**Kwok Kuin Ek Jeremy – Project Portfolio for Dr. Duke**

**About the Project**

My team of 4 software engineering students and I were tasked with enhancing a basic command line interface desktop application named Duke for our Software Engineering project. We chose to morph it into a GUI-based personal assistant for doctors to manage their patients' data named Dr. Duke. The morphed application provides house officers with a streamlined notebook environment, which allows them to concentrate on quickly and accurately recording patient data in a standard format.

My role was firstly to design and maintain the data model representing the Patients and their relevant information. I was also responsible for the `find` feature. The following sections illustrate these enhancements in more detail, as well as the relevant documentation I have added to the user and developer guides in relation to these enhancements.

**Summary of contributions**

This section shows a summary of my coding, documentation and other helpful contributions to the team project.

Data Model: I am in charge of the classes which represented information in Dr. Duke.

- What it does: The data model store information about Patients or pieces of information about the Patients such as their diagnosis or medical results.
- Justification: Information stored by hospitals can be classified into several kinds such as the results of tests, medicine prescribed, observations about the Patients. The different types of information also are inherently similar to each other in that they may store the Name of the piece of information, and maybe a short Description. Their similarities can be attributes of abstract objects. Structuring and classifying them as objects can help in data manipulation and organisation. This also allows us to record patient data in a standard format.
- Highlights: This data model also allows us to sort our data and present the most important and relevant information to the user first. An in-depth analysis of design alternatives was necessary to determine what was the best way of storing the collection of data objects. We also considered how the data model would update the UI. The implementation was also challenging because modelling the data can affect how the GUI presents information to the user. We also have to tie different classes together as one piece of information belongs to only one patient.

Enhancement added: I added the ability to search for information within the entire system

- What it does: The find command allows the user to search for any piece of information within the system. It additionally searches subsets of the system depending on where the search is initiated or if an attribute is specified.
- Justification: When more users are added to the system and the amount of data to navigate through grows or when the user wishes to directly access a piece of data. A search function can make the navigation easier.
- Highlights: This enhancement works well to allow the user to quickly navigate to a piece of data. It also be used as a helper function for disambiguation of user input when we are unsure of what the user has entered by narrow down the possible options based on existing data in the system and presenting to the user for selection.
- Credits: Teammate John Khoo for general refactoring and switching to `contains` method for more target searching.

**Code contributed:**

Please click these links to see a sample of my code.

Functional code:

- https://github.com/AY1920S1-CS2113-T14-1/main/blob/master/src/main/java/duke/data/DukeData.java
- https://github.com/AY1920S1-CS2113-T14-1/main/blob/master/src/main/java/duke/ui/window/SearchContextWindow.java

Test code:

- https://github.com/AY1920S1-CS2113-T14-1/main/blob/master/src/test/java/tests/ImpressionTest.java
- https://github.com/AY1920S1-CS2113-T14-1/main/blob/master/src/test/java/tests/SearchTest.java

**Other contributions:**

- Enhancements to existing features:
    - Extended the GUI for Impression and Search Windows #253
    - Fixed Gson storage when deserialising abstract classes #244
    - Changed reference to parent from String to transient DukeObject #229
    - General bug fixes #182, #243, #270
- Documentation:
    - Generated updated UML diagrams.

- Community:
    - Review Pull Requests (with non-trivial review comments): #236, #235
    - Reported bugs for other teams in class

**Contributions to the User Guide**

We had to add information into the User Guide with instructions for the features we morphed. The following is an excerpt from our User Guide, showing additions made for the find feature.

#### 4.1.3. `find` - Find items matching certain criteria

Format: `find ["<search string>"] [type]`
`[type]` → ( `-p[atient]` | `-im[pression]` | `-e[vidence]` | `-t[reatment]` )

Display a list of all Patients, Impressions, Treatments, and Evidences matching the criteria specified in the search. If none of the `type` switches are used, all types of objects listed above will be shown. If at least one of them is listed, only objects whose type is used as a switch will be listed.

**Example**

**find** "cough" (This command searches for Patients only)
**find** "John" `-p` (This command searches for Patients only)
**find** "aspirin" `-im` `-t` (This command searches for Impressions and Treatments)

**Contributions to the Developer Guide**

The following shows my additions to the Developer Guide for the Data Model and find feature.

Figure 2. Class Diagram
The Class Diagram shown above describes the relationship among the different data classes invloved in storing information used in `Dr. Duke`. The class is named after the object it represents. All objects extend the `DukeObject` abstract class, which stores basic information to identify the object and its parent.

The `DukeObject` class specifies several abstract functions crucial for the UI to access. All `DukeObjects` also have a `parent` DukeObject which is transient and may be null. This is to facilitate storing in Gson and allow objects to reference their parent if needed. A `String` representation of `DukeObjects` can be obtained using the `toString` and `toReportString` methods.

#### 3.3.1. Patient

Figure 3. Class Diagram
The class diagram shown above shows the `Patient` class and how it is stored.
Patients entered into our system are stored as `Patient` objects in our `PatientData` object. This can be converted to Gson easily after accounting for abstract objects. All patients may have `Impressions` associated with them which are created by the Doctor's impression of a Patient. This is supported with `DukeData` objects as evidences or treatments.

The `Patient` object should provide the following functionality: * Input validation to ensure it stores valid input * Sorting of Impressions ** Currently, Primary Impressions are also stored at the head of the `impressions` list. If a future metric for assessing importance of impressions are suggested by users, it can be added here as well. * Filtered list of important critical `DukeData` * Filtered list of uncompleted `Treatments` which require follow ups * Quick notes on the Patient

### 3.3.2. Impression

Impressions are what a doctor diagnoses a Patient of. Each impression may be supported by Evidences and associated with Treatments.

The `Impression` object should provide the following functionality: * Input validation * Sorting of Treatments **High priorities are the first metric** Incomplete status requiring follow up is the second metric * Sorting of Evidences ** High priorities are the first metric

### 3.3.3. DukeData

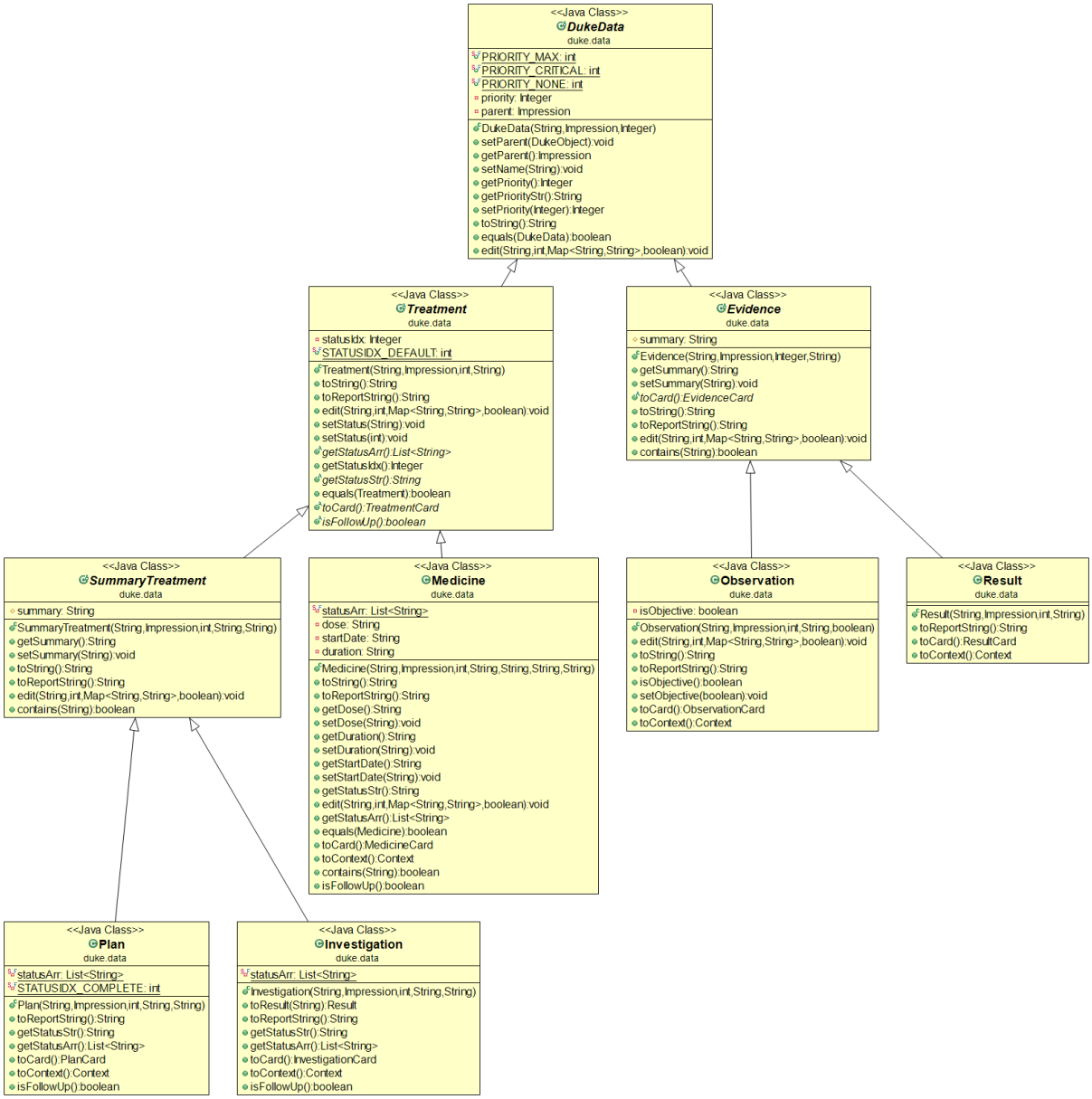



Figure 4. Class Diagram

The diagram above shows the DukeData class and its concrete implementations. The `DukeData` objects represent evidence and treatment recorded by the doctor.

### 3.3.4. Extension

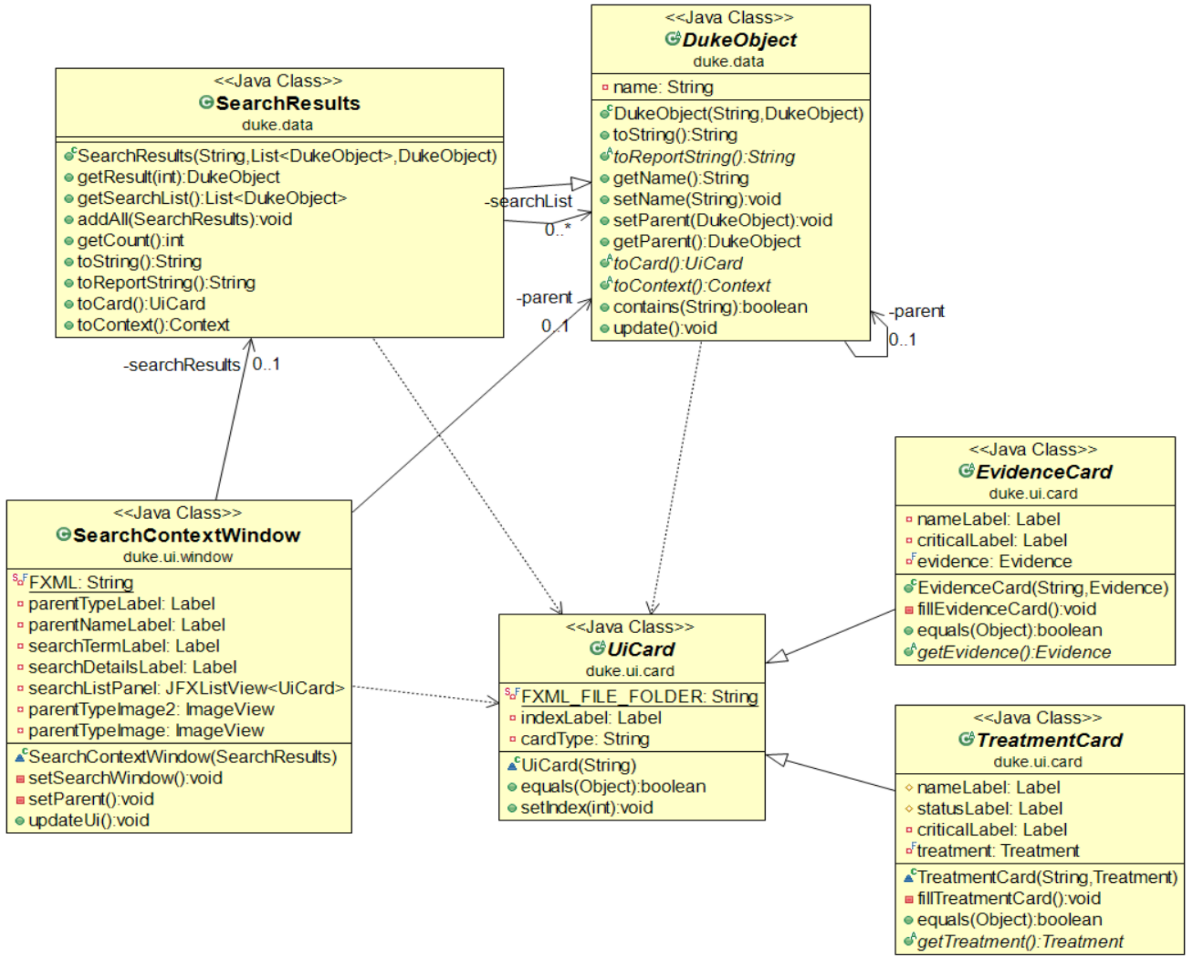
To define new forms of `DukeData` representing information on the Patient, extend `DukeData` or its abstract subclasses To define other types of data, extend `DukeObject`.

If the class is abstract and needs to be stored, an adaptor implementing `JsonSerializer` and `JsonDeserializer` for it needs to be created for Gson storage. Any circular referencing must be stored as transient but must be reinitialised at

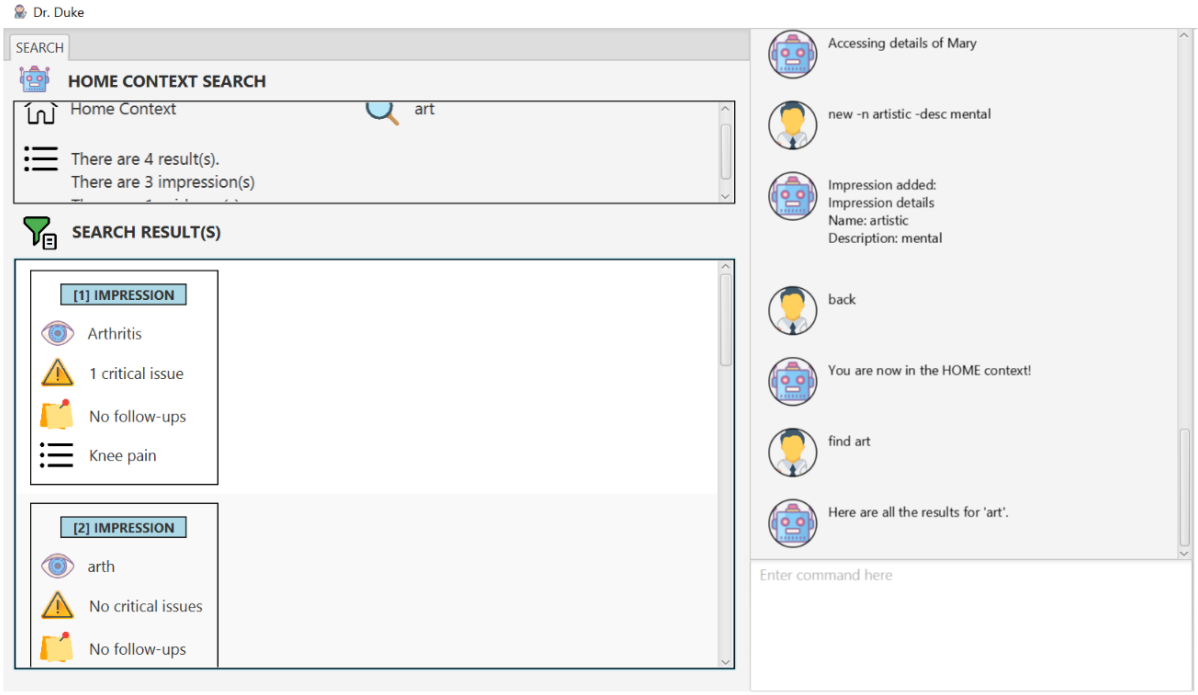
launch.

| Note | By convention, we store invalid values instead of null values to prevent nullptr exceptions. If there are attributes that may be null, consider returning an empty object instead. E.g. for `String`, return `""`. |
|---|---|

### 4.3. Search all records [KWOK KUIN EK JEREMY]




The above diagram shows the information a search result will store and the SearchContext its displayed in.




The picture above is an example of a `find` command.

#### 4.3.1. Rationale

Dr. Duke aims to assist House Officers in quick, accurate recording and retrieval of patient data required to provide efficient care. When more patients are added to the system and the system grows in size or the user want to directly access a piece of nested data we need a method to directly assess the data. Therefore, it makes sense to have a search function to search through the entire system or a subset of the system. Hence, a find feature is essential for users to quickly locate data or for disambiguation when it is unclear what the user wants to narrow down the possible options based on existing data in the system.

- Reduce the time taken for the user to enter details of the Patient and navigate in the system.
- Search a subset of the system or only for data of a certain type.

#### 4.3.2. Proposed Implementation

The search mechanism is facilitated by two main functions, namely `contains` and `find`.

`contains` is a method every concrete component of the data model has. It is specific to the type of information stored by the class. In our case, this facilitates searching for information by representing relevant attributes in String form and checking if the search term is contained within.

`find` method is included in every class that stores ArrayLists of other objects. It searches if an object contains a search term by utilising the `contains` method. Different flavours of the `find` function is post fixed with information on what its purpose is. For example, `findImpressionsByName` searches only the `name` field of `Impression` objects. The master `find` function is `searchAll` which searches through all related information from a particular object down.

Given below is an example usage scenario and how the search mechanism behaves at each step.

Step 1: The user launches the application and navigates to a particular patient context for example, `John`. The `TextField` in the `CommandWindow` is blank, and the context is `Patient:John`. The user wishes to search `John` for a particular piece of information e.g. Fever (a sample valid command syntax is `find Fever`).

Step 2: The find method will be called and all data related to the Patient will be searched for `Fever`, It will display the results in a new Context containing all impressions where `John` had `Fever` in a separate window

Step 3: The user can then select a particular impression and review the information or change the information if desired.

**Design Considerations**

When designing the data model, we had to make decisions on how best to represent the information and how to update the GUI with the information.

| Aspect | Alternative 1 | Alternative 2 |
|---|---|---|
| How to store collections of objects | Utilise ArrayList to store the object.<br><br>Pros: Can easily reference object using the index. Can sort the objects easily as well.<br>Cons: Cannot directly reference the object by their names.<br><br>This alternative was chosen.<br>Our original implementation used HashMaps, but we found that it was more difficult to reference an item in the HashMap using an index over reference an item in an ArrayList by its name attribute. An ArrayList can also be sorted easily to push high priority data to the front. | Utilise HashMaps to store the object.<br><br>Pros: Can easily reference object by name if the user keys in a name.<br>Cons: Complex to reference by index and sort.<br><br>We considered using linked HashMaps to overcome the indexing issue but we decided against it as there was no real benefit to the HashMap considering the user may enter the name wrongly as well. |
| How to update the User Interface | MVC Design pattern<br><br>Pros: Can easily update UI<br>Cons: Costly to update UI, slower.<br><br>We chose this implementation as using an Observer Pattern complicated storage and updating any attribute meant that removal and reinsertion is necessary for the observer to observe the change. | Observer Pattern<br><br>Pros: Can quickly update UI<br>Cons: Observables cannot be directly stored in Gson, a non-observable version needs mirror the observable for storage |
| Hierarchy of Objects | Use abstraction. | Use concrete implementations |

| | | |
|---|---|---|
| | Pros: Store collections of abstract objects. There are similarities in attributes and methods of different objects.<br>Cons: Abstract objects cannot be deserialised trivially from json.<br><br>We chose this as it reduces code needed and makes the code base more maintainable and understandable. | Pro: Simpler to implement<br>Cons: Difficult to maintain |

When designing the find functionality, we had to make decisions on how best to search, what information to search for.

| Aspect | Alternative 1 | Alternative 2 |
|---|---|---|
| What attributes to search | Utilise our String representations of objects created by the toString method. This will include the children of the object as well.<br><br>Pros: Can find a parent object if the user knows something about its child object.<br><br>Cons: There may be repeated terms such as “Name: “. Looking at certain attributes such as age may not be useful. | Utilise contains to search.<br><br>Pros: Can specify more relevant attributes.<br>Cons: Cannot include information on children.<br><br>We chose to use contains to have finer control over what attributes to look at and to prevent the possibility of overflow. |