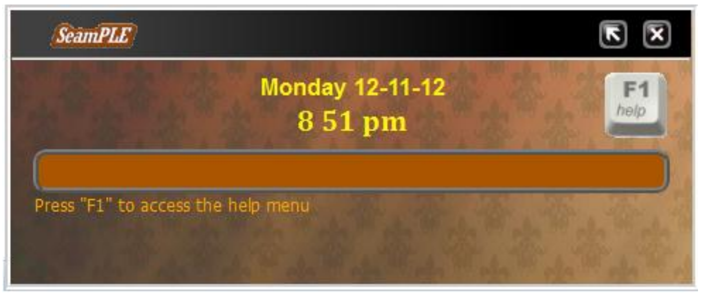
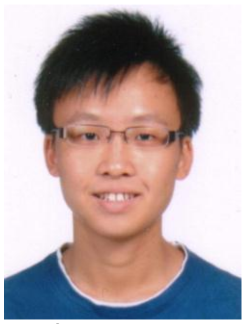
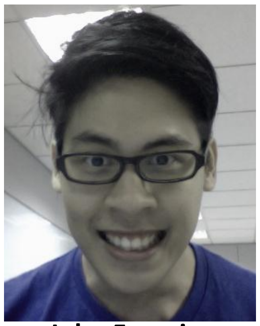
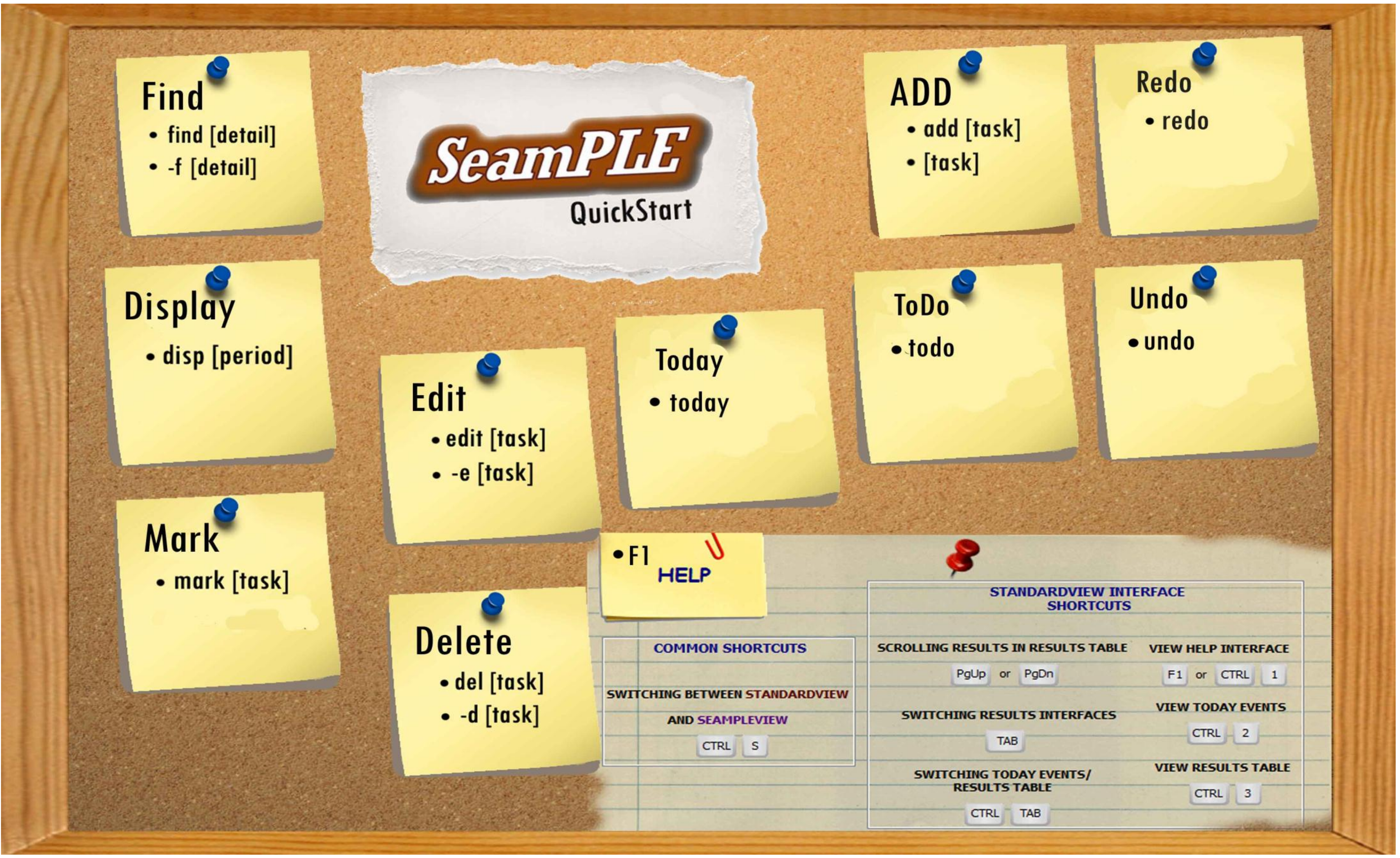
SeamPLE



User and Programmer Manual

V 0.5

Liu Weiyuan Programmer, Designer, Content Producer	**Riandy** Content Producer, Programmer, Time-Keeper	 **Cham Wen Bin** Leader, Designer, Programmer	**Pan Wenren** Programmer, Designer, Content Producer	 **John Francisco** Programmer, Designer, Content Producer



SeamPLE QuickStart

Find
- find [detail]
- -f [detail]

Display
- disp [period]

Mark
- mark [task]

Edit
- edit [task]
- -e [task]

Delete
- del [task]
- -d [task]

Today
- today

ADD
- add [task]
- [task]

Redo
- redo

ToDo
- todo

Undo
- undo

- F1 HELP

COMMON SHORTCUTS

SWITCHING BETWEEN STANDARDVIEW AND SEAMPLEVIEW
CTRL S

STANDARDVIEW INTERFACE SHORTCUTS

SCROLLING RESULTS IN RESULTS TABLE
PgUp or PgDn

SWITCHING RESULTS INTERFACES
TAB

SWITCHING TODAY EVENTS/ RESULTS TABLE
CTRL TAB

VIEW HELP INTERFACE
F1 or CTRL 1

VIEW TODAY EVENTS
CTRL 2

VIEW RESULTS TABLE
CTRL 3

What is SeamPLE Software?

SeamPLE is a todo list widget that is aimed at allowing users (you!) to have total control over day-to-day events. We help you achieve this through our simplistic yet intuitive user interface, as well as our intelligible widget command list.

Getting SeamPLE to run

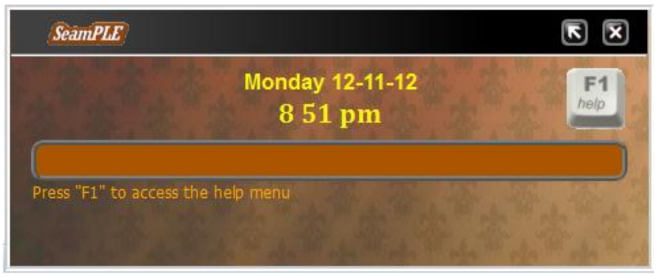
Operating System Requirement: WindowsXP/WindowsVista/Windows7**[RECOMMENDED]**

To run the program, double click SeamPLE.exe icon. (Simple, ain't it?)

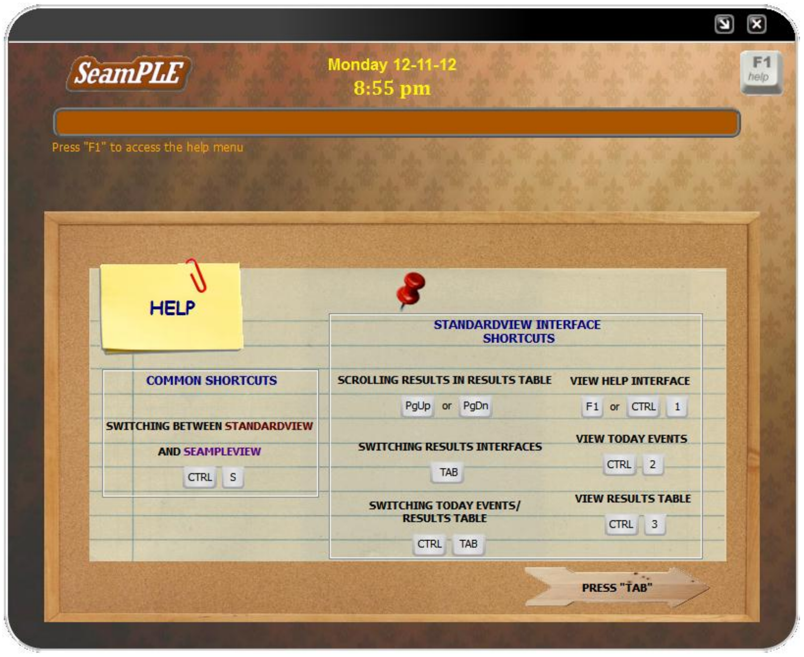
Experiencing SeamPLE

There are two views to our SeamPLE application: **SeamPLE view and Standard view**

SeamPLE view



Standard view


SeamPLE
Monday 12-11-12
8:55 pm
F1 help
Press "F1" to access the help menu
HELP
STANDARDVIEW INTERFACE SHORTCUTS
COMMON SHORTCUTS
SWITCHING BETWEEN STANDARDVIEW AND SEAMPLEVIEW
CTRL S
SCROLLING RESULTS IN RESULTS TABLE
PgUp or PgDn
VIEW HELP INTERFACE
F1 or CTRL 1
SWITCHING RESULTS INTERFACES
TAB
VIEW TODAY EVENTS
CTRL 2
SWITCHING TODAY EVENTS/ RESULTS TABLE
CTRL TAB
VIEW RESULTS TABLE
CTRL 3
PRESS "TAB"

SeamPLE view is seen most regularly as a toolbar notification while **Standard view** is seen only when the program is maximized (using the CTRL – S shortcut).

Using the **Standard view** offers the user more customization options, such as having the ability to check through the day's events and tasks or even searching for an event that is keyed into **SeamPLE** previously. The **SeamPLE view** offers the user an alternate and more convenient way of using the program, by allowing the user to issue commands quickly without the need for maximizing the widget view.

To view the different commands that can be used with **SeamPLE**, please flip to the next page.

Command Walkthrough

ADD

Description: Use this command to add a specific task that you want.
Just type the detail that you want and SeamPLE will help you to manage them!

```
Command : add [task] [date] [category] [time] [priority]

        : [task] [date] [category] [time] [priority]
```

Ex:
```
add dinner With mary       → Add task into today's list of things to do.
Meet investors Monday      → Add task into the upcoming Moday.
```

Edit

Description: Use this command to edit any details of a specific task.
Just type the details that you want to edit and SeamPLE will update accordingly!
Navigate across fields that you wish to edit with the **`tab`** button.

```
Command : edit [task] [date] [category] [time] [priority]

        : -e   [task] [date] [category] [time] [priority]
```

Ex:
```
Edit meeting HIGH         → Edit task's Priority  to be high .
-e homework Monday        → Edit task's Date into the upcoming Moday.
```

Delete

Description: Use this command to delete any specific task.
Just type the task with the ID that you want to delete and SeamPLE will clear them!

```
Command : del [task ID] [eventName] [all]

        : -d  [task ID] [eventName] [all]
```

Ex:
```
del Tennis with Jamie     → Delete Task  "Tennis with Jamie" .
-d 34          → Delete Task "Walk dog". (Task embedded with ID 34 is deleted)
```

Find

Description: Use this command to find a specific task that you want or even list out several task with the same category. Just type the detail that you want and SeamPLE will help you list all of them!

```
Command : find [category] Enter , find [event name] Enter ,

          find [date] Enter

        : -f [category] Enter , -f [event name] Enter , -f [event ID] Enter
```

Ex:	`Find meeting`	→ List all tasks that contains the word "meeting"
	`Find 101`	→ Show the task "Tennis with Jamie".
	`Find 12122012`	→ List all task that starts at this date.

Display

Description: Use this command to display all tasks

```
Command : dsp Enter

        : display Enter
```

Ex: `dsp` → Show all tasks that are saved in text file

Mark

Description: Use this command to mark/unmark your events/tasks. You can mark the tasks that you have done and put it in archive and also unmark an event if you accidentally mark it done.

```
Command : mark [task] Enter
```

Ex: `mark meeting with John` → Mark "meeting with John" event done

Note : command in red is optional. But you have to provide at least one of it to guarantee that the command you want is executed.

Undo

Description: To revert the latest change made by the user. To allow the user to reverse the command of the also cancel or reverse the last command added

```
Command : undo Enter
```

Redo

Description: To reverse an undo operation

```
Command : redo Enter
```

Todo

Description: List all floating task in the database

```
Command : todo Enter
```

Today

Description: List all Today's task in the database

```
Command : today Enter
```

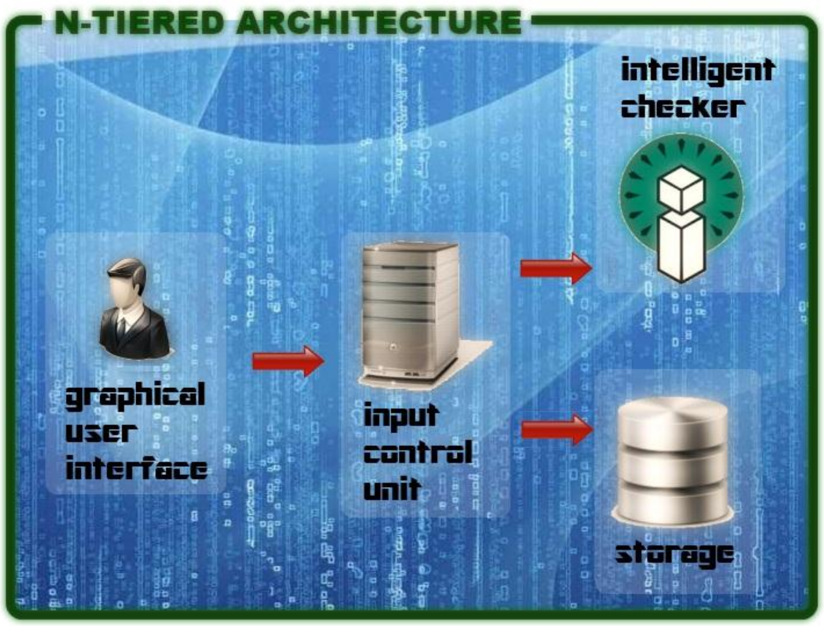
Display Archive

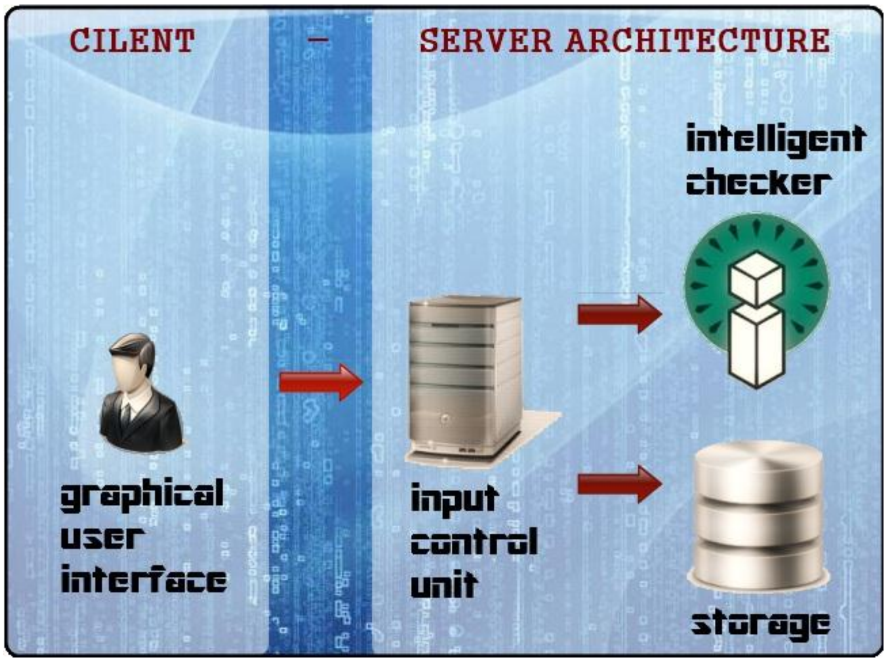
Description: List all archived past events

```
Command : displayarchive Enter
```

Architectural Style

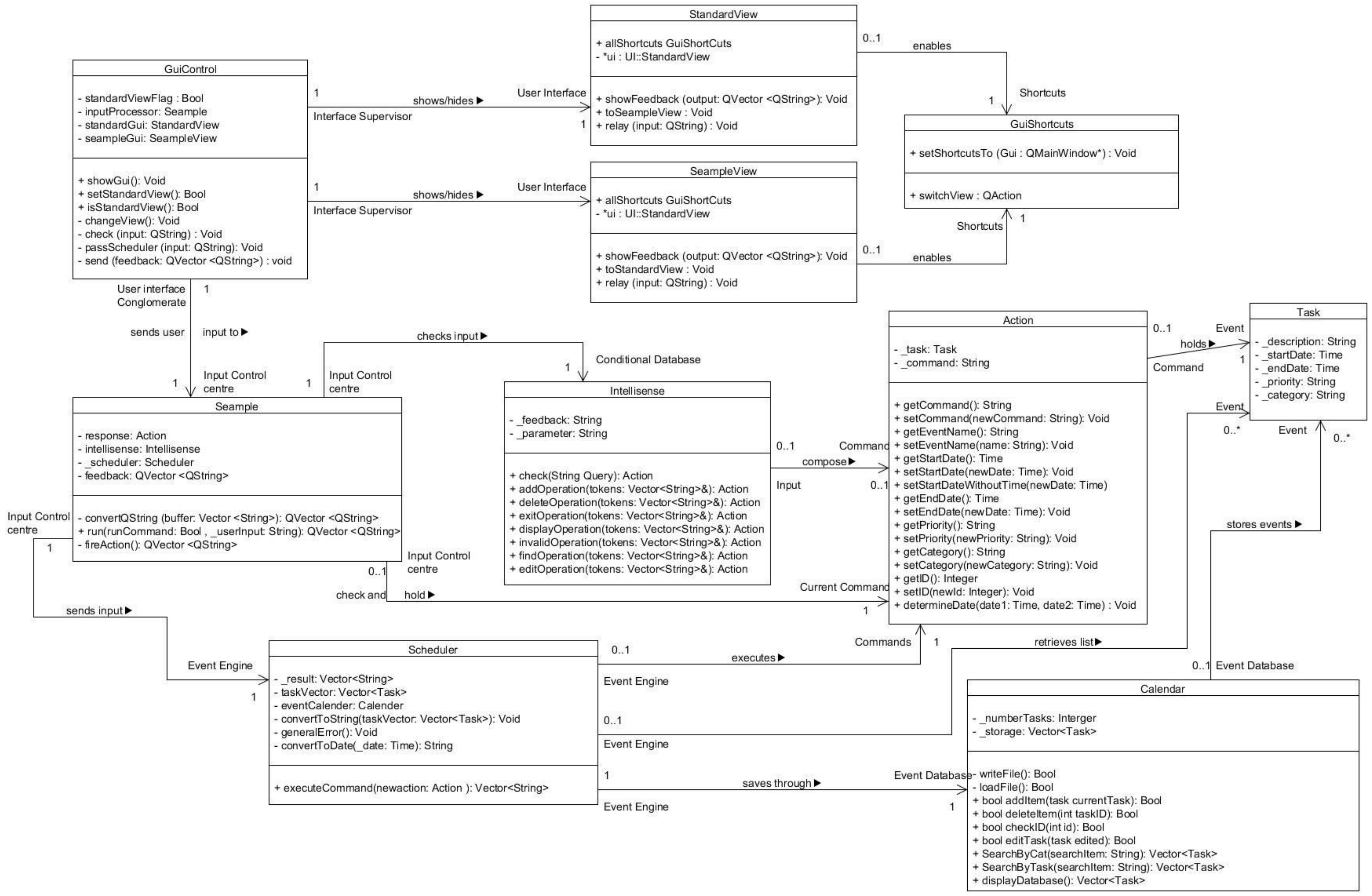
SeamPLE utilizes the N-Tiered Architectural style to power its operations. Starting with the highest level, users interact with various **Graphical User Interfaces (GUI)** that have been prepared specially for them. This input is sent to the **Input Control Unit (ICU)** which takes control of this input and decides whether it should be sent to the **Intelligent Checker (IC)** for validating the input or the **storage** for running the user inputs and storing or retrieving the appropriate details.





SeamPLE has been cleverly engineered to suit developers' needs and purposes. By adopting the approach of attaining minimum coupling between the **GUI** and the **ICU**, we allow the **GUI** and other components of the products to be easily exchangeable with other developer created classes. The "Client" Component, as represented by the **GUI**, can be easily exchanged with other developer created **GUI** or even the command prompt. The "Server" component, as represented by the rest of our system, can be replaced by developer classes to process any possible user inputs.

Next, we'll be looking at the overall class diagram of **SeamPLE**. The class diagram is an essential tool for any prospective developers to continue development or to create new innovations from out system.


StandardView
+ allShortcuts GuiShortCuts
- *ui : UI::StandardView
+ showFeedback (output: QVector <QString>): Void
+ toSeampleView : Void
+ relay (input: QString) : Void
0..1
enables
Shortcuts
1
GuiShortcuts
+ setShortcutsTo (Gui : QMainWindow*) : Void
+ switchView : QAction
GuiControl
- standardViewFlag : Bool
- inputProcessor: Seample
- standardGui: StandardView
- seampleGui: SeampleView
+ showGui(): Void
+ setStandardView(): Bool
+ isStandardView(): Bool
- changeView(): Void
- check (input: QString) : Void
- passScheduler (input: QString): Void
- send (feedback: QVector <QString>) : void
1
shows/hides ▶
User Interface
Interface Supervisor
1
SeampleView
+ allShortcuts GuiShortCuts
- *ui : UI::StandardView
+ showFeedback (output: QVector <QString>): Void
+ toStandardView : Void
+ relay (input: QString) : Void
0..1
enables
Shortcuts
1
User interface
Conglomerate
1
sends user
input to ▶
Input Control
centre
1
checks input ▶
Conditional Database
1
Intellisense
- _feedback: String
- _parameter: String
+ check(String Query): Action
+ addOperation(tokens: Vector<String>&): Action
+ deleteOperation(tokens: Vector<String>&): Action
+ exitOperation(tokens: Vector<String>&): Action
+ displayOperation(tokens: Vector<String>&): Action
+ invalidOperation(tokens: Vector<String>&): Action
+ findOperation(tokens: Vector<String>&): Action
+ editOperation(tokens: Vector<String>&): Action
Seample
- response: Action
- intellisense: Intellisense
- _scheduler: Scheduler
- feedback: QVector <QString>
- convertQString (buffer: Vector <String>): QVector <QString>
+ run(runCommand: Bool , _userInput: String): QVector <QString>
- fireAction(): QVector <QString>
Input Control
centre
1
0..1
Input Control
centre
check and
hold ▶
Current Command
1
0..1
Command
compose ▶
Input
0..1
Action
- _task: Task
- _command: String
+ getCommand(): String
+ setCommand(newCommand: String): Void
+ getEventName(): String
+ setEventName(name: String): Void
+ getStartDate(): Time
+ setStartDate(newDate: Time): Void
+ setStartDateWithoutTime(newDate: Time)
+ getEndDate(): Time
+ setEndDate(newDate: Time): Void
+ getPriority(): String
+ setPriority(newPriority: String): Void
+ getCategory(): String
+ setCategory(newCategory: String): Void
+ getID(): Integer
+ setID(newId: Integer): Void
+ determineDate(date1: Time, date2: Time) : Void
0..1
Event
holds ▶
1
Command
Task
- _description: String
- _startDate: Time
- _endDate: Time
- _priority: String
- _category: String
Event
0..*
Event
0..*
stores events ▶
retrieves list ▶
sends input ▶
Event Engine
1
Scheduler
- _result: Vector<String>
- taskVector: Vector<Task>
- eventCalender: Calender
- convertToString(taskVector: Vector<Task>): Void
- generalError(): Void
- convertToDate(_date: Time): String
+ executeCommand(newaction: Action): Vector<String>
0..1
executes ▶
Commands
1
Event Engine
0..1
Event Engine
1
saves through ▶
Event Database
1
Event Engine
0..1
Event Database
Calendar
- _numberTasks: Interger
- _storage: Vector<Task>
- writeFile(): Bool
- loadFile(): Bool
+ bool addItem(task currentTask): Bool
+ bool deleteItem(int taskID): Bool
+ bool checkID(int id): Bool
+ bool editTask(task edited): Bool
+ SearchByCat(searchItem: String): Vector<Task>
+ SearchByTask(searchItem: String): Vector<Task>
+ displayDatabase(): Vector<Task>

Through the class diagram of SeamPLE, the different relations between the various components are explored. It is important to note that GuiControl must always contain exactly one instance of Seample and Seample must be owned by one instance of GuiControl. This supports the Client – Server architecture that was mentioned earlier on, where one unique client can have only one relation with another unique server.

API Descriptions

Next, we look at possible API descriptions for our program. The API descriptions support what we have implemented in the class diagram above. These descriptions can be used for further implementation (extending on current purposes) and testing (for writing stubs that emulates component behavior)

GUI CONTROL class

send

Description

Function to display feedback to users through the GUI. Call this function when a component wants to interact with the user. API will display all elements accordingly.

Syntax

void send(QVector <QString> feedback)

Input value(s): Qvector of Qstrings containing feedback strings

Return values (s) : None. Output displayed by GUI

passScheduler

Description

Event trigger function when “enter” key is pressed. Function will call scheduler to execute user’s command.

Syntax

void passScheduler(QString input)

Input value(s): Qstrings containing user command input

Return values (s) : None. Output displayed by GUI

check

Description

Calls for intellisense.check to parse user command input.

Syntax

void check(QString input)

Input value(s): Qstrings containing user command input

Return values (s) : None.

INTELLISENSE class

check

Description

Parses user’s command input in the textfield into Action files

Syntax

Action check(string query)

Input value(s): String containing user command input

Return values (s) : None.

getParameter

Description

Determine the input requirements from user command input and generates a feedback string to inform the user of the required fields.

Syntax

string getParameter()

Input value(s): None

Return values (s) : String of colour coded feedback indications

SCHEDULER class

executeCommand

Description

Performs the relevant actions based on the type of operation, input fields.

Syntax

vector<string> executeCommand(Action newaction)

Input value(s): Action class containing input field information

Return values (s) : Vector of string containing feedback messages depending on operation.

CALENDER class

addItem

Description

Stores a user task into the database.

Syntax

bool addItem(task currentTask)

Input value(s): task class containing input field information

Return values (s) : Boolean true if operation is successful false otherwise.

deleteItem

Description

Remove a user task from the database .

Syntax

bool deleteItem(int taskID) or bool deleteItem(string eventName) or bool delete all()

Input value(s): task class containing input field information

Return values (s) : Boolean true if operation is successful false otherwise.

editTask

Description

Edit a user task in the database .

Syntax

bool editTask(task edited)

Input value(s): task class containing input field information

Return values (s) : Boolean true if operation is successful false otherwise.

Undo

Description

Undo the last action/command executed (max 3 consecutive undo) .

Syntax

bool undoAction()

Input value(s): -

Return values (s) : Boolean true if operation is successful false otherwise.

Redo

Description

Redo the last action/command executed (max 3 consecutive redo) .

Syntax

bool redoAction()

Input value(s): -

Return values (s) : Boolean true if operation is successful false otherwise.

MarkTask

Description

Mark the task specified and archive it

Syntax

bool markTask(task markedTask)

Input value(s): task to be marked

Return values (s) : Boolean true if operation is successful false otherwise.

getFloatingEvents

Description

retrieve all user floating task (dateType=4) tasks in the database .

Syntax

vector<task> getFloatingEvents()

Input value(s): None

Return values (s) : Vector<task> containing all floating task information

getToday

Description

Retrieve all today's task in the database

Syntax

vector<task> getToday()

Input value(s): None

Return values (s) : Vector<task> containing all today task information

displayDatabase

Description

Display all user tasks in the database .

Syntax

vector<task> displayDatabase()

Input value(s): None

Return values (s) : Vector containing all task information

displayArchive Events

Description

Display all user tasks in the archive database.

Syntax

vector<task> displayArchiveEvent()

Input value(s): None

Return values (s) : Vector containing all task information

SearchByTask

Description

Search all user tasks in the database based on task ID.

Syntax

vector<task> SearchByTask(string searchItem)

Input value(s): string containing search field

Return values (s) : Vector containing all task information

SearchByCat

Description

Search all user tasks in the database based on task category.

Syntax

vector<task> SearchByCat(string searchItem)

Input value(s): string containing search field

Return values (s) : Vector containing all task information

SearchByDate

Description

Search all user tasks in the database based on date specified.

Syntax

vector<task> SearchByDate(string searchDate)

Input value(s): string containing search field

Return values (s) : Vector containing all task information.

In the creation of our product, the production team has identified a list of patterns and principles that can be used to enhance the quality of the code as well as to assist in the problem solving. Identical patterns that were observed were applied to the code. These patterns are described below.

SINGLETON PATTERN

Certain classes within our program can utilize this pattern as these components *(can be global components such as GUI)* should carry or control consistent values. Several of the modified classes can also contain methods and objects that can or should be shared across other classes that require these classes.

The issue here is to remove and also to prevent numerous instances of any of the prospective **SINGLETON** classes to be created. Before achieving this, we have to identify the prospective **SINGLETON** classes:

- **GuiControl**
    - The **SINGLETON** pattern is applied as GuiControl is the control class for all user interfaces programmed into the application. Multiple instances of GuiControl will lead to multiple points of control for the user interfaces. This will lead to logic errors as each instance of GuiControl can contain different values for objects (such as flags) within the class. When displaying to the user interfaces from different instances of GuiControl simultaneously, an illogical sequence of messages may be shown.

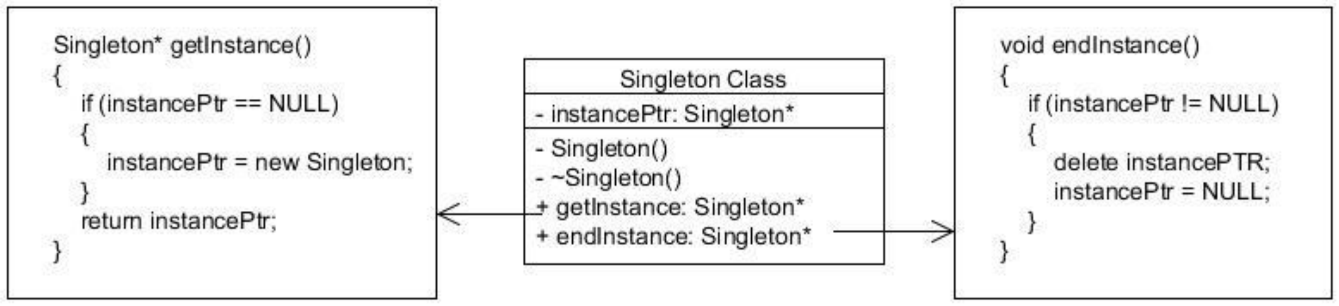
- **Gui interfaces – StandardView and SeampleView**
    - User interfaces used should be similar across the entire application. Multiple instances of each user interface will lead to several interfaces being created at one point of time.

- **Seample (control class)**
    - The control class is the centre of operations for the entire application. Similar to GuiControl, there shouldn't be multiple instances of of Seample since objects within Seample will differ across different instances, leading to widely different values being computed as a result.

- **Intellisense**
  - Intellisense is used to check if the command entered by the user is appropriate for execution. While multiple instances of Intellisense does not lead to drastically different results that affects application performance, it is wise to apply the SINGLETON pattern as the methods can be shared across different classes that owned Intellisense.

- **Scheduler**
  - Commands that are entered through user interfaces and authorized by Intellisense are sent into Scheduler for executing. Similar to Intellisense, multiple instances of Scheduler will not affect performance but it is best to apply the SINGLETON pattern as the methods and objects can be shared across different classes that own Scheduler.

From the above, different problems from classes that can be solved via applying the SINGLETON pattern are identified. Hence, a common solution affixed to the SINGLETON pattern can be applied across these affected classes



```
Singleton* getInstance()
{
    if (instancePtr == NULL)
    {
        instancePtr = new Singleton;
    }
    return instancePtr;
}
```

Singleton Class
- instancePtr: Singleton*
- Singleton()
- ~Singleton()
+ getInstance: Singleton*
+ endInstance: Singleton*

```
void endInstance()
{
    if (instancePtr != NULL)
    {
        delete instancePTR;
        instancePtr = NULL;
    }
}
```

The solution is as seen above. Two methods, getInstance and endInstance, are created for each Singleton class. instancePtr is initially NULL. Hence, when getInstance is called for the first time, the Singleton class is dynamically created and the address is sent to the calling object. Subsequently calls to getInstance will return the same address that instancePtr is pointing to. Similarly, when deleting the dynamically allocated memory through endInstance, it will only occur when instancePtr is not NULL. instancePtr is only assigned the NULL value after it is deleted for the first time, hence subsequent deletes will be nullified.

Note: Alternative solutions include the passing of objects' addresses across classes that own any of the SINGLETON classes. However, this solution is inferior as it is harder to handle the dynamically created memory, where issues such as multiple delete leading to deleting at NULL may occur.

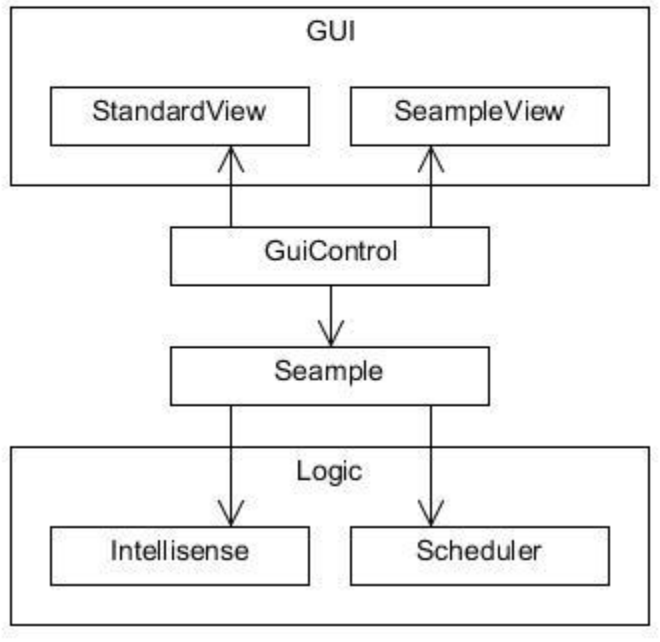
FACADE PATTERN

For certain classes, accessing other classes is important to obtain information that is crucial for their operations. However, it is not advisable for every class in the application to be able to access every other class, even classes that it does not require. It is also better if certain classes operate without knowledge of the intrinsic details of other classes, thus promoting the act of low coupling, which lowers the dependency of components between each other.

Again, we need to identify classes that serve well as the **FACADE**. The two classes identified are:

- **GuiControl**
  - GuiControl needs to serve as a buffer for the user interfaces. All information sent from the logic components of the application should not access the attributes of the interfaces and display the information directly.
- **Seample**
  - Similar to GuiControl, all information sent from the user interfaces should not be sent directly to the individual logic components for processing and executing.

Since the similar issue is observed for the two classes identified above, the same solution can definitely be applied to resolve this recurring issue.

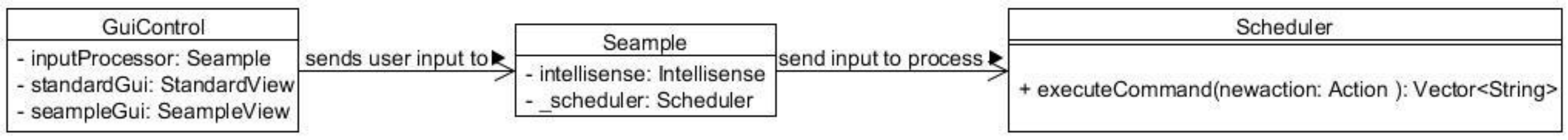


As seen on the left, there is a simplified class diagram on the solution for implementing the **FACADE** pattern. GuiControl and Seample are the facades for each of their own components. While Seample is the control class for the entire program, it can serve as a **FACADE** in this scenario as it prevents the user interface elements from accessing the intrinsic details within the logic component. Similarly, by allowing Seample to access interface elements through GuiControl, GuiControl can also be a **FACADE** class.

While applying patterns to solve recurring problems is essential, crucial principles should be applied to the code so that the quality of the code can be enhanced.

LAW OF DEMETER

The principle for Law of Demeter is achieved through our code by the use of encapsulation of objects within our different classes. Here, we have an example.


GuiControl
- inputProcessor: Seample
- standardGui: StandardView
- seampleGui: SeampleView
sends user input to
Seample
- intellisense: Intellisense
- _scheduler: Scheduler
send input to process
Scheduler
+ executeCommand(newaction: Action): Vector<String>

GuiControl contains an instance of Seample, which in turns holds an instance of Scheduler. According to the law, GuiControl is not allowed to access methods of belonging to objects of Seample. The **LAW OF DEMETER** is not violated as Scheduler is a private object of Seample, hence GuiControl is unable to access its methods.

By conforming to what was observed above and writing code using the above logic, the **LAW OF DEMETER** is observed.

SINGLE RESPONSIBILITY PRINCIPLE

Each class has a single responsibility that it has to uphold *(which is also most likely as described by their names, with the exception of Seample).* To ensure this, the principle of **SEPARATION OF CONCERNS** should be observed as far as possible. This is so that concerns can be correctly identified and regrouped into classes, each with their own responsibility. From this concept, the different classes that are pieced up to meet this principle are classes such as the Intellisense. Different functions that are used to determine if a command entered by the user is authorized are created, each with a sole purpose of just performing one task (An example would be the tokenize function whose sole purpose is to separate a sentence to individual words).

The following contains a short write-up on the roles and responsibility of the classes in our application:

GuiControl – Control which user interface is to be displayed and the inputs and outputs to and from the appropriate interfaces
SeampleView and StandardView – User interfaces that take in input and shows output to users
Seample – Control class that redirects input from user interfaces into logic components and output from logic components to user interfaces.
Intellisense – Takes in user input and checks if the command inputted is an acceptable command

Scheduler – Execute commands inputted by user

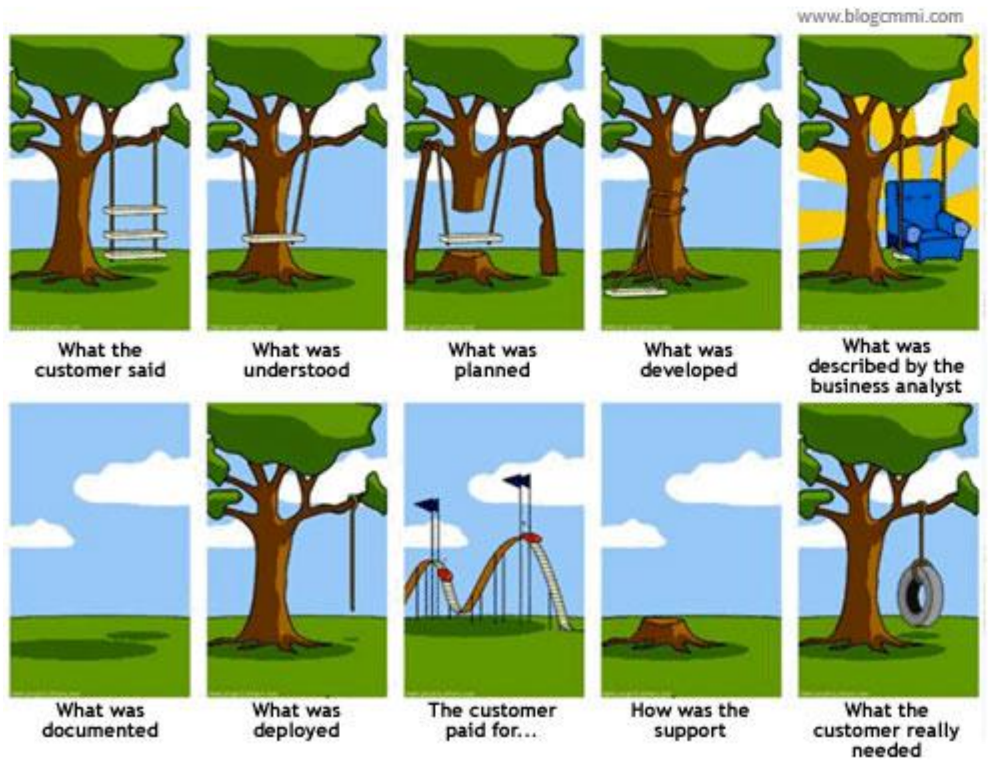
The later you find a bug, the more costly it is to fix

CONGRATULATIONS! YOU HAVE JUST DISCOVERED A COMPLETELY NEW ERROR!
GOFF

The Funny Times <ft@funnytimes.com> 216/371-8600
PO Box 18530 / Cleveland Heights, OH 44118

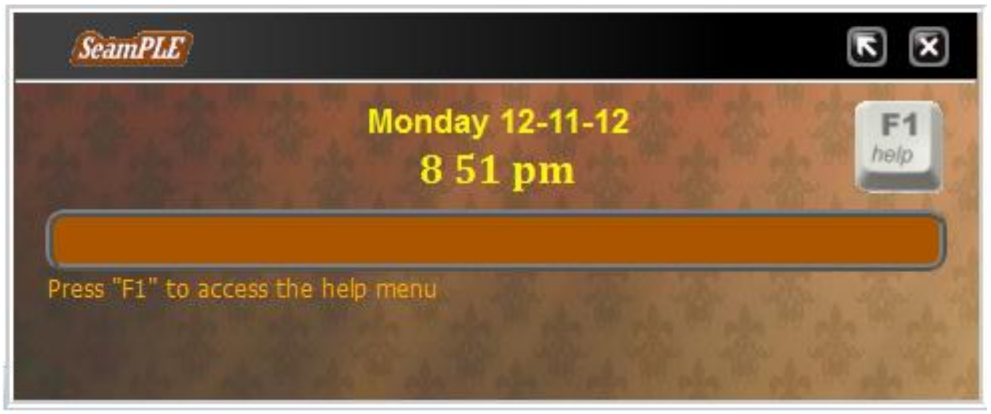
Manual testing is done regularly by each of our members after each implementation. This helps in the sourcing of possible errors and preventing bugs from accumulating and turning into larger errors. Unit testing has also been added by one of our members for the Qt environment that is used for programming the application. While cost may sometimes refer to material costs, it can also refer to the concept of time and effort, where errors left unsolved over time may lead to larger amounts of time and effort required to resolve them.

Communication

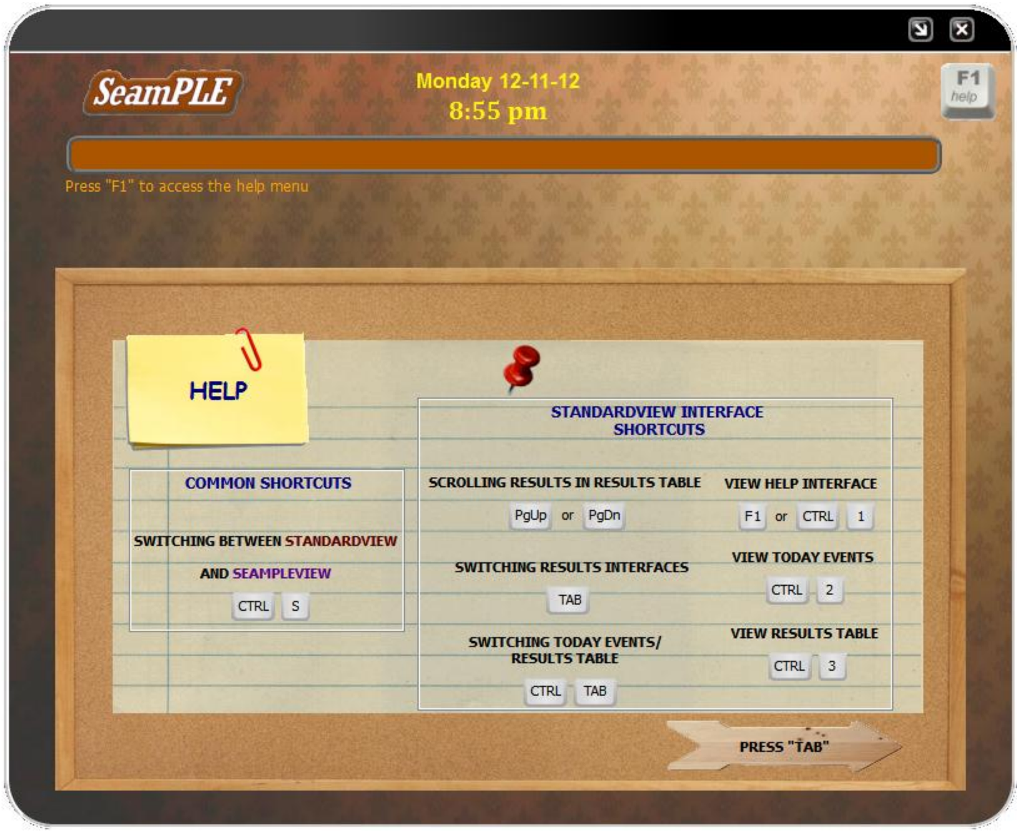


Communication is very important, and not just from prospective consumers who provide us with much needed valuable feedback, but also from each of our teammates as well. While this principle is essentially not focused on the physical code itself, it requires the programmers to focus on another essential component leading to the code, which is the physical demand and requirement for the code (essentially, why we're programming in the first place!). In other words, validation and verification are very much affected by this principle. By enforcing strong communication channels between programmers and those not in the programming team, code that fits its demand and with proper implementation can be created. These communication channels, for example, can exist in the form of meetings that are conducted regularly to see if the aims for the code are met.

Appendix (User Tutorial)



When running our program for the first time, you will be greeted by our friendly **SeamPLE** interface. As seen in the top right hand corner of the window, you can choose to press "F1" to trigger the help menu. You may want to do that, so that you can gain a better idea on what you can achieve with our application.



COMMON SHORTCUTS

SWITCHING BETWEEN STANDARDVIEW AND SEAMPLEVIEW: CTRL S

STANDARDVIEW INTERFACE SHORTCUTS

SCROLLING RESULTS IN RESULTS TABLE: PgUp or PgDn

VIEW HELP INTERFACE: F1 or CTRL 1

SWITCHING RESULTS INTERFACES: TAB

VIEW TODAY EVENTS: CTRL 2

SWITCHING TODAY EVENTS/ RESULTS TABLE: CTRL TAB

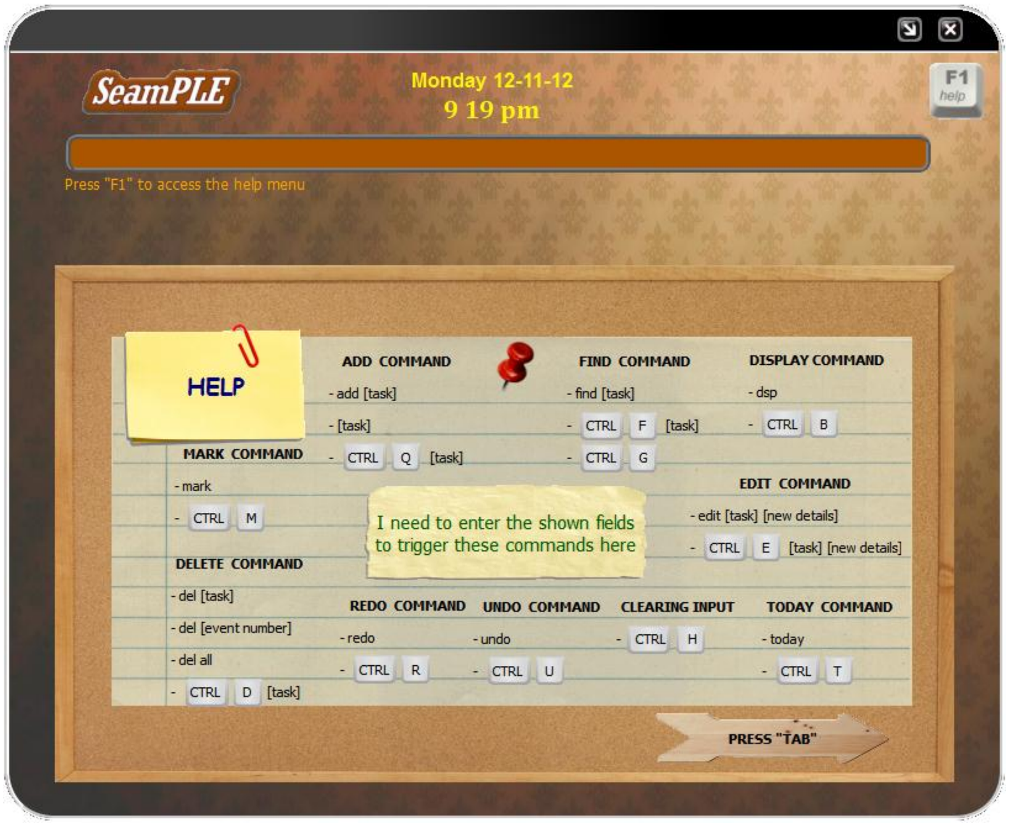
VIEW RESULTS TABLE: CTRL 3

PRESS "TAB"

The first page of the help menu will show shortcuts that you can utilize in the program. For example, pressing "Pg Up" or "Pg Dn" in the results table will allow for traversing through the results.

Remember that we mentioned about the help menu being on the first page? To traverse to other pages, press "Tab" when you are currently at the help menu.

Page 2



SeamPLE

Monday 12-11-12
9 19 pm

F1 help

Press "F1" to access the help menu

HELP

ADD COMMAND
- add [task]
- [task]
- CTRL Q [task]

FIND COMMAND
- find [task]
- CTRL F [task]
- CTRL G

DISPLAY COMMAND
- dsp
- CTRL B

MARK COMMAND
- mark
- CTRL M

EDIT COMMAND
- edit [task] [new details]
- CTRL E [task] [new details]

I need to enter the shown fields to trigger these commands here

DELETE COMMAND
- del [task]
- del [event number]
- del all
- CTRL D [task]

REDO COMMAND
- redo
- CTRL R

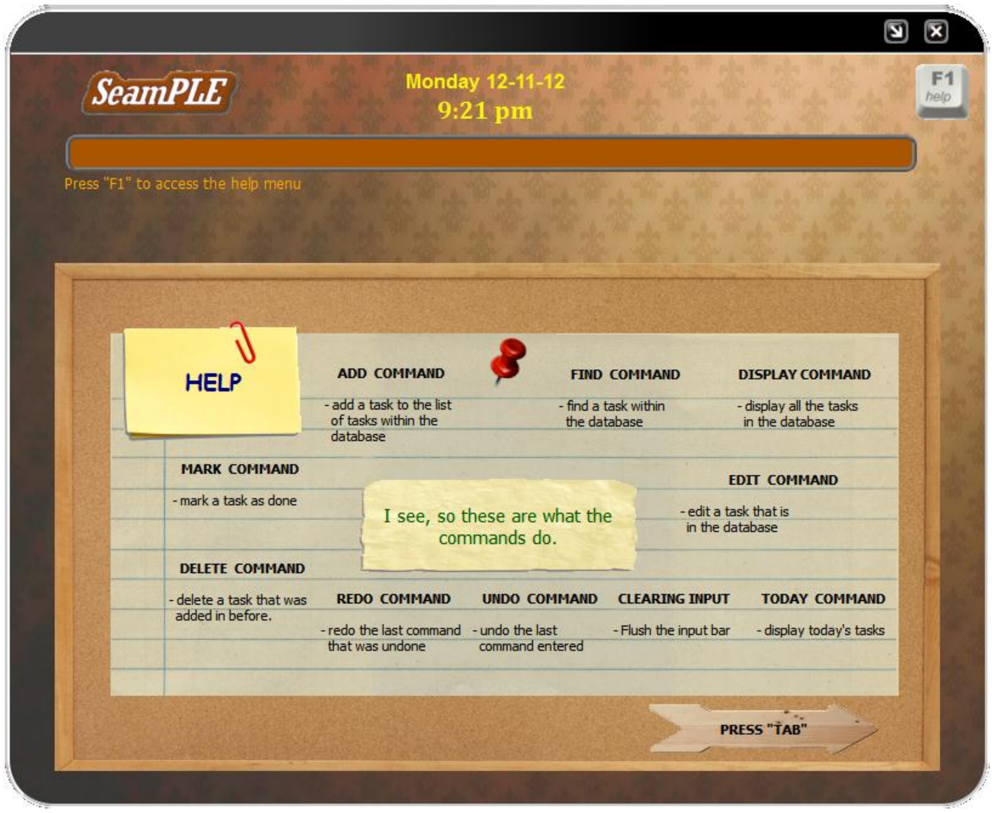
UNDO COMMAND
- undo
- CTRL U

CLEARING INPUT
- CTRL H

TODAY COMMAND
- today
- CTRL T

PRESS "TAB"

Page 3



SeamPLE

Monday 12-11-12
9:21 pm

F1 help

Press "F1" to access the help menu

HELP

ADD COMMAND
- add a task to the list of tasks within the database

FIND COMMAND
- find a task within the database

DISPLAY COMMAND
- display all the tasks in the database

MARK COMMAND
- mark a task as done

EDIT COMMAND
- edit a task that is in the database

I see, so these are what the commands do.

DELETE COMMAND
- delete a task that was added in before.

REDO COMMAND
- redo the last command that was undone

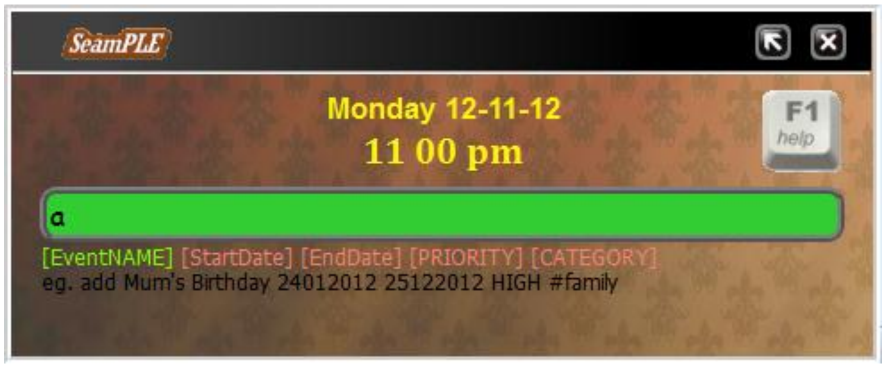
UNDO COMMAND
- undo the last command entered

CLEARING INPUT
- Flush the input bar

TODAY COMMAND
- display today's tasks

PRESS "TAB"

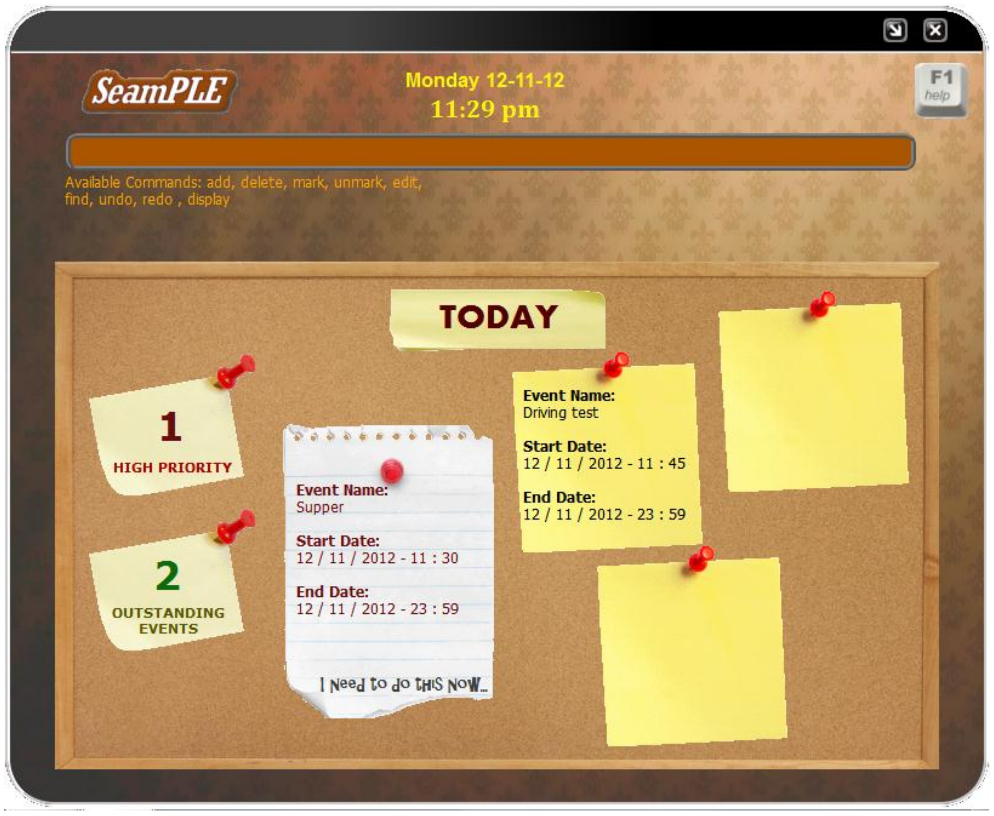
Now that we have gained a bit more understanding on how to use the program, let's move on to customize the scheduler with today's events. First, press CTRL + S to return back to SeamPLE interface. Trying pressing any alphabet keys on your keyboard.



a

[EventNAME] [StartDate] [EndDate] [PRIORITY] [CATEGORY]
eg. add Mum's Birthday 24012012 25122012 HIGH #family

This is our quick-add feature, which allows you to add an event without the use of the add command. In the above, you will see an example shown on how to input an event. Try adding a few events in the same format as the example, using today's date and time later then your computer time. Try to alternate between HIGH priority and setting no priority. Enter CTRL + S (and CTRL + 2 if you are not at today's view) to access **Standard View.** You should be able to see something like this.



Available Commands: add, delete, mark, unmark, edit, find, undo, redo , display

Event Name:
Supper
Start Date:
12 / 11 / 2012 - 11 : 30
End Date:
12 / 11 / 2012 - 23 : 59

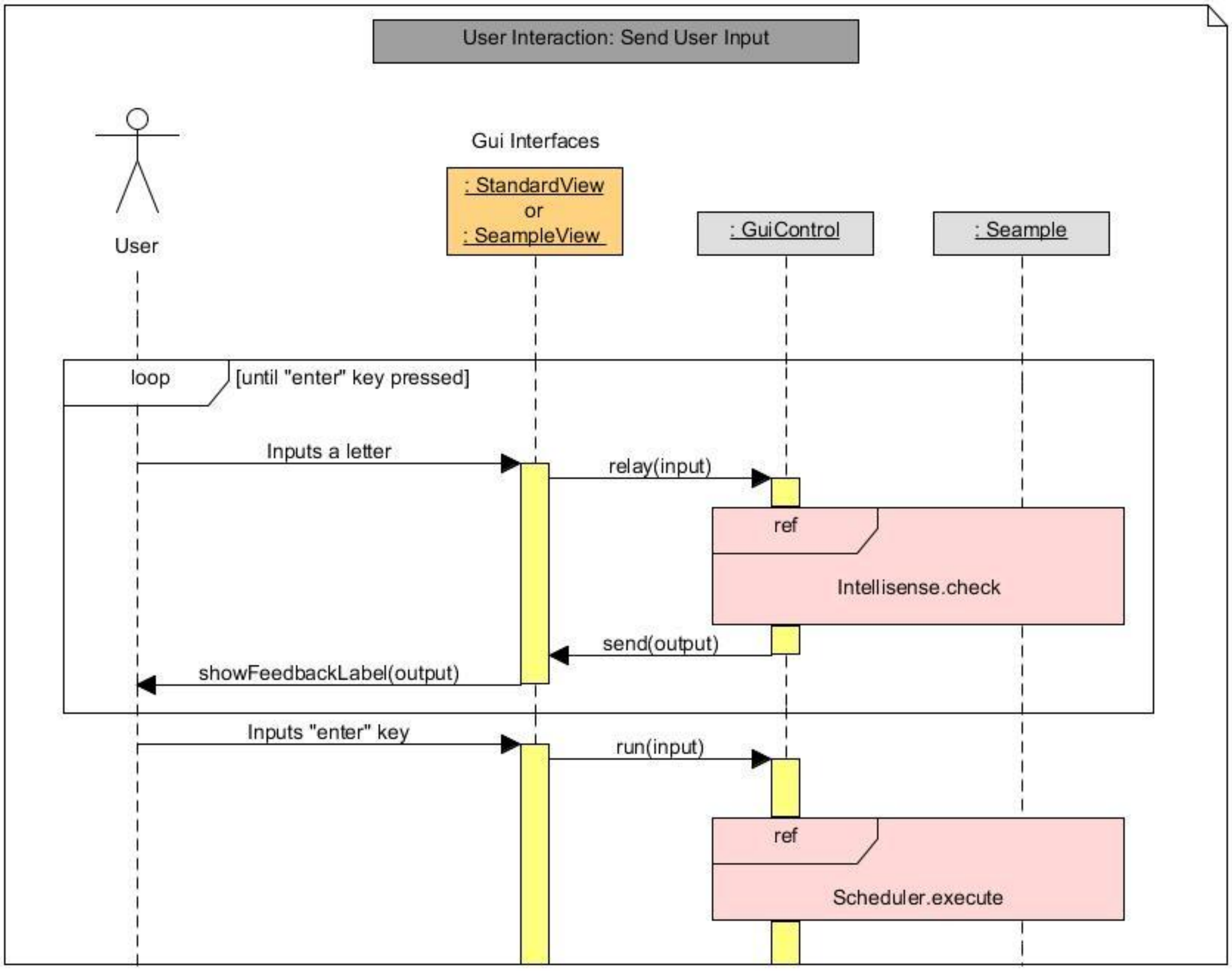
Event Name:
Driving test
Start Date:
12 / 11 / 2012 - 11 : 45
End Date:
12 / 11 / 2012 - 23 : 59

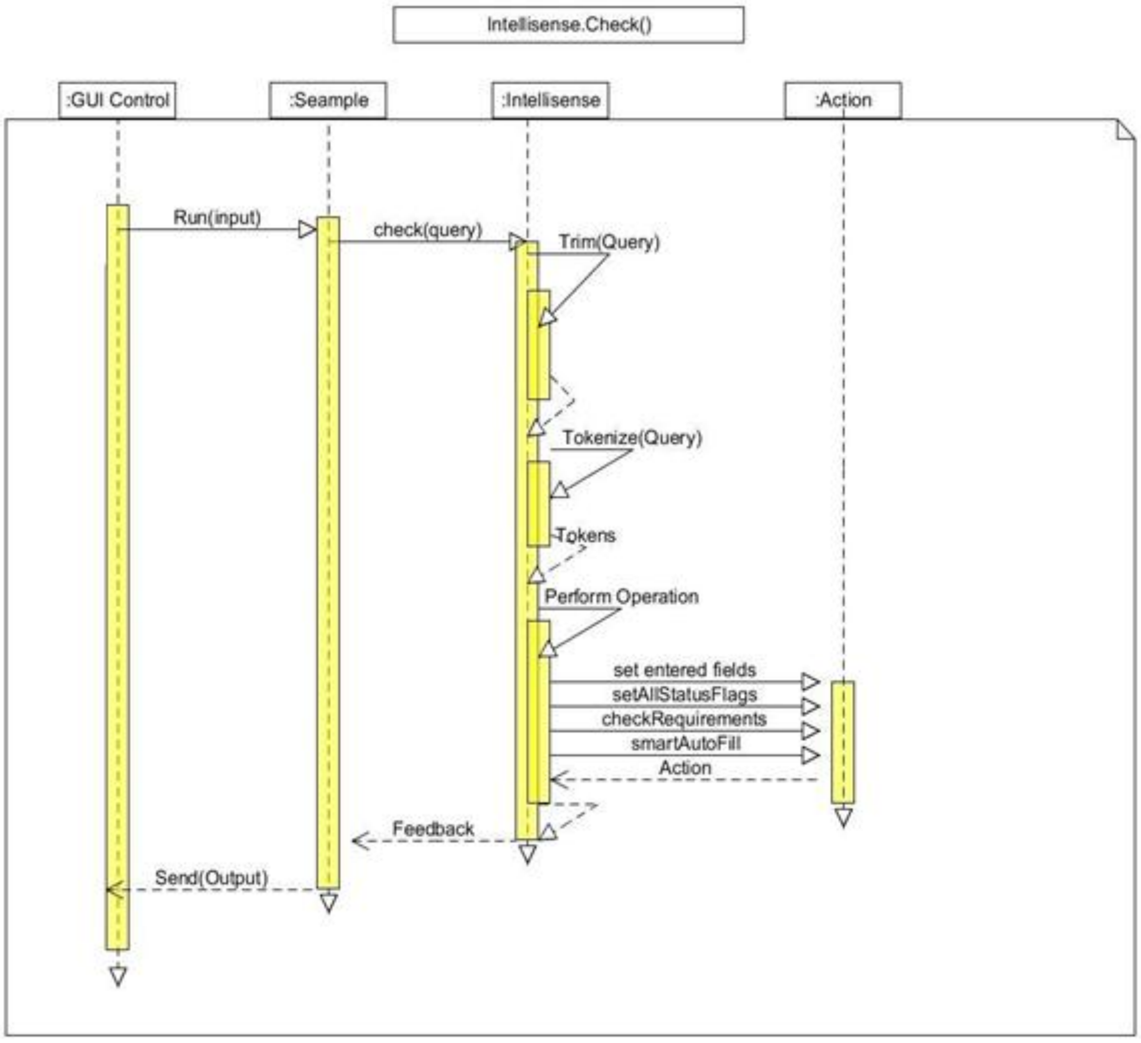
With this, you would have already created a few events to be recorded in the program. You have also successfully explored the features to perform a self-update. Do try to use more features from the help menu, which will allow your experience in using our program to be more enriching.

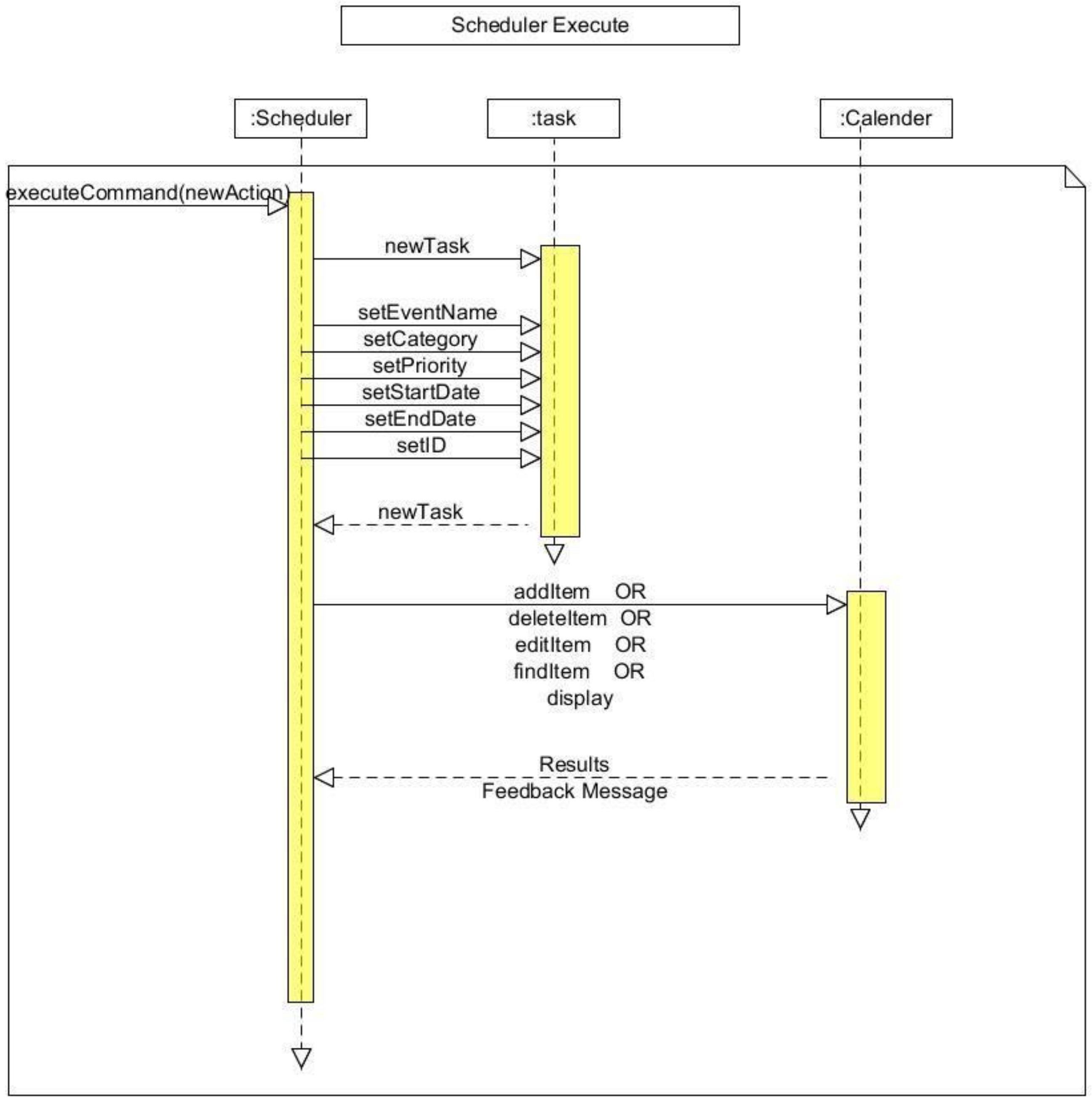
Appendix (Sequence Diagrams to understand how our program runs)

Seample Command Listing

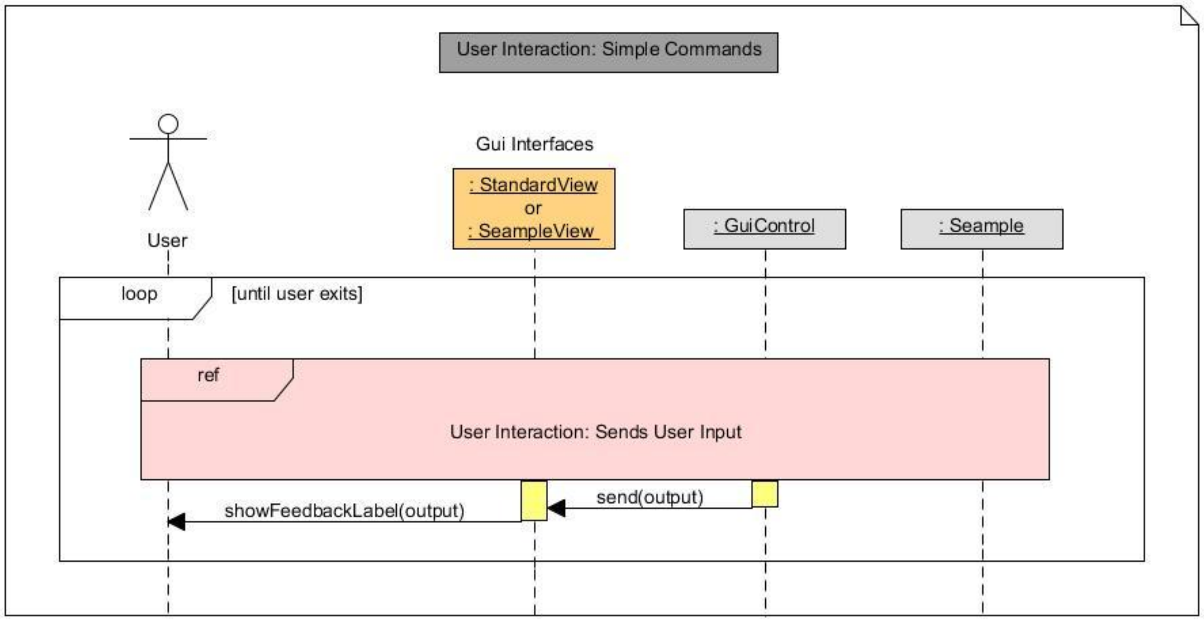
Operation	Standard Command word	Alternative command word
Add	add	"-a", "create" ,"new","++","-a"
Delete	delete	"del","dd","cancel","-d"
Mark	mark	"done","-m"
Display	display	"show","-d","dsp"
Exit	exit	"quit","-q"
Find	find	"search","-f"
Edit	edit	"change","-e","defer","reschedule","change"
Undo	undo	"revert","-u"
Redo	redo	"-r"
Todo	todo	
Today	today	
DisplayArchive	displayarchive	


User Interaction: Send User Input
User
Gui Interfaces
: StandardView or : SeampleView
: GuiControl
: Seample
loop
[until "enter" key pressed]
Inputs a letter
relay(input)
ref
Intellisense.check
send(output)
showFeedbackLabel(output)
Inputs "enter" key
run(input)
ref
Scheduler.execute

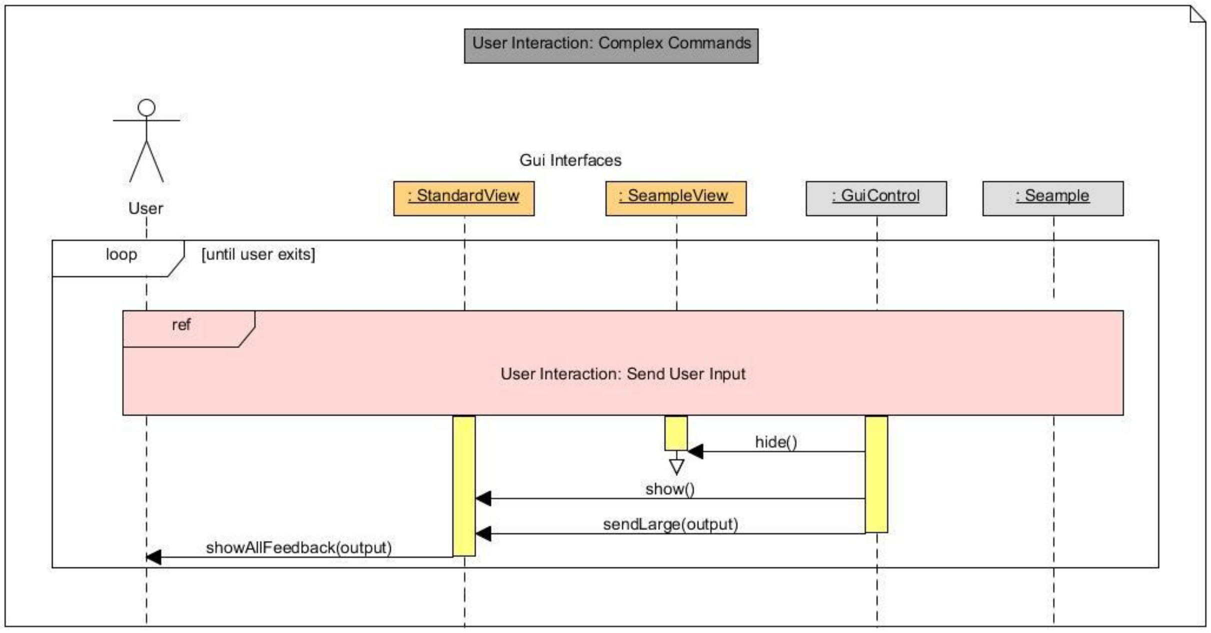

Intellisense.Check()
:GUI Control
:Seample
:Intellisense
:Action
Run(input)
check(query)
Trim(Query)
Tokenize(Query)
Tokens
Perform Operation
set entered fields
setAllStatusFlags
checkRequirements
smartAutoFill
Action
Feedback
Send(Output)

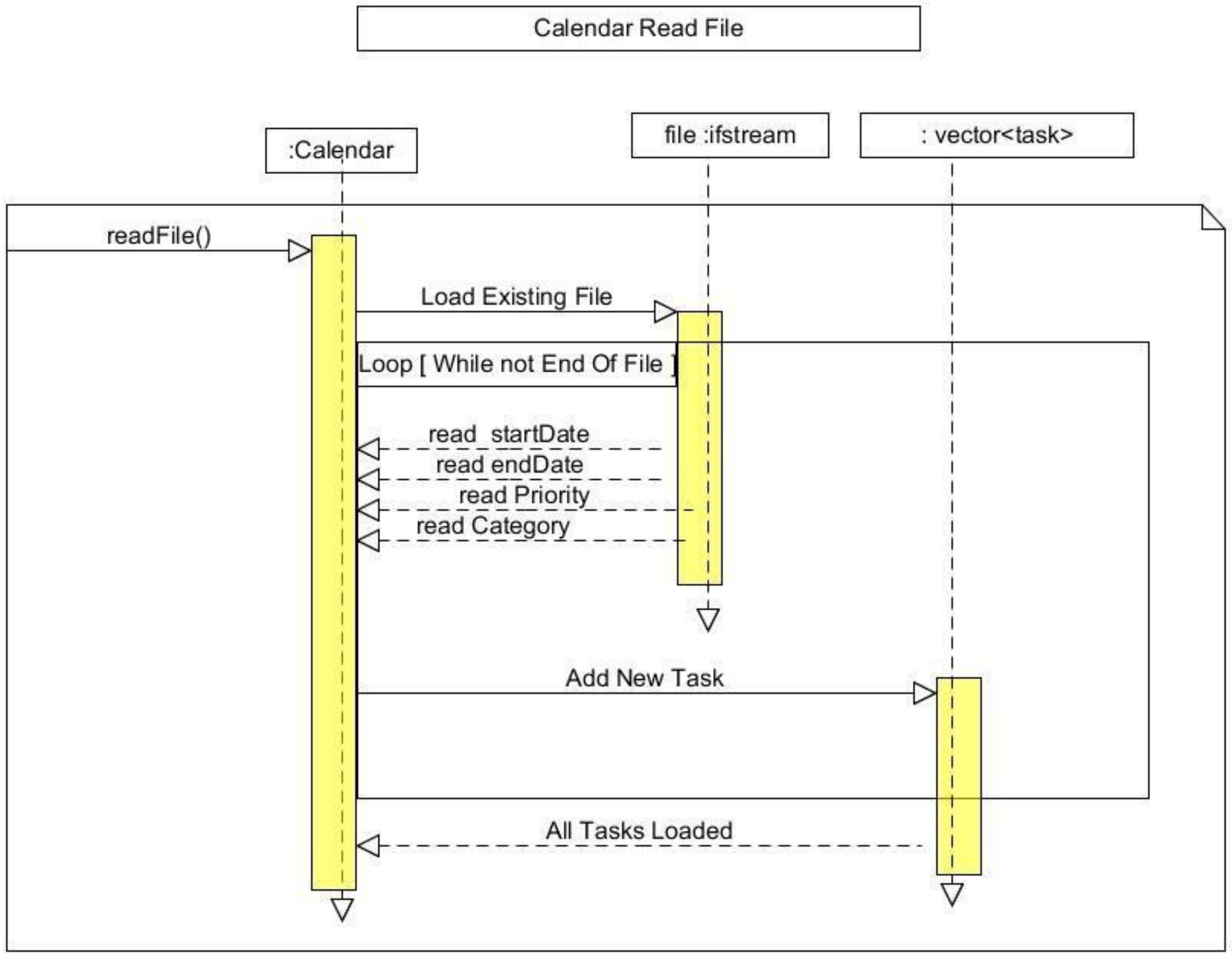

Scheduler Execute
:Scheduler
:task
:Calender
executeCommand(newAction)
newTask
setEventName
setCategory
setPriority
setStartDate
setEndDate
setID
newTask
addItem OR
deleteItem OR
editItem OR
findItem OR
display
Results
Feedback Message

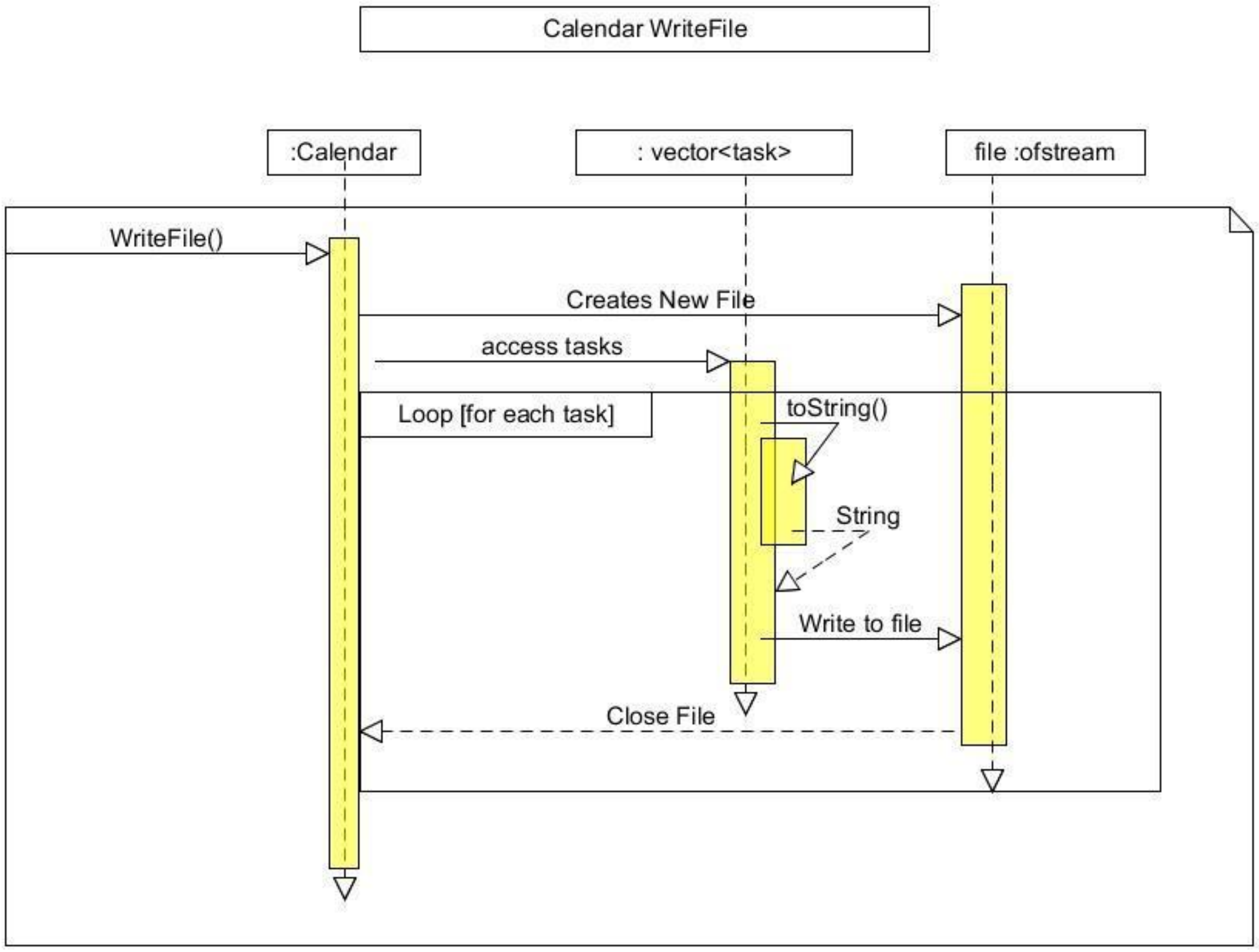
Note: Simple Commands refer to commands such as “add”, “delete” and “edit” where output consist of only one string


User Interaction: Simple Commands
Gui Interfaces
User
: StandardView or : SeampleView
: GuiControl
: Seample
loop
[until user exits]
ref
User Interaction: Sends User Input
showFeedbackLabel(output)
send(output)

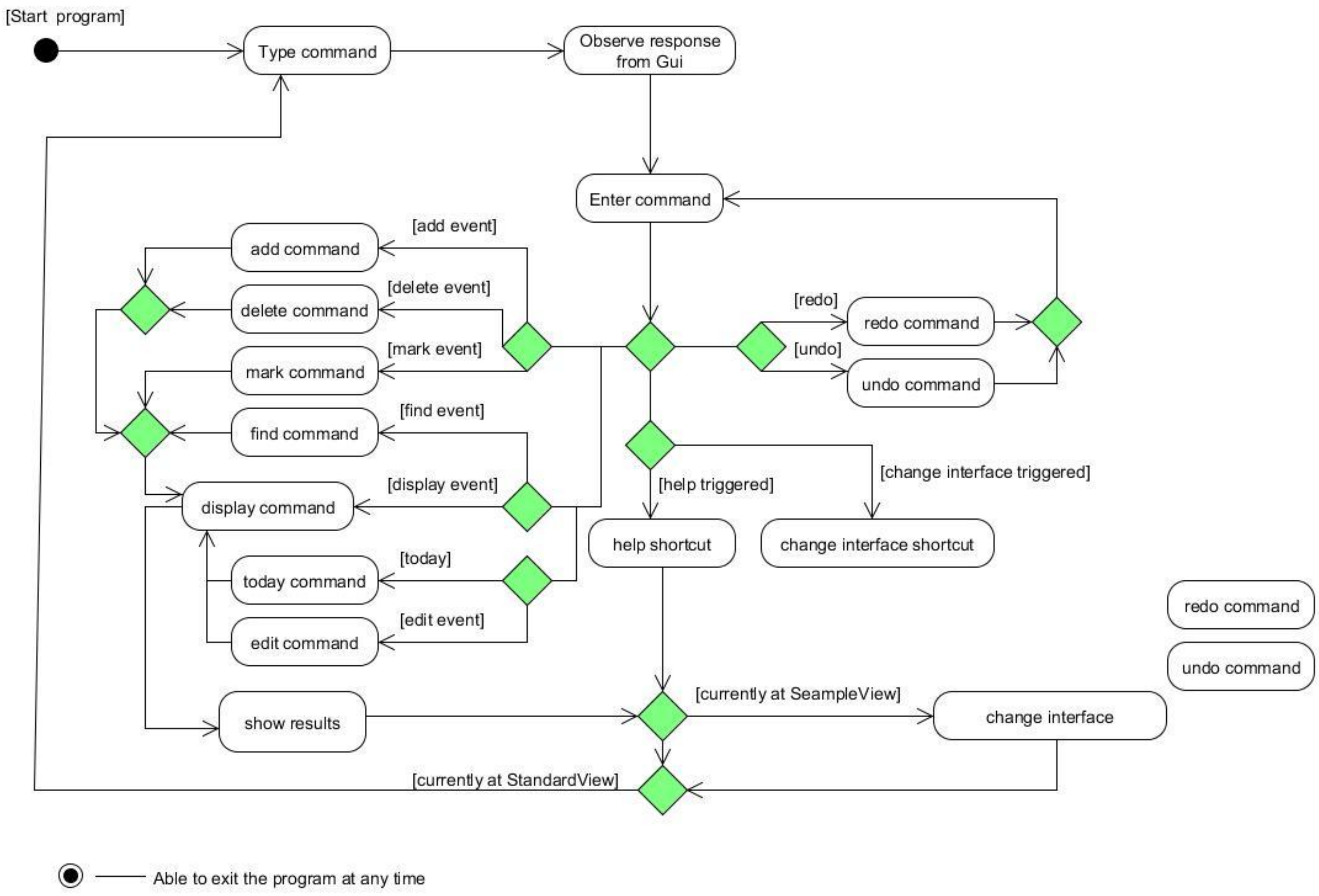
Note: Complex Commands refer to commands such as “find” and “display” where output can consist of more than one string due to multiple results.




Calendar Read File
:Calendar
file :ifstream
: vector<task>
readFile()
Load Existing File
Loop [While not End Of File]
read_startDate
read endDate
read Priority
read Category
Add New Task
All Tasks Loaded


Calendar WriteFile
:Calendar
: vector<task>
file :ofstream
WriteFile()
Creates New File
access tasks
Loop [for each task]
toString()
String
Write to file
Close File

Activity Diagram



The diagram above explains the traverse process for the user as he/she issue commands using the user interfaces