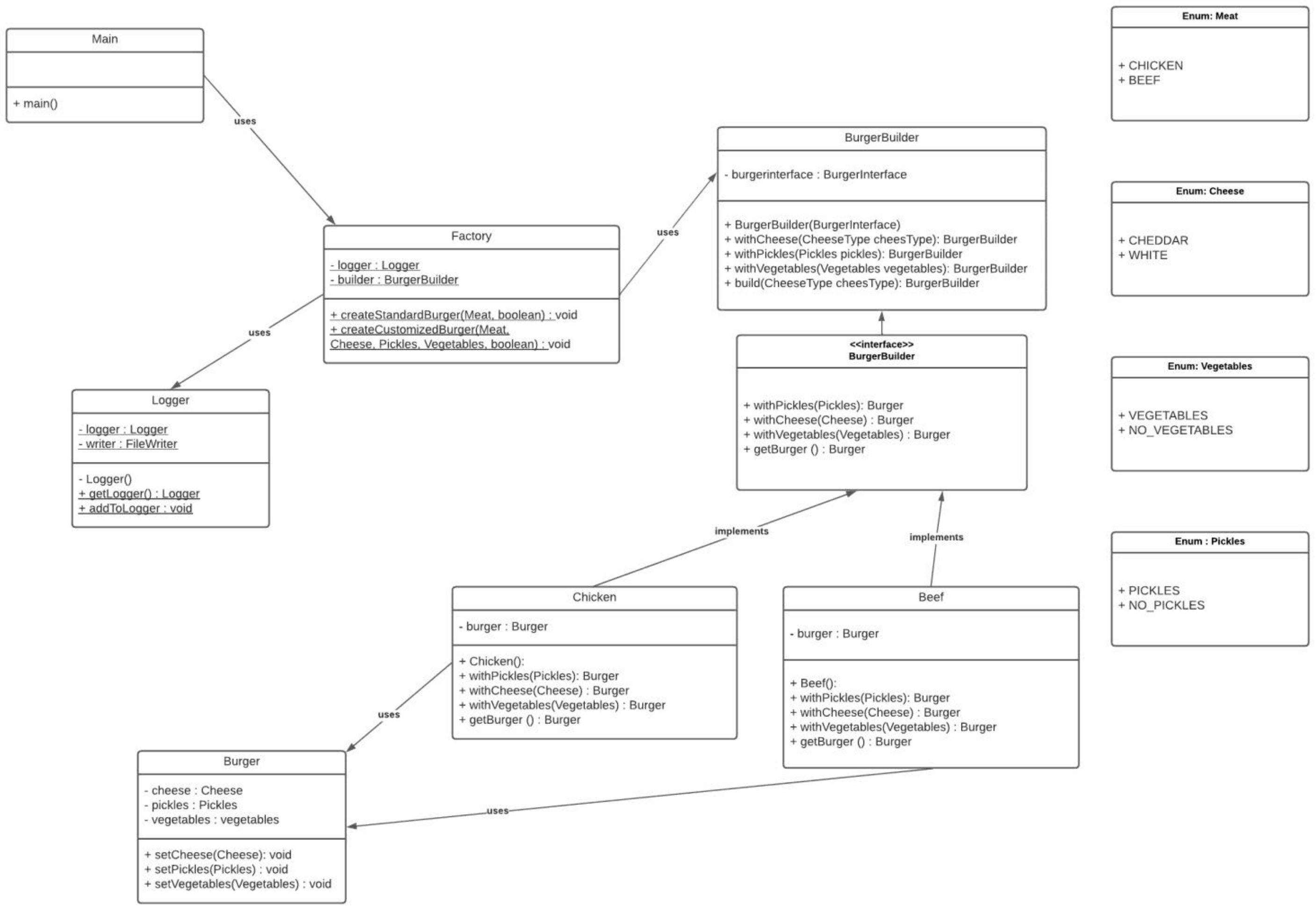
DESIGN PATTERNS - CPIT252

HOMEWORK_1

Shehab

CONTENTS

1- UML DIAGRAM



Main
+ main()
uses
Factory
- logger : Logger
- builder : BurgerBuilder
+ createStandardBurger(Meat, boolean) : void
+ createCustomizedBurger(Meat, Cheese, Pickles, Vegetables, boolean) : void
uses
Logger
- logger : Logger
- writer : FileWriter
- Logger()
+ getLogger() : Logger
+ addToLogger : void
uses
BurgerBuilder
- burgerinterface : BurgerInterface
+ BurgerBuilder(BurgerInterface)
+ withCheese(CheeseType cheesType): BurgerBuilder
+ withPickles(Pickles pickles): BurgerBuilder
+ withVegetables(Vegetables vegetables): BurgerBuilder
+ build(CheeseType cheesType): BurgerBuilder
<<interface>> BurgerBuilder
+ withPickles(Pickles): Burger
+ withCheese(Cheese) : Burger
+ withVegetables(Vegetables) : Burger
+ getBurger () : Burger
implements
implements
Chicken
- burger : Burger
+ Chicken():
+ withPickles(Pickles): Burger
+ withCheese(Cheese) : Burger
+ withVegetables(Vegetables) : Burger
+ getBurger () : Burger
Beef
- burger : Burger
+ Beef():
+ withPickles(Pickles): Burger
+ withCheese(Cheese) : Burger
+ withVegetables(Vegetables) : Burger
+ getBurger () : Burger
uses
uses
Burger
- cheese : Cheese
- pickles : Pickles
- vegetables : vegetables
+ setCheese(Cheese): void
+ setPickles(Pickles) : void
+ setVegetables(Vegetables) : void
Enum: Meat
+ CHICKEN
+ BEEF
Enum: Cheese
+ CHEDDAR
+ WHITE
Enum: Vegetables
+ VEGETABLES
+ NO_VEGETABLES
Enum : Pickles
+ PICKLES
+ NO_PICKLES

2- BREIF DESCRIPTION AND ENUMS

The restaurant in this project is a burger restaurant, It only offers two types of burgers. beef and chicken with different addons.

So, we only have 4 enum classes which they are:

1- CHEESE

```java
public enum Cheese {
    CHEDDAR,
    WHITE;

    @Override
    public String toString() {
        return name().toLowerCase();
    }
}
```

2- MEAT

```java
public enum Meat {
    CHICKEN,
    BEEF;

    @Override
    public String toString() {
        return name().toLowerCase();
    }
}
```

3- PICKLES

```java
public enum Pickles {
    PICKLES,
    NO_PICKLES;

    @Override
    public String toString() {
        return name().toLowerCase();
    }
}
```

4- VEGETABLES

```java
public enum Vegetables {
    VEGETABLES,
    NO_VEGETABLES;

    @Override
    public String toString() {
        return name().toLowerCase();
    }
}
```

3- MAIN CLASS

```
public static void main(String[] args) {
    BurgerFactory.createStandardBurger(Meat.BEEF,true);

    BurgerFactory.createCustomizedBurger(Meat.BEEF,Cheese.CHEDDAR
           ,Pickles.PICKLES,Vegetables.VEGETABLES,false);
   }
}
```

Main class uses **BurgerFactory** class to create either a customized burger where users can choose the adds on based on their taste, or they can use the standard burger.

4- BURGERFACTORY CLASS

```
public class BurgerFactory {
    private static Logger logger = Logger.getLogger();
    private static BurgerBuilder builder;
    public static void createStandardBurger(Meat meat,boolean loggerStatus) {

        switch (meat) {
            case BEEF:
                builder = new BurgerBuilder(new Beef()).withCheese(Cheese.CHEDDAR)
                     .withPickles(Pickles.PICKLES).withVegetables
                     (Vegetables.VEGETABLES)
                     .build();
                logger.addToLogger("\t*\tBurger is "
                     + Meat.BEEF + "\n\t*\tCheese is " + Cheese.CHEDDAR
                     + "\n\t*\tWith " + Pickles.PICKLES + "\n\t*\tWith "
                     + Vegetables.VEGETABLES);
                break;
            case CHICKEN:
                builder = new BurgerBuilder(new Chicken())
                     .withCheese(Cheese.WHITE)
                     .withPickles(Pickles.NO_PICKLES)
                     .withVegetables(Vegetables.VEGETABLES).build();
                logger.addToLogger("\t*\tBurger is "
                     + Meat.CHICKEN + "\n\t*\tCheese is "
                     + Cheese.WHITE + "\n\t*\tWith " + Pickles.NO_PICKLES
                     + "\n\t*\tWith " + Vegetables.VEGETABLES);
        }
        if(!loggerStatus)
            Logger.close();
    }
    public static void createCustomizedBurger(Meat meat,
          Cheese cheese, Pickles pickles,
          Vegetables vegetables, boolean loggerStatus){
        if (meat.equals(Meat.BEEF)) {
            builder = new BurgerBuilder(new Beef()).withCheese(cheese)
                    .withPickles(pickles).withVegetables(vegetables).build();
                     logger.addToLogger("\t*\tBurger is " + Meat.BEEF
                     + "\n\t*\tCheese is " + cheese
                    + "\n\t*\tWith " + pickles + "\n\t*\tWith " + vegetables);
        }
        else {
            builder = new BurgerBuilder(new Chicken()).withPickles(pickles)
                     .withVegetables(vegetables).build();
            logger.addToLogger("\t*\tBurger is "
                     + Meat.BEEF + "\n\t*\tCheese is " + cheese
                     + "\n\t*\tWith " + pickles + "\n\t*\tWith " + vegetables);
        }
        if (!loggerStatus)
            Logger.close();
    }

}
```

The class has 2 methods which they basically use BurgerBuilder Class to create either a standard or customized burgere, I've made the switch statements choose between enums so I don't have to maintain errors, we can also see the **logger** and it's status, I've created the status so if it's true it means user will still use the program so **PrintWriter** object don't close the file using PrintWriter(obj).close(), details will be covered in Logger class.

5- BURGERBUILDER CLASS

```
public class BurgerBuilder {

    private BurgerInterface burgerInterface;

    public BurgerBuilder(BurgerInterface burger) {
        if (burger == null)
            throw new IllegalArgumentException("Meat can not be null");
        else if (burger instanceof Beef) {
            burgerInterface = new Beef();
        } else{
            burgerInterface = new Chicken();
        }
    }

    public BurgerBuilder withCheese(Cheese cheese) {
        burgerInterface.withCheese(cheese);
        return this;
    }

    public BurgerBuilder withPickles(Pickles pickles) {
        burgerInterface.withPickles(pickles);
        return this;
    }

    public BurgerBuilder withVegetables(Vegetables vegetables) {
        burgerInterface.withVegetables(vegetables);
        return this;
    }

    public BurgerBuilder build() {
        return this;
    }
}
```

This class accepts any class that has implemented **BurgerInterface** interface in its constructor it has 4 methods that use method chaining technique to build the burger.

6- BURGERINTERFACE INTERFACE

```
public interface BurgerInterface {
    public Burger withPickles(Pickles pickles);
    public Burger withCheese(Cheese cheese);
    public Burger withVegetables(Vegetables vegetables);
    public Burger getBurger();
}
```

So, we can stick with how builder design pattern is implemented.

7- BEEF CLASS

```
public class Beef implements BurgerInterface{
    private Burger burger;

    public Beef() {
        burger = new Burger();
    }

    @Override
    public Burger withPickles(Pickles pickles) {
        burger.setPickles(pickles);
        return burger;
    }
    @Override
    public Burger withCheese(Cheese cheese) {
        burger.setCheese(cheese);
        return burger;
    }
    @Override
    public Burger withVegetables(Vegetables vegetables) {
        burger.setVegetables(vegetables);
        return burger;
    }
    @Override
    public Burger getBurger() {
        return burger;
    }
}
```

This one of the two concreate classes we do have; the class do implement the interface and it has only one field which it's **Burger** object so every time a new beef burger is created the constructor does initialize a new burger object.

8- CHICKEN CLASS

```
public class Chicken implements BurgerInterface {
    private Burger burger;

    public Chicken() {
        burger = new Burger();
    }
    @Override
    public Burger withPickles(Pickles pickles) {
        burger.setPickles(pickles);
        return burger;
    }
    @Override
    public Burger withCheese(Cheese cheese) {
        burger.setCheese(cheese);
        return burger;
    }
    @Override
    public Burger withVegetables(Vegetables vegetables) {
        burger.setVegetables(vegetables);
        return burger;
    }
    @Override
  public Burger getBurger() {
        return burger;
    }
}
```

Same as **Beef** class it's just a different meat type.

9- BURGER CLASS

```java
public class Burger {
    private Cheese cheese;
    private Pickles pickles;
    private Vegetables vegetables;

    public void setCheese(Cheese cheese) {
        this.cheese = cheese;
    }

    public void setPickles(Pickles pickles) {
        this.pickles = pickles;
    }

    public void setVegetables(Vegetables vegetables) {
        this.vegetables = vegetables;
    }

}
```

This is the class which has the abstract idea of a burger.

For example, if the project was about vehicles the **Beef** class would be **Toyota** and **Chicken** would be **Ford**. This class would be **Car**.

10- LOGGER (EXTRA)

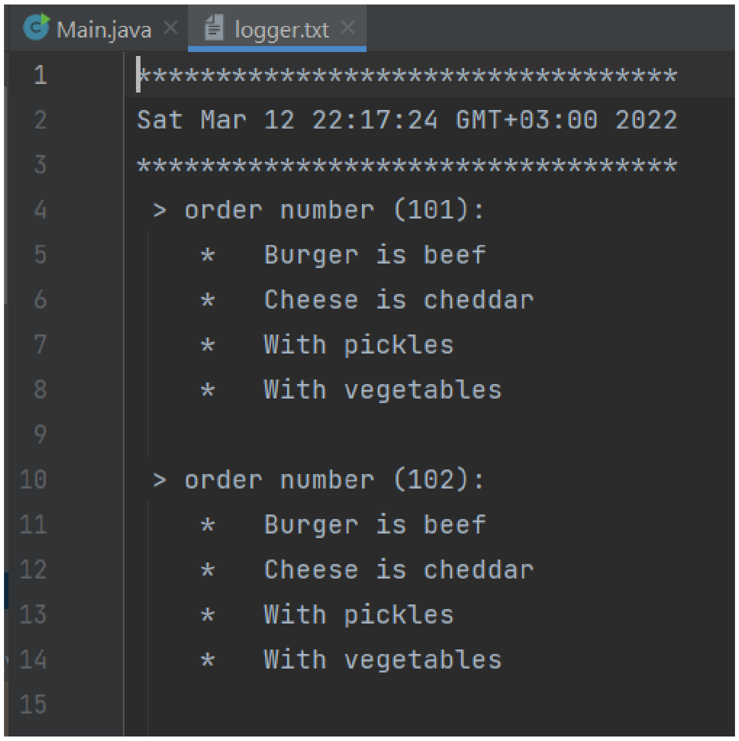
```java
public class Logger {
    private static Logger logger = null;
    private static FileWriter writer;
    int counter = 100;

    private Logger() {
        try {
            writer = new FileWriter("logger.txt",true);
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            writer.write("**********************************\n"+
                    new java.util.Date() + "\n" +
                    "**********************************\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static Logger getLogger() {
        if (logger == null)
            return logger = new Logger();
        else
            return logger;
    }
    public static void close(){
        try {
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void addToLogger(String str) {
        try {
            writer.write(" > order number (" + ++counter + "): \n" + str + "\n\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

as we can see, we do have 3 fields; static logger object, static PrintWriter object and an integer counter and the constructor do tell us that the design we are using is singelton design.

close() method is created to close the PrintWriter object.

addToLogger() is created to add the orders details to the logger file.

And here is the output for the main mehod.



```
**********************************
Sat Mar 12 22:17:24 GMT+03:00 2022
**********************************
 > order number (101):
    *   Burger is beef
    *   Cheese is cheddar
    *   With pickles
    *   With vegetables

 > order number (102):
    *   Burger is beef
    *   Cheese is cheddar
    *   With pickles
    *   With vegetables

```