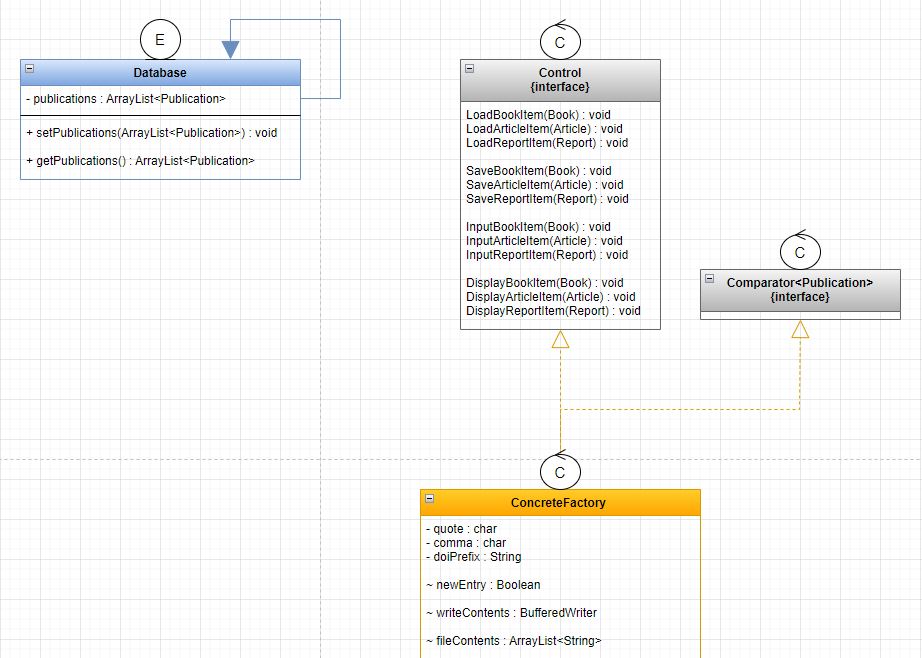
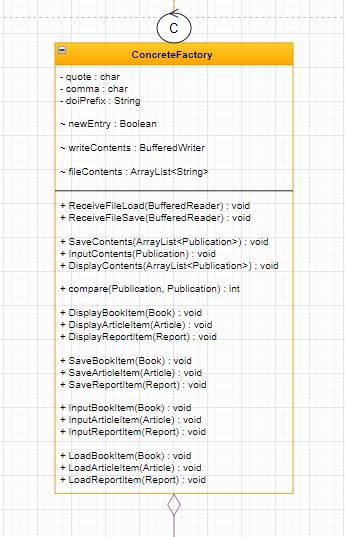
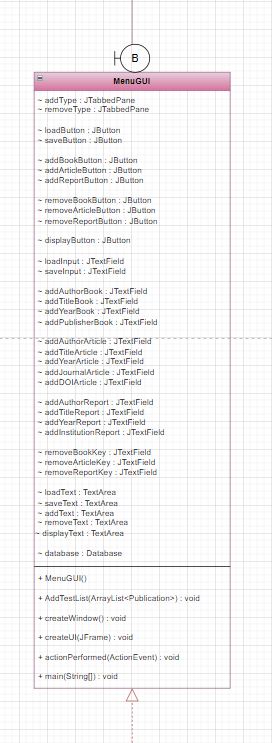
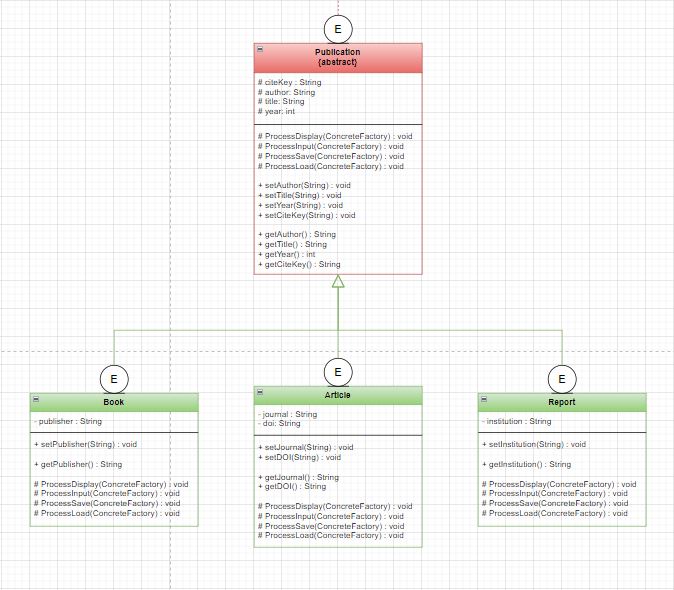
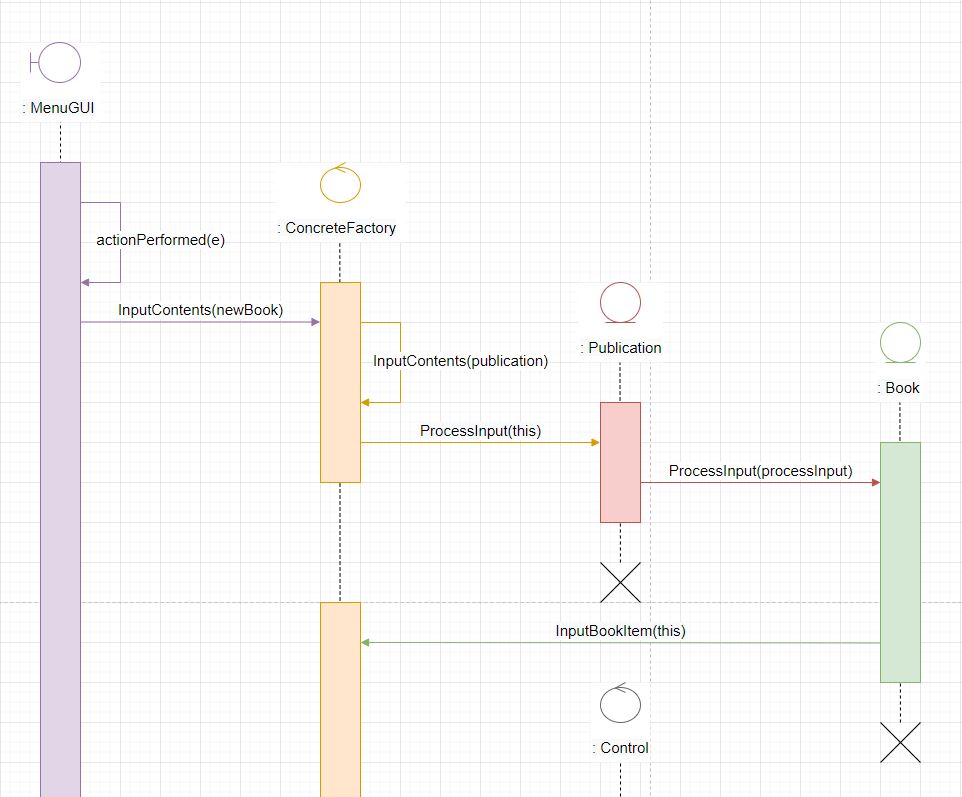
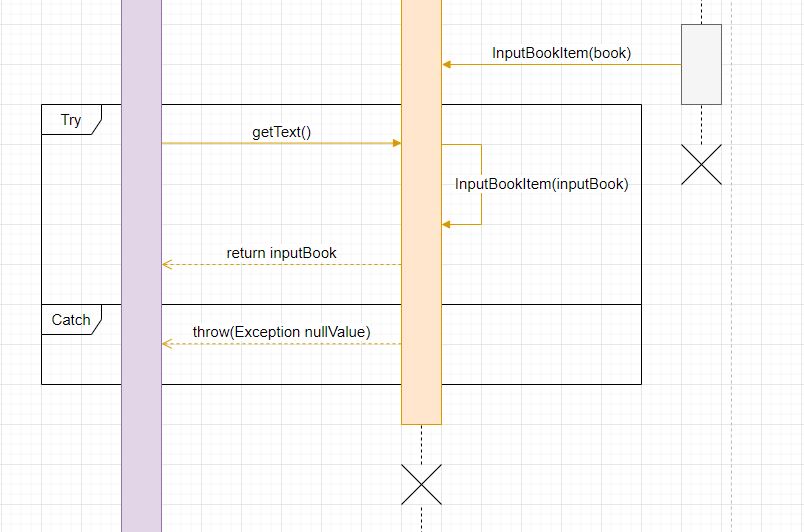
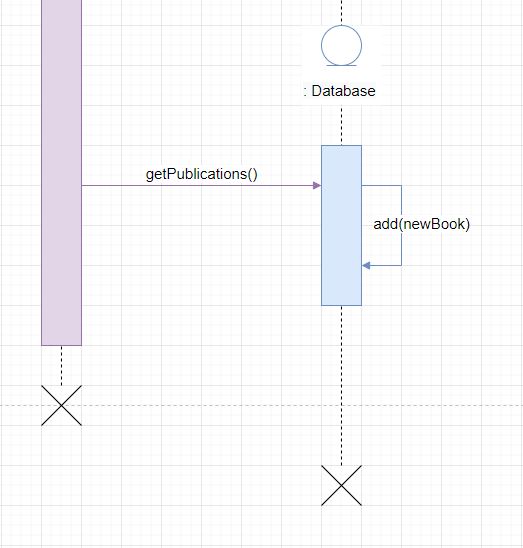
**Task 1**

****

****

**Task 2**

****

****

**Task 5**

**Explain the significance of the factory pattern and the strategy pattern in the solution.**

Strategy pattern allows us to call and override predefined methods/behaviours of our publication types during runtime whenever we need them.

These behaviours are encapsulated.

This pattern allows for a flexible method of executing type-related behaviours without needing to call individual instances and methods in numerous code blocks/switch statements. The algorithm simply switches as needed depending on (In our case a gui) what is selected, and the strategy algorithm handles the allocation to the correct method. (Factory method pattern methods)\*

Factory pattern works by creating an object that is client hidden and referenced using a common interface. It allows us to achieve pseudo polymorphism by letting the subclass decide what object to instantiate.

A description of how both work as follows:

Strategy:

We want add Publication (Type Book)

We click add book, instance is passed to concreteFactory InputContents(Publication) which defers to Publications processInput(ConcreteFactory) super method, which then defers to the correct override method in the subclass the instance is associated with, processInput(ConcreteFactory)

Factory:

Following on from add book:

Since we clicked add book, our processInput(ConcreteFactory) will now defer to the concreteFactory class an object of type book to a method called InputBookItem(Book) which is referenced in our interface.

**What changes would be required to extend the system to support a console-based interface instead of a GUI-based one?**

To extend the system to facilitate both gui and console-based functionality we would first need to create a sperate menu (menuConsole) that is designed to take input parameters via scanner/input stream.

Since all methods (Excluding those which are not directly affected by gui input e.g., Publication type data members and getters and setter or load, save and comparator to name a few) are directly designed to process input from and display to, a gui, we will need to copy and modify these methods to process and display to and from the console.

Essentially, we will have ‘duplicate’ methods for every gui based method but designed and named for the console-based interface instead.

As for the excluded methods, because they are not directly affected by the gui, and by extension the console, they do not need a console / gui version of themselves e.g., comparator.

The most efficient and compartmentalized way of implementing these changes is to move the new unique console-based factory methods to a duplicate class, minus the gui methods.

Do the same for the interface methods.

Move common methods from factory classes to a shared class, e.g., comparator method.

And finally remove main from menuGui and place in a singleton class which will call an instance of both console and gui interfaces.

This way core functionality of accessing the Publications/Sub Types as well as data storage is not compromised if 1 or both input interfaces fail or network difficulties occur.

And by extension if one of the input interfaces needs to be updated, (Assuming the actual system can run both console and gui concurrently – could use multithreading for this) we can disable local/server access to one of the interfaces while still having access to the systems tools via the remaining input interface.

Class list would be as follows:

Database

MenuGUI and MenuConsole

ControlGUI and ControlConsole (Interfaces)

CommonFactoryClass

ConcreateFactoryGUI and ConcreateFactoryConsole

Publications

Book, Article, Report

We can also separate main into a separate class and just call both gui and console from there.

**What changes would be required to extend the system to include a new type of publication (e.g. a conference publication that includes a “proceedings” field)?**

To extend the system to include a new publication type and data member we can do the following:

First make a copy of a similar subclass with 1 data member (Book) and refactor it to the new type and data member.

Following the steps above, we an do the same in the factory class and interface, copy similar methods and refactor them to conference type.

Lastly we need to add the functionality to the gui to support the tools needed for adding, removing etc a conference type.

Very little new code is needed to extend the system once the factory and strategy patterns are already implemented. Essentially just copy and paste a similar class and all related methods in the package, objects/instances etc and refactor to the new type and data member.