Design Document

Team 1

11 March 2012

Table 1: Team

Name	ID Number
Jonathan Bergeron	9764453
Marc-André Faucher	9614729
Jeffrey How	9430954
Dmitry Kuznetsov	5679311
William Ling	9193480
Thomas Paulin	9333630
Alain Sakha	9770836
Kai Wang	5652723

1 Introduction

The purpose of this document is to provide the reader with an understanding of the design of the Task Manager system. Each section provides differing levels of detail. We begin with a high-level description of the architectural design followed by a detailed description of the Model, View and Controller subsystems. We conclude the document with examples of execution scenarios to provide the reader with concrete examples of design data flow.

2 Architectural Design

A foundational background of the overall system is required in order to understand the goals of the respective subsystems and their relationships.

2.1 Architectural Diagram

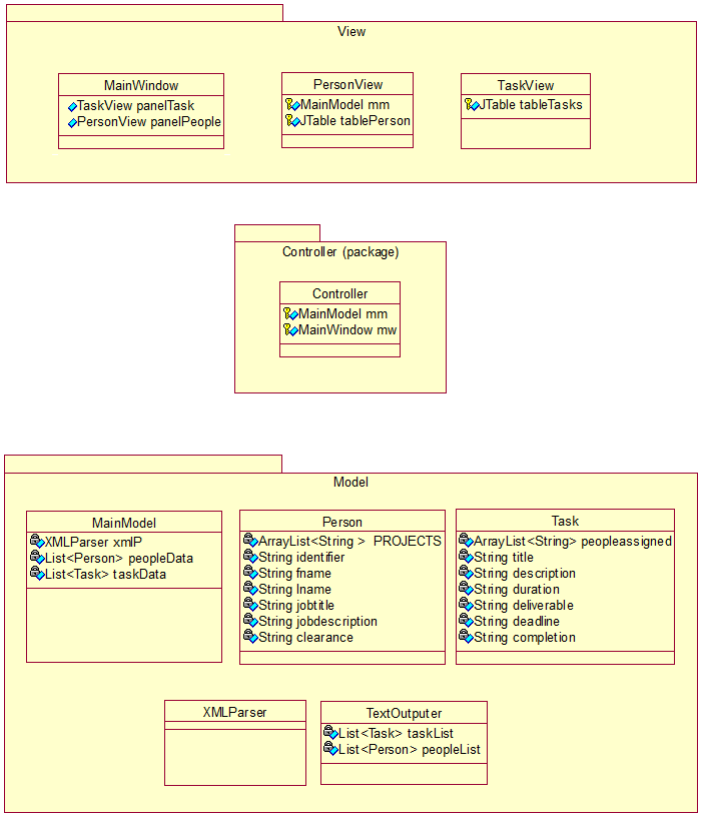

Figure 1: Architectural Design Diagram

The Task Manager consists of three main subsystems; Model, View, and Controller. This design was chosen due to its feasibility and synergy with simple Graphical User Interfaces.

The Model is the hub for all essential data and data manipulation. It interfaces with input/output classes to read and write data to XML and text files. Additionally, it handles all operations that directly modify the Person and Tasks information.

The View supports the user inteface. It is used to display information to the user in varying table modes.

Finally, the Controller facilitates communication and data synchronization between the Model and View subsystems. It acts as an intermediary and allows for a highly cohesive and lowly coupled interface environment.

2.2 Subsystem Interface Specifications

The following flowchart displays the various subsystems of the Task Manager . Each column represents a subsystem, except for I/O which is a part of the Model subsystem. The arrows represent specific function calls used in order to communicate between subsystems. The legend discusses the interface relationships and how they are used cross-subsystem to achieve different functional goals. Please refer to the UML sequence diagrams for more examples related to the interfacing between subsystems.

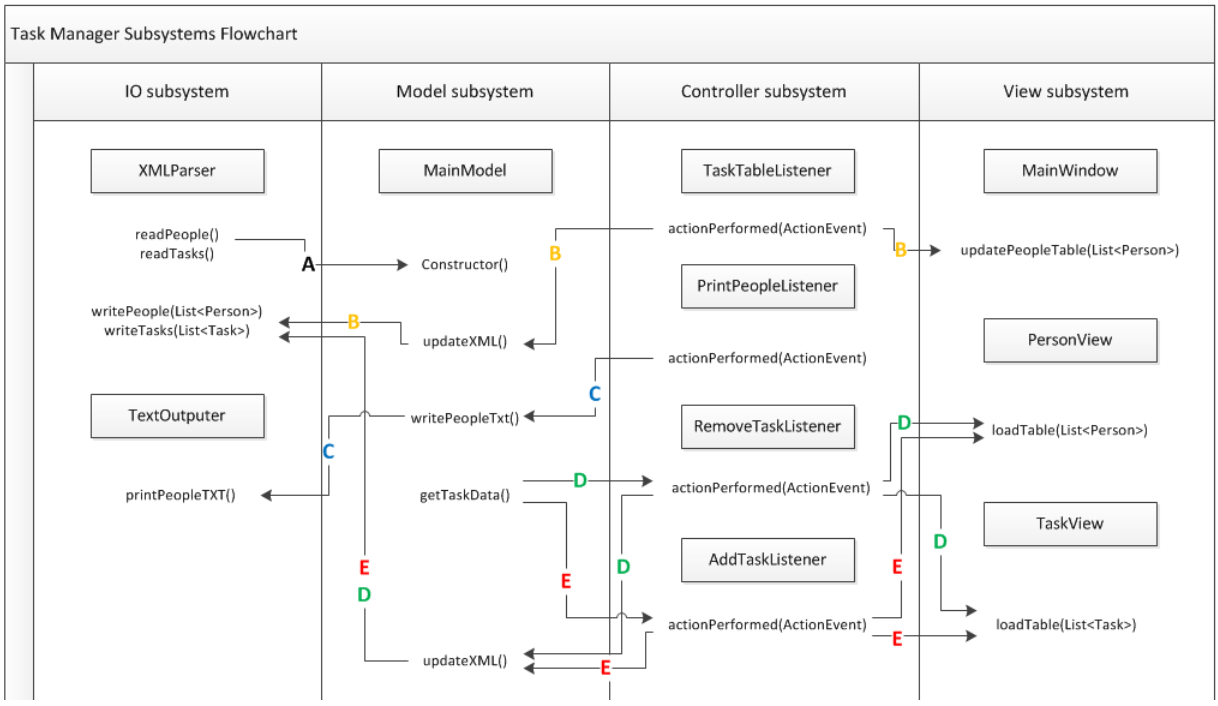

Figure 2: Subsystem Interface Flowchart

Flowchart Legend:

A: Constructor for MainModel calls readPeople and readTasks from XMLParser to instantiate Model data.

B: When a change is performed by the user in the Task Table, the TaskTableListener handles the event by calling updateXML in the MainModel and updatePeopleTable in MainWindow. This ensures that the data in both structures is updated and consistent.

UpdateXML is used to update the XML document. The function uses the writePeople and writeTasks function in XMLParser.

C: When a user presses the 'Print' button, the PrintPeopleListener in the Controller calls writePeopleTxt in the MainModel. The function writePeopleTxt interfaces with printPeopleTXT from the TextOutputter class, which does all the work for outputting to a text file.

D: When the user removes a task on the table, the RemoveTaskListener obtains the Task data using the getTaskData function from the instance of MainModel. Afterwards, it updates the Table Views accordingly by calling the loadTable function in the PersonView and TaskView. Additionally, the XML document also gets updated via a call to the updateXML function.

E: When the user adds a task on the table, the AddTaskListener obtains the Task data using the getTaskData function from the instance of MainModel. Afterwards, it updates the TaskView accordingly by calling the loadTable function. After a task is added, the user will have to input task data, which will be handled by the TaskTableListener. Thus, the AddTaskListener does not need to update PersonView or call updateXML().

Figure 3: Flowchart Legend

3 Detailed Design

As stated previously, the system is composed of the Model, View and Controller subsystems.

3.1 Model Subsystem

Detailed Design Diagram

Please refer to Figure 4.

Units Description

MainModel

Description

MainModel contains the XMLParser class (part of our I/O Subsystem), TextOutputer class (also part of I/O subsystem, responsible for writing to an XML file). MainModel also contains Person and Task classes. The purpose of this class is to group Person and Tasks data into same class and add functionality for writing that data to a file (I/O subsystem).

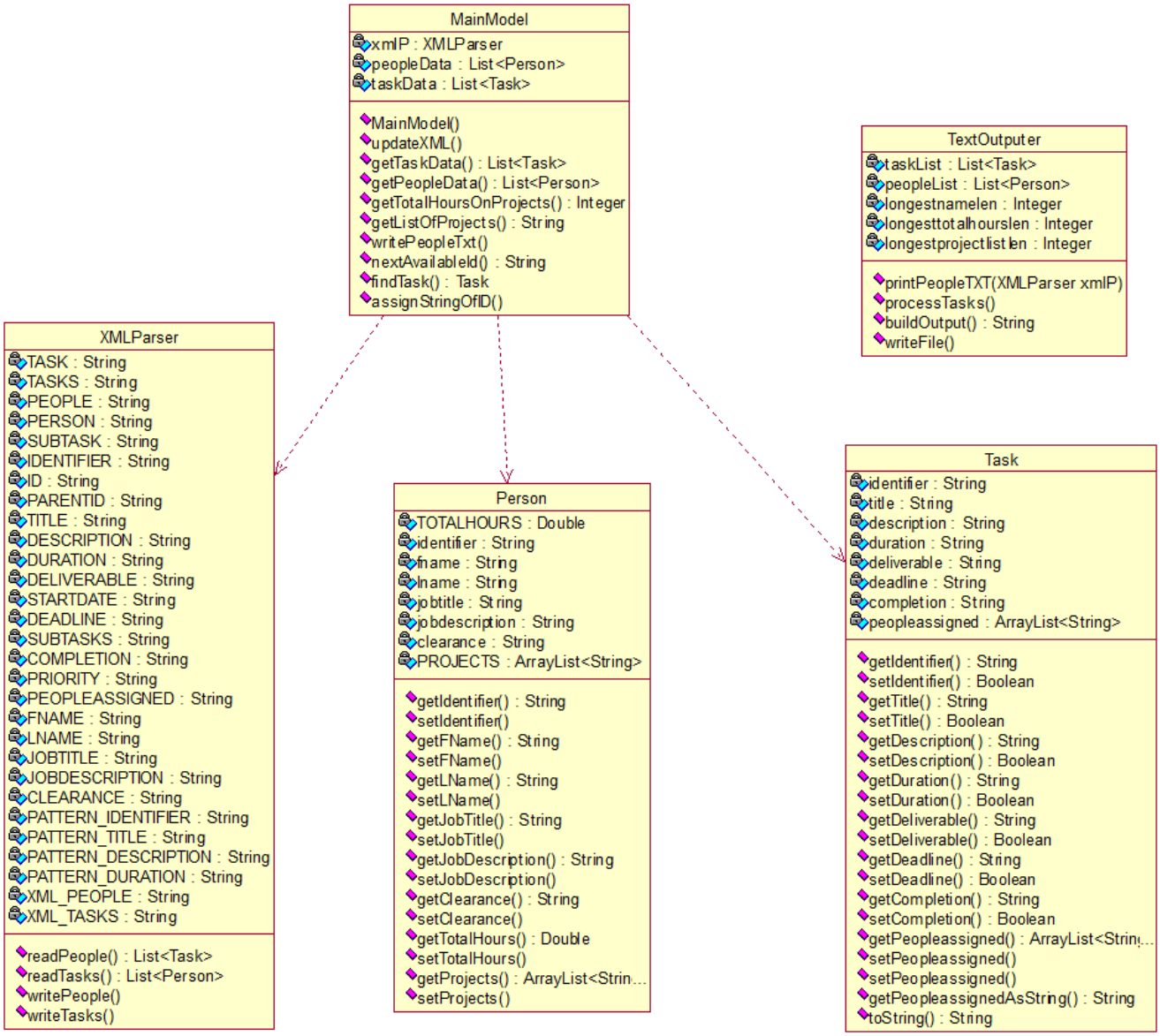

Figure 4: Model Design Diagram

Attributes:
`XMLParser xmlP`: The XML Parser
`List<Person> peopleData`: List of persons in the XML file
`List<Task> taskData`: List of task in the XML file

Methods:
`MainModel()`: *Constructor*
`void updateXML()`: *writes to XML file using XMLParser's writePeople and writeTasks methods*
`void writePeopleTxt()`: *calls TextOutputer's printPeopleTXT(xmlP) method*
`Task findTask(String)`: *Find a tastk by id from taskData List*
`List<Task> getTaskData()`: *getter*
`List<Person> getPeopleData()`: *getter*
`int getTotalHoursOnProjects(String)`: *gets Person's total hours*
`void writePeopleTxt()`: *writes xmlParser to file using TextOutputer's static function*
`String nextAvailableId()`: *Gets the next available ID (from the list of Tasks) when creating a new Task*
`String getListOfProjects(String)`: *Gets the list of Tasks in a string, separated by a comma ','.*
`void assignStringOfID(Task task, String data)`: *When assigning new people to a Task, it validates the format of string ids and assigns new People to a Task*

XMLParser

Description
The purpose of this class is to parse people.xml and tasks.xml files and extract appropriate data. It also writes to XML files.

Attributes:(Tags used to extract information when parsing)
`String TASK`
`String TASKS`
`String PEOPLE`
`String PERSON`
`String IDENTIFIER`
`String ID`
`String PARENTID`
`String TITLE`
`String DESCRIPTION`
`String DURATION`
`String DELIVERABLE`
`String STARTDATE`
`String DEADLINE`

`String SUBTASKS`
`String COMPLETION`
`String PRIORITY`
`String PEOPLEASSIGNED`
`String FNAME`
`String LNAME`
`String JOBTITLE`
`String JOBDESCRIPTION`
`String CLEARANCE`
`String PATTERN_IDENTIFIER`
`String PATTERN_TITLE`
`String PATTERN_DESCRIPTION`
`String PATTERN_DURATION`
`String XML_PEOPLE`
`String XML_TASKS`
`String IDENTIFIER`

Methods:
`void writeTasks(List<Task>)`: *receives list of tasks and writes them to XML file*
`void writePeople(List<Person>)`: *receives list of people and writes them to XML file*
`List<Task> readTasks()`: *parses XML file and reads in tasks information, returning List of tasks*
`List<Person> readPeople()`: *parses XML file and reads in people information, reurning List of people*

TextOutputer

Description
The purpose of this class is to format and write data (People) to people.txt

Attributes:
`List<Task> taskList`
`List<Person> peopleList`
`int longestnamelen`
`int longesttotalhourslen`
`int longestprojectlistlen`

Methods:
`void printPeopleTXT(XMLParser)`: *Writes new data to file using "writeFile" by first reading it from XMLParser, building output then calling "writeFile".*
`void processTasks()`
`String buildOutput()`: *Builds output string from list of Tasks and Persons*

and returns the string
`void writeFile(String output)`: *Gets the string and writes it to people.txt*

Person

Description
The purpose of this class is to abstract and encapsulate People data.

Attributes:
```
double TOTALHOURS
ArrayList<String> PROJECTS
String identifier
String fname
String lname
String jobtitle
String jobdescription
String clearance
```

Methods:
```
String getIdentifier()
void setIdentifier(String)
String getFName()
void setFName(String)
String getLName()
void setLName(String)
String getJobTitle()
void setJobTitle(String)
String getJobDescription()
void setJobDescription(String)
String getClearance()
void setClearance(String)
double getTotalHours()
void setTotalHours(double)
ArrayList<String> getProjects()
void setProjects(String)
```
`String toString()`: *Overrides toString method and returns a list of attributes and their values, separated by ',' comma*

Task

Description
The purpose of this class is to abstract and encapsulate Task data.

Attributes:

`String identifier`
`String title`
`String description`
`String duration`
`String deliverable`
`String deadline`
`String completion`
`ArrayList<String> peopleassigned`

Methods:
`public Task()`: *Default Constructor*
`public Task(String)`: *Constructor*
`String getIdentifier()`
`boolean setIdentifier(String)`
`String getTitle()`
`boolean setTitle(String)`
`String getDescription()`
`boolean setDescription(String)`
`String getDuration()`
`boolean setDuration(String)`
`String getDeliverable()`
`boolean setDeliverable(String)`
`String getDeadline()`
`boolean setDeadline(String)`
`String getCompletion()`
`boolean setCompletion(String)`
`ArrayList<String> getPeopleassigned()`
`void setPeopleassigned(String)`
`void setPeopleassigned(ArrayList<String>)`
`String getPeopleassignedAsString()`
`String toString()`: *Overrides toString method and returns a list of attributes and their values, separated by ',' comma*

3.2 View Subsystem

Detailed Design Diagram

Please refer to Figure 5.

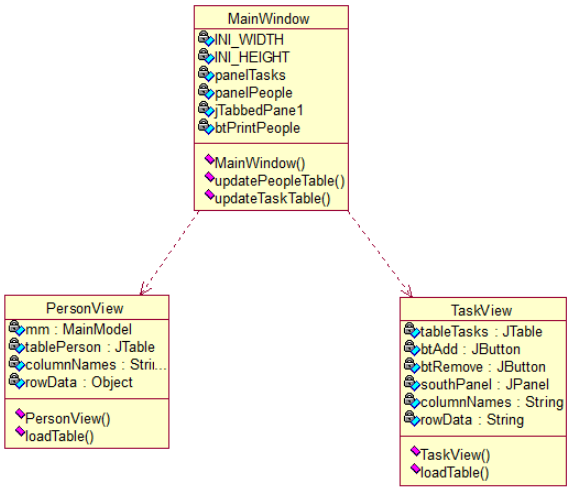

Figure 5: View Design Diagram

Units Description

TaskView

Description

This class is responsible for displaying the list of tasks in a table format. It gets the data in the constructor, which uses its loadTable method to populate the table.

Attributes:

`JButton btAdd`
`JButton btRemove`
`String[] columnNames`
`String[][] rowData`
`JPanel southPanel`
`JTable tableTasks`

Methods:

`Taskview(List<Task> taskData)`: *Constructor*
`void loadTable(List<Task>)`: *Gets the list of tasks from the model and populates the table with data*

PersonView

Description

This class is responsible for displaying the list of people in a table format. It

gets the data in the constructor, which uses its loadTable method to populate the table.

Attributes:
`String[] columnNames`
`MainModel mm`
`Object[][] rowData`
`JTable tablePerson`

Methods:
`Personview(MainModel mm)`: *Constructor*
`void loadTable(List<Person>)`: *Gets the list of people from the model and populates the table with the data*

MainWindow

Description
This class creates the window that contains the Tasks and People tabs. It sets the layout, buttons and panel at the bottom of the screen. In the event a button is pressed, this class does not handle the event. A listener in the Controller catches the event and sends MainWindow the updated data.

Attributes:
`int INI_HEIGHT`
`int INI_WIDTH`
`JButton btPrintPeople`
`JTabbedPane jTabbedPane1`
`PersonView panelPeople`
`TaskView panelTasks`

Methods:
`MainWindow(MainModel mm)`: *Constructor*
`updatePeopleTable(List<Person>)`: *Calls Person's view loadTable method, which gets the data from the model and loads intot he table*
`updateTaskTable(List<Task>)`: *Calls task's view loadTable method, which gets the data from the model and loads into the table*

3.3 Controller Subsystem

Detailed Design Diagram

Please refer to Figure 6.

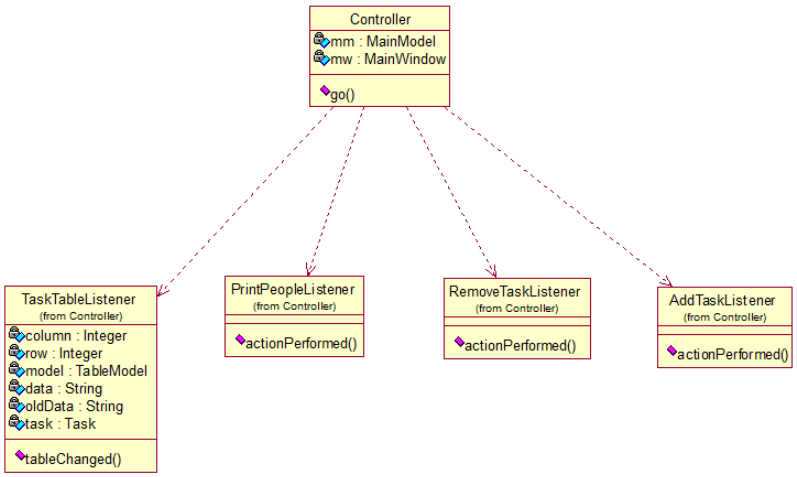

Figure 6: Controller Design Diagram

Units Description

Controller

Description

Controller class contains MainModel and MainWindow variables. And it also contains 4 listeners. MainModel is a model attribute which has underlying data and methods to manipulate the data (read, update, write to files). The MainWindow is the main application window class, which contains 2 view: PersonView and TaskView.

Attributes:

`MainModel mm`
`Mainwindow mw`
`class TaskTableListener`
`class PrintPeopleListener`
`class RemoveTaskListener`
`class AddTaskListener`

Methods:

`void updatePeopleTable(List<Person> pList)`: *Creates and initializes a new model MainModel and creates and initializes a new window MainWindow which contains 2 views: PersonView and TaskView.*

4 Dynamic Design Scenarios

The following sequence diagrams represent three execution scenarios of the system. Each diagram follows the flow of data through the system design to achieve a system-level service. Note that the execution scenario for removing a task is almost identical to the scenario for adding a task.

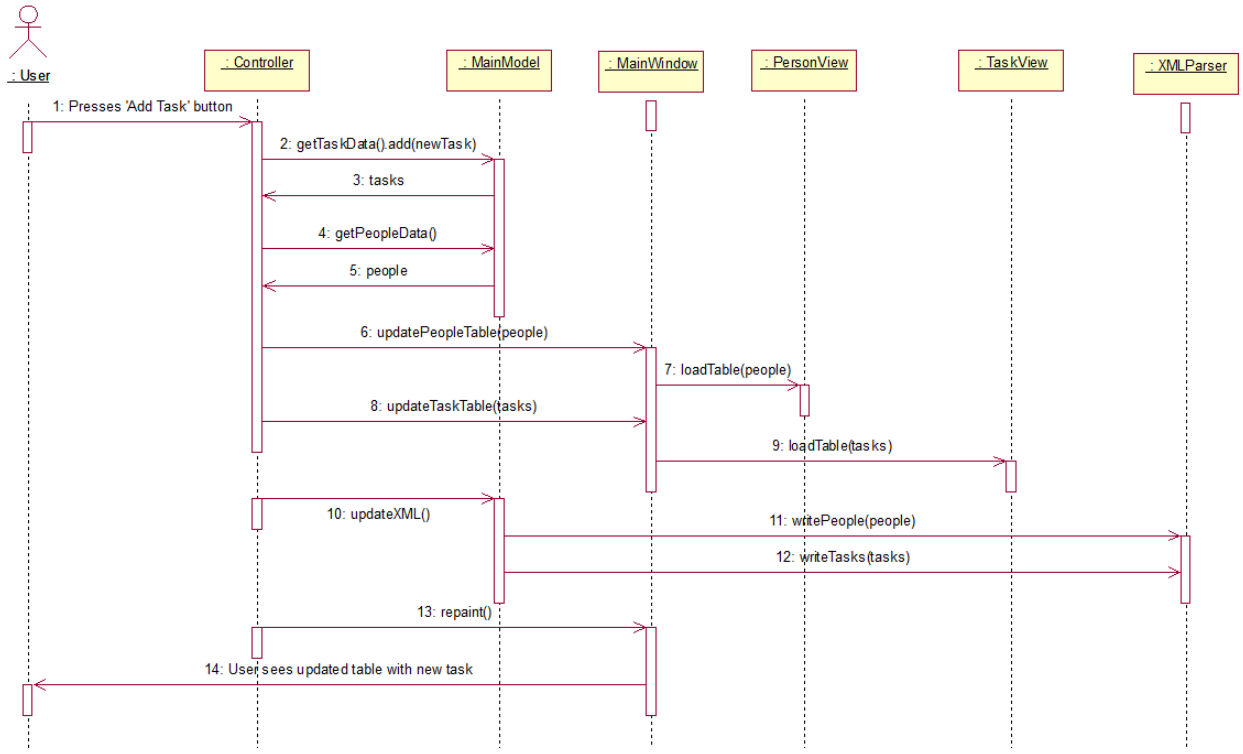

Figure 7: Execution Scenario: User adds task to table

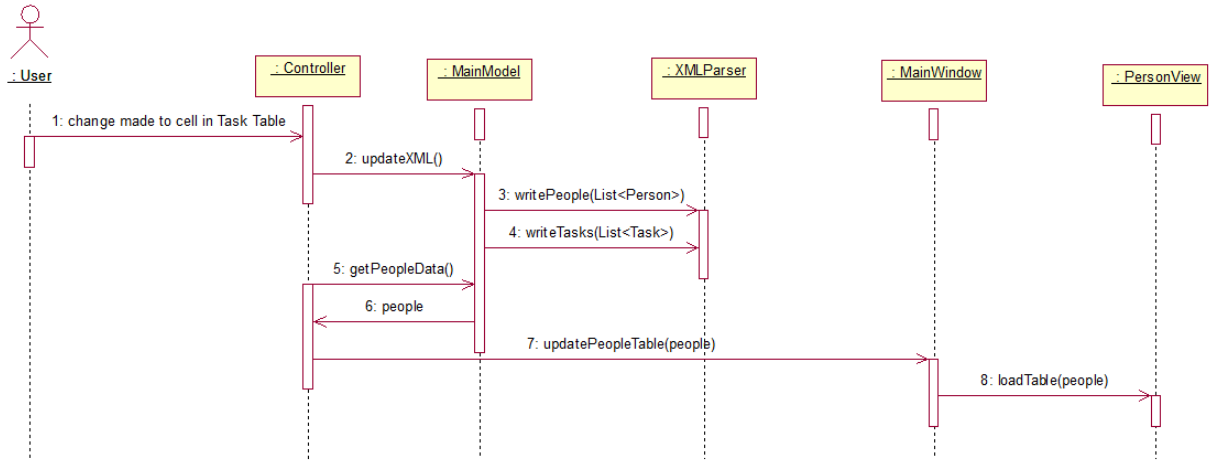

Figure 8: Execution Scenario: User changes cell on table

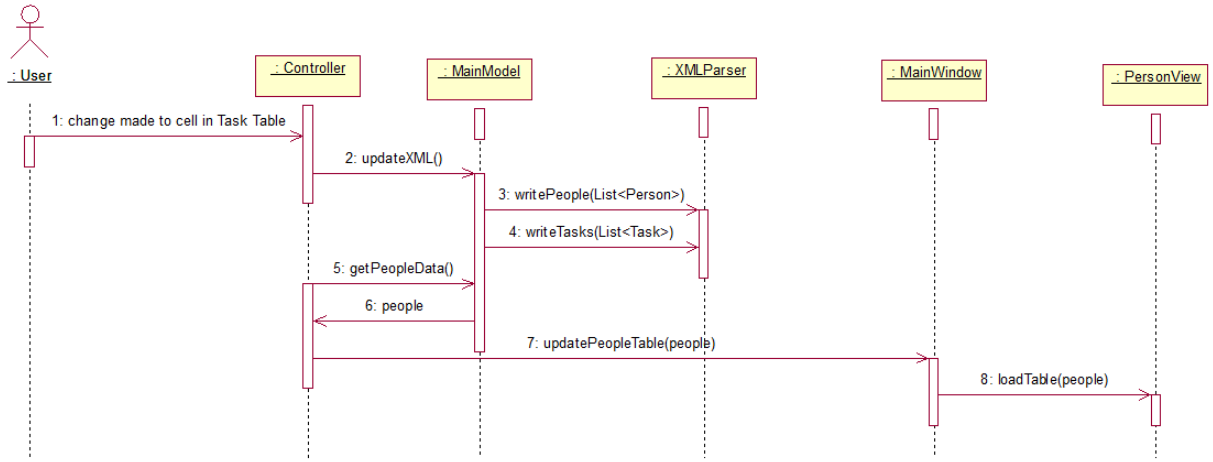

Figure 9: Execution Scenario: User requests to print text file