Student ID_____**Student Name**_____

Advanced Software Development DE Final Exam October 5 2019

PRIVATE AND CONFIDENTIAL

1. **Allotted exam duration is 2 hours.**
2. **Closed book/notes.**
3. **No personal items including electronic devices (cell phones, computers, calculators, PDAs).**
4. **Cell phones must be turned in to your proctor before beginning exam.**
5. **No additional papers are allowed. Sufficient blank paper is included in the exam packet.**
6. **Exams are copyrighted and may not be copied or transferred.**
7. **Restroom and other personal breaks are not permitted.**
8. **Total exam including questions and scratch paper must be returned to the proctor.**

5 blank pages are provided for writing the solutions and/or scratch paper. All 5 pages must be handed in with the exam

BE VERY CAREFUL WITH THE GIVEN 2 HOURS AND USE YOUR TIME WISELY. THE ALLOTED TIME IS GIVEN FOR EVERY QUESTION.

Write your name and student id at the top of this page.

Question 1 [20 points] {30 minutes}

Given is the following Package class:

```
public class Package {
	private int length;
	private int width;
	private int height;
	private int weight;

	public Package(int length, int width, int height, int weight) {
		this.length = length;
		this.width = width;
		this.height = height;
		this.weight = weight;
	}

	public int getLength() {
		return length;
	}

	public void setLength(int length) {
		this.length = length;
	}

	public int getWidth() {
		return width;
	}

	public void setWidth(int width) {
		this.width = width;
	}

	public int getHeight() {
		return height;
	}

	public void setHeight(int height) {
		this.height = height;
	}

	public int getWeight() {
		return weight;
	}

	public void setWeight(int weight) {
		this.weight = weight;
	}

}
```

One problem with this class that if we use it like this:

```
Package package = new Package(24, 19, 15, 20);
```

Then we don't know what these numbers `24, 19, 15, 20` mean. It is also easy to make mistakes with them. Another problem is that this class is mutable.

Rewrite this Package class so that

1. The package class is immutable
2. We can create a package class with self-explaining code so that it is clear from the code what the numbers 24, 19, 15, 20 mean.

Write both the code of the Package class, and the code that creates a package with length=24, width=19, height=15, weight=20.

```
public class Package {
	private int length;
	private int width;
	private int height;
	private int weight;
	public static class Builder {
		private int length=0;
		private int width=0;
		private int height=0;
		private int weight=0;

		public Builder withLength(int length) {
			this.length = length;
			return this;
		}
		public Builder withWidth(int width) {
			this.width = width;
			return this;
		}
		public Builder withHeight(int height) {
			this.height = height;
			return this;
		}
		public Builder withWeight(int weight) {
			this.weight = weight;
			return this;
		}

		public Package build() {
			return new Package(this);
		}
	}
	public Package(Builder builder) {
		this.length = builder.length;
		this.width = builder.width;
		this.height = builder.height;
		this.weight = builder.weight;
	}
	public int getLength() {
		return length;
	}
	public int getWidth() {
		return width;
	}
	public int getHeight() {
		return height;
	}
	public int getWeight() {
		return weight;
	}
}
```

```
	Package package1 = new Package.Builder()
			.withLength(24)
			.withHeight(15)
			.withWidth(19)
			.withWeight(20)
			.build();
```

Question 2 [35 points] {50 minutes}

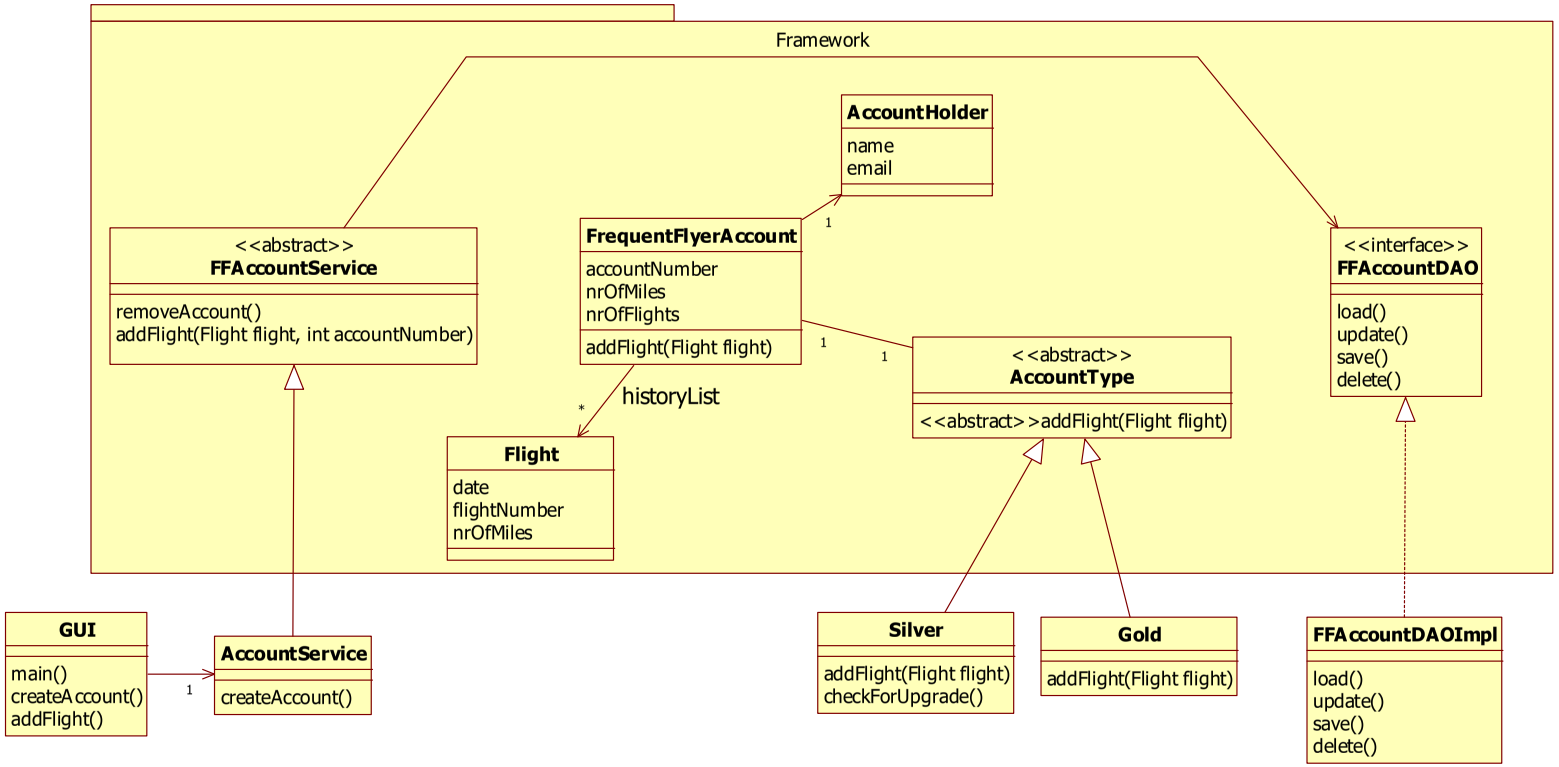
Your company writes software for different airlines, and airline A requests you to develop an Frequent Flyer Program application with the following requirements:
There are 2 types of accounts, “silver” and “gold”.
Everyone starts with a “silver” account.
When you have more than 10.000 miles or more than 15 flights, you are upgraded to a “gold” account.
Silver accounts receive the same number of miles as the actual miles of their flights.
Gold accounts receive 2 times the number of miles as the actual miles of their flights.

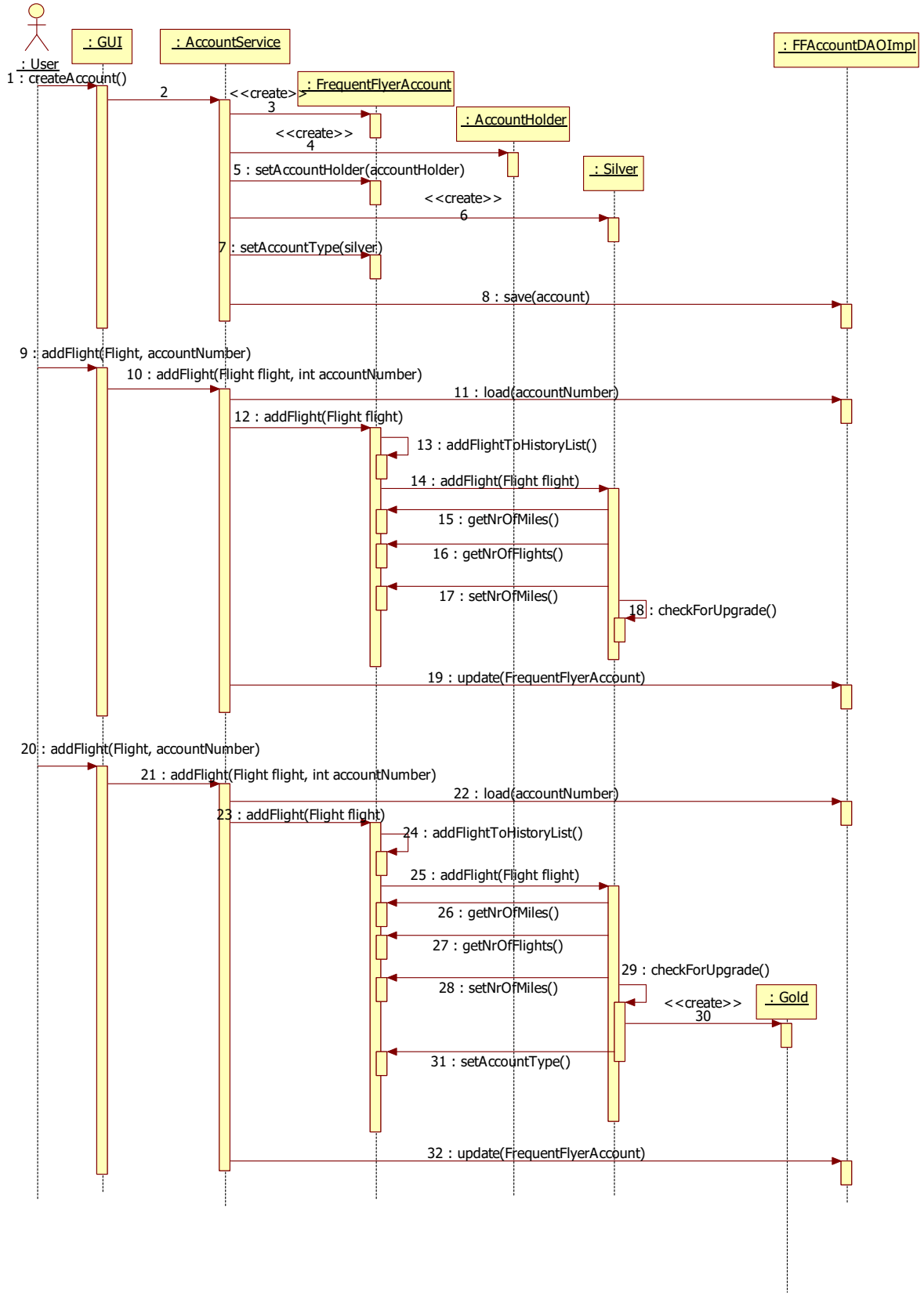
Another airline, airline B, also wants a Frequent Flyer Program application, but it wants 3 account types: basic, medium and advanced. It has also different business rules regarding upgrades and bonus miles.
It is very likely that other airlines also want a Frequent Flyer Program application.

You decide to write a generic Frequent Flyer Program framework with the following requirements:

- The framework should support functionality for adding and removing accounts
- The framework should support functionality for adding miles to an account
- The framework keeps track of the history of all flights of the accountholder with the corresponding date, flightNumber and nrOfMiles.
- An account has the following properties: accountNumber, nrOfMiles, name of the accountholder, email of the accountholder.
- All accounts are stored in the database
- It should be easy to add new account types with its own business rules regarding upgrades and bonus miles.

a. Draw the class diagram of your Frequent Flyer Program framework. **Make sure you add all necessary UML elements (interfaces, abstract classes, attributes, methods, multiplicity, etc) to communicate the important parts of your design.** Make use of all the best practices we learned in this course.
b. Within the class diagram of part a, draw the necessary classes that you need to implement the Frequent Flyer Program application for **airline A**. Show clearly which classes are within the framework and which classes are outside the framework.
c. Draw the sequence diagram with the following scenario for **airline A**:
    1. A new account is created for customer 1.
    2. Customer 1 travels 9.000 miles, and these miles are added to her account.
    3. Customer 1 travels 3.000 miles, and these miles are added to her account.


Framework
AccountHolder
name
email
<<abstract>>
FFAccountService
removeAccount()
addFlight(Flight flight, int accountNumber)
FrequentFlyerAccount
accountNumber
nrOfMiles
nrOfFlights
addFlight(Flight flight)
<<interface>>
FFAccountDAO
load()
update()
save()
delete()
<<abstract>>
AccountType
<<abstract>>addFlight(Flight flight)
historyList
Flight
date
flightNumber
nrOfMiles
GUI
main()
createAccount()
addFlight()
AccountService
createAccount()
Silver
addFlight(Flight flight)
checkForUpgrade()
Gold
addFlight(Flight flight)
FFAccountDAOImpl
load()
update()
save()
delete()
1
1
1
1
*
1


: User
: GUI
: AccountService
: FrequentFlyerAccount
: AccountHolder
: Silver
: FFAccountDAOImpl
1 : createAccount()
2
<<create>>
3
<<create>>
4
5 : setAccountHolder(accountHolder)
<<create>>
6
7 : setAccountType(silver)
8 : save(account)
9 : addFlight(Flight, accountNumber)
10 : addFlight(Flight flight, int accountNumber)
11 : load(accountNumber)
12 : addFlight(Flight flight)
13 : addFlightToHistoryList()
14 : addFlight(Flight flight)
15 : getNrOfMiles()
16 : getNrOfFlights()
17 : setNrOfMiles()
18 : checkForUpgrade()
19 : update(FrequentFlyerAccount)
20 : addFlight(Flight, accountNumber)
21 : addFlight(Flight flight, int accountNumber)
22 : load(accountNumber)
23 : addFlight(Flight flight)
24 : addFlightToHistoryList()
25 : addFlight(Flight flight)
26 : getNrOfMiles()
27 : getNrOfFlights()
29 : checkForUpgrade()
28 : setNrOfMiles()
<<create>>
30
: Gold
31 : setAccountType()
32 : update(FrequentFlyerAccount)

Question 3 [20 points] {20 minutes}

In the lab we wrote our own test framework. Suppose all classes we want to test are annotated with **@Service** annotation:

```
public interface Calculator {
        public void reset() ;
        public int add(int newValue);
        public int subtract(int newValue);
}

@Service
public class CalculatorImpl implements Calculator {
        private int calcValue=0;

        public void reset() {
                calcValue=0;
        }

        public int add(int newValue) {
                calcValue=calcValue+newValue;
                return calcValue;
        }

        public int subtract(int newValue) {
                calcValue=calcValue-newValue;
                return calcValue;
        }
}
```

Our framework should now support the following tests classes:

```
@TestClass
public class MyTest {
        @Inject  //calculator is injected by the testframework
        Calculator calculator;

        @Before
        public void init() {
                calculator = new CalculatorImpl();
        }
        @Test
        public void testMethod1() {
                assertEquals(calculator.add(3),3);
                assertEquals(calculator.add(6),9);
        }
        @Test
        public void testMethod2() {
                assertEquals(calculator.add(3),3);
                assertEquals(calculator.subtract(6),-1);
        }

}
```

Our framework needs a framework context, and ee can start this framework with the following code:

```
public class Application {

	public static void main(String[] args) {
		FWContext fWContext = new FWContext();
		fWContext.start();
	}
}
```

Now all test methods in the test classes should be executed.

You can assume that the framework contains the following Asserts class:

```
public class Asserts {

	public static void assertEquals(int x, int y) {
		if (x != y)
			System.out.println("Fail: result = "+x+" but expected "+y);
	}
}
```

You can also assume that the framework contains all the definitions for the annotations (@Inject, @Before, etc)

The only thing we still need to do for this test framework is writing the **FWContext** class.

Given is the partial pseudo code of this FWContext class. Write the pseudo code of the **performDI()** and **start()** method so that the framework does the necessary tasks. Write the pseudo code in a similar way as the pseudo code given in the constructor and the method getServiceBeanOftype().

```
public class FWContext {

	private static List<Object> testObjects = new ArrayList<>();
	private static List<Object> serviceObjects = new ArrayList<>();

	public FWContext() {
	  find and instantiate all classes annotated with the @TestClass annotation
	          and add them to the testObjects list
	  find and instantiate all classes annotated with the @Service annotation
	          and add them to the serviceObjects list
	}

	private void performDI() {
	       loop over all classes in testObjects
	         for every testObject loop over its fields
	           if the field is annotated with @inject {
	             Then get the type of the field
	             Then call getServiceBeanOftype to find the object of this type
	             Set the field to accessible
	             Do the injection by setting the field to the found object
	           }
	}

	public Object getServiceBeanOftype(Class interfaceClass) {
	       loop over serviceObjects list
	       for every serviceObject, get its interfaces
	       loop over the interfaces
	       if (interfaces is equal to interfaceClass.getName()){
	         return current serviceObject
	       }
	}

	public void start() {
	       loop over all classes in testObjects
	         For every testObject loop over its methods
	           If the method is annotated with @before then beforemethod =
	       method.
	         Loop again over its methods
	          If the method is annotated with @test then{
	            Invoke the before method if available
	            Invoke the test method
	          }

	}
```

Question 4 [20 points] {20 minutes}

a. Given is the following code:

```
public class Application implements CommandLineRunner {

      private CustomerService customerService = new
                                          CustomerService();

      public static void main(String[] args) {
            SpringApplication.run(Application.class, args);
      }


      public void run(String... args) throws Exception {
            customerService.addCustomer(new Customer("Frank Brown",
                                          "fbrown@gmail.com"));
      }
}

public class Customer {
      private String name;
      private String emailAddress;

      public Customer(String name, String emailAddress) {
            this.name = name;
            this.emailAddress = emailAddress;
      }
      // getter and setter methods are not shown
}


public class CustomerService{

      CustomerDAO customerDAO = new CustomerDAO();


      EmailSender emailSender = new EmailSender();

      public void addCustomer(Customer customer) {
            customerDAO.save(customer);
            emailSender.sendEmail(customer.getEmailAddress(), "Welcome
                                          new customer");
      }
}
```

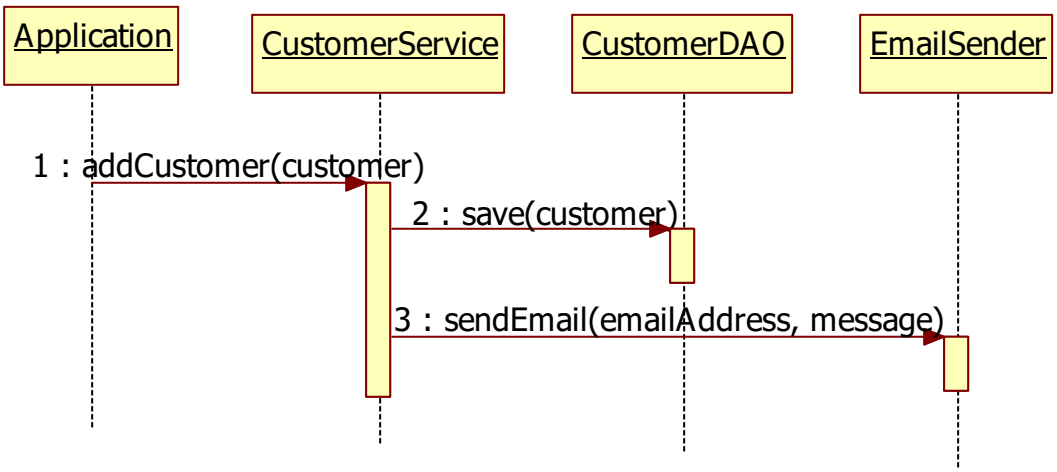
```
public interface ICustomerDAO {
      void save(Customer customer);
}


public class CustomerDAO implements ICustomerDAO {
    public void save(Customer customer) {
      System.out.println("Saving customer " +customer.getName());
    }
}


public interface IEmailSender {
      void sendEmail(String emailAddress, String message);
}


public class EmailSender implements IEmailSender {
  public void sendEmail(String emailAddress, String message) {
        System.out.println("Send email with message="+message+
                    " to "+emailAddress);
  }
}
```

Modify the given code so that this becomes a Spring Boot application that works as follows:



Do not write the code again (that takes too much time), but modify the given code. **One requirement is that it should be easy to use a different CustomerDAO or EmailSender**. Make sure you show all required changes.

```java
@SpringBootApplication
public class Application implements CommandLineRunner {
	@Autowired
	private CustomerService customerService;

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}


	public void run(String... args) throws Exception {
		customerService.addCustomer(new Customer("Frank Brown",
					"fbrown@gmail.com"));
	}
}

@Service
public class CustomerService{
	@Autowired
	CustomerDAO customerDAO ;

	@Autowired
	EmailSender emailSender;

	public void addCustomer(Customer customer) {
		customerDAO.save(customer);
		emailSender.sendEmail(customer.getEmailAddress(), "Welcome
			new customer");
	}
}

@Repository
public class CustomerDAO implements ICustomerDAO {
    public void save(Customer customer) {
     System.out.println("Saving customer " +customer.getName());
    }
}



@Component
public class EmailSender implements IEmailSender {
	public void sendEmail(String emailAddress, String message) {
		System.out.println("Send email with message="+message+
				" to "+emailAddress);
	}
}
```

b. Suppose we add the following class to the Spring boot application of part a.

```
@Aspect
@Configuration
public class StopWatchAdvice {

  @Around("execution(* customers.CustomerDAO.*(..))")
  public Object invoke(ProceedingJoinPoint call) throws Throwable {
      StopWatch sw = new StopWatch();
      sw.start(call.getSignature().getName());
      Object retVal = call.proceed();
      sw.stop();

      long totaltime=sw.getTotalTimeMillis();
       System.out.println("Time to execute
         "+call.getSignature().getName()+" = "+totaltime+" ms");

      return retVal;
      }
}
```

Draw in a new sequence diagram what happens if I run this application again. Show **all** object that are involved on your sequence diagram.