Project Portfolio for Dr. Duke

By: *John Khoo* (@aquohn)

1. Project Description

Dr. Duke is a cross-platform Java app designed to help medical house officers manage their patients' medical information more effectively. It uses a command line input with graphical feedback to allow users to quickly and easily input and retrieve data. It was designed a team consisting of myself and three other software engineering students at the National University of Singapore. It was morphed from an original codebase primarily written by myself for *Duke*, a personal assistant app. This is the home page of Dr. Duke, which users will see when they first open the app:

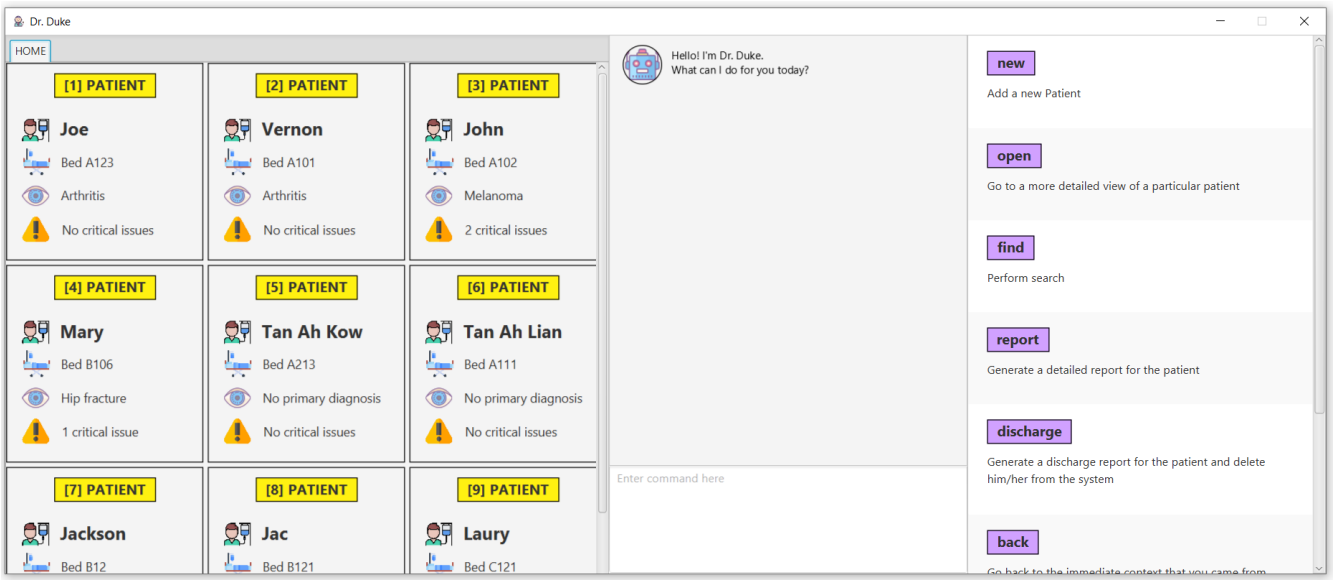


Figure 1. The UI of Dr. Duke

Do note the following conventions in this document:

- `Mark-up` is used to indicate a code literal. This can be a `method()`, a `Class`, an `ENUM_VALUE`, or `literal input`. It is generally used when discussing concrete implementation details, as opposed to abstract ideas, e.g. "patient" refers to an actual human patient, while `Patient` refers to the Patient class, or an object thereof, in **Dr. Duke**.
    - A method may be specified with its arguments: `method(String strArg, int intArg)` in order to let the reader know what arguments it takes, or to differentiate between two methods with the same name but different arguments. However, unless explicitly stated, a `method()` specified without its arguments **does not** imply that the method takes no arguments. A method may be referred to without arguments after being introduced with its arguments for brevity.
- *Italics* are used when introducing a new term or concept.

NOTE	This box draws attention to quirk or caveat that may not be obvious.

2. Contributions

My role in the project was focused on parser and command logic: translating input from the user into commands to manipulate the app's data. I also developed and implemented several other features, notably a system for allowing the user to disambiguate the intended target of a command, and contributed to project planning and management and the class community. My code contributions can be found here, and further details on my implemented features can be found in the User and Developer Guide excerpts below.

2.1. Primary Features

2.1.1. Parser

I wrote a simple parser for extracting commands, arguments, switches and switch arguments from a user's input and translating them into `Command` objects, which can be executed to perfrom the user's desired action. In order for my teammates to easily extend the system by creating more commands, I designed a system for specifying a command's operation, and the switches required for it, such that my parser would be able to automatically check the user's input to see if it matches the requirements of the command. We had decided to write our own parser instead of using a library as it was a relatively straightforward task and gave us much more control over the formats of our commands, and for providing feedback to the user based on his input. I wrote the `Parser`, `ArgParser`, `Command`, `CommandSpec`, `ArgCommand` and `ArgSpec` classes that constitute the parser system, the use of which is detailed in the relevant section below.

2.1.2. Object Disambiguation

I created a system to allow users to execute commands even if their input was ambiguous (e.g. `"open jo"` when there are two patients named "John" and "Joe" being managed).The command will result in a search window being opened, which holds all possible matches for the user's command in an indexed list, and on selecting an object, the command will be executed on an object.

This feature is designed to help the users navigate through the app more fluidly. Users can type in just snippets of names and have the relevant data quickly brought up, allowing them to quickly complete the execution of the command with minimal interruption to their workflow. It is also more natural that our users will think in terms of names, rather than the index of an object in a list. This improves user experience and input speed.

This system builds on and generalises the `find` command, allowing for the execution of any command which may need user input. Unlike the other commands, which followed a simple "input-parse-execute" model, commands with disambiguation require control to be returned to the UI to get new user input before completing. Therefore, the state of the initial input needed to be stored before obtaining disambiguating input from the user. I wrote the `ObjCommand`, `ObjSpec`, `SearchResults`, and `SearhSpec` classes, and the `search()` and `executeQueuedCmd()` methods in `DukeCore` which are essential to this feature. To support it, I also wrote the `ImpressionObjSpec`, `PatientObjSpec`, and `HomeObjSpec` classes, and the `findByName()`, `find()` and `searchAll()` methods in `Patient` and `PatientData`, the use of which is detailed in the relevant section below.

2.2. Further Features

I additionally developed the following features:

- Commands in the `IMPRESSION` context: `new`, `edit`, `delete`, `priority`, `primary`, `status`, `result`, `open`, `move` (particularly difficult as `move` potentially needs to disambiguate twice)
- `open` and `edit` commands in the `PATIENT` context, and `new` command in the `HOME` context
- All commands in the various `Treatment` and `Evidence` contexts
- Refactoring of helper functions into `PatientUtils`, `HomeUtils`, `ImpressionUtils`, `CommandUtils`
- General refactoring in `Impression` and `Patient`:
  - Add methods `getFollowUpCountStr()` and `getCriticalCountStr()` to produce descriptor strings instead of getting and processing raw data
  - `equals()` method to perform comparisons for tests
- An autocorrect system that was not able to be integrated with the codebase in time.

2.3. Additional Contributions

I wrote almost the entirety of the first draft of the User Guide, which served as an informal specification for our project.

My contributions also include:

- Contributions to tests:
  - An abstract class `CommandTest` that performs the necessary setup for testing more complex features to simplify test writing
  - Tests: `HomeCommandTest`, `PatientCommandTest`, `ImpressionCommandTest`, `ObjCommandTest`, `ImpressionMoveTest`
- Team issue tracker:
  - Organising user stories and epics (e.g. #48, #49, #50, #47, #43, #39, #38)
  - Starting discussions and proposing features on the issue tracker (e.g. #28, #31)
- Participating in forum
  - Reporting website issues and starting a tracker issue for them
  - Contributing a tip to the forum
  - Asking questions #13, #15, #25, #43, #44, #58, #63 and #71 to help clarify doubts.
  - Contributing to discussions #10, #12, #30, #33, #49, #61

3. User Guide Excerpts

3.1. Commands

Each command consists of a *name*, an *argument* (information supplied to a command to tell it where or how to act), and some number of *switches* (settings for a command that modify its behavior). Switches can also have arguments.

For example, `discharge "John Doe" -sum "John Doe was discharged on 9 October 2019 at 3:54 pm".` represents the command `discharge` with the argument `"John Doe"`, modified by the switch `-sum` (meaning "summary"), which has the switch argument `"John Doe was discharged on 9 October 2019 at 3:54 pm."`.

Only a few characters have a special meaning when you are giving **Dr. Duke** commands. These are `\`, `-`, `<Space>`/`<Newline>`, and `<Enter>`:

- `\` → Escape character: the special character after a backslash loses its special meaning. To type a literal backslash, you need to escape the backslash: `Hello\\World` becomes `Hello\World`.
- `-` → Indicates the start of a switch. Must immediately be followed by an argument, if the switch requries one.
- `<Space>`/`<Newline>` → A switch is separated from its argument by a space or a newline.
- `<Enter>` → This sends a command to Dr. Duke. `<Shift-Enter>` would insert a new line.
- `<Page Up>` and `<Page Down>` → Access the input history.

The documentation below uses the following notation to describe the structure of the commands recognised:

- `<>` → input of the type specified
- `[]` → optional element of a command
- `(a | b | c)` → input that can be one of several possible options, in this case `a`, `b`, or `c`
- `"<>"` → string, must be surrounded by quotes
- `[]*` → 0 or more copies of the contents of `[]`, separated by spaces

The following italicised words refer to specific repeated patterns:

- *patient_id* → `(<patient□s index in list> | "<search string>" | -b[ed] "<bed number>")`
- *string_or_idx* → `(<index> | "<search string>")`
  - `<index>` refers to the numerical position assigned to the object in the displayed list.
  - A `<search string>` will be searched for in the names of all relevant objects.

Parsing rules:

- If a `"<search string>"` is given, the user will be presented with a window of search results that have names matching the search string (ignoring case), if the object to which it is referring is ambiguous. Refer to the open command in the Home context for a concrete example. Within the search result window, the user may enter the index of an object to select it.
- Switches can be in any order.

- Switches with optional parts of their names can be recognised with any portion of the optional part. E.g. `-crit` matches `-c[ritical]`.
- If it is ambiguous whether an argument is for the command itself, or one of the switches, it will be presumed to belong to the switch.

3.2. Home

3.2.1. `open` - Go to a more detailed view of a particular Patient

Format: `open patient_id [-im[pression]]`

If opening by <*search string*>, Patients will be searched through by **name**, and results presented and selected as per the `find` command. For example, if the user manages three Patients named "John", "Jack" and "Joe", `open jo` will open a search result context with "John" and "Joe" but not "Jack", even if "Jack" has, e.g. an address of "42 Joy Street" (because his name does not contain "jo").

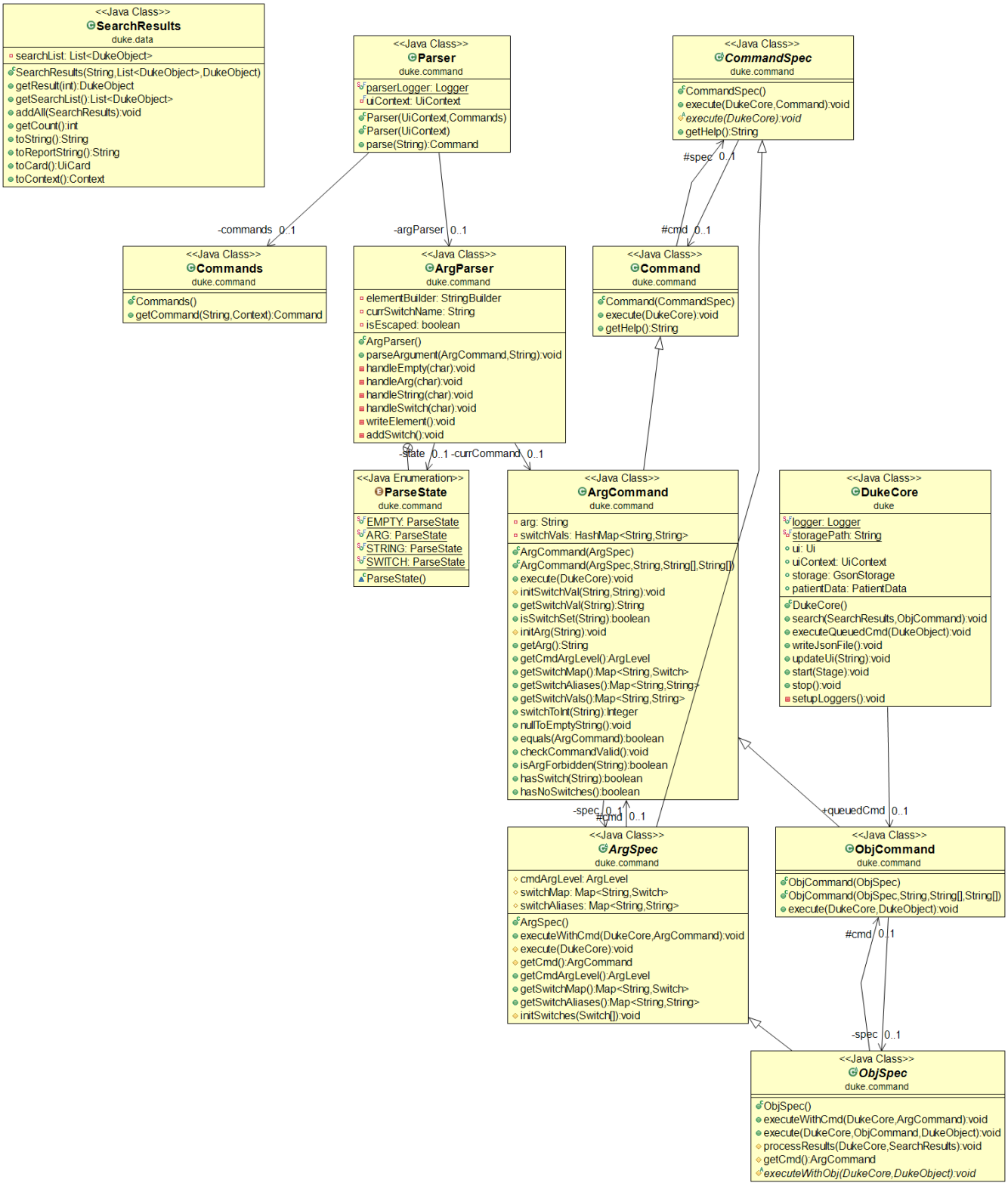
If the optional `-im[pression]` switch is provided, access the primary diagnosis for that particular Patient.

Example

open 1 `-impress` (This command opens the first Patient in the list and goes to his primary Impression)
open `-b` "C210" (This command opens the Patient with the specified bed number)
open "Duke" (This command searches for a Patient named "Duke" and opens his context, or displays all Patients with "duke" in their names if there is more than one)

4. Developer Guide Excerpts

4.1. Parser Logic [JOHN CUTHBERT KHOO TENG FONG]



This class diagram describes the relationships between the various core classes involved in parsing the user's input into `Command` s. The two highest-level components are the `Parser` and the `Executor`, both members of the `CommandWindow`. They begin parsing when the user enters some input through the `CommandWindow`.

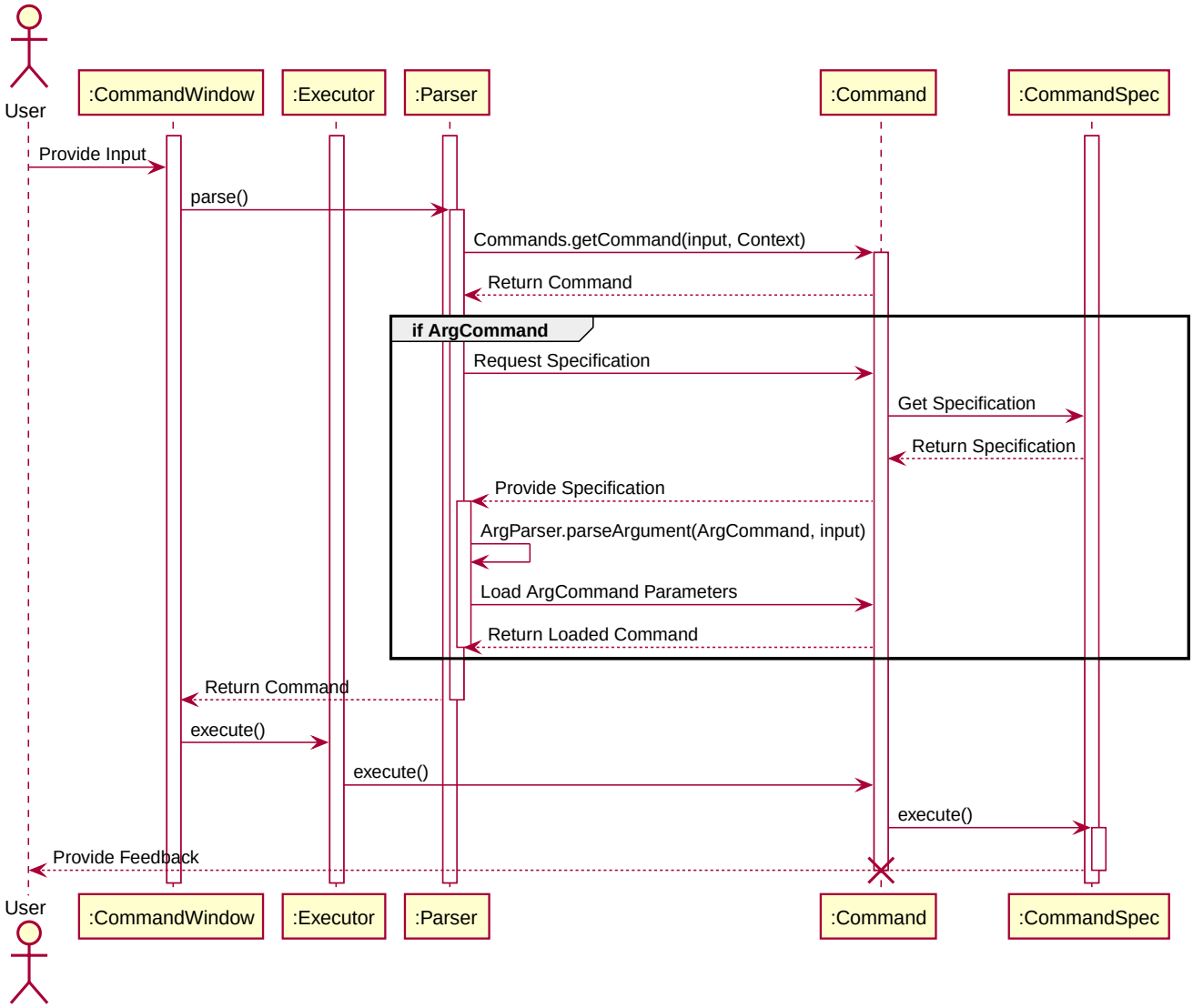
The first word (delimited by a space or newline) of the user's input is the *command name*. All commands extend the `Command` class, which provides enough functionality for basic commands consisting of a single word. The operation of the `Command` is specified in the `CommandSpec` singleton it is constructed with, via the `execute` method.

The mapping from the command name to the `CommandSpec` should be created in the `Commands` class, which is loaded by the default `Parser` constructor. A `Parser` can also be constructed with a subclass of `Commands` to specify a different set of commands.

The `Commands` class has a single function `getCommand()`, which takes, as arguments, a String that should uniquely identify the requested `CommandSpec` within a particular `Context`, and a `Context` enum representing the context from which `getCommand()` was called. It then returns a new instance of the `Command`, constructing it with the required `CommandSpec`. The `Parser` will supply the command name and the `context` field in the `DukeCore` instance to the `getCommand()` method in its `Commands` instance.

If the command requires an argument, the `ArgParser` object in the `Parser` will parse the rest of the input to determine the argument of the command, the switches supplied to it, and the arguments of the switches, and will set these parameters in the `Command`. Finally, after the `Command` has been constructed (and loaded with parameters if necessary), it is returned to the `CommandWindow`. The `Executor` class will then call the `execute()` method of the command, supplying the `DukeCore` object to the `Command`, which will allow it to execute its operations.

This transaction is modelled by the following sequence diagram:



4.1.1. `Command` s and `CommandSpec` s

As seen in the class diagram above, `Command` has a subclass `ArgCommand`, which has a subclass `ObjCommand`. Parallel to them are the abstract `CommandSpec`, `ArgSpec` and `ObjSpec` classes, which follow a similar inheritance structure.

If a command has no arguments, it can be represented by a `Command` configured with a `CommandSpec` object. If it takes any arguments, it requires an `ArgCommand` and an `ArgSpec`. And finally, in the special case of commands where user input may be ambiguous, an `ObjCommand` and `ObjSpec` is used. The discussion of `ObjCommand` s and how they facilitate disambiguation is left to the respective section.

Each `CommandSpec` is a singleton, which defines an abstract `getSpec()` method. This method is required to be implemented in its children, providing a means of enforcing the singleton pattern. For an `ArgSpec`, the private constructor sets the parameters of the `ArgCommand`: `cmdArgLevel` (an `ArgLevel` enum indicating whether an argument for the command is necessary, optional, or forbidden) and the data structures `switchMap` and `switchAliases`, generated by the `switchInit()` function. The `switchInit()` function takes a vararg of `Switch` objects, which should specify the switches for the particular `ArgSpec`. These parameters will be provided to the `ArgParser`, which will use them to parse the user's input.

`switchMap` maps the full name of a switch to a `Switch` object, describing its properties, and `switchAliases` maps *aliases* to the full name of the switch they represent. An alias is a string that, when provided by the user as a switch, is recognised as a specific switch. For example, for the switch `investigation` (given as `-i[nv(x|estigation)]` in the User Guide) has the following aliases:

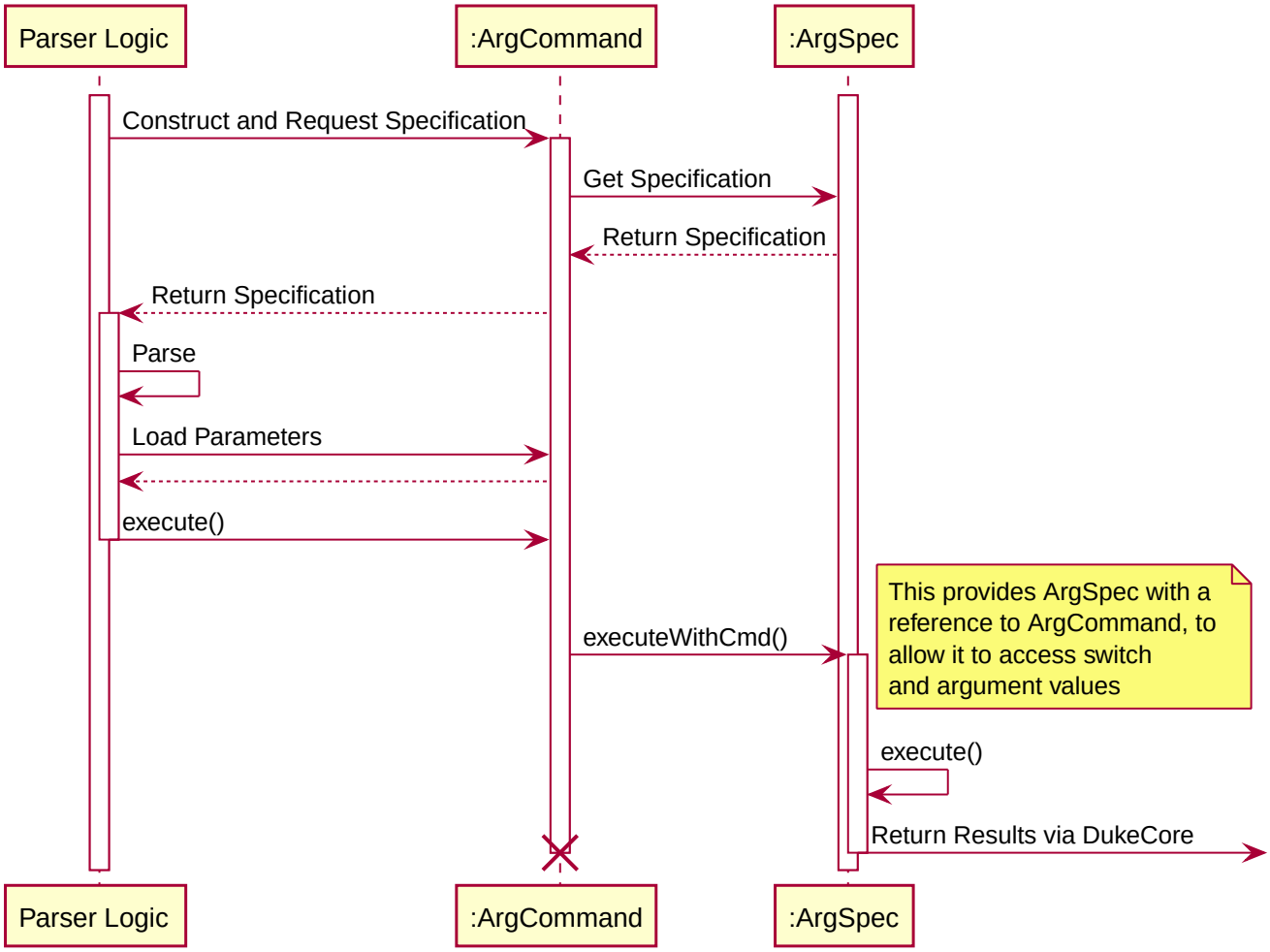
- `i`, `in`, `inv`, `invx`, `inve`, `inves`, `invest`, `investi`, `investig`, `investiga`, `investigat`, `investigati`, `investigatio`, `investigation`

As this would be very tedious to list manually, it is automatically generated by the `switchInit()` function, using the data in the `Switch` objects provided to it. Observe that almost all these aliases are prefixes of the word `investigation`, with the shortest being `i`. This follows from the requirement that the switch can be recognised as long as the user has input enough characters for it to be unambiguous. Let `i` in this example be the *root*, the shortest unambiguous part of the full name of the switch. Then, every prefix of the word `investigation` starting from the root is an alias of the switch `investigation`. All aliases of this form are generated by a loop in `switchInit()`, from the root and the full name in the `Switch` object. Any additional aliases can be supplied via the `aliases` vararg in the `Switch` constructor. Refer to the Javadoc of `Switch` for further details on its fields.

Switch and argument values identified by `ArgParser` are loaded into the `ArgCommand` using the `initArg()` and `initSwitchVal` methods. These values are then accessed by the `ArgSpec` from the `ArgCommand` with the `getSwitchVal()` method, which takes the name of a switch, as a String argument, and returns a String containing the argument supplied for the switch, and `getArg()`;

When executing a command, the `Command` 's `execute()` method is called. In a base `Command`, this would directly call the `execute()` method of the `CommandSpec`. For an `ArgCommand`, this would instead call `executeWithCmd()` on the `ArgSpec`, supplying the command to it. This stores a reference to the calling command in the `ArgSpec`, allowing it to access its switch values during the execution.

This system is illustrated by the following sequence diagram:



Note that the "Parser Logic" abstraction represents the system of `Parser`, `ArgParser`, `Commands` and `Executor`.

This model of having `Command` objects configured by configuration objects is somewhat unconventional, but it provides the benefit of enforcing the static initialisation of the switches, and facilitates testing - `Command`, `ArgCommand` and `ObjCommand` are equipped with public constructors that can take in switch values and arguments, hence allowing us to set them up for testing without making the switch setters public, and without copying these constructors across every subclass (as constructors are not inherited).

In summary, to define a new command:

1. Define a subclass of `CommandSpec`
2. Specify its execution in `execute` of `CommandSpec`
3. Define the private static field `spec` and the public static method `getSpec()` to provide singleton behaviour
4. Update `Commands` to link the command name to the new `CommandSpec`

If this command requires arguments, in addition to doing the above for a subclass of `ArgSpec` (instead of `CommandSpec`):

1. Create a private constructor for the subclass, and within the constructor:
    a. Define `cmdArgLevel`

b. Construct the switches for the command and supply them as arguments to `switchInit()`
    i. If there are no switches, call `switchInit()` with no arguments

NOTE	If there is no argument given for a switch, `getSwitchVal(<switch name>)` returns `null`. However, if a switch is not given, `getSwitchVal(<switch name>)` also returns `null`. The former case can be distinguished by the fact that `switchVals` will contain `<switch name>` as a key.

4.1.2. Parsing

The `Parser` object scans through a user-supplied input string. The first word is extracted, and if the corresponding command is an `ArgCommand`, it uses a finite state machine (FSM) which switches on the characters in the input. Switches are extracted, using the aliases in `switchAliases` to identify the full names of the corresponding switches. The switch arguments are then compared against the requirements of the `ArgCommand`, as stored in the `switchMap`.

The finite state machine for input parsing has the following states:

- `EMPTY`: parsing whitespace, which has no semantic meaning aside from serving as a separator
- `ARG`: parsing an argument that is not quoted, which may be for a switch or for the command itself
- `STRING`: parsing an argument that is surrounded by double quotes
- `SWITCH`: parsing a switch name

The state transitions are as follows:

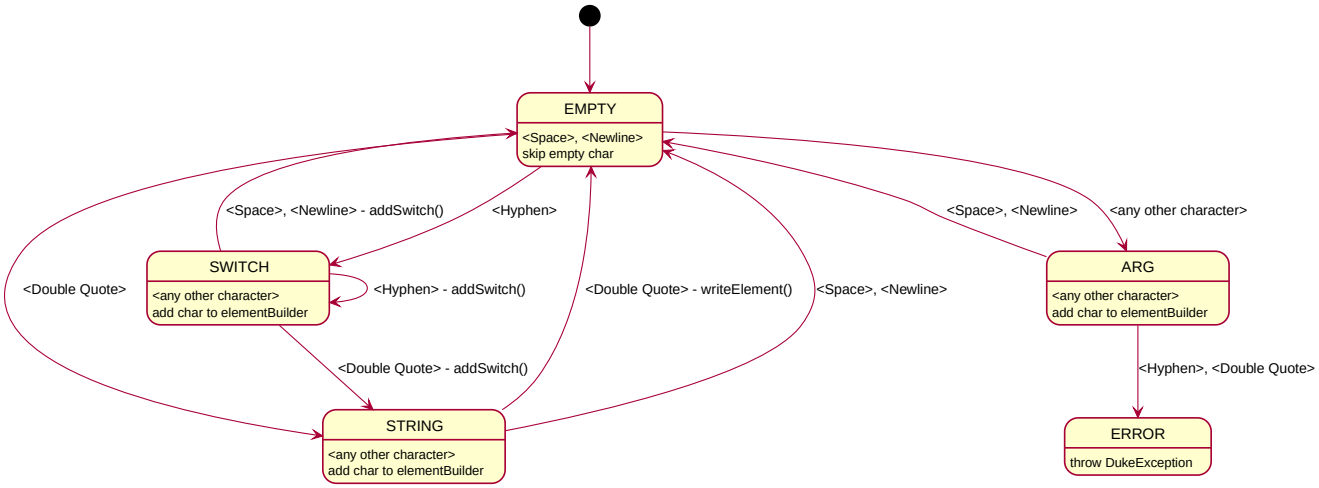
- `EMPTY`
    - `EMPTY` → `EMPTY`: <Space> or <Newline>
    - `EMPTY` → `SWITCH`: `-`
    - `EMPTY` → `STRING`: `"`
    - `EMPTY` → `ARG`: <any other character>
- `SWITCH`
    - `SWITCH` → `EMPTY`: <Space> or <Newline>
    - `SWITCH` → `SWITCH` (add current switch and begin processing a new switch): `-`
    - `SWITCH` → `STRING` (add current switch and begin parsing a string as an argument): `"`
    - `SWITCH` → `SWITCH` (add char to elementBuilder): <any other character>
- `STRING`
    - `STRING` → `EMPTY` : `"`
    - `STRING` → `STRING` (add char to elementBuilder) : <any other character>
- `ARG`
    - `ARG` → `EMPTY`: <Space> or <Newline>

- `ARG` → `DukeException`: Unescaped `"` or `-`
- `ARG` → `ARG` (add char to elementBuilder): <any other character>

Preceding any transition character with a backslash `\` will escape it, allowing it to be treated as an ordinary character.

While in the `ARG`, `STRING` or `SWITCH` states, each character that is read is added to a StringBuilder `elementBuilder`. When exiting the state, the string is processed as a switch via `addSwitch()`, or written as an argument to the `Command` being constructed by `writeElement()`. These functions also check if adding a switch or argument would be valid. This can be an argument for the `Command` itself, or a switch argument. `elementBuilder` is then cleared, and the parser continues parsing input characters.

These transitions are summarised in the following finite state diagram:



For more details on how switches are processed, see above on `Command` objects, and on the Switch Autocorrect feature.

When every character in the input has been consumed, cleanup will be performed based on the state that the `ArgParser` is in at that point:

- `EMPTY`: nothing is done
- `ARG`: call `writeElement()` to write a command or switch argument
- `SWITCH`: call `addSwitch()` to process the switch name
- `STRING`: call `writeElement()`, assuming the user simply forgot to close the string

The `ArgParser` also checks for the corner case of a switch without an argument at the end, in which case it attempts to write a `null` value for the switch.

4.2. Object Disambiguation [JOHN CUTHBERT KHOO TENG FONG]

4.2.1. Rationale

In order to provide the smoothest experience and least delay to our users, we want to allow them to identify the targets of operations such as reading, updating and deleting with minimal effort. Given a clear, unambiguous identifier like an index in a list, this is straightforward, but if the user wishes to access something by part of its name, or by one of its attributes, and there are multiple objects matching his criterion, he needs some way to disambiguate between them. Having such a disambiguation system in place instead of rejecting ambiguous input (e.g. anything other than an exact name) or preventing it (e.g. access by index only) would improve the user's experience and input speed by allowing more free-form input, without needing to worry so much if the input is of the correct form.

4.2.2. Implementation

This system extends and generalises the search feature: instead of only being able to open objects from the search context, we are able to perform any other command on objects identified from a search. This is done by storing the original command before opening the search context. After the user selects a particular object, the system executes the original command again, with the identified object supplied to it. Commands that are capable of such operations are `ObjCommand` s, and their behaviour is controlled by `ObjSpec` s. This system allows the user to search for objects based on any attribute, to select a result from that search, and perform an arbitrary command on it.

A brief recap of **Dr. Duke's** other systems is necessary here. All of **Dr. Duke's** components can be accessed from the `DukeCore` object. The `DukeCore` object is supplied to a command whenever it is executed, as commands may require all these systems to function. In the `DukeCore`, the `PatientMap` holds all patients being managed. `Patient` s, their `Impression` s, and the `Treatment` s and `Evidence` s of the `Impression` s are all `DukeObject` s. Each `DukeObject` can be viewed, and has an associated context which displays its information. "Viewing" `null` would open the `HOME` context, and would display all `Patient` s in the `PatientMap`. Searching in **Dr. Duke** is done by constructing a `SearchResults` object, using a search method of the current `DukeObject` being viewed. This will only find matching results that are the children of the `DukeObject`, and that `DukeObject` will be the the parent of the `SearchResults` returned. These search methods populate the `SearchResults` object through various strategies, such as matching all immediate children whose names contain the search term (`findByName()`), matching all immediate children whose fields contain the search term(`find()`), and matching any children whose fields contain the search term(`searchAll()`). Refer to the individual objects' Javadocs to see what capabilities they offer.

NOTE	`findByName()`, `find()` and `searchAll()` refer generically to these strategies, rather than to specifc methods implementing them (which may have different names).

`SearchResults` are constructed with a name, which is the search term used to populate it, a `List` of `DukeObject` s, which are the results of the search, and a parent `DukeObject`, which indicates the scope of the search. `SearchResults` can be combined using the `addAll()` method.

`ObjSpec` extends `ArgSpec` to provide the method `execute(DukeCore core, ObjCommand cmd, DukeObject obj)`, while `ObjCommand` extends `ArgCommand` to provide the method `execute(DukeCore core, DukeObject obj)`, which calls the `ObjSpec execute` method, with itself as the `cmd` parameter. Finally, `ObjSpec` has an abstract `executeWithObj(DukeCore core, DukeObject obj)` method, which specifies the operation

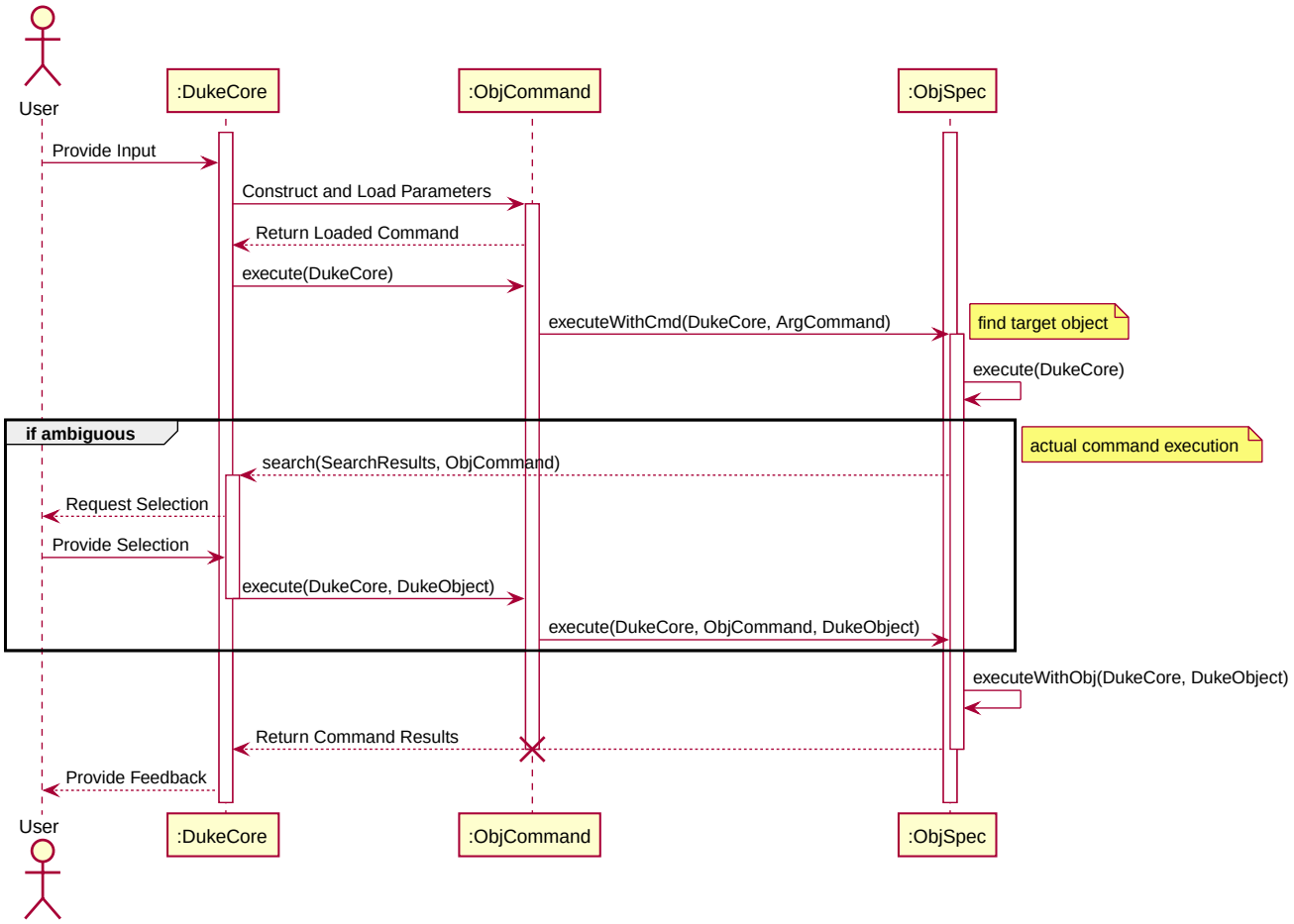
of the command once the object in question has been identified.

When an `ObjCommand` is executed via the regular `execute(DukeCore core)` method, it first attempts to see if the object can be disambiguated without requesting for explicit intervention by the user, via the `execute(DukeCore core)` method inherited by `ObjSpec`. Although there are no constraints on how this is to be done, the typical `ObjCommand` allows user input in either index or string form. If the user did not input an index, the `ObjCommand` will typically perform a `findByName()` search, as the user likely intends to select an object based on what is visible to him (which is primarily the name of the object). The typical behaviour detailed here is implemented in `HomeObjSpec`, `PatientObjSpec`, and `ImpressionObjSpec`, which provide these behaviours in the specific contexts, using the functions in `HomeUtils`, `PatientUtils` and `ImpressionUtils` respectively. These classes contain helper functions that can assist in the extraction of argument and switch values from typical commands in their respective contexts.

If there is only one result in the returned `SearchResults` object (or if a valid index was supplied), then the command can be performed on that object without ambiguity, with a direct call wot `executeWithObj`. If none are found, the command fails with an exception. However, if more than one result is found, then disambiguation is required. The `ObjCommand` then calls `search(SearchResults results, ObjCommand objCmd)` from the `DukeCore`, which opens the `SearchResults` in a search context, and stores the `ObjCommand`, with its `ObjSpec` and the switches in the `ObjSpec` set, as `queuedCmd`.

When viewing a `SearchResults` object, the user can only issue one command (whose behaviour is specified by `SearchSpec`), by selecting the index of the item he wishes to execute. This command, specified by `SearchSpec`, calls `executeQueuedCmd(DukeObject obj)` from the `DukeCore` on the object identified. This method would then call the `execute(DukeCore core, DukeObject obj)` of the stored `queuedCmd`, providing the identified `DukeObject` as an argument. The `ObjCommand` thus gains access to the object selected by the user, clearing up the ambiguity and allowing the user's desired operation to be executed.

This entire sequence of operations is summarised in this diagram (note that the UI and Parser have been abstracted into the `DukeCore` object):



To summarise, in order to use `ObjCommand` s: . Perform the steps for `ArgSpec` s. but using an `ObjSpec` instead . In `execute(DukeCore core)`, if the user's input is ambiguous as to which object it refers to,construct a `SearchResults` object containing the possible candidates, and call `search(SearchResults results, ObjCommand objCmd)` .. The `processResults()` method in `ObjSpec` will throw an exception if the `SearchResults` object contains no objects, will call `executeWithObj()` if there is only one object (using that object), and will call `search()` if there is more than one object. . Implement the abstract method `executeWithObj(DukeCore core, DukeObject obj)`. All operations that actually affect the system should be in `executeWithObj()`.

4.2.3. Comparison with Alternatives

Possible alternatives to this system would be the strict use of indices or the requirement for full names to be provided, as discussed above. However, in addition to failing to provide the flexibility discussed above, this solution does not work as well because our users are likely to think primarily in terms of names when dealing with their data. Being able to access objects by part of a name instead of scrolling through a (potentially large) collection of objects to find an index or trying to remember an exact name would increase the speed at which they navigate through the app and provide input to it.

Another suggestion proposed was the use of switches to differentiate between the use of an index or a name. This was also rejected as differentiating the two is simple enough to do without needing switches to identify the type of input. It is also less natural: when the user wishes to view the details of a patient, for example, `open Bob` is closer to a natural-language expression for this than `open -n Bob`. Commands that are closer to natural language would allow the user to more quickly and efficiently translate his intentions into input, thereby enabling him to more quickly and fluidly

input data.