Fsk Encoder Developers Guide

October 2025

Contents

List of Figures

Introduction

The Fsk Encoder application is a useful tool for SW development in retro computing environment.
It's capable of converting binary code or data files into FSK encoded sound samples which can be played on the computers sound card. With an appropriate interconnection cable, the sound output of the host computer can be connected to the sound input of a retro computer system to upload code or data.

Also it's extendible to support new target systems or different source file formats.

This document gives (hopefully) all information needed to set up the Eclipse project for program maintenance and extension development.

For a better understanding of this document, please refer to the Javadoc of the API classes and interfaces which can be found in the subpackages of the `extension` package.

1 Project setup

The project is split into two parts:

1. The FskEncoder-Application, which is only a 'mainframe' and needs at least one input file reader and one target system extension to be ready for work. It can be downloaded from Github
   `https://github.com/kamaso-macha/FskEncoder-Application`.
2. The FskEncoder-Extension, which serves the different file reader and target system extensions. This project can also be downloaded from Github using the URL
   `https://github.com/kamaso-macha/FskEncoder-Extensions`.

1.1 3pty Libraries

External (3pty) libraries can be used but it's a good advice not to overdo it. This means, that if there is a low effort to write the code for a functionality, then do it instead of acquiring and installing an external library.

3pty libraries **should** be imported by Maven. Only if there is no Maven repository for the required library, a manual download into the directory `./lib` is permitted and the library must be manually added to the build path. However, this exception **must** be documented in the extension's documentation.

1.2 Default configuration

The `./cfg` directory contains a default Plugin.properties file which defines the built-in extension modules. It **must** be left unchanged when building and distributing extensions. Instead, provide the extension configuration in a file named `<extension_name>.properties`.

It's also always welcome to add a readme file containing the code needed to patch the Fskencoder.bat file. That's in particular the extension of the class path for the extension itself and where applicable the class path of the used libraries.

1.3 Building

Apache ANT is used for building.

For each extension, a buildfile **must** be provided in the directory `./build`. This buildfile is responsible for the build of only one specific extension and **should** be named after the extension e.g. `BinReaderExtension.xml` for the *BinReaderExtension*.

A second level buildfile `BuildExtensions.xml` in the projects root directory builds **all** extensions by iterating over the files in the `./build` directory.

Commonly used properties and macros are put into the file `ExtensionsBuildSupport.xml`.

2 Architectural overview

The FskEncoder follows the linear MVC model where only two interaction paths are possible:

1. between View and Control and vice versa,
2. between Control and Model and vice versa.

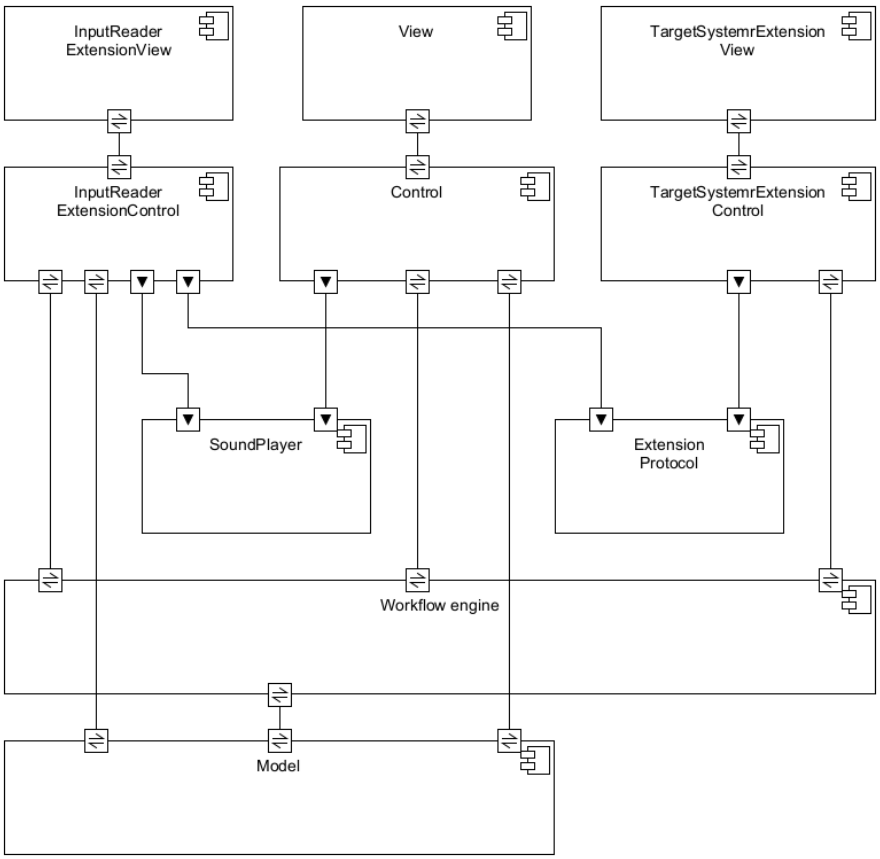


Figure 2.1: Architecture overview

The right-hand side shows the main window with two regions marked with a red frame.
The top-one, with the caption *Region* is the extension GUI of the input reader extension while the lower one (with the field *File Name*) is from the target system extension.

These GUI parts are specific to each extension and are part of the extension packages.

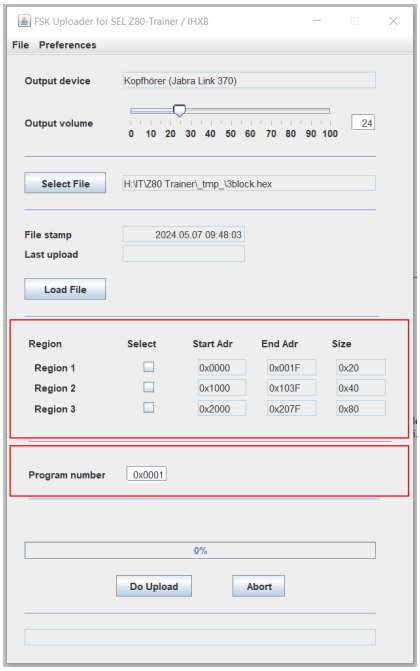


Figure 2.2: Main widow

Figure 2.1, *Architecture overview*, on page 5 depicts the global structure of the application together with the extensions:

- **View:** This is the main window with all GUI elements exept that of the GUI extensions.
- **Control:** This is the control for the main window and deals with every thing exept the extension GUIs.
- **Workflow engine:** Encapsulates the *business logic* of the FskEncoder application and the extensions.
- **Model:** The data model of the application. It holds some run-time data and the properties of the two configuration files *FskEncoder.properties* and *Plugin.properties* .
- **Soundplayer:** Plays the sound samples on the selected output device with the desired volume.
- **InputReader Extension view:** This is the GUI part of the selected input reader extension and **must** provide all informations needed to deal with the input format.
  The example in figure 2.2 on top of this page shows the GUI of the Ihx8InputReaderExtension which can handle more than one memory section in a single file.

- **InputReader Extension control:** Extension specific control which must handle all controls of the extension gui.
- **TargetSystem Extension view** This is the GUI part of the selected target system extension and **must** provide all informations needed to deal with the specific protocol format.
- **TargetSystem Extension control** Extension specific control which must handle all controls of the extension gui.
- **TargetSystem Extension protocol:** The protocol used to encode the binary data and to convert them into sound samples.

Each input reader and target system extension must be fully decoupled from the FskEncoder source (except the interface / base classes needed) and **shall** only interact with the application via the provided interfaces.

A deep dive into how to build those extensions is discussed in the section 3.2, *Extension instantiation*, on page 13.

By rule, the application itself and each extension lives in it's own .jar file which **must** contain all the necessary classes but no 3pty libraries (please refer to section 1 *Project setup* on page 4 for library management).

2.1 Project structure

This section gives an explanation of the different packages and their purposes. The classes are only discussed if there is a real need for the understanding of building extensions. Most of this information can be found in the JavaDoc of the project.

For each package is specified if its classes and interfaces should not, can or must be used in extensions.

2.2 Application part

The following source folder / packages and their contents currently exist:

```
.
+---src (folder)
     +---application
     +---control
     |   +---gui
     |   +---validator
     +---extension
     |   +---control
     |   +---encoder
     |   +---factory
     |   +---model
     |   +---protocol
     |   +---sond
     |   +---source
     |   +---target
     |   +---view
     +---model
     +---protocol
     +---sound
     +---view
         +---gui
```

2.3 Packages of the 'src' folder

2.3.1 application

Don't use it for extensions.

Contains the application class together with classes to handle cli parameter, some 'hard coded' properties and for program exit.

2.3.2 control

Don't use it for extensions.

Contains the controller logic for work-flow and background task execution.

2.3.3 control.gui

Don't use it for extensions.

Held all controllers for the basic GUI panels.

2.3.4 control.validator

Don't use it for extensions.

Contains various validators for the GUI elements.

2.3.5 extension

Can / must be used for extensions.

This package provides the logic needed for extending the application. A deeper discussion can be found in section 3, *API*, on page 12.
The content of this package is packed in a seperate file *FskEncoderExtension.jar* during build process and **must** be used as library in the FskEncoder-Extensions part.

2.3.6 extension.control

Can / must be used for extensions.

Contains classes and interfaces needed for extension controllers.

2.3.7 extension.encoder

Can / must be used for extensions.

This is more a library of basic functionallity and contains all (currently) necessary classes for sound sample generation.

Protocol specific encoders **must** be placed in the package of the target system extension protocol.

Please refer to appendix A, *Protocol implementation*, on page 21 for more details.

2.3.8 extension.execution

Can / must be used for extensions.

Contains classes and interfaces needed for background task extension of target system extension protocols.

2.3.9 extension.factory

Can / must be used for extensions.

This package is the place for the base classes and interfaces needed to build a extension factory.

2.3.10 extension.model

Can / must be used for extensions.

This package contains all classes and interfaces of the data model needed to implement a input reader extension.

2.3.11 extension.protocol

Can / must be used for extensions.

This package contains all classes and interfaces needed to implement a target system extension protocol.

2.3.12 extension.sound

Some classes can / must be used for extensions.

This package contains the definition of the audio format of the FskEncoder soundplayer and is required in an extension protocol.

2.3.13 extension.source

Can / must be used for extensions.

This package contains all classes and interfaces needed to implement a input reader extension.

2.3.14 extension.target

Can / must be used for extensions.

This package is currently empty.

2.3.15 extension.view.gui

Can / must be used for extensions.

Holds all base classes for extension guis.

2.3.16 model

Don't use it for extensions.

This package contains all classes of the internal data model.

2.3.17 protocol

Can / must be used for extensions.

This package is currently empty.

2.3.18 sound

Don't use it for extensions.

All stuff around the sound player.

2.3.19 view

Currently empty.

2.3.20 view.gui

Don't use it for extensions.

Contains all classes needed for the main window and the basic GUI panels.

2.4 Extensions part

In the extension part are all packages located, which are related to extensions.
They **must** be strictly kept apart of the application part!
The following source folders / packages and their contents currently exist:

```
.
+---src (folder)
    +---source
    |   +---bin
    |   +---ihx
    |       +---x8
    +---target
        +---microprofessor1
        +---z80trainer
```

2.4.1 source.bin

Don't use it for extensions.

Contains all sources for the binary input file reader.

2.4.2 source.ihx

Some classes can / must be used for extensions.

Contains all common used sources for the Intel-Hex input file reader.

2.4.3 source.x8

Don't use it for extensions.

Contains the 8-bit specific implementation of the Ihx-reader extension.

2.4.4 target.microprofessor1

Don't use it for extensions.

Contains the classes of the Microprofessor 1 extension.

2.4.5 target.z80trainer

Don't use it for extensions.

Contains the classes of the Z80 trainer extension.

3 API

A input reader extension is at least based on four classes:

1. a ReaderExtension,
2. a ReaderGui
3. a ReaderControl and
4. a Reader.

By convention, this classes **should** be named like

1. `<extension_name>`ReaderExtension,
2. `<extension_name>`ReaderGui
3. `<extension_name>`ReaderControl and
4. `<extension_name>`Reader,

with `<extension_name>` as descriptive name which reflects the format of the input file like *Bin* for binary files or *Ihx8* for Intel-Hex format.

A target system extension requires at least the four classes

1. Extension,
2. ExtensionGui
3. ExtensionControl and
4. Protocol.

The naming conventions follow the rules of the input reader and **should** be like

1. `<extension_name>`Extension,
2. `<extension_name>`ExtensionGui
3. `<extension_name>`ExtensionControl and
4. `<extension_name>`Protocol,

To build a apropriate memeory model for an extension, a input file reader **should** use the classes of the `extension.model` and `extension.source` packages.

3.1 Functional interface

The communication between the main application and an extension is and **must** be kept restricted to predefined methods to keep the dependencies between the application and the extension as low as possible. The methods used are discussed later in the deeper explanation of the differen parts of the extensions.
Figure 3.1, *Sequence diagram of user interaction*, on page 14 depicts how the application and the different parts of an extension work together.

3.2 Extension instantiation

Extensions are created in extension factories which are part of the extension itself. Theese extension factories are instantiated by the PluginFactory and the extension factory is responsible for creating the required model, controller and protocol classes.
Figure 3.2 on page 15 shows the sequence diagram of the creation of an InputReaderExtension and TargetsystemExtension.

3.3 Target system extension

A target system extension provides all logic and functionality needed to translate the (binary) source data from the source buffer into the sound samples which can be played on the sound card of the host computer.

It consists at least of:

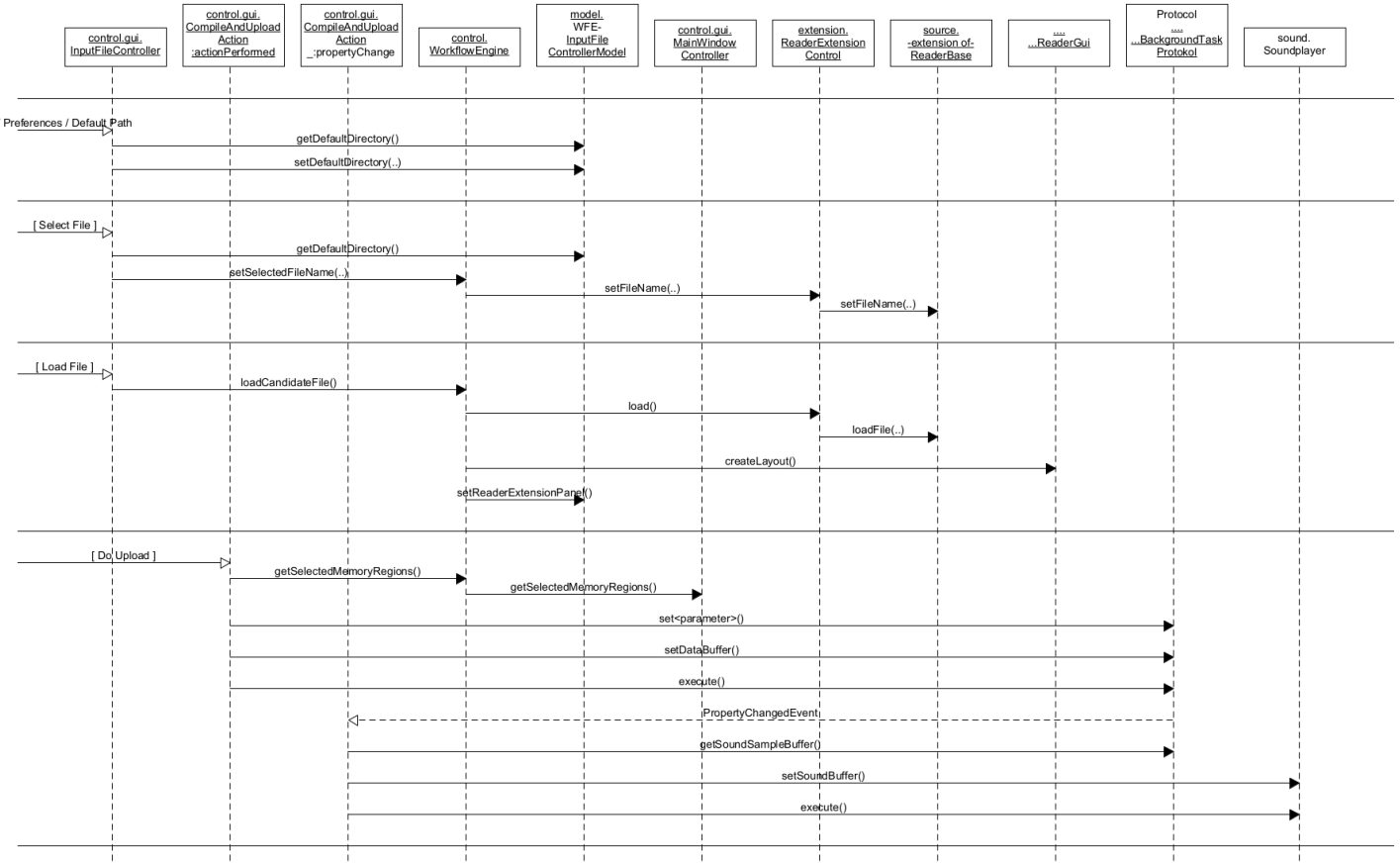


Figure 3.1: Sequence diagram of user interaction

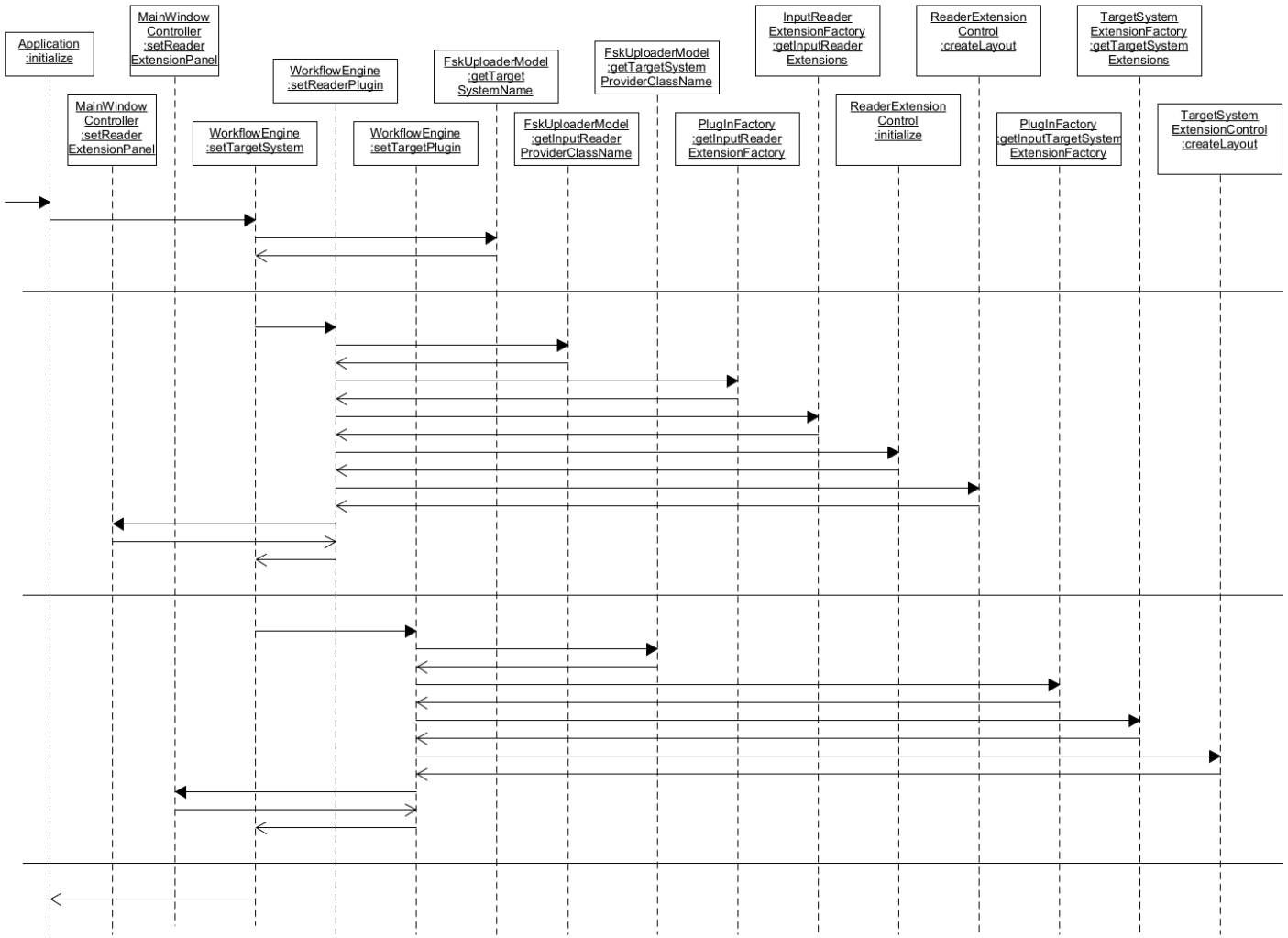


Figure 3.2: Example Sequence diagram of extension load process

1. A extension factory, which is the implementation of the interface `TargetSystemExtensionFactory` and will be discussed in more detail in the section *TargetSystemExtension* below.

2. The extension GUI, which **must** extend the abstract `ExtensionGui` base class and **must** be implemented using Java Swing. It's recommend to use the *miglayout* layout manager as the main window does.

3. The controller logic (MVC model), which is derived from the abstract `TargetSystemExtensionControl`.

4. The so called 'protocol', which is capable for the translation process. It is an extension of the abstract `BackgroundTaskProtokol` base class and **should** use the various encoder classes of the `extension.encoder` package for encoding sound samples.

Figure 3.3 on page 17 shows the class diagram of the Z80-Trainer extension as an example of how an target system extension is to build.

3.3.1 TargetSystemExtension

This factory class is responsible for the creation of the extensions MVC pattern and **must** implement the `TargetSystemExtensionFactory` interface. It must create an instance of the ExtensionControl and a `TargetSystemExtensionDao` which is later on registered in the application.

Because this class is registered in the *Plugin.properties* it **should** be named like

`<system_name>Extension` what results in a .jar file named

`<system_name>Extension.jar`.

`<system_name>` is a descriptive name like *Z80Trainer* , *Mpf1* or anything else.

3.3.2 TargetSystemExtensionGui

Some transmission protocols need additional information which can't be obtained from the input file. This is e.g. the *Program Number* of the Z80-Trainer. The TargetSystemExtensionGui is the place where to put those input fields. As said before, it **must** be derived from the abstract class `ExtensionGui` and is instantiated later on in the assocciated controller class.

3.3.3 TargetSystemExtensionControl

The TargetSystemExtensionControl must extend the abstract class `TargetSystemExtensionControl`.

It **should** at least

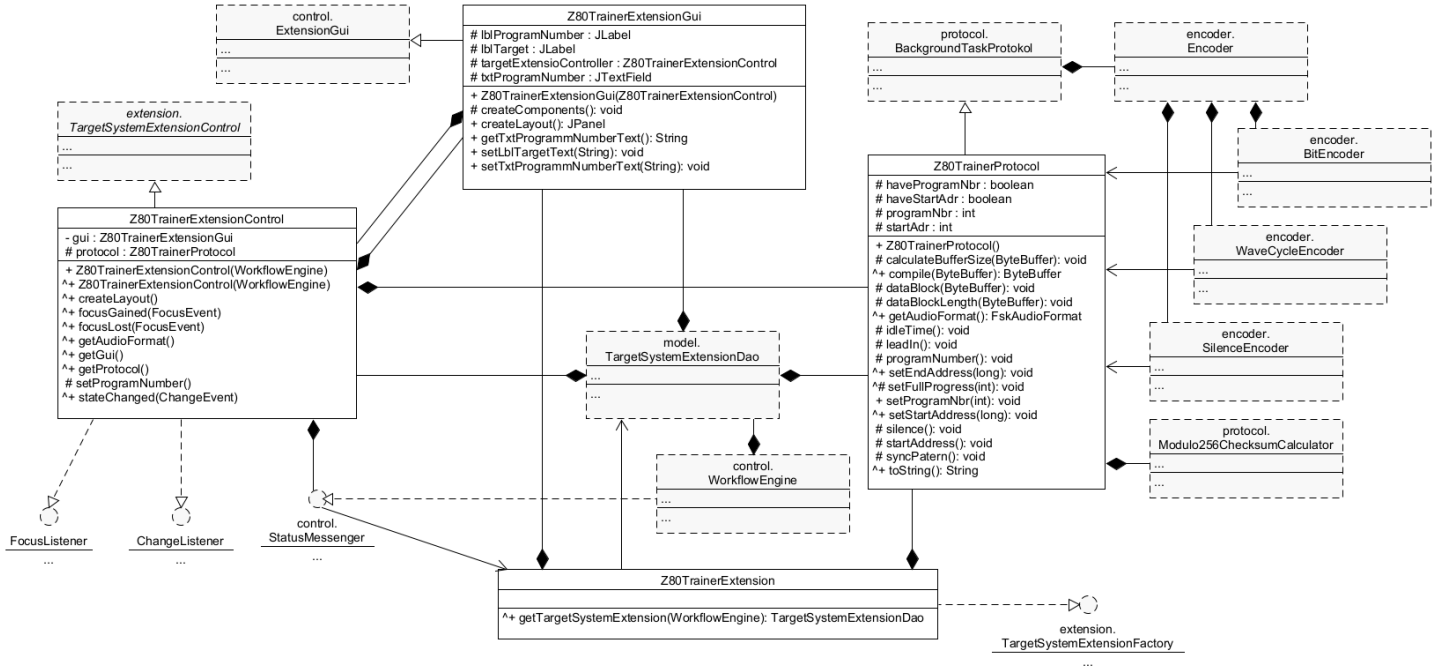


Figure 3.3: Example Z80TrainerExtension class diagram

1. create an instance of the protocol implementaion,
2. create an instance of the related GUI and pass a referenze to itself,
3. initialize the GUI with appropriate default values,
4. implement the methods of the base class (which are inherited from Java Swing ActionListener) and
5. provide the handle methods for the GUI elements by implementing required Java Swing interfaces (like FocusListener, ChangeListener, ...).

3.3.4 TargetSystemProtocol

The TargetSystemProtocol is derived from the abstract base class `BackgroundTaskProtokol` and **must** implement the two methods `compile(...)` and `setFullProgress(int)(...)`.
It **may** override the `setStartAddress(...)`, `setEndAddress(...)` or `setSize(...)` methods with its own behavior.
Please go to section A, *Protocol implementation*, on page 21 to learn how to implement a protocol.

3.4 Input reader

A input reader provides all functionality needed to read a data file in a specific format and to convert the data read into plain binary bytes. It builds a data buffer which later on is handed over to the protocol part of the target system extension.

It consists at least of:

1. A extension factory which is in fact the implementation of the `InputReaderExtensionFactory` interface.
2. The Java Swing extension GUI which must extend the abstract `MemoryMapGui` base class.
3. The controller logic (MVC model) which is derived from the abstract `ReaderExtensionControl` and must implement the two Java Swing interfaces `ChangeListener` and `FocusListener`.
4. The so called 'reader', which is capable for the translation process. It is an extension of the abstract `ReaderBase` base class.
5. An extension of the abstract `MemoryRegionBuilder` class which provides the interface to the protocol implementation.
6. Optionally an extension of the abstract base class `Record` if the input file format is record structured.

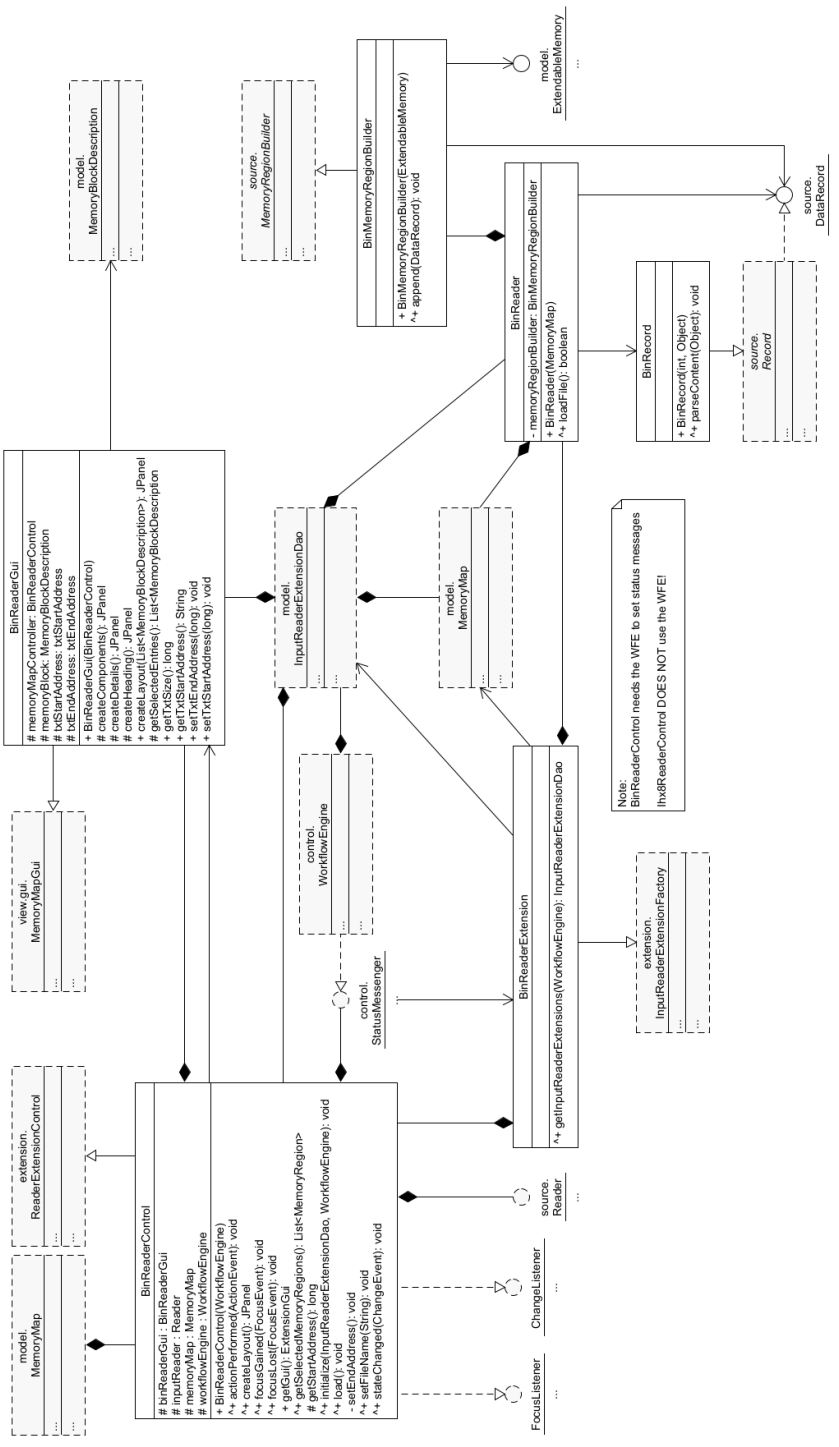


Figure 3.4: Example BinReaderExtension class diagram

Page 19 shows the class diagram of the binary input file reader extension as an example of how an input reader extension is to build .

3.4.1 ReaderExtension

The ReaderExtension class is a factory which creates

- a instance of the `MemoryMap` class,
- a instance of the `Reader` class,
- a instance of the `ReaderControl` class and
- a instance of the `InputReaderExtensionDao`, which held references to the previous three classes.

The `InputReaderExtensionDao` is returned to the caller `PluginFactory` and further to `WorkflowEngine`.

This class is registered in the Extension.properties and should be named according to the rules previously stated in section 3.3.1, *TargetSystemExtension*, on page 16.

3.4.2 MemoryMapGui

This base class provides a filed which holds a list of `MemoryBlockDescription` objects. Each object of this list describes a memory block of the source file with the three attributes *START_ADDRESS* , *END_ADDRESS* and *SIZE*. These attributes are required to set up the reader extension GUI.

3.4.3 ReaderExtensionControl

The ReaderExtensionControl must extend the abstract class `ReaderExtensionControl`.

It **should** at least

1. create an instance of the reader implementaion,
2. create an instance of the related GUI and pass a referenze to itself,
3. initialize the GUI with appropriate default values,
4. implement the methods of the base class (some inherited from Java Swing ActionListener),
5. provide the handle methods for the GUI elements by implementing required Java Swing interfaces (like FocusListener, ChangeListener, ...) and
6. hold an instance of the `MemoryMap` object.

3.4.4 Reader

The Reader implementation **must** be derived from the abstract `ReaderBase` class and **must** implement the inherited methods.

It's responsible for reading and parsing the input file and storing the file content in a `MemoryRegion` object via a `MemoryRegionBuilder` instance.

3.4.5 MemoryRegionBuilder

This is the storage container for the data red by the reader implementation. It extends the abstract `MemoryRegionBuilder` and implements it's abstract methods.

3.4.6 MemoryRegion

A MemoryRegion represents a bunch of binary data together with it's start address and it's size.

3.4.7 MemoryMap

The MemoryMap holds a map of `MemoryRegion` objects. It is responsible for creation of `MemoryBlockDescription` objects used by the GUI and for delivery a specific `MemoryRegion` on request.

3.4.8 Record

The input reader philosophy is based on the *Record* structure because every input file can be transformed into records. A binary file contains only one record where on the other hand a Intel-Hex formatted file is strictly based on the record structure. To simplify the reading / parsing process and the data handling, it is strongly adviced to rely on this philosophy.

A Protocol implementation

How to implement an protocol?

Lets study this on the Z80-Trainer protocol which is very simple to implement.

First, we have to know the encoding rules which are defined by the systems vendor. For this example they are as:

```
SEL Z80-Trainer tape format
```

```
Bit format
-------------------------------------------------------------------------------

'0'   1 cycle   568,18Hz (1760µs)
'1'   1 cycle  1136,36Hz ( 880µs)


Envelope (Byte format)
-------------------------------------------------------------------------------

  1 start bit '0'
  8 data bits, lsb first (b0 to b7)
  3 stop bit '1'


File format
-------------------------------------------------------------------------------

 1.    12'288   bit '1'   Lead sync                        ->leadIn

 2.         1   bit '0'   Measurement                     ->syncPatern
 3.        16   bit '1'     for period length             +>

 4.         1   envlp     Programm number - high byte     ->programNumber
 5.         1   envlp     Programm number - low byte      +>

 6.         1   envlp     Start address - high byte       ->startAddress
 7.         1   envlp     Start address - low byte        +>
 8.         1   envlp     Start address checksum          +>

 6.         1   envlp     Data block length - high byte   ->dataBlockLength
 7.         1   envlp     Data block length - low byte    +>
 8.         1   envlp     Data block length checksum      +>

 9.        16   bit '1'   Idle time for chksum calculation  ->idleTime

10.         n   envlp     Data block                      ->dataBlock

11.         1   envlp     Data block checksum             +>
```

`Bit format` We can see, that there is a definition of how to encode a single bit. In this case it is one single wave cycle with a spcific frequency.

`Envelope` Next is the definition of an *envelope*, which consists of one start

bit, 8 databits in a specific order and one stop bit.

`File format` The previously defined bit- and envelop formats are put together to build a transmission protocol. In this example, there are 11 different sections defined as shown above.

Next, we take a closer look to the `extension.encoder` package, its classes and usage.

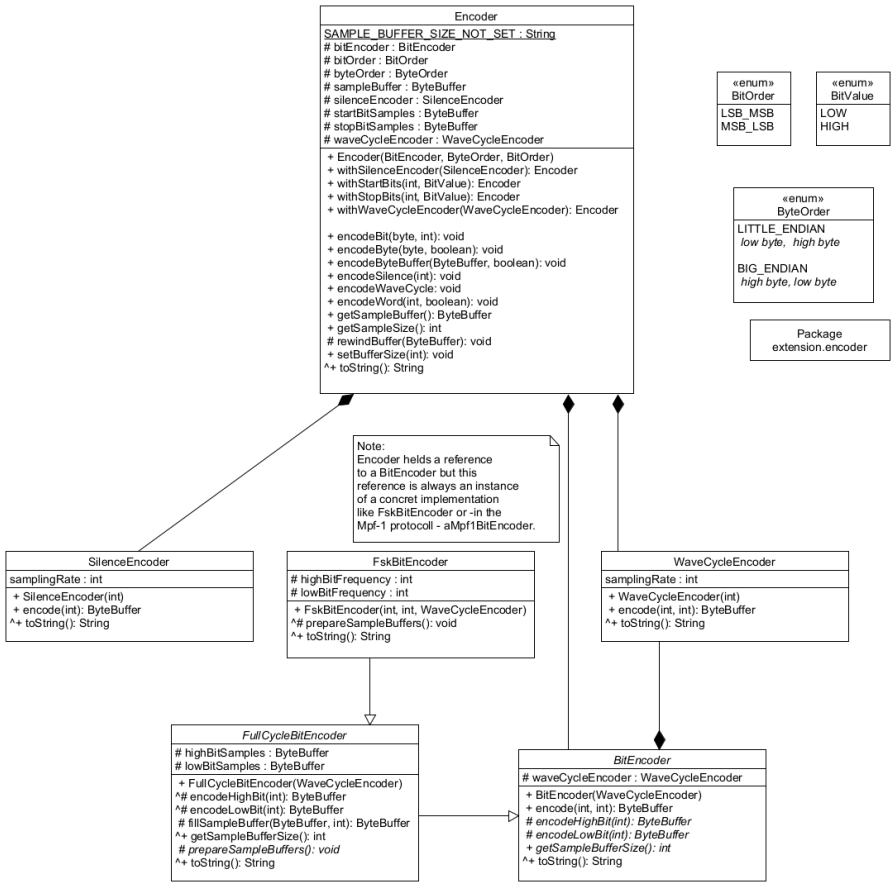


Figure A.1: Package extension.encoder class diagram

Let's walk trouh it from bottom up to the top and start with the abstract class `BitEncoder`.

Because the encoding strategy preferes copying a couple of bytes over calculating them over and over, we use tiny sound sample buffers - so called *atoms* - one each for a single '0' and '1' bit.

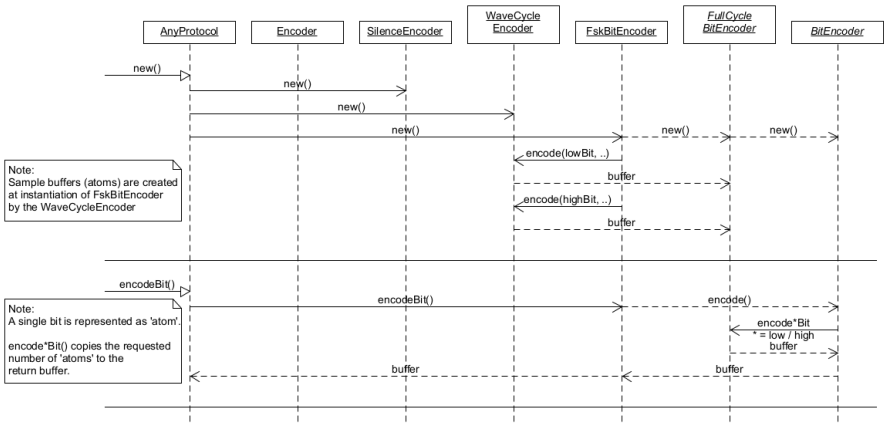


Figure A.2: Sequence diagram of bit encoding

The concrete implementation of the abstract `BitEncoder` must be able to produce these atoms, according to the protocol specification. In our example, this is an easy job and is done by the `FskBitEncoder`.

The `FskBitEncoder` has no big logic. It's purpose is to define the FSK frequencies, build the *atoms* and to provide a class with a suitable and self-explantory name.

It's abstract base class `FullCycleBitEncoder` has a abstract method `prepareSampleBuffers()` which is implemented in `FskBitEncoder`. By convention, this method **must** be implemented in the derived class and **must** be calld in it's constructor! It uses an instance of `WaveCycleEncoder` (held by super class `BitEncoder`) to build the sound samples for the atoms.

These atoms are held in the super class `FullCycleBitEncoder` because of the fact that the `encodeHigBit()` and `encodeLowBit()` methods are located in this class (as they are common to all bit encoders).

Beside the bit encoding exists the `SilenceEncoder` who is responsible to create a buffer filled with '0' bytes if a protocoll needs to send silence for a defined time.

The class `Encoder` is used by a protocol implementation to encode all defined parts and sections. In this example that are the numbers 1 to 11 in the *File format* section. For this, it is set up with instances of the required encoders at instantiation (usually in the specific implementation of the protocol).

Three enum classes complete the `encoder` package and define common used terms for typesafety:

`BitValue` defines simply the two possible states of a bit.

`BitOrder` defines which bit of a byte is encoded first.

`ByteOrder` defines the *endianess* of words.

Please feel free and take a look to the two protocols Z80-Trainer and Mpf1, which can be found in the sources of that extensions.

The Mpf1 protocol uses a specific bit encoder and is a good starting point for the implementation of a more complex encoding.

B Nomenclature

B.1 Keywords

The key words [1] "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as:

1. MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.
2. MUST NOT This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.
3. SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
4. SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
5. MAY This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to inter-operate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to inter-operate with another implementation which does not include the option (except, of course, for the feature the option provides.)

B.2 UML grapics

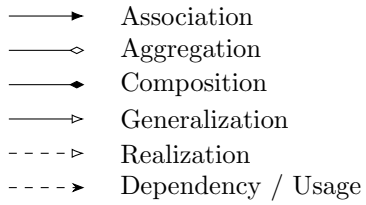
This document contains some UML class diagrams. In this diagrams are two styles used to draw the classes:

- First the standard style, white background with attributes and methods defined for all classes / interfaces in the package of interest and
- second the greyed style, grey background and dashed lines, without definition of attributes and methods for all collabrating classes / interfaces of different packages.

[1] Taken from IETF RFC 2119, Key words for use in RFCs to Indicate Requirement Levels

The greyed classes / interfaces are included for completeness and to lay out the dependencies between packages.

Arrows in the relationship of classes and interfaces and their meanign :



If two classes A and B have a relationship, than

association between A and B means that A hold a reference to B so that A can communicate with B,

aggregation means, that A holds one or more refeerences to B,

composition means, that B is part of A in that way, that B can't exist without A.

generalization is, when B extends A

realization is, when A implements the interface B

dependency means, that A uses B but without keeping a reference to B.