

Manual for developers

This document describes how to start developing for Defraser.

Defraser is a forensic desktop application for detecting, analyzing and repairing video data. It consists of an application and detector plug-ins. These detectors contain the knowledge of the supported video and container file formats. Defraser is distributed with detectors for common video and container formats. If you want to handle another format, you can write your own detector and copy the new detector plug-in to the proper location. Defraser shall now automatically load the new plug-in at startup and thus support the new format. You don't need to change the application itself!

Detector development does not require purchase of any license. To change Defraser itself, you probably want to buy a licensed version of Visual Studio and a development license of Infralution's Virtual Tree. This is explained in chapter 4.

Table of contents

1	Development environment requirements.....	3
2	Setting up the development environment	4
3	Detector development.....	5
3.1	Setup a new detector	5
3.2	How a plug-in works.....	6
3.2.1	Headers	6
3.2.2	Header Content	6
3.2.3	Files	7
3.2.4	Data block.....	7
3.2.5	Codec streams	7
3.3	Defraser userinterface	7
3.4	Defraser under the hood.....	11
4	Application development.....	13

1 Development environment requirements

The source of Defraser is distributed with all DLL's required to build and run the application. This enables you to build it right away. To optionally download external libraries and documentation, follow the steps below. As stated, these steps are not necessary. Follow them if you need the documentation, to use newer versions of an external library or to run the unit tests.

Software required for Defraser development:

- The latest version of the Defraser source files.
- Microsoft Visual Studio C# 2008 Express Edition. This is Microsoft's free development tool for C# and suitable for developing most of the application, including detectors. Only the installer cannot be build by the Express Edition. To build the Defraser installer, you need to obtain a licensed version of Visual Studio.
- .NET 3.5 framework

Optional:

- The unit test framework NUnit version 2.4.8 to run the accompanying unit tests of Defraser.
- The mock test framework Rhino Mocks 3.5 - for .NET 3.5.
- Inversion of Control container Autofac version 1.4.4.561.
- log4net version 1.2.10 or newer. This component is used for the logging functionality of Defraser.
- Infralution VirtualTree 3.14.0. This is a commercially available tree component used in the user interface. This manual describes how to install a trial version. If you don't have a license, you are able to build Defraser for 30 days but a message dialog will pop up every time you start the application. This is not a problem if you only do detector development, since new detectors can be added to the application without rebuilding it. If you are going to extend or change the behavior of the application itself, you should buy a license.
- The zip-utility for building the source-zip. This utility is only needed if you want to create the source-package for distribution of the Defraser source files.

2 Setting up the development environment

The following section provides a step-by-step description of setting up the development environment.

Required download and installation steps:

1. Download and unpack the latest version of the [Defraser source zipfile](#).
2. Download and install [Microsoft Visual Studio C# 2008 Express Edition](#).
3. Download and install [Microsoft .NET Framework 3.5](#).

Optional download and installation steps:

1. Download and install [NUnit 2.4.8 for .NET 2.0](#).
2. Download and install [Rhino Mocks 3.5 - For .NET 3.5](#).
3. Download and install [autofac version 1.4.4.561](#).
4. Download and install the file [incubating-log4net-1.2.10.zip](#). See also <http://logging.apache.org/log4net/>.
5. Download and install [Infralution VirtualTree 3.14.0](#).
6. Download [zip232xN.zip](#) or newer. Unzip this file to an execution folder (e.g. to C:\Program Files\zip) and add this folder to the PATH system environment variable, located in the Control Panel under 'Performance and Maintenance' - 'System'. Click on the 'Advanced' tab and then select 'Environment Variables'.

Build steps:

1. To start developing, double-click the solution file Defraser.sln. This will start Visual Studio. You might be informed that the solution was created with a previous version of Visual Studio and that it will be converted. That's OK. You will notice that all project files of the solution will be converted to the current version of Visual Studio.
2. Right-click "GuiCe" in the solution explorer and select "Set as StartUp Project" to make it the startup project.
3. One project of the solution is the "Installer" project (.vdproj). This setup project is not supported by the Express Edition of Visual Studio. You can safely remove this project from the solution. To build the Installer, you need to obtain a licensed version of Visual Studio.
4. To run the unit tests: build Defraser and start NUnit. Open the file 'TestDefraser.dll' in NUnit by selecting 'File' - 'Open Project' from the main menu. You can find this DLL at: ...\\Test\\bin\\Debug or ...\\Test\\bin\\Release in the folder where you unpacked the sources. The selection of the debug or release path depends on your build configuration. Note that you need to download the optional NUnit application for the last step.

3 Detector development

3.1 Set up a new detector

At this point you have set up your development environment and you will be able to extend or change all parts of the application. However, you probably just want to add support for a new format by writing a new detector plug-in.

Defraser consists of an application and detectors, where the detectors contain the knowledge of the supported video formats. These detectors are modeled as plug-ins, making it easy to extend Defraser with support for new video or container formats. When there is no detector for a format, you can decide to write one yourself. Most likely you do not need to touch the application code itself. You can use the existing detectors as examples to create you own.

In the following description you will write your own detector using an example detector as a template and install the detector (a .DLL file) by copying it to the Defraser execution folder. The following description assumes that you have set up the development environment as described above and have opened the Defraser solution file with Visual Studio.

- First, make a copy of the folder ExampleDetector and name it after the detector you want to create. In the following description we use '*MyDetector*'.
- Now - in the Visual Studio solution - you have to create a new project for your new detector. To do so select Add Existing Project... from the context menu of the solution. Browse to the folder you copied the ExampleDetector to (in our example *MyDetector*) and select ExampleDetector.csproj. The new project (called ExampleDetector) will be added to the solution. You might be informed that the project will be converted to a new version of Visual Studio.
- Rename the project using the Rename option from the context menu of the project (in our example rename it to *MyDetector*).
- Open the properties of the new project and change the Assembly name and Default namespace to the detector's name (*MyDetector*). Then click the button Assembly Information... and make the appropriate changes. Set the Assembly Version and File Version to 0.0.1.0! Save the project properties.
- The new project contains three source files: ExampleDetector.cs, ExampleHeader.cs and ExampleParser.cs. Rename these files to the proper name (*MyDetector.cs*, *MyHeader.cs* and *MyParser.cs*). Use the Rename option in the context menu's of these files.
- Open the Detector source file *MyDetector.cs*. Use the Refactor::Rename... option of Visual Studio to rename the following types: namespace ExampleDetector: rename to *MyDetector*.
- class ExampleDetector: also rename to *MyDetector*.
- class ExampleHeader: rename to *MyHeader*.
- class ExampleParser: rename to *MyParser*.
- enum ExampleHeaderName: rename to *MyHeaderName*.
- method Parser.FindStartOfExampleHeader: rename to *Parser.FindStartOfMyHeader*.

- Now replace every other occurrence of the text 'Example' with the appropriate text.
- Open the ExampleHeader.cs and ExampleParser.cs source files and make the appropriate changes in the same way as described above.
- Save all three changed files.
- Select Build from the project context menu. The detector shall build with no warnings and shall appear in the file system as a .dll file in the ...\\bin\\release\\ folder of the project (in our example\\MyDetector\\bin\\Release).
- To add the detector to Defraser copy the detector (MyDetector.dll) to the execution folder of your Defraser installation, default C:\\Program Files\\Defraser. When running Defraser from Visual Studio the execution folder will be "<solution folder>\\ GuiCe\\ bin\\ Release" or "<solution folder>\\GuiCe\\bin\\Debug" when debugging the application. The detector must be copied to these folders. For your convenience you can add these copy actions as a post build step in the properties of the detector project. To do so, select the Properties of the project, select the Build Events tab and add the following line to the Post-build event command line:
 - copy "\$(TargetPath)" "\$(SolutionDir)\\GuiCe\\bin\\\$(ConfigurationName)"
- Congratulations, you have setup the core of your new detector. Now you can start writing detection code.

3.2 How a plug-in works

3.2.1 Headers

A plug-in is created for a file format. The description of a file format can be found in its specification. For example the [ASF](#) and [Matroska](#) specifications. Such specification is used to create a plug-in. A format is built up from pieces of information. The names for those pieces can be different for each format. For example:

- Quicktime: atom
- AVI: chunk
- Matroska: element
- MPEG-1, MPEG-2, MPEG-4: header
- ASF: object

The word 'header' is used of those pieces in the rest of this document and in the Defraser framework. A header can be recognized by a unique byte sequence. The value of this unique byte sequence can be found in the format specification. The unique byte sequence can have a fixed length (for example 4 bytes (Quicktime, AVI) or 16 bytes (ASF)) or a varying length for each header (Matroska). Most formats have their headers structured like a tree. So a header can have a parent and one or more child headers. The unique byte sequence specifies the header type. The length of the header depends on the data it contains. Some formats have a field for the header length.

3.2.2 Header Content

Data contained in a header is parsed and stored for presentation. What data a header contains depends on its unique byte sequence and can be found in the format specification.

3.2.3 Files

Defraser lets the user select one or more files (disk images) and plug-ins. The purpose of each Defraser plug-in is to find as much headers of that format as possible in the files fed to it. Ideally without giving false hits. It should extract as much information from the files fed to it as possible, even when the data (videos and/or images) is partially overwritten.

3.2.4 Data block

Although the algorithm for finding headers is implemented in each detector it comes down to the same idea:

1. find a header that conforms to the format implemented by the plug-in;
2. try to connect a header that lies behind the header already found;
3. goto step 2.

You can find this code in method 'DetectData()' of each specific Detector class (AsfDetector.cs, AviDetector.cs, etc).

When no more headers can be joined with the headers already found, the headers found up till that point are gathered in one data block and the program starts looking for the next header (step 1).

3.2.5 Codec streams

Container formats are envelope formats that contain one or more codec format streams. A codec format consists of headers just like the container format does. Container formats are for example: 3GPP, ASF, AVI, Matroska, MPEG-2 System, QuickTime, etc. Codec formats are for example: H263, MPEG-2 Video and MPEG-4. Codec formats are important because they contain the actual payload data (video, sound, image, etc). The codec data is extracted from the container headers. How this is done, is specific to each container format. After the codec data is extracted, it is fed to codec detector(s) selected by the user.

3.3 *Defraser user interface*

This paragraph explains how the elements introduced in the previous paragraph are presented by the Defraser user interface.

The added red text shows which class or interface is used for an element. This makes it easier to know where to look in the code if you want to add or change something.

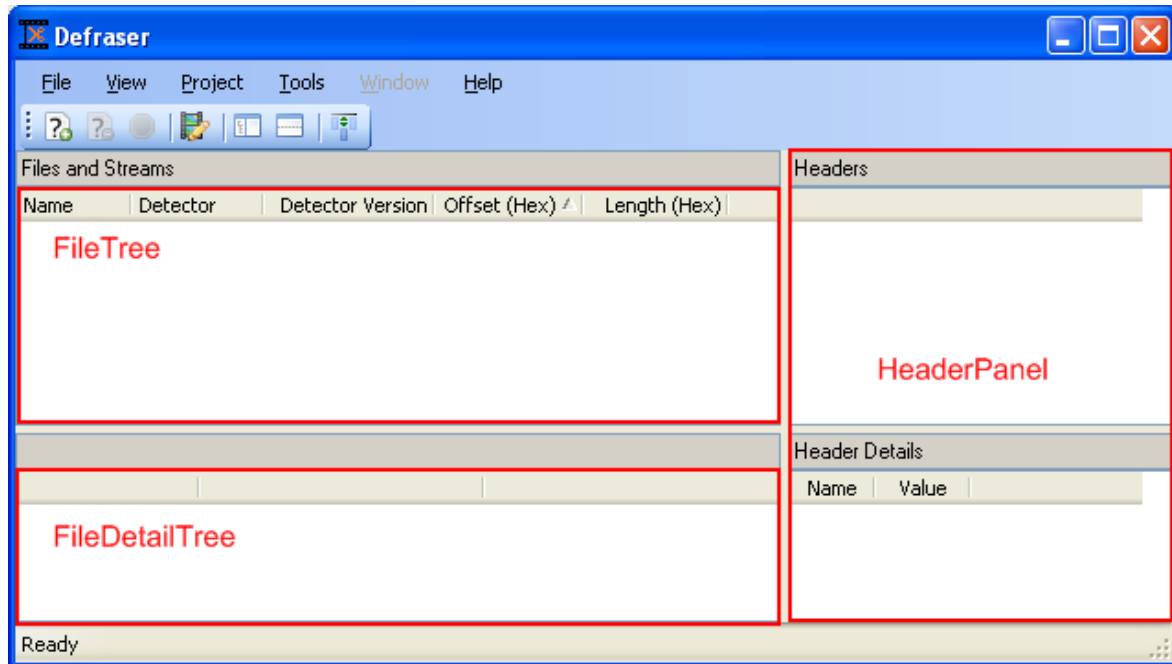


Figure 1: Screen shot of Defraser with three important controls annotated

Figure 1 shows the main window of Defraser. The files, data blocks and codec stream from the previous paragraph will be displayed in the FileTree. The headers will be displayed in the 'Headers' part of the HeaderPanel and the header content will be displayed in the 'Header Details' part of the HeaderPanel. FileDetailTree shows which detectors were used to scan a file. The FileTree, FileDetailTree and HeaderPanel are classes you can find in the GuiCe project.

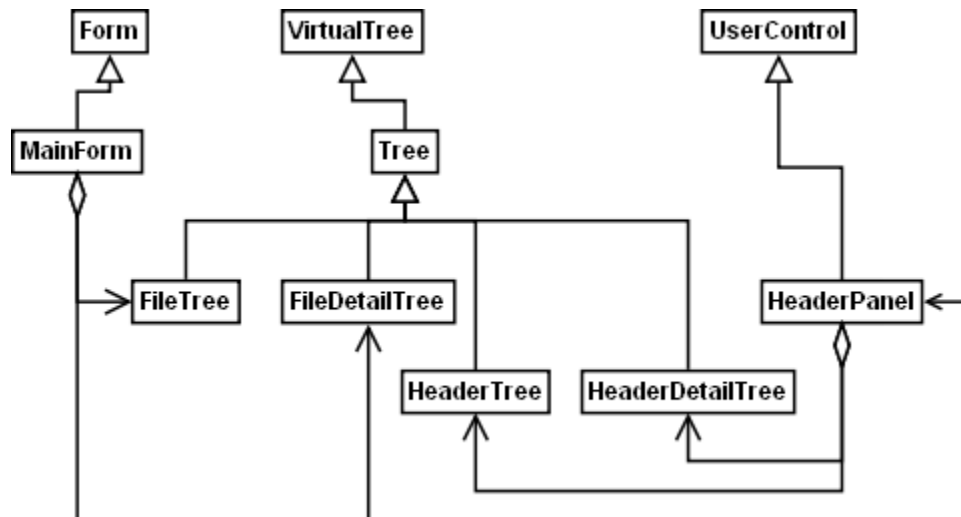


Figure 2: Class diagram of the controls from figure 1

Figure 2 shows the classes of figure 1 in relation to each other and other classes in the system.

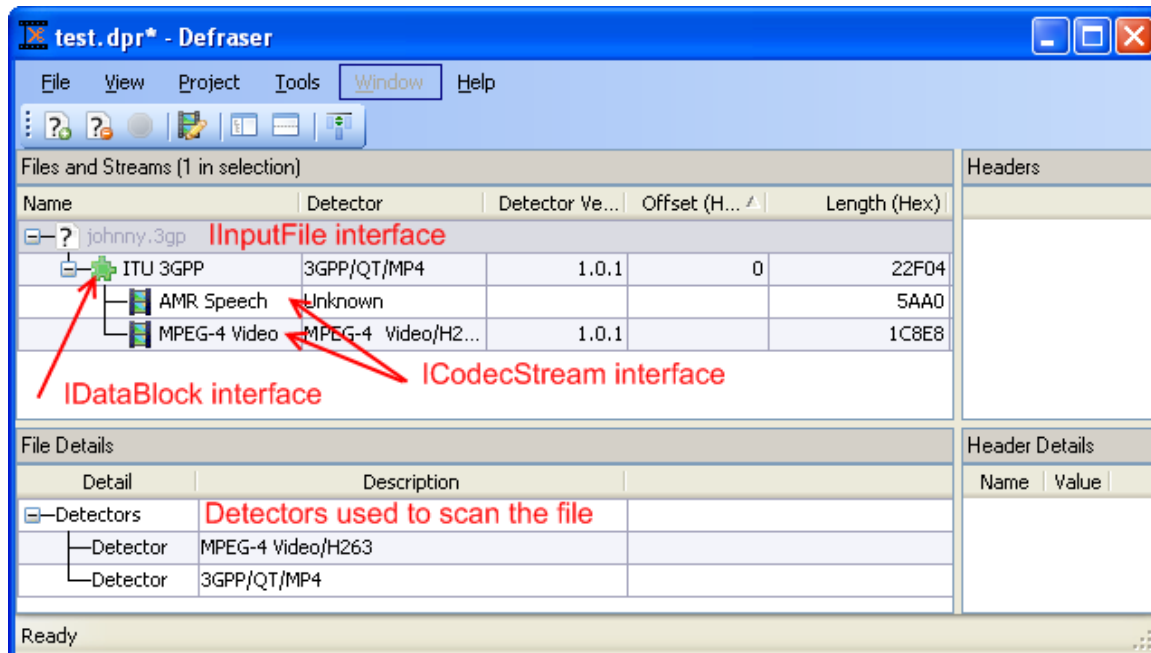


Figure 3: Screen shot of Defraser with the interfaces to display the information from annotated

Figure 3 shows the result of a file scan. The red text shows which interfaces are used to display the information from. The first node in the tree view shows the scanned file itself (IInputFile). The second node is the container format (3GPP) stream found (IDataBlock). The third and fourth nodes are codec format (AMR-speech and MPEG-4 Video) streams (ICodecStream). The content of the File Detail pane (below in the in the left corner) is only visible when the file node is selected.

Sometimes a codec format stream is wrapped by a container format for which there is no Defraser container plug-in. The codec format plug-ins are written in such a way that they do a best effort to get most of the codec headers from the file anyway. How successful depends on the codec format and the gaps between the codec headers caused by the container format. When codec headers are directly scanned (i.e. without the container detector extracting the codec data first), the result will be represented and displayed as a data block.

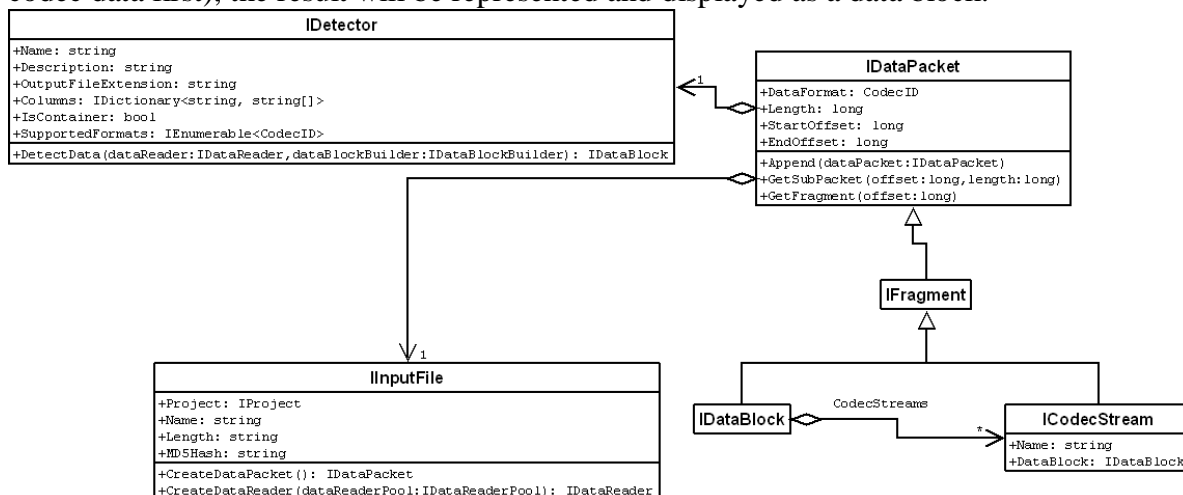


Figure 4: Class diagram of the interfaces shown in figure 3

Figure 4 shows the interfaces from figure 3 in relation to each other.

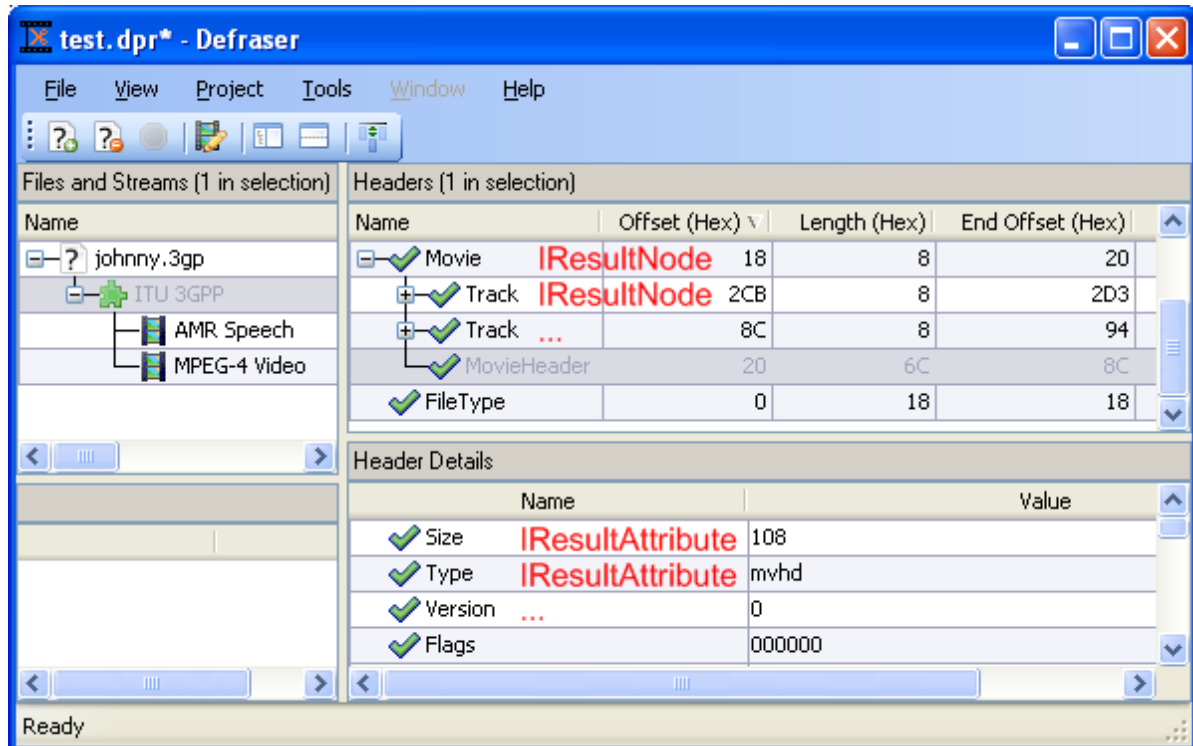


Figure 5: Defraser main window with interfaces used in 'Headers' and 'Header Details' view annotated

Figure 5 shows the headers you see when you select a data block or codec stream and the header details (is header content) when you select a header. Each node in the 'Headers' tree view shows the data of one *IResultNode* (implemented by a specific header) and its position in the tree. Each node in the 'Header Details' list view represents the data contained in one of the classes that implement the *IResultAttribute* interface. A row in the 'Header Detail' view can also be made visible in the 'Headers' view as column. This can be used to view that value for more headers at once and/or used to sort the headers on.

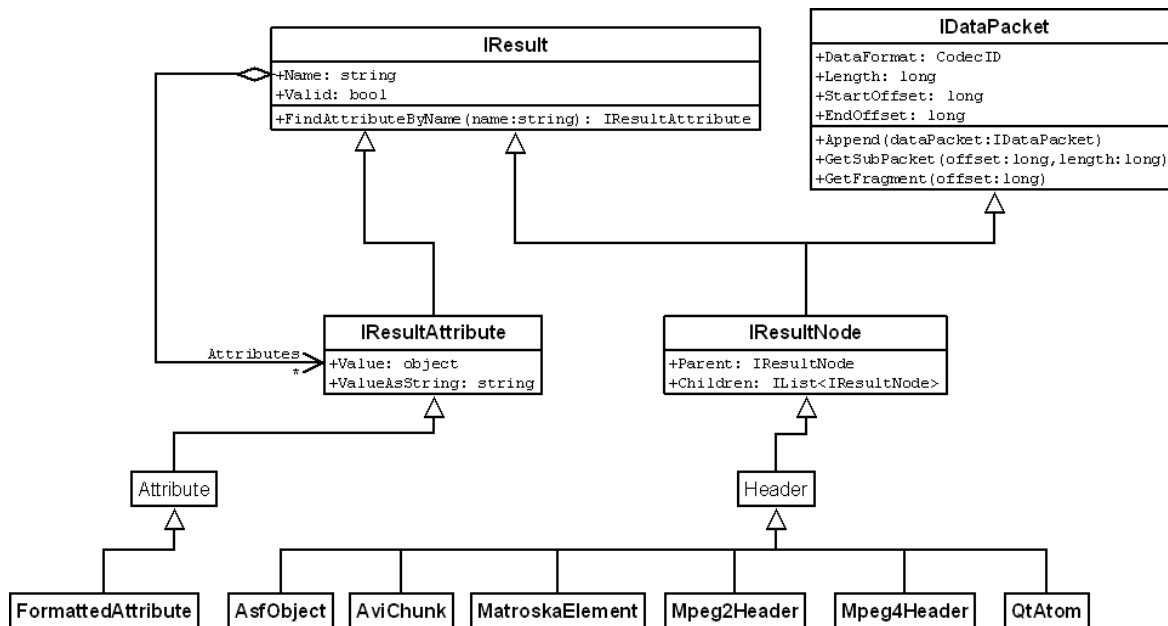


Figure 6: the interfaces of figure 5 in relation to the class diagram of figure 4

Figure 6 shows the interfaces from figure 5 in relation to each other.

3.4 Defraser under the hood

The classes derived from 'Header' in figure 6 (AsfObject, AviChunk, etc) are the base classes for all headers of that specific format. For each header found in the specification document a class will be created.

As described earlier, each header can be identified by a unique byte sequence. Each plug-in project has a file containing an enumeration that contains all header names. This file is called AsfHeaderName, AviHeaderName, Mpeg4HeaderName, etc. An attribute is added to each of the enumeration values. This attribute is used to add extra information like the unique byte sequence and possible parent(s). Next code snippet is from the AviChunkName.cs file (line numbers added):

```

1. [Avi("LIST", new AviChunkName[] { AviChunkName.Riff,
   AviChunkName.HeaderList } , ChunkFlags.ContainerChunk)]
2. HeaderList,
3. [Avi("avih", new AviChunkName[] { AviChunkName.HeaderList } ,
   ChunkFlags.SizeAndType)]
4. AviMainHeader,
  
```



The second and fourth line contain the enumeration value. The first and third line contain the attribute. The first value on the attribute line between the '()' characters is the unique byte sequence, the second value is a list of possible parents. The attributes added to the enumeration values are parsed during initialization of the plug-in and used during the scan process. The unique byte sequences from the attributes are collected in one list which will be referenced during the scan process. If subsequent bytes found in the stream are equal to one of the unique byte sequences in the list, the corresponding header will be created.

```
public HeaderList(AviChunk previousHeader)
: base(previousHeader, AviChunkName.HeaderList)
{ }
```

The code snippet above shows the constructor of a randomly chosen header (the AVI HeaderList chunk) which is called to create that header during the scan process. As you can see, the value of the enumeration (AviChunkName.HeaderList) is fed to the base class.

The next step is to parse the header content. If parsing fails there are two options:

- throw away the header;
- keep the header but mark it as invalid.

The kept headers that were marked as invalid can be recognized by a orange question mark: . Headers parsed without a problem are marked with a green tick: . See Figure 5.

Which option to choose depends on the certainty that the header is not correct and how important is to show a part of the header that might be correct. To just throw away the header the Parse method returns false. To keep the header but mark it as invalid, the parse method sets the Valid property to false and returns true.

4 Application development

If you want to change the Defraser application itself, you will need to obtain a license for the VirtualTree component that is used in the application. Compiling the application without this license will display a dialog every time you start the application, informing you that the application was build using a trial version of the VirtualTree.