SPL: Assignment 2

Java Concurrency and Synchronization

TAs in Charge:
Benny Lutati bennyl@post.bgu.ac.il
Maor Ashkenazi maorash@post.bgu.ac.il

1 Introduction

In the following assignment you are required to implement a simple micro-service framework and afterwards use it to build a simple shoe store. The micro-services architecture has become quite popular in recent years. In the micro-services architecture, complex applications are composed of small, independent *services* which have the ability to communicate with each other using *broadcast-messages* and *requests*. The micro-service architecture allows us to compose a large program from a lot of small independant parts. This not only allows for better testability and much clearer seperation of concerns, but it also allows to replace, add and remove different parts of the system without breaking the system as a whole or even shutting it down.

The assignment is composed of two main parts:

1. Building a simple but powerful micro-service framework
2. Implementing a simple shoe store application on top of this framework

It is very important to read the entire work before starting. Don't be lazy here, the work will be much easier if you read and understand the entire work in advance.

2 Part 1: Micro-Services Framework Architecture

In this part we will build a simple micro-services framework. A micro-services framework is composed of two main parts, Micro-Services and a MessageBus. Each micro-service is a thread that can send and receive messages using a shared object referred to as the message-bus.

2.1 Example of a Micro-Services Architecture

In order to explain how this framework works lets assume that we want to write a small application for an online shoe store. The shoe store in our example have 3 types of services: SaleService, WebSiteClientService, and StoreManagmentService. In addition, the store defines 2 types of messages - PurchaseRequest and FiftyPercentDiscount.

The SaleService responsibility is to handle PurchaseRequests which are received from a WebSiteClientService. The WebSiteClientService may send a PurchaseRequest on behalf of a real human client that is logged in to the store website. In order to support large number of clients, the store creates a single WebSiteClientService per logged in client and in addition it starts multiple SaleServices that will share the load of handling purchase requests. Finally, in order to promote sales, the store owner may occasionally start a "50% End-Of-Season-Discount", when this happens, he uses the StoreManagmentService in order to send a FiftyPercentDiscount message to all the existing WebSiteClientServices and SaleServices, each WebSiteClientService which receives such message will then show a popup in the corresponding logged-in user browser. Correspondingly each SaleService that receives the FiftyPercentDiscount message can then know that he is allowed to sale the shoes in lower price.

The Micro-Services architecture is designed to help us with this type of applications, Figure 1 show how the micro-services framework can be used in this case. As the figure shows, each of the services is a thread which have an assigned message-queue in the message-bus (Q1-Q5).For each

message type, the message-bus also maintains a list of the message-queues (Q1-Q5) of the micro-services that interested in **receiving** this message. Corresponding to our example we can see that the WebSiteClientServices are interested in receiving the FiftyPercentDiscount messages and the SaleService are interested in receiving both FiftyPercentDiscount messages and PurchaseRequest messages. Finally, the StoreManagmentService does not need to receive any of these message-types. Each micro-service is responsible to notify the message-bus about the message-types it would like to receive.

As can be seen in Figure 2, the micro-services framework defines 3 different types of message (interfaces): Broadcast, Request, and RequestCompleted. Micro services can explicitly send only subclasses of the first two types of messages, while the RequestCompleted type will be send implicitly while completing a request. FiftyPercentDiscount and PurchaseRequest are defined by the shoe store application and not by the micro-services framework. The difference between message types is important, when a micro-service uses the message-bus in order to send a Broadcast message, the message-bus will add the message to all the queues of the micro-services that are interested in this type of message. i.e., when the StoreManagmentService sends the FiftyPercentDiscount using the message-bus, the message will be added into the queues of all the SaleServices and WebSite-ClientServices (Q1-Q4 in Figure 1). On the other-side, when a micro-service uses the message-bus in order to send a Request message, the message-bus will choose a **single** micro-service from the micro-services that are interesting in receiving this type of message and add the message only to its queue. i.e., when a WebSiteClientService sends a PurchaseRequest using the message-bus, it will received to a single SaleService in a round-robin manner, explained below. The message-bus also supports marking requests as completed and adding a result to it. When a micro service tells the message-bus that a request is completed, it will implicitly add the special RequestCompleted message to the queue of the requesting micro-service, the RequestCompleted message will also contain the result of the request. i.e., when a SaleServices finished handling a PurchaseRequest, it marks it as completed with a receipt as its result. As a result a RequestCompleted message containing a receipt will be added to the queue of the sending WebSiteClientService which when it will take this message it will be able to show the receipt to the logged-in client.

How micro-services handle messages in their queue? Figure 3 depict a close-up look of a micro-service. We can see that a micro-service has a data-structure of callbacks corresponding to message-types. The execution code of the micro-service is quite simple, it simply waits for a new message, when such is received it finds an appropriate callback (as learned in the practical sessions), call it and repeat. This type of code is called an *Event-Loop*. For example, when a SaleService receives a PurchaseRequest its callback will contain all the code that should be executed in order to handle and complete this request. We will see later how callbacks are added.

2.2 Implementation of the Micro-Services Architecture

To this end, In our framework each micro-service will run on its own thread. The different micro-services will be able to communicate with one-another using a shared object which we will call the *message-bus*. The message-bus supports sending and receiving two types of messages. Broadcast message which upon being sent, will be delivered to every subscriber of that message type (by adding it to the queue of every micro-service that is interested in that message type). The second message type the message-bus supports is the *Request* which upon being sent, will be delivered to only one of its subsribers (in a round robin manner, described below). The different micro-services will be able to subscribe for message types they would like to receive using the message-bus. The different micro-services does not know of one another, all they know is about messages that were received to

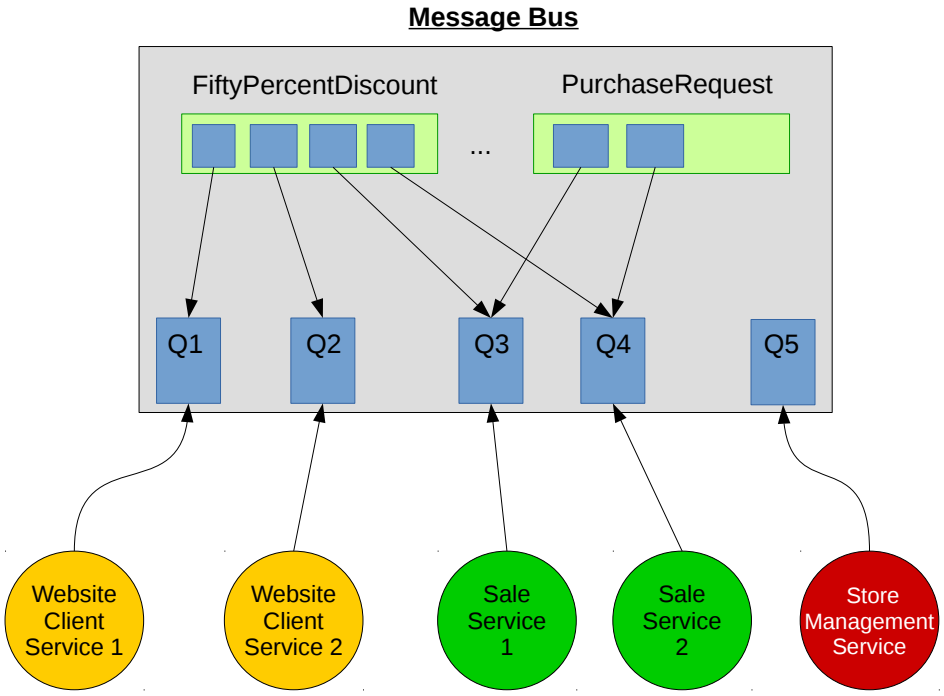

Figure 1: Shoe-store example, Overview

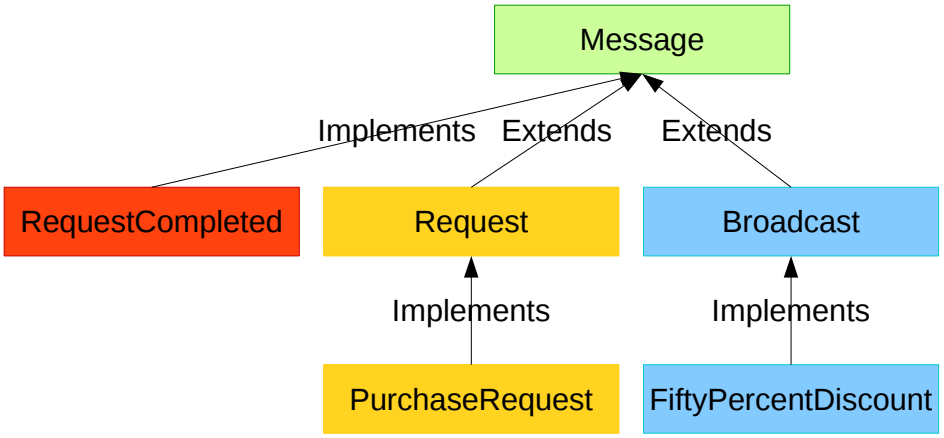

Figure 2: Shoe-store example, Messages

their queue in the message-bus and that the message-bus will add their messages to queues of other services that are interested in those message types.

When building a framework, one should change the way he thinks. Instead of thinking like programmer which writes software for end users, he should now think like a programmer writing a software for other programmers. Those other programmers will use this framework in order to build their own applications. For this part of the assignment you will build a framework (write code for other programmers), the programmer which will use your code in order to develop its application will be the future you while he works on the second part of this assignment.

Attached to this assignment is a set of interfaces that define the framework you are going to implement. The interfaces are located at the *bgu.spl.mics* package. Any Framework related classes that you will add in order to implement the framework should be under the package *bgu.spl.mics.impl*.

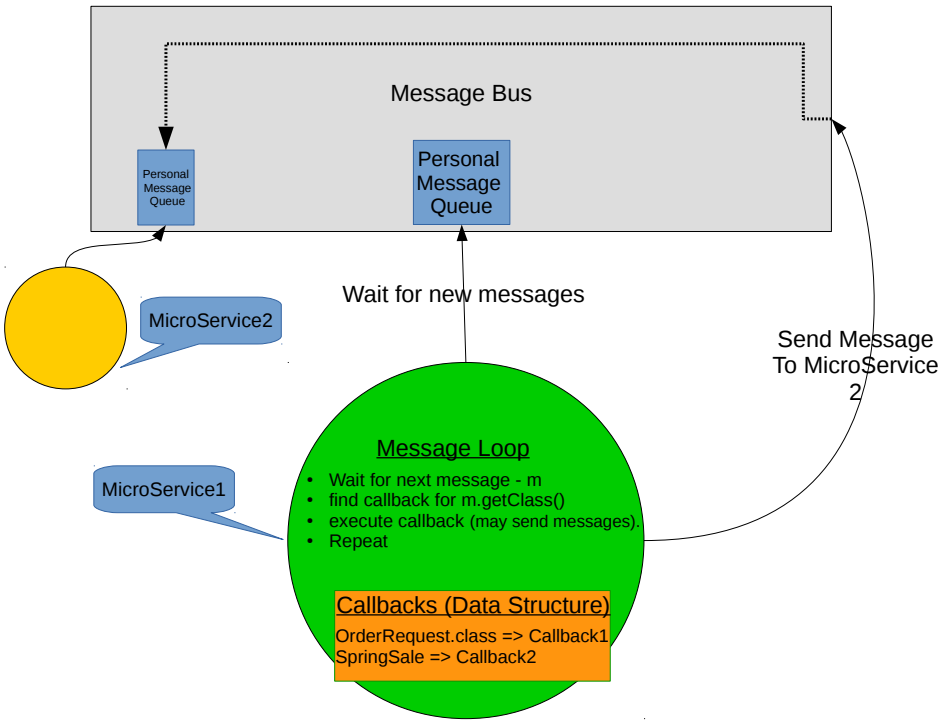

Figure 3: Shoe-store example, Microservice

Read the javadoc of each interface carefully. You are only required to implement the MessageBus and the MicroService for this part. The following is a summary and additional clarifications about the implementation of different parts of the framework.

- **Message** - A message is a data-object which is passed between micro-services as a means of communication. The Message interface is a "Marker" interface which means that it is used only to mark other types of objects as messages. It does not contain any methods but every class that you want to send as a message (using the message-bus) must implement it.
- **Broadcast** - A "Marker" interface extending Message. When sending Broadcast messages using the message-bus it will be received by all the subscribers of this Broadcast-message type (the message Class).
- **Request<T>** A "Marker" interface extending Message. A micro-service that sends a Request Message expects to be notified when a micro-service that received the request completed handling it, The request has a generic type variable T which indicate the expected result of the request (that should be sent when completed). The micro service that received this request must call the *complete* method of the message-bus. This method adds a RequestCompleted message to the appropriate queue. When sending a request, it will be received only by single subscriber - in a Round-Robin fashion. For example, assume 4 micro-services A,B,C,D and a request type R
  1. A and B subscribes to R
  2. D send request of type R $\rightarrow$ A receives it
  3. C subscribe to R

4. D send request of type R → B receives it
5. D send request of type R → C receives it
6. D send request of type R → A receives it
7. B unsubscribe to R
8. D send request of type R → C receives it

- **MessageBus** - The message-bus is a shared object used for communication between micro-services. It should be implemented as a *thread-safe singleton*. Create a class called *bgu.spl.mics.impl.MessageBusImpl* which implements the MessageBus interface. The message-bus implementation must be thread-safe as it is shared between all the micro-services in the system. For a list of the methods this class should implement, refer to the MessageBus javadoc. There are several ways in which you can implement the message-bus methods - be creative and find a good, correct and efficient solution. Remember, fully synchronizing this class will effect all the micro-services in the system (and your grade!) - try to find good ways to avoid blocking threads as much as possible.

  Subscribing to message types is done by using the required message class (the Class object you learn about in the practical sessions). If you find this concept hard to understand, take a look at the services in the attached code example (this code is described briefly in the appendix).

  The message-bus manages the micro-services queues, it creates a queue for a micro-service using the *register* method. When the micro-service calls the *unregister* method the message-bus should remove its queue and clean all references related to the micro-service. Once the queue is created, a micro-service can take the next message in the queue using the *await-Message* method. This method is blocking - meaning that if no messages are available in the micro-service queue it should wait until a message became available.

- **MicroService** - The MicroService is an abstract class that any micro-service in the system must extend. The abstract MicroService class is responsible to get and manipulate the singleton MessageBus instance. Derived classes of MicroService should never directly touch the message bus. Instead, they have a set of internal protected wrapping methods they can use . When subscribing to message-types, the derived class also supplies a callback function (see the *subscribe* method **of the abstract micro service**). The MicroService stores this callback function together with the type of the message is related to. The micro-service is a Runnable (i.e., suitable to be executed in a thread) its run method implements a message loop. The message-loop should handle registering and unregistering to the message-bus as needed. When a new message is taken from the queue, the micro-service will invoke the appropriate callback function.

  When the micro-service starts it register itself with the message-bus and then calls the abstract *initialize* method. The initialize method allows derived classes to perform any required initialization code (e.g., subscribe to messages). Once the initialization code completes, the actual event-loop should start. The micro-service should fetch messages from its message queue using the message-bus's *awaitMessage* method, for each message it should and execute the corresponding callback. The MicroService class also contains a *terminate* method that should signal the message-loop that it should end.

  Each micro-service contains a *name* given to it in construction time (the name is not guarantee to be unique).

  In the attached code, the abstract MicroService class is incomplete - you need you complete it using the explanations above and its javadoc.

IMPORTANT

- All the callbacks that belong to the micro-service must be executed inside its own event-loop.
- A micro-service should not block for a long time while handling a nessage as it will not allow the message loop to continue handling other messages.

3 Part 2: Building an Online Shoe Store

In this part of the assignment, you will use the micro-services framework you built in order to implement the functionality needed for a simple online shoe store with simulated clients. Please read this part carfully - the store you will implement is not exactly the same as the store shown on the example of part one.

For the purpose of the simulative parts of the system you will implement a time service. The time service keeps track of time passing in our execution, it serves as a global clock that counts the number of clock ticks that have passed and broadcast TickBroadcast messages (which contains the number of the current tick).

The following is a list of required classes that you need to implement, you may add classes as you wish. You may (and should) also add methods to the objects described below as you see fit. Every application related classes should be in the package *bgu.spl.app* or a sub-package of it.

3.1 Passive Objects

This section will contain list of passive classes (a.k.a., non runnable classes) that you need to implement.

3.1.1 ShoeStorageInfo

An object which represents information about a single type of shoe in the store (e.g., red-sneakers, blue-sandals, etc.). It contains the following fields:

- *shoeType: String* - the type of the shoe (e.g., red-sneakers, blue-sandals, etc.) that this storage info regards.
- *amountOnStorage: int* - the number of shoes of *shoeType* currently on the storage
- *discountedAmount: int* - amount of shoes in this storage that can be sale in a discounted price. (i.e., if amountOnStorage is 3 and discountedAmount is 1 it means that 1 out of the 3 shoes have a discount, after selling this one shoe the discount will end)

It may also contain different methods that allow modifying this values by the store.

3.1.2 Receipt

An object representing a receipt that should be sent to a client after buying a shoe (when the client's PurchaseRequest completed). The receipt contains the following fields:

- seller: string - the name of the service which issued the receipt

- customer: string - the name of the service this receipt issued to (the client name or “store”)
- shoeType: string - the shoe type bought
- discount: boolean - indicating if the shoe was sold at a discounted price
- issuedTick: int - tick in which this receipt was issued (upon completing the corresponding request).
- requestTick: int - tick in which the customer requested to buy the shoe.
- amountSold: int - the amount of shoes sold

3.1.3 Store

This object must be implemented as a thread safe singleton
The store object holds a collection of *ShoeStorageInfo*: One for each shoe type the store offers. In addition, it contains a list of receipts issued to and by the store.

Only the following methods should be publicly available from the store :

```
public void load(ShoeStorageInfo[] storage)
```

This method should be called in order to initialize the store storage before starting an execution (by the ShoeStoreRunner class defined later). The method will add the items in the given array to the store.

```
public BuyResult take(String shoeType, boolean onlyDiscount)
```

This method will attempt to take a single showType from the store. It receives the shoeType to take and a boolean - onlyDiscount which indicates that the caller wish to take the item only if it is in discount. Its result is an enum which have the following values:

- NOT_IN_STOCK: which indicates that there were no shoe of this type in stock (the store storage should not be changed in this case)
- NOT_ON_DISCOUNT: which indicates that the "onlyDiscount" was true and there are no discounted shoes with the requested type.
- REGULAR_PRICE: which means that the item was successfully taken (the amount of items of this type was reduced by one)
- DISCOUNTED_PRICE: which means that was successfully taken in a discounted price (the amount of items of this type was reduced by one and the amount of discounted items reduced by one)

Important: If there is discount available for the requested shoe type on stock it should be taken, even if the *onlyDiscount* parameter is false.

```
public void add(String shoeType, int amount)
```

This method adds the given amount to the *ShoeStorageInfo* of the given *shoeType*.

```
public void addDiscount(String shoeType, int amount) {
```

Adds the given amount to the corresponding *ShoeStorageInfo*’s discountedAmount field.

```
public void file(Receipt receipt)
```

Save the given receipt in the store.

```
public void print()
```

This method prints to the standard output the following information:

- For each item on stock - its name, amount and discountedAmount
- For each receipt filed in the store - all its fields

Please print this information in a comperhansive and easy to follow format.

3.1.4 PurchaseSchedule

An object which describes a schedule of a single client-purchase at a specific tick. It contains the fields:

- *shoeType: string* - the type of shoe to purchase.
- *tick: int* - the tick number to send the *PurchaseOrderRequest* at.

3.1.5 DiscountSchedule

An object which describes a schedule of a single discount that the manager will add to a specific shoe at a specific tick. It contains the fields:

- *shoeType: string* - the type of shoe to add discount to.
- *tick: int* - the tick number to send the add the discount at.
- *amount: int* - the amount of items to put on discount (i.e., if the amount is 3 than after selling 3 items the discount should be over).

3.1.6 Messages

The following message classes are **mandatory**, the fields that these messages holds are omitted - you need to think about them yourself:

- *PurchaseOrderRequest:* a request that is sent when the a store client wish to buy a shoe. Its response type expected to be a Receipt. On the case the purchase was not completed successfully null should be returned as the request result.
- *ManufacturingOrderRequest:* a request that is sent when the the store manager want that a shoe factory will manufacture a shoe for the store. Its response type expected to be a Receipt. On the case the manufacture was not completed successfully null should be returned as the request result.
- *TickBroadcast:* a broadcast messages that is sent at every passed clock tick. This message must contain the current tick (int).
- *NewDiscountBroadcast:* a broadcast message that is sent when the manager of the store decides to have a sale on a specific shoe.

- *RestockRequest:* a request that is sent by the selling service to the store manager so that he will know that he need to order new shoes from a factory. Its response type expected to be a boolean where the result: true means that the order is complete and the shoe is reserved for the selling service and the result: false means that the shoe cannot be ordered (because there were no factories available).

You are encouraged to add any new messages you find helpful both for debugging reasons and just for fun.

You should implement all the above messages and use them in the shoe store application as needed.

3.2 Active Objects (Micro-Services)

The following will describe the micro-services you are required to implement. In some of the micro-services definitions, a list of used messages is provided, this list is **not complete**, meaning that you **may or may not** want to listen or send other types of messages. In addition, you may add parameters to the micro-services constructors as you see fit.

Passing information between micro-services is allowed only using messages, asides from the message-bus and the Store - there should be no other singleton objects. Using non final static fields in order to pass information between the micro-services is not allowed. The ShoeStoreRunner class (the main class) can pass information to services only at creation time (i.e., using their constructor) and micro-services cannot refer to the ShoeStoreRunner class or hold references to one another.

3.2.1 TimeService

service name = "timer"
This micro-service is our global system timer (handles the clock ticks in the system). It is responsible for counting how much clock ticks passed since the beggining of its execution and notifying every other microservice (thats intersted) about it using the TickBroadcast.

The TimeService receives the number of milliseconds each clock tick takes (speed:int) toogether with the number of ticks before termination (duration:int) as a constructor arguments.
Be careful that you are not blocking the event loop of the timer micro-service. You can use the Timer class in java to help you with that.

The current time always start from 1.

3.2.2 SellingService

This micro-service handles *PurchaseOrderRequest*. When the *SellingService* receives a *PurchaseOrderRequest*, it handles it by trying to take the required shoe from the storage. If it succedded it creates a recipt, file it in the store and pass it to the client (as the result of completing the PurchaseOrderRequest). If there were no shoes on the requested type on stock, the selling service will send RestockRequest, if the request completed with the value "false" (see ManagementService) the *SellingService* will complete the PurchaseOrderRequest with the value of "null" (to indicate to

the client that the purchase was unsuccessfull). If the client indicates in the order that he wish to get this shoe only on discount and no more discounted shoes are left then it will complete the client request with null result (to indicate to the client that the purchase was unsuccessfull)

3.2.3 WebsiteClientService

This micro-service describes one client connected to the web-site. The *WebsiteClientService* expects to get two lists as arguments to its constructor:

- *purchaseSchedule: List<PurchaseSchedule>* - contains purchases that the client needs to make (every purchase has a corresponding time tick to send the *PurchaseRequest*). The list does not guaranteed to be sorted. Important: The WebsiteClientService will make the purchase on the tick specied on the schedule irrelevant of the discount on that item.
- *wishList: Set<String>* - The client wish list contains name of shoe types that the client will buy only when there is a discount on them (and immidiatly when he found out of such discount). Once the client bought a shoe from its wishlist - he removes it from the list.

In order to get notified when new discount is available, the client should subscribe to the *NewDiscountBroadcast* message. If the client finish receiving all its purchases and have nothing in its wishList it must immidiatly terminate.

3.2.4 ManagementService

service name = "manager"
This micro-service can add discount to shoes in the store and send *NewDiscountBroadcast* to notify clients about them. In order to do so, this service expects to get a list of DiscountSchedule as argument to its constructor (the list does not guaranteed to be ordered).

In addition, the *ManagementService* handles *RestockRequest*s that is being sent by the *SellingService*. Whenever a RestockRequest of a specific shoe type recived the service first check that this shoe type is not already on order (and if it does, it checks that there are enough ordered to give one to the seller) if it doesnt (or the ordered amount was not enough) it will send a *ManufacturingOrderRequest* for (current-tick%5) + 1 shoes of this type, when this order completes - it update the store stock, file the receipt and only then complete the RestockRequest (and not before) with the result of true. If there were no one that can handle the *ManufacturingOrderRequest* (i.e., no factories are available) it will complete the *RestockRequest* with the result false.

For example, assume that in tick=2 the *ManagementService* received a *RestockRequest* of brown-flip-flops, it does not already ordered them so it send a *ManufacturingOrderRequest* for 3 brown-flip-flops which take some time to complete. Now assume that in tick=3 the *ManufacturingOrderRequest* is not yet completed but the *ManagementService* received another *RestockRequest* of brown-flip-flops - since it knows that it ordered 3 brown-flip-flops (1 of which was alredy reserved) it will not make a new order but also not complete the *RestockRequest*. once the *ManufacturingOrderRequest* is completed with a receipt as its result, the *ManagementService* will file the receipt, complete the two *RestockRequest* and add only 1 brown-flip-flops to the store (the one left).

3.2.5 ShoeFactoryService

This micro-service describes a shoe factory that manufacture shoes for the store. This micro-service handles the *ManufacturingOrderRequest* it takes it exactly 1 tick to manufacture a single shoe, this means that completing a ManufacturingOrderRequest of 3 shoes will take it 3 ticks to complete

(starting from tick following the request). When done manufacturing, this micro-service completes the request with a receipt (which has the value "store" in the customer field and "discount" = false). The micro-service cannot manufacture more than one shoe per tick. For example, if on tick=5 the factory received ManufacturingOrderRequest for 3 purple-heel-shoes and in tick=7 it receive another order for 2 orange-flip-flops it will complete the first request on tick 9 and the second on tick 11:

- tick: 5, receive request 1
- tick: 6, manufacture 1 purple-heel-shoe
- tick: 7, manufacture 1 purple-heel-shoe and receive the second request
- tick: 8, manufacture 1 purple-heel-shoe
- tick: 9, completes the first request, manufacture 1 orange-flip-flop
- tick: 10, manufacture 1 orange-flip-flop
- tick: 11, completes the second request

4 Input Files And Parsing

4.1 The JSON Format

All your input files for this assignment will be given as JSON files. You can read about JSONs syntax here and here.
In Java, there are a number of different options for parsing JSON files. Our recommendation is to use the library *Gson*.
See the Gson User Guide and APIs to see how to work with Gson. There are a lot of informative examples.

4.2 Execution File

Defined below is a json input file that describes a single execution of the our shoe store, The following is an example of that file.

```
{
    "initialStorage": [
       {shoeType: "red-boots", amount: 10},
       {shoeType: "green-flip-flops", amount: 7}
     ],
     services: {
       time: {
        speed: 1000,
        duration: 24
       },
       manager: {
          discountSchedule: [
            {shoeType: "red-boots", amount: 1, tick: 3},
            {shoeType: "green-flip-flops", amount: 3, tick:10}
          ]
       },

       factories: 3,
       sellers: 2,
       customers: [
          {
            name: "Bruria",
            wishList: ["green-flip-flops"],
            purchaseSchedule: [
              {shoeType: "red-boots", tick: 3}
            ]
          },
          {
            name: "Shraga",
            wishList: [],
            purchaseSchedule: [
              {shoeType: "green-flip-flops", tick: 12}
            ]
          }
       ]
     }
}
```

The file holds a json object which contains the following fields:

- *initialStorage: array* an array contains shoes toogether with their amount that should be in stock when the execution starts
- *services: object* an object describing the services that should run on the execution. It contains the following fields
  - *time*: an object contains the arguments for the *TimeService* constructor.

- *manager*: an object contains the arguments for the *ManagementService* constructor.
- *factories*: a number represents the number of *ShoeFactoryService* to start, for each *ShoeFactoryService* started, its name will be "factory i" where i is the number of factory (i.e., if factories = 2, there will be 2 new *ShoeFactoryService*s one with the name "factory 1" and one with the name "factory 2").
- *sellers*: a number represents the number of *SellingService* to start, for each *SellingService* started, its name will be "seller i" where i is the number of seller (i.e., if sellers = 2, there will be 2 new *SellingService*s one with the name "seller 1" and one with the name "seller 2").
- *clients*: an array, each item in the array represents the constructor arguments for an *WebsiteClientService* that needs to be started.

You can expect the input file to always be in the correct format.

5 Program Execution

You should create a class called ShoeStoreRunner with a main function inside. When started, it should accept as argument (command line argument) the name of the json input file to read - there are some example input files in the files attched to this work and you can create more yourself). The ShoeStoreRunner should read the input file (using Gson), it then should add the initial storage to the store and create and start the micro-services. When the current tick number is larger than the duration given to the *TimeService* in the input file all the micro-services should gracefully terminate themselves. You must make sure that micro-services will not miss the first TickBroadcast (because they was not started yet).

After all the micro-services terminate themselves the ShoeStoreRunner should call the *Store*'s print function and exit. Important: all the threads in the system should be terminated gracefully.

6 Requirements

6.1 Application Progress: Logger

Logging is the process of writing log messages during the execution of a program to a central place. This logging allows you to report and persists error and warning messages as well as info messages (e.g. runtime statistics) so that the messages can later be retrieved and analyzed.
Loggers are a great asset especially for a thread oriented application. See guide here, on how to use a logger appropriately.
Your application needs to implement a logger mostly for informative reasons, and to follow the progress of the application and its different threads. Please print logging data to the screen in order to explain the progress of your application to the grader. Make your logger print informative and readable text, so it is easier to follow and understand. The better your logger is, the easier the grader can understand your application. So be sure to invest some time to decide where and what to print your logging information.

6.2 Building the Application: Maven

Use the following command to install maven in the lab:

```
/home/studies/course/spl161/java/install-maven
```

Please do not attempt to install it in your lab any other way.

In this assignment you are going to use maven as your build tool. Maven is the de-facto java build tool. Infact, maven is much more than a simple build tool, it described by its authors as a software project management and comprehension tool. You should read a little about how to use it, you can find a short tutorial here. IDEs such as eclipse, netbeans or intellij all have native support for maven and may simplify interaction with it - but you should learn yourself how.

Maven is a large tool, in order to make your life easier, you have (in the code attached to this assignment) a maven *pom.xml* file that you can use to get started - you should learn what it is and how you can add dependancies to it.

In this assignment you required to work with java 8 (1.8), You will need to have JDK8 installed and to configure your maven build to use java 8 - you should findout how to acheive that yourself.

6.3 Deadlocks, Waiting, Liveness and Completion

Deadlocks You should identify possible deadlock scenarios and prevent them from happening. Be prepared to explain to the grader what possible deadlocks you have identified, and how you have handled them.

Waiting You should understand where wait cases may happen in the program, and how you've solved these issues.

Liveness Locking and synchronization are required for your program to work properly. Make sure not to lock too much so that you don't damage the liveness of the program. Remember, the faster your program completes its tasks, the better.

textbfCompletion As in every multi-threaded design, special attention must be given to the shut down of the system. When the CLI service receives a "shutdown" command, the program needs to terminate. This needs to be done *gracefully*, not abruptly.

6.4 Unit Tests

Testing is an important part of deveolping. It helps us make sure our project behaves as we expect it to. This is why in this assignment, you will use Junit to test your project. You are required to write unit tests for the classes in your project, You must do this for **at least** the following classes:

- Store
- MessageBus

You may add new helper methods in order to successfully write your tests, but you *must not* change any of the required methods given in the interfaces provided to you in the assignment description.

6.5 Documentation and Coding Style

There are many programming practices that **must** be followed when writing any piece of code.

- Follow Java Programming Style Guidelines. A part of your grade will be checking if you have followed these mandatory guidelines or not. It is important to understand that programming is a team oriented job. If your code is hard to read, then you are failing to be a productive part of any team, in academics research groups, or at work. For this, you must follow these guidelines which make your code easier to read and understand by your fellow programmers as well as your graders.
- Do not be afraid to use long variable and method names as long as they increase clarity.
- Avoid code repetition - In case where you see yourself writing same code block in two different places, it means this code must be put in a private function, so both these places may use it.
- Full documentation of the different classes and their public and protected methods, by following Javadoc style of documentation. You may, if you wish, also document your local methods but they are not mandatory.
- Add comments for code blocks where understanding them without, may be difficult.
- Your files must not include commented out code. This is garbage. So once you finish coding, make sure to clean your code.
- Long functions are frowned upon (30+ lines). Long functions mean that your function is doing too many tasks, which means you can move these tasks to their own place in private functions, and use them as appropriate. This is an important step toward increased readability of your code. This is to be done even in cases where the code is used once.
- Magic numbers are bad! Why? Your application must not have numbers throughout the code that need deciphering.
- Do *not* send collection of items to constructors. They can reveal internal implementation. Instead, create a method for the object which adds one item to the object. Example: instead of adding a collection of items to the database, we define the function addProduct to add a single product.

Your implementation must follow these steps in order to keep the code ordered, clean, and easy to read.

7 Submission Instructions

- Submission is done *only* in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You must submit one .tar.gz file with all your code. The file should be named "assignment2.tar.gz".
  Note: We require you to use a .tar.gz file. *Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.*

- The submitted zip should contain the *src* folder and the *pom.xml* file only! no other files are needed.
- Extension requests are to be sent to majeek. Your request email must include the following information:
  - Your name and your partner's name.
  - Your id and your partner's id.
  - Explanation regarding the reason of the extension request.
  - Offcial certification for your illness or army drafting.

  Requests without a compelling reason will not be accepted.

8 Grading

Although you are free to work wherever you please, assignments will be checked and graded on Computer Science Department Lab Computers - so be sure to test your program thoroughly *on them* before your final submission. "But it worked fine on my windows based home computer" will not earn you any mercy points from the grading staff if your code fails to execute at the lab.
Grading will tackle many points, including but not limited to:

- Your application design and implementation.
- Automatic tests will be run on your application. Your application must complete successfully and in reasonable time.
- Liveness and Deadlock, causes of deadlock, and where in your application deadlock might have occurred, without your solution to the problem.
- Synchronization, what is it, where have you used it, and a compelling reason behind your decisions, points will be reducted for overusing of synchronization.
- Checking if your implementation follows the guidelines detailed in "Documentation and Coding Style"6.5

9 Questions and Assistance

- The assignment forum.
- We will *not* answer emails related to the assignment. Please refrain from sending them and post them in the forum instead (so that everybody can see the answer).
- F.A.Q will be updated at the website of assignment 2. It is **mandatory** to read the F.A.Q **daily** for updates. All changes noted there are **binding**.
- We will visit the labs and help the srudents at the times published on the website.

10 Appendix

10.1 Example Code

For your benefit, the assignment also includes an example application code in the *bgu.spl.mics.example* package. You can use the example code to check your framework in part one and even as a reference for part two. This code is only for testing - it deliberately does not follow this work guideline. You may read the code yourself and understand what it does (google is your friend). **No questions about how this code works will be answered**.

To run the example code once you complete implementing the framework, run the class ExampleManager which waits for one of the following commands:

- *start <service-type> <service-name> <service-args>...*
  starts a service of type *<service-type>*, supplies it with the name *<service-name>* and the arguments *<service-args>....* The supported services are:
  - *req-handler* - Expects a single nummeric argument which we will refer to as its "mbt" (messages before termination). The command will start the ExampleRequestHandlerService which subscribe to the ExampleRequest message and complete delivered requests. After completing "mbt" requests it will terminate itself.
  - *brod-listener* - Expects a single nummeric argument which we will refer to as its "mbt" (messages before termination). The command will start the ExampleBroadcastListenerService which subscribe to the ExampleBroadcast message. After receiving "mbt" messages it will terminate itself.
  - *sender* - Expects a single string argument "mode" which can either be "broadcast" or "request". The command will start the ExampleMessageSenderService which, in the case "mode" was "request", publishes a single ExampleRequest, awaits for it to complete and terminates. In the case where "mode" was "broadcast" the sender publishes a single ExampleBroadcast and terminates.
- *quit*
  will *ungracefully* terminate the program.

For example, the following is the output of a simple (and incomplete) test for your round-robin code:

```
Example manager is started - supported commands are: start,quit
Supporting services: [sender, req-handler, brod-listener]

> start req-handler A 10
Request Handler A started

> start req-handler B 1
Request Handler B started

> start sender D request
Sender D started
Sender D send a request and wait for its completion
Request Handler A got a new request from D! (mbt: 9)
Sender D got notified about request completion with result
        "Hello from A"- terminating

> start req-handler C 10
Request Handler C started

> start sender D request
Sender D started
Sender D send a request and wait for its completion
Request Handler B got a new request from D! (mbt: 0)
Request Handler B terminating.
Sender D got notified about request completion with result
        "Hello from B" - terminating

> start sender D request
Sender D started
Sender D send a request and wait for its completion
Request Handler C got a new request from D! (mbt: 9)
Sender D got notified about request completion with result
        "Hello from C" - terminating

> start sender D request
Sender D started
Sender D send a request and wait for its completion
Request Handler A got a new request from D! (mbt: 8)
Sender D got notified about request completion with result
        "Hello from A" - terminating
```