ASSIGNMENT OP2 REPORT

This assignment aims to create an algorithm for encoding user passwords in the system.

It must make use of design principles (particularly, the SOLID), as well as patterns.

"SOLID" PRINCIPLES:

Definition:

SOLID principles are a set of recommendations that determine the way code should be. They were promoted by Robert C. Martin and they are used across the object-oriented design spectrum, even though the name "SOLID" was given by Michael Feathers.

The idea is to develop software that is easy to maintain and extend. They make it easy for developers to avoid code smells, refactor code easily, and are also a part of the agile or adaptive software development.

SOLID stands for:

- **S** - Single Responsibility Principle
- **O** - Open-Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Now that we know what SOLID stands for, we can take a look at which of these principles has been implemented in the code.

SINGLE RESPONSIBILITY PRINCIPLE:

This principle states that:

> *A class should only have one and only one reason to change, meaning that a class should only have one job.*

The objective of the Single Responsibility Principle (SRP) is to make the code more maintainable, easily modifiable, and very extensible.

As said before, the principle claims that every object should have a single responsibility entirely encapsulated in the class. All its services should be narrowly aligned with that responsibility.

But, what is a responsibility? There is a quote from Robert C. Martin that explains this very well:

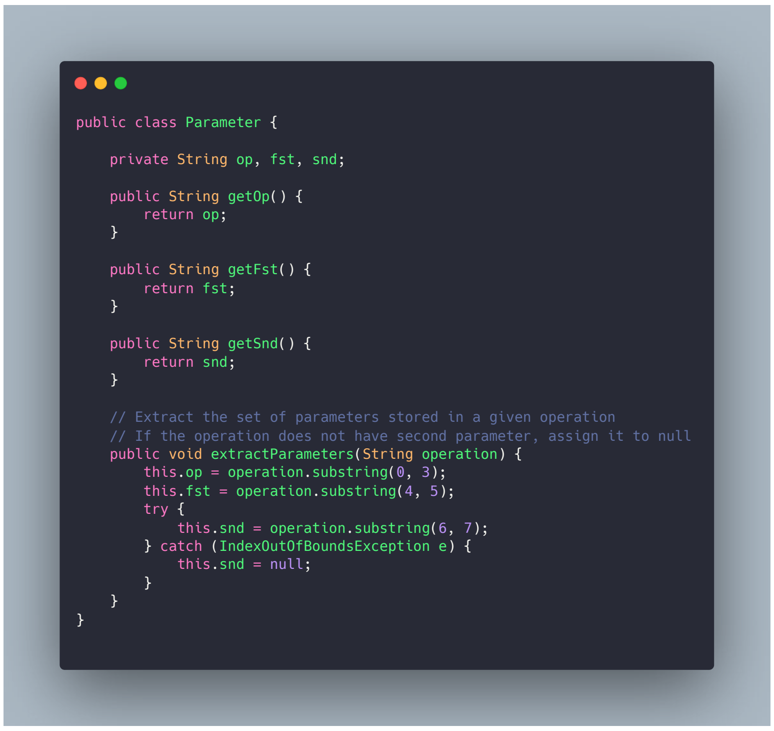
> *Gather together those things that change for the same reason, and separate those things that change for different reasons.*

Where can we see an example of this principle in the code?

If we take a look at any of the classes corresponding to the encoding/decoding operations (SWP, REP, SWL, etc.) we can see that they are only responsible for performing the corresponding encode/decode operation. Despite implementing two contrary methods (encode and decode), it is the same responsibility. But why?

Well, according to Robert, we should gather together those things that change for the same reason. In this case, the method "decode" performs the opposite operation to the method "encode". Consequently, if an operation changed its implementation of the "encode" method, the implementation of the "decode" method would also have to change.

Another example would be the Parameter class, whose only responsibility is to manage the parameters from the operation passed. Every method in the Parameter class is related.



```java
public class Parameter {

    private String op, fst, snd;

    public String getOp() {
        return op;
    }

    public String getFst() {
        return fst;
    }

    public String getSnd() {
        return snd;
    }

    // Extract the set of parameters stored in a given operation
    // If the operation does not have second parameter, assign it to null
    public void extractParameters(String operation) {
        this.op = operation.substring(0, 3);
        this.fst = operation.substring(4, 5);
        try {
            this.snd = operation.substring(6, 7);
        } catch (IndexOutOfBoundsException e) {
            this.snd = null;
        }
    }
}
```

OPEN-CLOSED PRINCIPLE:

This principle states that:

> *A module (class) should be open for extension but closed for modification.*

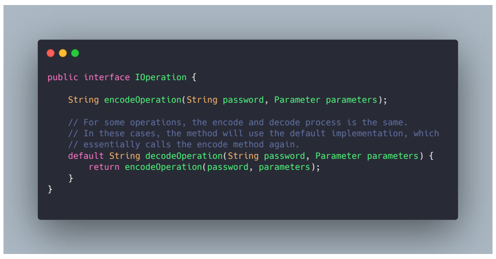
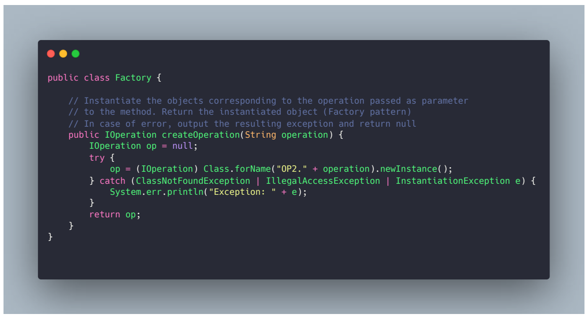
The aim of the Open-Closed Principle (OCP) is to be able to change what the modules do, without changing the source code of the modules. In general terms, modifying a functionality during the life cycle tends to cause several changes to dependent modules, propagating in cascade.

The principle advocates to prevent that from happening.

Where can we see an example of this principle in the code?

The code presented with this report fulfills the Open-Closed principle. Nevertheless, it is important to keep in mind that the current operations only allow one or two parameters, at most. Consequently, there would be no problem when adding a new type of operation. But, if later we added one that uses three or more parameters, we would have to modify the Parameter class, in charge of managing the operation parameters, violating the principle.

As we can see in the code, we have implemented an interface that defines the methods for encoding and decoding. By creating classes that implement this interface we can override its methods to create new operations. This allows us to create as many new operations as we want without needing to make any modifications to the source code of the modules, consequently, fulfilling the principle. The operations are instantiated by the Factory class, which uses reflection to avoid breaking the principle, usually caused by the "if-else" operator.

```java
public interface IOperation {

    String encodeOperation(String password, Parameter parameters);

    // For some operations, the encode and decode process is the same.
    // In these cases, the method will use the default implementation, which
    // essentially calls the encode method again.
    default String decodeOperation(String password, Parameter parameters) {
        return encodeOperation(password, parameters);
    }
}
```

```java
public class Factory {

    // Instantiate the objects corresponding to the operation passed as parameter
    // to the method. Return the instantiated object (Factory pattern)
    // In case of error, output the resulting exception and return null
    public IOperation createOperation(String operation) {
        IOperation op = null;
        try {
            op = (IOperation) Class.forName("OP2." + operation).newInstance();
        } catch (ClassNotFoundException | IllegalAccessException | InstantiationException e) {
            System.err.println("Exception: " + e);
        }
        return op;
    }
}
```

DEPENDENCY INVERSION PRINCIPLE:

This principle states that:

> *Entities must depend on abstractions, not on concretions. High-level modules must not depend on the low-level modules, but they should depend on abstractions.*

Hence, the goal of the Dependency Inversion Principle (DIP) is to decouple higher-level components from their dependency upon lower-level components.

Where can we see an example of this principle in the code?

"High-level modules must not depend on the low-level modules". Translating this statement to the code, we know the high-level module we are referring to is the Scrambler class, which is the one in charge of starting the encoding phase. The low-level modules are the different classes corresponding to the operations that implement the IOperation interface, which is our abstraction.

Entities must depend on abstractions, not on concretions. The interface IOperation is the abstraction, the concretions are its actual implementations, this is, the operation classes.

LISKOV SUBSTITUTION PRINCIPLE:

This principle is very simple, it states that:

> *Subclasses should be substitutable for their base classes.*

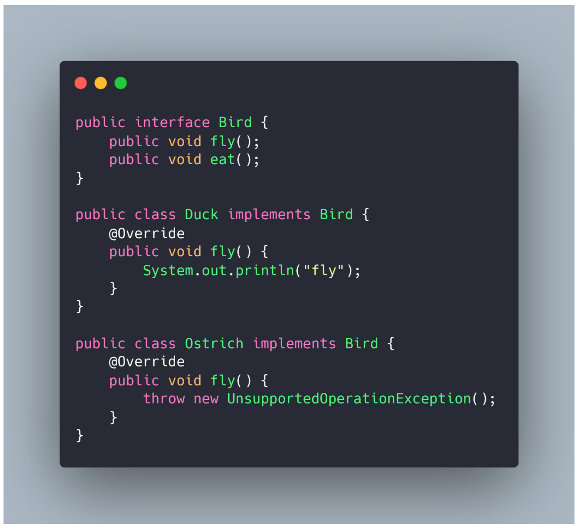
A base class can always be used when it is needed, but that does not guarantee that a subclass preserves the behavior of the base class (behavioral subtype).

The aim of the Liskov Substitution Principle (LSP) is to make code easier to maintain and extend in the future.

Where can we see an example of this principle in the code?

Taking a look at the Scrambler class we can see that we return an IOperation object from the Factory method "createOperation", that we later use to call the encode/decode process. So, even though we are returning a specific operation object from the Factory, we are assigning it to an IOperation object, which works as a base class.

Moreover, due to the assignment approach, we know that every new operation that is added will implement the methods of the IOperation interface in a meaningful way. This will not break the LSP. Let's see an example of an LSP violation:



```java
public interface Bird {
    public void fly();
    public void eat();
}

public class Duck implements Bird {
    @Override
    public void fly() {
        System.out.println("fly");
    }
}

public class Ostrich implements Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException();
    }
}
```

It is known that Ostriches cannot fly, hence when we add an Ostrich class to the program we have a problem implementing the fly method from its interface. Here the subtype (Ostrich) is not replaceable for the supertype (Bird).

Due to the requirements of the assignment, we would never find ourselves facing this kind of situation. Every operation will implement the encode and decode method in a functional way since they are both needed for the program to work correctly.

INTERFACE SEGREGATION PRINCIPLE:

This principle states that:

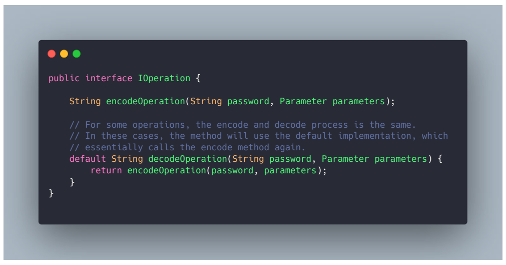
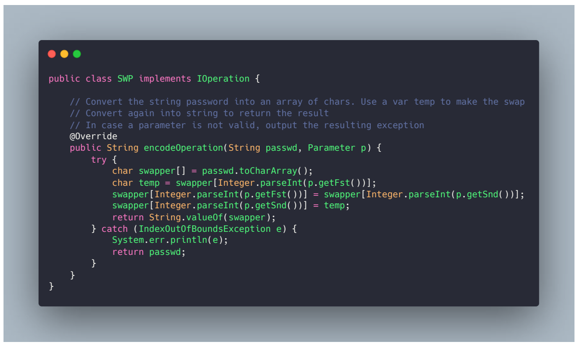
> *Having many client-specific interfaces is better than having one general-purpose interface.*

When clients are forced to use interfaces they do not implement completely, they are subject to changes in that interface. This results in unnecessary coupling between clients.

The goal of the Interface Segregation Principle (ISP) is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

Where can we see an example of this principle in the code?

If we take a look at the interface IOperation, we can see that all classes implementing it override the methods declared there. It is important to remark that there are some classes such as SWP or SWL where the encode and decode methods have the same implementation. In order to simplify the code, we have written a default implementation for the decode method. This way, for the operations where the encode and decode is essentially the same we do not need to override it. Nonetheless, this does not violate the principle since every new operation that will be created in the future is required to implement the encode and decode methods, due to the program requirement to know both how to code and decode the password for a given operation. If we use an operation to encode a password, we need to know how to undo the codification. Consequently, we do not need to split the interface.

```java
public interface IOperation {

    String encodeOperation(String password, Parameter parameters);

    // For some operations, the encode and decode process is the same.
    // In these cases, the method will use the default implementation, which
    // essentially calls the encode method again.
    default String decodeOperation(String password, Parameter parameters) {
        return encodeOperation(password, parameters);
    }
}
```

```java
public class SWP implements IOperation {

    // Convert the string password into an array of chars. Use a var temp to make the swap
    // Convert again into string to return the result
    // In case a parameter is not valid, output the resulting exception
    @Override
    public String encodeOperation(String passwd, Parameter p) {
        try {
            char swapper[] = passwd.toCharArray();
            char temp = swapper[Integer.parseInt(p.getFst())];
            swapper[Integer.parseInt(p.getFst())] = swapper[Integer.parseInt(p.getSnd())];
            swapper[Integer.parseInt(p.getSnd())] = temp;
            return String.valueOf(swapper);
        } catch (IndexOutOfBoundsException e) {
            System.err.println(e);
            return passwd;
        }
    }
}
```

DESIGN PATTERNS:

Definition:

Design patterns are general reusable solutions to commonly occurring problems within a given context. They are like pre-made customizable blueprints to solve a recurring design problem in the code.

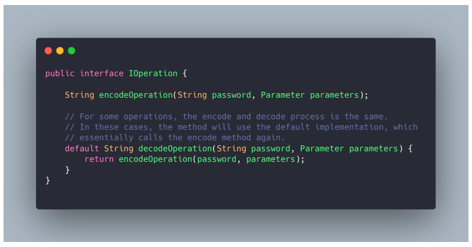
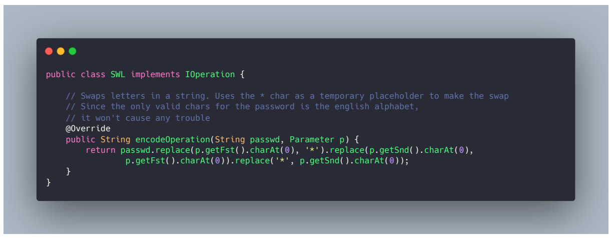
STRATEGY:

Strategy is a behavioral pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Thanks to its implementation we can change between the different operations to encode or decode the password in a simple way. If we would like to add a new operation we just need to create a class with its corresponding implementation. There is no need to change the existing code. This fulfills the Open-Closed Principle.

Where can we see the pattern in the code?

The pattern is made of a Strategy, represented by the interface IOperation in the code, which is common to all concrete strategies. Besides, the concrete strategies, which are represented by the different operations (SWP, REP, MOP, etc.) in the code, implement the methods declared in the IOperation interface.

Strategy	Concrete Strategy
	

```java
public interface IOperation {

    String encodeOperation(String password, Parameter parameters);

    // For some operations, the encode and decode process is the same.
    // In these cases, the method will use the default implementation, which
    // essentially calls the encode method again.
    default String decodeOperation(String password, Parameter parameters) {
        return encodeOperation(password, parameters);
    }
}
```

```java
public class SWL implements IOperation {

    // Swaps letters in a string. Uses the * char as a temporary placeholder to make the swap
    // Since the only valid chars for the password is the english alphabet,
    // it won't cause any trouble
    @Override
    public String encodeOperation(String passwd, Parameter p) {
        return passwd.replace(p.getFst().charAt(0), '*').replace(p.getSnd().charAt(0),
                p.getFst().charAt(0)).replace('*', p.getSnd().charAt(0));
    }
}
```

FACTORY METHOD (PARAMETERIZED):

Factory Method is a creational design pattern that is used to instantiate objects of the same type with different implementations.

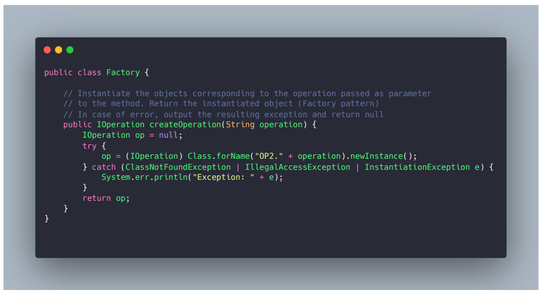
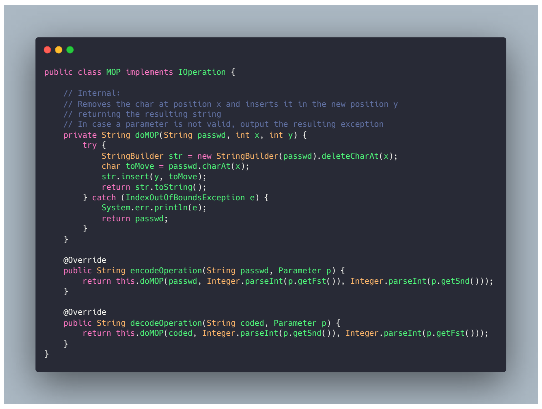
In this particular case, we are using the Factory Method Pattern parameterized, which means objects are created according to a parameter passed to the factory.

We are complementing this pattern with Strategy. The parameterized Factory Method usually breaks the OCP, due to the use of "if-else" operators or "switch". Hence, we will make use of reflection to instantiate the objects created by the factory.

This pattern consists of a "creator class", in the code, the Factory class, that is in charge of instantiating different "concrete products", in our case operations (SWP, REP, etc.), that implement the same interface (IOperation). This way we do not expose the creation logic to the client.

Where can we see the pattern in the code?

If we take a look at the Factory class, we can see there is a method "createOperation" which is in charge of instantiating the corresponding operation. As said before, this method uses reflection intending to fulfill the Open-Closed Principle. The object is instantiated by calling the "newInstance()" method after determining its corresponding class using "Class.forName()".

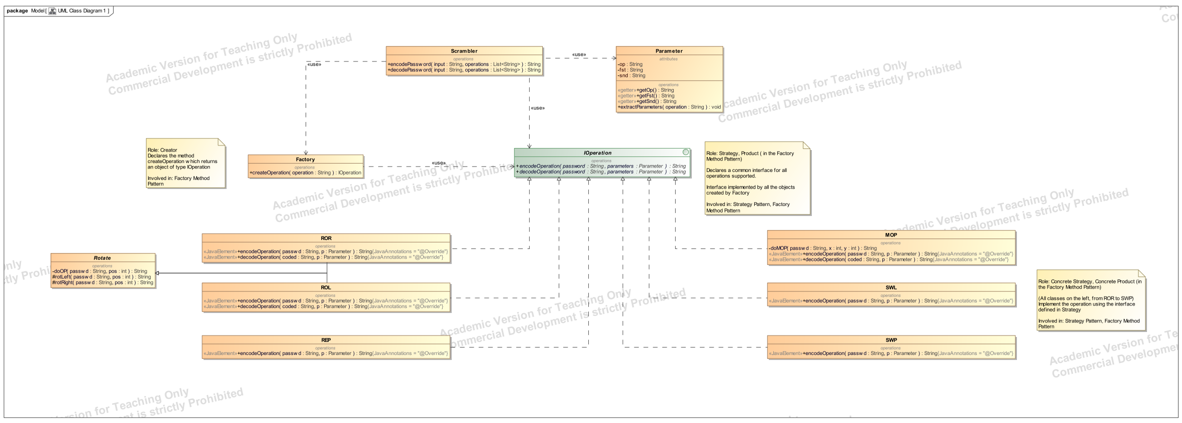
Creator	Concrete Product
	

```java
public class Factory {

    // Instantiate the objects corresponding to the operation passed as parameter
    // to the method. Return the instantiated object (Factory pattern)
    // In case of error, output the resulting exception and return null
    public IOperation createOperation(String operation) {
        IOperation op = null;
        try {
            op = (IOperation) Class.forName("OP2." + operation).newInstance();
        } catch (ClassNotFoundException | IllegalAccessException | InstantiationException e) {
            System.err.println("Exception: " + e);
        }
        return op;
    }
}
```

```java
public class MOP implements IOperation {

    // Internal:
    // Removes the char at position x and inserts it in the new position y
    // returning the resulting string
    // In case a parameter is not valid, output the resulting exception
    private String doMOP(String passwd, int x, int y) {
        try {
            StringBuilder str = new StringBuilder(passwd).deleteCharAt(x);
            char toMove = passwd.charAt(x);
            str.insert(y, toMove);
            return str.toString();
        } catch (IndexOutOfBoundsException e) {
            System.err.println(e);
            return passwd;
        }
    }

    @Override
    public String encodeOperation(String passwd, Parameter p) {
        return this.doMOP(passwd, Integer.parseInt(p.getFst()), Integer.parseInt(p.getSnd()));
    }

    @Override
    public String decodeOperation(String coded, Parameter p) {
        return this.doMOP(coded, Integer.parseInt(p.getSnd()), Integer.parseInt(p.getFst()));
    }
}
```

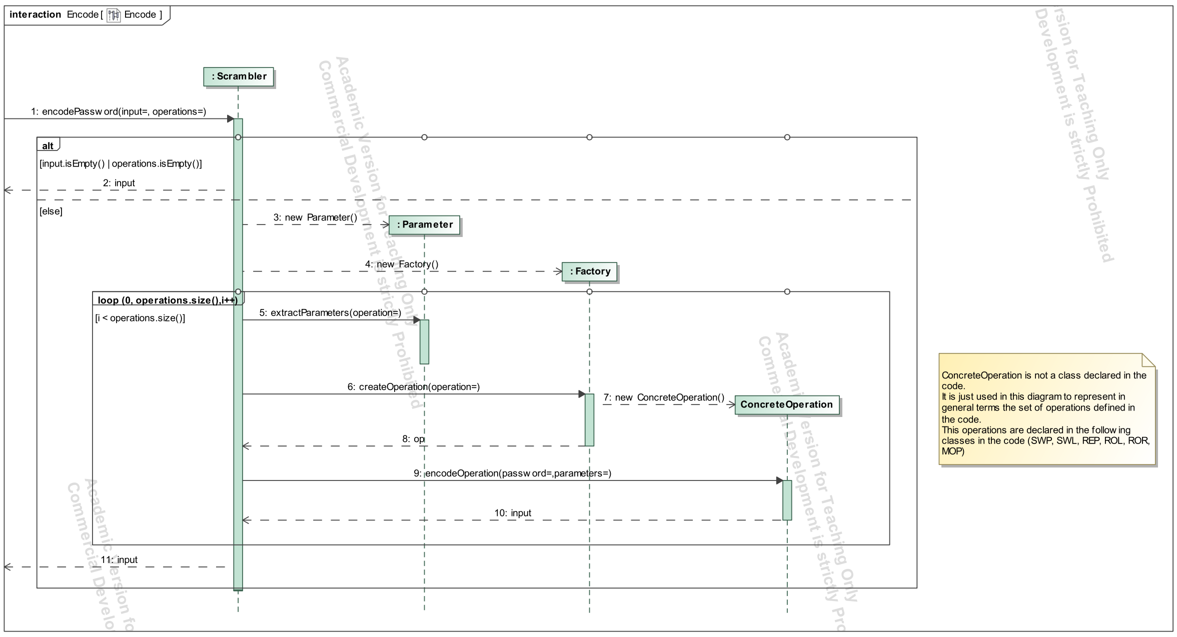
DIAGRAMS:

CLASS DIAGRAM:

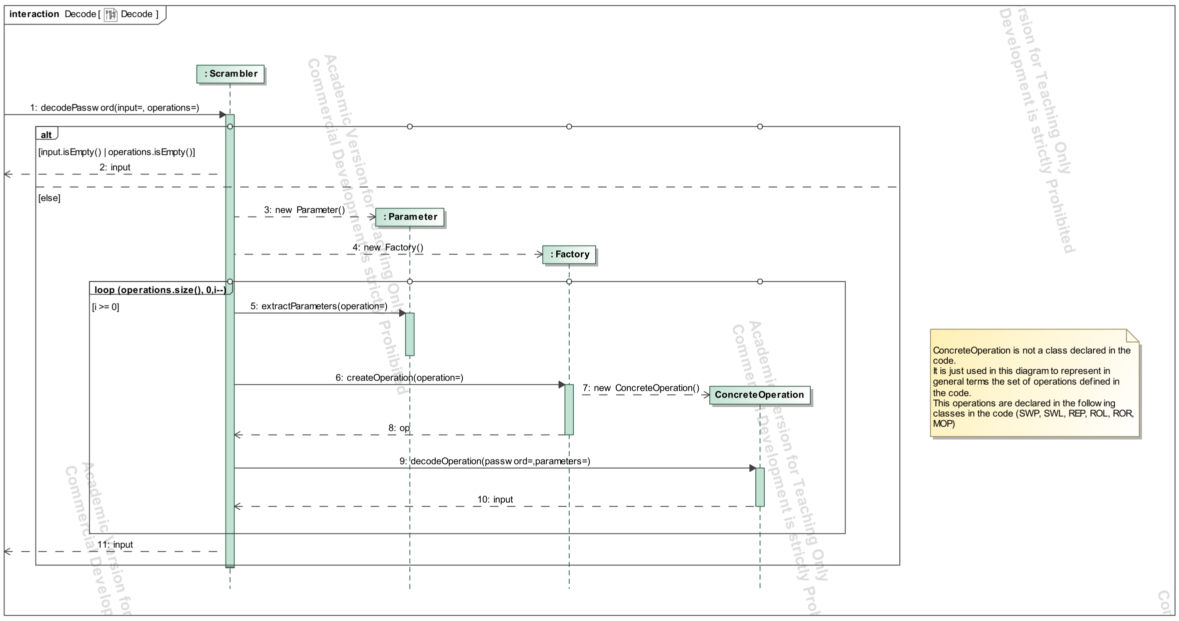

package Model[UML Class Diagram 1]
Scrambler
operations
+encodePassword(input : String, operations : List<String>) : String
+decodePassword(input : String, operations : List<String>) : String
«use»
Parameter
attributes
-op : String
-fst : String
-snd : String
operations
«getter»+getOp() : String
«getter»+getFst() : String
«getter»+getSnd() : String
+extractParameters(operation : String) : void
Role: Creator
Declares the method createOperation which returns an object of type IOperation
Involved in: Factory Method Pattern
Factory
operations
+createOperation(operation : String) : IOperation
IOperation
operations
+encodeOperation(password : String, parameters : Parameter) : String
+decodeOperation(password : String, parameters : Parameter) : String
Role: Strategy, Product (in the Factory Method Pattern)
Declares a common interface for all operations supported.
Interface implemented by all the objects created by Factory
Involved in: Strategy Pattern, Factory Method Pattern
Rotate
-doOP(passwd : String, pos : int) : String
#rotLeft(passwd : String, pos : int) : String
#rotRight(passwd : String, pos : int) : String
ROR
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
«JavaElement»+decodeOperation(coded : String, p : Parameter) : String{JavaAnnotations = "@Override"}
ROL
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
«JavaElement»+decodeOperation(coded : String, p : Parameter) : String{JavaAnnotations = "@Override"}
REP
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
MOP
-doMOP(passwd : String, x : int, y : int) : String
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
«JavaElement»+decodeOperation(coded : String, p : Parameter) : String{JavaAnnotations = "@Override"}
SWL
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
SWP
«JavaElement»+encodeOperation(passwd : String, p : Parameter) : String{JavaAnnotations = "@Override"}
Role: Concrete Strategy, Concrete Product (in the Factory Method Pattern)
(All classes on the left, from ROR to SWP) Implement the operation using the interface defined in Strategy
Involved in: Strategy Pattern, Factory Method Pattern

SEQUENCE DIAGRAMS:

ENCODE SEQUENCE:

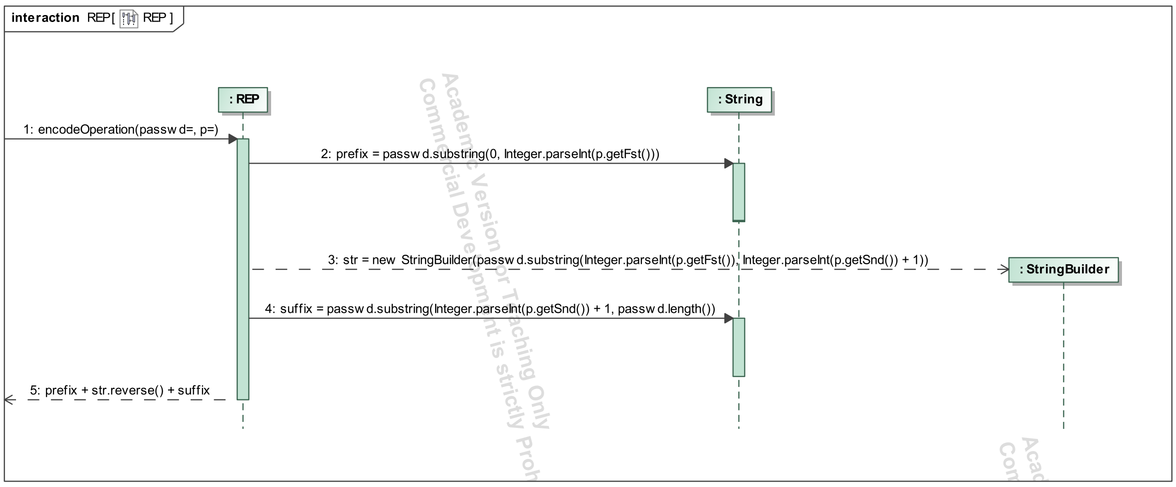

interaction Encode [Encode]
: Scrambler
1: encodePassword(input=, operations=)
alt
[input.isEmpty() | operations.isEmpty()]
2: input
[else]
3: new Parameter()
: Parameter
4: new Factory()
: Factory
loop (0, operations.size(),i++)
[i < operations.size()]
5: extractParameters(operation=)
6: createOperation(operation=)
7: new ConcreteOperation()
ConcreteOperation
8: op
9: encodeOperation(password=,parameters=)
10: input
11: input
ConcreteOperation is not a class declared in the code.
It is just used in this diagram to represent in general terms the set of operations defined in the code.
This operations are declared in the following classes in the code (SWP, SWL, REP, ROL, ROR, MOP)

DECODE SEQUENCE:


interaction Decode [Decode]
: Scrambler
1: decodePassword(input=, operations=)
alt
[input.isEmpty() | operations.isEmpty()]
2: input
[else]
3: new Parameter()
: Parameter
4: new Factory()
: Factory
loop (operations.size(), 0,i--)
[i >= 0]
5: extractParameters(operation=)
6: createOperation(operation=)
7: new ConcreteOperation()
ConcreteOperation
8: op
9: decodeOperation(password=,parameters=)
10: input
11: input
ConcreteOperation is not a class declared in the code.
It is just used in this diagram to represent in general terms the set of operations defined in the code.
This operations are declared in the following classes in the code (SWP, SWL, REP, ROL, ROR, MOP)

REP OPERATION:

Sequence diagram of a particular operation, in this case, REP:


interaction REP[REP]
: REP
: String
1: encodeOperation(password=, p=)
2: prefix = password.substring(0, Integer.parseInt(p.getFst()))
3: str = new StringBuilder(password.substring(Integer.parseInt(p.getFst()), Integer.parseInt(p.getSnd()) + 1))
: StringBuilder
4: suffix = password.substring(Integer.parseInt(p.getSnd()) + 1, password.length())
5: prefix + str.reverse() + suffix

VIDEO LINK:

https://web.microsoftstream.com/video/094b4297-e46e-4d50-a77a-59cb6692a8a5

Author:

Martín Azpilcueta Rabuñal (m.azpilcueta) | Group 6.1