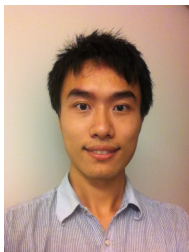


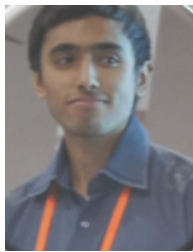


Jasca

 <p>Wu Haoyang Testing & Deliverables</p>	 <p>Narinderpal Singh Dhillon Deadline watcher & Integrator</p>	 <p>Nigel Cheok Jianxing Documentation & Testing</p>	 <p>Mohit Shridhar Team lead, Scheduling, Deliverables & Code quality</p>
---	---	---	---

Developer's Guide

This guide is for developers who intend to use TASCA's tools exclusively for their own development purposes and/or to contribute to the maintenance and improvement of this project. The following information should not be used as a definitive guide to access and implement TASCA's resources, as it presents a mere high-level overview of the system and its components.

Architecture:

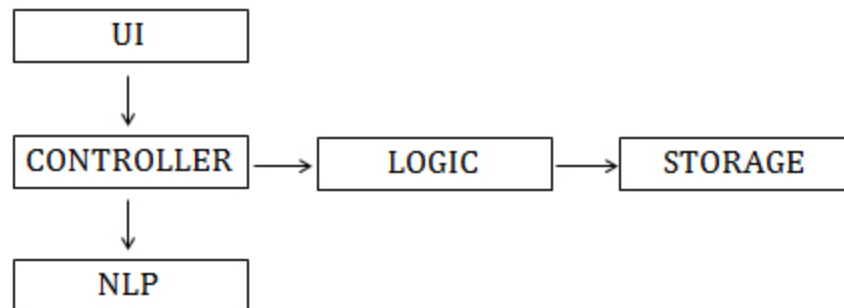


Figure 1: *Architecture Diagram for TASCA*

The user interacts with the UI class, UI passes this information to the Controller. Components NLP and Logic are called from the Controller. The Controller queries NLP for the command the user wants to execute. The NLP then processes parameters such as date/time, location etc. Logic executes the command given by manipulating fields in the storage by through the use of the Storage class.

Sequence Diagram:

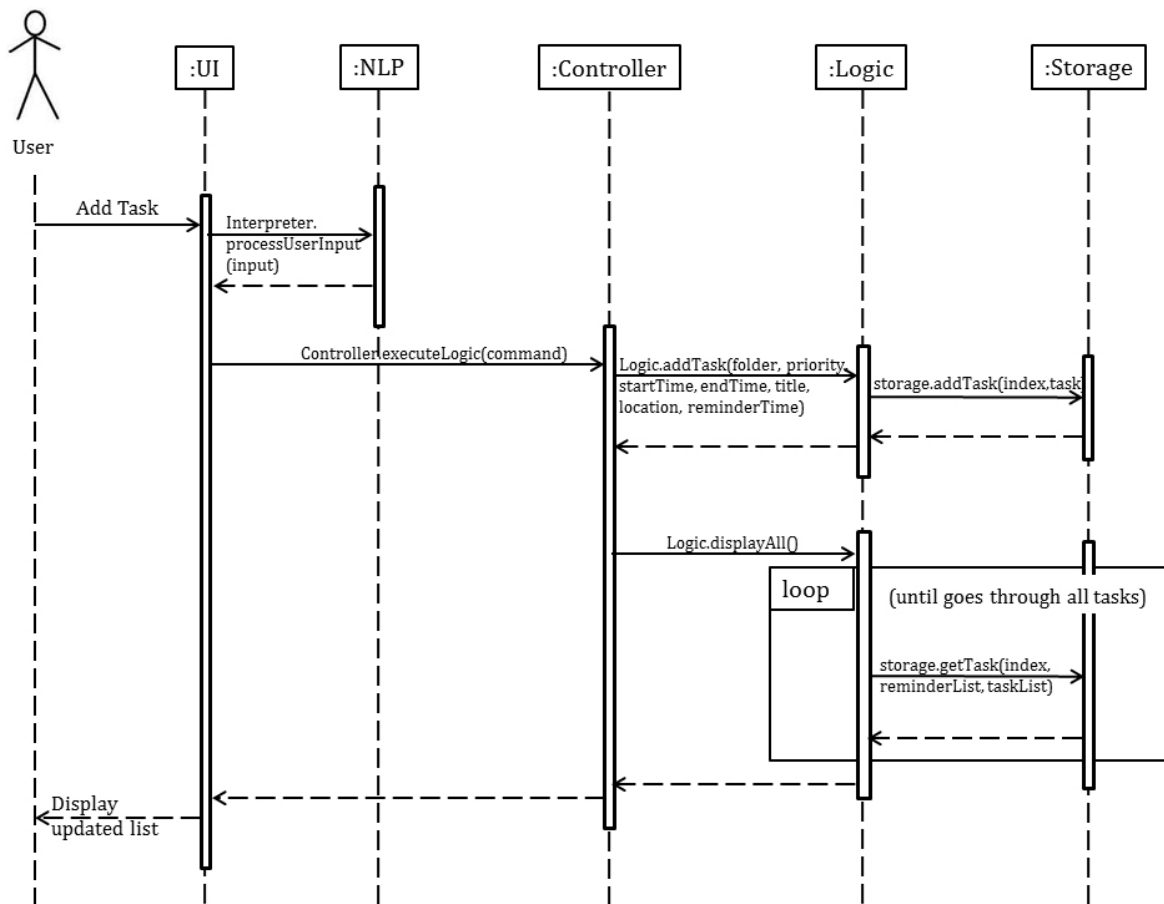


Figure 2: Sequence Diagram when user adds a task

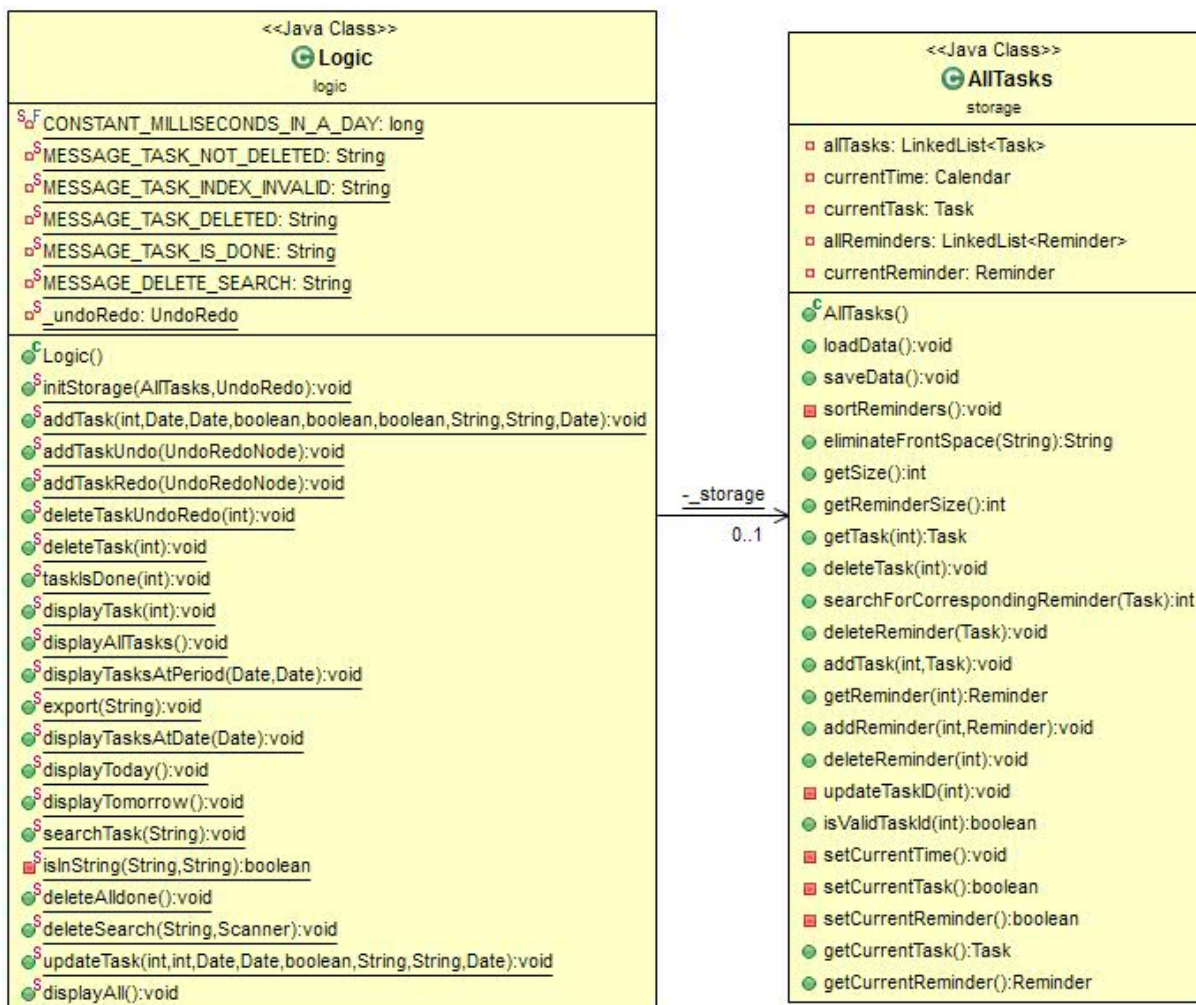
The above diagram illustrates the case on how the components work when the user adds a task. The GUI interacts with NLP to check if the user command input is valid. Thereafter, the user input is passed over to the controller component. The controller component is the main facade for logic and storage. Controller is the “brain” of the program i.e. it does the adding of the task and then getting the list of total tasks for display.

Components:

LOGIC:

The logic component of the architecture is responsible for executing commands and manipulation of fields. The logic accesses the storage by first creating an internal variable of `_storage` of type `AllTasks`, from the method `initStorage`. Hence, each method call for individual commands does not require parameter of `AllTasks`, but rather using the variable `_storage` to access the tasks.

Logic has 2 different types of commands. The first type is executing user commands such as `add`, `delete`, `search` and `modify`. The second type is history editing commands such as `undo` and `redo` operations. The API of the methods is given below:



Logic: Useful methods and Class design

addTask:

The method checks if start and end timings are valid by ensuring end time after start time. `addTask` only allows unfinished tasks to be added. Task will be added in storage by systematically going through each tasks and slotting in the given task based from its starting time. If the specific task has a reminder, the reminder is also added in the same fashion.

deleteTask:

The method deletes the Task at the specified index. The index of the task must be known in order to delete the correct task. The method also searches for the corresponding reminder index through the use of Storage API and deletes the reminder that maps to the specified task.

searchTask:

The method searches for the given String in the description of the Task. It then displays all the tasks that (partially) matches the description - the given String is found inside the description of the Task.

updateTask:

The method updates the given task with the new stated parameters. The `updateTask` is basically `deleteTask` and `addTask` methods that are used one after another respectively. The specified task is deleted and then added back again with the new parameters.

searchDelete:

The method integrates the function of `searchTask` and `deleteTask` to achieve direct deletion of the tasks that are related to the keywords the user has searched for. Before deletion is executed, user would be prompted to confirm the tasks to be deleted.

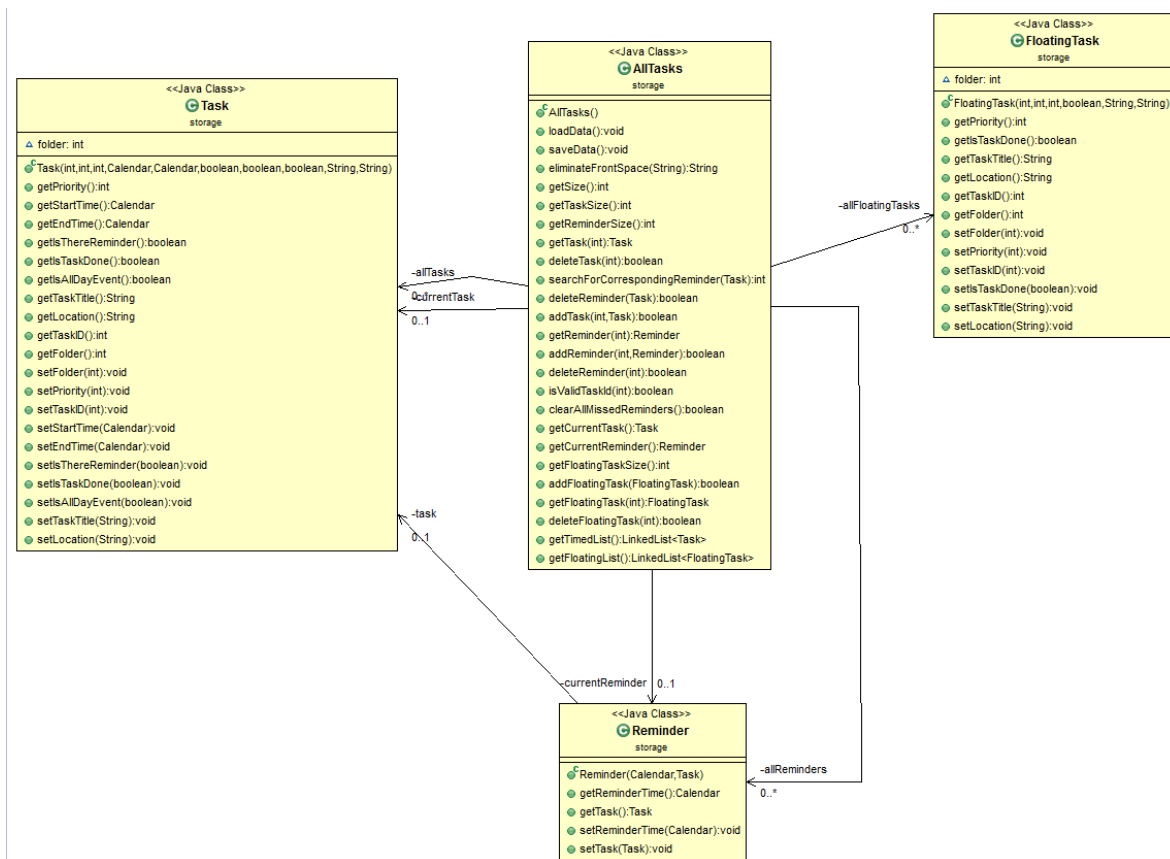
deleteAllDone:

The method will search all the tasks that have been marked as done and delete the done tasks. When deletion is executed, the number of done tasks that have been deleted will be shown to the user.

STORAGE:

The storage component of the architecture being implemented is responsible for the data structures used to handle and store information given by the user. Other components such as the controller and the logic manipulates the data found in the storage using the well defined API given.

The storage component is basically broken down into three major portions. They are namely the `AllTasks` class, `UndoRedo` class and the `TimedTasks` class. Each of the class is used by different components in the overall system to achieve its goal.

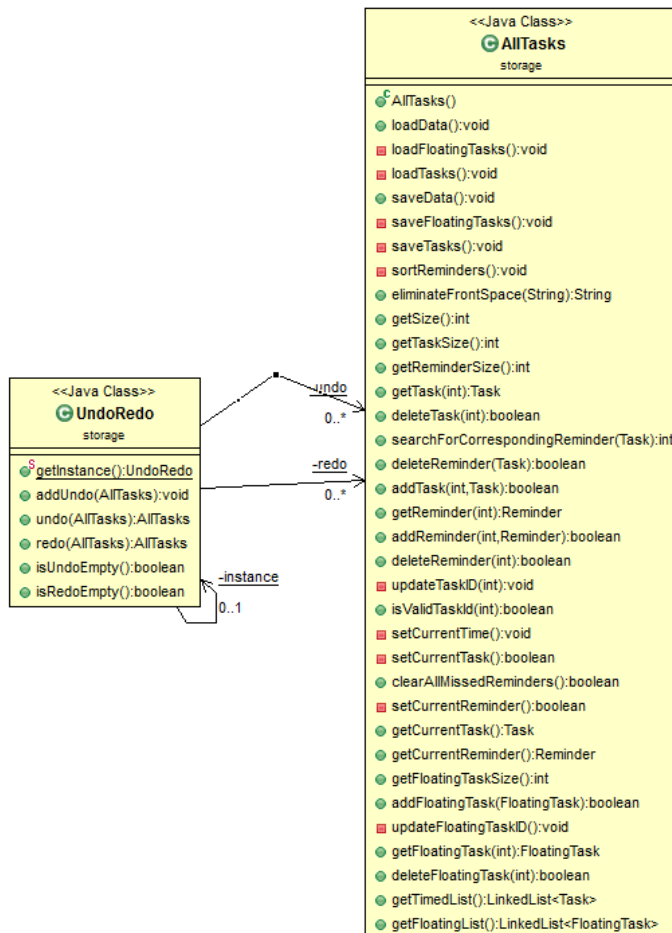


AllTasks: Interaction with Reminder and Task

AllTasks:

This class handles the main storage portion of the program. It stores all the tasks, floating tasks and reminders of the tasks input by the user. It contains three lists, floating tasks, reminders and tasks, using the `FloatingTask`, `Reminders` class and the `Task` class. The three lists are maintained in a sorted order based on the start time of each task and the reminder timing and lastly based on the time of addition of the floating task. The reminder

of each task, if there is one, points to the task it belongs too while storing the reminder time. It is important to note that the reminder of the task and the task itself are two separate entities but having a reminder means that they connected by the pointer of the reminder node pointing to the task node so that reminders are maintained to the task they belong too, even though the lists are sorted chronologically. The class diagram above shows this dependency. For unit-testing `AllTasks`, refer to [AllTasksTest.java](#).



Undo/Redo Class design

UndoRedo Class:

This class handles the undo or redo feature of the entire program. This component basically tracks any changes made to the the the saved tasks. Features such as `add`, `delete` or any kind of modifications are tracked based on saving the state of the `AllTasks` state. These are put on the undo stack every time such a change is made. Hence, when the controller receives an undo command to execute, it directly gets the `AllTask` node on top of the stack and replaces the current one with it. Also, this puts the now popped node from the undo stack onto the redo stack so that the user can choose to execute redo.

It is important to note that once a new change has been made, meaning a new `node` is added to the undo stack, the redo stack will always be cleared. which means that previous undo operations will no longer be saved and cannot be recovered by the redo feature.

The class diagram above shows how the `UndoRedo` class works. It is important to note that the `undoRedo` class is implemented as a singleton so that there can only be one single instance of the class which is shared among the entire system. This will ensure that there is no corruption in the undo or redo process. The class diagram above also lists the api available to the controller as well.

ReminderTimerTask:

This class was made to implement the automatic part of our system. It is calibrated to run every 5 seconds to run. This class was made to achieve and do the following:

- 1) Provides real time reminders of tasks that require them.
- 2) Update the tasks that have already passed to be marked as done.

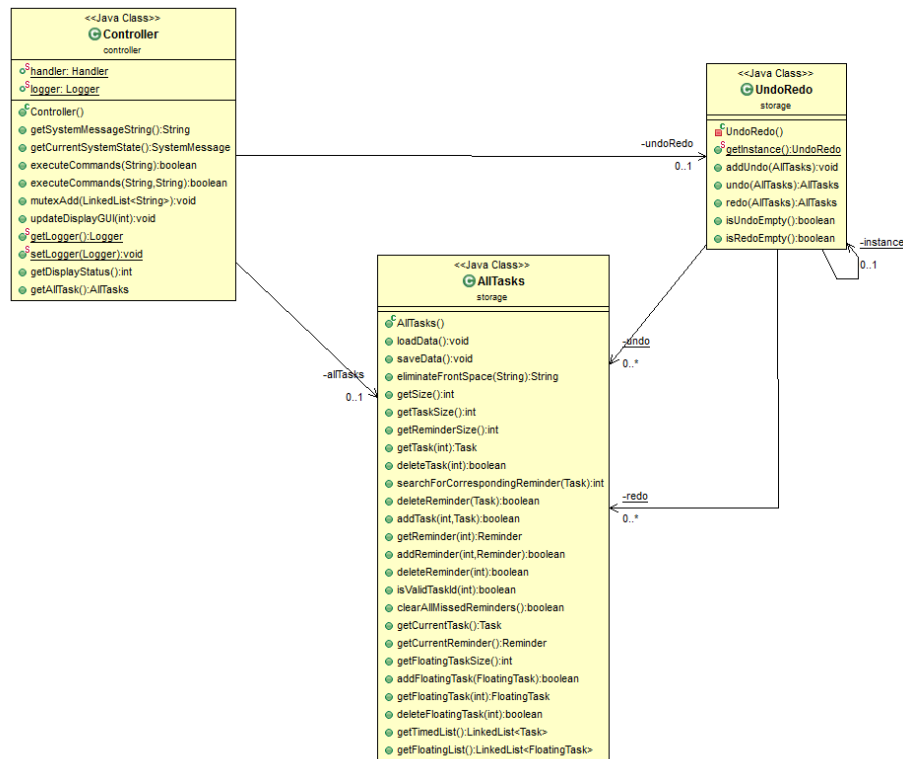
As the class name suggests, the class hopes to achieve the real time part of the system that enables the overall program to be more vibrant and less static in its features. By execution of certain tasks periodically or at a certain time, Tasca becomes more flexible and vibrant in its capabilities.

CONTROLLER:

The `Controller` class can be seen as the “brain” or “filter” of the Tasca program. The controller is the only class that interacts with the GUI and receives information from it. From the information it receives from the user, it determines what the user wants to be done. This is done with the help of the Interpreter component. The interpreter will return to the controller with the main command and the parameters needed to run the command.

With this, the controller then calls the correct function from the logic component to execute the command. The controller function is also responsible for directly executing the `undo`, `redo` or the timed tasks of the system. This is done by directly getting information from the storage component.

The following class diagram will show diagrammatically the way controller function as explained below. It is important to note that controller class is responsible for initialising all the tasks when the program is started up but after start up, it does not interact with the `AllTasks` class directly. Hence, the association of the controller class with the `AllTasks` class.



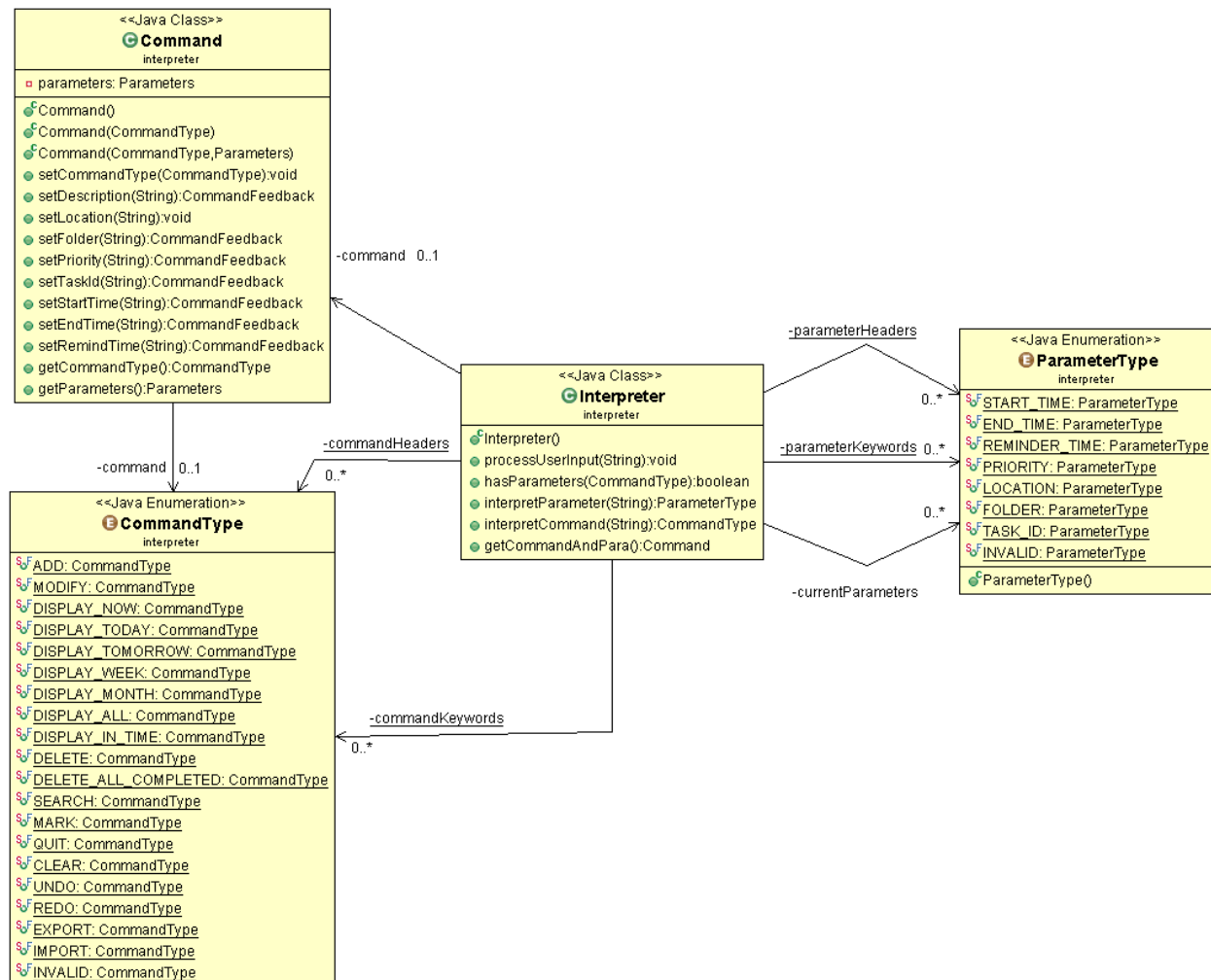
Controller Design

SystemState:

The second part of the controller design involves the class `systemState`. This class is the main source whereby the GUI gets information to display in the display bar. The class involves a message `String` and the two lists, namely floating tasks and timed bound tasks, that have to be displayed by the GUI. It is important to note that the tasks are sorted by whether or not the tasks have been completed before the GUI reads the lists. The system state is updated every time a command is given to be executed by the user.

INTERPRETER:

The interpreter class is the main medium of communication between the user and the system. At its core, it functions as a simple input/output mechanism. Input - `processUserInput(String userInput)`: the controller passes the user input as a string; Output - `getCommandAndPara()`: the controller is given back a system-recognized command and its associated parameters. Other important methods of this class are shown below in the class diagram. These functions are to be mainly used by the GUI to validate the user input (eg: `interpretCommand` can be used for identifying and color-coding a command keyword)



Interpreter: Important API & class design

Keywords Hash-Map:

As mentioned in the user-guide, the user has the ability to specify custom keywords for

commands and parameters. Two linked hash-maps are used to achieve this. The [Config.cfg](#) file (first hash-map) stores all the synonyms of a command/parameter, and it can be edited by the user through the settings pane. These synonyms are then added to a second hash-map, which contains their respective `CommandType`. Note that this hash-map is based on collision; multiple synonyms correspond to the same `CommandType`. To add a new command, three files need to be edited:

[CommandType.java](#): Add `NEW_COMMAND` to the enumeration of system recognized commands

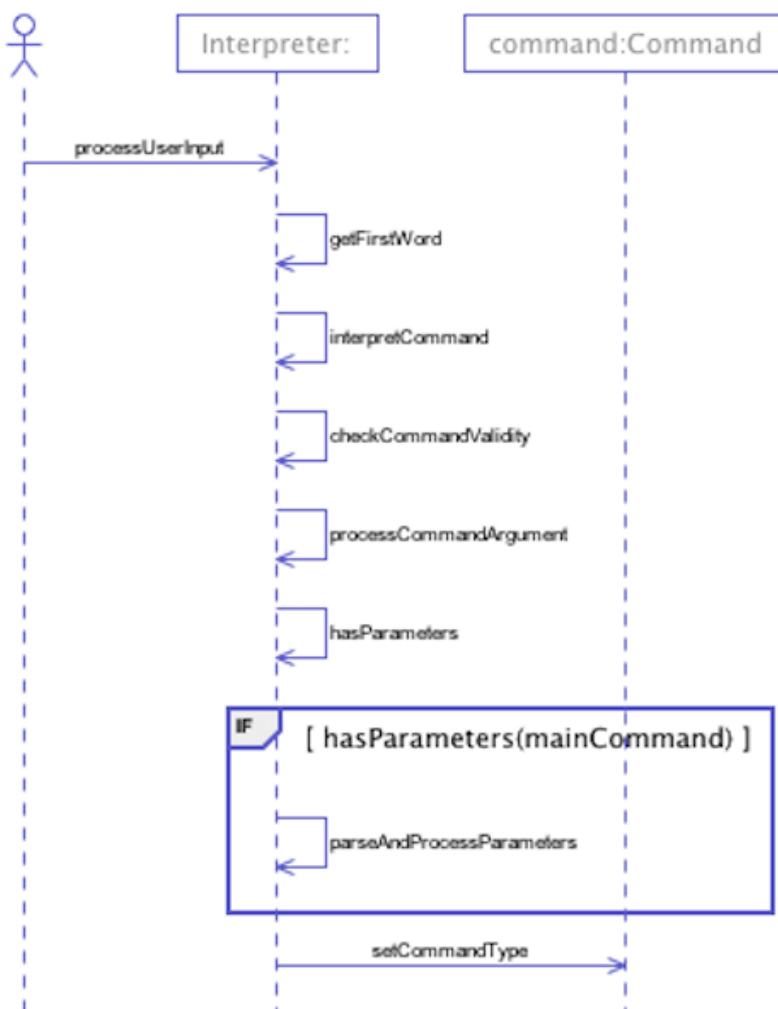
[Config.cfg](#): Add the synonyms of the new command: `newCommand = synonym1, synonym2...`

[Interpreter.java](#): Link the synonyms with the main command

```
commandHeaders.put("newCommand", CommandType.NEW_COMMAND);
```

Deleting existing commands can be done similarly by removing the appropriate (three) references. And likewise for parameter keywords.

Break, interpret, process:



This sequence diagram shows the core-functionality of the interpreter class at work as a chain of events. The first word of the user input contains the main command. So it's identified with the hash-table (keywords and commands) and checked for its validity. Then we check if that specific command has any parameters, e.g: UNDO doesn't have any associated parameters, whereas ADD might have a `startTime` parameter. If the command can have other parameters, then the respective arguments are parsed, processed and stored in an instance of a `Command` class. Finally, the corresponding `CommandType` is stored in the same `Command` instance.

Command Instance:

The output feature `getCommandAndPara()` returns a `Command` instance with `CommandType` and `Parameter(s)`. The `Command` class has various 'set-methods', which are used by the `Interpreter` class to load the specific parameters and their arguments (see class diagram in the previous page). The actual processing of the parameters, e.g: parsing natural language date/time input, is done by the `Parameter` class. When the controller receives the `Command` instance, various 'get-methods' can be used to access the `CommandType` and `Parameter(s)`. Here are some accessor examples (note: `commandAndPara` is a `Command` instance)

```
Command Type: : commandAndPara.getCommandType()

Description: : commandAndPara.getParameters().getDescription();
Location: : commandAndPara.getParameters().getLocation();
Folder: : commandAndPara.getParameters().getFolder();
Priority: : commandAndPara.getParameters().getPriority();
Task ID: : commandAndPara.getParameters().getTaskId();
Start Time: : commandAndPara.getParameters().getStartTime().getTime();
End Time: : commandAndPara.getParameters().getEndTime().getTime();
Reminder Time: : commandAndPara.getParameters().getRemindTime().getTime();
```

Exception Handling:

The `Interpreter` package is designed to handle various exceptions (mostly generated by user input). The exception handling can be broken down into three sections: exceptions while reading the keyword database, exceptions while interpreting/processing the commands, exceptions while interpreting/processing the parameters. A simple try/catch block can be implemented before using interpreter's methods (see [here](#)).

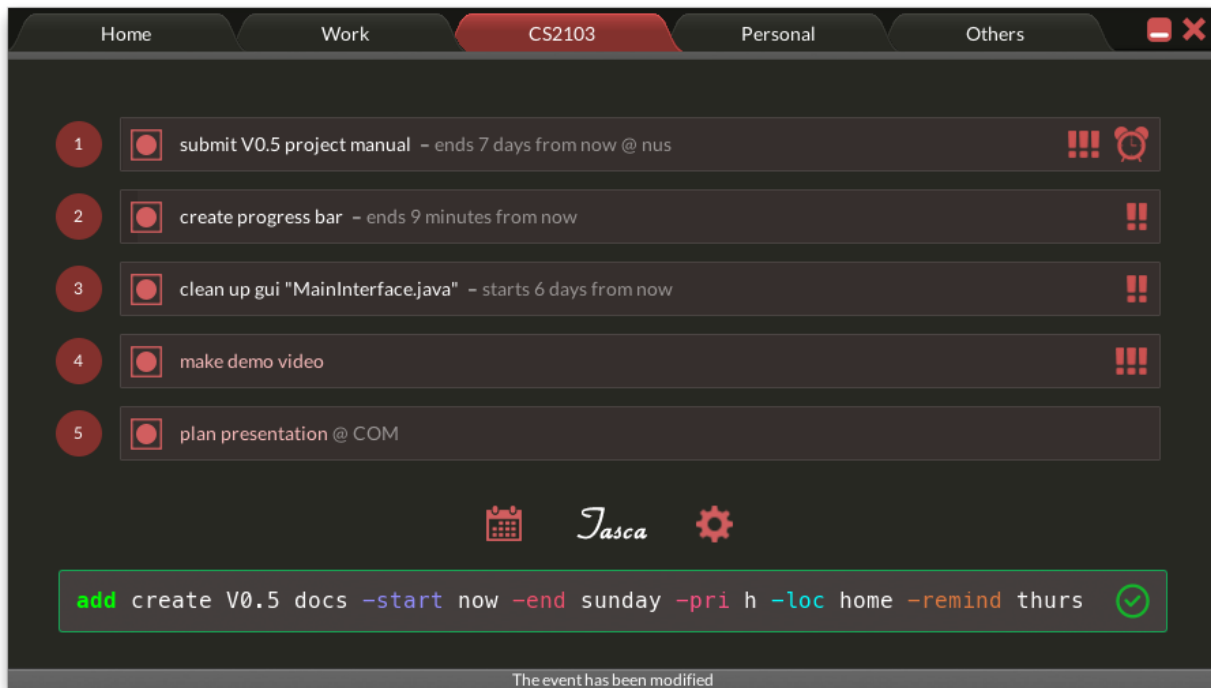
```
INVALID_COMMAND_TYPE = "Invalid command type";
EXCEPTION_EMPTY_ARGUMENT = "Cannot accept empty parameter argument";
EXCEPTION_DUPLICATE_PARAMETERS = "Duplicate parameters found";
INVALID_START_TIME = "Invalid start time";
INVALID_END_TIME = "Invalid end time";
INVALID_REMINDER_TIME = "Invalid remind time";
INVALID_PRIORITY_REF = "Invalid priority reference";
INVALID_FOLDER_REF = "Invalid folder reference";
INVALID_PARAMETER_TYPE = "Invalid parameter type";
INVALID_COMMAND_ARGUMENT = "The description for this command cannot be empty";
ERROR_DATABASE_DUPLICATE PARA = "Duplicate keywords were found in the 'parameter' database";
ERROR_DATABASE_DUPLICATE_COMMAND = "Duplicate keywords were found in the 'command' database";
EXCEPTION_EMPTY_LOCATION = "Initiated location parameter cannot have a empty argument";
EXCEPTION_EMPTY_DESCRIPTION = "Description cannot be empty";
```

Testing Instructions:

Extensive unit-testing was done during the development of the `Interpreter` package. [InterpreterTest.java](#) can be used as reference for further testing. A simple *public* accessor

method can be used to evaluate private methods.

Graphical User Interface:

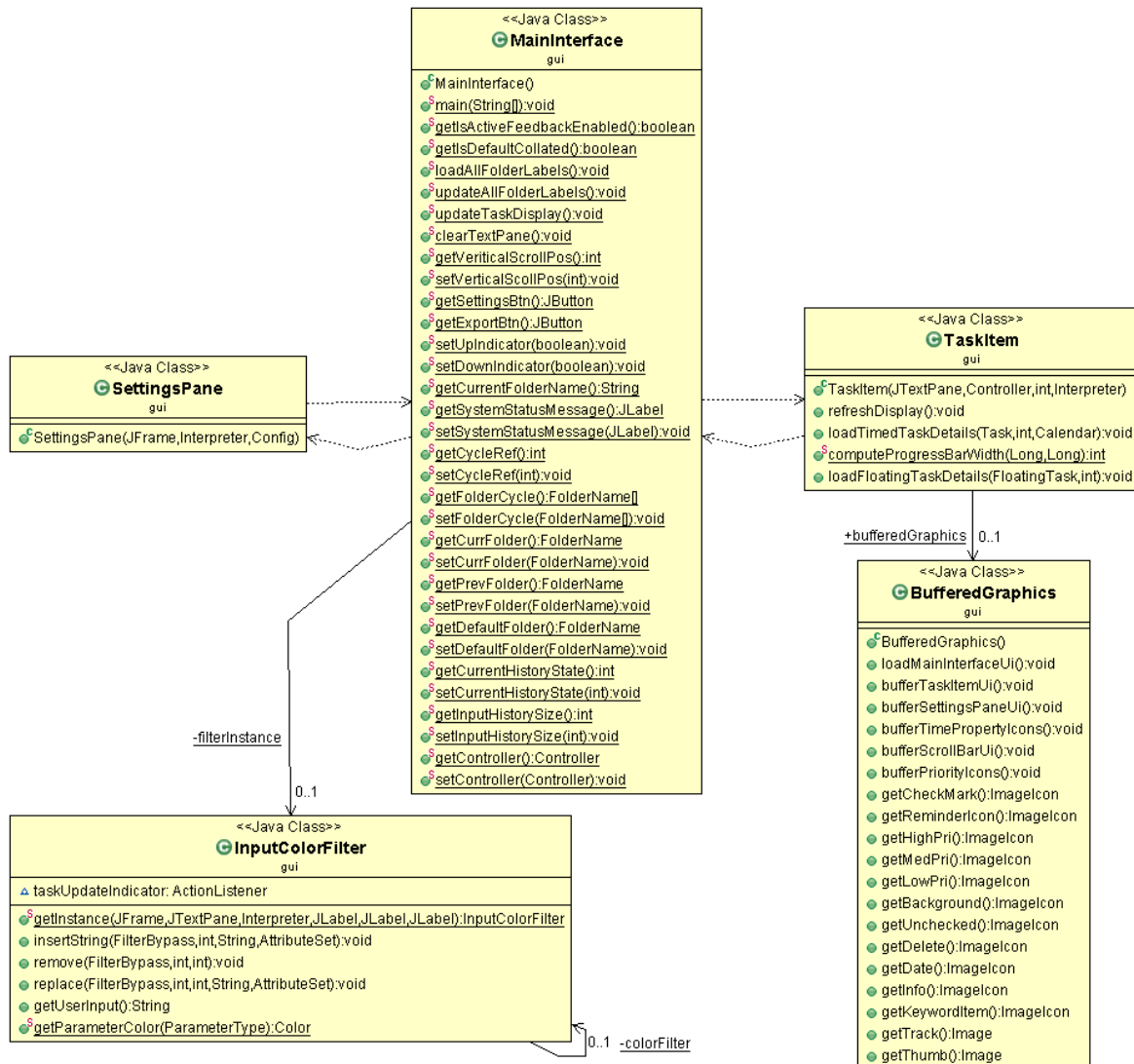


The primary mode of user input in TASCA is command-line. However, the application is also equipped with a powerful GUI to assist the communication between the user and the system. Besides Logic & Storage, this is one of the most comprehensive components in the entire program. Although the component itself is designed to be very flexible for future iterations (through consistent OOP practices), the graphical elements themselves are not easily amendable. This is because almost all the UI elements rely heavily on bitmap graphics, which can't be rescaled without quality loss. Future developers might consider replacing these with vector graphics.

The following sections provide a brief overview on the key constituents of the GUI. Bear in mind that the actual implementation is much more extensive.

Main Interface:

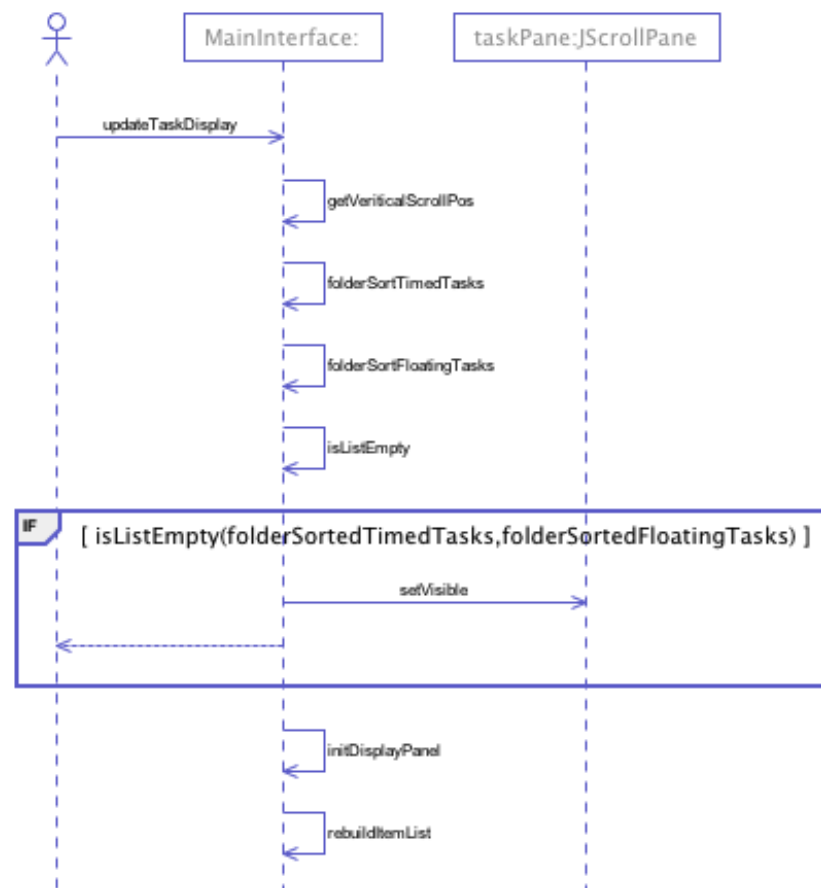
This class is the main medium of interaction between the user and TASCA. It's the only class in the entire project that contains a *main* function. It directly interacts with two other classes: NLP & Controller. But before the user input is passed on to the Controller, there are several other internal components with which it can possibly interact (see GUI: Simplified Class Diagram). These are: *TaskItem* (an individual element that contains all the information relating to a specific task - multiple instances), *FolderTab* (for controlling folder states - multiple instances), *InputColorFilter* (for highlighting user input - singleton), *InputHistory* (for managing previous user inputs - single instance), *IOPane* (interface for Export/Import function - single instance), *ProgressBar* (for managing the visual progress indicator - multiple instances), *SettingsPane* (customization interface) and *ScrollBarUI* (for scrolling pane interaction - multiple instances).



GUI: Simplified Class Diagram

Because of the mass outsourcing of task execution to its neighbouring classes, the *MainInterface* itself does minimal work. In fact, the only important element managed by *MainInterface* is the task scroll-pane. This pane is responsible for managing the list of tasks provided by the controller (upon request) with the help of *TaskItem*. The following sequence diagram shows the result of calling *updateTaskDisplay()*. Before adding the graphical elements to the pane, the incoming list of tasks must be sorted according to the current folder context to show only the items that are relevant. Furthermore, timed tasks & floating tasks have to be added separately as they exhibit different characteristics when displayed to the user. During future development, if more 'types' of tasks are added to the system, it's suggested that a separate class be created to manage the contents of the pane.

updateTaskDisplay() is called by various components after the execution of an user imposed action. Supposedly, this is the most computational intensive process managed by the GUI. Prospectively, *Swing* multi-threading can be implemented to boost the performance of this component. However we have yet to experience any sort performance issues with the current scroll-pane.



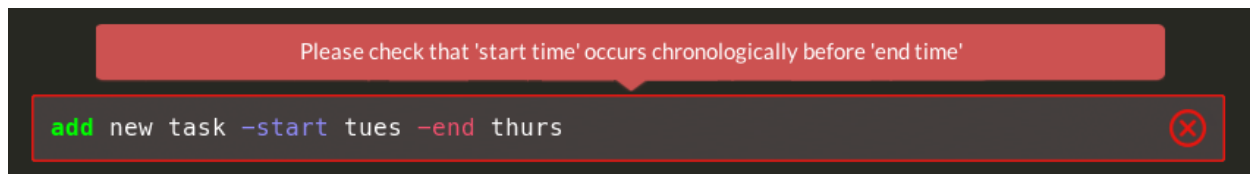
Task Item:

Besides *MainInterface*, *TaskItem* is one of the few classes which are exclusively built for user interaction. For example, if the user wishes to modify any visible property of a specific task, they can simply click on it. In addition, *TaskItem* also translates all the *Task* & *FloatingTask* attributes into minimalistic graphical elements as it's initiated. To ensure rapid duplicability, all the bitmap images are buffered and accessed through a separate class named *BufferedGraphics*. Occasionally, *TaskItem* directly interacts with *Controller* for actions like cycling through priorities. An obvious question that arises from this is: why

doesn't the GUI strictly talk to *Logic* instead of communicating via *Controller*? Because by doing so, we violate the 'Law of Demeter' on multiple occasions. *Logic* is not designed to be independent; it's coupled with other classes like *AllTasks* and *Storage*, which have their own overheads that have to be managed before calling functions inside *Logic*. In the current system, the *Controller* can be still referred to as the main façade for interacting with *Logic* & *Storage*.

Input Color Filter:

Active input feedback is a crucial feature of TASCA's user design. New developers should treat the filter class with caution and ensure that the responsiveness of the system is not seriously affected by the addition of new features. As with many components in GUI, most of the processing is outsourced. In this case, the *Interpreter* class is used extensively for



checking the validity of the user input. And this verification is done in real time (using *Interpreter's processUserInput(input)* method), after each keystroke; hence the need for quick responsiveness. The exception messages thrown by the *Interpreter* (if any) are then appended to the feedback bar. Likewise, the color highlighting functions *addCommandColors* & *addParameterColors* rely on the *Interpreter's* keyword hash-map for checking synonyms. The color filter follows a very simple scheme: highlight the first word if it's a valid command, highlight all subsequent words with the delimiter "-" if they are recognized as valid parameters by the *Interpreter*. If future development requires the addition of new commands/parameters, the coloring hash-map has to be updated along with the *Interpreter's* equivalent:

```
commandColors.put(CommandType.ADD, Color.green.brighter());
commandColors.put(CommandType.MODIFY, Color.yellow.darker());
commandColors.put(CommandType.DISPLAY_NOW, Color.cyan.darker());
commandColors.put(CommandType.DISPLAY_TODAY, Color.cyan.darker());
commandColors.put(CommandType.DISPLAY_TOMORROW, Color.cyan.darker());
commandColors.put(CommandType.DISPLAY_WEEK, Color.cyan.darker());
commandColors.put(CommandType.DISPLAY_MONTH, Color.cyan.darker());
```

```
commandColors.put(CommandType.NEW_COMMAND, Color.<NEW_COLOR>); ...
```

Since the program needs at most one instance of the color filter (because there is only one input pane), the *InputColorFilter* class was designed as a singleton.

Settings Pane:

The *SettingsPane* class is quite similar to that of *MainInterface*'s in a sense that they are both standalone frames with common UI elements. Both components also interact with each other on a regular basis. But the most important interaction takes place between *SettingsPane* & *Config*. As the user edits the keyword database, new synonyms have to be checked for exceptions (duplicates, multiple-words & empty strings). If the amended database is valid, then *Config* is called before writing the *Config.cfg* file to the local directory. Precaution has been taken such that if the user accidentally deletes *Config.cfg*, a default configuration file will be generated from the JAR file (although saved settings will be lost). An intuitive response to this problem might be to put *Config.cfg* inside the JAR file itself so it becomes inaccessible to the user (except through the settings pane), but IO operations cannot be executed inside JAR (Java Archive) files.

Testing:

GUI is the least tested (i.e. automated tests) package in the project. Most of the evaluation was carried out through unscripted, low-key, 'black-box' testing. However, the development team has taken extensive measures to reduce GUI-related bugs and maximize user experience. Future developers might consider scripted, fully-automated testing.

Developers Note:

TASCA is developed to be a light-weight, compact and user friendly task management system. TASCA is a dependable way to track your day events and ensure that you are reminded every time that you ask it too. Its accurate, on time, reminders complements the way the user ensures that he is able be on top of things and never late! TASCA comes with a beautiful GUI to complement its functionality for the average user. However, our main customer group is still allowed to perform every task in command line as they please. TASCA is aimed to be flexible in terms of storing tasks, be it being floating or time bound tasks. TASCA will always be able to provide a means for you to be able to organise your day.