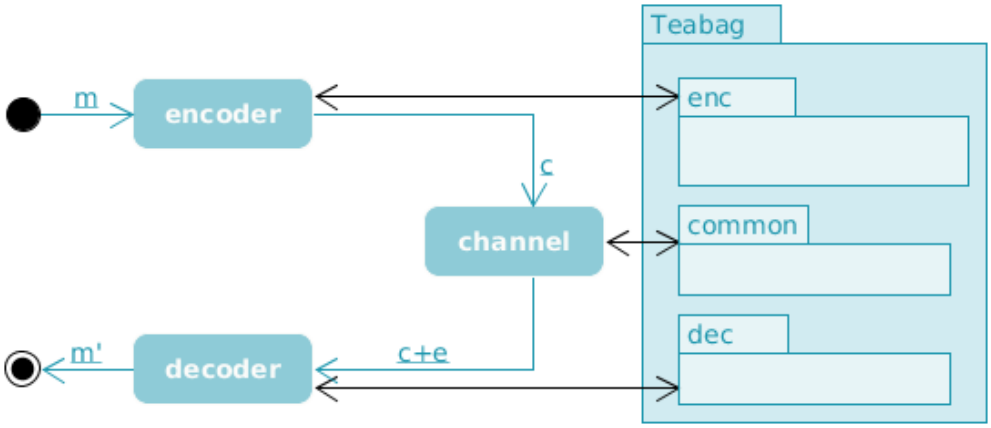
Seminar Thesis and Study Project

Application Programming Interface for Error Control Codes

Fikrat Talibli



Date of hand out:	December 1, 2018
Date of hand in:	June 14, 2019
Supervisor:	Dr.Ing Christian Senger

submitted to

University of Stuttgart
Institute of Telecommunications

Abstract

The topic of study project and seminar thesis is about application programming interface for error control codes called Teabag API. Initially the goal was to exploit generalized concatenated Reed-Solomon codes for synthesized DNA storage. As a choice programming language was Python and unfortunately there were no established error control coding libraries. The goal has been altered to develop an application programming interface for error control coding field with model driven software engineering techniques.

Software design patterns such as builder pattern have been utilized for development of so-called plugin system. Alongside the Teabag API we also provide documentation and abstract syntax - rules for development of generic system, for example software that can generate code, test cases and system components. Such a project can even lead to development of a domain-specific language.

Title page image: Mapping of error-control code system to Teabag API.

Contents

Notations

x	vector variables, especially in message vector
m, m', c	message, message estimate and codeword
$\mathbf{X}$	matrix variables: boldface upright upper case letters
$x(t)$, $p(x)$	functions of continuous variables: argument is placed in round parentheses
x_k	scalar elements of a vector or element of a sequence: element index is a subscript
$a \cdot b$, $\mathbf{H} \cdot \mathbf{x}$	scalar product of two vectors or matrices
G	generator matrix for error control codes
`variable`	brown typewriter font constitutes to any variable color
`use_function()`	blue typewriter font constitutes to any function color
ClassName, ClassName	camel case italic or default constitute to class name

1. Introduction to domain

Before going into realization of Teabag API and its details we first need to introduce the prerequisites of this work - coding theory, finite fields and Reed-Solomon codes - domain specifics in this work.

1.1. Coding theory

Coding theory is a field of mathematics where we study the codes in order to get reliable data transmission. There are several applications of coding theory: cryptography, data compression, error control coding and networking. Coding for each type is different:

- Cryptography: In cryptography the coding is called cryptographic and is used for construction and analysis of protocols that prevent the adversaries or 3rd parties from reading private messages. Cryptography is one of the extensively researched areas of computer science; the researches are also held in big companies.
- Error Control Coding: In error control coding the coding is used for error detection and its correction, resulted from a channel. Basically in between transmitter and receiver there is a channel which distorts the message. The channel is in fact necessary. For error correction there are mathematical tools available to recover the message.
- Data Compression: In data compression the ultimate goal is to remove redundancy from source. From the previous corresponds that the process therefore reduces the number of bits that are necessary to represent the information. Data compression is also referred to as source coding.
- Networking: In networking this is used more as correcting mechanism to identify and correct errors in a fly. In networking the reliability of data being received increases since possibility of receiving what you sent is going to be higher with correction of the errors.

1.2. Finite Fields

Finite field[1] is a mathematical term and has applications in many subjects such as control coding, cryptography and computer science. Basically as a name suggests, it is a finite set of numbers that consists of its own arithmetic, specifically *addition* and *multiplication*, by following some definite rules. Addition and multiplication behave in finite fields similar to real

and rational numbers. It is worth mentioning that *Finite Field* has additive and multiplicative neutral elements, which are normally 0 and 1.

Prime size Galois field is a finite field where the arithmetic is carried out by modulo. So basically, we construct the *addition* and *multiplication* tables, where the operation is carried out between corresponding elements. After the latter as soon as the value of element is greater than the number of field elements, in a given case size is prime, we determine the value by performing modulo operation - *mod(p)*. Quantity of elements in the finite field is its order. Each element has additive and multiplicative inverses except for 1 and 0. Multiplication and addition tables look like in Table 1.1 and Table 1.2.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 1.1.: Addition table for *mod(3)*

∘	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Table 1.2.: Multiplication table for *mod(3)*

According to Galois theory any field of power q and prime p can be constructed if there is given an irreducible polynomial $p(x)$ of order q. For example in order to construct F_{p^q}, where $p = 3$ and $q = 2$ one should use the following polynomial: $p_{3^2}(x) = x^2 + 2x + 2$. By using a given polynomial one can list all the possible elements (9 elements in a field) that will appear in a given field. As a consequence this polynomial will serve for a modulo operation as : $mod(p_9(x))$ while performing the operations like multiplication and addition. In fact all elements of this field will be constructed from a given polynomial and as a consequence we will have a field with 9 elements, that are listed below in list : 0 –> 0, 1 –> 1, 2 –> 2, 3 –> x, 4 –> x+1, 5 –> x +2, 6 –> 2x, 7 –> 2x+1 and 8–> 2x+2

1.3. Error control coding

Error control coding[2] is a field that deals with principles of getting the information from source to destination with minimum amount of errors. In principle for ECC system that is discussed in this work we deal with a message vector which goes through an encoder, traverses the channel and passes to decoder, where message is received. Detailed procedure of ECC system is depicted in Figure 1.1.

The message $m = (m_1, m_2, m_3, ..., m_k)$ is passed through `[enc]` where it is encoded into $c = (c_1, c_2, c_3, ..., c_n)$, here $n \geq k$. Then it passes through channel where the errors/erasures e are

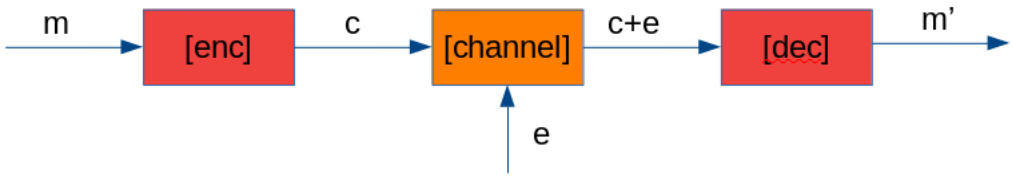


Figure 1.1.: Error Control Coding Model

added to the codeword $c+e$. Depending on a situation either the erasure occurs, where the symbols are deleted or errors, where the codeword changes its value. Then distorted codeword passes through `[dec]`, where the former is decoded into m',where m' is a message estimate.

Technique described above is useful for delivery of information through unreliable channels. Since communication channels are normally prone to erasures and errors ECC model is essential for transferring the information. Detection in ECC various methods enables to detect the error, and correction - to correct it.

1.4. Error correcting codes

American scientist Richard Hamming has led to creation of this field and invented the famous error correcting code family called Hamming Codes, e.g. $[F_2;7,4]$ - binary code with codeword length 7 and message length 4. Given system can correct 1 error. Error correcting code is a technique for adding redundancy to a message in a way such that it can be recovered using various mathematical methods. Mainly there are two types of error correcting codes:

1. **Algebraic Codes:** Given information is split into small pieces, where each piece corresponds to a message. Messages are consequently encoded into codewords (blocks). Examples of algebraic codes are Reed-Solomon codes, Hamming codes, Golay codes and e.g. These types of codes are based on Boolean algebra and also referred to as block codes.
2. **Convolutional codes:** Here the codes may operate on potentially infinite sequences of elements and not on finite blocks. However, there is no specific algebraic theory behind the construction of convolutional codes. Examples of techniques used on convolutional codes is Viterbi algorithm.

1.5. Generalized Reed-Solomon codes

Reed-Solomon[3] codes are class of algebraic codes that were introduced by I.S. Reed and G. Solomon. The class is characterized by following parameters: $q = p^m$ - field size, where

p is prime number, and m-is degree, k is message length and n is codeword length , where $k < n \leq q$. Field elements are constructed using finite field with size $q = p^m$, where q is also referred to as order. It is worth mentioning that for Reed-Solomon encoder and decoder we must provide sequence/vector of locators and multipliers which are α and β correspondingly. From above one can formulate the definition of Reed Solomon codes*[4]*:

A GRS code over F_q with locators $A = (a_0, a_1, ..., a_{n-1}) = \left(1, \alpha, ..., \alpha^{ord[\alpha]-1}\right)$ and multipliers $B = (b_0, b_1, ..., b_{n-1}) = \left(1, \alpha^{\delta}, ..., \alpha^{\delta(ord[\alpha]-1)}\right)$ for some $\alpha \in F_q$ with multiplicative order $n = ord[\alpha]$ and some $\delta \in \mathbb{Z}$ is a $[F_q; n, k]$ *Reed-Solomon Code*.

Generator matrix for encoding the message is obtained as follows:

$$G = \begin{bmatrix} 1 & \alpha^{\delta} & \cdots & \alpha^{\delta(n-1)} \\ 1 & \alpha^{1+\delta} & \cdots & \alpha^{(1+\delta)(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{k-1+\delta} & \cdots & \alpha^{(k-1+\delta)(n-1)} \end{bmatrix}$$

Above generator matrix is obtained from the locators and multipliers of RS code. In a given project decoding of the RS code is performed using the bi-variate polynomial interpolation. The algorithm for bi-variate polynomial interpolation is described in algorithm 1.1.

Algorithm 1.1 PE Decoder

Input: received_vector

1. *gather_points:* In this step simply the received locators are aligned with received vector as follows: $I = \{(a_i, \frac{r_i}{b_i}) : for\ i\ in\ range\ 0, ...n-1\}$
2. *interpolate_points:* In this step two polynomials Q_0, Q_1 are generated from solution space of homogeneous linear system of equations, calculated using mathematical operations (concrete mathematical formula is omitted from algorithm)
3. *obtain_message_estimate:* In this step message estimate is calculated using the formula: $f(x) = -Q_0(x)/Q_1(x)$

Output: message_estimate

The RS codes are able to correct $t = \frac{n-k}{2}$ errors with $d_{min} = n - k + 1$.

2. Structural Viewpoint of Teabag API

In development of any software it is necessary to have a corresponding hierarchy of modules, connection between them and models that describe various objects/classes in a project. Since in a given project we are developing the library it is necessary to have a precise structure alongside with clear documentation from the beginning. In a given context it is necessary to make a clear separation of modules.

In this chapter the techniques for such separation are described. As it has been explained in prerequisites of the work, the Error Control Coding can be viewed as set of independent fields. The goal of the work is to keep a clean separation between those fields, make them loosely-coupled and highly-cohesive, or as it is referred to by software engineers ability to test the program.

2.1. Introduction

It is necessary to choose the correct and suitable tools in the beginning in order to sustain the project later. As a programming language Python was a choice. Python is dynamically typed PL and has the garbage-collection capability, furthermore it supports various paradigms, such as object-oriented and functional programming. For the purposes of documentation and modeling UML 2.0[5] is used and serves as the foundation of models the project describes. This chapter will first introduce all necessary tools for achieving the structural division, elaborate on diagrams constructed using those tools and give an idea of implementation.

2.1.1. Python

The main reasons for choosing Python[6] were:

- lack of ECC libraries for Python,
- growth of usage specifically in academia,
- ability to support various paradigms,
- readable and easy-to use.

Python3 is specifically used in this project and as an additional tool ***numpy***[7] - one of the stable libraries for mathematical operations is also used;in future Teabag API can be independent of numpy.

2.1.2. UML 2.0

The reason behind choosing UML 2.0 for modeling were:

- its stability,
- broad documentation possibilities,
- it is backed up by OMG (consortium, that provides the standards for software engineers),
- there are plenty of reliable tools for modeling in UML 2.0.

The most important reason why UML2.0 has been chosen as a tool is ability to maintain a complex system with a help of structured documentation and capability to translate at least the structure to any object oriented programming language[8].

UML 2.0 has viewpoints and views[9] for modeling the software. Viewpoints are structural or behavioral entities (in a given context viewpoints are types of diagrams) that encompass the views, where the latter are represented by UML 2.0 diagrams. Views in this project will also be referred to as UML 2.0 diagram or simply diagram. For a given project the structural viewpoint is necessary, since it builds the modules of ECC Model. Exploitation of those models can be achieved using class diagrams.

2.2. Package structure of Teabag

Before diving into structure of Teabag in terms of class diagram it is necessary to make clear separation of packages for the model. In given project, the decision was based on ECC Model's structure from figure 1.1. The package diagram is shown in figure 2.1.

The package structure represents the ECC Model and has following parts:

- ***common:*** this package contains all the elements that potentially appear in major parts of Teabag library. For example, *reed_solomon_encoder_gf* file uses *Trial* and *ExtensionField/look_up_table.* Each module will be described later.
- ***enc***: this package contains all the necessary modules for encoder, as it is shown in ECC model. Since there can be multiple encoder types the decision was to create separate package for it.
- ***dec***: this package was created similar to *enc* package for decoder.

2.3. Common package

From Figure 2.1, one can see that there are six files, each with dedicated functionality. Files and correponding classes are enumerated below:

- *channel.py* - **Channel:** functionality of channel

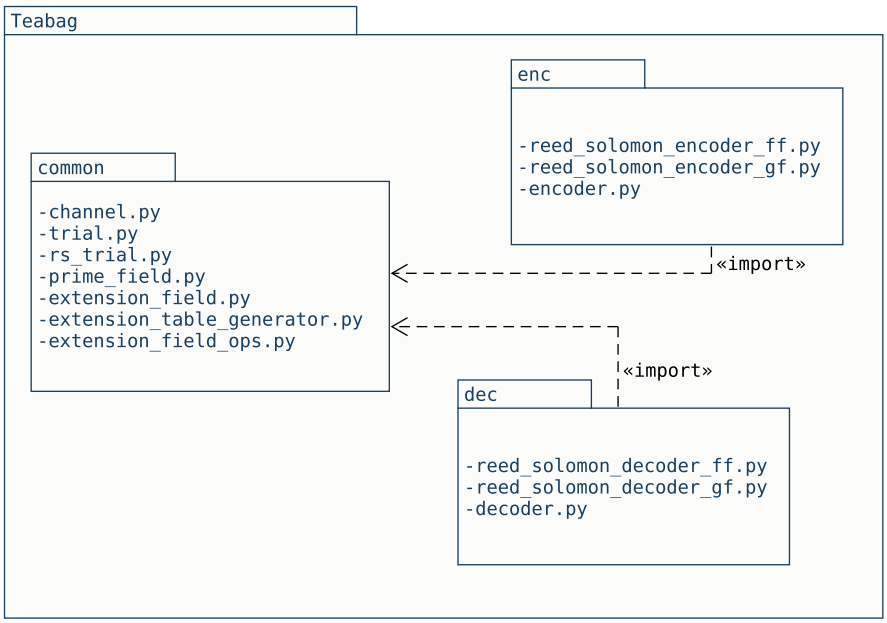


Figure 2.1.: Package diagram of ECC model

- *trial.py* - **Trial:** type that imitates transaction in field of communication
- *rs_trial.py* - **RSTrial:** type that imitates Reed-Solomon transaction in field of communication
- *extension_field_ops.py* - **ExtensionFieldOps:** describes all the operations from extension fields
- *extension_table_generator.py* - **ExtensionFieldTableGenerator**: used for generating all the possible relationships between the elements from galois field
- *extension_field.py* - **ExtensionField**: uses generated tables in order to store the possible relationships between the elements from ExtensionField with same strategy as PrimeField
- *primefield.py* - **PrimeField**: describes all the operations of *mod(p)* by overloading the arithmetics.

First it is necessary to establish formal logical separation between the modules. Figure 2.2 demonstrates that separation and structure(the diagram in this figure is purely logical and complies with OMG standards).

From figure 2.2 it is visible that *ExtensionField* and *PrimeField* classes are totally independent modules and can be tested absolutely without importing any other module. It is obvious that any look up table generator (where all the extension field operators are exploited for the sake of storage saving) must have at least one instance of ExtensionField. Now that big picture of classes is shown, one can describe each module individually.

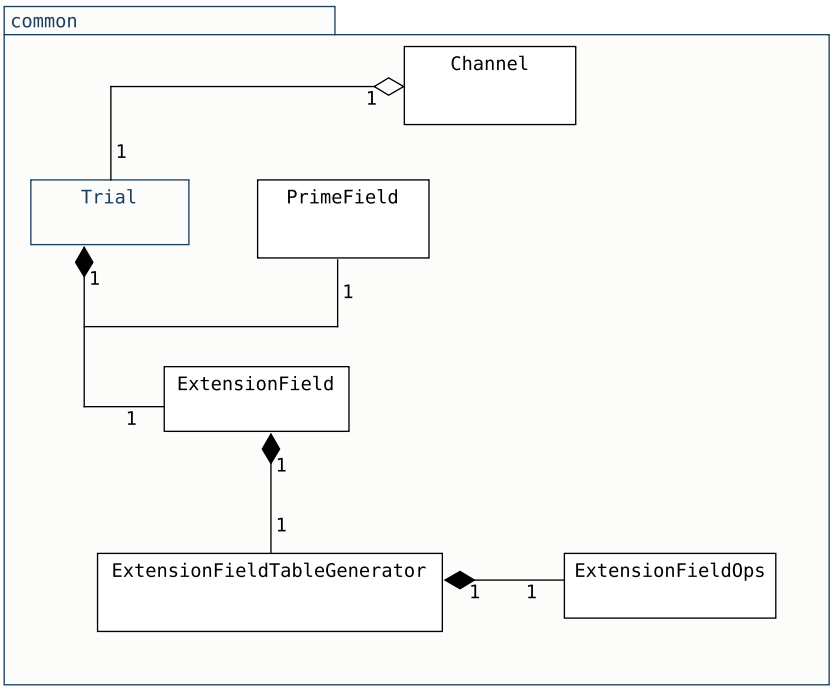


Figure 2.2.: Class-relationship diagram of common package

2.3.1. PrimeField

File prime_field in reality contains a method with inner class. It is a function that defines a class, which is called *PrimeField*. Basically this class is overloading the operators, since for prime field in coding theory $c_{mod(p)} = a_{mod(p)} + b_{mod(p)}$ which constitutes to: given elements $a \in F_p$ added with $b \in F_p$ equals $c \in F_p$. Using latter statement the PrimeField is constructed. Figure 2.3 demonstrates the definition of PrimeField class.

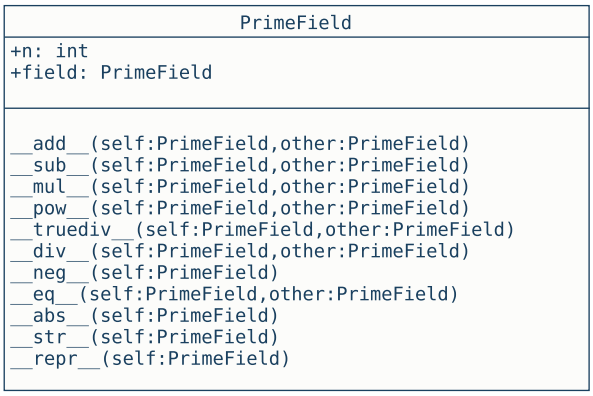


Figure 2.3.: PrimeField class

For example the operators such as + constitutes to the function **__add__** where latter takes two arguments of type PrimeField and adds them up correspondingly. All the other operators work using the same principle. Any object of type PrimeField constitutes to the element from $mod(p)$.

2.3.2. ExtensionFieldOps

ExtensionFieldOps is very similar to the *PrimeField* class except it makes use of algebra in extension field and serves as a helper for *ExtensionField*, here the operators are in field $mod(p(x))$, where that field is constructed over some polynomial $p(x)$. Figure 2.4 demonstrates definition of ExtensionFieldOps.

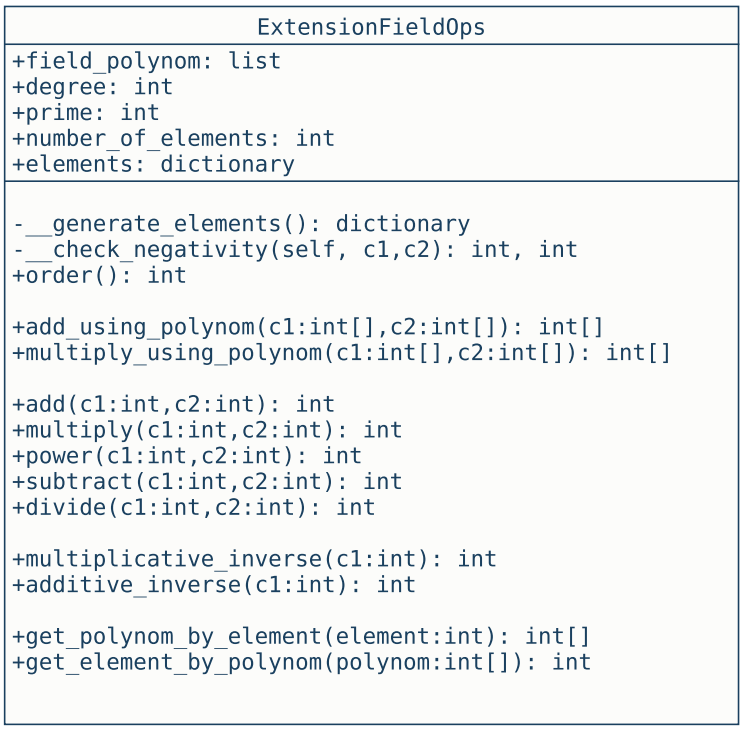


Figure 2.4.: ExtensionFieldOps class

From the figure it is clear that the operators here are not overloaded which means that it is not possible to exploit given class using the same strategy as in PrimeField. However, since the structure of extension fields is not as trivial as the *mod(p)* arithmetics, this class will only serve as a building block of the whole ExtensionField system in this project. From the beginning of a project it has been thought to have a kind of *"Look Up Table"* as shown in figure 1.2. Short description of a class is

- ***Attributes of ExtensionFieldOps:*** The field polynomial is defined as a list in Python or as an array of integers and is named *field_polynom*. Variable *degree* is an integer that

shows the degree of the field polynomial. Variable *prime* is an integer that represents prime element out of which the field is constructed. *number_of_elements* shows the quantity of elements in a field. Finally, the *elements* stores all the elements in dictionary or map and is generated using internal function *__generate_elements()*.

- ***Methods of ExtensionFieldOps:*** The functions `add`, `multiply`, `subtract`, `divide`, `power` correspond to mathematical operations +, *, –, /, e^p. Function *multiplicative_inverse* and *additive_inverse* correspond to inverses of elements. Get methods: *get_polynom_by_element* and *get_element_by_polynom* fetch either the value by key or key by value.

The module is completely independent and can be tested or applied in various field.

2.3.3. ExtensionFieldTableGenerator

ExtensionFieldTableGenerator serves as a helper class to construct the look up tables, which are further used in *ExtensionField* class. The goal of making the look up tables was to accelerate the software speed (pre-store in a dictionary/map all the field information from the beginning). Structure of *ExtensionFieldTableGenerator* is shown in figure 2.5.

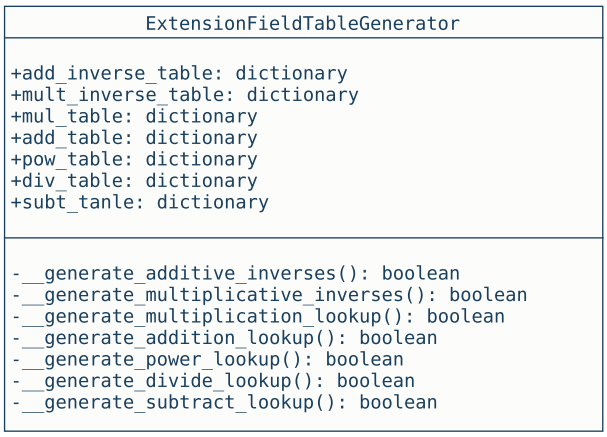


Figure 2.5.: ExtensionFieldTableGenerator class

It consists of attributes and methods that are basically generating dictionary of all combinations, where key is of type string and value is of type integer. Example of fetching the value by key is expressed as follows: `add_table["a*b"]`, where *a* and *b* are elements from the ExtensionField. The module is independent (except it must use *ExtensionFieldOps*) and can be tested.

2.3.4. ExtensionField

The working principle of the *ExtensionField* is very similar to the one from *PrimeField*. The method extension_field takes as an input three arguments: `degree, prime,`

`field_polynom` which constitute to degree of polynomial for field, prime number out of which the field is constructed and finally the polynomial $p(x)$, since it is $mod(p(x))$. The definition of class *ExtensionField* is shown in figure 2.6.

Attributes and methods defined here are all the same as in *PrimeField* class, with implementation appropriate for extension field.
This class can be tested independent of the other classes.

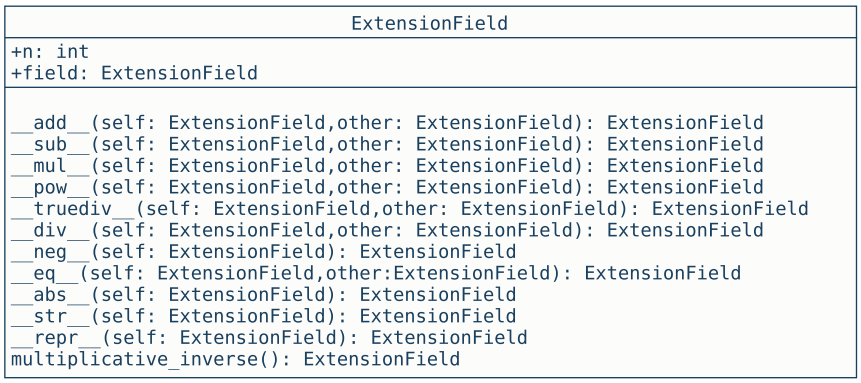


Figure 2.6.: ExtensionField class

2.3.5. Trial

Trial class defines the common attributes and methods of transmission in ECC Model . The definition of class is depicted in figure 2.7.

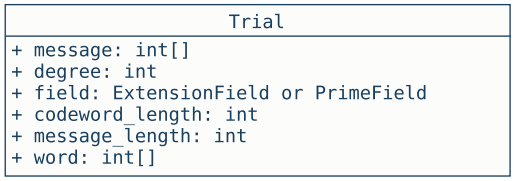


Figure 2.7.: Trial class

Trial class consists of following attributes:

- *Attributes*:

- **message** is of type int[] and constitues to the message used in src(from fig. 1.1), or after decoding as a message estimate. **degree** is the degree of polynomial used in Galois field. **field** is of type *ExtensionField* or *PrimeField* described in chapter 2.3.1 and 2.3.4. **word** is a list of type int[] which is further converted to Galois field.
- In order to define, for example Reed-Solomon trial one has to extend Trial class. For this project it has been decided to create *RSTrial*. **multipliers** and **locators** are `int[]` for Reed Solomon type decoder and encoder which are used in *RSTrial*.

The class can be tested independently.

2.3.6. Channel

Channel component works as a plugin system, which basically defines builder methods. For example here we have the generic channel and subsequently defined **apply_DMC_channel(trial,p=0.1)** method which serves as an individual plugin method, where DMC channel is exploited. Figure 2.8 shows the definition of *Channel* class.

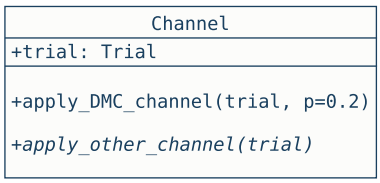


Figure 2.8.: Channel class

If it is required to create additional channel simply add method **<apply_other_channel(trial)>**. Channel class can be tested independently.

2.4. *enc* and *dec* Packages

The packages **enc** and **dec** are responsible for encoder and decoder portion of ECC Model correspondingly. The idea behind these both packages is based upon the *Builder* design pattern(from creational design patterns for software development[10]). According to definition of Builder design pattern is that construction of class *A* is separated from its representation, where some other class Builder is used in order to construct *A*. Figure 2.9 represents that idea for encoder and decoder.

From figure 2.9 it is clear that classes *Encoder* and *Decoder* are consequently builders of Reed-Solomon encoder and decoder and depending on field they pick which class to utilize. The choice of the Builder pattern came from the idea of the possibility of having multiple

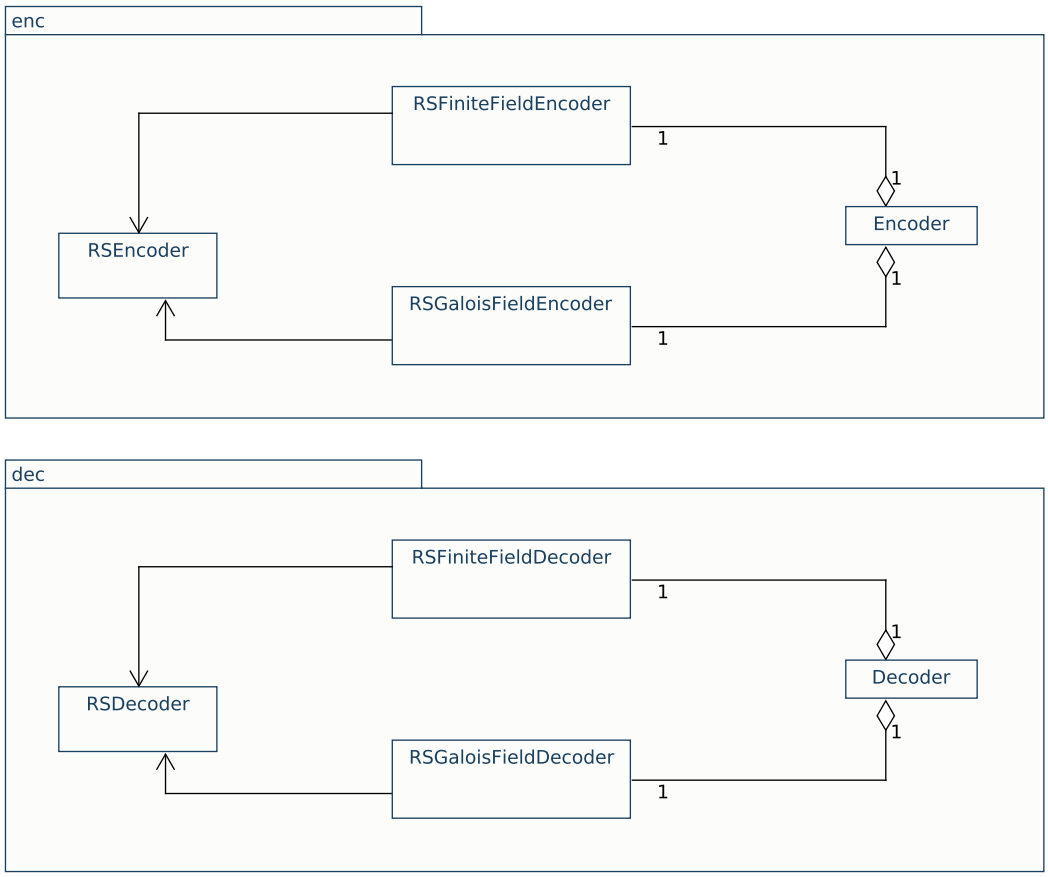


Figure 2.9.: *enc* and *dec* logical decomposition

decoders/encoders in dec/enc and choosing the corresponding one. Consequently, classes *Encoder/Decoder* serve as kind of a manifest file, where it is simply necessary to add the case to Encoder/Decoder as a method and utilize it. The utilization and definition of Decoder and Encoder are discussed in chapter 2.4.1.

2.4.1. Encoder and Decoder

The organization of *Decoder/Encoder* classes is based up on the possibility of having(removing or adding) multiple decoders/encoders[11] and utilizing the functions for additional encoder instantiation. Figure 2.10 demonstrates the way the classes are defined.

Given attribute `trial` is needed for the builder method of Decoder/Encoder that is going to invoke the implemented corresponding Decoder/Encoder.

Obviously, the definition and the implementation of the concrete decoder/encoder must be known to the developer - author of decoder/encoder.

From diagram 2.10 it is also obvious that there are four methods from Decoder/Encoder:

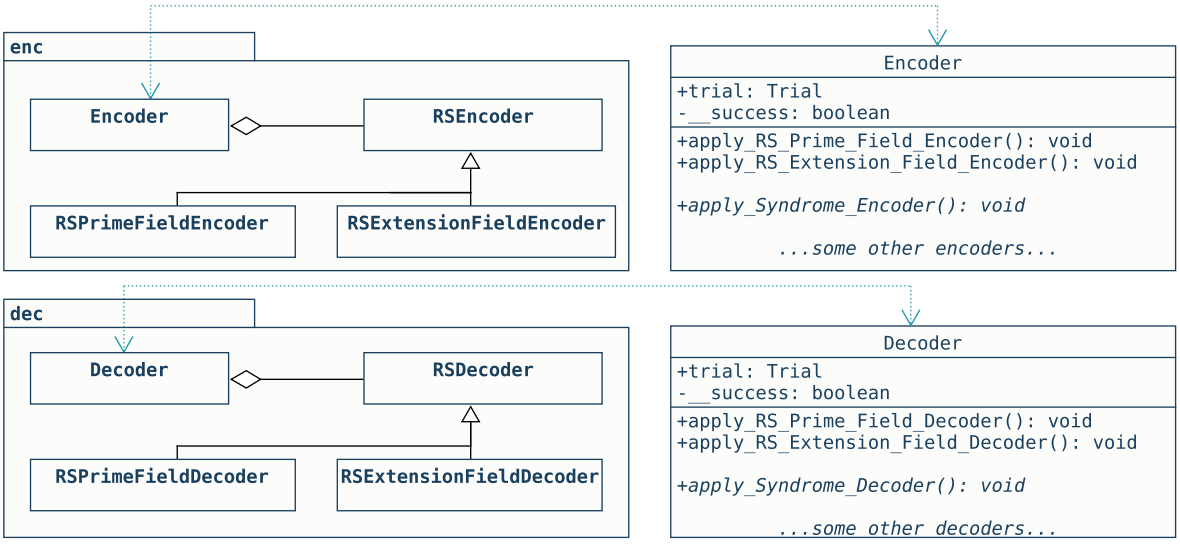


Figure 2.10.: Encoder and Decoder classes

`apply_RS_Prime_Field_Decoder()`, **`apply_RS_Extension_Field_Decoder()`**, **`apply_RS_Prime_Field_Encoder()`**, **`apply_RS_Extension_Field_Encoder()`** - all four take Trial-type argument. Even though theoretically concrete implementations for RSPrimeFieldEncoder and RSExtensionFieldEncoder can be used as one entity - since it is the same decoder but for safety type check is defined as two methods(since decoders take different arguments and implemented with slight difference).

The third method written in italic in figure 2.10 is used for the demonstration scenario:

Given a study project where one must write additional Decoder and Encoder - syndrome decoder and encoder per se.

1. Simply create additional class file with implementation of encoder and decoder,
2. Add a function *use_Syndrome_Decoder() and use_Syndrome_Encoder()* which does following:
   a) pass trial to standard SyndromeDecoder,
   b) return generated instance of the decoder decode method
3. While implementing decoder it is necessary to use the **`trial`** attribute of type ***Trial*** to define positional argument of constructor that is passed to syndrome decoder, while instantiating the latter in method.

The design of class is flexible enough and possible to utilize with ease.

2.4.2. RSEncoder

RSEncoder is a class responsible for the implementation of Reed-Solomon Encoder in $mod(p(x))$ or $mod(p)$ arithmetic. The structure of the class is shown in Figure 2.11.

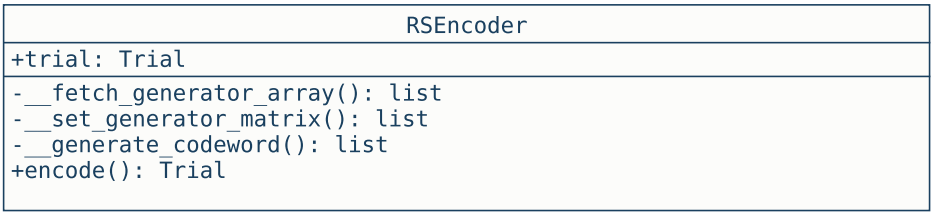


Figure 2.11.: RSEncoder class

Class is designed to encode the *RSTrial* object, through which the system communicates. Properties of the class are:

- *Attributes*: The attribute `trial` of type *RSTrial* is defining the attributes necessary for RS Codes.
- *Methods*: `__fetch_generator_array()`, `__set_generator_matrix()` and `__generate_codeword()` are helper methods designed to perform necessary mathematical operations for Reed Solomon Encoder. `encode()` returns corresponding *RSTrial* - generated trial out of message passed to Trial object.

2.4.3. RSDecoder

RSDecoder is a class responsible for the implementation of Reed-Solomon Decoder in $mod(p(x))$ or $mod(p)$ arithmetic. The structure of the class is shown in Figure 2.12.

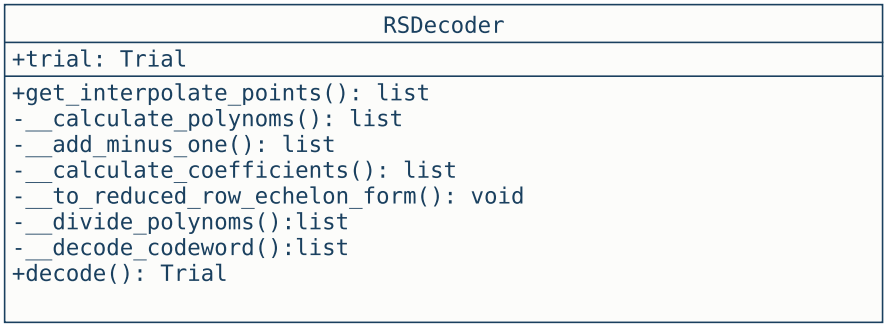


Figure 2.12.: RSDecoder class

Class is designed to decode the *RSTrial* object, through which the system communicates. Properties of the class are:

- *Attributes*: The attribute `trial` of type *RSTrial* is defining the attributes necessary for RS Codes.
- *Methods*: methods that start with underscores are helper methods designed to perform necessary mathematical operations for Reed Solomon Decoder. `decode()` modifies corresponding *Trial*.

2.5. Possible Improvements

Originally the structure exploited was designed with an idea directed towards implementation on Python purely. However it became clear that the structure is flexible and can be used for implementing in both dynamic and static programming languages, such as Java, JavaScript, C# and other. Even though the structurally organization of packages and classes is very similar to the ECC Model, the improvements, such as creation of a *IArithmeticOperations* Interface(in OO terms) in *ExtensionField* or *PrimeField* will be required. Naming conventions for other PLs also shall be different.

Also, if the project is implemented for Web and published on AWS (server provided by Amazon), one should consider the database option for *ExtensionField*, since generating ExtensionField at every execution requires memory.

And finally if we are in Java or C# world exposing attributes by making them public(for example *Trial* class) is not a good idea. It would be better to make the attributes of the class private and add setters and getters. Since current project is realized in Python, there was no need for getters and setters. Given that the idea of ECC model is very clear, one can simply restructure according to the guidelines of the desired programming language.

3. Behavioral Viewpoint of Teabag API

In software engineering it is necessary to have structural viewpoint while maintaining and developing additional components, but without the specifics of usage the structure is sometimes confusing. There are various techniques developed towards exploiting those specifics. For the Teabag API behavioral diagram in UML2.0 notation defined by OMG was chosen.

This chapter is concentrated around the behavioral viewpoint of Teabag API. It describes the communication between various classes and dynamic aspects of the system. Whole chapter is based on a scenario of ECC Model which is shown in figure 1.1.

3.1. Global Workflow of System

It is necessary to establish workflow and working algorithm of Teabag API. In order to establish the workflow, *statechart*[12] view from behavioral viewpoint is chosen. Figure 3.1 demonstrates the statechart diagram of Teabag API, how models shown in structural viewpoint communicate.

In order to understand the diagram from Figure 3.1 one should consider a scenario.

Scenario: Given a task to encode a message using Reed Solomon codes, pass it through channel and decode it. It is necessary to walk step by step through all activities from the statechart diagram and also try to associate the task statechart diagram shown in figure 3.1.

1. First in order to encode a message one shall create an instance of *RSTrial* and pass all the necessary parameters by simply instantiating them using attributes of the *RSTrial* class (in this case the message is passed as a `rs_trial_one` attribute).
2. Here one has to instantiate generic *Encoder*, *Channel* and *Decoder,* as shown in figure 3.1.
3. In order to encode the *Trial* object with a particular message, locators, multipliers and field operations one needs to pass the `rs_trial_one` instance to instance of *Encoder's* method `apply_Reed_Solomon_prime_field_encoder()`.
4. In order to distort the trial, one shall pass the `rs_trial_one` returned from encoder as an argument to the channel class's method, where one can choose the channel for distortion. Here as well one should choose the channel, for distortion of the `rs_trial_one`,

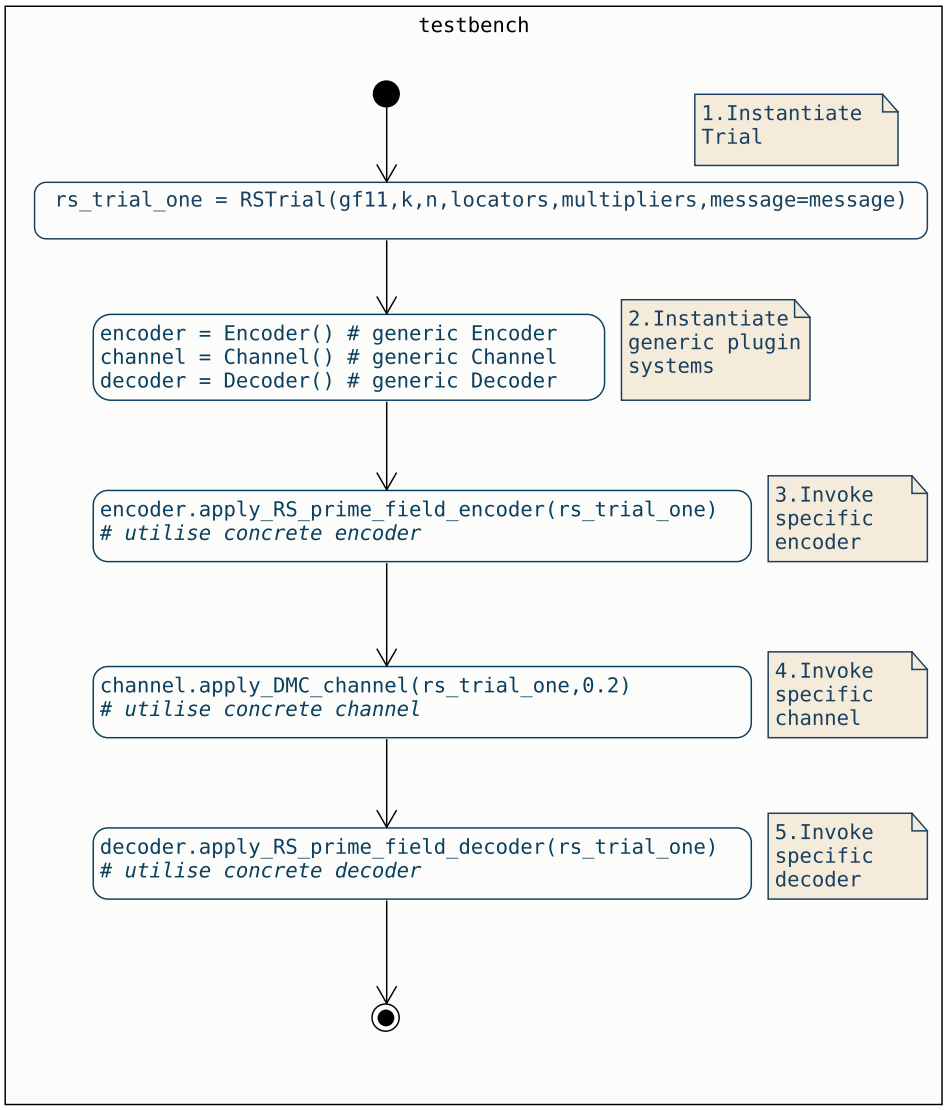


Figure 3.1.: `testbench` demo statechart

using the method **`apply_DMC_channel(trial,0.2),`** where the message will be erased and word distorted **`.`**

5. Using the same principle used in *Encoder* class, one has to repeat the same for *Decoder* class - pass **`rs_trial_one`** instance to method of Decoder and select the appropriate builder method.

Obviously at any point of the program the execution may stop (due to unpredicted outcome, such as invalid input or excessive calculations).

3.2. Local Workflow of Components in common package

Figure 3.1 demonstrated the global workflow of the whole system. This chapter will demonstrate the local workflow, or workflow of every component depicted in Figure 3.1. Additionally to statechart diagrams we will look at sequence diagram of a Teabag API as well. Sequence diagrams[13] are used to create documentations for readable and understandable requirements for developers and users of software products. They also show the communication between the instances of the classes, which consequently helps developers to have a comprehensible documentation.

3.2.1. Workflow of Trial

From figure 3.1 it is clear that first step in Teabag API system is *Trial* creation. Structure of Trial is not as trivial as its implementation (while implementing one line of code is required to create the Trial instance). Figure 3.2 displays the state diagram of trial instantiation(the elements of activity diagrams are also used while designing the diagram).

Consider scenario given in section 3.1. Activity diagram describes the first step of that scenario in a very detail. The description of the diagram is as follows:

- Before going into complexities of the Trial instantiation one shall initialize simple attributes such as message, word_length and e.g.
- As soon as the simple parts are initialized based on the passed arguments the program will select which field type to use:extension field based on $mod(p(x))$ or prime field based on $mod(p)$. Depending on that decision the Trial instantiates:
  - Prime field - 1 step process,
  - Extension field - 3 step process.

Figures 3.2 and 3.3 (`__init__()` stands for constructor in Python) demonstrate the message exchange and timeline of messages and their types sent in between corresponding classes. For prime field there is only one class invocation - *PrimeField.*

As it is seen from figures implementation of prime field is trivial. Since mathematical structure of extension field is quite sophisticated it was necessary to come up with structure that can handle complex calculations by using pre-calculation technique, here it is referred to as *"ExtensionField"*. As a result we have 3 classes responsible for extension field definition and implementation.

Considering scenario given in section 3.1 while creating the trial object will lead to following sequence of objects created (forward arrows in *Figure 3.3*):

1. As one of the initializations in constructor invocation of the *Trial* class API initiates call to *ExtensionField* class, where the arithmetic operations are overloaded(also refer to Chapter 2).

2. *ExtensionField* use the *ExtensionFieldTableGenerator* class in order to get the generated tables. They return the *dictionary* (*map* in "C++ or Java world") data type, where the search is performed.
3. Finally when *ExtensionFieldOps* class is reached the calculations are made, with the rules of *ExtensionField* calculations.

Since complex calculations are involved, the structure is quite sophisticated, but flexible enough to be able to implement in various programming languages.

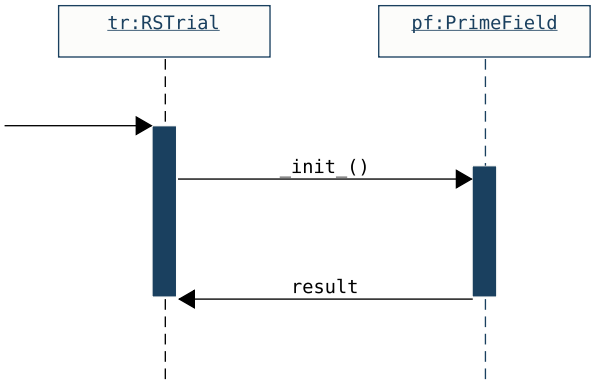


Figure 3.2.: PrimeField usage

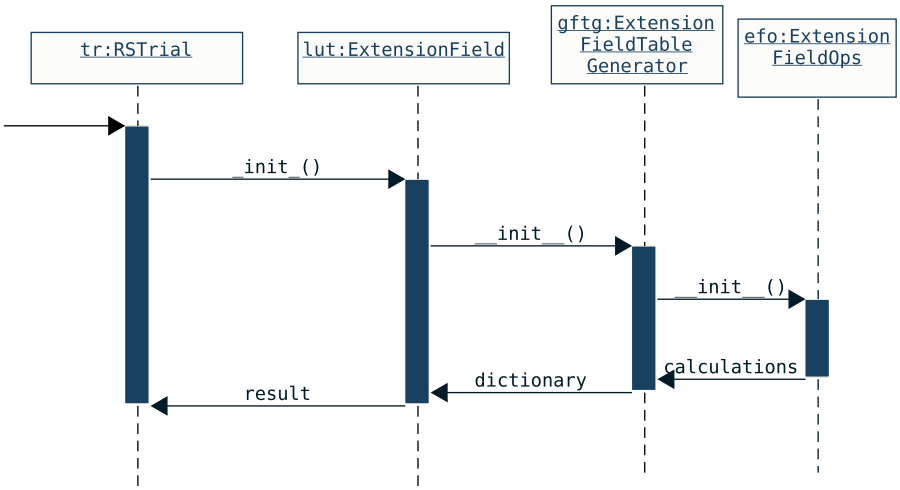


Figure 3.3.: ExtensionField usage

3.3. Workflow of channel

Distortion is another important concept in ECC Model and in a given project distortion is heavily based on the *Channel* class. Basically, the main concept of Channel in this project is mainly to assign the message to empty array and edit the value of attribute word in some manner (for example for DMC channel using random probability of each element in word). Obviously, since the structure is quite flexible, one can add his/her own channel, by simply adding the corresponding method, as described in chapter 2. Figure 3.4 demonstrates the correct usage of Channel.

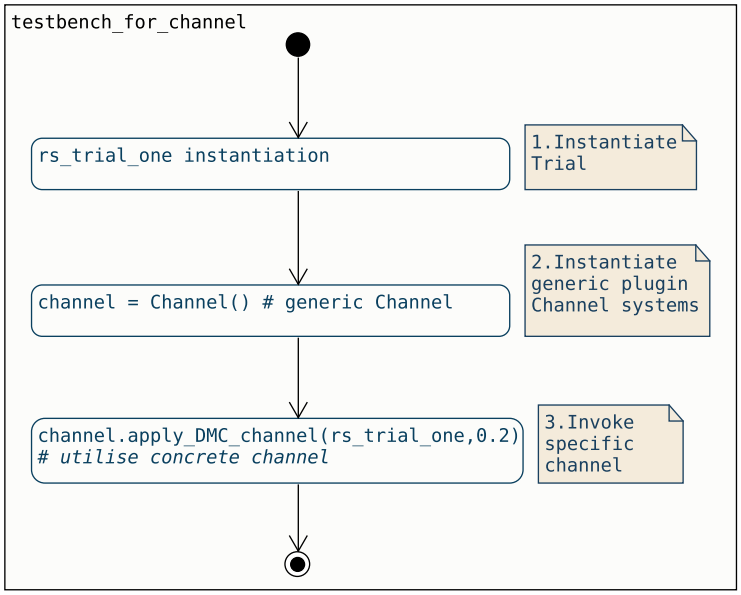


Figure 3.4.: `channel_testbench` statechart

Since the state diagram exploits all necessary aspects of the working principle of channel, sequence diagram is omitted (no interaction between the objects is performed, except for the modification of the rs_trial_one instance). The fields that are being modified are **`message`**(which is set to `[]` - empty array) and **`word`**(which is distorted at certain positions).

3.4. Workflow of encode and decode

From chapter 2 and algorithmical review of Figure 3.1 it is clear that the implementations of encoder and decoder are very similar. This section will be concentrated around those two states. Figure 3.4 shows the statechart representation of the **encode** and **decode** states implementation.

Taking into account the scenario, introduced in section 3.1 let us exploit step by step the

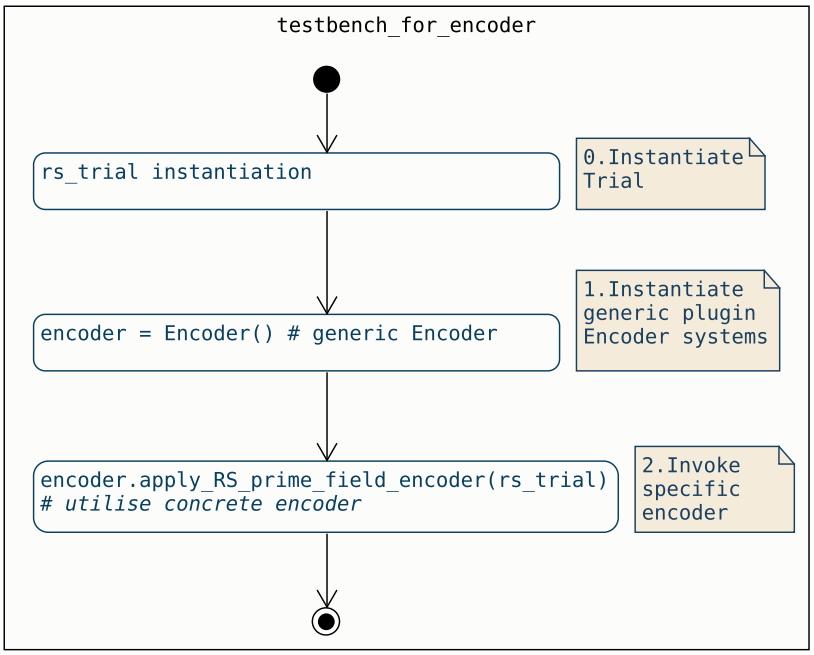


Figure 3.5.: Statechart diagram of **encode**

established working principle of **encode** (step 0 is not included) and **decode** states, using figures 3.5 and 3.6.

1. First the `encoder` is instantiated through *Encoder* class. Here the *Encoder* class is represented as a selector of one of the specific encoders. Please note you can use this instance of Encoder to call other plugins later on.
2. By calling the builder method of `encoder` instantiation of specific Encoder goes into motion. In a given example the method used is `apply_RS_prime_field_encoder`.
3. Then the modified *RSTrial* object is returned to the testbench, and `rs_trial` is ready for the further modification.
4. `rs_trial` first passes through the Channel for further work with the Decoder (described in section 3.3).
5. The working principle of `decoder` is exactly same as `encoder`. The only difference is algorithm of Decoder and Encoder.

Using the diagrams it is possible to recreate the project in various programming languages.

3.5. Possible adjustments

As it has been mentioned in structural viewpoint there are possibilities to adapt the Teabag API to a desired structure. Since the structure here is not exploited, in behavioral design one

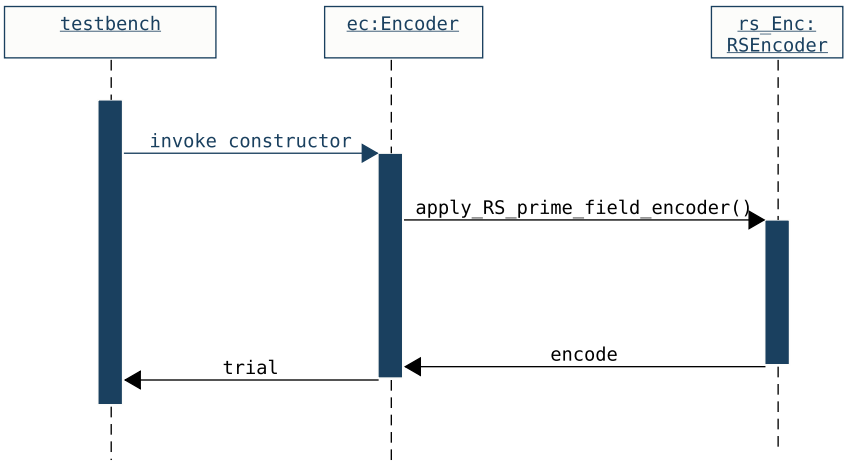


Figure 3.6.: Encoder sequence diagram

can adjust only algorithms the API library, for instance by optimizing specific functions, such as **`rref`** or **`long_division`**. Very obvious examples of improvements are exploiting the symmetry of look up tables or remove the usage of *numpy*.

There exist a lot of other functional viewpoints that we could have exploited for various mathematical functions and specifics of the system, but they are omitted in this project.

4. Testability and Usability of API

When it comes to sophisticated systems and APIs, developers mainly care about usability or testability. In this section it is necessary to go through the example of how library can be used for various applications.

4.1. Web-application

Given a scenario for a student to develop a simple web site where the look-up-tables are generated and the properties of extension fields are exploited. One of the most effective ways to approach this problem is to use Python's *flask* library for back-end and *bootstrap.css* for styling.

4.1.1. Toolset

Let us take a brief look at *flask[14]* and *bootstrap.css[15]* overview.

Flask is a framework for creating web applications in the Python programming language using the *Werkzeug* toolkit, as well as the *Jinja2* template engine. It belongs to the category of so-called micro-frames - minimalist web application frameworks that consciously provide only the most basic features. It has been developed by Armin Ronacher, Austrian programmer. Also one can use *bootstrap.css* for styling the work, but it is not necessary. Installation of both microframeworks is comprehensive and is omitted in this section.

4.1.2. Development basics

Now that development tool set is established let us start developing the website. Since it is a very simple static page, which just loads and shows the Galois Field look up tables only one route and one function is necessary: `@app.route("/")` and `welcome()`-which means "when routing to the web-page '/' execute welcome function.

4.1.3. Functionality development

Now that main mechanism of flask is exploited one shall go to functional development of the web application. Now we consider the storage mechanism for the tables and the most effective

and easy way are arrays, where all possible calculations are converted to string and outputted to the website. So, the function will look like in listing 4.1:

```
def format_for_lp_tables(a,b,res,op):
        """Function responsible for oututting as string"""
        return str(a) + op + str(b) + "\t=\t" + str(res)
```

Listing 4.1: Python formatter for web-app

Now one shall develop function for testing the lookup tables. Algorithm for testing is demonstrated in listing 4.2(comments allow non-Python programmers to understand the code).

```
def test_look_up_tables(q,p,polynom):
    #initialize the lookuptables
    lp_tables = extension_field(q,p,polynom)
    #declare array for each of the operations
    #sum_arr=+ mul_arr=* sub_arr=- pow_arr=to_the_power_of

    sum_arr=[]
    mul_arr=[]
    sub_arr=[]
    pow_arr=[]

    #iterate over all possible combinations of p to the power q
    for i in range(p**q):
        for j in range(p**q):
                #assign a to wrapped elements as look_up_tables
                a=lp_tables(i)
                #assign b to wrapped elements as look_up_tables
                b=lp_tables(j)
                #assign res to sum of elements from field
                res=a+b
                #assign sum_arr to sum of elements from field
                sum_arr.append(format_for_lp_tables(a,b,res,"\t+\t"))

    #repeat same for sub_arr
    for i in range(p**q):
        for j in range(p**q):
                a=lp_tables(i)
                b=lp_tables(j)
                res=a-b
                sub_arr.append(format_for_lp_tables(a,b,res,"\t-\t"))

    #repeat same for mul_arr
    for i in range(p**q):
        for j in range(p**q):
                a=lp_tables(i)
                b=lp_tables(j)
                res=a*b
                mul_arr.append(format_for_lp_tables(a,b,res,"\t*\t"))

    #repeat same for pow_arr
    for i in range(p**q):
```

```
42          for j in range(p**q):
43                  a=lp_tables(i)
44                  b=j
45                  res=a**b
46                  pow_arr.append(format_for_lp_tables(a,b,res,"\t^\t"))
47
48      return sum_arr, mul_arr, sub_arr, pow_arr
```

Listing 4.2: Python Look Up Tables test

From Program Listing 4.2 notice at line three the ease of instantiation of *extension_field*. Also it is worth mentioning how the mathematical operators work at lines 20-addition,29-subtraction,37-multiplication,45-power. Simply by wrapping the elements, one can easily add, subtract, multiply, divide and perform other mathematical operations on the elemens.

In order to get the desired result to the web page which is served, per se locally one shall write the main welcome function. It is shown in listing 4.3.

```
1  @app.route("/")
2  def welcome():
3          #assign the corresponding arrays to output of function
4          #here we are looking at Galois Field over x^3+x+1
5          sum_arr, mul_arr, sub_arr, pow_arr=test_look_up_tables
               (3,2,[1,0,1,1])
6          #pass the result to home.html and variables to jinja environment
7          return render_template('home.html',len = len(sum_arr), sum_arr =
               sum_arr, sub_arr=sub_arr, mult_arr=mul_arr,pow_arr=pow_arr)
```

Listing 4.3: Render webpage

Finally one shall write the jinja statement in html in order to get the desired result as a web-page. Excerpt from Jinja statement is shown in listing 4.4(specifics of Jinja statements are ommited in the chapter).

```
1  <div class="col">
2     <div class="alert␣alert-warning" role="alert">
3        <h3>Plus</h3>
4      </div>
5      {%for i in range(0, len)%}
6        <p>{{sum_arr  [i]}}</p>
7      {%endfor%}
8    </div>
9  <div class="col">
10   <div class="alert␣alert-danger" role="alert">
11     <h3>Subtract</h3>
12   </div>
13   {%for i in range(0, len)%}
14     <p>{{sub_arr  [i]}}</p>
15   {%endfor%}
16 </div>
17 <div class="col">
18   <div class="alert␣alert-info" role="alert">
```

```
19      <h3>Multiply</h3>
20    </div>
21    {%for i in range(0, len)%}
22      <p>{{mult_arr  [i]}}</p>
23    {%endfor%}
24  </div>
25  <div class="col">
26    <div class="alert alert-primary" role="alert">
27      <h3>Power</h3>
28    </div>
29    {%for i in range(0, len)%}
30      <p>{{pow_arr  [i]}}</p>
31    {%endfor%}
32  </div>
33  </div>
```

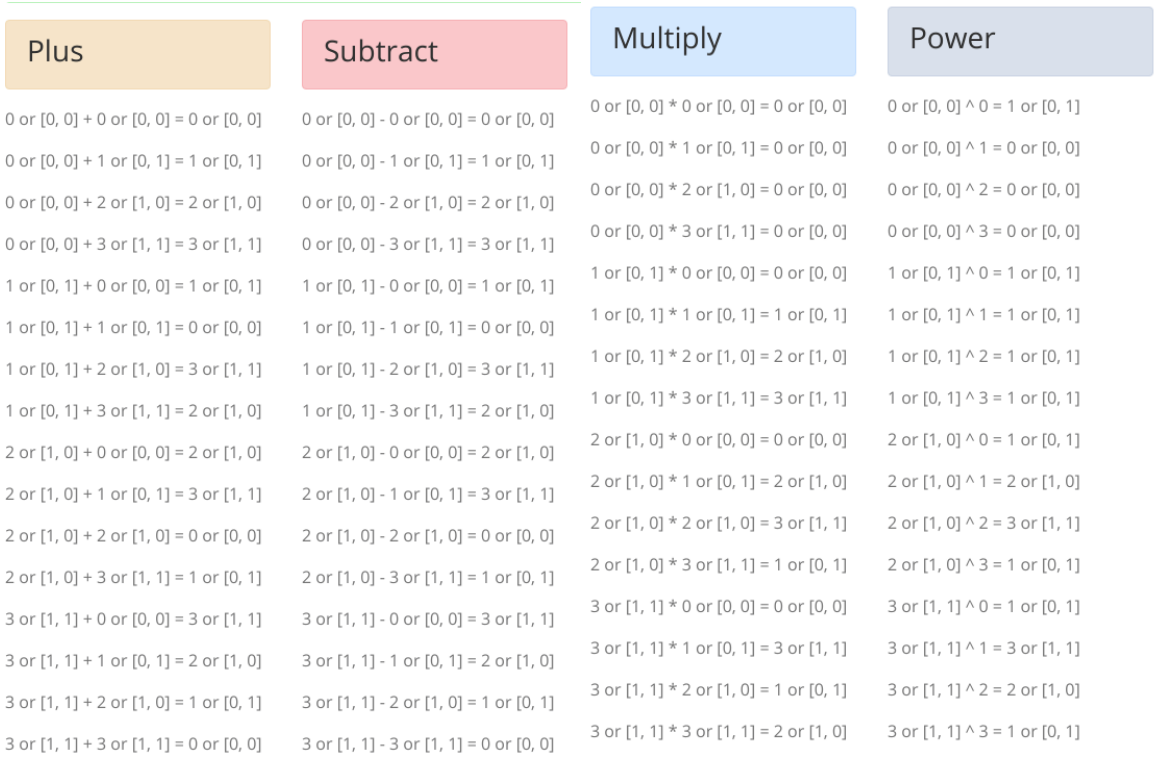
Listing 4.4: Webpage

The program from listing 4.4 is simply iterating over the array of elements, passed to the rendered web-page at line seven shown in listing 4.3.

The web output served at *localhost:5000* is shown in figures 4.1.a and 4.1.b. As seen from figures the representation of the elements is **<element>** and **<polynom_array>.**

4.2. Adjustments

Possible adjustments will depend on how developers want to use given library. For example the function **__repr__** can be helpful while one wants to represent the element from galois_field in certain manner (from figures 4.1.a and 4.1.b they are represented as element and polynomial array). Slight adjustments can be made to the layout of the table for web, but it is just a demonstration of how easily one can work with library.



(a) Addition and subtraction output

(b) Multiplication and power output

Figure 4.1.: Output of developed website

5. Future Works

This work is not only the realization of API for Python programming language, but also a well-established reference for the ECC Model from the perspective of Model-Driven Software Engineering(Development). The large companies, such as IBM and Vector, are using the ideas behind MDSD. Since the work has the model, which is heavily based on object management group's meta object facility(company has influenced to development of object oriented programming) with the idea that one can re-utilize the schemes and write the library in other programming languages. In MDSE this is called abstract syntax or meta-models.

I think the work can lead also to development of a software - ECC model generator. The idea behind the model generators is that there is a well-established model, main parts of which can be left invariable. For the ECC Model they are: *Encoder*, *Channel*, *Decoder* and *Trial*. Since we already have the abstract syntax, only the concrete syntax will be required. The software shall also include some kind of an interpreter, and rules for the compiler/interpreter. Rules for compiler/interpreter can be developed using dynamic semantics (knowledge of the ECC as a field) and compiler/interpreter itself using static semantics (here the knowledge of an expert is required). In future the software can serve as a standard and product for developing generating the ECC Model. One more prospect of this project is the creation of a domain-specific language, which is specifically designed for error control coding field.

A. Appendix

The work also contains source codes and documented models. In order to access the files:

- refer to git repository for Teabag API in `python/sources` directory,
- refer to the **.uxf** or **.pdf** files for UML2.0 models in `docs` folder of repository,
- refer to demo in git repository under `python/sources.`

Bibliography

[1] G. L. Mullen and D. Panario, *Handbook of Finite Fields*, 1st ed. Chapman & Hall/CRC, 2013. (p. 1)

[2] R. Roth, *Introduction to Coding Theory*. New York, NY, USA: Cambridge University Press, 2006. (p. 2)

[3] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. DOI: 10.1137/0108018 (p. 3)

[4] C. Senger, "Lecture notes in error control coding: Algebraic and convolutional codes," Institute of Telecommunications, Stuttgart, 2018. (p. 4)

[5] T. Goldschmidt, S. Becker, and E. Burger, "Towards a tool-oriented taxonomy of view-based modelling," in *Modeling 2012*, E. J. Sinz and A. Schuerr, Eds. Bonn: Society for computer science e.V., 2012, pp. 59–74. (p. 5)

[6] "Python homepage," https://www.python.org/, accessed: 2018-07-14. (p. 5)

[7] "Numpy homepage," https://www.numpy.org/, accessed: 2018-07-14. (p. 5)

[8] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of uml documentation on software maintenance: An experimental evaluation," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 365–381, Jun. 2006. DOI: 10.1109/TSE.2006.59 (p. 6)

[9] S. Becker, "Lecture notes in model-driven software engineering," Institute of Reliable Software Systems, Stuttgart, 2018. (p. 6)

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. (p. 12)

[11] T. Duerschmid, "Design pattern builder: A concept for refinable reusable design pattern libraries," pp. 45–46, 11 2016. DOI: 10.1145/2984043.2998537 (p. 13)

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987. DOI: 10.1016/0167-6423(87)90035-9 (p. 17)

[13] OMG, "OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1," August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1 (p. 19)

[14] "Flask library documentation," http://flask.pocoo.org/docs/1.0/, accessed: 2018-07-14. (p. 24)

[15] "Bootstrap library homepage," https://getbootstrap.com/, accessed: 2018-07-14. (p. 24)

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Das elektronische Exemplar stimmt mit den gedruckten Exemplaren überein.

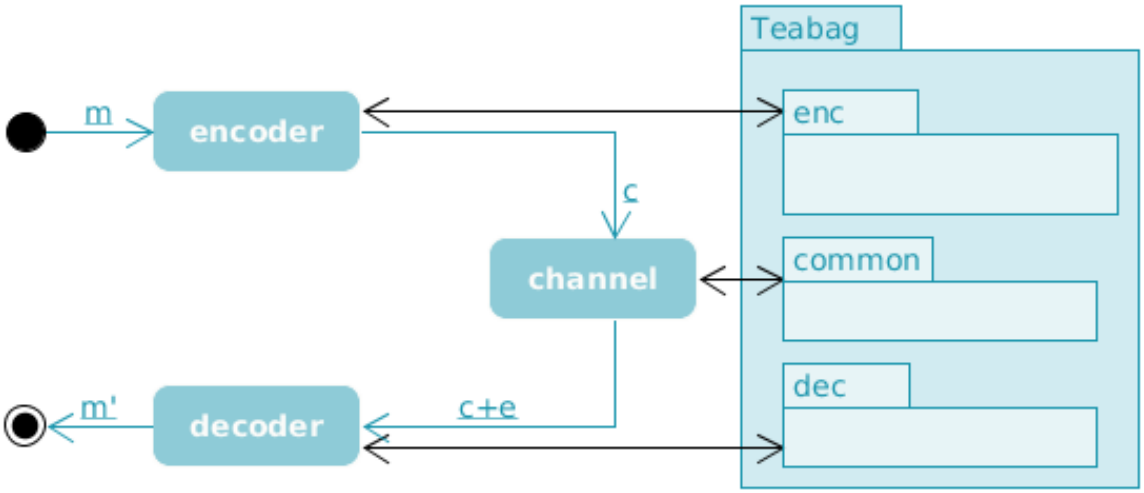
Stuttgart, 14. Juni 2019

Fikrat Talibli

Seminar Thesis and Study Project

Application Programming Interface for Error Control Codes

Fikrat Talibli



Date of hand in: June 14, 2019
Supervisor: Dr.Ing Christian Senger

Abstract

The goal was to develop an application programming interface for error control coding field with model driven software engineering techniques. Software design patterns such as builder pattern have been utilized for development of so-called plugin system. Alongside the Teabag API we also provide documentation and rules for development of generic system - software that can generate code, test cases and system components. Such a project can also lead to development of a domain-specific language.

submitted to

University of Stuttgart
Institute of Telecommunications