THREAD POOLS AND ASYNCHRONOUS PROGRAMMING

1 OBJECTIVES

The main goal of this assignment is to learn and implement the following concepts of multithreading:

- Thread pools
- Tasks as worker threads
- Asynchronous programming together with thread pools

We are going to simulate a practical case in this assignment. The above topics are examined using the modern Java and C# techniques.

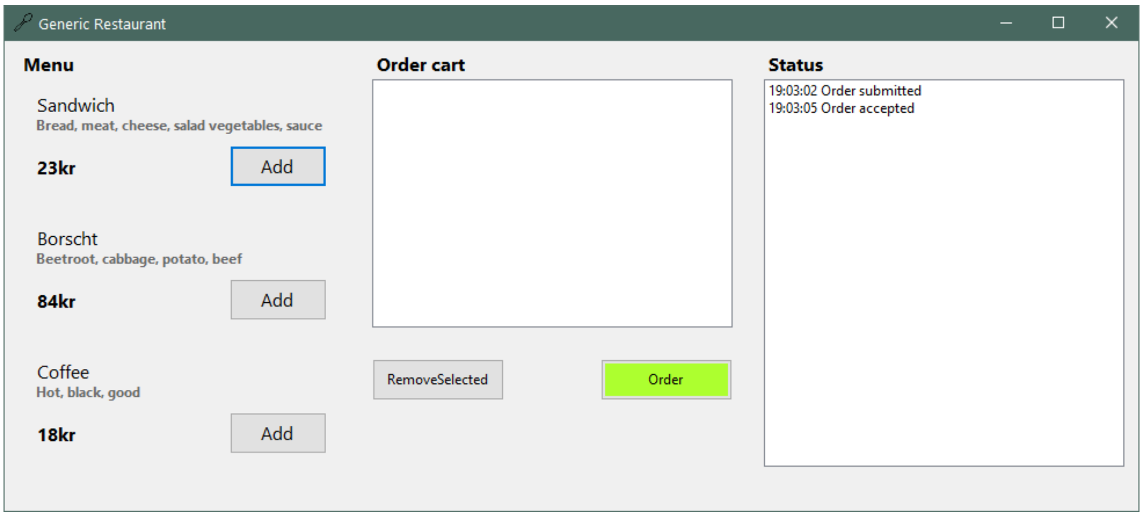
This assignment can be done individually or in "collaboration groups" of maximum two students. In the latter case, both group members can work together, discuss solutions, but each one should program separately on her/his own computer. If the solutions look the same or are similar, it is of course understandable. Each member should submit her/his solution, but both are expected to do the presentation together and each one should answer questions.

2 DESCRIPTION

Write a program that simulates client-server application for handling a restaurant's order system. As a major requirement, you have to implement all communications between the client and the server using asynchronous tasks. If you are familiar with socket programming, you may create a local client-server program. If not, you can implement the client and the server as two classes. Add random delays to simulate transmission time.

The main purpose of the assignment is to get practice with asynchronous programming and thread pools. The cooking process in the restaurant should be implemented as tasks submitted to a thread pool representing a kitchen. Additionally, message exchanges between a client and the server should be implemented with asynchronous calls, so that GUI stays responsive.

Although such applications are usually web-based, we are going to create a desktop application with a graphical user interface (GUI). If you are keen to create a web-application, it is of course not a restriction. A suggested desktop GUI is depicted in Figure 1.



19:03:02 Order submitted
19:03:05 Order accepted

Figure 1: A suggested GUI

3 The Generic Restaurant Application

In Canvas you may find the template that we propose you to use. The template consists of two main classes: **AbstractOrderClient** and **AbstractKitchenServer.** The method signatures and the comments may help you to start the implementation. Your task is to implement an **OrderClient** and a **KitchenServer** classes extending **AbstractOrderClient** and **AbstractKitchenServer** respectively. The proposed client-server API is depicted in Figure 2. The expected features are described as follows:

3.1 **OrderClient** should represent a self-service app, similar to Figure 1. It should allow a user to assemble an order, submit it to the KitchenServer and output the status of the order. Omit the process of paying for simplicity (or do it as an optional part).

3.2 As we assume a client-server application, requests to the **KitchenServer** through the network may take an unpredictable amount of time. Yet, GUI should stay responsive so that user may do other tasks. Use asynchronous method calls.

3.3 When the **KitchenServer** receives a new order, it should initialize the cooking process. Use a thread pool to represent the kitchen and submit a dummy task that represents cooking process. It is enough for the cooking task to sleep some period of time and modify the status of an order. If you are programming in Java, use a separate thread pool for cooking in the **KitchenServer**. In C#, only one system thread pool is to be used (there are custom libraries that support instantiating multiple thread pools, but we do not recommend that for this assignment).

3.4 Add random sleep intervals into **KitchenServer** methods to represent network delay.

3.5 You are expected to implement two main features:

3.5.1 Asynchronous client-server communication.

3.5.2 Explicit usage of a thread pool in the **KitchenServer** for cooking.

3.6 Optionally, you may implement a periodic task in the OrderClient to poll for status of an order. The template is made with this feature in mind, but you may simplify it adding a button to GUI for status requests.

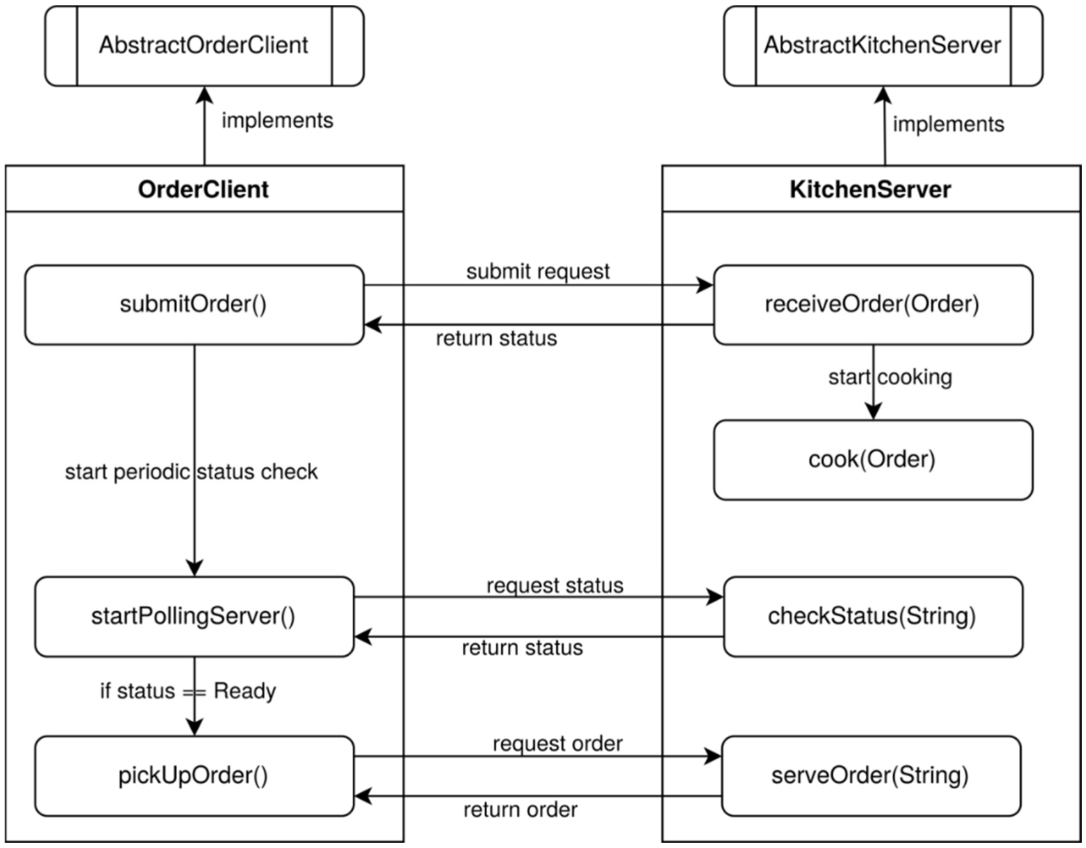
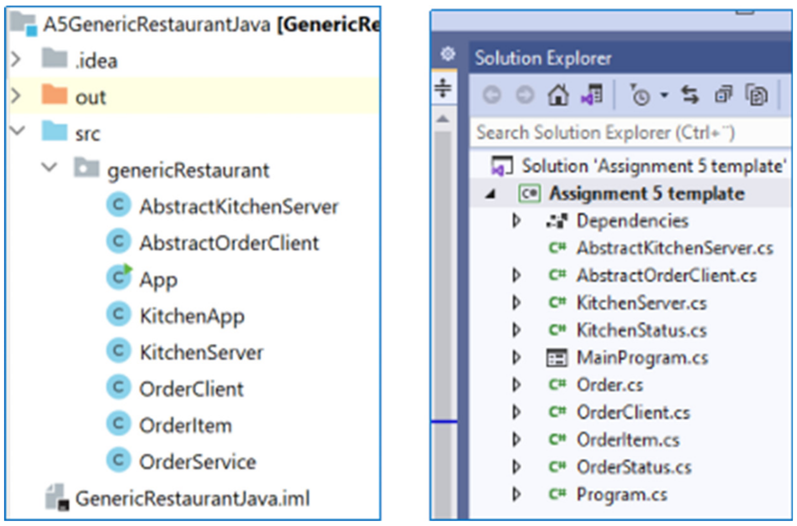


Figure 2: Pseudo-class diagram for the restaurant program

Figure 3 illustrates a possible class structure for a Java and a C# project. The templates are available for downloading on the assignment page on Canvas. You are free to use it, change or expand it, or develop your own solution.



```
A5GenericRestaurantJava [GenericRe
> .idea
> out
v src
  v genericRestaurant
      AbstractKitchenServer
      AbstractOrderClient
      App
      KitchenApp
      KitchenServer
      OrderClient
      OrderItem
      OrderService
  GenericRestaurantJava.iml
```

```
Solution Explorer
Search Solution Explorer (Ctrl+")
Solution 'Assignment 5 template'
  Assignment 5 template
    Dependencies
    AbstractKitchenServer.cs
    AbstractOrderClient.cs
    KitchenServer.cs
    KitchenStatus.cs
    MainProgram.cs
    Order.cs
    OrderClient.cs
    OrderItem.cs
    OrderStatus.cs
    Program.cs
```

Figure 3: Project templates (Java at left and C# at right)

4 Specifications and Requirements for a pass grade (G)

4.1 To qualify for a passing grade, you should implement this with good code quality.

4.2 Keep your work well structured, well organized and always have OOP in mind. Use proper variable and method names and do not forget to document your code by writing comments.

4.3 You may certainly make changes to the application in order to make it more fun and full-featured.

Test your application carefully before submitting; make sure there are no breaking bugs.

5 Help and Guidance

Java:

5.1 In the **KitchenServer**, you may use a **CompletableFuture** object as a “promise” to return value asynchronously. We discourage you to use the **complete**( ) method as it may lead to blocking of GUI. Use **SupplyAsync**( ) method to start an asynchronous task and **thenAccept( )** to assign a call-back (e.g. update of GUI) on completion of the asynchronous task.
(Useful ref: https://javagoal.com/completablefuture-in-java/).

5.2 You may use **Executors.newFixedThreadpool(int)** or preferably ExecutorService to create a thread pool and myPool.**submit(task)** to submit a task for execution. For more info on **ExercutorService**, you can visit: https://www.baeldung.com/java-executor-service-tutorial.

5.3 For setting up a periodic task, you may use Timer.**scheduleAtFixedRate**( ) together with a **TimerTask**.

C#:

5.4 You can only await in an **async** method and you can only await an **awaitable** object **(Task, Task<T>)**.

5.5 **Task.Run** queues a **Task** in the thread pool.

5.6 Use **Task.Delay**( ) instead of **Thread.Sleep( )** to simulate a delay. If you are using **Thread.Sleep( )** in a single-threaded asynchronous program, the whole program is blocked for the sleep duration.

5.7 The execution waits at **await** for the task to finish and returns its results without blocking the main thread because **async-await** uses a state-machine to let the compiler give up and take back the control over the awaited Task in an **async** method.

5.8 By putting **async** in the method signature, you are telling the compiler to use a state machine (a state machine consists of a finite number of states) to call this method (no threading so far). Then by running a **Task** you use a thread to call the method inside the task. And by awaiting the task you prevent the execution flow to move past the **await** line without blocking UI thread.

5.9 Additionally see the description in the template projects provided together with this assignment.

6 GRADING AND SUBMISSION

Submit your solution in Canvas and show your work to your lab-supervisor as before. If you have worked together with another student, both of you must be present at the same time presenting your solution to your supervisor.

Good Luck!

Farid Naisan,
Course Responsible and Instructor