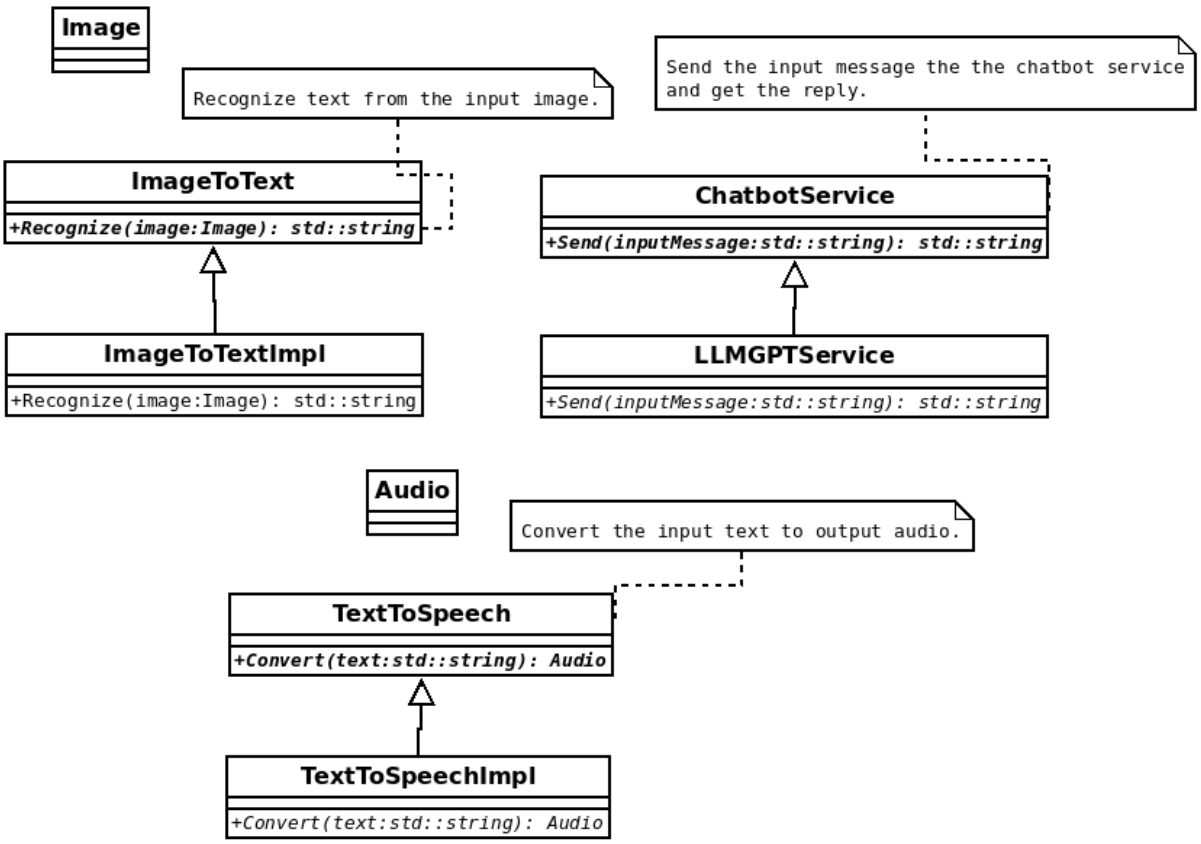
Suggested Solutions to HW#4 Problems

1. You are working on an online service that allows the user to input an image of handwritten questions. The system answers the user's question as speech. The system has the following interfaces and concrete implementation classes of image-to-text, chatbox service and text-to-speech functions:



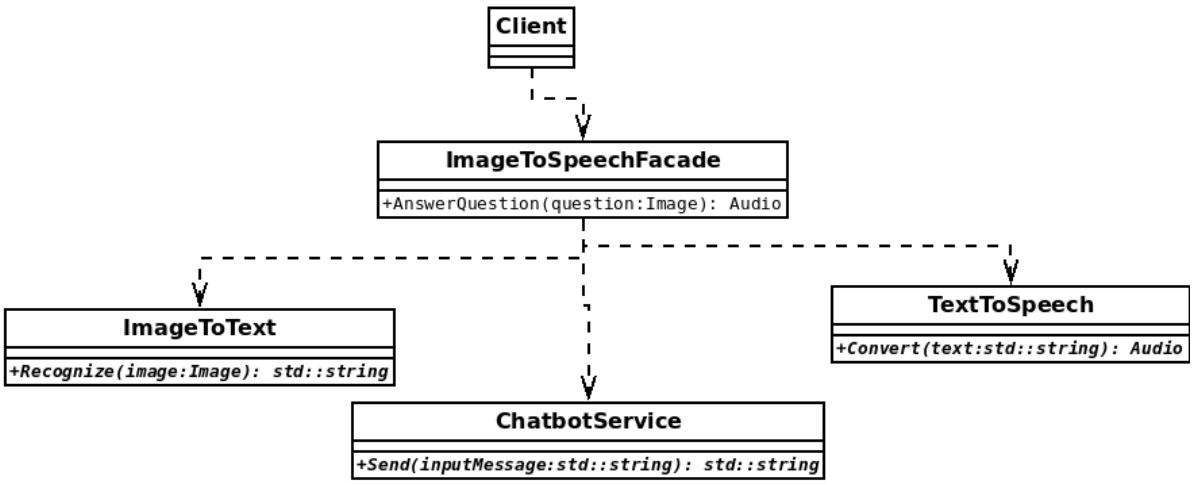
Suppose you want to make it easier for other services to use your image-to-speech answering service. It's a good idea to provide your service as a single interface so that the user doesn't need to interact with the individual classes.

(a) (15 %) How can design patterns help solve this design problem?

Solution. The facade pattern solves this design problem by providing a single simplified interface that hides the complexity of using the internal implementation classes. □

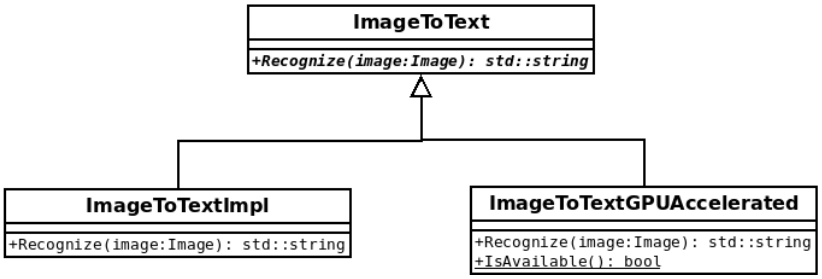
(b) (15 %) Please provide your design in a UML class diagram.

Solution. The class diagram:



□

2. After your service went online, you found that the pure-software implementation of ImageToTextImpl is too slow and consumes too much CPU time on your servers. You decided to accelerate text recognition using graphics hardware (GPU). The feature is provided by class ImageToTextGPUAccelerated:



Not all of your servers are equipped with the GPU hardware. Which image-to-text implementation class to use depends on the result of ImageToTextGPUAccelerated::IsAvailable(). In a sprint you managed to ship this improvement on time, but the code creates future maintenance problems. Your code base contains lots of snippets like:

```
1 ImageToText* imageToTextConverter =
2     ImageToTextGPUAccelerated::IsAvailable() ?
3         new ImageToTextGPUAccelerated() :
4         new ImageToTextImpl();
```

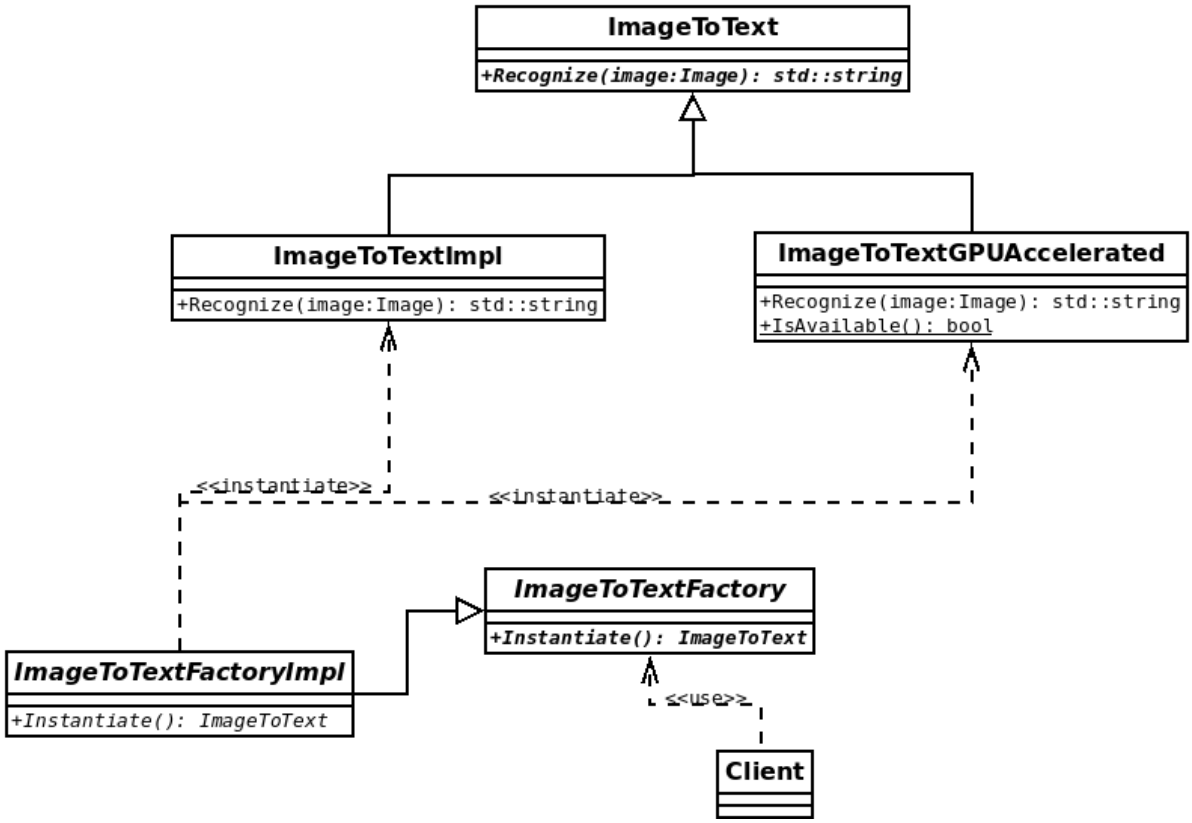
(a) (15 %) What's the problem with the above design? How can design patterns help solve the maintenance problem?

Solution. Direct instantiation of the implementation classes and the condition of which class to instantiate are sprinkled all over the code base. This is error-prone and inflexible if the instantiation process needs further changes in the future.

The factory method pattern solves this design problem by factoring out the instantiation process to the factroy method implementation. □

(b) (20 %) Please provide your solution in a class diagram.

Solution. The class diagram:



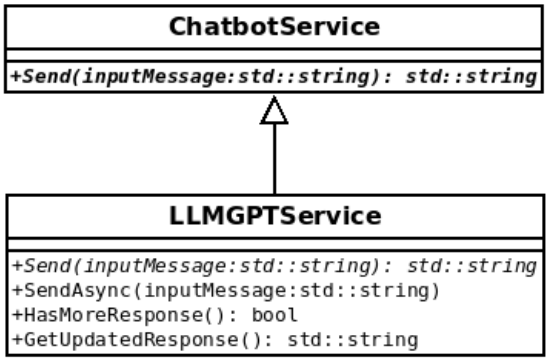
□

3. You then find that the synchronous interface of ChatbotService isn't good for user experience: it may take 30 seconds or longer for the implementation class, LLMGPTService, to respond to a question. This means the user starts to hear the answer only after the full response is returned from the interface.

   You notice that LLMGPTService provides an interface that allows you to get partial results. For example, after you send a message "What is virtual memory?", it can provide partial results like:

   "Virtual memory is a technique used in computer operating system " after 1 second,
   "that allows a process to think " after 2 seconds
   "that it can use the full contiguous address space " after 3 seconds and so forth until there is no more update.

   It provides this feature in the following interface:

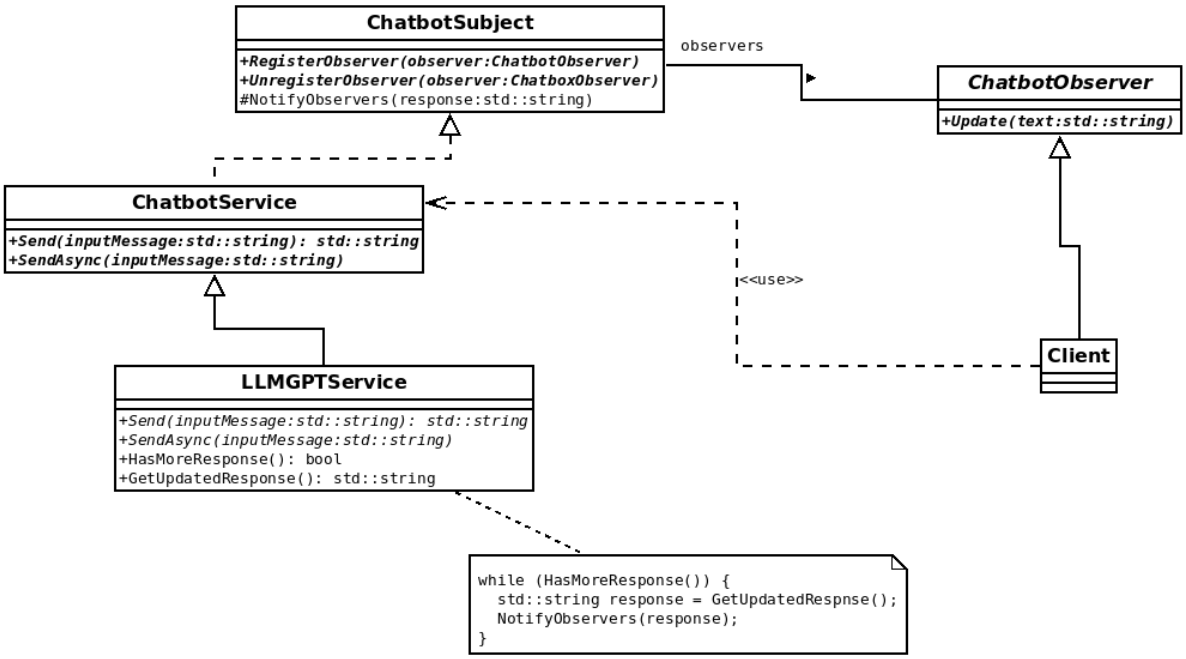


(a) (15 %) How can design patterns be used to allow the user of the ChatbotService to get updates on new responses from the remote chatbot service?

Solution. The observer pattern solves the design problem: the chatbot service works as the subject and the client class works as an observer. By implementing the subject, the chatbot service allows its observers to register for receiving updates when the service has partial results available. □

(b) (20 %) Please apply the pattern to LLMGPTService in code or a UML diagram

Solution. The class diagram:



□