. If we cannot fund the whole expense under this grant we go to the next potential one on its list and look for funds. If we cannot find any more funds or there are no more potential grants, we stop processing and throw an exception.

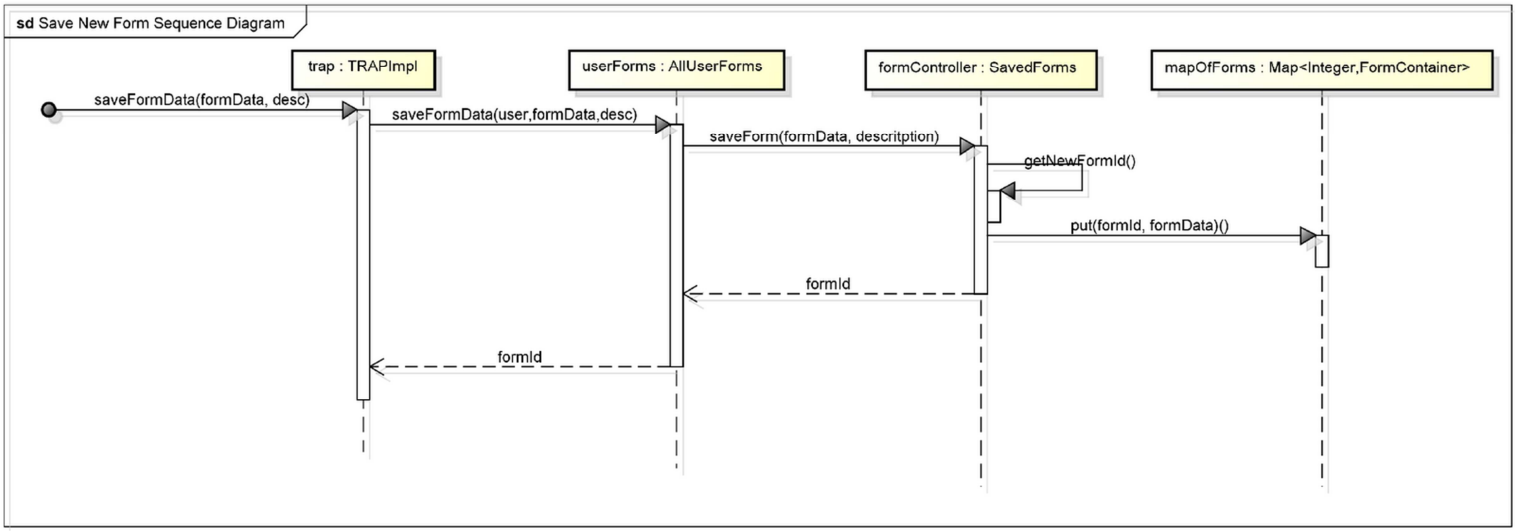
This is a greedy approach to assigning the RLI's to funding sources and it could end up with many restricted expenses being funded under one grant when they have other potential options. This could cause TRAP to fail when another restricted expense is rejected since its only potential grant is all drained. To counter this we will process RLI's in increasing order of the number of potential grants they have. This way, expenses which can only be funded under 1 grant *x* won't be rejected when other expenses who have more potential grants get funded under *x* as well. Without formally proving it, we believe this will prevent the aforementioned problem.

5. Dynamic Model

5.1 Scenarios

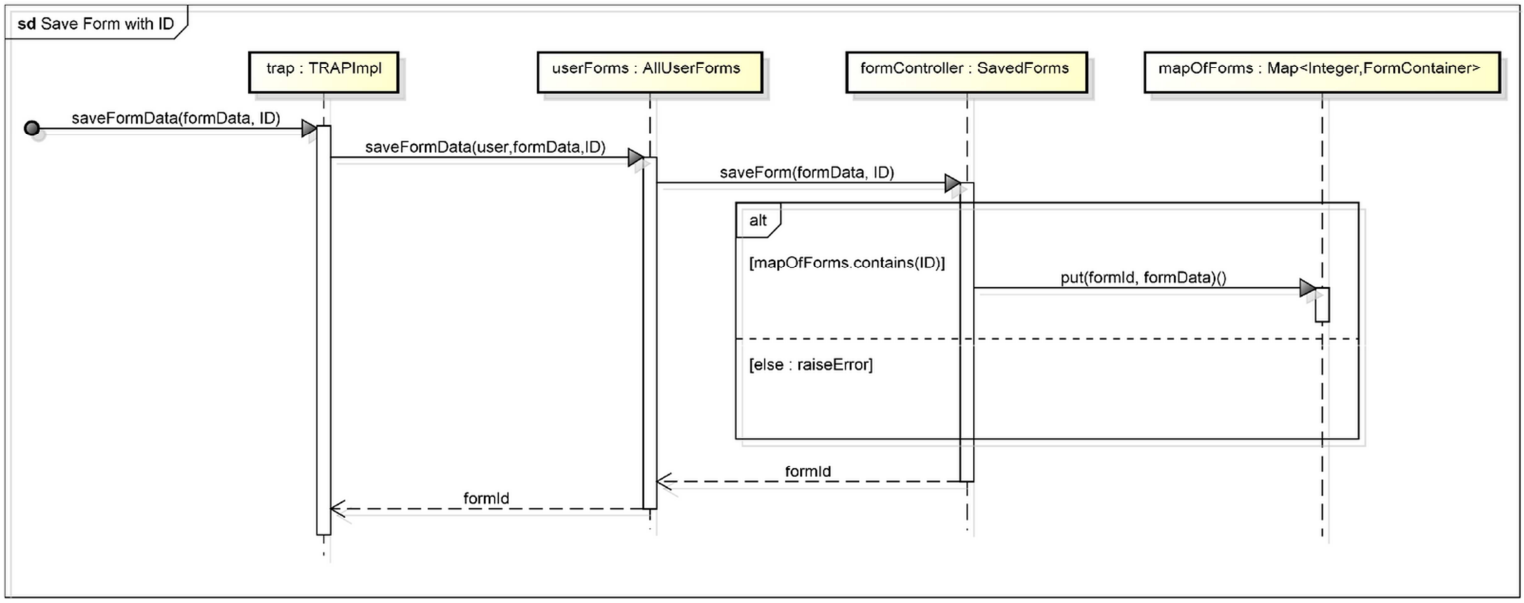
All use case diagrams are included in their raw image form along with the submission of this document in case these are too small to read. See section 7 for information on these supplementary documents.

5.1.1 Saving a form with description



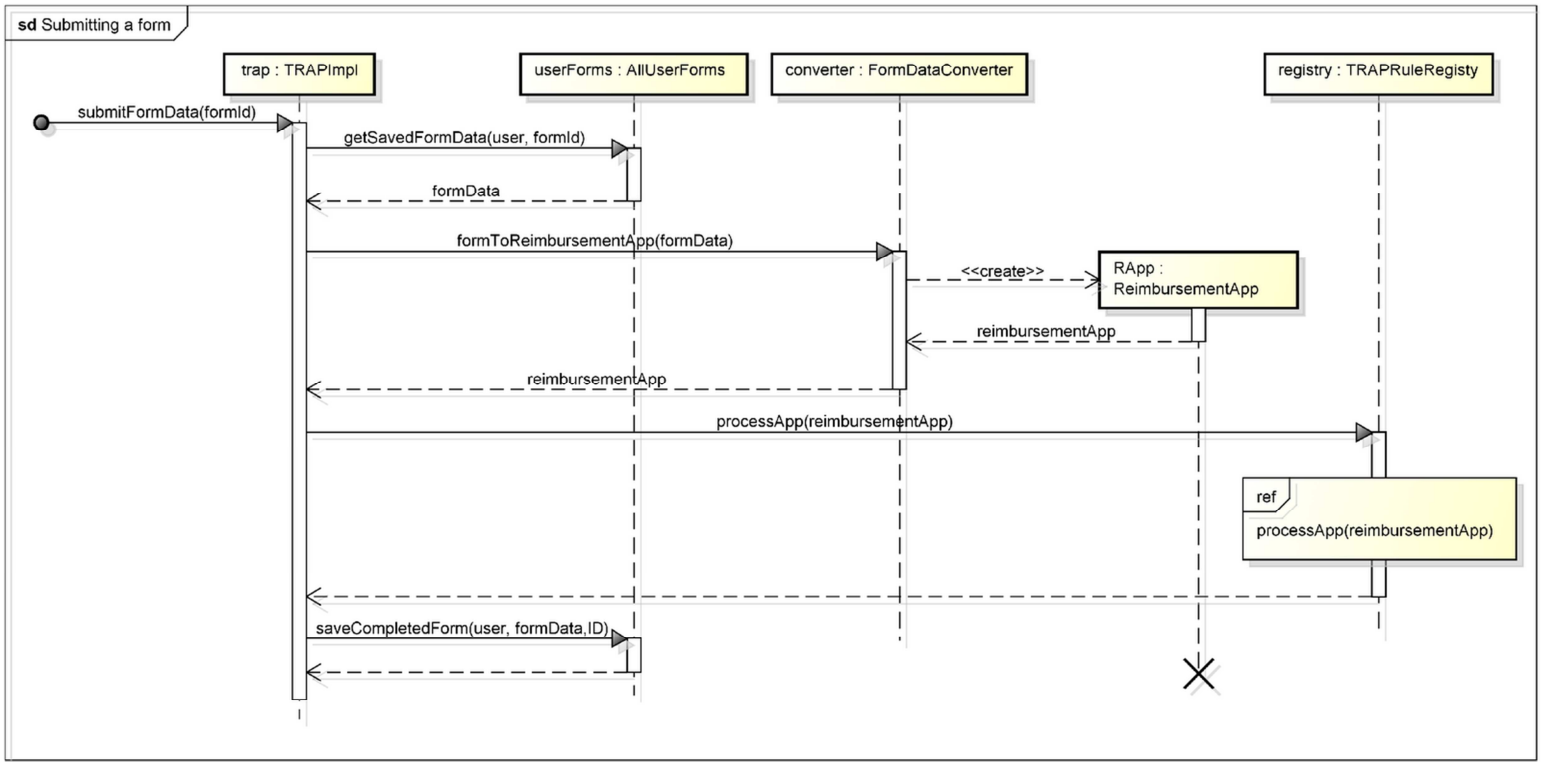
Saving a form with a description assigns the form a new ID and saves it in form storage. This storage ends up being a map of ID's to form data as shown on the right of this sequence diagram.

5.1.2 Saving a form with an ID



Saving a form with an ID saves over an existing form with the same ID. Before the form data is put into the map it must check that the storage already has a saved form with that ID. If it doesn't, TRAP shall raise an error which is represented as the conditional fragment on the right. While not very well represented, the *raise error* alternative means that an exception would be raised.

5.1.3 Submitting a form

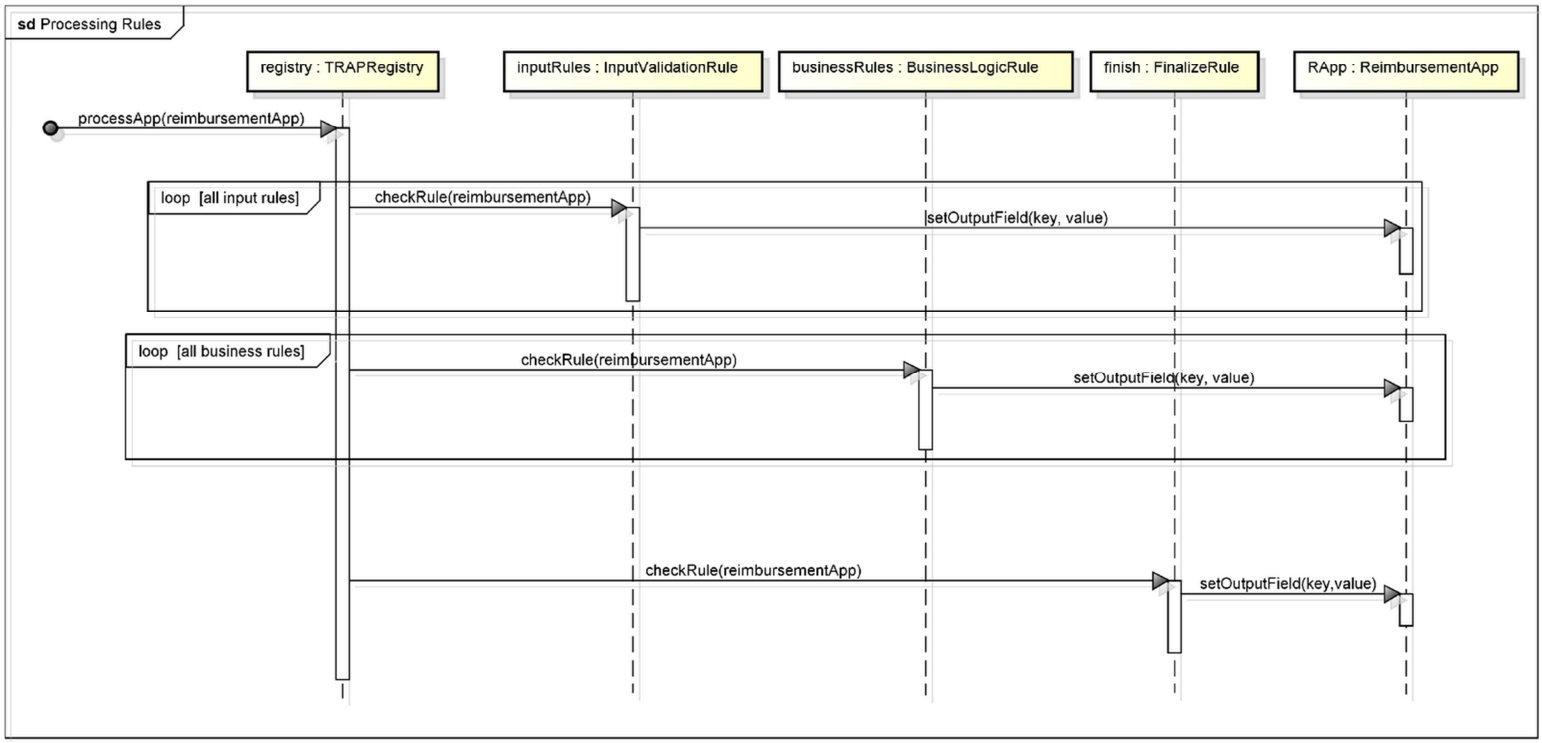


Submitting a form contains quite a few steps, some of which fall into another sequence diagram. Here are the steps diagramed above:

1. To begin, the form data for the submitted formID must be retrieved from storage. If the formID an exceptions will be raised.
2. The form data is then converted using the FormDataConverter class. This will create a new ReimbursementApp (RApp) object initialized with all data that was in the form. For the majority of form fields, all this conversion will do is copy the input strings into a representative object.
3. This RApp will be passed to the TRAPRuleRegistry instance for processing. For more detail on this processing check the referenced diagram (5.1.4).

4 After processing, if no exceptions were thrown, the RApp will have the output fields set and this can be saved back to form storage under the same formID.

5.1.4 Checking a form against rules



As rules are processed they may call the *setOutputField* method on the RApp to set a field in the output form data map. The order of rule processing will always go in this order, InputValidationRules, BusinessLogicRules and last is the FinalizeRule. The finalize rule deals with splitting the reimbursement total amongst grants. Check section 4.3 for more information on the FinalizeRule and related classes.

5.1.5 Generate Form Output

Form Output is generated as a result of checking the rules as the 5.1.4 sequence diagram shows. This form output is then saved over the input form data as shown at the bottom of the 5.1.3 sequence diagram. With our system it does not make sense to have a separate sequence diagram for the form output generation when it is intertwined with these other activity sequences.

6. Non-Functional Requirements

6.1 Database Timeouts

The database interface provided is a synchronous interface so there is the possibility that we could block indefinitely on the DB unless a timeout is set. To achieve this we will need to break one of the world assumptions we mentioned earlier which is the completely synchronous nature of TRAP. In all fairness, that assumption is primarily concerned with assumed synchronicity to the external world but to some degree it was implied that TRAP would be completely synchronous internally as well.

The DB method calls will still appear synchronous to the caller but internally there will need to be asynchronous calls to avoid blocking. With proper protection of local state this will operate identically from the caller's perspective but with the addition of a timeout exception.

6.2 Easy Business Logic Modifications

A high-level overview of how this design of TRAP supports easy business logic modification is discussed in sections 2.1 and 2.4. In the language of this design document, this non-functional requirement also includes input validation rules as well. When constructing the requirements we made no clear distinction between these two but there are some important distinctions within our system so they are considered separately.

Input Validation Rules:
Input validation rules will not add to the reimbursement total or add RLI's. Their purpose is to check the formatting or presence of input data.

Business Logic Rules:
While there is only one named type here there are really two distinctions which will be called normal rules and restriction rules. Normal rules are those business rules that input data and produce an amount to be added to the total. Normal rules may also optionally add RLI's.

Restricted rules are rules that check a restriction. An example would be the restriction that all domestic car rentals must be through the National Traveller carrier. When this rule runs it shall not add to the total since the rule for adding transportation expenses may or may not have done it but we know it will be added in the end so we don't want to double count it. What the restriction rule can do is add a RLI.

The purpose for this distinction in rules is because we have assumed, for the ease of rule modification, that there is not strict ordering of the rules with the exception of the broad input, business and finalization rule categories. As long as this rule is followed, the proper reimbursement amount will be calculated for the application. This is the only potential argument point against the ease of business logic modification in our system since it requires some cognitive effort from the maintainer and could lead to problems.

6.3 Java Implementation Language

This requirement is simply met by writing the TRAP system in Java.

7. Supplementary Documents

Included with this design are full non-partitioned versions of most images in this document. Here is what should be included (in a directory structure style):

- class_diagrams
  - All_TRAP.png - The entire class diagram
  - databases.png
  - exceptions.png
  - expenses.png
  - form_storage.png
  - grants_and_money.png
  - other_info.png
  - RApp.png
  - rules.png
  - trap_iface.png
- sequence_diagrams
  - form_process.png
  - save_w_desc.png
  - save_w_id.png
  - submit_form.png

In addition to the raw images, included are the updated requirements and test documents, *TRAPRequirements.pdf* and *TRAPTests.pdf* respectively. The changes to the requirements document are available in RequirementsDocumentChanges.pdf.

8. References

[1] D. Bettermann, A. Helgeson, B. Maurer, and E. Waytas, “Travel Reimbursement Application Processing: Requirements Document”, requirements specifications, University of Minnesota, October 2012.

[2] D. Bettermann, A. Helgeson, B. Maurer, and E. Waytas, “Travel Reimbursement Application Processing: Requirement Based Tests”, requirements based tests, University of Minnesota, October 2012.