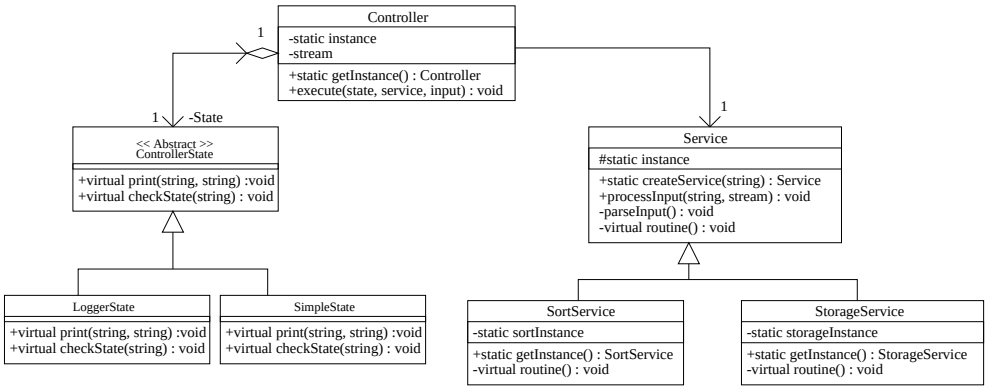
CS 246 - Assignment 4

Due Thursday, July 20 at 5pm

7% Final Mark

The following UML diagram is part of a specification for a service/controller system.



The program, `scsystem`, will take in file names from standard input one-by-one. For each file, the program will open and read in the file line-by-line. Each line will be parsed into 3 strings: The controller state, the type of service to run and the input for the service. The program calls `execute(state, line, input)` on its controller. After the controller finishes, the program continues to parse the next line of the file or to open another file. When the file name is `stop`, the program shuts down.

The Controller is responsible for processing the input using the desired service. It is able to run in 2 states: LOGGER and SIMPLE. The LOGGER prints data to a file called `log.txt`, while the SIMPLE state does not log any information. Only 1 instance of the controller is allowed to exist at any given time (i.e. it will prevent attempts to create a second instance). When the controller is created, it starts in the SIMPLE state.

The exchange between the controller and a service is as follows: The controller's state starts in a "waiting" state. This is represented in the code by the `ControllerState` being ready to execute a command from the controller. Once `execute(state, service, input)` is called, the controller calls `checkState(state)`, which checks the controller's current state and changes it, if necessary. When `checkState(state)` is called, the controller's state goes to its waiting state. `checkState(state)` is proceeded by a call to the `print(service, input)` method by the controller. If the controller's state is LOGGER , it will print the service that is being called (i.e. `SERVICE=string`), the input for the service (i.e. `INPUT=string`) and `OUTPUT=` to the `log.txt` file.

During the print state, the controller's state will also set the controller's stream to filestream for `log.txt` (if the state is LOGGER) or to standard output (if the state is SIMPLE). The controller continues to process the input by creating the correct service using the `createService(string)` method of `Service`. During this "processing" state it also calls `processInput(string, stream)`. After `processInput(string, stream)` has completed, the controller goes back to its "waiting" state.

A service is responsible for processing any input data sent by the controller. The processing varies according to the type of service (sorting or storing). There will only exist 1 instance of the service for any specific type of service. The service uses the string passed to it from the controller to output its result. If the service is storing, then it will print to `storage.txt`, as well as the given `stream`.

The exchange between the service and the controller is as follows: a service is instantiated and waits for input. Once the service is requested to process data, it receives the input and parses it to extract the information needed and executes the internal routine. The result of the routine is printed using the given `stream`. The service then waits for the next request.

Question 1

Based on the information given, provide a state diagram for the controller. You do not need to include state information about the services.

Question 2

Identify all 4 design patterns present on the UML diagram, indicating the classes and functions that compose the pattern.

Question 3

Indicate where and how the Strategy pattern could be applied to extend this design.

Question 4

Implement the above system in C++ according to the following guidelines:

Controller guidelines:

Controller::getInstance()
: creates a `Controller` if one does not already exist.

Controller::execute(state, service, input)
Uses the state `string` to set its state, the service `string` to start the appropriate service, and this service processes the input `string`.

ControllerState guidelines:

ControllerState::checkState(state)
is used to switch between the 2 different states.

ControllerState::print(service, input)
is dependent on the state. If it is *LoggerState::print(service, input)* being called, it will print to `log.txt`, or it will do nothing if it is *SimpleState::print(service, input)*. During this method, the state will set `Controller`'s `stream` to either standard output or a filestream to `log.txt`.

LoggerState::print(service, input)
should print the following message to `log.txt`:
"SERVICE=<service type> INPUT=<input> OUTPUT="
Where <service type> is the desired service and <input> is the actual service input.

SimpleState::print(service, input)
should print no message before the service is called.

Service guidelines:

Service::createservice(string)
The input defines what type of Service to create. This should be implemented in such a way that only this method can create an instance of Service.

Service::processinput(string, stream)
The `string` passed in is the set of data to be parsed, and the `stream` is the stream to use for output.

Service::parseInput() Uses the string given by `processInput(string, stream)` to create a vector of integers to be used by `routine()`

Service::routine()
`StorageService` will output its result to a file called `storage.txt`, as well as the given output `stream`. `SortService` will sort the input and output only to the output `stream`. The `Service` class must force this method to be implemented by its children.

Additional Information:

It is left up to you to figure out how to use file and output streams, as well as how to implement static methods. You may assume that all input will be valid. When implementing your solution, you may add attributes to your classes

where necessary. Simple `input.txt`, `log.txt`, `storage.txt`, and `output.txt` files have been provided to demonstrate the expected input/output format. The program must reset `log.txt` and `storage.txt` at some point during execution so that it does not append to the `txt` files of a previous run. An executable `scsystem` will also be provided eventually.

The given files would be generated by the following set of commands

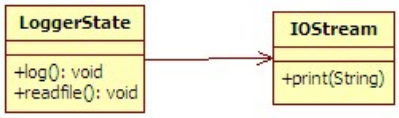
scsystem

input.txt

stop

Question 5

Suppose that the LOGGER state has a dependency to the *IOStream* class, as displayed on the diagram bellow. Explain why this is not recommended and how to invert the dependency and extend the design to also include a *FileStream* class. Draw a new UML diagram showing these modifications. HINT: The Adapter pattern may be of use.



Submission

The files you need to submit for this assignment are:

`a4.pdf`

`Controller.h, Controller.cc`

`ControllerState.h, LoggerState.h, LoggerState.cc, SimpleState.h, SimpleState.cc`

`Service.h, Service.cc, SortService.h, SortService.cc, StorageService.h, StorageService.cc`

`Main.cc`