Module Interface Specification for Plutos

Team #10, Plutos
Payton Chan
Eric Chen
Fondson Lu
Jason Tan
Angela Wang

April 4, 2025

1 Revision History

Table 1: Revision History

Date	Version	Notes
01/13/2024	0.1	Sections 2–5
01/17/2024	0.2	Section 6
03/09/2025	0.3	Updates to Item, Expense, Income, Budget, Categorization modules
04/04/2025	0.4	Updates to Transaction
...	...	...

Refer to the Software Requirements Specification (SRS) document for the list of abbreviations and acronyms (Section 1.3) and the list of symbolic constants (Section 10).

Additional abbreviations and acronyms are listed below.

Table 2: List of Abbreviations and Acronyms

symbol	**description**
M	Module
MG	Module Guide
MIS	Module Interface Specification
R	Requirement
SRS	Software Requirements Specification

Contents

List of Tables

2 Introduction

The following document details the Module Interface Specifications (MIS) for the Plutos project.

Complementary documents include the System Requirement Specifications (SRS) and Module Guide (MG). The full documentation and implementation for the project can be found in the Plutos repository.

3 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Plutos.

Table 3: Data Types

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
string	string	a sequence of characters
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
float/real	$\mathbb{R}$	any number in (-∞, ∞)
boolean	Boolean	a binary value, either true or false

The specification of Plutos uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Plutos uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

4 Module Decomposition

An overview of the module decomposition can be found in Section 5 of the Module Guide document.

5 MIS of Image Processing Module

5.1 Module

Image Processing Module

5.2 Uses

- Image processing libraries (OpenCV, PIL)
- Text parsing utilities (Pytesseract OCR)
- Input Format Module (MIS of Input Format Module)
- Categorization Module (MIS of Categorization Module)

5.3 Syntax

5.3.1 Exported Constants

None

5.3.2 Exported Access Programs

Table 4: Image Processing Module Access Programs

Name	**In**	**Out**	**Exceptions**
processImage	Image file (binary)	Text data (structured)	FileError
validateImage	Image file (binary)	Boolean	FormatError

5.4 Semantics

5.4.1 State Variables

None

5.4.2 Environment Variables

This module interacts with the file system to read image files and uses external OCR libraries or APIs to extract text data.

5.4.3 Assumptions

The input image is in a supported format (e.g., JPEG, PNG). The OCR library or API is available and correctly configured.

5.4.4 Access Routine Semantics

processImage(image: binary):

- input: A valid image file in a supported format.
- precondition: N/A
- transition: Preprocesses the image (see preprocess()) and converts it into structured text data (see extractText()).
- output: Structured text data extracted from the image.
- postcondition: N/A
- exception: Throws a FileError if the image cannot be read or an unsupported format is provided.

validateImage(image: binary):

- precondition: N/A
- transition: Validates the input image format and dimensions.
- output: Returns true if the image is valid; false otherwise.
- exception: Throws a FormatError if the image format is invalid.
- postcondition: N/A

5.4.5 Local Functions

preprocess(image: binary):

- input: A valid image file in a supported format.
- transition: This function applies pre-processing steps to the image, such as resizing, noise reduction, or thresholding, before OCR is applied.

- output: An image object.

extractText(image: binary):

- input: A valid image object.
- transition: This function uses an OCR library to extract raw text from the pre-processed image.
- output: A string containing the extracted text.

5.5 Exception Handling

The Image Processing Module may encounter various exceptions during image processing, such as:

- FileError: Raised when the image file cannot be opened or read.
  - To mitigate this, the module should check the file path and permissions before attempting to read the file.
- OCRProcessingError: Raised when the OCR library fails to extract text from the image.
  - To mitigate this, the module should implement error handling to retry the OCR process or log the error for further analysis.
  - The module should also provide feedback to the user if the OCR process fails, allowing them to take corrective action.

6 MIS of Categorization Module

6.1 Module

Categorization Module

6.2 Uses

Image Processing Module (MIS of Image Processing Module)
Expense Module (MIS of Expense)

6.3 Syntax

6.3.1 Exported Constants

N/A

6.3.2 Exported Access Programs

Table 5: Categorization Module Access Programs

Name	**In**	**Out**	**Exceptions**
categorize	item: Expense	category: string	InvalidInputError

6.4 Semantics

6.4.1 State Variables

N/A

6.4.2 Environment Variables

N/A

6.4.3 Assumptions

This specification assumes that the model has been trained and is ready to classify items based on the input data.

6.4.4 Access Routine Semantics

categorize(items: List[Expense]):

- precondition: The items in the list are valid Expense objects, that do not have a category assigned.
- transition:
$$\forall \text{item} \in \text{items}, \text{item.category} := \text{category}$$
- output: List[Expense] – a list of items with their categories set
- postcondition: The items in the list have their categories set.
- exception: InvalidInputError – thrown if the input is not a list of Expense objects

6.4.5 Local Functions

__train_model(csv: File):

- input: csv – a CSV file containing labeled data (item_name, category) from parsed receipt item names and manually verified categories.
- transition: N/A
- output: three .pkl files (model, vectorizer, encoder) for item categorization. These files are opened and loaded during the initialization of the Categorization class and are used within categorize(). It is assumed that this model has been trained and does not need to be modified during runtime.
- model: The model will build off of Cohere's categorization classify feature, using a dataset of receipt items and categories to fine-tune the model. The expected accuracy of the model is *CATEGORIZATION_ACCURACY*%.

6.5 Exception Handling

The categorization routine depends on a trained ML model and may misclassify ambiguous or novel items. To mitigate this:

- Users may manually edit the predicted category. These edits are stored for future improvement of the model (model retraining is outside this system scope).
- Items without a confident category are not assigned a class by default, ensuring users fill out all empty categories before being able to save expense items.

7 MIS of Budget Calculation Module

7.1 Module

Budget Calculation Module

7.2 Uses

Categorization Module (MIS of Categorization Module)
Transaction Module (MIS of Transaction)
Income Module (MIS of Income)
Budget Module (MIS of Budget)

7.3 Syntax

7.3.1 Exported Constants

N/A

7.3.2 Exported Access Programs

Table 6: Budget Calculation Module Access Programs

Name	**In**	**Out**	**Exceptions**
calculate_budget	expenses: List[Expense]	suggested_budget: List[Dict[string, float]]	InvalidInputError
	income: List[Income]		
	budget: List[Budget]		

7.4 Semantics

7.4.1 State Variables

N/A

7.4.2 Environment Variables

N/A

7.4.3 Assumptions

N/A

7.4.4 Access Routine Semantics

calculate_budget(history: List[Transaction], income: List[Income], budget: List[Budget]):

- precondition: The input is a list of transactions, a list of income, and a list of budgets.
- transition: N/A
- output: Dict[string, float] – a list of remaining budgets for each category and unallocated

$$\text{Categories} := \{\text{budget.category} \mid \text{budget} \in \text{Budget}\}$$

$$\text{expenses}_C := \sum_{\substack{\text{expense}_i \in \text{Expenses} \\ \text{expense}_i.\text{category} == C}} \text{expense}_i.\text{cost}$$

$$\text{budget}_C := \text{budget[C].amount}$$

$$\text{remaining}_C := \begin{cases} \text{budget}_C - \text{expenses}_C & \text{if } C \in \text{Categories} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{total\_income} := \sum_{\text{income}_i \in \text{Income}} \text{income}_i$$

$$\text{unallocated} := \text{total\_income} - \sum_{C \in \text{Categories}} \text{budget}_C - \sum_{\text{expenses}_C \mid C \notin \text{Categories}} \text{expenses}_C$$

$$\text{output} := \bigcup_{C \in \text{Categories}} (\{\text{C} : \text{remaining}_C\}) \cup \{\text{unallocated} : \text{unallocated}\}$$

- exception: InvalidInputError – thrown if the input is not valid

7.4.5 Local Functions

N/A

8 MIS of Authentication Module

This module manages user authentication, utilizing Firebase Authentication to handle login and registration. It also interacts with React Native components for the user interface.

8.1 Module

Authentication Module

8.2 Uses

This module uses Firebase for authentication purposes. It interacts with Firebase Authentication to handle user login and registration, as well as React Native components for the user interface.

8.3 Syntax

8.3.1 Exported Constants

- **auth**: An instance of Firebase Authentication, initialized using the Firebase app.
- **db**: An instance of Firebase Firestore, initialized using the Firebase app.

8.3.2 Exported Access Programs

Table 7: Authentication Module Access Programs

Name	**In**	**Out**	**Exceptions**
loginWithEmailPassword	email (string), password (string)	user object	InvalidEmail, MissingPassword, InvalidCredential, GeneralError
createUserWithEmailAndPassword	email (string), password (string)	user object	InvalidEmail, WeakPassword, GeneralError

8.4 Semantics

8.4.1 State Variables

The module maintains state for username, password, and credential error messages to support user interactions and error handling.

8.4.2 Environment Variables

The module interacts with the Firebase Authentication API for user management and relies on Firebase configuration to communicate with the backend services.

8.4.3 Assumptions

It is assumed that the Firebase configuration is valid and correctly set up. Network connectivity is also assumed to be available for authentication operations.

8.4.4 Access Routine Semantics

loginWithEmailPassword(email: string, password: string):

- precondition: The user is not logged in.
- transition: if $\exists$ (email, password) such that (email, password) $\in$ db $\Rightarrow$ auth := FirebaseAuth
- output: Returns a user object containing user details upon successful login.
- exception:
  - **InvalidEmail:** The email address is not valid.
  - **MissingPassword:** No password was provided.
  - **InvalidCredential:** Email or password is incorrect.
  - **GeneralError:** A generic error occurred during login.
- postcondition: The user is logged in and the session is active.

createUserWithEmailAndPassword(email: string, password: string):

- precondition: The user does not exist.
- transition: auth := FirebaseAuth
- output: Returns a user object containing user details upon successful registration.
- exception:
  - **InvalidEmail:** The email address is not valid.
  - **WeakPassword:** The password provided is too weak.
  - **GeneralError:** A generic error occurred during registration.
- postcondition: The user now exists in the database.

8.4.5 Local Functions

validateCredentials(username: string, password: string):

- Checks the validity of the entered username and password before attempting login or registration.

handleErrors(error: FirebaseError):

- Maps Firebase error codes to user-friendly error messages displayed on the UI.

9 MIS of Upload Interface Module

9.1 Module

Upload Interface Module

9.2 Uses

Input Format Module (MIS of Input Format Module)

9.3 Syntax

9.3.1 Exported Constants

N/A

9.3.2 Exported Access Programs

Table 8: Upload Interface Module Access Programs

Name	In	Out	Exceptions
uploadFile	File (binary)	Boolean	FileError
captureImage	Image (binary)	Boolean	CaptureError

9.4 Semantics

9.4.1 State Variables

None

9.4.2 Environment Variables

This module interacts with the device's file system for file uploads and the device's camera for image capture. It also forwards the captured or uploaded files to the Input Format Module for preprocessing.

9.4.3 Assumptions

- The device has a functioning file system for uploads or a working camera for image capture.
- The file or image is in a supported format (e.g., JPEG, PNG).

- The Input Format Module is available and correctly configured.

9.4.4 Access Routine Semantics

uploadFile():

- transition: Validates and uploads the file, then forwards it to the Input Format Module.
- output: Returns true if the file upload is successful; false otherwise.
- exception: Throws a FileError if the file cannot be uploaded or is in an unsupported format.

captureImage():

- transition: Captures an image using the device's camera and forwards it to the Input Format Module.
- output: Returns true if the image capture is successful; false otherwise.
- exception: Throws a CaptureError if the image capture fails.

9.4.5 Local Functions

- validateFile(): Ensures the uploaded file meets format and size requirements before processing.
- preprocessImage(): Applies basic pre-processing steps to captured images, such as resizing or compression.

10 MIS of Results Display Module

10.1 Module

Results Display Module

10.2 Uses

Input Format Module (MIS of Input Format Module)
Budget Calculation Module (MIS of Budget Calculation Module)
Categorization Module (MIS of Categorization Module)

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

Table 9: Input Format Module Access Programs

Name	In	Out	Exceptions
displayResults	CategorizedData, BudgetPlan	Visualization (UI Component)	DisplayError
generateExportFile	FormatType (CSV, PDF)	File	ExportError

10.4 Semantics

10.4.1 State Variables

- `currentView`: Stores the current state of the results view, such as the displayed budget or expense category.
- `dataCache`: Temporarily holds the processed data for display purposes.

10.4.2 Environment Variables

- Screen interface: Used to display the results in a user-friendly format.
- File system: Accessed for exporting reports in specified formats.

10.4.3 Assumptions

- Assumes the input data has been properly processed and categorized by upstream modules.
- Assumes the screen interface and file system are operational and accessible.

10.4.4 Access Routine Semantics

`displayResults`(CategorizedData, BudgetPlan):

- **transition:** Updates the `currentView` to display the given data.
- **output:** Renders categorized expenses, financial suggestions, and budget plans in a user-friendly format.
- **exception:** Throws `DisplayError` if the UI components fail to render properly.

`generateExportFile`(FormatType):

- **transition:** Creates an exportable file (CSV or PDF) from the current view data.
- **output:** Returns a file object for download or storage.
- **exception:** Throws `ExportError` if file generation fails due to unsupported format or data corruption.

10.4.5 Local Functions

- `formatDataForExport(Data, FormatType)`: Converts the processed data into the specified format for export purposes.
- `updateView(ViewState)`: Updates the UI state to reflect the latest data or user interaction.

11 MIS of Input Format Module

11.1 Module

Input Format Module

11.2 Uses

Results Display Module (MIS of Results Display Module)
Budget Calculation Module (MIS of Budget Calculation Module)
Categorization Module (MIS of Categorization Module)

11.3 Syntax

11.3.1 Exported Constants

None

11.3.2 Exported Access Programs

Table 10: Input Format Module Access Programs

Name	**In**	**Out**	**Exceptions**
parseInputData	RawInputData	StructuredData	ParseError
validateInputFormat	RawInputData	Boolean	ValidationError

11.4 Semantics

11.4.1 State Variables

- `lastParsedData`: Stores the most recent successfully parsed input data.

11.4.2 Environment Variables

- File system: For reading input data files or streams.

11.4.3 Assumptions

- Assumes input data adheres to a general predefined structure, such as receipts in text or image format.
- Assumes input errors (e.g., invalid formats) will be handled through exceptions.

11.4.4 Access Routine Semantics

`parseInputData`(RawInputData):

- **transition:** Updates `lastParsedData` with the processed input.
- **output:** Converts raw input data into a structured format usable by other modules.
- **exception:** Throws `ParseError` if the input cannot be parsed due to formatting issues or invalid data.

`validateInputFormat`(RawInputData):

- **transition:** None
- **output:** Returns `true` if the input format is valid, `false` otherwise.
- **exception:** Throws `ValidationError` for non-parsable or unsupported formats.

11.4.5 Local Functions

- `extractRelevantFields(RawInputData)`: Isolates key fields from raw input for processing.
- `mapToStructuredFormat(Fields)`: Maps extracted fields to the final structured format.

12 MIS of Output Generation Module

12.1 Module

Output Generation Module

12.2 Uses

Budget Calculation Module (MIS of Budget Calculation Module)
Results Display Module (MIS of Results Display Module)

12.3 Syntax

12.3.1 Exported Constants

N/A

12.3.2 Exported Access Programs

Table 11: Output Generation Module Access Programs

Name	In	Out	Exceptions
generateReport	ProcessedData, FormatType	ReportFile	ReportGenerationError
displayOnScreen	ProcessedData	DisplayOutput	DisplayError
exportFile	ProcessedData, ExportType	ExportFile	ExportError

12.4 Semantics

12.4.1 State Variables

- `lastGeneratedReport`: Stores the last successfully generated report for reference or re-export.

12.4.2 Environment Variables

- File system: For saving reports and exported files (e.g., CSV, PDF).
- User interface: For displaying outputs on screen.

12.4.3 Assumptions

- Assumes input data (ProcessedData) is correctly formatted and validated by preceding modules.
- Assumes the requested output format (e.g., CSV, PDF) is supported by the module.

12.4.4 Access Routine Semantics

`generateReport`(ProcessedData, FormatType):

- **transition:** Updates `lastGeneratedReport` with the newly created report.
- **output:** Produces a formatted report file in the specified format (e.g., PDF, CSV).
- **exception:** Throws `ReportGenerationError` if the report generation fails (e.g., due to unsupported format or missing data).

`displayOnScreen`(ProcessedData):

- **transition:** None
- **output:** Renders the processed data into a user-friendly display format on the screen.
- **exception:** Throws `DisplayError` if the display rendering fails.

`exportFile`(ProcessedData, ExportType):

- **transition:** Saves the processed data into an exportable file format (e.g., CSV, Excel).
- **output:** Returns a reference to the exported file.
- **exception:** Throws `ExportError` if the export operation fails.

12.4.5 Local Functions

- `formatData(ProcessedData, FormatType)`: Converts the processed data into the requested format.
- `renderScreenOutput(ProcessedData)`: Prepares the processed data for screen display.

13 MIS of Transaction

13.1 Module

Transaction module

13.2 Uses

Expense module (MIS of Expense)

13.3 Syntax

13.3.1 Exported Constants

- **store**: string
- **transaction_date**: datetime
- **user_id**: Uuid
- **items**: List[Expense]

13.3.2 Exported Access Programs

Table 12: Expense Module Access Programs

Name	In	Out	Exceptions
expense	store: string transaction_date: datetime user_id: Uuid items: List[Expense]		

13.4 Semantics

13.4.1 State Variables

N/A

13.4.2 Environment Variables

N/A

13.4.3 Assumptions

N/A

13.4.4 Access Routine Semantics

transaction(store: string, transaction_date: datetime, user_id: Uuid, items: List[Expense]):

- transition: N/A
- output: Transaction
- exception: InvalidInputError – thrown if the input is not valid

13.4.5 Local Functions

N/A

14 MIS of Expense

14.1 Module

Expense module

14.2 Uses

N/A

14.3 Syntax

14.3.1 Exported Constants

- **name**: string
- **cost**: float
- **category**: string

14.3.2 Exported Access Programs

Table 13: Expense Module Access Programs

Name	In	Out	Exceptions
expense	name: string cost: float category: string		

14.4 Semantics

14.4.1 State Variables

N/A

14.4.2 Environment Variables

N/A

14.4.3 Assumptions

N/A

14.4.4 Access Routine Semantics

expense(name: string, cost: float, category: string):

- transition: N/A
- output: Expense
- exception: InvalidInputError – thrown if the input is not valid

14.4.5 Local Functions

N/A

15 MIS of Income

15.1 Module

Income module

15.2 Uses

N/A

15.3 Syntax

15.3.1 Exported Constants

- **name**: string
- **amount**: float
- **recurring**: bool
- **frequency**: Optional[string]
- **start**: datetime
- **user_id**: Uuid

15.3.2 Exported Access Programs

Table 14: Income Module Access Programs

Name	**In**	**Out**	**Exceptions**
income	name: string amount: float recurring: bool frequency: Optional[string] start: datetime user_id: Uuid		

15.4 Semantics

15.4.1 State Variables

N/A

15.4.2 Environment Variables

N/A

15.4.3 Assumptions

N/A

15.4.4 Access Routine Semantics

income(name: string, amount: float, recurring: bool, frequency: Optional[string], start: datetime, user_id: Uuid):

- transition: N/A
- output: Income
- exception: InvalidInputError – thrown if the input is not valid

15.4.5 Local Functions

N/A

16 MIS of Budget

16.1 Module

Budget module

16.2 Uses

N/A

16.3 Syntax

16.3.1 Exported Constants

- **category**: string
- **amount**: float
- **user_id**: Uuid

16.3.2 Exported Access Programs

Table 15: Budget Module Access Programs

Name	**In**	**Out**	**Exceptions**
budget	category: string amount: float user_id: Uuid		InvalidInputError

16.4 Semantics

16.4.1 State Variables

N/A

16.4.2 Environment Variables

N/A

16.4.3 Assumptions

N/A

16.4.4 Access Routine Semantics

budget(category: string, amount: float, user_id: Uuid):

- transition: N/A
- output: Budget
- exception: InvalidInputError – thrown if the input is not valid

16.4.5 Local Functions

N/A

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

   During the deliverable, the team was able to divide the work up well and the process went seamlessly for each member finishing up their corresponding section punctually. As well, any questions or concerns that the team had were brought up and discussed in an orderly manner in order to resolve any confusion.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   During the completion of the MG document, we were confused regarding the structure of the DAG diagram, and what the module hierarchy was supposed to look like. The issues surrounding the module hierarchy were resolved during our meeting with Lucas, and the remaining issues we had with the DAG diagram were discussed as a group. A major pain point was the structure of the DAG diagram and after a lengthy discussion, we reached out to Lucas with a couple of solutions to compare which meets the requirements of the section better.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   One of the main design decisions that we weren't sure about was what users want to see on their home page. After asking potential users, we found that a potential pain point (when it comes to budgetting) for many users was that they are unaware of how much they've spent over a certain interval, and how much is left in their budget due to the accumulation of small expenses. Furthermore, we asked users about potential features that they'd like to have included into the app and one of the most requested features were spending metrics. The combination of these feedback points led to us displaying users spending metrics and habits on the home page.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

   During the completion of the design doc, we expect that the MIS doc may undergo changes as we haven't fully built out all of the modules outlined in the MG/MIS document. We anticipate that there may be changed regarding the software archietecture. However, we may need to update our Hazard Analysis and SRS corresponding to what receipt types are expected after further testing.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   In order to make the project better, we were thinking of adding more functionality (i.e. a social aspect where you could split bills with others within the app) and adjusting the classification and parsing model to be able to identify any type of receipt. For example, currently, some receipts heavily abbrieviate their items on the receipt, making it difficult for the classification model to be able to identify the item and categorize it. With more resources, we could potentially integrate our system with common grocery store chains to train the model in order to identify those cryptic items on receipts.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   One of the main design decisions that we stayed away from was the use of too many user inputs. Our solution is to make a more efficient experience for users, and as a result we decided to stay away from designs that would require many user input fields. We also considered the design of using a dictionary for data types instead of an object, but we decided that an object would be cleaner and more efficient for our purposes. While a dictionary would have been easier to define, we decided to go with creating new classes because it would ensure that the data is structured in a way that would minimize errors (e.g., Item (**??**) and Expense (14)).