JavaCake

Developer Guide

v1.4

Prepared by

Claire Chan Yen Hwa
Glen Wong Shu Ze
Kishore R
Rusdi Haizim B Rahim

JavaCake - Developer Guide

1. Introduction

1.1. Purpose

This document specifies architecture and software design decisions for the educational app, JavaCake. The intended audience of this document is the software developers, designers, and testers of JavaCake.

1.2. Design Goals

Target user profile:

- Has a need to gain a basic grasp of Java
- Prefers desktop apps over mobile
- Can type quickly
- Prefers typing over mouse input
- Is reasonably comfortable using Command Line Interface (CLI) apps
- Is goal-driven (likes setting goals for themselves)

Value proposition: Managing the learning of Java and managing the learning progress through a desktop app which is CLI-based. Developing JavaCake poses a unique software development situation. The developers work closely to ensure both technical and non-technical requirements are met in the development process. The design of JavaCake takes into consideration various factors such as user-friendliness and relevance.

Note the following symbols to be used:

This symbol indicates important information.

`list` A grey highlight (called a mark-up) indicates that this is a command that can be inputted into the command line and executed by the application.

`Logic` Blue text with grey highlight indicates a component, class or object in the architecture of the application.

`currentFilePath` Italicised `Consolas` font will be used to denote variable/attribute names used inside a Java Class.

2. Setting up

2.1 Prerequisites

1. JDK 11 or above.
2. IntelliJ Integrated Development Interface (IDE)

IntelliJ by default has Gradle and JavaFx plugins installed. Do not disable them.
If you have disabled them, go to `File` > `Settings` > `Plugins` to re-enable them.

2.2. Project set-up

1. Install JDK 11, if not yet installed.
2. Fork this GitHub repository to your GitHub account: https://github.com/AY1920S1-CS2113T-W13-2/main.
3. Clone your fork to your local computer.
4. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first)
5. Set up the correct JDK version for Gradle
    a. Click `Configure` > `Project Defaults` > `Project Structure`
    b. Click `New...` and find the directory of the JDK
6. Click `Import Project`
7. Locate the `build.gradle` file and select it. Click `OK`
8. Click `Open as Project`
9. Click `OK` to accept the default settings
10. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`).
    It should finish with the `BUILD SUCCESSFUL` message.
    This will generate all resources required by the application and tests.

3. Design

3.1 Architecture

The *Architecture diagram* given below (Figure 1) explains the high-level design of the app. Below it is a quick overview of each component.

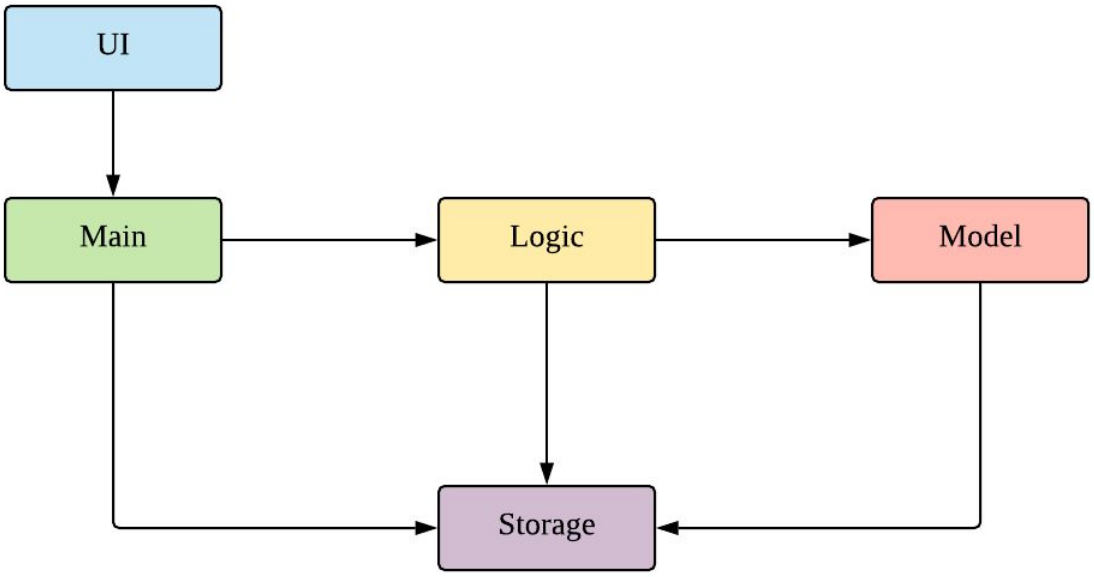

Figure 1: Architecture Diagram

The figure above showcases the high-level view of how JavaCake is run. The program can be abstracted into these five modules which are explained below.

- `UI`: Handles user interaction and user interface.
- `Main`: Acts as the bridge for transferring inputs and outputs from the `UI` to `Logic` and vice versa.
- `Logic`: Executes the respective commands after user input is parsed.
- `Model`: Holds the current content and quiz questions in memory during the operation of the program.
- `Storage`: Reads data from and writes data to hard disk when required.

The following sections below provide more information on each module.

3.2 Logic

Within the logic module, the major classes include `Logic`, `Parser`, and `QuizManager`. The `Parser` class interprets user input and instantiates the relevant `Command` object.

`QuizSession` and `ReviewSession` interacts with the `Model` module to retrieve questions and set up a quiz or review.

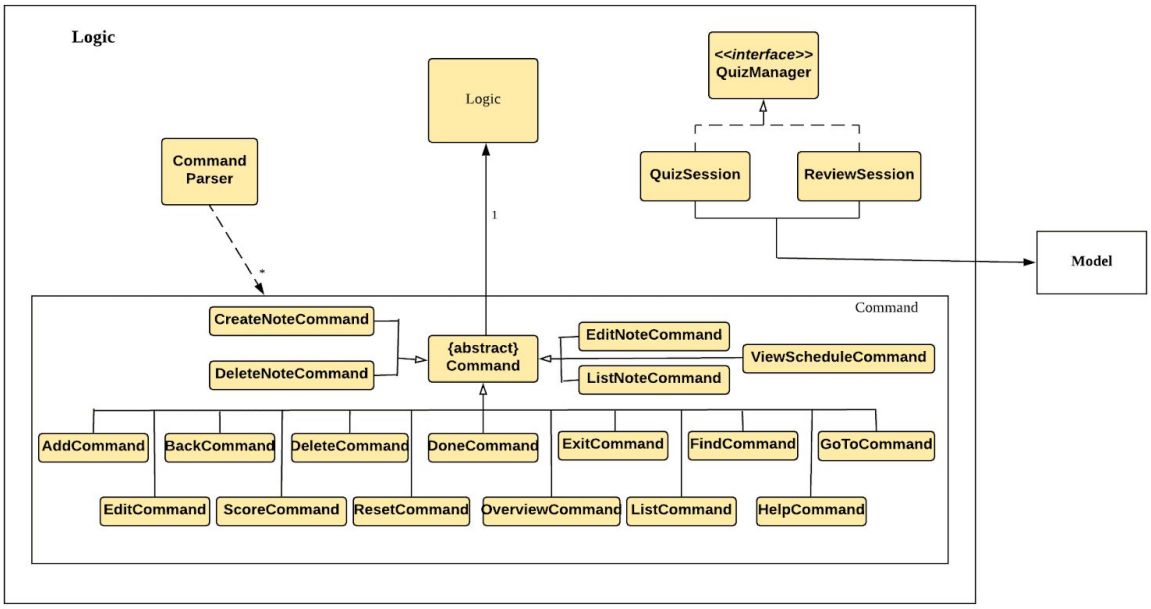

Figure 2: Structure of the Logic Component

3.3 Model

The `Model` stores a `QuestionList` object which itself consists of one or more `Question` objects.

`Model` also references `Storage` in order to load and generate the `QuestionList` object from `Storage` when a `QuestionList` object is created.

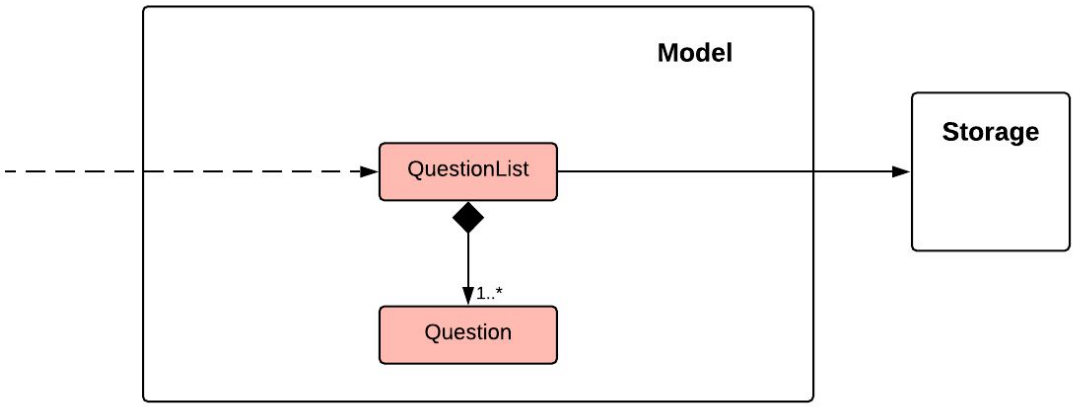

Figure 3: Structure of the Model Component

3.4 Storage

The `Storage` module handles fetching data from and writing data to the hard disk. This module consists of the following classes: `Profile` , `Storage`, `TaskList` and `StorageManager`.

There is also an enumeration class which keeps track of the different state of the task(`TaskState`).

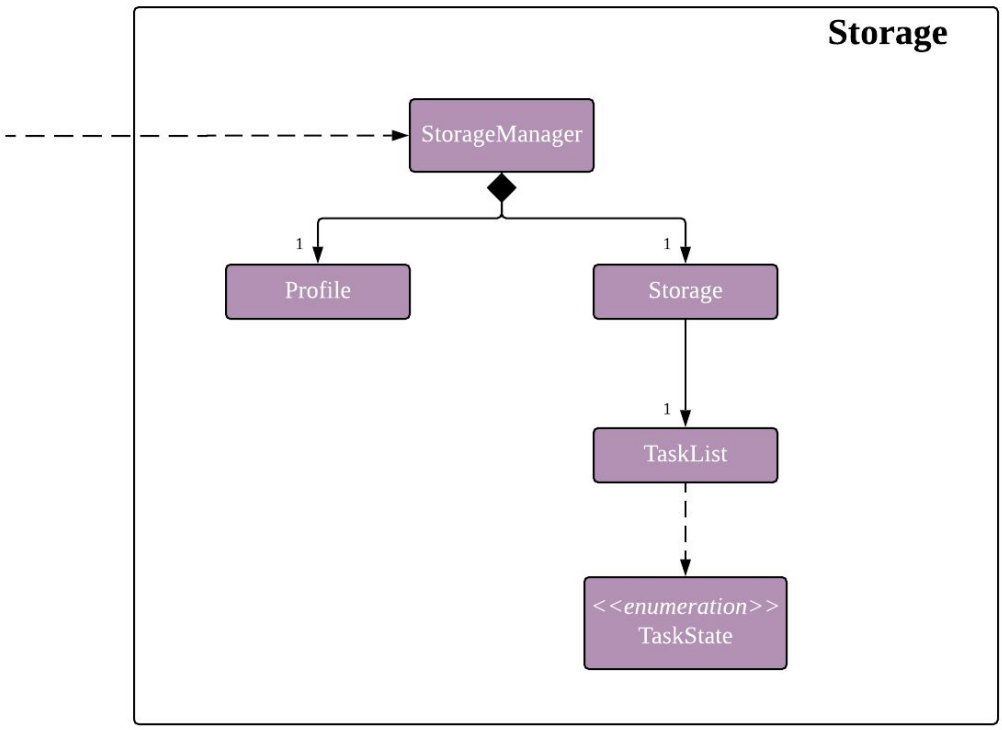

Figure 4: Structure of the Storage Component

The `Storage` component,

- can save `Profile` objects in text files and read it back.
- can save the `Storage` data in text files and read it back.

3.5 UI

This module handles the user interface and user interaction with the program. It consists of a `MainWindow`, which consists of the following parts: `DialogBox`, `TopBar` and `AvatarScreen`. The user input is directly handled and processed by `MainWindow` in **GUI** mode. The module also contains a `Ui` class which handles user input in **CLI** mode. The `Main` component is included inside this module, since both `Ui` and `MainWindow` directly navigates to the `JavaCake` class (which transfers user input to the `Logic` module for parsing).

GUI: Graphical User Interface
An interface which allows users to view the app along with the respective graphical icons/views.
Used for product release.

CLI: Command Line Interface
An interface which only allows users to interact with the app through text-based inputs, usually restricted to a single view.
Only used for debugging and not for product release.

The following diagram depicts how this module is implemented in GUI and CLI modes.

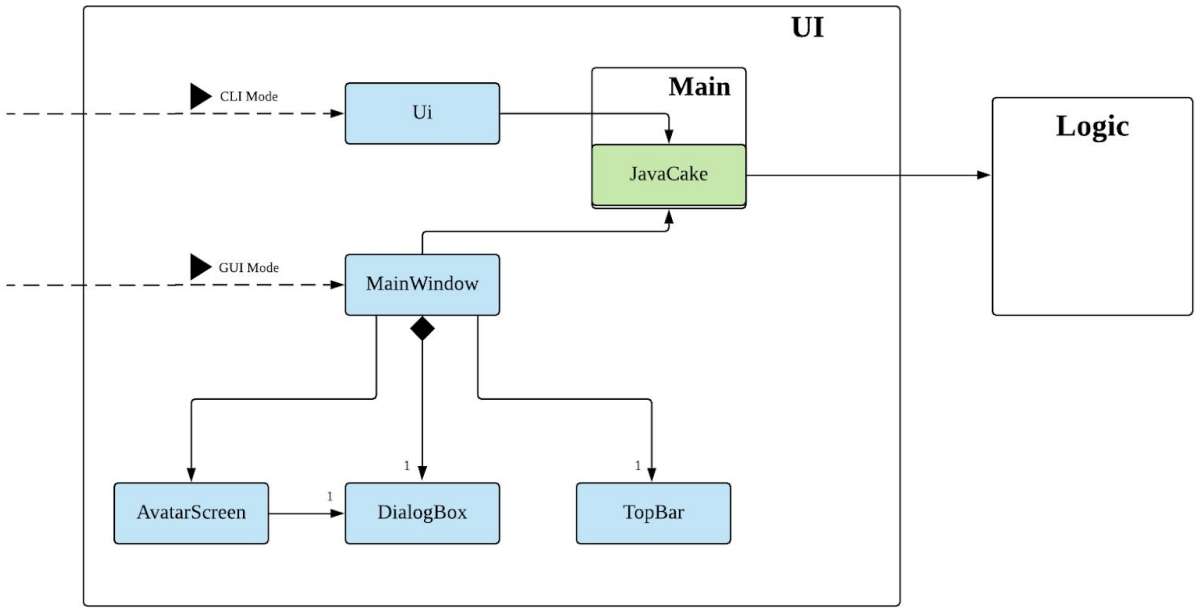

Figure 5: Structure of the UI Component

The `UI` component,

- In **CLI** Mode, executes user commands through the `Main` component which executes the respective commands in `Logic`. The output is shown in the developer's console.
- In **GUI** Mode, executes user commands through the `Main` component which executes the respective commands in `Logic`. The output is then shown in the `DialogBox` and `TopBar` classes.
- Listens for changes to data so that components under `MainWindow` can be updated.

4. Implementation

This section describes some noteworthy details on how certain features are implemented.

Technologies Used:

- Our source code is written in Java. However, we also leverage on JavaFX, fxml and Cascading Style Sheets (CSS) to design our Graphical User Interface (GUI).

- The Apache Commons IO library is used to handle file cleanup in the `Storage` module.

- Travis, a continuous integration service, was used to test our JUnit tests on their remote servers, which ensures that other developers hoping to develop our project further are not breaking the current implementation of the product when carrying out pull-requests.

4.1 Content Browsing feature

4.1.1 Proposed Implementation

The browsing of content feature is facilitated by `Logic`, which allows users to dynamically navigate through the content in the content directory without the need to hardcode any of the content in our codebase.

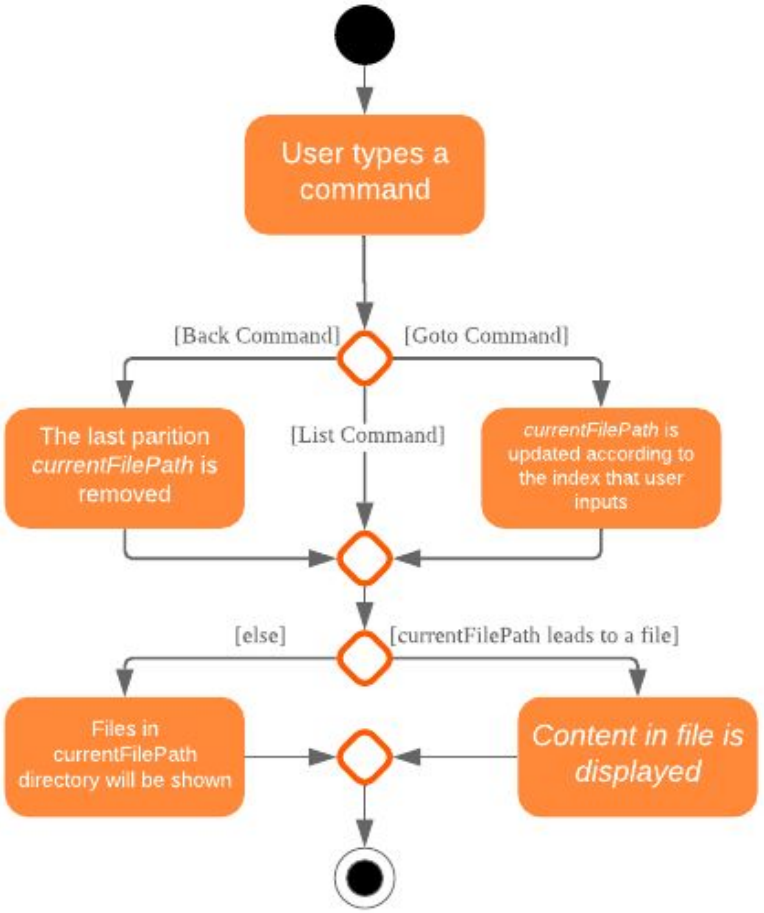

Figure 6: Activity Diagram for Content Browsing in JavaCake

The figure above shows the overall activity diagram for content browsing. Two variables of *`defaultFilePath`* and *`currentFilePath`* are used, in which *`defaultFilePath`* stores the file path towards the start of our content directory and

currentFilePath is used to store the updated file path towards the content requested by the user.

When a command, such as `list`, `back` and `goto`, requires the program to traverse through the content directory is called, *currentFilePath* is updated by concatenating the name of the file to itself.

The files found under the *currentFilePath* can be either text files or directories. If the file in *currentFilePath* are text files, the *currentFilePath* will be updated once more to enter the filename (specified by user-inputted index) in order to read the content stored in the file. Then the content will be displayed to the user. If the files contained in *currentFilePath* are directories, the name of the directories under it will be displayed to the user.

The name of the file(s) found in the current directory will be stored in *listOfFiles*, which is a list container for strings.

4.1.2 Design Considerations

Aspect 1: How reading of content works

- **Alternative 1 (current choice):** Dynamically reads and displays content based on the names of directories or names of files.
    - Pros: Very scalable, no hard-coding required.
    - Cons: Slightly harder implementation of reading content.

- **Alternative 2:** Creating individual classes for each subtopic.
    - Pros: Easier to code since it only requires hard-coding.
    - Cons: Not scalable, expanding content files require redoing of codebase.
- **Alternative 3:** Hardcoding location of every file and directory.
    - Pros: Very easy brute force way to implement.

○ Cons: Tedious and not scalable when content increases.

Aspect 2: Data structure to keep track of current location in program

- **Alternative 1 (current choice):** Storing current file path in a string variable.

○ Pros: Very scalable, concatenate string variable with new file path.

○ Cons: Slightly harder implementation since the file locations are harder to find and keep track in jar files.

- **Alternative 2:** Using a stack data structure to store current progress in program.

○ Pros: Easy to implement.

○ Cons: Not scalable especially when content files are expanded since every new path location has to be properly indexed.

4.2 ListCommand feature

4.2.1 Proposed Implementation

When the command entered by the user is `list`, *`currentFilePath`* will be reset to *`defaultFilePath`* in which the names of the directories stored within the start of our content file will be displayed. To make it more scalable, we conveniently renamed our directories to have proper indexing.

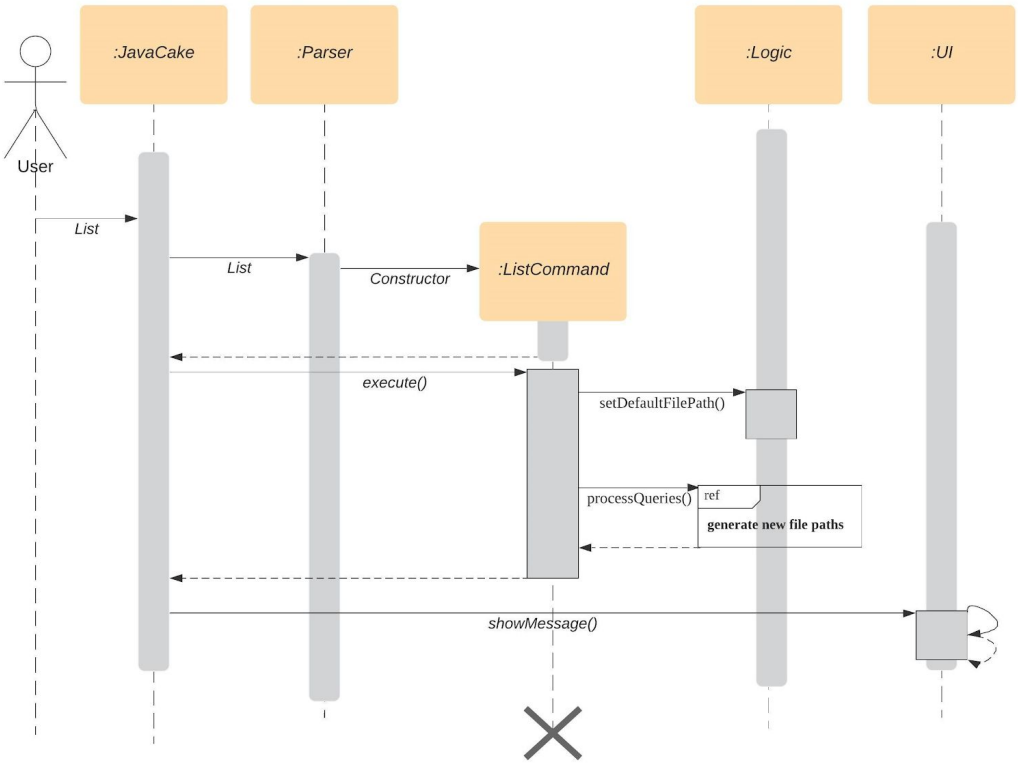

Figure 6: Sequence diagram for ListCommand

ListCommand implements the following methods in `Logic` as shown in Figure 2:

- `Logic#setDefaultFilePath()` — Resetting the file path back to default.
- `Logic#processQueries()` — Storing all possible file paths from current directory.

4.3 GoToCommand feature

4.3.1 Proposed Implementation

When the command entered by the user is `goto [index]`, *currentFilePath* will be updated by concatenating the file or directory name found in the *index* at that particular directory.
If the name refers to a new directory, a list of the items in the directory will be shown. Else, content, which may include the quiz, is shown.
If the user knows the location of the file/directory and wishes to view it directly instead of going through the directories one by one, the user just needs to concatenating the index of the content or directory with a '.'. The index of files goto command is expected to go through is stored in a queue called *indexQueue*. *currentFilePath* will be called until all the index in the queue is popped as shown in *Figure 7* below.

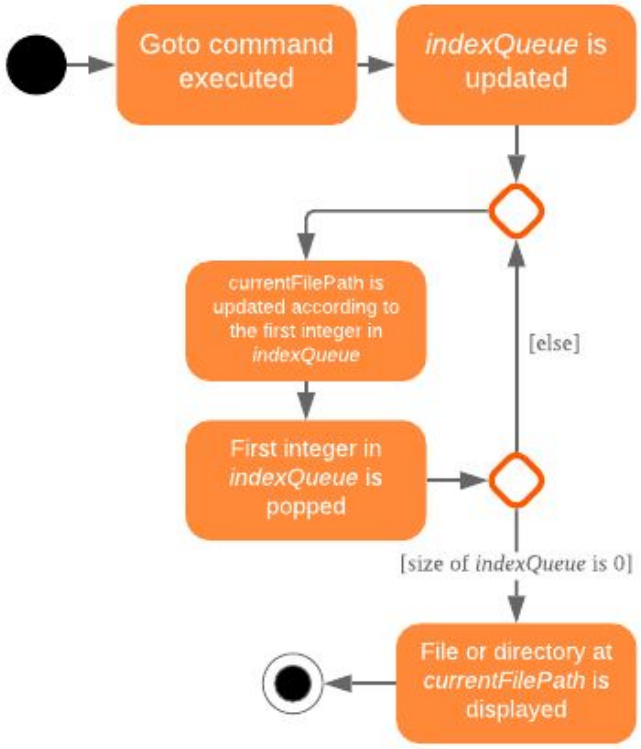

Figure 7: Activity diagram for Goto Command

This feature implements the following methods in `Logic`:

- `Logic#gotoFilePath()` — Depending on the *`index`*, a particular file path will be selected from the collection of file paths generated from previous command.

- `Logic#updateFilePath()` — Updates *`currentFilePath`*.

- `Logic#insertQueries()` — Insert all possible file paths based on current directory.

- `Logic#displayDirectories()` — Prints out all files that are directories.

- `Logic#readQuery()` — Read the content in a text file.

4.4 BackCommand feature

4.4.1 Proposed Implementation

When the command entered by the user is `back`, *`currentFilePath`* will be updated depending if the current file is a directory or a text file. If the current file is a directory, the last partition of the *`currentFilePath`* will be removed. If the current file is a text file, the last 2 partitions of the *`currentFilePath`* will be removed. After which, the appropriate content will be displayed to the user.

This feature implements the following methods in `Logic`:

- `Logic#insertQueries()` — Insert all possible file paths based on current directory.
- `Logic#displayDirectories()` — Prints out all files that are directories.
- `Logic#readQuery()` — Read the content in a text file.
- `Logic#backToPreviousPath()` — Checks if current file is a directory or file. If it is a directory, removes last partition of *`currentFilePath`*, else removes last 2 partitions of *`currentFilePath`*.

4.5 Profile feature

4.5.1 Proposed Implementation

When the user first launches the program, the user will be prompted to fill out his username. The `Profile` keeps a reference of the default file path and creates the respective directories and files if they do not exist.

Two variables of *`filepath`* and *`username`* are used.
The *`filepath`* stores the default file path of where the save file should be stored, along with its respective file name.
The *`username`* is used to store the current user's username which can be accessed either internally within `Profile` or externally via external function calls.

This feature implements the following methods in `Profile`:

- `profile#getUsername()` — Gets the username of the user.

- `profile#resetProfile()` — Resets the current user's profile, along with their respective data after calling the `reset` command.

- `profile#overwriteName()` — Overwrites the default username of the user, when either first launching the program or when `reset` is called.

- `profile#setIndividualMarks()` — Sets the marks of the user for a particular quiz in a particular difficulty (specified in the function parameters).

- `profile#getTotalProgress()` — Gets the overall marks of the user for all the quizzes.

- `profile#getIndividualContentMarks()` — Gets the marks of the user for a particular quiz in a particular difficulty (specified in the function parameters).

- `profile#getOverallContentMarks()` — Gets the overall marks of the user for a particular quiz. (specified in the function parameters).

4.6 Quiz and Review feature

4.6.1 Proposed Implementation

When `goto [index]` leads to the location of the quiz content, `QuizSession` is called. The question type *`qnType`* and level of difficulty *`qnDifficulty`* are both set by reading the *filePath* from `Logic`. A `QuestionList` object, containing an ArrayList of `Question` objects, is initialized by reading all text files containing the questions in its current location, then randomly selecting *`MAX_QUESTIONS`* number of questions. `QuizSession` will use this `QuestionList` object to facilitate the quiz session.

The quiz session starts with a *`currScore`* of 0. A question is displayed and the program awaits user input. Once user input is received, it is compared to the correct answer of that question and *`currScore`* increases by 1 if they match. The next question is then displayed to await user input. This continues until all questions in the session are attempted.

When the quiz session is complete, a results screen will be shown to the user, displaying the final *`currScore`* out of *`MAX_QUESTIONS`*. Custom messages and different avatar expression will be displayed as well, determined by a *`scoreGrade`* of `BAD`, `OKAY` or `GOOD`, which in turn is determined by the calculated percentage score in the quiz session. Additionally, if *`currScore`* is higher than the user's previous score that was stored in `Profile`, the new score will be updated. `TopBar` will also be updated to show a visual change in the progress bar.

On the results screen, only `review` and `back` are available to be executed. `review` will initialize `ReviewSession`, where the questions from the previously completed quiz sessions are shown again along with the user's answer and the correct answer. The user input in a review session should either be integers that correspond to the question number in order to jump to that question, or to execute a `back` command.

Both `QuizSession` and `ReviewSession` implement the interface `QuizManager`.

This feature implements the following methods:

- `QuestionList#loadQuestions()` — loads all the questions in text files and stores them in an ArrayList.

- `QuizSession#getQuestion()` — returns the question string.

- `QuizSession#parseInput()` — parses user input. If input is integer sets user's answer to the question, else if the quiz is complete and input is a relevant command returns a response to execute the command.

- `QuizSession#getQuizResult()` — returns the text of the result screen, including *currScore*.

- `QuizSession#overwriteOldScore()` — updates both the score in `Profile` to the new score from the quiz session and the `TopBar` if the new score is higher than the score in `Profile`.

- `ReviewSession#getQuestion()` — returns the question string, along with the user's answer and the correct answer.

- `ReviewSession#parseInput()` — parses user input. If input is integer converts the input to valid index for calling the next question and returns it. If input is `back`, returns response to execute the command.

4.6.2 Design Considerations

Aspect 1: How quiz content is stored and read

- Alternative 1: (Current choice) Quiz questions are stored with their answers in the text files. The QuestionList#loadQuestions() method will iterate through all the files and store them into the quiz array.

  - Pros: Very scalable as additional questions and answers can be easily added without having to manually change the code.

  - Cons: Each text file that contains the quiz must follow a certain naming format.

- Alternative 2: `QuestionList` class contains and maintains all hard coded Question objects and the number of quizzes that each topic contains.

- Pros: Easy to implement and test as it is not susceptible to IO or File exceptions that may arise from reading from an external file.

- Cons: As all questions and answers have to be hard coded within the class, it is not scalable as number of quiz questions increases.

4.7 OverviewCommand feature

4.7.1 Proposed Implementation

When the command entered by the user is `overview`, *`currentFilePath`* will be reset to *`defaultFilePath`*. By iterating through the files and comparing with *`defaultFilePath`*, we store and format the name of the files depending on the number of parent directories it contains.

This feature implements the following methods in `Logic`:

- `Logic#setDefaultFilePath()` — Resetting the file path back to default.
- `Logic#insertQueries()` — Insert all possible file paths based on current directory.

4.8 CRUD feature for *notes*

4.8.1 Proposed Implementation

In the context of *JavaCake, notes* are personalised and text-based notes written by the user that are meant to consolidate their learning. These *notes* are stored as text files when created. We used the CRUD-based approach to implement the basic 4 features listed in Table 1 below.

Table 1: CRUD functions/features implemented on *notes*

Creating of *Note*(s)	CreateNoteCommand
Reading of *Note*(s)	ViewNoteCommand
Updating of *Note*(s)	EditNoteCommand
Deleting of *Note*(s)	DeleteNoteCommand

4.9 GUI Feature

Upon launching the jar file, a window will pop up showing the GUI of JavaCake.

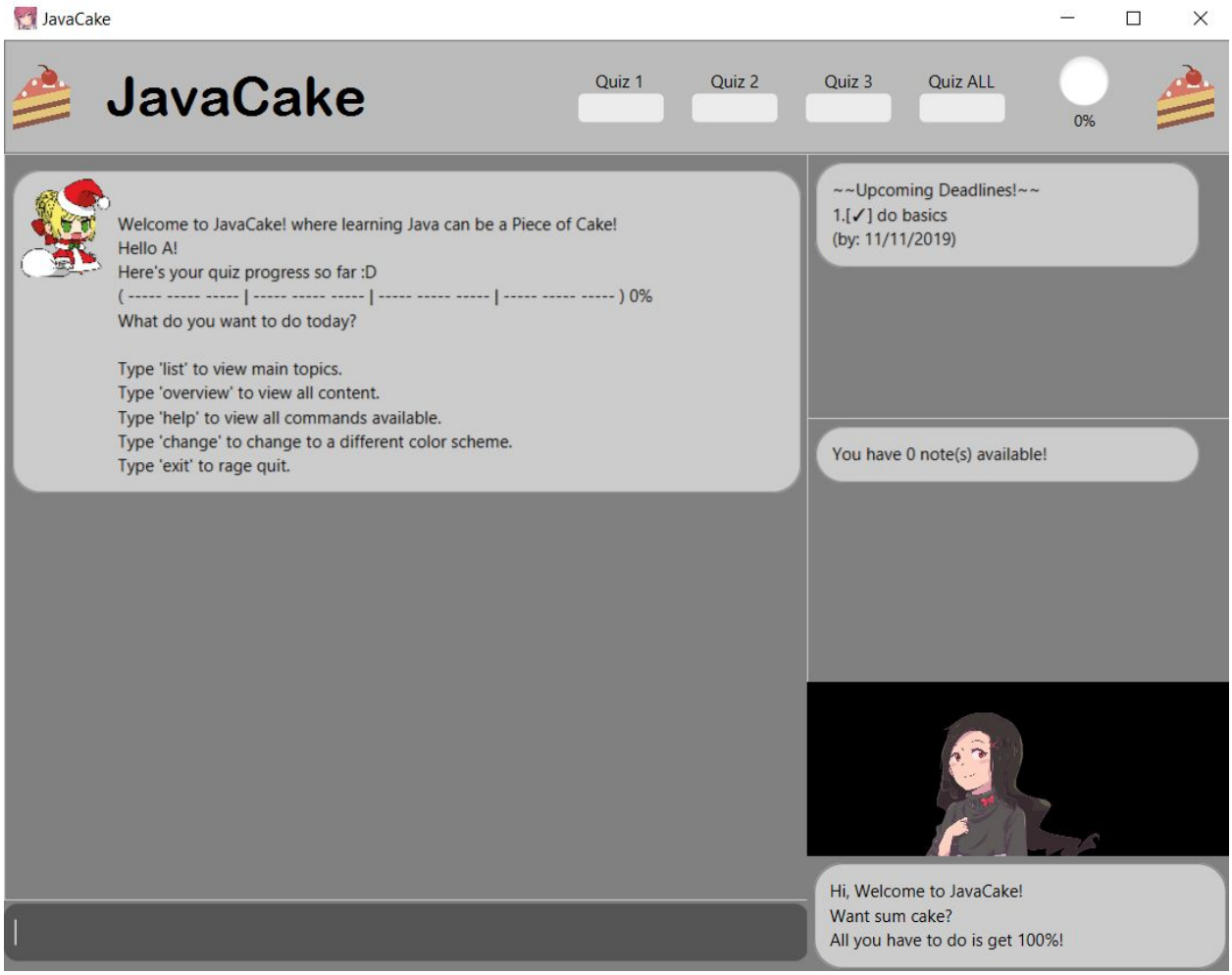

Figure 8: Graphics User Interface (GUI) for JavaCake

4.9.1 Proposed Implementation

Currently, the GUI has a total of 5 viewable interfaces, along with an additional *textfield* for users to input their commands.

Table 2: Viewable interfaces and corresponding description

View	Description
TopBar	This view contains the title of the app, JavaCake, as well as the individual and total quiz scores at the side.
ContentDialogBox	This view displays the main content and quizzes, which mainly changes according to user commands like `list` and `goto`.
DeadlineDialogBox	This view displays the list of deadlines, which is automatically updated after every deadline-related command or after the `reset` command is called.
NoteDialogBox	This view displays the list of notes, which is automatically updated after every note-related command or after the `reset` command is called.
AvatarScreen	This view displays our app's avatar. The avatar changes its animation to blink every few seconds as well as when the user gets different quiz scores. There is also an additional dialog box below which displays some motivational quotes for the user.

The diagram below depicts a simplified activity diagram on the current implementation of user input processing in the GUI. It only covers the quiz and review portion of the application, since the other features are similar in nature and only differ in the type of Boolean flags required.

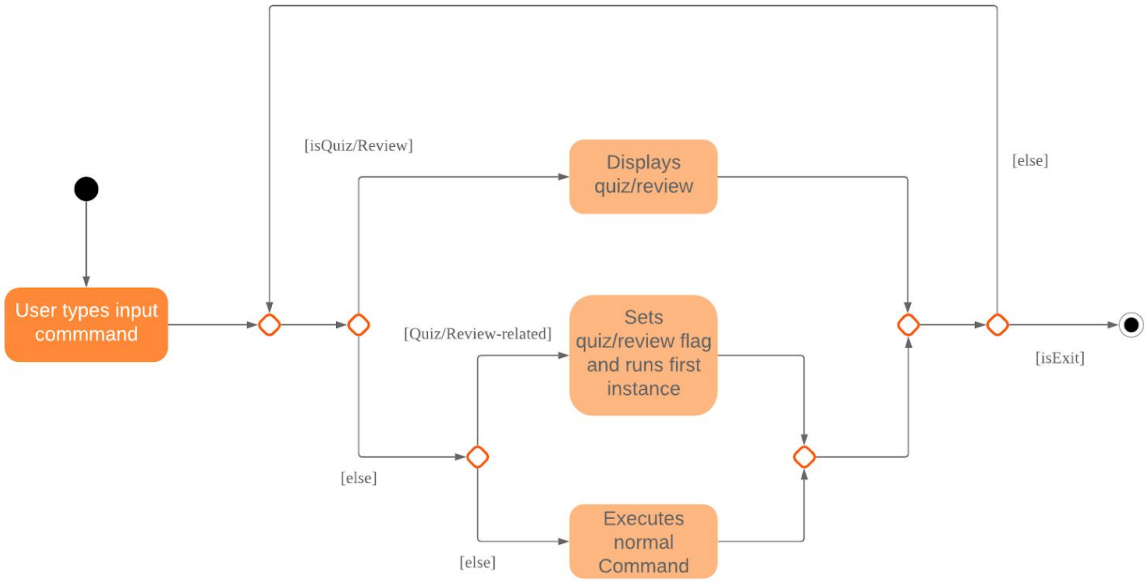

Figure 9: Simplified Activity Diagram for processing of user input in GUI

4.9.2 Design Considerations

When designing the `MainWindow#handleUserInput`, there was a dilemma on how best to parse and handle the inputs which cause the user to enter the quiz, review and other modes which block the usual input operation by the user. The display of the user quiz scores at the `TopBar` was also deliberated.

The following is a brief summary of the alternatives explored and decisions made in the design.

Table 3: Design considerations for GUI implementation

Aspect	Alternative 1	Alternative 2
User input for Quiz and Review	Implement a *textfield* swap and new methods just to handle the `QuizSession`, `ReviewSession`, and other commands that block the initial *textfield*. • **Pros:** Easy to implement • **Cons:** Will be more memory-intensive and not scalable.	Keep track of different modes of operation via boolean variables like *isQuiz*, *isReset* and such. • **Pros:** Easy to implement • **Cons:** Normal commands will usually not be executed when these boolean variables are set.

		(Current Choice) It has a lesser burden on the user's CPU and the execution of normal commands can be resolved when the order of *if-else* statements are implemented properly.
Display of Quiz Scores in `TopBar`	Create a fixed number of progress bars and progress wheel objects to display. ● **Pros:** Easy to implement ● **Cons:** Not scalable, since if another developer wants to add another quiz, they have to modify the source code directly. **(Current Choice)** For the scope of this project, we have decided to limit the number of quizzes to be only 4.	Dynamically create progress bars and progress wheel objects according to the number of Quiz text files found in the resources folder. ● **Pros:** Scalable for future implementations of additional quizzes ● **Cons:** Hard to implement and prone to errors if quiz files are not found or properly formatted.

Appendix A: Product Scope

Target user profile:

- Has a need to gain a basic grasp of Java
- Prefers desktop apps over mobile
- Can type quickly
- Prefers typing over mouse input
- Is reasonably comfortable using CLI apps
- Is goal-driven (likes setting goals for themselves)

Value proposition:
Managing the learning of Java and managing the learning progress through a desktop app which is CLI-based.

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	New user	see usage instructions	refer to instructions when I forget how to use the App
* * *	User and newbie to programming	see table-of-content	view the different topics required to gain a basic understanding of Java
* * *	Goal-oriented user	add a new deadline	pace myself when learning Java
* * *	User	delete a deadline	remove entries that I no longer need
* * *	User	mark a deadline as done	keep track of topics I have completed
* * *	User	check reminders for topics to read	be reminded of what topics to read before a previously set deadline
* * *	User	do a quiz at the end of a topic	reinforce my learning

* * *	User	hard reset my profile	start my progress over from scratch
* *	User	find a topic by name	locate details of topics without having to go through the entire list
* *	User who cannot absorb the content fast	do quizzes of varying difficulties	challenge myself more when I'm confident of the content
* *	User	Have stars/visual aids to represent my progression	Have an easier time to track my progress
* *	User	ability to know the progress of individual subtopics	I can choose to focus on the topics that require more attention.
* *	User who is an auditory learner	dynamic audio content	feel more engaged with the app
*	User with many deadlines in the reminders	sort deadlines by date	locate a deadline easily

*	User with uptight preferences	Choose between light and dark mode for the GUI	have more flexibility in viewing the content
*	User that may find the content too boring	have motivational quotes or jokes	make the learning process more interesting
*	User	have a progression system which is dynamic (can see obvious changes in the Ui)	have an intuitive feel when viewing their progress throughout the app
*	User	have different music for different levels	Be more engaged with the quiz portions of Java Cake
*	User	know when my last login is	keep track of my progression in completing Java Cake

Appendix C: Use Cases

(For all use cases below, the System is the `Cake` and the Actor is the `user`, unless specified otherwise)

1) Use case: Go to topics

MSS

1. User requests to list topics
2. Cake shows topics (in format 1. X\n 2. Y\n…)
3. User types the topic number e.g. `1`
4. Cake shows subtopics within that topic (in format 1. X\n 2. Y\n…)
5. User types the sub-topic number e.g. `1`
6. Cake shows the content in the sub-topic
   Use case ends.

Extensions

- 3a. If user types in `goto 1.1`, user can immediately jump to sub-topic content
    - 3a1. If no sub-topic present, Cake shows error message
      Use case resumes at step 5.

2) Use case: Check progress

MSS

1. *User finishes topic/quiz*
2. Cake shows progress bar ([# # # # _ _ _ _])
   Use case ends.

Extensions

- 2a. If user wants to know the progress of each individual topic, the user can request to list topics, each topic will have the progress represented by the number of stars which ranges from 0 to 3.
  Use case ends.

3) Use case: Do quiz from sub-topic

MSS

1. *User requests for sub-topic list*
2. User types in the index of the quiz denoted as “Test Yourself!” in the list e.g. `goto 4`
3. User chooses the desired difficulty level
4. Cake launches quiz questions
5. User answers the questions individually
6. Cake displays the results for the quiz
   Use case ends.

Extensions

- 4a. User types input not in the form of an integer
    - 4a1. Cake shows error message
      Use case resumes at step 4.
- 4b. User types wrong answer
    - 4b1. Cake shows "Wrong Answer" and proceeds to the next question.
      Use case ends

4) Use case: Set deadline for topics/subtopics to read

MSS

1. User intends to add a new deadline
2. User types input `deadline <TASK/> /by <DATE/>`
3. Cake shows confirmation message (adds to list of deadlines)
   Use case ends.

Extensions

- 4a. If user launches program again, the deadlines for each topic are shown in most recent deadline order.

5) Use case: View reminders of deadlines for topics to read

MSS

1. User requests to view reminders
2. Cake shows a list of topic with deadlines in most recent deadline order
   Use case ends.

Extensions

- 2a. If there are no deadlines, Cake will inform user that there is nothing to show.
  Use case ends.

6) Use case: Writing personal notes on the topic

MSS

1. User wants to write notes at a specific topic
2. User requests to write notes by typing `write` command
3. Cake receives notes from user
4. Cake saves the notes.

Extensions

- 1a. If user has written notes before, Cake will automatically display the notes so that user can copy, paste and edit accordingly.
  Use case ends.

7) Use case: Finding a specific content *{for v2.0}*

MSS

1. User requests to view a specific content piece
   `view <CONTENT/>`
2. Cake shows list of subtopics, if applicable, to user.

Extensions

2a. If there are no subtopics that matches, Cake will inform user that there is nothing to show.
Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be quick in outputting content for the user to read and quizzes for user to play through.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Storage size requirement for program to run, since there is a need to store textfiles.

Appendix E: Instructions for Manual Testing

Given below are instructions to test the app manually.

These instructions only provide a starting point for testers to work on, and is not exhaustive. Exploratory testing is recommended.

E.1. Launch and Saving notes, progress and deadlines

1. Initial launch
    i. Download the jar(Java ARchive) file and copy into an empty folder
    ii. Double-click the jar file
    Expected: Shows the GUI with a set of sample notes and deadlines.
2. Saving notes
    i. Create a note with a filename of your choice and close the application
    ii. Re-launch the app by double-clicking the jar file.
    Expected: The notes should have been saved and can be viewed again by using the `editnote` command.

3. Saving progress
    i. Complete some of the quizzes, to gain more progress and close the application.
    ii. Re-launch the app
    Expected: The progress should have been saved with the updated score.
4. Saving deadline
    i. Create a deadline.
    ii. Close the application.
    iii. Re-launch the application
    Expected: The deadlines should have been saved and can be viewed in the top left hand

E.2. Resetting progress

1. Use the `reset` command to delete all progress, deadlines and notes.
2. Close the application and relaunch it.
   Expected: All progress is now zero, no deadlines or notes are saved.

E.3. Missing and Corrupted files

1. Dealing with missing data files
    a. Delete the savefile.txt file in the file path data/save/savefile.txt.
    Expected: JavaCake will preload with default *BakaTester* profile.
    b. Delete the deadline.txt file in the file path data/tasks/deadline.txt.
    Expected: JavaCake will preload with default *BakaTester*'s list of deadlines.
    c. Delete the color.txt file in the file path data/colorconfig/color.txt.
    Expected: JavaCake will default to original light theme.
2. Dealing with corrupted data files (files not following proper format)
    a. Modifying savefile.txt. The following will cause JavaCake error screen to be displayed.
        i. Removing one of the lines containing numbers.
        ii. Changing the 4 topmost numbers to a value less than 0 or greater than 15, or replacing with a non-number.

iii. Changing the 12 bottommost numbers to a value less than 0 or greater than 5, or replacing with a non-number.
iv. Causing the 12 bottommost numbers to not match the 4 topmost numbers.

b. Modifying deadline.txt. The following will cause JavaCake error screen to be displayed.
    i. Causing the first parameter to not match with “D” exactly.
    ii. Causing the second parameter to not match with “✓” or “✗” exactly.
    iii. Causing the third/task parameter to exceed 39 characters.
    iv. Causing the fourth/date parameter to not adhere to the date format specified in the User Guide Section 3.7.
    v. Adding an extra “|” in at the start or in between any of the parameters in the same line.
    vi. Adding more than one additional blank line after the last parameter line.
    vii. Adding blank line(s) before the first parameter line or in between.
c. Modifying color.txt
    i. Replace the current text with any string or remove it entirely. Expected: JavaCake will default to original light theme.

Appendix F: Glossary

Mainstream OS: Windows, Linux, Unix, OS-X