ACS-3913-001
Assignment 3 - *8% of final mark*
Due by Monday, November 21 at 11:59 pm

Read and review the GitHub documentation on Nexus. All code, documents, and diagrams will be submitted through GitHub Classroom for this course.

To accept Assignment 3: https://classroom.github.com/a/vi-yXawb

- *If you have not done so already, you will first claim your name with your GitHub account*

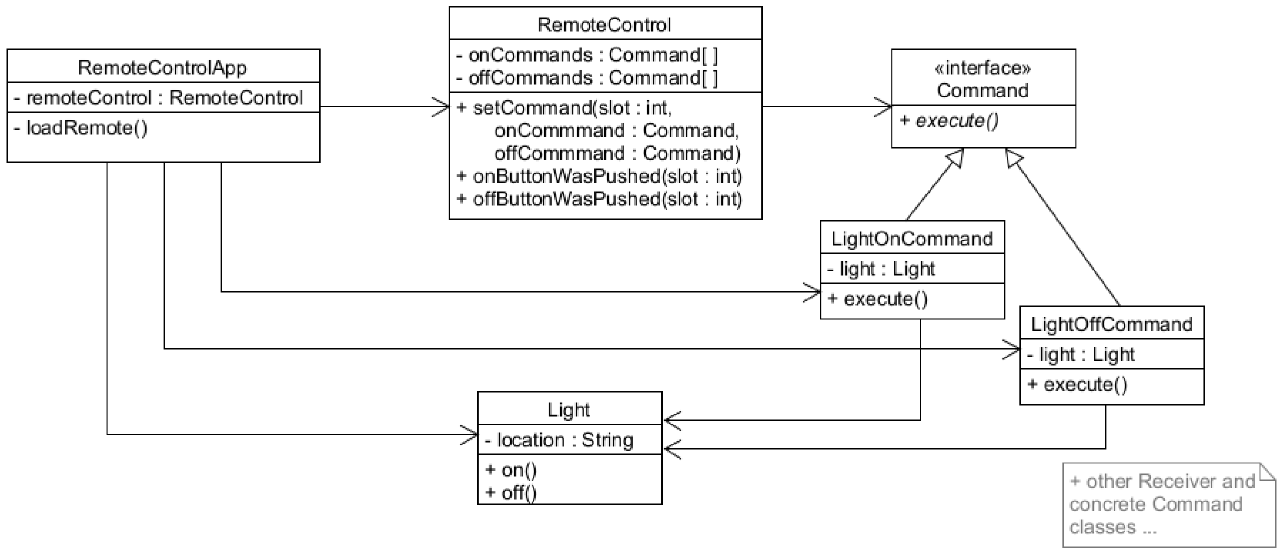
Use the `assignment-3-<your_github_username>` *repository on GitHub as the remote repository for Assignment 3.*

Refer to the Workflow tutorial section of the git and github fundamentals document to clone the repo to your local.

PART A

In this part of the assignment you will work with the Command example in your textbook, but with a GUI using Swing.

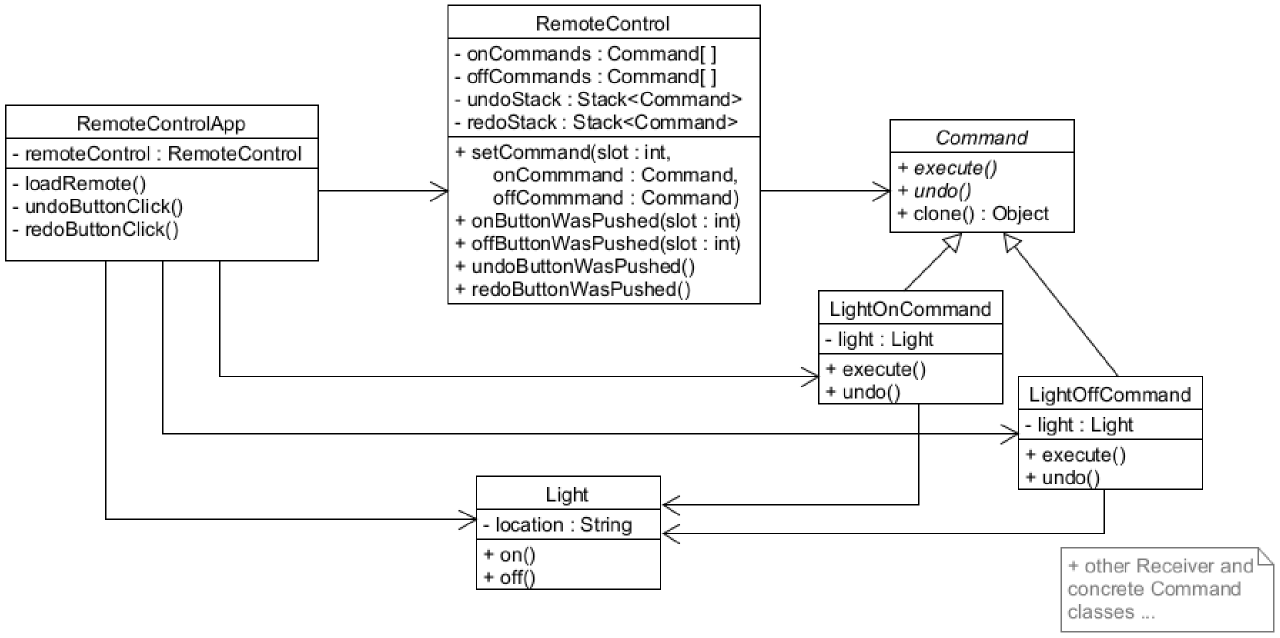
The startup code in your Assignment 4 repository consists of a simple remote control GUI working with the Remote example from your textbook (simplified with only Light and Stereo commands). Note that this example does not include the Undo function. Class diagram:


RemoteControlApp
- remoteControl : RemoteControl
- loadRemote()
RemoteControl
- onCommands : Command[ ]
- offCommands : Command[ ]
+ setCommand(slot : int, onCommmand : Command, offCommmand : Command)
+ onButtonWasPushed(slot : int)
+ offButtonWasPushed(slot : int)
«interface» Command
+ execute()
LightOnCommand
- light : Light
+ execute()
LightOffCommand
- light : Light
+ execute()
Light
- location : String
+ on()
+ off()
+ other Receiver and concrete Command classes ...

Look over the Swing code for RemoteControlApp. The `loadRemote()` method includes the setup like the RemoteLoader class in the original example. Test by running the application, and note the simulation output in the terminal window.

1. **Command Pattern**

   Continue working with the **Command** pattern and modify the code to implement an unlimited number of undos and redos as shown in the following class diagram:



RemoteControl
- onCommands : Command[ ]
- offCommands : Command[ ]
- undoStack : Stack<Command>
- redoStack : Stack<Command>
+ setCommand(slot : int, onCommmand : Command, offCommmand : Command)
+ onButtonWasPushed(slot : int)
+ offButtonWasPushed(slot : int)
+ undoButtonWasPushed()
+ redoButtonWasPushed()
RemoteControlApp
- remoteControl : RemoteControl
- loadRemote()
- undoButtonClick()
- redoButtonClick()
Command
+ execute()
+ undo()
+ clone() : Object
LightOnCommand
- light : Light
+ execute()
+ undo()
LightOffCommand
- light : Light
+ execute()
+ undo()
Light
- location : String
+ on()
+ off()
+ other Receiver and concrete Command classes ...

   In the text's example each slot of the remote control has a command object. Each command object is just reused whenever the user presses a button.

   Your command interface will include an `undo()` method.

   To manage command objects and undo/redo, you must use the `Stack` class in `java.util:`
   Use the approach where command objects are managed in two stacks: an undo stack and a redo stack:
   - when a command is to be executed (as opposed to redo or undo), do the command and then push it onto the undo stack
   - when undoing a command, pop it from the undo stack, undo it, and then push it onto the redo stack
   - when redoing a command, pop it from the redo stack, do it, and then push it onto the undo stack

Your code must utilize copies (or clones) of command objects. Note that the Command interface will then be an abstract class with the overridden clone method implemented.

Include the following commands in your code:

Receiver	Commands
Light	on, off
Stereo	on with CD Loud (volume at 11), on with CD (volume at 6), on with Radio (volume at 6), off
Ceiling Fan	high, medium, low, off
Hot Tub	on, off

Include a macro command that is customizable and can be programmed into the remote control. Test your remote control with different loaded commands.

****** BE SURE TO COMMIT ALL YOUR CHANGES BEFORE MOVING ON TO PART A Q2 ******

2. Singleton and Template Method Patterns

Program your remote control application to read from a configuration file and write to a log file.

Your program will now require two new **Singleton** classes:

i) `Configuration`: Upon instantiation, two pieces of information from the RemoteConfig.txt file are read and stored in the following fields:
   - `numberOfOptions:` the number of options to set the remote control
   - `height:` the height of the remote control in pixels

   Required methods: accessors for the two fields.
   The `RemoteControlApp` will use this information to set up the GUI.

ii) `Logger`: Logs every command with the current date and timestamp.
   Required method: `log()` that accepts a string (name of the command) as a parameter.

The abstract `Command` class will have 3 new methods

- `logCommand()` that passes the class name to the Logger Singleton for logging.
- `executeWithLog()` that calls `execute()` and `logCommand()`
- `undoWithLog()` that calls `undo()` and `logCommand()`

The Command hierarchy will also implement the **Template Method** pattern.

a. Produce the following design diagrams (for simplicity only include Livingroom Light commands i.e. exclude Stereo, Ceiling Fan, Macro commands)
   i) The new class diagram that includes the Singleton and Template Method patterns described above (you can draw over a screenshot of the image above)
   ii) Object diagram for objects that exist while the Remote is on and in use
   iii) The sequence diagram for a command followed by undo (use LightOn as an example)

b. Implement your design.

   Since there is no multithreading in this application, use lazy instantiation for your Singletons.

   Use the `Configuration` getters to set the constants in `RemoteApplicationApp` and `RemoteControl`

   `FileWriter` can be used for the logger. Use the `stop()` method of `Application` to invoke a method to close the file upon exiting the application.

   `RemoteControl` must now call the new template methods `executeWithLog()` and `undoWithLog()`

   Load your final remote with the following commands for submission (note that your remote should work with any possible command settings based on the set of commands built):
   - Living Room Light
   - Stereo with radio
   - Stereo with CD
   - Ceiling Fan high
   - Ceiling Fan medium
   - Macro named "Workout" with
     - Living Room Light on
     - Stereo with CD
     - Ceiling Fan high.

Notes:

- Hottub and Ceiling fans must be added (see your sample code for the Command pattern). Stereo commands must be updated to include states.
- Exception handling for when there is nothing to undo/redo is NOT required
- The `clone()` method must be overridden in the abstract command class. All concrete classes will use this method, there should be no duplicated code across concrete classes.
- Suggested steps for implementation:
    1. Add/Edit required commands
    2. Implement simple undo (1 level) to test commands
        - Load remote with commands that you want to test i.e. Stereo states
        - Add the undo call in RemoteControlApp
    3. Override clone method
        - You will have to handle exceptions at the call in the RemoteControlApp
    4. Work the unlimited undo/redo into the RemoteControl class
        - 2 stacks should replace the undoCommand variable
        - Add the redo call in RemoteControlApp
    5. Load the remote with the controls specified for submission
    6. COMMIT YOUR CHANGES
    7. Add the Configuration class, update the constants in RemoteControlApp
    8. Add the Logger class
        - Set up the file to write in the constructor
        - Include methods to:
            - log the command
                - use `LocalDate.now()`, `LocatTime.now()`, and `DateTimeFormatter(...)`
            - close the file (called when RemoteControlApp closes)
        - Add a `logCommand()` method in the abstract Command class
            - Simply call the `getClass()` of the command, then `getName()` to log the concrete Command class name
        - Add the `executeWithLog()` and `undoWithLog()` template methods in the abstract Command class
    9. Update the RemoteControl class with new calls

Sample -- button presses:
Light on
Stereo with CD on
Stereo with CD off
Ceiling fan high on

Stereo with radio on
Undo
Undo
Undo
Redo
Redo
Undo

Sample output:

```
------ Remote Control App Sim -------

Living Room light is on
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 6
Living Room stereo is off
Living Room ceiling fan is on high
Living Room stereo is on
Living Room stereo is set for Radio
Living Room Stereo volume set to 6
Living Room stereo is off
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 6
Living Room stereo is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
```

Sample log file:

```
2022-11-10 11:31:27 LightOnCommand
2022-11-10 11:31:31 StereoOnWithCDCommand
2022-11-10 11:31:34 StereoOffCommand
2022-11-10 11:31:37 CeilingFanHighCommand
2022-11-10 11:31:42 StereoOnWithRadioCommand
2022-11-10 11:31:46 StereoOnWithRadioCommand
2022-11-10 11:31:47 CeilingFanHighCommand
2022-11-10 11:31:48 StereoOffCommand
2022-11-10 11:31:50 StereoOffCommand
2022-11-10 11:31:50 CeilingFanHighCommand
2022-11-10 11:31:51 CeilingFanHighCommand
```

Submit all your files via GitHub Classroom.

Note that all diagrams must be submitted as PDFs.

PART B

Factory Patterns

In the new platform game Super Mario Bros UW, Mario and Luigi are off to save Princess Peach once again from the evil king koopa Bowser.

1. Mario will face different enemies throughout the UW universe. In the different worlds Duckworth Centre, Centennial Hall, and Lockhart Hall, the player will encounter several enemies from each of the species Piranha Plant, Koopa Troopa, and Goomba.

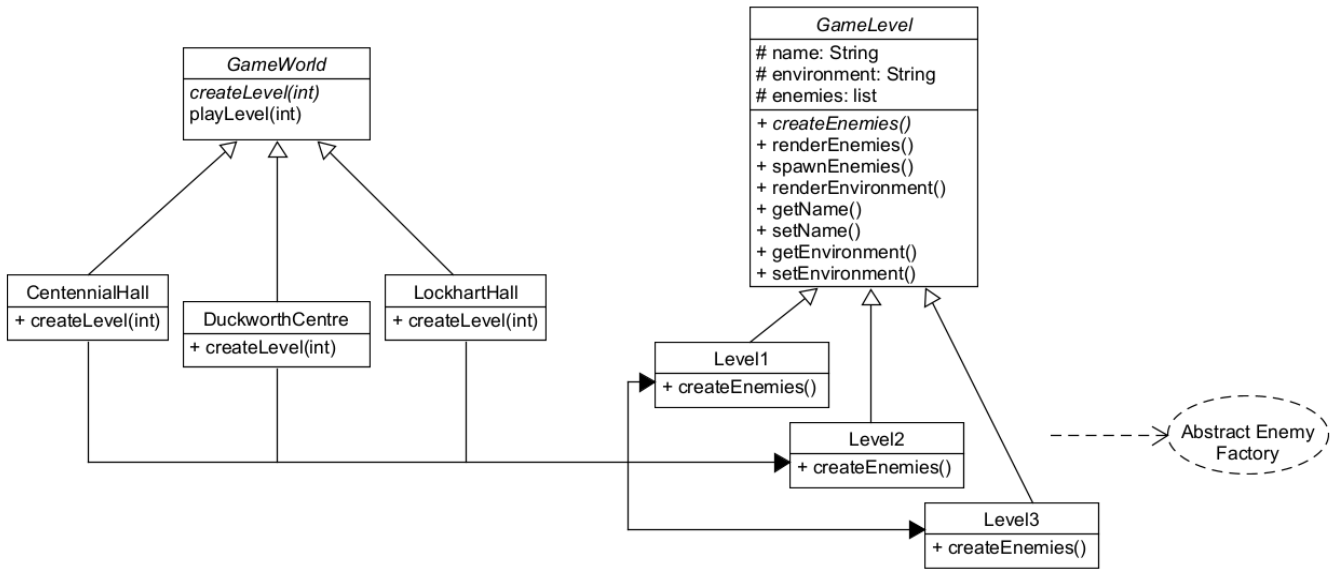
World/Species	**Piranha Plant**	**Koopa Troopa**	**Goomba**
Duckworth Centre	Piranha Plant	Koopa Troopa	Goomba
Centennial Hall	Poison Piranha Plant	Dry Bones Koopa	Galoomba
Lockhart Hall	Giant Piranha Plant	Super Koopa	Grand Goomba

   Use the **Abstract Factory Pattern** to create families of the different enemies based on the world.

   a) Draw the sequence diagram to show the messages sent when simulating Duckworth Level 2.
   b) Code the design. Interfaces are provided for you. Note that enemies are simplified by only having a toString method.
      - Test your code with the AbstractEnemyFactoryDriver.
      - See Sample test driver output

   ****** BE SURE TO COMMIT ALL YOUR CHANGES BEFORE MOVING ON TO PART B Q2 ******

2. Use the **Factory Method Pattern** to test different levels in the worlds according to the following class diagram:



Notes:

- levels use enemies from the Abstract Enemy Factories in Q1.
- enemies spawned in each level (in order):
    1. 7 piranha plants, 7 koopa troopas, 7 goombas
    2. 7 goombas, 8 piranha plants, 15 koopa troopas
    3. 10 of each species, randomly spawned
- environments are strings set when the levels are created
    - see sample output for descriptions
- other methods for simulation in the game level class simply display strings (see pizza example in text and sample output)

Sample test driver output

```
Duckworth Centre Enemies:
Piranha Plant
Goomba
KoopaTroopa

Manitoba Hall Enemies:
Poison Piranha Plant
Galoomba
Dry Bones Koopa

Lockhart Hall Enemies:
Giant Piranha Plant
Grand Goomba
Super Koopa
```

Sample game driver output

```
***Setting up Duckworth Centre Level 1***
Rendering environment: classrooms, labs, fitness facility, gym
Rendering enemies...
Spawning enemies:
Piranha Plant, Piranha Plant, Piranha Plant, Piranha Plant,
Piranha Plant, Piranha Plant, Piranha Plant, KoopaTroopa,
KoopaTroopa, KoopaTroopa, KoopaTroopa, KoopaTroopa, KoopaTroopa,
KoopaTroopa, Goomba, Goomba, Goomba, Goomba, Goomba, Goomba,
Goomba
***Completed playing Duckworth Centre Level 1***

***Setting up Centennial Hall Level 2***
Rendering environment: classrooms, cafeteria, security, escalators
Rendering enemies...
Spawning enemies:
Galoomba, Galoomba, Galoomba, Galoomba, Galoomba, Galoomba,
Galoomba, Poison Piranha Plant, Poison Piranha Plant, Poison
Piranha Plant, Poison Piranha Plant, Poison Piranha Plant, Poison
Piranha Plant, Poison Piranha Plant, Poison Piranha Plant, Dry
Bones Koopa, Dry Bones Koopa, Dry Bones Koopa, Dry Bones Koopa,
Dry Bones Koopa, Dry Bones Koopa, Dry Bones Koopa, Dry Bones
Koopa, Dry Bones Koopa, Dry Bones Koopa, Dry Bones Koopa, Dry
Bones Koopa, Dry Bones Koopa, Dry Bones Koopa, Dry Bones Koopa
***Completed playing Centennial Hall Level 2***

***Setting up Lockhart Hall Level 3***
Rendering environment: lecture halls, classrooms, mezzanine
Rendering enemies...
Spawning enemies:
Super Koopa, Grand Goomba, Giant Piranha Plant, Super Koopa, Grand
Goomba, Grand Goomba, Giant Piranha Plant, Grand Goomba, Giant
Piranha Plant, Giant Piranha Plant, Grand Goomba, Grand Goomba,
Giant Piranha Plant, Super Koopa, Giant Piranha Plant, Grand
Goomba, Grand Goomba, Super Koopa, Super Koopa, Grand Goomba,
Super Koopa, Giant Piranha Plant, Giant Piranha Plant, Super
Koopa, Super Koopa, Grand Goomba, Super Koopa, Giant Piranha
Plant, Super Koopa, Giant Piranha Plant
***Completed playing Lockhart Hall Level 3***
```

Submit all your files via GitHub Classroom.

Note that all diagrams must be submitted as PDFs.