

# Generic Encrypting Machine (GEM)

```
stefano@stefano-Swift-SF314-41:~$ GEM StdMorse
GEM      | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCt
GEM      | Graphic User Inteface        : Shell
Input   |
          -> Hello World
          -> ;
Output  |
          //.....//....//....//----//...//---//...//....//...
GEM     | Good bye!
```

GEM is a software made in May 2020 by **Stefano Savino X81000841** as project for “**Ingegneria del Software**” **9CFU** at Università degli Studi di Catania during year **2019/2020**.

# Index

0 Preface.....	4
1 Functional Requirements.....	6
1.1 Overview.....	6
1.2 Requirements.....	6
1.2.1 Legend.....	6
1.2.2 Correctness State.....	6
1.2.3 Text.....	6
1.2.4 Encrypters.....	7
1.2.5 Validators.....	7
1.2.6 GEM (Generic Encrypting Machine).....	8
1.2.7 GEMc (Generic Encrypting Machine configurer).....	8
1.2.8 Args (Arguments).....	8
1.2.9 GUI (Graphic User Interface).....	8
1.2.10 Language.....	9
2 Design.....	10
2.1 Overiew.....	10
2.1.1 Packages.....	10
2.1.2 Conventions.....	10
2.2 Correctness State.....	10
2.2.1 StdCorrect.....	10
2.3 Utilities.....	11
2.3.1 Words.....	11
2.3.2 Lines.....	11
2.3.3 Char.....	11
2.3.4 DirHandler.....	12
2.3.5 FileLoader and FileSaver.....	12
2.3.6 IO.....	12
2.4 Encrypters.....	13
2.4.1 Encrypter.....	13
2.4.2 StdSyntax.....	13
2.4.3 StdDivider.....	13
2.4.4 StdFixed.....	13
2.4.5 StdVariable.....	13
2.4.6 StdCesare.....	13
2.4.7 StdMorse.....	13
2.4.8 EncrypterHandler.....	14
2.5 Validators.....	14
2.5.1 Validator.....	14
2.5.2 StdValidator.....	14
2.5.3 StdAlphabetic.....	14
2.5.4 StdAlphaNumeric.....	14
2.5.5 StdAlphaNumericPoint.....	14
2.6 Args.....	14
2.6.1 Args.....	14
2.6.2 StdArgs.....	15
2.6.3 GEMArgs.....	15

2.6.4 GEMcArgs.....	15
2.7 GEM.....	15
2.7.1 StdGEM.....	15
2.7.2 GEM.....	15
2.7.3 GEMc.....	15
2.8 GUI.....	15
2.8.1 GUI.....	15
2.8.2 Shell.....	16
2.9 Languages.....	16
2.9.1 Language.....	16
2.9.2 Concrete Implementations.....	16
2.9.3 Languages.....	16
2.10 Tests.....	16
3 UML.....	17
3.1 UML : Encrypter and Validator.....	17
3.2 UML : Languages.....	19
3.3 UML : Lines and Words.....	20
3.4 UML : Main Diagram.....	21
3.5 Sequence UML : Encrypting/Translating execution.....	22
3.6 Sequence UML : Configuration Modality execution.....	23
4 Design Patterns.....	26
4.1 Overview.....	26
4.2 Singleton.....	26
4.3 State.....	26
4.4 Factory Method.....	27
4.5 Adapter.....	27
5 Code.....	28
5.1 Code : EcripterHandler.....	28
5.2 Code : Languages.....	29
5.3 Code : Encrypter Hierarchy.....	31
5.3.1 Code : StdCorrect.....	31
5.3.2 Code : Encrypter.....	32
5.3.3 Code : StdSyntax.....	33
5.3.4 Code : StdDivider.....	33
5.3.5 Code : StdFixed.....	34
5.3.6 Code : StdVariable.....	35
6 Usage.....	37
6.1 GEMc.....	37
6.2 GEM.....	37
6.3 Example 1.....	38
6.4 Example 2.....	39
6.5 Example 3.....	40
6.6 Example 4.....	40
6.7 Example 5.....	42
7 Summary.....	45

# 0 Preface

The choice of making a **Generic Encrypting Machine** is not casual.

I have been dealing with encrypting since many years now :

- As a twelve years experienced Scout I have had to work with many simple communication systems such as **Morse Language** (in this case we should talk about translation rather than encryption), **Caesar's Chipper** and many others
- My Scientific Lyceum Final Exam's Thesis has been about a C based emulator of Nazi's encrypting machine **Enigma** I realized after a light 4 Months C language course I did in my school

As a result I have a little knowledge in the field of encryption.

I do not aim to become a programmer or a project manager. I would rather be a researcher in the field of algorithms and computing but also in this case knowledge about Software Engineering has its role.

**Encryption/Translation** may be seen in two ways :

- **Algorithms** that produce encrypted text.
  - There are different algorithms for different encryption/translation systems but many of them have pieces in common. This is the key point **GEM** is based on. Language **Java** is appropriated for this.
- **Functions** of the form  $f : A \rightarrow B$  where  $f$  is the function defined in  $A$  that is the set of all possible strings and with values in  $B$  that is the set of all the possible encrypted strings.
  - In this way, encryption/translation system such as Morse could be easily defined as

$g : String \rightarrow String$

$g < x_1, x_2, \dots, x_{n-1}, x_n > = < f_{x_1}, f_{x_2}, \dots, f_{x_{n-1}}, f_{x_n} >$

$f : Char \rightarrow String$

$$\begin{cases} f a = \text{.-} \\ f n = \text{-.} \\ [...] \end{cases}$$

This means that writing it in a pure functional language such as **Haskell** would have as a result :

```
g :: String -> String
g x = map f x

f :: Char -> String
f 'a' = ".-"
f 'n' = "-."
[...]
f x = "Unknow_Char"
```

In this way code results more synthetic without all those **switch**, **if**, **else**, ecc... The correctness of the definition of the encrypting/translating function results would be easier to prove avoiding a certain number of bugs.

I had previously written an Haskell Functional version of Morse Translator and Caesar's Cipher but in this paper I am going to talk about the Java based software GEM.

What I would like the reader to appreciate is its **modular** structure that makes it easy to **Maintain** and **expand**.

This paper is organized as Software Engineering teacher *Emiliano Tramontana* suggested but in order to better understand the whole thing I would suggest an approach similar to the one adopted by *Andrew Stuart Tanenbaum* in *Modern Operating Systems* :

- **Section 0** : Preface
- **Section 1** : Functional Requirements
- **Section 6** : Usage
- **Section 2** : Design
- **Section 4** : Design Patterns
- **Section 3** : UML
- **Section 5** : Code
- **Section 7** : Summary

Thanks for reading

Stefano Savino

# 1 Functional Requirements

## 1.1 Overview

The aim of **GEM** is to realize a software able to join many encrypting/translating systems. It has to be easy expanding, easy maintaining and most of all easy understanding.

The only concrete systems implemented in this first version of **GEM** are :

- Caesar Cipher
- Morse Language
- Leon Battista Alberti's Disk (Rotor) *not implement yet*
- Enigma Machine *not implement yet*

## 1.2 Requirements

### 1.2.1 Legend

- Classes
- Attributes
- Operations

### 1.2.2 Correctness State

- **Correctness State** informs if errors have happened
- **Correctness State** may only be switched from True to False
- **Correctness State** may be checked

### 1.2.3 Text

- The aim of the program is to encrypt Text
- Text is made of **Lines**
  - **Lines** are made of ordered **Words** separated by line feed
    - **Words** are made of ordered **Strings** separated by spaces
      - **Strings** are made of chars
      - Chars may be :
        - Alphabetic
        - Upper Case

- Lower Case
- Numeric (Digit)
- Symbol
- A combination of previous ones

#### 1.2.4 Encrypters

- The program should emulate different **Encrypters**
- **Encrypters** have **Correctness State**
- Different **Encrypters** refer to different encrypting systems
- **Encrypters** encrypt Lines
- **Encrypters** may have a **syntax Validator**
- **Encrypters** may partition encrypting process :
  - **Encrypters** may encrypt a single Line into a single Line (Lines to Lines)
    - **Encrypters** may encrypt a single Word into a single Word (Words to Words)
      - **Encrypters** may encrypt a char into a single char (Char to Char)
      - **Encrypters** may encrypt a single char into a variable length string (Char to String)
- **Encrypters** may need to be configured
  - **Encrypters** may be configured and configurations saved
  - **Encrypters** may load configurations
- Implementations of **Ecripters** are identified ; the identifier is called **Modality**

#### 1.2.5 Validators

- **Validators** check if lines are grammatically valid
- **Validators** may partition validation process :
  - **Validators** may validate one line per time
    - **Validators** may validate one word per time
      - **Validators** may validate one char per time
        - A **Validator** could check if chars are Alphabetic
        - A **Validator** could check if chars are Alphabetic or Numeric
        - A **Validator** could check if chars are Alphabetic, Numeric or Symbols

## 1.2.6 GEM (Generic Encrypting Machine)

- Software may be run in encrypting modality : GEM
- GEM loads execution arguments
- GEM checks if execution arguments are correct
- GEM gets information from execution arguments
- GEM loads an Encrypter (maybe configured)
- GEM gets the input for the Encrypter
- GEM shows the output of the Encrypter
- GEM may show Errors, Warnings and Messages

## 1.2.7 GEMc (Generic Encrypting Machine configurer)

- Software may be run in configuration modality : GEMc
- GEMc loads execution arguments
- GEMc checks if execution arguments are correct
- GEMc gets information from execution arguments
- GEMc sets the Encrypter configuration
- GEMc may show Errors, Warnings and Messages

## 1.2.8 Args (Arguments)

- Arguments have Correctness State
- Arguments analyzes and holds the hidden information behind execution arguments
- To different execution arguments correspond different Arguments, to different Arguments correspond different executions of the software
- Arguments may refer to Language settings
- Arguments may refer to an Encrypter
- Arguments may refer to an Encrypter configuration
- Arguments may refer to IO Modality for input and output
- Arguments may refer to an input/output file
- Arguments may show Errors, Warnings and Messages

## 1.2.9 GUI (Graphic User Interface)

- GUI actually shows user Errors, Warnings and Messages

- **GUI** actually **shows** user output
- **GUI** actually **gets input** from user
- **GUI** actually **shows** user **initialization, closure and rebooting** of the program

### 1.2.10 Language

- The software may be run in different **Languages**
- A **Language** **obtains** all the strings referring to an **Errors**, **Warnings** and **Messages**
- **Language** may be set at the beginning or during **execution**
- Default **Language** is English

## 2 Design

### 2.1 Overview

#### 2.1.1 Packages

Classes included in the project are organized in the following way

- Package **GEM**
  - Package **args (Arguments)** that contains all the classes aimed to deal with **Arguments (UML color blue)**
  - Package **corr (Correctness State)** that contains the basic Correctness State implementation (**UML color white**)
  - Package **encr (Encrypters)** that contains all the **Encrypters (UML color red)**
  - Package **gem (GEM)** that contains the **GEM (UML color white)**
  - Package **gui (Graphic User Interface)** that contains all the classes handling the **GUI (UML color dark green)**
  - Package **lan (Languages)** that contains all the **Language utilities (UML color purple)**
  - Package **tests (Tests)** that contains all the tests classes
  - Package **utils (Utilities)** that contains all the useful classes (**UML color yellow**)
  - Package **val (Validators)** that contains all the **Validators (UML color orange)**

#### 2.1.2 Conventions

Classes in this software have explicit names. Some of them have prefix **Std**. This is given by the fact that the software is thought to be easily expanded. A new implementation of an **Encrypter** just be an expansion of an already defined class **Std** for example abstract class **StdFixed** (see StdFixed).

## 2.2 Correctness State

### 2.2.1 StdCorrect

As seen in requirements some of classes localized have a **Correctness State**. **Correctness State** could be seen only as an attribute but the best choice has turned to be the definition of abstract class **StdCorrect** that include :

- attribute **correct** that represents the Correctness State and is not visible outside of **StdCorrect**
- method **setError()** that sets correct to False and has a protected visibility
- method **isCorrect()** that returns correct value

Many of the classes in GEM project are extensions of **StdCorrect**.

## 2.3 Utilities

### 2.3.1 Words

As seen in requirements Text can be seen as **Lines** that are made of **Words** that are made of **Strings**.

Class **Words** is just an alias or in better words an **Adapter** of **ArrayList<String>**. If **String** *a* is at index *j* and **String** *b* is at index *j+1* this means that in the text *a* and *b* are separated by a space.

Class **Words** offers :

- method **size** (taken from **ArrayList<String>**)
- method **get** (taken from **ArrayList<String>**)
- method **add** (taken from **ArrayList<String>**)
- method **iterator** (taken from **ArrayList<String>**)
- method **toString** that returns a semantically corresponding **String**
- method **toWords** that returns an instance of **Words** semantically corresponding to a given **String**

### 2.3.2 Lines

As seen in requirements Text can be seen as **Lines** that are made of **Words** that are made of **Strings**.

Class **Lines** is just an alias or in better words an **Adapter** of **ArrayList<Words>**. If **Words** *a* is at index *j* and **Words** *b* is at index *j+1* this means that in the text *a* and *b* are separated by a line feed.

Class **Lines** offers :

- method **size** (taken from **ArrayList<Words>**)
- method **get** (taken from **ArrayList<Words>**)
- method **add** (taken from **ArrayList<Words>**)
- method **iterator** (taken from **ArrayList<Words>**)
- method **toString** that returns the corresponding **String**
- method **toWords** that returns an instance of **Words** corresponding to a given **String**

### 2.3.3 Char

As seen in requirements some considerations may need to be made dealing with chars.

Many methods are thus needed. They are all static and grouped in class **Char**. Because it would be just a waste of paper they are not listed here.

### 2.3.4 DirHandler

As seen in requirements, **Encrypters** may be configured (and then configurations saved) and the software may also be installed in the machine. This means the software may need to deal with directories inside the OS.

The adopted scheme is the following :

- **home** , User's home directory
  - **GEM** directory that holds everything concerning the software
    - **GEM** directory that holds the executable files
      - *Packages*
    - **GEMData** directory that holds saved data
      - **Encrypter** directory that holds data referring to encrypters' configurations
        - **StdCesare** directory that holds configuration files referring to the implementation of the Encrypter Caesar Cipher
        - [...]

Class **DirHandler** offers :

- method `getMain` that returns **home**'s path
- method `getData` that returns **GEMData**'s path
- method `getEncrypter` that returns **Encrypter**'s paths
- method `getEncrypter` that returns an implementation of **Encrypter**'s directory path
- method `getConfFile` that returns a configuration file paths

### 2.3.5 FileLoader and FileSaver

As seen in requirements input and output may be taken/setted into a file and configurations may be saved. **FileLoader** and **FileSaver** are just adapters that make possible to read/write **Lines** into files and also check file existence.

### 2.3.6 IO

As seen in requirements input and output may be taken/setted into a file. This leads to the creation of class **IO** that contains

- enum **Mod** :
  - **IO**
  - **file**
- useful methods to deal with IO (`isIOModIO`, `isIOModFile`, ecc...)

## 2.4 Encrypters

### 2.4.1 Encrypter

An extension of **StdCorrect** and also one of the most important entities in the software is **Encrypter**. Different **Encrypters** refer to different encrypting systems. At the same time different **Encrypters** may have similar approaches to encryption. This is why **Encrypter** is just an abstract class. **Encrypter** includes :

- abstract method `encrypt` that returns **Lines** encrypted
- static method `setConfig` that has to be implemented by subclasses that need configurations
- static method `getConfig` that has to be implemented by subclasses that need configurations

### 2.4.2 StdSyntax

As seen in requirements some **Encrypters** may have some syntax restrictions in their input files. This is why Abstract Class **StdSyntax** extends **Encrypter** adding a **Validator**.

### 2.4.3 StdDivider

As seen in requirements some **StdSyntax** may have the property that the encrypting process may be partitioned (this could be used in future to speedup the software). Abstract Class **StdDivider** extends **StdSyntax** and **does the partitioning down to the String level**.

### 2.4.4 StdFixed

As seen in requirements some **StdDivider** may **encrypt one char into another char**. Abstract Class **StdFixed** extends **StdDivider** and implements this property.

### 2.4.5 StdVariable

As seen in requirements some **StdDivider** may **encrypt one char into a variable/fixed length string**. Abstract Class **StdVariable** extends **StdDivider** and implements this property.

### 2.4.6 StdCesare

**StdCesare** is an extention of **StdFixed** that implements the Caesar Cipher.

- It implements **encryption char to char**
- It instantiates an **StdAlphaNumericPoint** validator
- It needs configuration so it implements `setConfig` and `getConfig`

### 2.4.7 StdMorse

**StdMorse** is an extension of **StdVariable** that implements the Morse translation.

- It implements **encryption char to String**

- It instantiates an **StdAlphaNumericPoint** validator
- It does not need configuration so it does not implement `setConfig` and `getConfig`

## 2.4.8 EncrypterHandler

Users while running the software may desire to use an implementation of **Encrypter**. **EncrypterHandler** implements design pattern FactoryMethod for class **Encrypter**. More about later in this paper.

## 2.5 Validators

### 2.5.1 Validator

As seen in requirements **Encrypters** may have a **Validator**. **Validator** is very wide concept. In the versions made up to this version of the software it can be seen just as a collection of methods that could even be static. Despite of this it may occur that in further versions of the software a **Validator** may need its own data and **state**. Because of this **Validator** has been implemented as an interface that offers only :

- abstract method `isValid` that checks if a **Lines** is valid

### 2.5.2 StdValidator

As seen in requirements also **Validators** may **partition validation process** (this could be used further to speed up the process) . This feature is implemented in abstract class **StdValidator**.

### 2.5.3 StdAlphabetic

**StdAlphabetic** extends **StdValidator** for grammars that **accept only alphabetic chars**.

### 2.5.4 StdAlphaNumeric

**StdAlphaNumeric** extends **StdValidator** for grammars that **accept only alphabetic and numeric chars**.

### 2.5.5 StdAlphaNumericPoint

**StdAlphaNumericPoint** extends **StdValidator** for grammars that **accept only alphabetic, numeric and symbolic chars**.

## 2.6 Args

### 2.6.1 Args

As seen in requirements executions of the software are different because of some parameters : **Arguments**.

These ones are understood and set by class **Args** that is an extension of **StdCorrect** (it has **Correctness State**).

Class **Args** offers utilities to deal with arguments that may be seen by subclasses :

- String `current` that contains the current **execution argument analyzed**

- method `hasNext`
- method `goBack`

## 2.6.2 StdArgs

Abstract Class **StdArgs** is an extension of **Args** that contains useful methods and attributes for both further extensions **GEMArgs** and **GEMcArgs**. These ones refer to **Language**, configuration and **Encrypter**.

## 2.6.3 GEMArgs

Class **GEMArgs** is an extension of **StdArgs** that implements **Args** for the execution of the software in encrypting/translating modality.

## 2.6.4 GEMcArgs

Class **GEMcArgs** is an extension of **StdArgs** that implements **Args** for the execution of the software in configuration modality.

## 2.7 GEM

### 2.7.1 StdGEM

As seen in requirements the software could be run in two different modalities : encrypt/translate and configuration. Both modalities have many pieces in common that are gathered in abstract class **StdGem**.

### 2.7.2 GEM

**GEM** extends **StdGEM** for encrypt/translate modality.

### 2.7.3 GEMc

**GEMc** extends **StdGEM** for configuration modality.

## 2.8 GUI

### 2.8.1 GUI

As seen in requirements the software has to be ready for further extensions. One of main features is that it has to be predisposed for **Graphic User Interface**. The interface has been seen in the following way :

- An error channel
- A warning channel
- A message channel
- An input/output channel

to **deal with**.

The interface should also be able to :

- Initialize itself
- Reboot
- Close

Interface **GUI** contains all the methods required. Further subclasses could act as adapters with **GUIs**.

## 2.8.2 Shell

In this first version of the software the only concrete implementation of **GUI** is **Shell** that implements **GUI** with shell utilities.

## 2.9 Languages

### 2.9.1 Language

As seen in requirements the software has to be ready for further extensions. This also regards languages.

Interface **Language** contains all the methods that have to be overwritten by implementations to return correct **error**, **message** or **warning**. To make this easier three enums have been created : **Error**, **Warning**, **Msg**.

### 2.9.2 Concrete Implementations

In this first version of the software the only concrete implementations of **Language** are **English**, **Italiano** and **Français**.

### 2.9.3 Languages

The **current language** the software is running could be seen as a state. In order to manage this class **Languages** has been realized implementing design patterns Singleton and State. More about in Design Patterns' section.

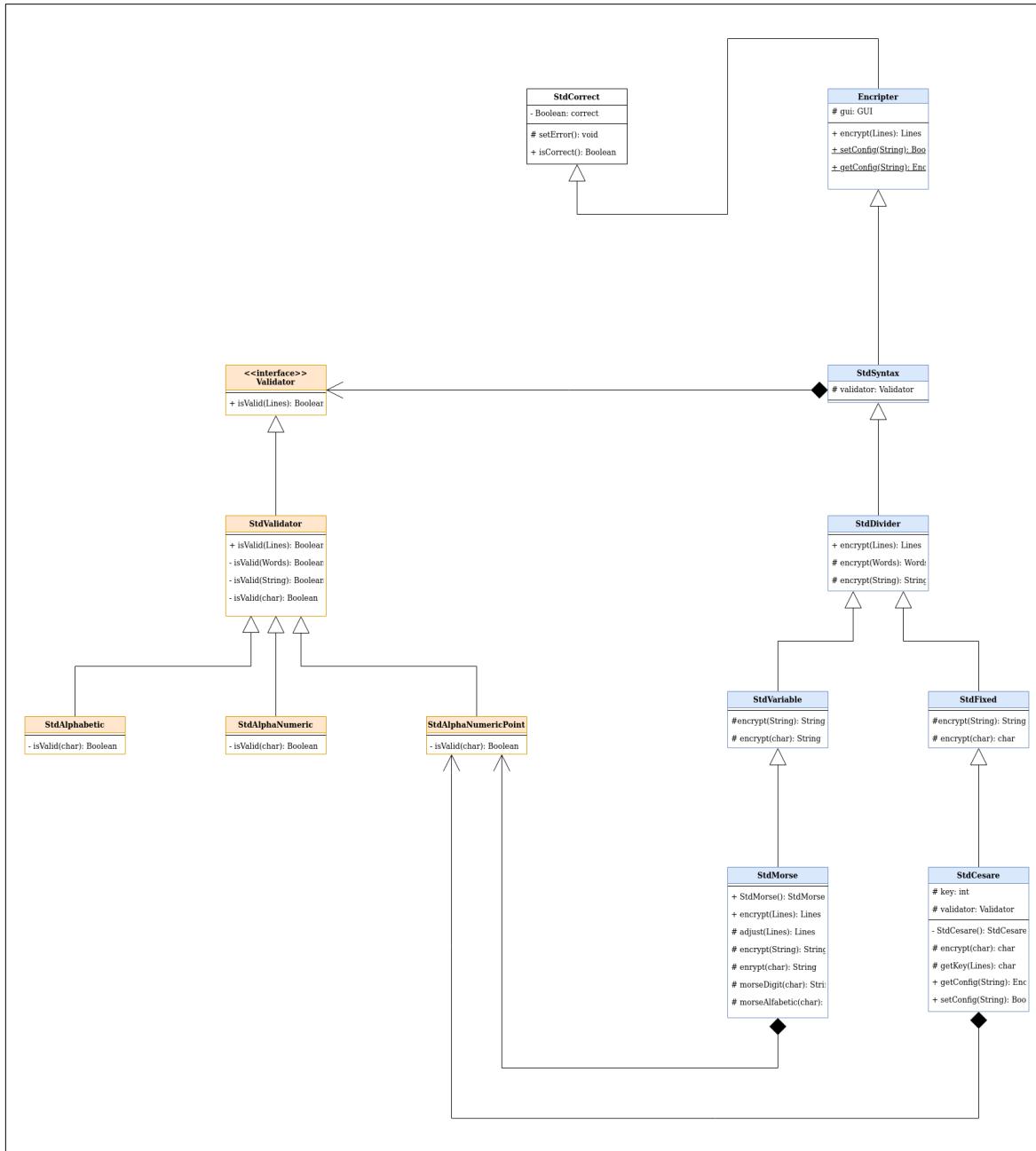
## 2.10 Tests

Many **Test** classes have been built during software development. These ones are all gathered in package tests.

### 3 UML

The full UML Diagram of the software is too big to fit in this paper (it would need an A2 paper to have an acceptable quality) for this reason the full version is attached as [GEMx.y.html](#). Only some extracts are shown here.

#### 3.1 UML : Encrypter and Validator



In this first paragraph we are going to analyze the structure of **Encrypters** (blue) and **Validators** (orange) :

- As stated before **Encrypter** has the **Correctness State** so it is an extension of class **StdCorrect**. It defines abstract method `encrypt(Lines):Lines` and static methods `getConfig` and `setConfig` that have to be overwritten by those extensions that are configurable.
- Abstract Class **StdSyntax** extends **Encrypter** adding the syntax **Validator**.
- Abstract Class **StdDivider** extends **StdSyntax** partitioning the encrypting/translating process down to Strings leaving abstract method `encrypt(String):String`
- Abstract Class **StdFixed** extends **StdDivider** for those encrypting/translating systems that `encrypt/translate a char into a char` leaving abstract method `encrypt(char):char`
- Abstract Class **StdVariable** extends **StdDivider** for those encrypting/translating systems that `encrypt/translate a char into a variable length String` leaving abstract method `encrypt(char):String`
- **Validator** is an interface that gives method `isValid(Lines):Boolean` for **Lines**
- Abstract Class **StdValidator** implements **Validator** partitioning work leaving abstract method `isValid(char):Boolean`

Classes :

- **StdAlphabetic**
- **StdAlphaNumeric**
- **StdAlphaNumericDigit**

are concrete implementations of **Validator** because they implement `isValid(char):Boolean`.

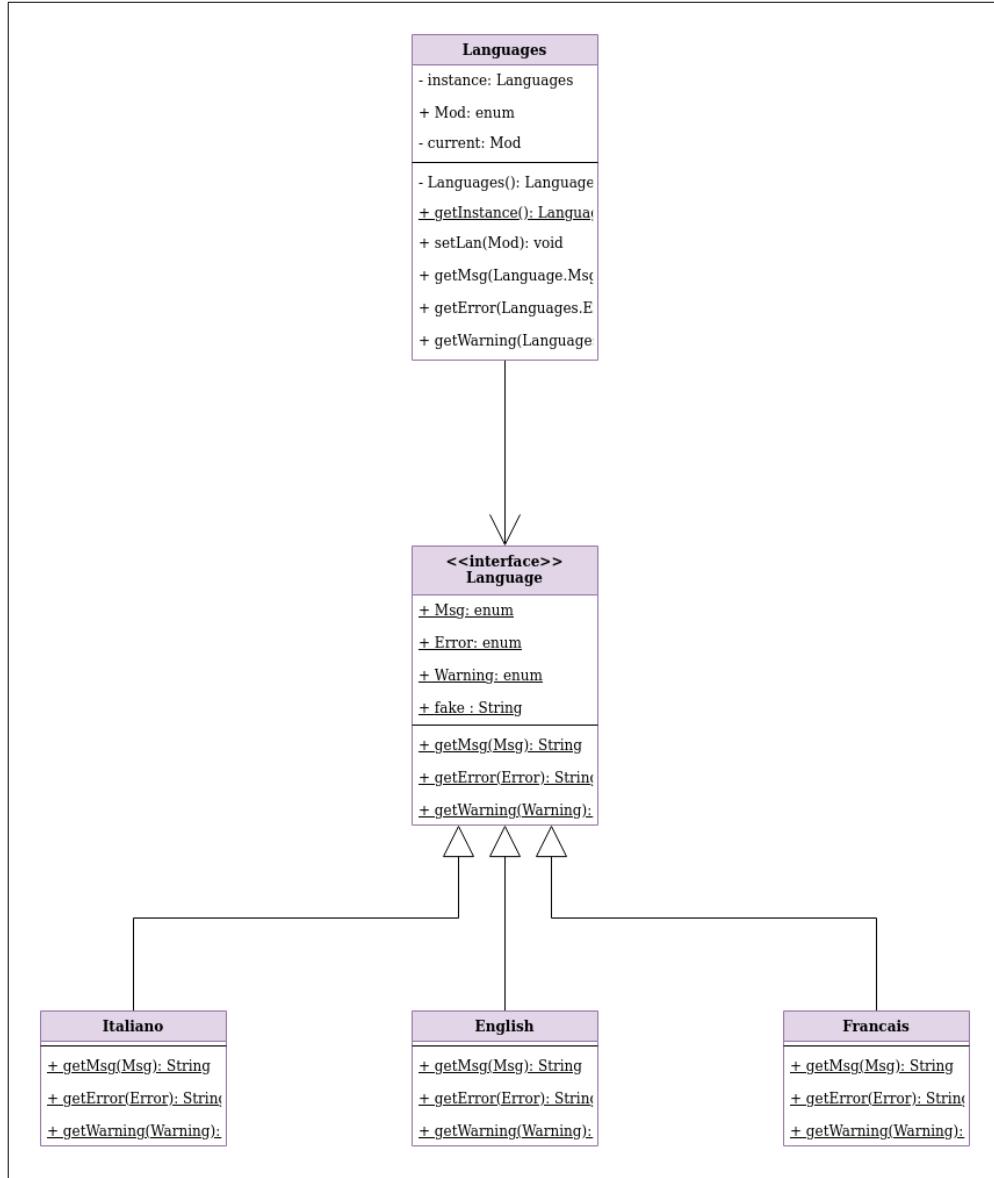
Classes :

- **StdMorse**
- **StdCesare**

are concrete implementations of **Encrypter** because :

- **StdMorse** implements **StdVariable** `encrypt(char):String`
- **StdCesare** implements **StdFixed** `encrypt(char):char`

## 3.2 UML : Languages



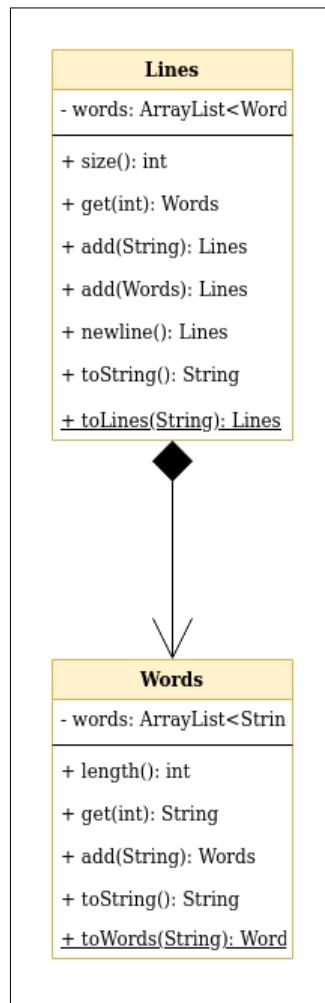
In this paragraph we are going to analyze the part of the program that deals with languages (purple). As said before it is all contained in package lan.

- Interface **Language** defines all the methods required by a language that supports the system :
  - `getMsg`
  - `getError`
  - `getWarning`

in order to make the code easy understanding errors, warnings and messages are not identified by a number code but by enum **Error**, enum **Warning** and enum **Msg**.

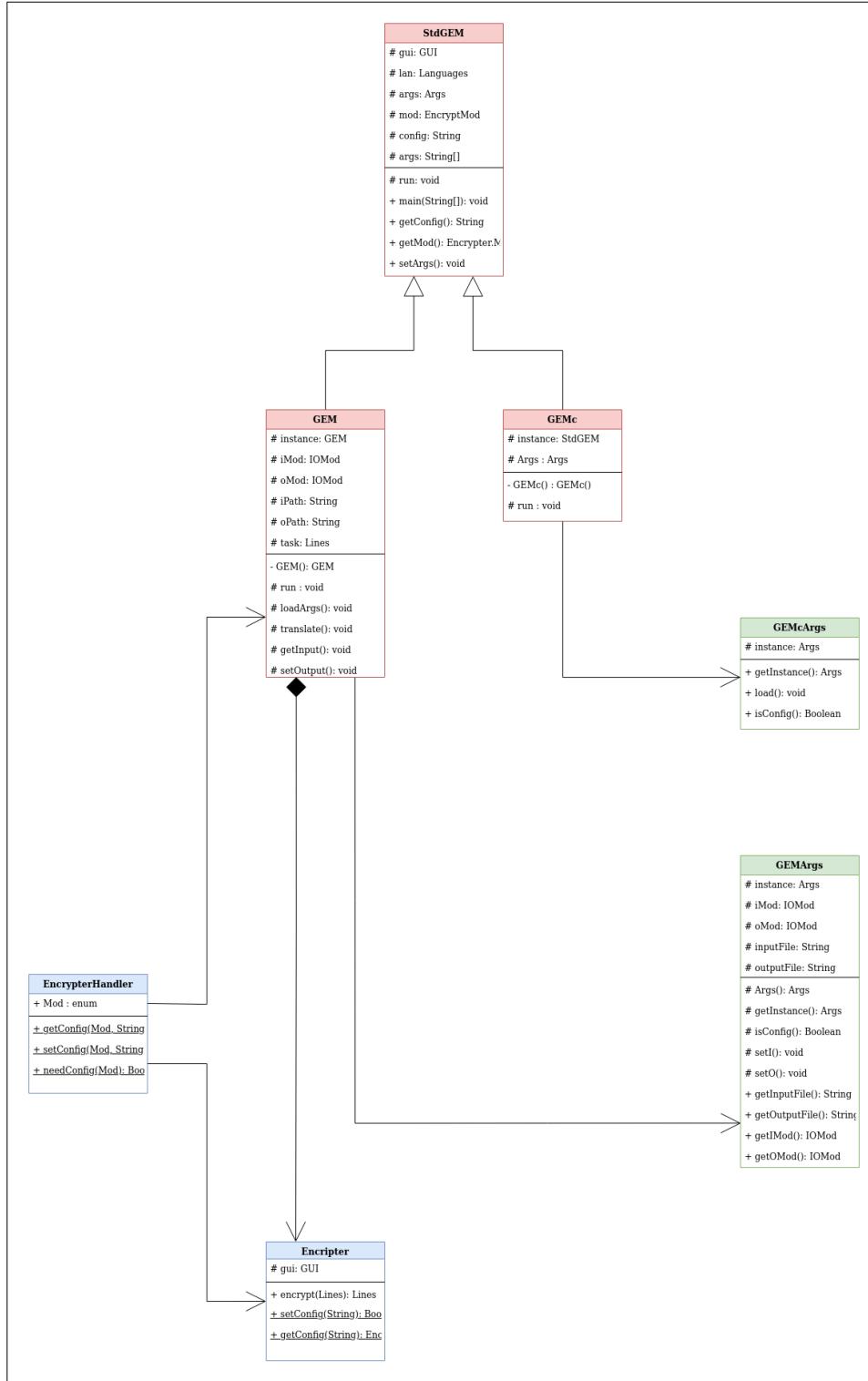
- Italiano, English and Français are concrete implementations of Language.
- As demonstrated in the section regarding Design Patterns it has been needed to create class Languages that implements Design Patterns Singleton and State. Class Languages contains :
  - Languages instance that is the only instance of the class and method getInstance to access to it
  - enum Mod that contains all the possible choices : en, it, fr.
  - Mod current that is the contains the state (the language the software is currently running) that could be changed anywhere with method setLan

### 3.3 UML : Lines and Words



As seen in previous sections Text could be seen as Lines made of Words made of Strings. This way to see Text is directly taken from functional language Haskell. In language java this could be seen as `ArrayList<ArrayList<String>>` but that would not be easy understanding. Two adapters have been defined : Words and Lines. They are bound by composition : Lines are made of Words.

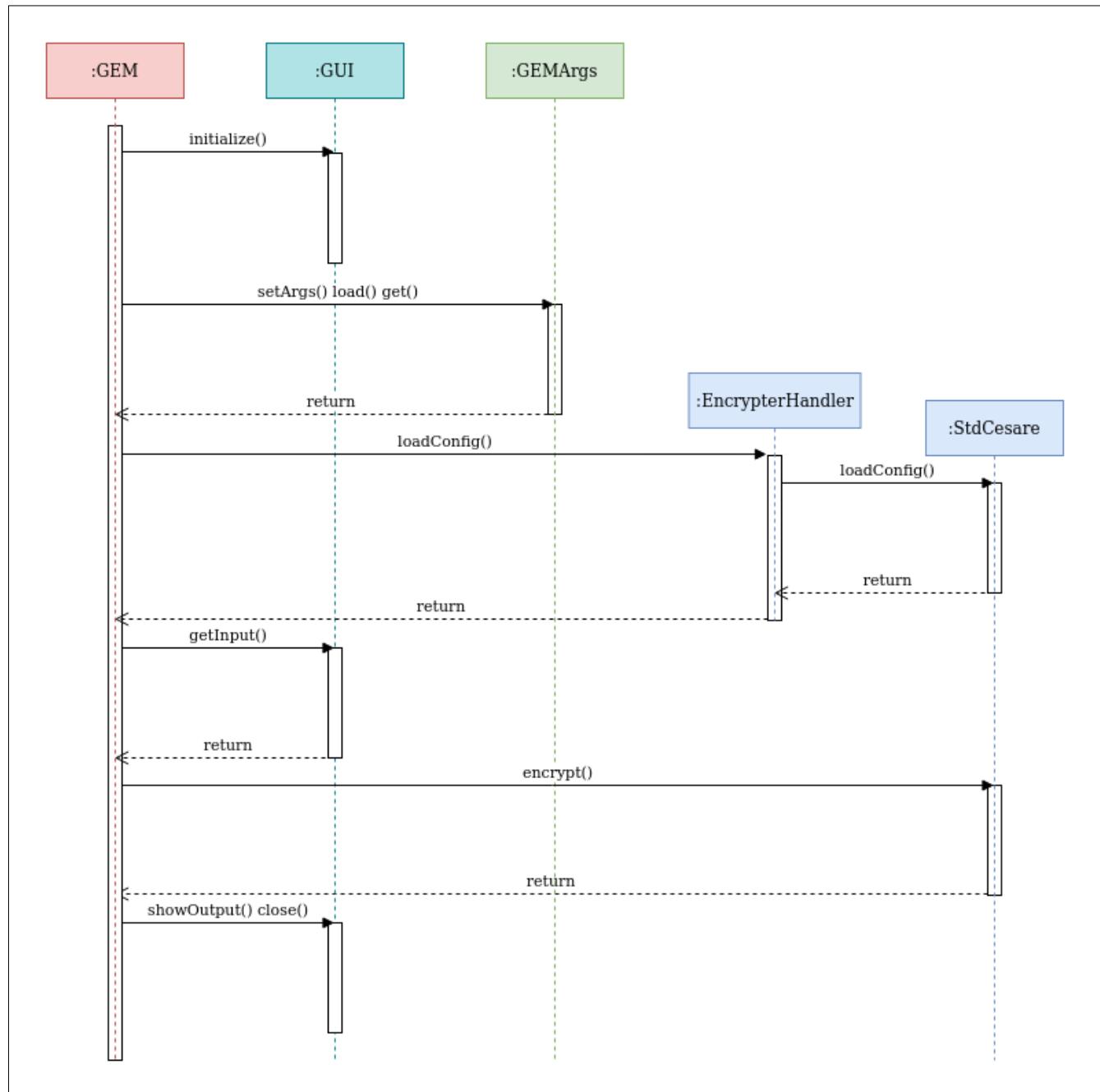
### 3.4 UML : Main Diagram



The last extract shown in this paper shows the main section of the software. As shown, classes **GEM** and **GEMc** (red) are in relationship with classes **GEMArgs** and **GEMcArgs** (green). In fact they give to the second ones the

execution arguments to analyze. After that, if arguments are correct, they query `EncrypterHandler`. `GEM` receives from `EncrypterHandler` the right `Encrypter` (configured if necessary). `GEMc` calls the right `Encrypter`'s configuration method thanks to `EncrypterHandler`. For better understanding this scheme let's go to next paragraphs.

### 3.5 Sequence UML : Encrypting/Translating execution

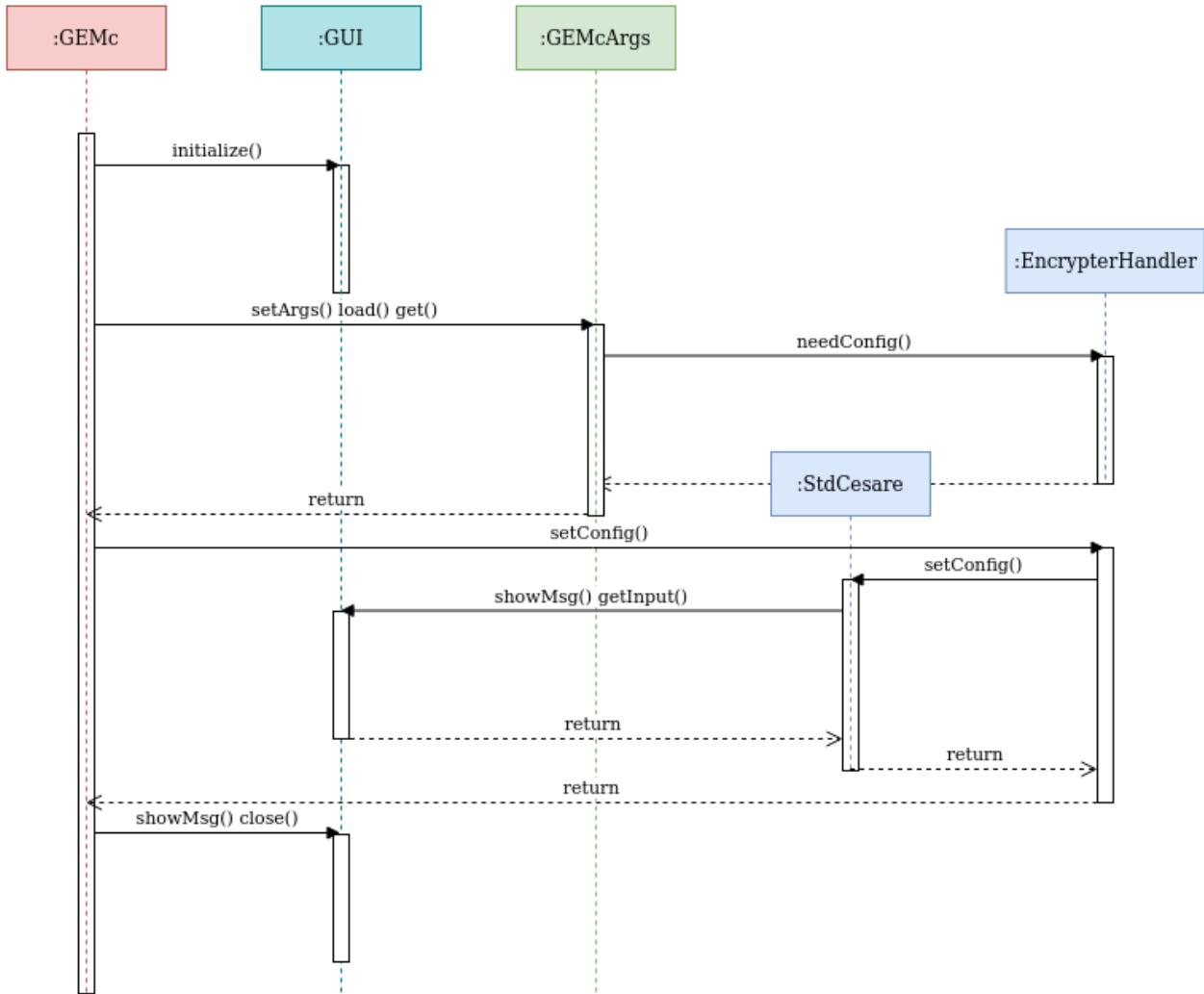


This scheme briefly explains the execution of the software in **Encrypting/Translating modality**. In this example the software is executed in the following way : “**GEM StdCesare myConfig**”. (see Usage)

Software then runs :

1. The program begins from class **GEM** that asks class **GUI** to **initialize**
2. After **GUI**'s **initialization** **GEM** gives **execution arguments** to **GEMArgs** and asks **GEMArgs** to **load** (undestand) them
3. **GEM** checks **GEMArgs Correctness State** that is positive because the given arguments are correct so it asks **GEMArgs** to **obtain all parameters needed**.
4. **GEM** then obtains the **StdCesare** with **myConfig configuration** thanks to **EncrypterHandler**
5. As **input modality** is IO **GEM** asks **GUI** for input
6. Once **input obtained** **GEM** asks **StdMorse** to **encrypt it**
7. Once **obtained the encrypted lines** from **StdCesare**, **GEM** checks if issues have occurred **checking StdCesare Correctness State** that we suppose positive
8. **GEM** can now ask **GUI** to **show the output** an then to **close**
9. The execution is now terminated

### 3.6 Sequence UML : Configuration Modality execution



This scheme briefly explains the execution of the software in configuration modality. In this example the software is executed in the following way : “**GEMc StdCesare myConfig**”. (see Usage)

Software then runs :

1. The program begins from class **GEMc** that asks class **GUI** to **initialize**
2. After **GUI**'s initialization **GEMc** gives **execution arguments** to **GEMcArgs** and asks **GEMcArgs** to **load** (understand) them
3. **GEMcArgs** reads execution parameters and sees that user is trying to configure **StdCesare** so it asks **EncrypterHandler** if **StdCesare** is configurable (needs a configuration)
4. **GEMc** checks **GEMcArgs Correctness State** that is positive because the given **arguments** are correct so it asks **GEMcArgs** to obtain all parameters needed.
5. **GEMc** calls **EncrypterHandler.setConfig** and **EncrypterHandler** calls **StdCesare.setConfig**

6. StdCesare creates and saves the new configuration. In order to do this it may ask GUI to show messages, errors, warnings or to obtain input and does all the stuff making use of all utilities in package utils
7. GEMc asks GUI to show the outcome of the task and then to close
8. The execution is now terminated

# 4 Design Patterns

## 4.1 Overview

During development some design patterns have been adopted in order to make the code easier to maintain.

These ones are :

- **Singleton**
- **State**
- **Factory Method**
- **Adapter**

## 4.2 Singleton

Design Pattern Singleton has been adopted several times in this software :

- In class **GUI** the Graphic User Interface in fact has to be **unique**. Many instances of it could cause strange issues to user's experience.
- In class **Languages** because the language settings (**current language**) should be **unique** for the whole software.
- In class **GEM** and **GEMc** because they have states that have to be **unique**. In these two classes method `getInstance()` has not been implemented because the only instance should only be accessed **running the class itself**.
- In class **GEMArgs** and **GEMcArgs** because they contain the **execution arguments** and the corresponding **Correctness State** that have to be **unique** during whole execution.

## 4.3 State

Design Pattern State has been adopted once in this software :

- In class **Languages**. As seen before there are many implementations of **Language** (**English**, **Italiano**, **Français**) but only one of them has to **be used according to the current language** that may be changed by users. Class **Languages** contains the **current language state** and returns the right Error, Warning, Msg according to it.
  - **Context**'s role is played by class **Languages** that acts in different ways according to the **current language state**
  - **State**'s role is played by interface **Language**
  - **ConcreteState**'s role is played by **English**, **Italiano** and **Français**

## 4.4 Factory Method

Design Pattern Factory Method has been adopted once in this software :

- In class **EncrypterHandler**. **Encrypter**'s implementations are someway the main part of the software but as stated before user may desire to use one instead of another. During encrypt/translate execution an instance of **Encrypter** is needed. The right one is given by **EncrypterHandler**. **EncrypterHandler** does not only give the right implementation of **Encrypter** but also can call the right configuration method and tell if an implementation of **Encrypter** needs to be configured.
  - The role of **Product** is played by **Encrypter**
  - The role of **Concrete Product** is played by **StdCesare** and **StdMorse**
  - The role of **Creator** and **Concrete Creator** is played by **EncrypterHandler**

## 4.5 Adapter

Design Pattern Adapter is used twice in the software :

- In class **Words** where it hides an **ArrayList<String>**
  - The role of **Target** and **Adapter** is played by **Words**
  - The role of **Adaptee** is played by **ArrayList<String>**
- In class **Lines** where it hides an **ArrayList<Words>**
  - The role of **Target** and **Adapter** is played by **Words**
  - The role of **Adaptee** is played by **ArrayList<String>**

# 5 Code

## 5.1 Code : EcripterHandler

### Class EncrypterHandler

```
public class EncrypterHandler {  
    /**  
     * Enum Mod : all the possible Encrypters + fake value  
     */  
    public static enum Mod{  
        none,  
        StdCesare,  
        StdMorse  
    }  
    /**  
     * Method getEncrypter that returns the right encrypter according to the arguments  
     * @return instance of Encrypter  
     */  
    public static Encrypter loadConfig(Mod mod, String config){  
        switch (mod){  
            case StdCesare :  
                return StdCesare.getConfig(config);  
            case StdMorse :  
                return new StdMorse();  
            default:  
                return null;  
        }  
    }  
    /**  
     * Method set that calls the appropriate instance of EncrypterHandler.setConfig and returns outcome  
     * @return outcome  
     */  
    public static Boolean setConfig(Mod mod, String config){  
        switch(mod){  
            case StdCesare:  
                return StdCesare.setConfig(config);  
            default:  
                return false;  
        }  
    }  
    /**  
     * Method need config that returns if a modality needs configuration  
     * @param mod
```

```

    * @return
    */
public static Boolean needConfig(Mod mod){
    switch (mod){
        case StdCesare:
            return true;
        default :
            return false;
    }
}

```

As seen before class **EncrypterHandler** implements Design Pattern Factory Method. **EncrypterHandler** defines enum Mod that identifies each concrete implementation of **Encrypter** plus a fake value: none.

**EncrypterHandler** also defines methods :

- **loadConfig** that return the concrete implementation of **Encrypter**
- **setConfig** that calls the appropriate configuration method of **Encrypter**
- **needConfig** that checks if an **Encrypter** needs to be configured

## 5.2 Code : Languages

### Class Languages

```

/**
* Class Languages that implements desing pattern singleton and design pattern state with classes Language
*/
public class Languages {
    /**
     * Languages instance only instance of Languages
     */
    private static Languages instance = new Languages();
    /**
     * Method getInstance that returns the only instance of Languages
     * @return
     */
    public static Languages getInstance(){
        return instance;
    }
    /**
     * Enum Mod that represents the languages that can be used
     */
    public static enum Mod {

```

```

        en,
        it,
        fr
    }
    /**
     * Language current that is an instance of the current language
     */
    private Mod current = Mod.en;
    /**
     * Method setLan that sets the state current
     * @param l language mod desired
     */
    public void setLan(Languages.Mod l){
        current = l;
        GUI.getInstance().showMsg(instance.getMsg(Language.Msg.switchLan) +
l.name());
    }
    /**
     * Method getError that return the error string in the correct language
     * @param e code of error desired
     * @return error string
     */
    public String getError(Language.Error e){
        switch(current){
            case en: return English.getError(e);
            case it: return Italiano.getError(e);
            case fr: return Francais.getError(e);
            default : return Language.fake;
        }
    }
    /**
     * Method getWarning that return the warning string in the correct language
     * @param w code of warning desired
     * @return warning string
     */
    public String getWarning(Language.Warning w){
        switch(current){
            case en: return English.getWarning(w);
            case it: return Italiano.getWarning(w);
            case fr: return Francais.getWarning(w);
            default : return Language.fake;
        }
    }
    /**
     * Method getMsg that return the msg string in the correct language

```

```

    * @param m code of msg desired
    * @return msg string
    */
    public String getMsg(Language.Msg m){
        switch(current){
            case en: return English.getMsg(m);
            case it: return Italiano.getMsg(m);
            case fr: return Francais.getMsg(m);
            default : return Language.fake;
        }
    }
}

```

As seen before, class `Languages` implements :

- Design Pattern State where the state is represented by the `current language` (Mod current)
- Design Pattern Singleton because the `current language` has to be unique during whole execution.

## 5.3 Code : Encrypter Hierarchy

In this paragraph we are going to shortly show the hierarchical structure given by the concept of `Encrypter` :

- `StdCorrect`
  - `Encrypter`
    - `StdSyntax`
      - `StdDivider`
      - `StdFixed`
        - `StdCesare`
      - `StdVariable`
        - `StdMorse`

In order to make it better understanding we will not see concrete implementations of `Encrypter`.

### 5.3.1 Code : StdCorrect

<b>Abstract Class StdCorrect</b>
<pre> /**  * Abstract Class StdCorrect that implements a class where :  * - initial status is correct  * - status can be changed only to incorrect  * - correctness is an information that can be obtained from outside  */ public abstract class StdCorrect {     /** </pre>

```

 * Boolean error, check if Args is valid
 */
private Boolean correct = true;
/**
 * Method is correct that returns if Arguments are correct
 * @return
 */
public Boolean isCorrect(){
    return correct;
}
/**
 * Method setError that sets correct to false
 */
protected void setError(){
    correct = false;
}
}

```

At the top of this hierarchical system is **StdCorrect** that easily implements **Correctness State**. **StdCorrect** is set in package **corr**.

### 5.3.2 Code : Encrypter

#### Abstract Class Encrypter

```

/** 
 * Abstract class Encrypter that contains all the methods needed to encrypt
 */
public abstract class Encrypter extends StdCorrect {
    /**
     * GUI gui reference to GUI to make the code shorter
     */
    protected GUI gui = GUI.getInstance();
    /**
     * Method encrypt that returns Lines encrypted
     * @param ls Lines that has to be encrypted
     * @return Lines ls encrypted
     */
    public abstract Lines encrypt(Lines ls);
    /**
     * Method setConfig that creates a new configuration (config) using the GUI
     * @param config
     * @return
     */
}

```

```

public static Boolean setConfig(String config){return false;}
/*
 * Method getConfig that returns an Encrypter configured
 * @param config
 * @return
 */
public static Encrypter getConfig(String config){return null;}
}

```

**Encrypter** extends **StdCorrect** defining abstract method **encrypt** and methods **setConfig** and **getConfig** that should be overwritten by further implementations that need configurations. **Encrypter** also defines a reference to **GUI** just in order to make further code shorter and better understanding.

Concrete extensions of **Encrypter** must implement method **encrypt(Lines):Lines**.

Encrypter and its extensions are set in package **encr**.

### 5.3.3 Code : StdSyntax

#### Abstract Class StdSyntax

```

/**
 * Class StdSyntax that adds to Encrypter a validator
 */
public abstract class StdSyntax extends Encrypter {
    /**
     * Validator validator that validates the input
     */
    protected Validator validator = new StdAlfaNumericPoint();
}

```

**StdSyntax** extends **Encrypter** adding a syntax **Validator** further used.

### 5.3.4 Code : StdDivider

#### Abstract Class StdDivider

```

/**
 * Class StdDivider that partitions the encrypt task
 */
public abstract class StdDivider extends StdSyntax {
    /**
     * Method encrypt that returns Lines encrypted
     */
    public Lines encrypt(Lines ls){

```

```

        Lines r = new Lines();
        if(!validator.isValid(ls))
            setError();
        else
            for(Word ws : ls)
                r.add(encrypt(ws));
        return r;
    }
    /**
     * Method encrypt that returns Words ws encrypted
     * @param ws Words that has to be encrypted
     * @return words ws encrypted
     */
    protected Words encrypt(Words ws){
        Words r = new Words();
        for(String cs: ws)
            r.add(encrypt(cs));
        return r;
    }
    /**
     * Method encrypt that returns String cs encrypted
     * @param cs String that has to be encrypted
     * @return String cs encrypted
     */
    protected abstract String encrypt(String cs);
}

```

Abstract Class **StdDivider** extends **StdSyntax**. In this first version it just partitions the encrypting/translating process but does all the stuff sequentially. In further versions this feature could be used to implement parallelization.

**StdDivider** has implemented **Encrypter**'s method **encrypt(Lines):Lines** but further concrete extensions of **StdDivider** must implement method **encrypt(String):String**.

### 5.3.5 Code : StdFixed

#### Abstract Class StdFixed

```

/**
 * Class StdFixed that implements a char parsed encryption that does not encrypt spaces ad newlines
 */
public abstract class StdFixed extends StdDivider {
    /**
     * Method encrypt that returns String cs encrypted
     * @param cs String that has to be encrypted
     * @return String cs encrypted

```

```

/*
protected String encrypt(String cs){
    String r = "";
    for(char c : cs.toCharArray()) r += encrypt(c);
    return r;
}
/** 
 * Method encrypt that returns char c encrypted
 * @param c char that has to be encrypted
 * @return char c encrypted
 */
protected abstract char encrypt(char c);
}

```

Abstract Class **StdFixed** extends **StdDivider**.

**StdFixed** has implemented **StdDivider**'s method **encrypt(String):String** but further concrete extensions of **StdFixed** must implement method **encrypt(char):char**.

### 5.3.6 Code : StdVariable

#### Abstract Class StdVariable

```

/**
* Class StdVariable that implements Encrypter for systems where char -> String
*/
public abstract class StdVariable extends StdDivider {
    /**
     * Method encrypt that returns String cs encrypted
     * @param cs String that has to be encrypted
     * @return String cs encrypted
     */
    protected String encrypt(String cs){
        String r = "";
        for(char c : cs.toCharArray()) r+= encrypt(c);
        return r;
    }
    /**
     * Method encrypt that returns char c encrypted
     * @param c char that has to be encrypted
     * @return char c encrypted
     */
    protected abstract String encrypt(char c);
}

```

Abstract Class **StdVariable** extends **StdDivider**.

**StdVariable** has implemented **StdDivider**'s method `encrypt(String):String` but further concrete extensions of **StdVariable** must implement method `encrypt(char):String`.

# 6 Usage

As said in requirements and also many other times the software may be run in two different ways.

## 6.1 GEMc

Let's see configuration modality.

After installation the following BNF shows how to run it :

**GEMc (<language>) <configurable\_encrypter> (<configuration\_name>)**

where

<*language*> ::= **en** | **it** | **fr** is the language used. Default language is **en**.

<*configurable\_encrypter*> ::= **StdCesare** is an Encrypter that can be configured.

<*configuration\_name*> is any possible string that would be the name of the configuration. Default “**config**”.

for example running

**GEMc fr StdCesare myConfig**

allows to use the software in **french** setting the configuration **myConfig** for encrypter **StdCesare**.

## 6.2 GEM

Let's see encryption modality.

After installation the following BNF shows how to run it :

**GEM (<language>) <configurable\_encrypter> <configuration\_name> (<input\_mode> (<output\_mode>))**

or

**GEM (<language>) <not\_configurable\_encrypter> (<input\_mode> (<output\_mode>))**

where

<*language*> ::= **en** | **it** | **fr** is the language used. Default language is **en**.

<*configurable\_encrypter*> ::= **StdCesare** is an encrypter that can be configured.

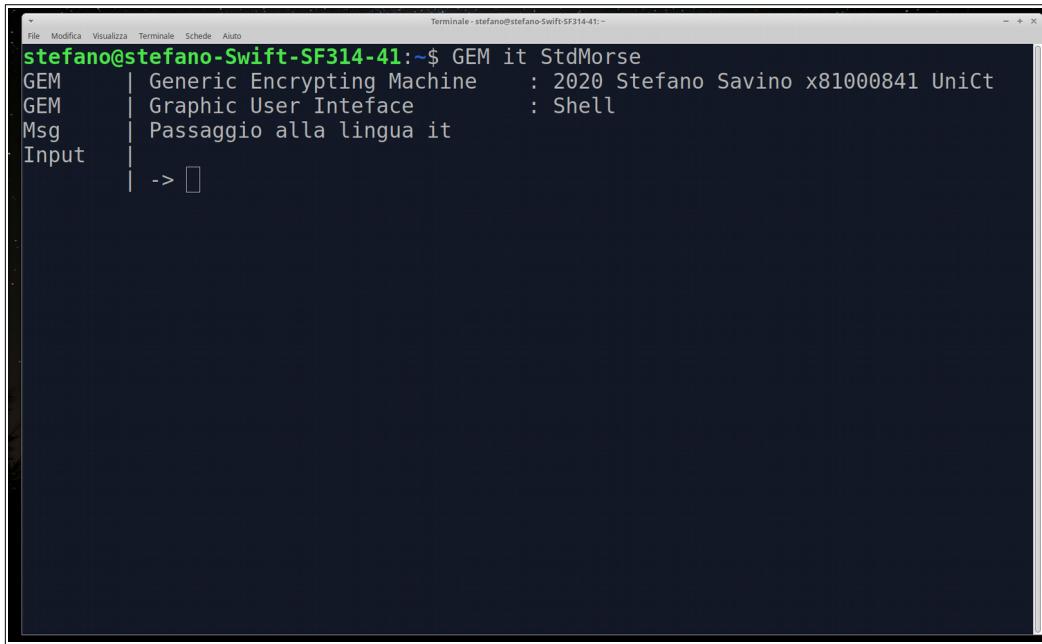
<*not\_configurable\_encrypter*> ::= **StdMorse** is an encrypter that cannot be configured.

<*configuration\_name*> is any possible string but configuration has to be defined before. (see 6.1)

<*input\_mode*> is “IO” for interactive use or the path of the source file. Default “IO”.

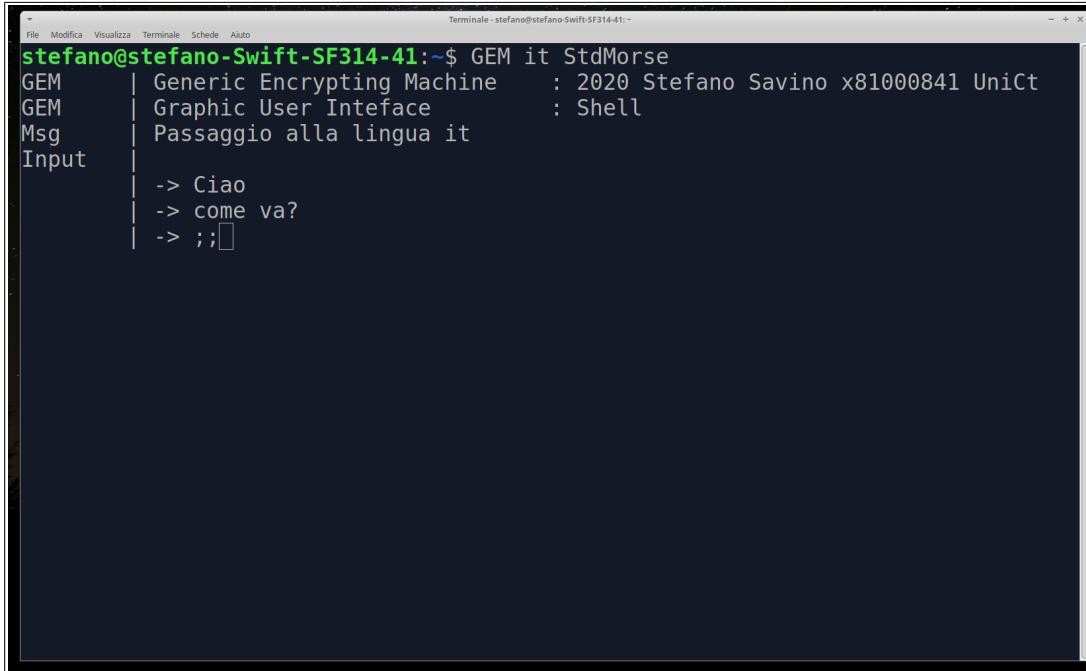
<*output\_mode*> is “IO” for interactive use or the path of the destination file. Default “IO”.

## 6.3 Example 1



```
stefano@stefano-Swift-SF314-41:~$ GEM it StdMorse
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Interface : Shell
Msg | Passaggio alla lingua it
Input | -> []
```

We invoke the program with “**GEM it StdMorse**” . We expect then the program to run in Italian the Encrypter StdMorse (Morse) with interactive input and output.



```
stefano@stefano-Swift-SF314-41:~$ GEM it StdMorse
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Interface : Shell
Msg | Passaggio alla lingua it
Input | -> Ciao
| -> come va?
| -> ;;[]
```

We type the text we want to be encrypted/translated. At the end in a single line we put the delimiter : “;;”. The message is ready to be encrypted.

The terminal window shows the GEM program running in Morse code mode. The user has inputted "Ciao", "come va?", and ";" into the "Input" field. The "Output" field displays the Morse code representation of the input. The program then outputs "A presto!" and exits.

```
steffano@stefano-Swift-SF314-41:~$ GEM it StdMorse
GEM | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Interface         : Shell
Msg  | Passaggio alla lingua it
Input | -> Ciao
Input | -> come va?
Input | -> ;;
Output | //---/.../..-/-//---/.../---/-//.../..-//-
GEM   | A presto!
steffano@stefano-Swift-SF314-41:~$
```

The output is shown and the program exits.

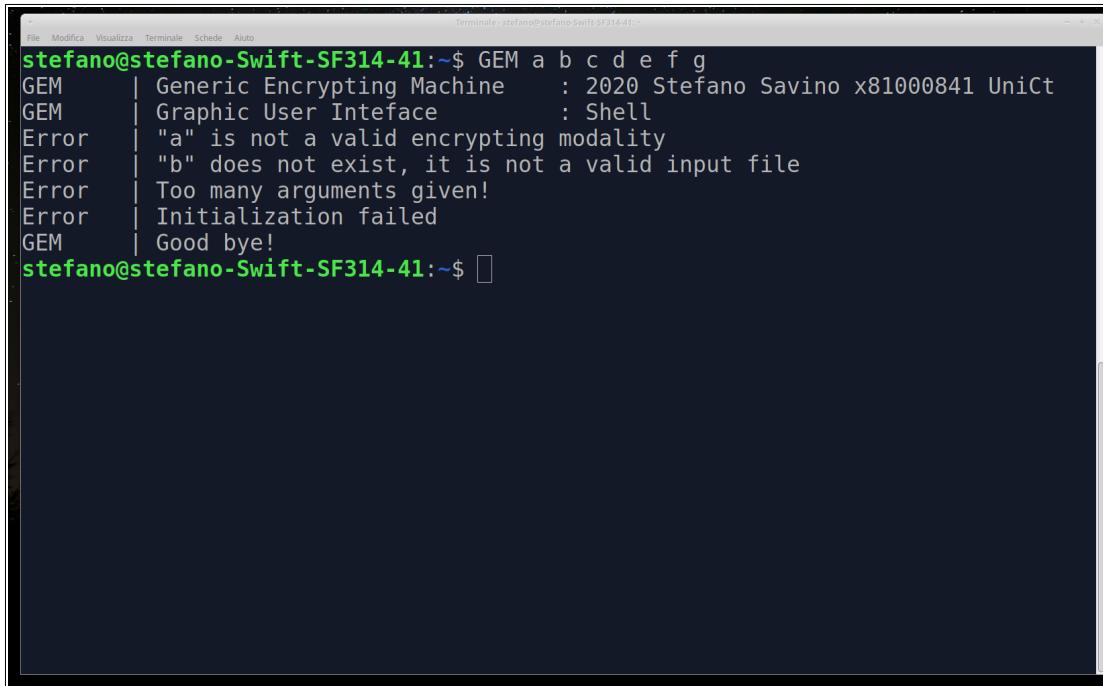
## 6.4 Example 2

The terminal window shows the GEM program exiting due to lack of arguments. The user has run the program without providing any input. The program outputs error messages about lack of arguments and initialization failure, followed by a "Good bye!" message and then exits.

```
steffano@stefano-Swift-SF314-41:~$ GEM
GEM | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Interface         : Shell
Error | Lack of arguments
Error | Initialization failed
GEM   | Good bye!
steffano@stefano-Swift-SF314-41:~$
```

We call the program with **lack of arguments**. The program shows the **error** and exits.

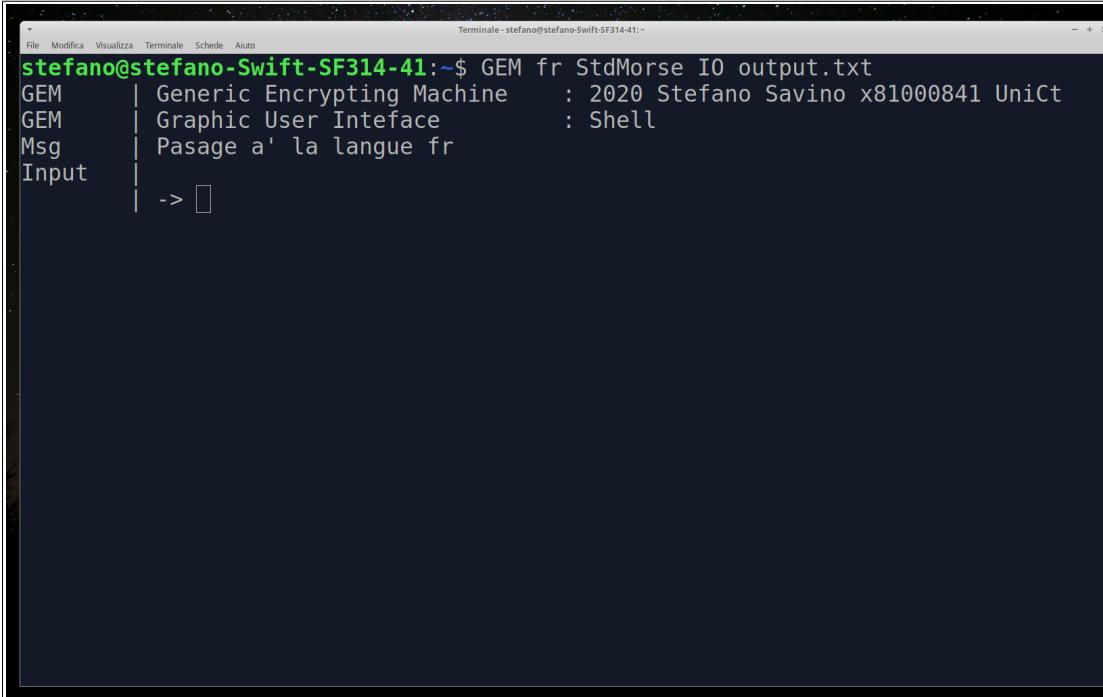
## 6.5 Example 3



```
File Modifica Visualizza Terminale Schede Aiuto
Terminale - stefano@stefano-Swift-SF314-41:~$ GEM a b c d e f g
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Inteface : Shell
Error | "a" is not a valid encrypting modality
Error | "b" does not exist, it is not a valid input file
Error | Too many arguments given!
Error | Initialization failed
GEM | Good bye!
stefano@stefano-Swift-SF314-41:~$
```

We call the program with casual arguments. The program shows all the **errors** found and exits.

## 6.6 Example 4



```
File Modifica Visualizza Terminale Schede Aiuto
Terminale - stefano@stefano-Swift-SF314-41:~$ GEM fr StdMorse IO output.txt
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Inteface : Shell
Msg | Pasage a' la langue fr
Input | -> □
```

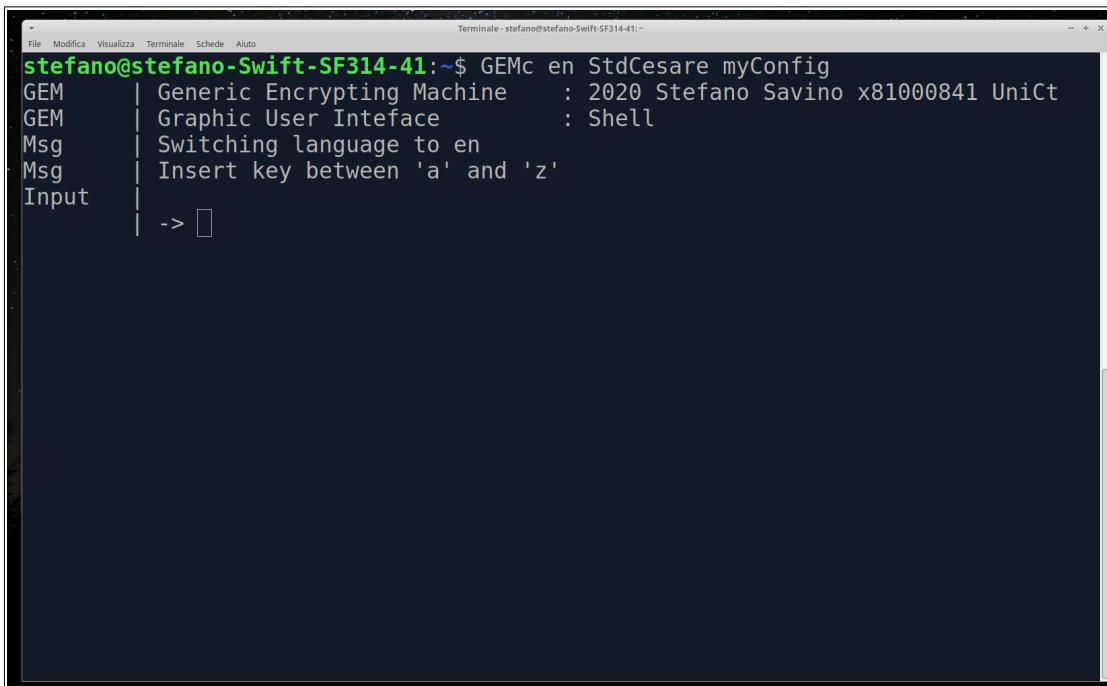
We call the program in **french**, asking for Encrypter **StdMorse**, interactive input and output in file **output.txt**.

```
File Modifica Visualizza Terminal Schede Aiuto Terminale - stefano@stefano-Swift-SF314-41:~  
stefano@stefano-Swift-SF314-41:~$ GEM fr StdMorse IO output.txt  
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt  
GEM | Graphic User Inteface : Shell  
Msg | Pasage a' la langue fr  
Input |  
      | -> Ceci n'est pas une pipe  
      | -> ;;  
Msg | Tache terminee avec succes  
GEM | A' bien tot!  
stefano@stefano-Swift-SF314-41:~$ 
```

We give the program the string that has to be translated. The program tells us the process has terminated successfully.

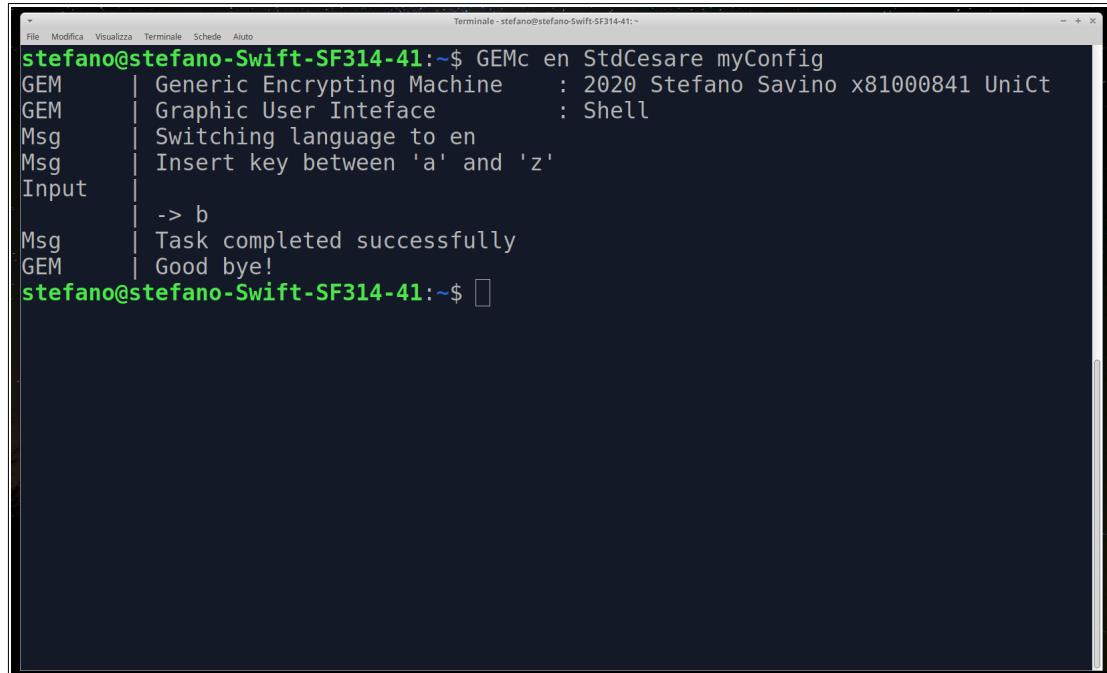
We have a look to the destination file and we find the encrypted message.

## 6.7 Example 5



```
steфано@стefанo-Swift-SF314-41:~$ GEMc en StdCesare myConfig
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Inteface : Shell
Msg | Switching language to en
Msg | Insert key between 'a' and 'z'
Input | -> □
```

We create a new configuration of **StdCesare** called **myConfig** running **GEMc** in English.



```
steфано@стefанo-Swift-SF314-41:~$ GEMc en StdCesare myConfig
GEM | Generic Encrypting Machine : 2020 Stefano Savino x81000841 UniCt
GEM | Graphic User Inteface : Shell
Msg | Switching language to en
Msg | Insert key between 'a' and 'z'
Input | -> b
Msg | Task completed successfully
GEM | Good bye!
stephanо@stefanо-Swift-SF314-41:~$ □
```

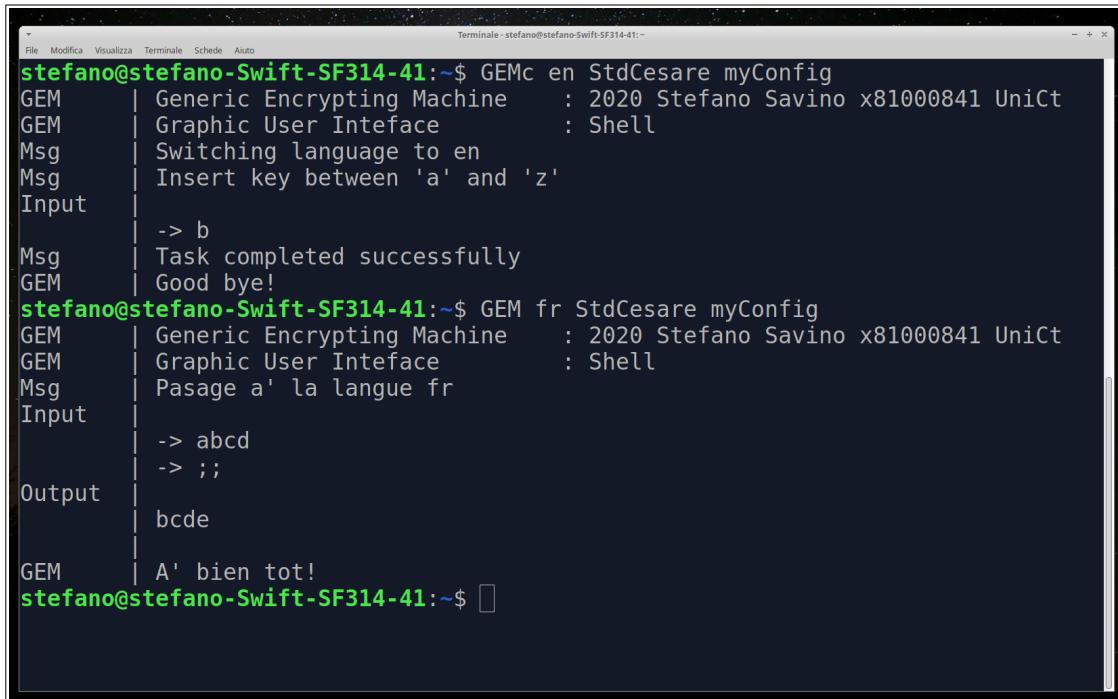
We follow instructions and make a valid configuration.

```
stefano@stefano-Swift-SF314-41:~$ GEMc en StdCesare myConfig
GEM | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCT
GEM | Graphic User Inteface          : Shell
Msg | Switching language to en
Msg | Insert key between 'a' and 'z'
Input
Error
Msg | Insert key between 'a' and 'z'
Input
| -> Wasabi
| This is not a key !
| Insert key between 'a' and 'z'
| -> 
```

The program takes care if errors are made by users.

```
stefano@stefano-Swift-SF314-41:~$ GEMc en StdCesare myConfig
GEM | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCT
GEM | Graphic User Inteface          : Shell
Msg | Switching language to en
Msg | Insert key between 'a' and 'z'
Input
| -> b
Msg | Task completed successfully
GEM | Good bye!
stefano@stefano-Swift-SF314-41:~$ GEM fr StdCesare myConfig
GEM | Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCT
GEM | Graphic User Inteface          : Shell
Msg | Pasage a' la langue fr
Input
| -> 
```

We call the Encrypter **StdCesare** loading the configuration just made, **myConfig**, running **GEM** in **French** with both input and output interactive.



The screenshot shows a terminal window titled "Terminale - stefano@stefano-Swift-SF314-41:~". The window contains the following text:

```
stefano@stefano-Swift-SF314-41:~$ GEMc en StdCesare myConfig
GEM      Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCt
GEM      Graphic User Inteface         : Shell
Msg      Switching language to en
Msg      Insert key between 'a' and 'z'
Input
      -> b
Msg      Task completed successfully
GEM      Good bye!
stefano@stefano-Swift-SF314-41:~$ GEM fr StdCesare myConfig
GEM      Generic Encrypting Machine      : 2020 Stefano Savino x81000841 UniCt
GEM      Graphic User Inteface         : Shell
Msg      Pasage a' la langue fr
Input
      -> abcd
      -> ;;
Output
      bcde
GEM      A' bien tot!
stefano@stefano-Swift-SF314-41:~$
```

We set the text to be encrypted and the software does all the work.

## 7 Summary

This first version of **GEM** is not able to do lots of stuff but it is ready to **expand** very easily and also to improve its behavior thanks to its **modularity**. StdCesare is the sixth class among its hierarchy, this could seem exaggerated for the little work it does but further implementations would be easy to think and realize making use of already made methods just expanding the right class.

Thanks for reading

Stefano Savino