COMP 303 Winter 2021

Assignment 5

Belle Pan 260839939

3rd April, 2021

`lastWatched()` method in `WatchList` class

- Implements the **observer design pattern** where many objects point to one, such that when the state of any of the objects change, they are all notified and updated automatically.
  - Every `Watchable` object added to a `WatchList` is assigned a reference to the `WatchList` in the variable a`WatchList`
  - This ensures that when the state of an object within a `WatchList`, i.e. when a watchable is watched, the `WatchList` is notified and updated.
  - All `Watchable` classes have been updated to have appropriate getter and setter methods to allow clients to know which `WatchList` it has been added to.

`Media` abstract class and its subclasses `TVShow`, `Episode`, and `Movie`

- These classes use the concept of **inheritance** to group together similar attributes and functions so as to reduce code duplication.
- The fields, aTitle, aLanguage, aStudio, aTags, and a`WatchList` are all declared in the abstract class Media, as well as their respective getter and setter methods. This is because these fields are used by the TVShow, Episode and `Movie` classes in a similar way, and have nearly identical code in the setter and getter functions.

`WatchList` class and its state-modifying actions and `Commands` – `undo()` and `redo()`

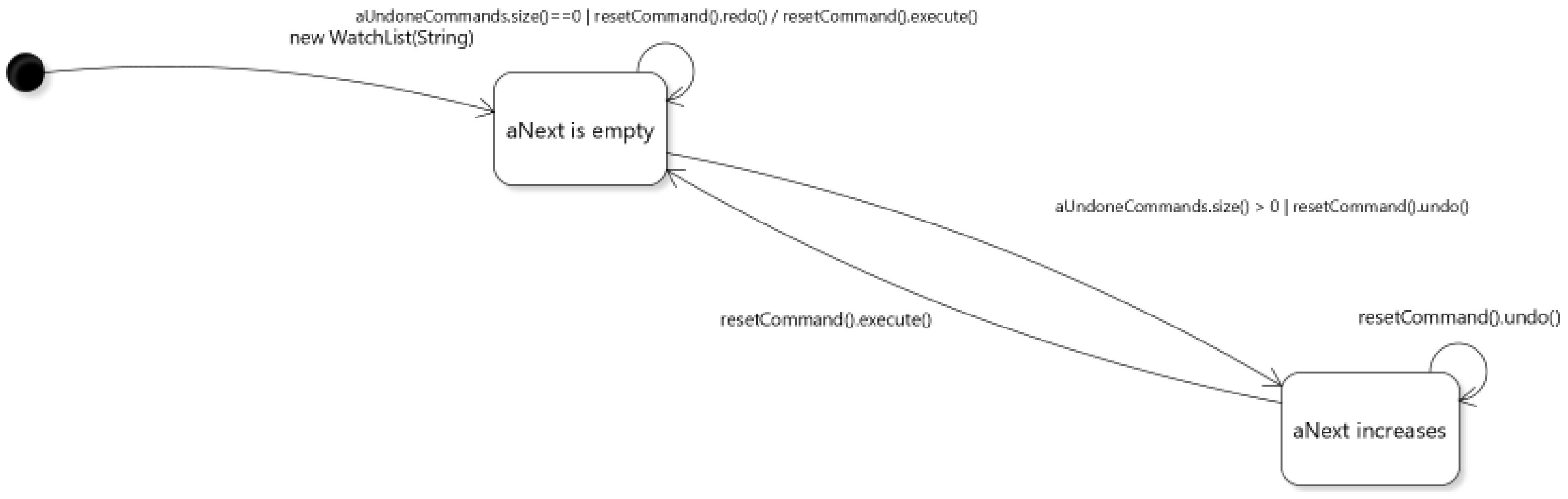
- Problem statement 3 required the use of the **command pattern** in which commands (i.e. function calls) can be stored and may be executed using various commands, such as `execute()`, `redo()`, and `undo()`.
- The interface Command has three functions: `execute()`, `undo()` and `redo()`. These are seen in class `WatchList` under the functions `addWatchableCommand(Media pWatchable)`, `removeWatchableCommand(int pIndex)`, `nextCommand()`, and `resetCommand()`. These functions all return an object of type `Command`, which indicates what kind of action it is.
- These actions or commands are executed by the `CommandProcessor`, which also keeps track of which functions have been executed or undone or redone in its `Deque`s, `aExecutedCommands` and `aUndoneCommands`.
  - When a command is executed by the `CommandProcessor`, it is added onto the `Deque` of executed commands, `aExecutedCommands`.
  - When an action is to be undone by the `CommandProcessor`, it first checks if there are any actions that may be undone (i.e. `aExecutedCommands` has a command that may be undone) and executes it and adds it onto the collection of undone commands,

aUndoneCommands. However, if there are no actions to undo, the CommandProcessor does nothing.

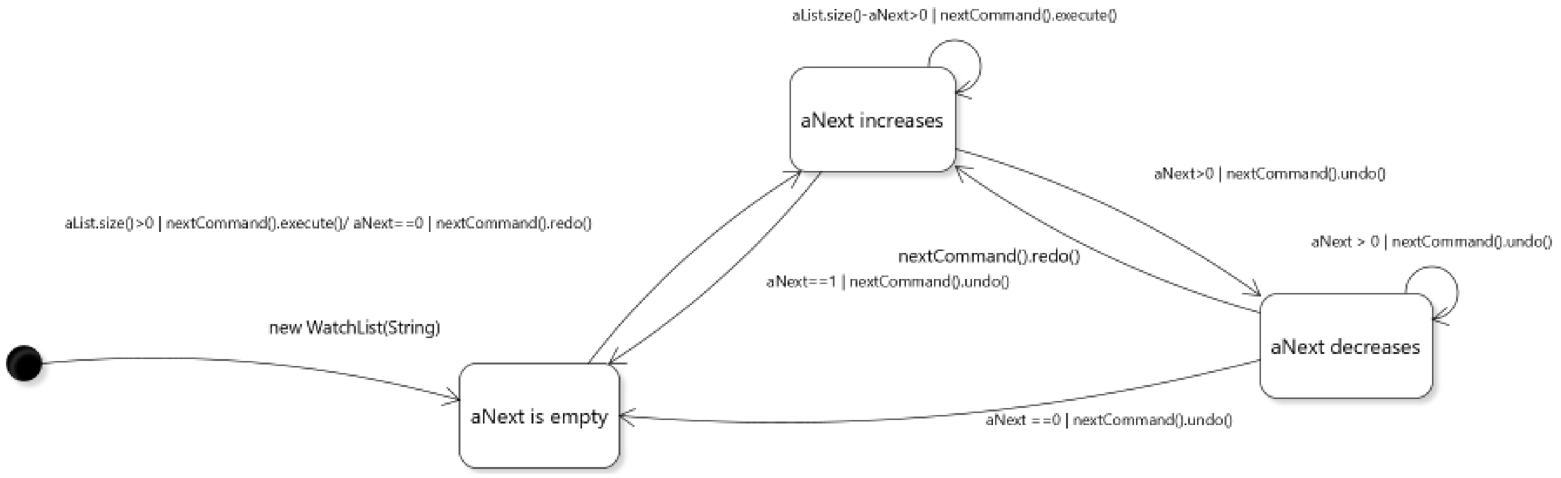
- When an action is to be redone, the CommandProcessor first checks if any commands can be redone (i.e. if there are commands in the aUndoneCommands deque) and executes it OR if no action can be undone, it repeats the last executed command.

The state diagrams below show the change of state within a WatchList given different commands:

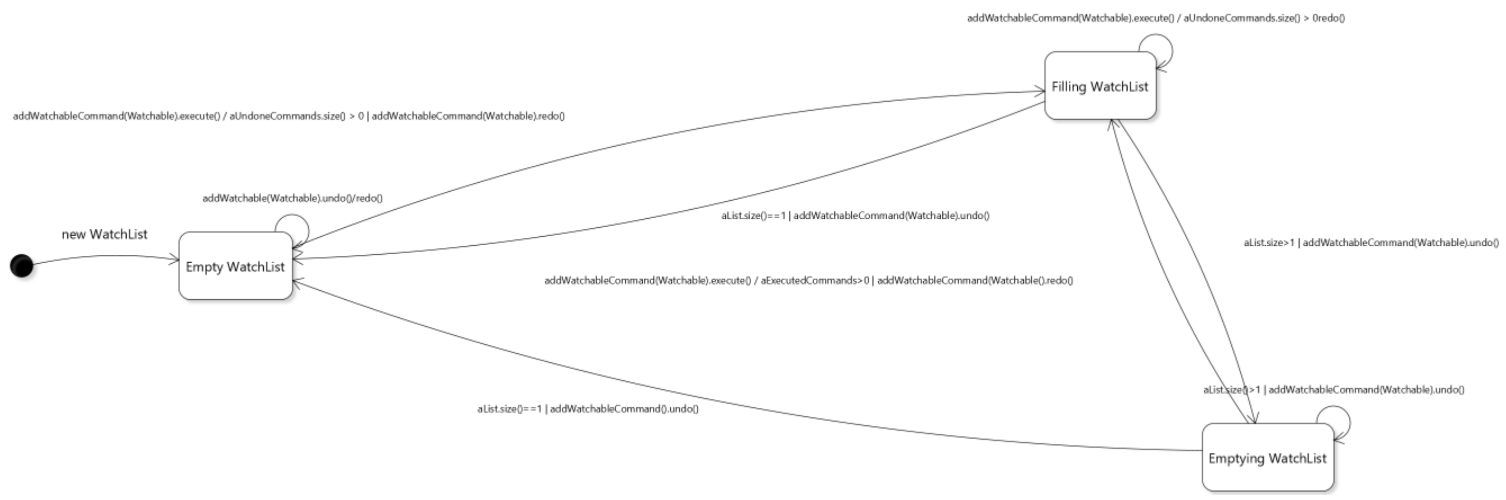
1. State diagram using the function resetCommand():


aUndoneCommands.size()==0 | resetCommand().redo() / resetCommand().execute()
new WatchList(String)
aNext is empty
aUndoneCommands.size() > 0 | resetCommand().undo()
resetCommand().execute()
resetCommand().undo()
aNext increases

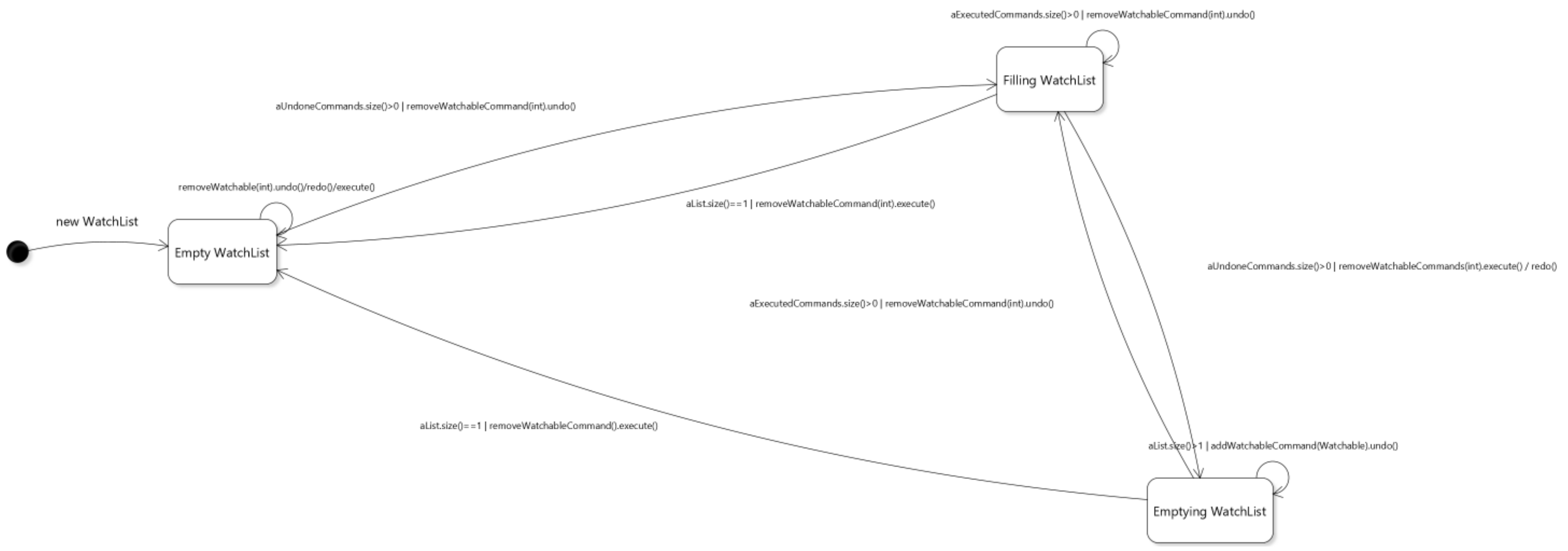
2. State diagram using the function nextCommand():


aList.size()-aNext>0 | nextCommand().execute()
aNext increases
aNext>0 | nextCommand().undo()
aList.size()>0 | nextCommand().execute()/ aNext==0 | nextCommand().redo()
aNext > 0 | nextCommand().undo()
nextCommand().redo()
aNext==1 | nextCommand().undo()
new WatchList(String)
aNext decreases
aNext is empty
aNext ==0 | nextCommand().undo()

3. State diagram using the function addWatchableCommand(Watchable):


addWatchableCommand(Watchable).execute() / aUndoneCommands.size() > 0redo()
Filling WatchList
addWatchableCommand(Watchable).execute() / aUndoneCommands.size() > 0 | addWatchableCommand(Watchable).redo()
addWatchable(Watchable).undo()/redo()
new WatchList
Empty WatchList
aList.size()==1 | addWatchableCommand(Watchable).undo()
aList.size>1 | addWatchableCommand(Watchable).undo()
addWatchableCommand(Watchable).execute() / aExecutedCommands>0 | addWatchableCommand(Watchable().redo()
aList.size()>1 | addWatchableCommand(Watchable).undo()
aList.size()==1 | addWatchableCommand().undo()
Emptying WatchList

4. State diagram using the function removeWatchableCommand(int):


aExecutedCommands.size()>0 | removeWatchableCommand(int).undo()
Filling WatchList
aUndoneCommands.size()>0 | removeWatchableCommand(int).undo()
removeWatchable(int).undo()/redo()/execute()
new WatchList
Empty WatchList
aList.size()==1 | removeWatchableCommand(int).execute()
aUndoneCommands.size()>0 | removeWatchableCommands(int).execute() / redo()
aExecutedCommands.size()>0 | removeWatchableCommand(int).undo()
aList.size()==1 | removeWatchableCommand().execute()
aList.size()>1 | addWatchableCommand(Watchable).undo()
Emptying WatchList