



Ravioli Game Tools Plug-in Developer's Guide

Version: 1.1

Ravioli Game Tools version: 2.10

Author: Stefan Mayr



1. Contents

1. Contents	2
2. General	3
3. Plug-in basics	3
4. Creating a new plug-in.....	3
5. Installing and uninstalling a plug-in.....	4
6. Implementing support for a new archive file format.....	5
6.1. Archives with simple structures	5
6.1.1. All data uncompressed and continuous (GenericArchive)	6
6.1.2. Whole archive compressed in one block (GenericCompressedArchive)	8
6.1.3. Data part of archive compressed in one block (GenericCompressedArchive2).....	8
6.1.4. Files are compressed individually (CompressedItemsArchive)	9
6.2. Archives with more complex structures.....	11
6.3. Additional archive features	16
6.3.1. IDataWriter interface	16
6.3.2. IRootDirectory interface	16
6.3.3. IExtensionIndependent interface	16

2. General

This guide is intended for developers who wish to add support for new file formats to Ravioli Game Tools.

Ravioli Game Tools is designed to be extensible using plug-ins to add support for new file formats without having to change the main program.

Plug-ins can be created for archive, image and sound file formats. Special types of plug-ins exist for game viewers and file type detectors that are used by the scanner.

This version of the guide contains only information on how to implement support for new archive file formats. Plug-in types that are not documented in this guide are unsupported.

3. Plug-in basics

Ravioli Game Tools uses the Microsoft .NET Framework.

All Ravioli Game Tools plug-ins are .NET class libraries compiled for the .NET Framework 4.5.1.

One plug-in can provide support for multiple file formats. One public class exists for every new file format to support. These classes implement specific interfaces or derive from base classes provided by Ravioli Game Tools.

Every plug-in uses the RArchiveInterface library (part of Ravioli Game Tools and the Ravioli SDK) to access the required interfaces and base classes.

The preferred language for writing a plug-in is C#. The RArchiveInterface library is CLS-compliant to ensure compatibility to other .NET languages. Beyond this, no special support is provided to help creating plug-ins in other languages.

The naming convention for plug-ins is generally "XYZPlugin". If a plug-in supports only a single file type, "XYZ" is the name of the supported file type, often in abbreviated form. For plug-ins supporting multiple file types, „XYZ“ is the name of the game, company, engine or program whose file types are supported. In some cases the naming convention "XYZPlugins" is used to indicate that a single plug-in supports multiple file types, but this is a very technical view and can be confusing.

4. Creating a new plug-in

The easiest way to create a new plug-in is to use a development environment that supports .NET Framework 4.5.1. If you are using Microsoft Visual Studio, you need at least Visual Studio 2013.

How to create a new plug-in with Microsoft Visual Studio 2013 and higher:

- 1) Create a new project (File > New > Project)
In the New Project dialog, select Visual C# > Windows Desktop > Class library.
Ensure the target .NET version is set to ".NET Framework 4.5.1".
- 2) Add a reference to RArchiveInterface.dll from the Ravioli SDK's „Libraries“ directory.
Set the property "Copy Local" of the reference to "False".

- 3) Create a new public class for every new file type to support. Implement interfaces or derive from bases classes depending on the file type. See the section "Implementing support for a new archive file format" for more information.
- 4) Build the project (Build > Build Solution). If there are no errors, there is now a DLL file in the project's output directory. This is the plug-in to install into Ravioli Game Tools (see "Installing and uninstalling a plug-in" below).
- 5) Test the plug-in with Ravioli Game Tools to make sure it works as expected. Make changes as necessary.
- 6) Set the plug-in version number.
The version numbers are set using the "AssemblyVersion" and "AssemblyFileVersion" attributes in the file "AssemblyInfo.cs" located in the project's "Properties" subdirectory. New plug-ins usually start with version number 1.0.0.0. For changed plug-ins, the version number must be increased according to the scale of changes. Set both attributes to the same version unless required otherwise.
- 7) Build the project in "Release" mode.
This removes information added automatically to the plug-in for troubleshooting during development and optimizes the generated code. The resulting DLL in the project's output directory can be given to others to install.

5. Installing and uninstalling a plug-in

The plug-ins are stored in the "Plugins" subdirectory of a Ravioli Game Tools installation.

Copy a plug-in into the "Plugins" directory to install it. It will be loaded at the next start of a Ravioli Game Tools application.

Delete a plug-in from the "Plugins" directory to uninstall it. It will no longer be loaded at the next start of a Ravioli Game Tools application.

6. Implementing support for a new archive file format

An archive plug-in has the following responsibilities:

- Performs basic format checks to determine whether a file is actually in a supported format.
- Provides a list of items that are present in an archive.
- Extracts selected or all items from an archive.

Depending on the structure and the complexity of the file format, the approaches to implement the format are different. For simple, common file structures, specialized base classes are provided. For more complex file structures, the implementation starts on lower levels.

6.1. Archives with simple structures

If the file type to support has one of the following structures, use the corresponding base class provided. Further information is in the chapters below.

GenericArchive (see chapter 6.1.1):

Directory
Uncompressed data of file 1
Uncompressed data of file 2
Uncompressed data of file 3
...

GenericCompressedArchive (see chapter 6.1.2):

Compressed archive	Decompression →	Directory
		Uncompressed data of file 1
		Uncompressed data of file 2
		Uncompressed data of file 3
		...

GenericCompressedArchive2 (see chapter 6.1.3):

Directory	Decompression →	Directory
Compressed data block		Uncompressed data of file 1
		Uncompressed data of file 2
		Uncompressed data of file 3
		...

CompressedItemsArchive (see chapter 6.1.4):

Directory	Decompression →	Directory
Compressed data of file 1		Uncompressed data of file 1
Compressed data of file 2		Uncompressed data of file 2
Compressed data of file 3		Uncompressed data of file 3
...		...

6.1.1. All data uncompressed and continuous (GenericArchive)

This is the most generic type of an archive and many simple file formats use it. The data of every file in the archive is a single continuous block that can be copied out of an archive without modifications. The archive directory (table of contents) is uncompressed or can be decompressed on the fly.

Directory
Uncompressed data of file 1
Uncompressed data of file 2
Uncompressed data of file 3
...

Structure of a „GenericArchive“

To implement support for a file format which this characteristic, do the following:

1) **Create a new public class derived from the GenericArchive class**

This is in the namespace Ravioli.ArchiveInterface.

2) **Implement the TypeName property**

This is the name of the file format that is displayed in the Ravioli Game Tools.

If the file format has a known name, use that name. Example: "Xerberus Zombie Format".

If the file format does not have a known name and is not game or company-specific, use the file extension without a leading period and in upper case, as well as the word "File", separated by a space. Example: "XZF File".

If the file format does not have a known name and is game or company-specific, use the name of the game or company, the file extension without a leading period and in upper case, as well as the word "File", all separated by spaces. Example: "CoolGame XZF File".

3) **Implement the Extensions property**

Returns the supported file name extensions for this file format. Always include the leading period and write the extension in lower case, for example ".xzf".

4) **Implement the IsValidFormat method**

This method verifies that a given file is of the type supported by the implementing class.

Ravioli Game Tools calls this method whenever it needs to find out which plug-ins support a given file. This method may be called often, so the format check should be brief and must not throw exceptions. Only as much as necessary should be checked to determine with a certain probability that the file is of the correct type. For example, if a file format has a magic header of at least 4 bytes that is unique enough, this might be sufficient as a format check.

At the start of the method, IsValidFormat should reposition the stream to the beginning of the file, for example using "stream.Seek(0, SeekOrigin.Begin);" in C#. This handles the case, where IsValidFormat is called multiple times on an open file.

5) Implement the ReadGenericDirectory method

This method reads the archive's directory and returns a set of GenericDirectoryEntry objects, each representing a file in the archive.

A GenericDirectoryEntry object consists of:

- * Name: The file name
- * Offset: The file start position, as seen from the beginning of the archive file
- * Length: The file length in bytes

The IsValidFormat method should be called immediately at the beginning of the ReadGenericDirectory method to ensure that the file processing continues only with a correct file, regardless whether IsValidFormat was called before or not. If IsValidFormat returns false, the method should raise an InvalidDataException with a message like "This is not a valid (name of file format) file."

The ReadGenericDirectory method is called with two parameters, "stream" for direct access to the file stream and "reader" for reading primitive data types from binary files. Use the reader as much as possible.

Example code:

The following is a full C# implementation of a GenericArchive for the file type „Quake PAK File“. The namespace definition has been removed for readability.

```
PakFile.cs
1 using System;
2 using System.IO;
3 using System.Text;
4 using Ravioli.ArchiveInterface;
5
6 public class PakFile : GenericArchive
7 {
8     public override string TypeName
9     {
10         get { return "Quake PAK File (Sample)"; }
11     }
12
13     public override string[] Extensions
14     {
15         get { return new string[] { ".pak" }; }
16     }
17
18     protected override bool IsValidFormat(Stream stream, BinaryReader reader)
19     {
20         stream.Seek(0, SeekOrigin.Begin);
21         uint signature = reader.ReadUInt32();
22         return (signature == 0x4B434150); // "PACK" signature
23     }
24
25     protected override GenericDirectoryEntry[] ReadGenericDirectory(Stream
26 stream, BinaryReader reader)
```

```

27         if (!IsValidFormat(stream, reader))
28             throw new InvalidDataException("This is not a valid PAK
file.");
29
30         uint directoryOffset = reader.ReadUInt32();
31         uint directorySize = reader.ReadUInt32();
32         stream.Seek(directoryOffset, SeekOrigin.Begin);
33         uint fileCount = directorySize / 64;
34         GenericDirectoryEntry[] directory = new
GenericDirectoryEntry[fileCount];
35         for (uint i = 0; i < fileCount; i++)
36         {
37             GenericDirectoryEntry entry = new GenericDirectoryEntry();
38             byte[] nameBytes = reader.ReadBytes(56);
39             entry.Name = Encoding.GetEncoding(1252).GetString(nameBytes);
40             int zeroPos = entry.Name.IndexOf('\0');
41             if (zeroPos >= 0)
42                 entry.Name = entry.Name.Substring(0, zeroPos);
43             entry.Offset = reader.ReadUInt32();
44             entry.Length = reader.ReadUInt32();
45             directory[i] = entry;
46         }
47         return directory;
48     }
49 }

```

6.1.2. Whole archive compressed in one block (GenericCompressedArchive)

The whole archive is one large compressed block. Decompressing the archive results in the directory and all files decompressed, and no further decompression is required.



Structure of a "GenericCompressedArchive" before and after decompression

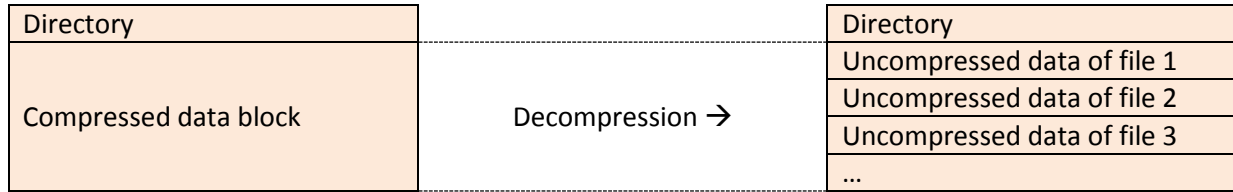
The implementation is similar to that of a GenericArchive (see chapter 6.1.1).

There are two differences:

- The implementing class is derived from the **GenericCompressedArchive** class instead of the GenericArchive class.
- An additional method **DecompressArchive** must be implemented. This method is called once before **ReadGenericDirectory**. It receives the whole archive data as a stream, and writes out the decompressed data. ReadGenericDirectory operates transparently on the decompressed data afterwards.

6.1.3. Data part of archive compressed in one block (GenericCompressedArchive2)

The archive's directory is uncompressed (or can be decompressed on the fly), while the remaining file data is one large compressed block. Decompressing the data block results in all files decompressed, and no further decompression is required.



Structure of a „GenericCompressedArchive2“ before and after decompression

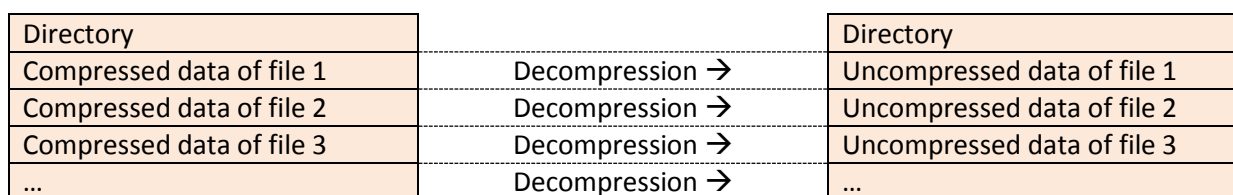
The implementation is similar to that of a `GenericArchive` (see chapter 6.1.1).

There are three differences:

- The implementing class is derived from the **GenericCompressedArchive2** class instead of the `GenericArchive` class.
- An additional method **DecompressArchive** must be implemented. This method is called once after calling **ReadGenericDirectory**. It receives the remaining file data and writes out the decompressed data. The `byteCount` parameter indicates the number of uncompressed bytes to write. The `ReadGenericDirectory` method processes the uncompressed directory and ensures that the file pointer is set to the start of the compressed data after reading the directory, so that `DecompressArchive` can operate properly.
- An additional property **UncompressedArchiveLength** must be implemented. This is the number of uncompressed bytes that **DecompressArchive** is going to write. The value of this property must be valid after the call to **ReadGenericDirectory**.

6.1.4. Files are compressed individually (**CompressedItemsArchive**)

The archive's directory is uncompressed (or can be decompressed on the fly), while the files in the archive are individually compressed.



Structure of a „CompressedItemsArchive“ before and after decompression

The implementation is similar to that of a `GenericArchive` (see chapter 6.1.1).

There are three differences:

- The implementing class is derived from the **CompressedItemsArchive** class instead of the `GenericArchive` class.
- A method **ReadCompressedDirectory** must be implemented instead of **ReadGenericDirectory**. The method name might imply that the directory is compressed, it does not have to be. The method name is different because it has a different return value than `ReadGenericDirectory` - it returns a list of **CompressedDirectoryEntry** objects instead of a list of **GenericDirectoryEntry** objects.

There are two additional properties in a `CompressedDirectoryEntry`:

- **Compressed:** A boolean value – true if the file is compressed, false if not.
 - **CompressedLength:** The compressed size of the file as it is stored in the archive. If the file is not compressed, `CompressedLength` must be set to the uncompressed length (same value as the `Length` property).
 - **CompressionType:** If the file is compressed, contains the method the file was compressed with. Can be ignored for simple archives that support only one compression method.
-
- A method **DecompressFile** must be implemented. This method is called whenever a compressed file in the archive (indicated by `Compressed=true`) must be decompressed. It receives the compressed data of a file as a stream and writes out the decompressed data. The `byteCount` parameter indicates the number of uncompressed bytes to write. The `compressionType` parameter indicates the method the file was compressed with.

6.2. Archives with more complex structures

For more complex file formats that are not covered by the specialized base classes, all handling of the file data in an archive must be implemented.

For example, the files in an archive could be fragmented into multiple parts:

Directory
Data of file 1, part 1
Data of file 2, part 1
Data of file 1, part 2
...

Structure of an archive whose files are fragmented

The file format implementation must be based on the **ArchiveBase** class.

The implementation steps are the following:

1) **Create a new public class derived from the ArchiveBase class**

This is in the namespace Ravioli.ArchiveInterface.

2) **Implement the TypeName and Extensions properties and the IsValidFormat method**

This is implemented in the same way as in classes based on the GenericArchive class (see chapter 6.1.1).

3) **Implement the Files property**

The getter of the property returns a flat list of files in the archive. Since the return value **IFileInfo** is an interface, a concrete class is needed. You can either have the class containing the archive's data structures implement the **IFileInfo** interface, or use a dedicated class to return the file information, such as the **ArchiveFileInfo** class.

The following properties must be implemented for every file in the archive:

- * **ID:** A number that uniquely identifies the file in the archive.
- * **Name:** The file name. File names are always specified as relative paths, e.g. a file name "file1.dat" indicates a file "file1.dat" in the root directory of the archive, and a file name "dir2/file2.dat" indicates file "file2.dat" in subdirectory "dir2".
- * **Size:** The file size in bytes after extracting it from the archive.

If the files are compressed, you must additionally implement the **ICompressionInfo** interface, or use the dedicated **CompressedFileInfo** class instead of **ArchiveFileInfo**.

4) **Implement the ReadDirectory method**

This method reads the archive's directory and stores it internally for further use, for example in a private "directory" class variable. There are no specific requirements how to store this data. The required data structures depend on the information from the archive that needs to be stored.

The IsValidFormat method should be called immediately at the beginning of the

ReadDirectory method to ensure that the file processing continues only with a correct file, regardless whether IsValidFormat was called before or not. If IsValidFormat returns false, the method should raise an InvalidDataException with a message like "This is not a valid (name of file format) file."

5) **Implement the OnClose method**

OnClose is called when the archive is no longer needed. Release all resources you are using in your archive implementation. In the .NET world, you must generally call the "Dispose" or "Close" method on all objects that have them, to make sure its resources are released immediately. If you have allocated native memory or are using other native resources, like handles, you must also release them.

Besides releasing resources, make also sure your member variables are in a state that does not confuse the implementation when the archive is reopened (with the same or a different file). If you want to be on the safe side, you can just reset all member variables to their default values (usually null or 0, depending on the data type).

6) **Implement the ExtractFile method**

This method extracts a file from an open archive. There are multiple overloads for this method, and you must implement all:

- * One for specifying a file name as a target
- * One for specifying a stream as a target
- * One for specifying a stream as a target, and the number of bytes requested to extract (the "byteCount" parameter).

All overloads have a "file" parameter of type IFileInfo, specifying the file to extract and a "stream" parameter representing the open archive's stream. Implement the necessary logic to read the file's content from the input stream and write it to the specified target (file or stream).

In the method that has the "byteCount" parameter, byteCount is only relevant if its value is between 0 and the file size, otherwise it is ignored and all bytes of the file are extracted. You can use the following C# formula to calculate the actual number of bytes to extract:

```
long bytesToExtract = (byteCount >= 0 && byteCount < fileSize) ? byteCount :
fileSize;
```

7) **Implement the OnFirstExtraction method (optional)**

If required for advanced implementations, OnFirstExtraction can be used to perform operations when the first file is to be extracted since the archive was opened, but before ExtractFile gets called.

Example code:

The following is a full C# implementation of an ArchiveBase for the file type „Quake WAD (WAD2/WAD3) File“. The namespace definition has been removed for readability. Some referenced objects are not part of this sample.

Wad3File.cs

```

1  using System;
2  using System.IO;
3  using System.Text;
4  using Ravioli.ArchiveInterface;
5
6  public class Wad3File : ArchiveBase
7  {
8      private Wad3DirectoryEntry[] directory;
9
10     public override string TypeName
11     {
12         get { return "Quake WAD (WAD2/WAD3) File (Sample)"; }
13     }
14
15     public override string[] Extensions
16     {
17         get { return new string[] { ".wad" }; }
18     }
19
20     public override IFileInfo[] Files
21     {
22         get
23         {
24             ArchiveFileInfo[] array = new
ArchiveFileInfo[this.directory.Length];
25             for (int i = 0; i < this.directory.Length; i++)
26             {
27                 Wad3DirectoryEntry entry = this.directory[i];
28                 array[i] = new ArchiveFileInfo(i, entry.Name,
entry.CompressedSize);
29             }
30             return array;
31         }
32     }
33
34     protected override void ExtractFile(IFileInfo file, Stream stream, string
outputFileName)
35     {
36         Wad3DirectoryEntry entry = this.directory[file.ID];
37         FileStream outputStream = new FileStream(outputFileName,
FileMode.Create, FileAccess.Write);
38         try
39         {
40             CopyData(entry, stream, outputStream, -1);
41             outputStream.Close();
42         }
43         catch (Exception)
44         {
45             outputStream.Close();
46             if (File.Exists(outputFileName))
47                 File.Delete(outputFileName);
48             throw;
49         }
50     }
51
52     protected override void ExtractFile(IFileInfo file, Stream stream, Stream

```

```

        outputStream)
53     {
54         CopyData(this.directory[file.ID], stream, outputStream, -1);
55     }
56
57     protected override void ExtractFile(IFileInfo file, Stream stream, Stream
outputStream, long byteCount)
58     {
59         CopyData(this.directory[file.ID], stream, outputStream, byteCount);
60     }
61
62     protected override bool IsValidFormat(Stream stream, BinaryReader reader)
63     {
64         stream.Seek(0, SeekOrigin.Begin);
65         uint signature = reader.ReadUInt32();
66         return (signature == 0x32444157 || signature == 0x33444157); //
WAD2/WAD3 signature
67     }
68
69     protected override void ReadDirectory(Stream stream, BinaryReader reader)
70     {
71         if (!IsValidFormat(stream, reader))
72             throw new InvalidDataException("This is not a valid WAD
file.");
73
74         // Header
75         stream.Seek(0, SeekOrigin.Begin);
76         uint signature = reader.ReadUInt32();
77         uint fileCount = reader.ReadUInt32();
78         uint directoryOffset = reader.ReadUInt32();
79
80         // Directory
81         stream.Seek(directoryOffset, SeekOrigin.Begin);
82         Wad3DirectoryEntry[] newDirectory = new
Wad3DirectoryEntry[fileCount];
83         for (uint i = 0; i < fileCount; i++)
84         {
85             Wad3DirectoryEntry entry = new Wad3DirectoryEntry();
86             entry.Offset = reader.ReadUInt32();
87             entry.CompressedSize = reader.ReadUInt32();
88             entry.UncompressedSize = reader.ReadUInt32();
89             entry.Type = reader.ReadByte();
90             entry.CompressionType = reader.ReadByte();
91             entry.Padding = reader.ReadUInt16();
92             byte[] nameBytes = reader.ReadBytes(16);
93             entry.Name = Encoding.GetEncoding(1252).GetString(nameBytes);
94             if (entry.Name.Contains("\0"))
95             {
96                 int zeroPos = entry.Name.IndexOf('\0');
97                 entry.Name = entry.Name.Substring(0, zeroPos);
98             }
99             newDirectory[i] = entry;
100         }
101         this.directory = newDirectory;
102     }
103
104     protected override void OnClose()
105     {
106         this.directory = null;
107     }
108
109     private void CopyData(Wad3DirectoryEntry entry, Stream stream, Stream
outputStream, long byteCount)

```

```
110     {
111         stream.Seek(entry.Offset, SeekOrigin.Begin);
112         long remaining = (byteCount >= 0 && byteCount <
entry.CompressedSize) ? byteCount : entry.CompressedSize;
113         const int bufferSize = 8192;
114         byte[] buffer = new byte[bufferSize];
115         while (remaining > 0)
116         {
117             int bytesToRead = (remaining > bufferSize ? bufferSize :
(int)remaining);
118             int bytesRead = stream.Read(buffer, 0, bytesToRead);
119             outputStream.Write(buffer, 0, bytesRead);
120             remaining -= (uint)bytesRead;
121         }
122     }
123 }
```

6.3. Additional archive features

Archives can implement additional interfaces for special uses:

- `IDataWriter`
- `IRootDirectory`
- `IExtensionIndependent`

6.3.1. `IDataWriter` interface

An interface for archive plug-ins that create private (plug-in specific) data files.

This interface has only one member, **`DataDirectory`**. The value of this property is the absolute directory name where all data from all archive plug-ins is stored. The host application sets this property before opening the archive and guarantees that the specified directory exists at that time.

Writing directly to the directory name specified in **`DataDirectory`** shares the data with all plug-ins. To store private plug-in data, combine the directory name with a unique subdirectory name and write all data into this directory only.

6.3.2. `IRootDirectory` interface

An interface for archive plug-ins to work with archives that have their items stored outside the archive file on the file system and the plug-in cannot determine the location of the items automatically.

This interface has only one member, **`RootDirectory`**. The value of this property is the absolute file system directory, where the items listed in the archive are stored. All files and subdirectories in this directory must be laid out as specified in the archive. The property is set by the host application before opening the archive. The user has to provide this directory name to the application in some way, which depends on the application.

When extracting a file from the archive, combine the **`RootDirectory`** with the name of the file to extract and use the data from the resulting file name.

6.3.3. `IExtensionIndependent` interface

An interface for archive plug-ins to indicate it implements a universal, highly popular file format.

This interface has only one member, **`ExtensionIndependent`** (a boolean). Set the value of this property to **`true`** to indicate that the archive format implemented by the plug-in is so universally used that maintaining the list of all supported file extensions is unreasonable. The ZIP file is an example of such a popular archive format.

If **`ExtensionIndependent`** is set to **`true`**, the host application will attempt to open all archives with this plug-in. For this to work well, ensure that you have a robust implementation of the archive's **`IsValidFormat`** method.