České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Vrátník Jan

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: C# knihovna pro verifikaci a validaci souborů

Pokyny pro vypracování:

Výsledkem diplomové práce bude knihovna usnadňující ověření správnosti různých souborů podle předem zvolených parametrů. Tyto parametry jsou například nepřítomnost virů, správný formát podle přípony a podobně.
Nastudujte problematiku verifikace a validace souborů. Zaměřte se především na ověření správnosti dokumentových souborů (například z rodiny MS Office, Libre Office a Adobe Acrobat). Proveďte analýzu antivirových programů a zhodnoťte jejich vhodnost pro použití s vytvářenou knihovnou.
Navrhněte a implementujte modulární knihovnu v C#, která detailně otestuje platnost vstupních souborů a ověří jejich nezávadnost pomocí antivirových nástrojů. Knihovna bude umožňovat jednoduchou integraci různých antivirových programů a volitelné testování souborů podle různých parametrů (například velikost, správný kontrolní součet, platnost dat v odpovídajícím formátu).
Navrhněte a zdůvodněte způsob otestování knihovny. Navržené testy proveďte a zhodnoťte.

Seznam odborné literatury:

[1] C# Reference. MSDN. [online]. 2017 [cit. 2017-01-10]. Dostupné z: https://msdn.microsoft.com/en-us/library/618ayhy6.aspx
[2] FOWLER, Martin. FluentInterface. MartinFowler.com. [online]. 20.12.2005 [cit. 2017-01-10]. Dostupné z: https://www.martinfowler.com/bliki/FluentInterface.html
[3] KESSLER, Gary. File Signatures Table. Gary Kessler Associates. [online]. 23.7.2016 [cit. 2017-01-10]. Dostupné z: http://www.garykessler.net/library/file_sigs.html

Vedoucí: Ing. Jan Kubr
Platnost zadání do konce letního semestru 2017/2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-
L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 20.2.2017

Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science

Master's thesis

File verification and validation C# library

Bc. Jan Vrátník

May 2017
Thesis supervisor: Ing. Jan Kubr

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 1. 5. 2017

[illegible]

Poděkování

Rád bych poděkoval Ing. Janu Kubrovi za odborné vedení této práce a za poskytnutí cenných rad. Velké díky patří také Jirkovi za nepostradatelné tipy kolem .NET a Tanje za korekturu. Dále samozřejmě nemohu zapomenout ani na svoji rodinu a přátele, bez kterých bych tuto práci dokončil dříve.

Abstrakt

Validace souborů mnohdy končí kontrolou jeho velikosti a přípony. I když něco takového ve většině případů naprosto stačí, najdou se i situace, kde je potřeba jít mnohem více do hloubky. Bohužel, existuje jen málo volně dostupných knihoven, které se zabývají velmi detailní validací souborů, se kterými běžně pracujeme na našich strojích. Cílem této práce je vytvořit knihovnu v jazyce C#, která bude schopna zkontrolovat soubory podle požadovaných parametrů - velikosti, přípony, kontrolního součtu apod. Zároveň bude schopna pro vybranou množinu formátů zjistit, zda se jedná o poškozené soubory, které není možné za pomoci běžných nástrojů otevřít. Součástí této kontroly bude i integrace existujících antivirových programů. Výsledná práce by měla poskytnout volně dostupnou alternativu pro programátory, kteří se ve svých projektech potýkají s problematikou detailní kontroly vstupních souborů a potřebují řešení, které je zdarma, které se dá snadno rozšířit o další funkcionalitu a které má uživatelsky přívětivé rozhraní.

Abstract

The validation of a file often begins and ends with checking its size and extension. While this is often sufficient, there are some cases where a much more thorough validation is needed. Unfortunately, there are only a few open source libraries that focus on validating files we work with on a daily basis. The aim of this thesis is to create a library in C# that can check files based on specified parameters such as size, extension, checksum, etc. Furthermore, it will also be able to detect damaged files which can't be opened by any standard software. This process will also know how anti-virus engines can be integrated should such software be installed on a user's system. The result of this thesis should provide an open source alternative for programmers who are dealing with detailed file validation in their projects and are looking for a solution that is free, user-friendly and easily extensible with new functionality.

Contents

List of Figures

List of Tables

Part 1
Introduction

When it comes to files and validation, most of us probably do not think about it too much. We work with files every day. They are a necessary part of the data we store. Without files, all the information would be just a pile of ones and zeroes without order and meaning. We keep our accounting information in XLS (Microsoft Excel Spreadsheet) files, invoices from our last online transactions in PDF (Portable Document Format), pictures from family vacation in JPEG (Joint Photographic Experts Group) files, backup of our company's database in some special file stashed somewhere safe and backed up ten times over.

We exchange files with others on a daily basis for many different reasons, be it a draft of our first book for the publisher, documentation of the application we've worked on for months which we want to share with the world or literally anything else.

More often than not, we do not put too much emphasis on checking whether these files are what we want them to be. What does it matter if the picture is one megabyte or two? What does it matter if it is in JPEG or PNG format? What does it matter if I cannot open one of the hundred pictures I took that day? And it is true. Most of the time, it does not matter at all.

The problem presents itself when it matters. Let us say we have received a request from our client with data attached to it in a particular set of files. Let us say we are a modern company that strives to handle these requests automatically (a thing that is not as common as you might think) via some online form and let us assume these attached files are critical for a correct evaluation of the request. One of the many steps we must take is to make sure these files are not corrupted or invalid in any other way. If that is the case, we want to notify the user immediately, so he or she can fix the problem and send the request again. This sounds like an issue many companies may encounter and solve by using one of many available form validation tools and libraries. Indeed, there are plenty of them.

However, form input validation is science in its own right and files are only a fraction of what might be sent inside an online form. Most of the libraries check only very generic meta-data like extension and file size. Granted, fairly often we do not need more. But what if we do?

There are plenty of frameworks, libraries and other tools that help with form validations. A quick internet search would yield dozens of results. But when we narrow it down to detailed file validations compatible with C# language, we are left with hardly anything. This is where we come in.

The primary goal of this thesis is to create a library which can provide us with a means of file validation and verification, be it from an online form or any other source. It should provide options to filter unsupported files using many various techniques that are beyond the capabilities of most standard form validation frameworks. The interface of this library should be simple, easy to pick up and should provide tools which can enable us or anyone else to easily extend the core functionality provided by this library.

For a truly thorough inspection of files, an anti-virus engine should also be included. This thesis will, therefore, include an analysis of a few popular engines, specifically their business model, usage and how they can be integrated into the library. The library should be able to work with most, if not all, anti-virus programs, but looking at the few most renowned, it could provide an important insight into how this can be achieved.

Another goal of this thesis is to analyze how office formats such as spreadsheets, text documents, and presentations can be thoroughly validated, whether the format in questions is a Microsoft Word document, LibreOffice Calc spreadsheet, Microsoft Access database or a PDF file. Such formats are very frequent in the business environment and should be a subject of more thorough validation in the library we will be working on.

1.1 Thesis outline

First we will analyze the requirements and functions our library should support by looking both at generic file characteristics as well as the characteristics of common office file formats. We will also look at how other similar existing libraries deal with the validation of files. We will then inspect a couple of popular anti-virus engines as they are essential for our library to do its job properly. This will be done in part two of this thesis. In the third part, we look at how the library is implemented and what steps were taken to make it easy to use. Next, in part four, we look at how our library's functionality is tested to ensure it works as intended. This chapter will also include the information and results of the performance tests that will be conducted as a part of this thesis. Lastly, in part five we will discuss the future of the final product, its possible enhancements; and we will also summarize what we have achieved in this thesis.

Part 2
Analysis

In this chapter we will first look at what is required of our software. Afterward, we will look whether there are some other libraries dealing with file validation and try to improve on them. The next part will cover all the things we can validate on a file's meta-data and how we can check the integrity of its data. In another section of this chapter, we will also review a few popular AV engines, how they can be used, what their business model is and how they can be integrated in our library.

Since the result of this thesis is to be made public under an open source license, it is also crucial that the library provides a simple interface and can easily be customized and extended as users see fit. We will explore our options in the last section of this chapter, where we will consider our selection of open source license.

2.1 Specification of requirements

We will first talk about the choice of the programming language. **The library will be written in C#** as that was one of the main requirements given by the submitter for the subject of this thesis.

The library will be divided into three distinct parts. The first part will be taking care of generic validations which can be used for any type of files. The second part will take care of verifying the integrity of common file types. We will primarily be looking at common document types such as .doc, .xls or .pdf. The third part will contain all the necessary functions which the other two parts can use and provide an interface which will allow users to work with the library. As such, we have written down requirements for all three of these sections.

2.1.1 Requirements for library's core functions

(C1) Library will be able to integrate user's anti-virus software as part of the validation scans. This integration should be independent of the particular AV software used. Should such a software be set up, it must be run before any other validations are started to prevent any damage being done by infected files.

(C2) Library will be extensible with additional validations.

(C3) Library will expose Fluent interface which will provide options to set up the library's filter parameters.

(C4) Library will be able to list all loaded validation modules.

(C5) Library will log its actions via a logging library.

2.1.2 Requirements for generic validations

(G1) Library will be able to filter through files based on their extension.

(G2) Library will be able to filter through files based on their size. Setting both the minimum and maximum allowed size will be possible.

(G3) Library will be able to check whether a file's extension matches its file signature (magic number). The library will not require an external database and will have its own list of known file signatures which can be easily edited.

(G4) Library will be able to filter through files based on their file checksum.

2.1.3 Requirements for integrity verifications

(I1) Library will be able to scan common file types and decide whether the file is corrupted and cannot be opened.

(I2) Library will be written in a way that will make implementing support for additional file types as simple as possible.

2.2 Similar existing libraries

To find libraries dealing with file validations, we start by looking for libraries dealing with form validations. Uploading files is very often a part of such forms, so this might be a good way to start.

We quickly learn, however, that there are not too many standalone libraries, as many frameworks have form validations integrated within them. There are libraries which help to validate text inputs, checking whether a string is an email, phone, address, etc. This is not quite what we are looking for, but the few examples we can find can serve as an inspiration. We will inspect them, see what they find relevant and expand on that. Many of their functions may not be relevant to us, but we can learn from the interface they provide and look at how we can improve on that.

One way or another, the libraries we discovered are either not very flexible (e.g. being fixed to web forms) or they do not provide the functionality we need. This further proves that our library can bring new options to the table.

2.2.1 Respect/Validation

Respect/Validation is a PHP (Hypertext Preprocessor) form validation library that offers many tools, most of which are not important to us. It does, however, provide us with a few essential tools for validating files. According to their GitHub documentation, the library offers the following rules.[1]

- Executable - Returns true if file is an executable.
- Exists - Returns true if file exists.
- Extension - Returns true if file has requested extension ('.jpg' for example).
- Mimetype - Returns true if file has requested mimetype ('text/plain' for example).
- Readable - Returns true if file exists and can be read from.
- Size - Returns true if file's size is within set limits.
  - First argument sets the minimum allowed size.
  - Second argument sets the maximum allowed size (optional).
  - Units can be set by appending number with 'KB', 'MB', 'GB' etc.
- SymbolicLink - Returns true if given file is a symbolic link.
- Uploaded - Returns true if file was uploaded via HTTP POST method.
- Writable - Returns true if file exists and can be written to.

We can see that some of the validated metadata like extension and size, while extremely basic, are too important to be omitted from any library. It is also obvious that some functions are made specifically with online forms in mind, namely *Mimetype* and *Uploaded*. *Exists* is a function that I believe should always be used before every validation, it should not exist as a mere function we can use at random. The idea is to make sure files passed by the user to our library always exist before we attempt to start the validation process.

2.2.2 JeremySkinner/FluentValidation

JeremySkinner/FluentValidation is a C# validation library, which offers methods for validating entities and their parameters. The library is not important to us because of the functions it provides, but because it implements them via Fluent interface (thoroughly described in chapter **Fluent interface** on page 21). From the library's GitHub documentation and a short code excerpt below we can see how the user can chain conditions placed upon each of validated parameters.[2]

```csharp
using FluentValidation;

public class CustomerValidator: AbstractValidator<Customer> {
  public CustomerValidator() {
    RuleFor(customer => customer.Surname).NotEmpty();
    RuleFor(customer =>
       customer.Forename).NotEmpty().WithMessage("Please specify a
       first name");
    RuleFor(customer => customer.Discount).NotEqual(0).When(customer =>
       customer.HasDiscount);
    RuleFor(customer => customer.Address).Length(20, 250);
    RuleFor(customer =>
       customer.Postcode).Must(BeAValidPostcode).WithMessage("Please
       specify a valid postcode");
  }
}
```

Code preview 1: Example of FluentValidation methods

The library does not seem to support any validation functions related to files, but we can see how this can easily be transformed into our library. For example, our library's interface could support something like this:

Validate(file).MaxSize(100).Extension(".jpg");

We do not even have to explain what it does, and it is quite obvious what it validates. The fluent interface looks like the perfect choice for us.

2.3 File characteristics

A file is a medium that allows us to store data on our computers discretely. Aside from the data it contains, it also comes with many other metadata that enables us to tell a lot about the file's purpose without even opening it - the date of creation, name, size, format as well as many other details. This can be useful to quickly filter out files we do not want to work with.

2.3.1 Format

The file format defines how to encode data into a file which we can later store on a hard drive. Most file formats have a very particular usage, while some may act as archive formats that can encapsulate other files within themselves.

2.3.1.1 Text file formats

Every file is saved on a drive as a sequence of ones and zeros, which can hardly come as a surprise. Some files, however, contain data in a way that allows it to be decoded into characters readable by humans using ASCII (American Standard Code for Information Interchange), UTF-8 (UCS/Unicode Transformation Format) or some other character encoding method. Such data can be easily read and edited for further use via any text editor.

While there are many formats, they can all mostly be rendered using the same text editors. The specific format identified by the file's extension or signature can serve as a hint for the text editor to parse contents of the file in a particular manner, should it be useful. With correct information, text editors may assist with highlighting the text - for example, .java files can be highlighted as Java source code or .csv files can be transformed into a plain table. Simply put, text files contain human-readable text, but with the right information about particular format we have many options when it comes to interpreting it.

2.3.1.2 Binary file formats

Binary files are files that are not meant to be interpreted as text files (very often this is not even possible). Some parts may contain plain text that can be read, but it is mostly a sequence of bytes that serve another purpose. Binary files also often consist of headers and other blocks of metadata to assist programs in reading these files correctly. The problem with binary files is that they are a sequence of bytes which could represent almost anything. A byte could substitute for a pixel, a number, a sound or any other information. Only after we try to interpret the sequence of bytes using an algorithm can we tell what kind of information and format the file represents.

For operating systems to quickly suggest correct program to open the file with (and a correct algorithm), a file extension can be used as a hint. More on file extensions can be found in section **Extensions** (page 8). For a program to quickly determine the file format, a file signature (also known as magic number) can be used. More on file signatures can be found in section **Signatures** (page 9).

2.3.2 Filename

Filenames allow us to identify files on a computer's file system. We can understand the name as the name of the file within a given directory (e.g., 'invoice.pdf') or as an absolute path on a given file system (e.g., 'C:\Users\Documents\invoice.pdf'). There are restrictions set in place to avoid issues that may occur when storing it, usually limiting the length of the name or characters it can contain.

Reserved characters may differ across systems. Focusing on those used by Windows and UNIX systems, we can find the reserved characters listed below.[3]

- / (forward slash) - Separates components in the path name.
- \ (backslash) - Similar to previous point.
- * (asterisk) - Used as a wildcard
- ? (question mark) - Used as a wildcard replacing a single character.
- < (less than) - Used to redirect input. Allowed in UNIX systems.
- > (greater than) - Used to redirect output. Allowed in UNIX systems.
- : (colon) - Among other things, a colon can be used to determine the mount point or drive in Windows.
- " (double quote) - Used to mark the beginning and end of names containing spaces in Windows.
- | (vertical bar or pipe) - Used for software pipelining both in Unix and Windows.

If our library is to validate files thoroughly, we should take validating filenames into consideration. Granted, if users pass files to the library for validation, they probably already exist within a given file system, and their name already meets all the requirements, but as with many other things in software development, there may be a use case where this does not hold true.

2.3.3 Extensions

The filename extension is a suffix regularly included after the name of the file, often delimited by a dot. Extensions often provide a hint for the operating system as to how data is stored within that file. The operating system can quickly assess this information and suggest the correct program to open it with. UNIX systems take the extension as part of the file's name and as such multiple extensions can be included in the name of the file (e.g., 'archive.tar.gz').
Windows does not treat extension as part of a file's name and hides it. This can lead to security issues as it allows the attacker to disguise malicious software under a file that may look innocent. Let us say we have an executable batch file *readme.txt.bat* which can run some malicious scripts in the background. The user will see this file as *readme.txt* and won't consider it a risk opening this supposed text file. The latest versions of Windows allow users to display extensions next to filenames, but it is not enabled by default and still poses a threat for inexperienced users.
There is nothing stopping users from changing file's extension. Simply by renaming the file and changing the extension we can mask a file's true format and slip through many simple file validators which only check the extension.

Format	**Signature**
BMP	42 4D
GIF	47 49 46 38
JPG	FF D8 FF E0 FF D8 FF E1 FF D8 FF E8
PNG	89 50 4E 47 0D 0A 1A 0A
TIFF	49 20 49 49 49 2A 00 4D 4D 00 2A

Table 1: Popular image file formats and their signatures
Source: Filesignatures.net

2.3.4 Signatures

To expand on the idea from **previous chapter**, we need to look at file signatures, sometimes also called "magic numbers". In a situation where a program can not recognize a file's format simply by looking at its extension, the file signature may be another option how to overcome this problem.
It is a short set of bytes usually found at the beginning of a file that indicates the format of the given file. There are no specific rules established by some authority, so formats can often have multiple different signatures and some formats can share a single signature as well. See examples in a table below.
Without an appointed authority no official database of signatures exists, so if we are interested in validating signatures in any way, we must find a reliable source of file signatures. Fortunately, there are a few sites that collect them.
File signatures table by Garry Kessler[4] is an ongoing attempt to track popular file formats and their respective signatures. This project goes back to the year 2002 and includes many helpful links to other sources relevant to the topic of file signatures.
Filesignatures.net[5] is a simple database which allows users to submit new file formats and their signatures to collaborate with others to improve its knowledge base.
As we can see, file signatures provide a quick and relatively reliable way to determine file's format, but it comes at a price. Not only do we need a database of file signatures, but we must also be able to open and read the contents of the file. In the context of file validation library, where we can be dealing with unknown and potentially malicious items, this must be taken into consideration.
It seems reasonable for our library, however, to have a test which compares a file's extension and signature. If they do not match, we can be dealing with a file which wears a mask of a different format and as such should be reported during the validation process.

2.3.5 Size

File size is probably one of the first things that comes to mind when dealing with file validation. Restricting them to a specific size can save our hard-drive from premature congestion, decrease the network usage or lower the load times of whatever we might want to do with such files afterward.
Inspecting file properties we quickly learn that every file has two values related to size - **size** and **size on disk**. These two values are almost always different, and it is important to understand this difference to find out the right way to validate these values in our library.
Files usually take more space on the disk than they need. The reason is that the file systems we regularly use – be it FAT32 (File Allocation Table), NTFS (New Technology File System), exFAT (Extended File Allocation Table) or some other – need to know where they can find their every file. Addressing every single byte located on the system would require an address table that would grow in size at the same rate as the data. So what is done instead is to have an allocation unit, the smallest addressable block of data there can be on a given file system. The size of the allocation unit can be changed to some degree, but on most file systems usually ranges between 4KB up to 32KB.[6]
This means the file can take a different amount of space on various computers. It is not a value that should be used during the process of file validation. Ideally, we want the result of the validation to be the same no matter where the files are stored.

2.3.6 Checksum

File checksum provides a means to verify the integrity of a file quickly. It is a short string which often has a form of a string or a number depending on the method we used to calculate such a checksum. The most frequently used methods of calculating file checksum is to use a common hash function such as MD5 or one of the SHA (Secure Hash Algorithm) variants (e.g., SHA-1, SHA-256, SHA-512).

```
$ md5sum README.txt
e0df5e079db0f1ec445e7825bbaea086 README.txt

$ sha1sum README.txt
73895333d62ff8a5a303cef6dfccdbfc6aaba969 README.txt

$ sha256sum README.txt
a73641cd891d971c475de11baed8c6316494082605717589d54e9654b22066ae
   README.txt
```

Code preview 2: Example of checksums computed via common UNIX functions

The nature of most hash functions is to give us wildly different results even when the source data was changed only ever so slightly. This is useful in situations where file content was tampered with between the time of creation and, in our case, time of validation. The author of the file can publish the checksum of the original, unmodified file and we, using the same hash function, can calculate and compare checksum for the file we received. Both of the checksums must be equal. Otherwise we are dealing with a modified file which could pose a security risk.
Another reason checksums can be useful is to check whether the file's contents didn't get corrupted due to an incomplete transmission or hard-drive failure. If parts of the file are missing or are broken in some way, it should not pass the validation since such a file can not be used afterward in any meaningful way.
It is worth noting that it is theoretically possible for an indefinite number of files to have an identical checksum. A situation in which two or more different inputs result in an identical result is called a **collision**. This is statistically improbable in reality, at least for well-designed hash functions. However, it is not impossible. Collisions have been documented both for MD5 (Message Digest algorithm)[8] and quite recently even for SHA-1[9]. These collisions did not happen by chance and were intentional. It is theoretically possible to have two versions of a file with an identical SHA-1 checksum where both of these versions look similar to the end user but one of them might contain malicious content. Both MD5 and SHA-1 are still frequently used, but efforts are being made to migrate to more secure algorithms (e.g., SHA-256, SHA-512).
It might be reasonable for our library to support validating files against multiple modern hashing algorithms and let the user decide whether security is the primary concern.
Checksum seems like a straightforward and reliable way to verify the integrity of a file, but it has one significant disadvantage. Without the original trusted checksum from a file's owner, this method cannot be used. Should users of our library require integrity verification of files coming from **unknown sources**, this cannot be used. It could, however, be considered for use cases where users know the original state of the file, work with it in some meaningful way without modifying it (e.g., transporting it) and then checking whether the file's checksum has changed.

2.3.7 Other meta-data

Most files carry other information about themselves as well. This can vary between different file systems, but there are some that are present almost everywhere. Timestamps, such as "date created", "date modified" and "date opened" on Windows NTFS file system or "date accessed" and "date modified" on most Linux system provide useful information about the life cycle of a file. The problem, however, is that these timestamps can easily be changed and are often subject to timezone and daylight saving time inconsistencies. Synchronizing these timestamps between two different systems can become complicated, which makes this information extremely unreliable and will not be a subject of our validation library.[10]

2.4 Common office file formats

Apart from validating generic meta-data of files, we would also like to verify the integrity of a file and check whether such a file can be opened or whether the data inside said file is corrupted. One option is to look at the file's checksum (more on that in chapter **Checksum** on page 10. This works only on the condition that we know the file's checksum before it got corrupted. Such information is often unknown to us.
Another option is to verify the file's integrity, which requires us to know how a given format stores data inside the file. If we do not know how to read and interpret data, we cannot say much. To overcome this issue we need a particular piece of software that knows how to work with a given file format. Such software can often open and work with this format. This can be another library or even application designed to open these files (e.g., Adobe Acrobat Reader for opening and reading PDF files or Microsoft Word for .doc and .docx formats).
Since we can not design programs that can understand all existing formats, we will look at a few common formats and how they can be verified through available libraries, frameworks or other means compatible with our C# library.

2.4.1 Microsoft Office family

Microsoft Office is a very popular and well-known set of applications. There is no need to introduce it. With around ten programs offered in the current standard desktop version, the first three that most people use and know are **Microsoft Word**, **Microsoft Excel** and **Microsoft PowerPoint** (with Microsoft Outlook right behind the three).
Files produced by these applications are arguably very often the subject of email attachments. Excel tables are frequently used to import or export data from applications. Comma separated value format (.csv) is one of the simplest ways how we can serialize data and Microsoft Excel is an ideal application to manage such a file format. And while such data can be saved in a .csv format that can be read even by simple text editors, the same data is often saved and used in formats native to Microsoft Excel (.xls or .xlsx).

2.4.1.1 Discrepancy between versions

Up until 2007, Microsoft Office data was saved in binary files with a relatively complex structure. This was due to the numerous features these files had to support. Going forward, this was not acceptable. Thus the OOXML (Open Office XML) came to be. This new standard was introduced in Microsoft Office 2007 and was very similar to the OpenDocument format for OpenOffice applications (more on OpenOffice in the next chapter **OpenOffice family** on page 13). An OOXML file is a ZIP file that contains XML (Extensible Markup Language) files along with the document's meta-data and object definitions. This allows Microsoft Office files to be smaller in size and backward compatible with older versions. It makes creating new features for Microsoft Office applications that much easier as well.[11] Before OOXML was introduced in Microsoft Office 2007, there was an attempt to store Office formats in a format based on XML. The Microsoft Office XML format was included in MS (Microsoft) Office XP to store Excel spreadsheets and later on this was implemented to Word documents in MS Office 2003, as well. However, this format was incompatible with older versions of these document types, and external add-ons (or third-party converters) had to be installed.

2.4.1.2 Verifying the integrity of Microsoft Office files

The most obvious solution is to try to open MS Office files with MS Office Suite and see whether this process fails or not. However, this requires a paid license and hundreds of megabytes in installed software. This is not a reasonable option we want to pursue if we are to develop a lightweight library. Microsoft does indeed provide an Interop library for .NET Framework Common Language Runtime which allows us to programmatically open and edit MS Office documents in MS Office applications via the COM (Component Object Model) interface. This is easy but requires Microsoft Office to be installed on the machine.
The new OOXML standard has an entire software development kit available, so there are dozens of libraries that can work with this format. One of them is an official library OfficeDev/Open-XML-SDK for C# that supports all three major extensions - .docx, .xlsx and .pptx. [12]
When it comes to older binary formats, it would seem there is very little support for them regarding existing libraries and frameworks (especially in the C# world). There is a library called **NPOI** that can deal with OOXML formats and .xls format for older Excel files. Distributed under the Apache License 2.0, this library could be used to handle this older format for spreadsheets should we need it.

2.4.2 OpenOffice family

While Microsoft was trying to find the right way to switch from binary office file formats to the XML format, another standard was being developed. The OpenDocument Format for Office Applications (ODF) was based on OpenOffice XML specifications that were supposed to be an open standard for all office documents. This was first

	Legacy MS Office	**Office Open XML**	**OpenDocument**
Document	.doc	.docx	.odt
Spreadsheet	.xls	.xlsx	.ods
Presentation	.ppt	.pptx	.odp

Table 2: Difference between office file formats extensions

introduced in the OpenOffice suite developed by Sun Microsystems and later also in the LibreOffice suite.
The OpenDocument Format works, in principle, much like the OpenOffice XML used by Microsoft Office, but they are not the same. It takes form of a ZIP file containing compressed files, where most (if not all) of them hold some data in XML format.
Much like the OOXML SDK (Software Development Kit) developed by Microsoft to work with their file types, OpenDocument Format is backed up by a similar project, the **Apache OpenOffice API Project**. It offers a *"language independent API (Application Programming Interface) which allows to program the office in different programming languages"*.[13]
One of very few C# implementations of this API is the **AODL (An Open Document Library)**. It offers tools to create and edit text and spreadsheet documents. The presentation documents seem to be left out, but with the GNU General Public License 2.0, this is the best option we have for verifying OpenOffice files.

2.4.3 Portable Document Format

PDF is another very common format found in almost every office. It was designed by Adobe to achieve a file format that would render the same document no matter which software or hardware was used to create it. PDF files are binary files consisting of four main parts - header, body, cross-reference table and trailer. The header contains basic information about the file. The body consists of all document objects including images, fonts, annotations, streams and other elements. The cross-reference table allows users to navigate between pages, chapters and other specific document objects. The trailer contains, among other things, the *EOF (End Of File)* string necessary for the PDF file to be processed correctly. Without it, the file is not complete. [14]
There are quite a few libraries designed to work with PDF files. One of them is the **itext/itextsharp** library, which provides a comfortable way to work with such files. It is distributed under the GNU Affero General Public License version 3, which means we are free to use it for verifying PDF files.

2.5 Popular Anti-Virus engines

The aim of this chapter is not to analyze which AV engine is better in terms of detection rate, but how their business model works and how they can be included in the file validation and verification process. If we are to work with files from an unknown source, open them and read their data, we must first be sure that they do not contain any malicious code that could harm the system where our validation library runs or anyone else's system down the road.
We do want to lock the library to any single AV engine. Ideally, we want an easy way for the user to setup access to their AV engine which can then be used by our library. We are also interested in whether or not engines regularly include a command line utility,

a program that can quickly be executed via our library to scan a file or a folder. We must also look at the output these programs produce as we will need to evaluate them if we are to correctly decide whether we can continue with the validation process or immediately terminate it to avoid any damage.

2.5.1 Avast

Avast offers multiple products related to malware detection, however, only paid alternatives offer a command line tool. As of March 2017, their non-free products start at around 30€/year. They support Windows, Mac, and Linux and also offer solutions for companies and servers.
The command line utility called **ashCmd.exe** can check files or folders for potential malware, much like a regular desktop AV program does. It accepts many different parameters to set up the scan. These options can prove useful, but we are mostly interested in how we can use such a tool to scan a file and handle the output result. In the example below, you can see how this utility can be used to test all files within a given folder and print the output of such a scan on the standard output.

```
> ashCmd.exe "C:\Users\Documents\Folder" /_

C:\Users\Documents\Folder\test1.txt OK
C:\Users\Documents\Folder\test2.txt OK
C:\Users\Documents\Folder\test3.txt OK
# ----------------------------------------------------------------
# Number of tested files: 3
# Number of tested directories: 1
# Number of infected files: 0
# Total size of tested files: 0
# Database VPS: 170517-14, 17.05.17
# Total runtime of the test: 0:0:0
```

Code preview 3: Use of Avast's ashCmd.exe in a Windows command line

The ashCmd.exe tool uses the following return values:

- 0 - No virus detected.
- 1 - Virus detected.
- 2 (or higher) - An error occured.

2.5.2 AVG

The free version of AVG already includes a command line tool **avgscana.exe** (or **avgscanx.exe** for 32-bit systems). However, their malware support does not include Linux systems (they do talk about Android support, but phones are not relevant to this analysis), which might be inconvenient for some users.

```
> avgscana.exe /SCAN="C:\Users\Documents\Folder"
```

Code preview 4: Use of AVG's avgscana.exe in a Windows command line

The avgscana.exe tool uses the following return values:

- 0 (RETURNCODE_OK) - Everything is OK.
- 1 (RETURNCODE_USERSTOP) - User iterrupted the scan.
- 2 (RETURNCODE_ERROR) - Error during the scan (e.g. when an incorrect parameter is used).
- 3 (RETURNCODE_WARNING) - Warning during the scan.
- 4 (RETURNCODE_PUPDETECTED) - Potentially Unwanted Program detected.
- 5 (RETURNCODE_VIRUSDETECTED) - Virus detected.
- 6 (RETURNCODE_PWDARCHIVE) - Password-protected archive found.

2.5.3 Eset

Eset includes their command line utility in every version of their product. However, all of them are monetized. They also provide solutions for companies and their servers, and they support all three of the major operating system families. In the example below, we scan a given folder and on a second try we also output the result into a file specified by the */log-file* parameter.

```
> ecls.exe /base-dir="C:\Users\Documents\Folder"
> ecls.exe /base-dir="C:\Users\Documents\Folder" /log-file=c:\ecls.txt
```

Code preview 5: Use of Eset's ecls.exe in a Windows command line

The ecls.exe tool uses the following return values:

- 0 - No threat found.
- 1 - Threat found and cleaned.
- 10 - Some files could not be scanned (might be threats).
- 50 - Threat found.
- 100 - An error occurred.

2.5.4 ClamAV

ClamAV is a fairly popular **open source** AV engine that supports all the mainstream operating systems. While some may question the quality and detection rate of such an engine, the fact that an open source software has certain advantages which might be important to a specific set of users cannot be ignored. In the example below, we first update the virus definitions and then we recursively scan a given folder.

```
$ sudo freshclam
$ clamscan -r /home/user/Folder

----------- SCAN SUMMARY -----------
Known viruses: 0
Engine version: 0.99.2
Scanned directories: 1
Scanned files: 5
Infected files: 0
Data scanned: 5.13 MB
Data read: 5.13 MB (ratio 1:1)
Time: 0.503 sec (0 m 0 s)
```

Code preview 6: Use of ClamAV's clamscan in a Linux terminal

The clamscan tool uses the following return values:

- 0 - No virus found.
- 1 - Virus(es) found.
- 2 - Some error(s) occured.

2.5.5 Windows Defender

Windows Defender is an AV engine available for free as a part of Windows 8 or newer versions. In earlier versions, it was also known as Microsoft Security Essentials which was at its core a very similar engine. Now it is pre-installed on most Windows systems. What's more is that it also contains a command line utility which supports scan options much like all the tools we've discovered in previous chapters. The only real disadvantage is the lack of support for any other operating system other than Windows. In the example below, we scan our folder with Quick scan identified by the *-ScanType 1* parameter.

```
> MpCmdRun.exe -Scan -ScanType 1 -File "C:\Users\Documents\eicar.txt"

MpCmdRun: Command Line:
MpCmdRun.exe -Scan -ScanType 1 -File C:\Users\Documents\eicar.txt
Start Time: Mon Apr 26 2017 09:05:21
Start: MpScan(MP_FEATURE_SUPPORTED, dwOptions=3, path
   C:\Users\Documents\eicar.txt, DisableRemediation = 0)
MpScan() started
file C:\Users\Documents\eicar.txt is infected.
MpScan() was completed
Finish: MpScanStart(MP_FEATURE_SUPPORTED, dwOptions=16385)
Finish: MpScan(MP_FEATURE_SUPPORTED, dwOptions=16385, path
   C:\Users\Documents\eicar.txt, DisableRemediation = 0)
Scanning C:\Users\Documents\eicar.txt found 1 threats.
MpScan() has detected 1 threats.
Start: MpCleanStart()
MpCleanThreats() started
MpCleanThreats() started for action Remove on threat 2147519003
MpCleanThreats() completed for action Remove on threat 2147519003
MpCleanThreats() was completed
Finish: MpCleanThreats()
MpCmdRun: End Time: Mon Apr 26 2017 09:05:29
```

Code preview 7: Use of Windows Defender's MpCmdRun.exe in a command line

The MpCmdRun.exe tool uses the following return values:

- 0 - Clean. No malware is found, or malware is successfully remediated, and no additional user action is required.

- 2 - Infect. Malware is found and not remediated or additional user action is required to complete remediation or there is an error in scanning.

	Windows	**Linux**	**Mac**		**CLU**	**Price**
Avast	X	X	X		X	~30€
AVG	X		X		X	Free
Eset	X	X	X			~40€
ClamAV	X	X	X		X	Open source
Windows Defender	X				X	Free

Table 3: Summary of analyzed AV engines

2.5.6 Evaluation of Anti-Virus engine integration options

All of the analyzed engines offer a **CLU (Command Line Utility)**, which should be the way to go regarding our library. Granted, some of them are hidden behind a pay-wall, but this depends on the end user, and we should not limit them in their choice. If we are to automate this process, we will need to use these utilities, the options they provide and be compatible with any one of them.
Fortunately for us, exit status codes of the AV utilities we have discussed follow general rules and best practices of software development. Zero indicates no problem detected, positive integer indicates an error and sometimes negative integers can indicate a warning.[15][16]
The mentioned utilities follow these rules, and we can expect most of the other, unexplored, AV engines to follow them too. If we are to evaluate the malware scan's result quickly, we can either parse the text report or only work with the status code. As we can see from the **preview 3** (page 15), **preview 6** (page 17) and **preview 7** (page 18) the output can vary significantly. Rather complex definitions are required to parse these outputs, perhaps in the form of regular expressions, which will probably be different for every engine. The other solution on the other hand, is universal and should work with most (if not all) AV programs. In the end, we only need to know if we can safely begin the validation process. This does not mean we should ignore the output produced by AV tools. The best option would probably be logging the output or directing it to the standard output (or both).

2.6 Open source license - Apache License 2.0

The product of this thesis, the source code of our validation library, will be distributed under the **Apache License 2.0**. It is one of the more permissive open source licenses available as it allows anyone to use our code for commercial purposes, to modify it as they see fit and to distribute it freely. On the other hand, we can not be held responsible should something go wrong in case of an incorrect integration of the code. All that is required from anyone using our program is that they always include the license, copyright and that they describe any significant changes made to the library.[18]
Another significant advantage of this license is that it is more often than not compatible

with other licenses. This makes finding the right libraries to work with easier. Similarly, if people decide to integrate our library into their project, they should not be, in most cases, limited by compatibility issues. For example, while we cannot incorporate a GPLv3 licensed project in an Apache 2.0 licensed project, it can be done the other way around (an Apache 2.0 licensed project in a GPLv3 licensed project). Since the end goal is that our library is used in other projects, this is exactly the issue we are trying to avoid.[19]

2.6.1 Popular software under Apache License 2.0

According to an article published on GitHub in 2015, Apache 2.0 is the third most popular license right after MIT and GPLv2. [20] Softwares and applications like .NET Compiler Platform, IntelliJ IDEA Community Edition or most of the Apache products are distributed under this license. Their source code is very often available via GitHub. Other products with the Apache License 2.0 include:

- Android Studio (Android development platform)
- Docker (software containers management)
- Gradle (alternative to build systems like Ant or Maven)
- Swift (the programming language)

Since our goal is to have a library that is easy to share and simple to integrate into other projects, choosing Apache License 2.0 is almost a necessity. Other possible options include MIT and GPLv2 as both of these open source licenses also come with very few restrictions.

2.7 Logging

Logging is a necessary part of every software, and our project should include it as well. There are a few libraries already available to us to choose from, which provide all the necessary functions we could possibly need.
We will be using the **NLog** library, that is distributed under the BSD 3-Clause License and is compatible with the Apache License 2.0 we selected for our library in the previous chapter in section **Open source license** on page 19.
The other viable option would be the **log4net** logging framework, another popular option among C# projects. Both would satisfy our needs, but as we can pick only one of them, we will be choose NLog. This is based purely on personal preference, and while we could discuss the advantages and disadvantages of both of these libraries, it is not necessary as we only need basic logging functions where the difference is minimal. Should anyone want to use a different logging library, they are free to fork the project and change it.

Part 3
Library implementation

In this chapter we will describe specific steps used in the implementation of our library in C# language, including the steps to implement a user-friendly interface and thorough validation functions.

3.1 Fluent interface

Fluent interface provides an alternative way how we can manipulate objects compared to how we would normally do it. To better illustrate this, we will refer to an example taken from Martin Fowler's article about Fluent interface, where it was first described in late 2005. In his examples, he compares order creation for a customer using the standard methods and the proposed "fluent" methods.[21]

```
private void makeNormal(Customer customer) {
        Order o1 = new Order();
        customer.addOrder(o1);
        OrderLine line1 = new OrderLine(6, Product.find("TAL"));
        o1.addLine(line1);
        OrderLine line2 = new OrderLine(5, Product.find("HPK"));
        o1.addLine(line2);
        OrderLine line3 = new OrderLine(3, Product.find("LGV"));
        o1.addLine(line3);
        line2.setSkippable(true);
        o1.setRush(true);
}
```

Code preview 8: Standard way of creating customer orders

```
private void makeFluent(Customer customer) {
        customer.newOrder()
                .with(6, "TAL")
                .with(5, "HPK").skippable()
                .with(3, "LGV")
                .priorityRush();
}
```

Code preview 9: Fluent way of creating customer orders

We can see that the context is managed by the return values of every method used, so it can be used and modified afterward again - we are always working with the same object. This can be achieved by letting the methods always return the object that owns them. An example below shows how this is done in our library. Notice how the instance of the Inspector class is always returned so it can be used repeatedly with modified values.

```
namespace Verifile {
  public class Inspector {
      ...
    public Inspector MinSize(int kilobytes) {
      stepSize.MinSize = kilobytes;
      return this;
    }

    public Inspector MaxSize(int kilobytes) {
      stepSize.MaxSize = kilobytes;
      return this;
    }

    public Inspector EnableSignatureTest() {
      stepSignature.Enable();
      return this;
    }

    public Result Scan() {
        // starts the validation process
    }
      ...
  }
}
```

Code preview 10: Implementation of the Fluent interface in our library

The other library classes do not look like this. It is only the Inspector class that users will work with and therefore it is the only class following the fluent interface guidelines. Notice how the Scan() method does not return an Inspector instance, but instead an instance of a Result class. It should always be the last method in the chain and should launch the validation process. There is no need for the Inspector to be returned, the user might be more interested in a Result object encapsulating the results of the validation. There are obviously more options, but we have decided to go with this design choice.

3.1.1 Disadvantages of Fluent API

So far we have discussed what advantages the fluent interface brings to the table. The exposed interface is easy to use and can often be written as a single line of code. This does have its drawbacks and having one long chain of method calls can make certain things difficult (if not impossible).
A debugging chain of commands can be problematic as debuggers often do not allow setting breakpoints in the middle of the method chain and stepping through such a code can be confusing as well.
Logging faces a similar issue when we want to log the state of the object between the methods contained in the chain. To solve this, one must split the chain into more lines, which basically puts an end to the idea of the fluent interface.

3.2 Structure of the library

The functionality will be divided into a couple of smaller libraries. This way we can give users a library with the core functionality (generic validators for size, extension, checksum, etc.) in one small package and in case they are interested in verifying the integrity of specific formats, they can download one or more of our modules intended for this format. They often come with additional dependencies, and it is preferred to have users download only those they will really be using.
During the analysis of **Common office file formats** on page 12, we identified libraries which will assist us in verifying common office formats. We will divide the package into the following libraries:

- **Main library** - Contains most of the scan process logic, exposes library's interface to users and can load all available validation modules
- **Core library** - Provides basic functions and classes for all other libraries
- **Optional libraries**
  - Image validation module
  - Microsoft Access database validation module (.accdb)
  - Microsoft OpenXML validation module (.docx, .pptx, .xlsx)
  - Microsoft binary format validation module (.xls) - the tools available to us do not support .doc or .ppt format (for reasons explained in the chapter about **Microsoft Office files verification** on page 13).
  - ODF validation module (.ods, .odt)
  - PDF validation module (.pdf)

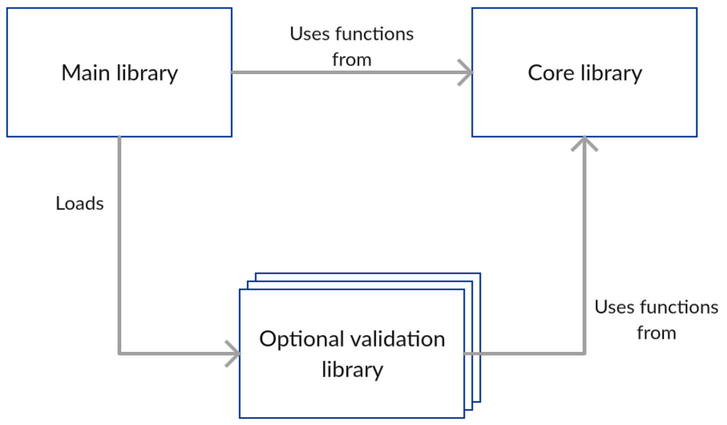

Figure 1: Dependency of library modules

3.2.1 Validation steps

Every validation step is a class inheriting basic properties from the parent class **Step**. We understand a validation step as a module checking a single file property (e.g. size). This way we can assign different error codes (or other properties) to each and every one of them. We will be releasing the library with the validation steps listed below. What they do and how they work is described in the chapter about the **Inspector interface** on page 27.

- **Generic validation steps** (usable for any file type)
  - AV scan - Enables users to integrate their installed AV software. This validation step would be the first to run in order to avoid a possible infection from malware contained in one or more of the scanned files.
  - Checksum
  - Extension
  - Signature
  - Size
  - VirusTotal - In case users are interested in some form of protection against malware, but are unable to have an AV installed, they can use an online solution called VirusTotal (more about **VirusTotal** on page 31).
- **Format specific validation steps used for verifying file's integrity** (whether the file can be opened or not). These are divided into multiple optional libraries.
  - ACCDBValidator
  - DOCXValidator

- ImageValidator (currently validating the two most common raster image formats - JPEG and PNG)
- ODSValidator
- ODTValidator
- PDFValidator
- PPTXValidator
- XLSValidator
- XLSXValidator

Another advantage of splitting validations steps in this manner is that we can easily create new validation steps. All that is required is to build a new class extending the parent **Step** class and implement whatever we want from this new module. The library keeps a list of validation steps and runs them sequentially, so in order to integrate this new step with the library, we only need to add it to the list with the rest of our validation steps. This is when we want to make a rather permanent change to the library.
There is another way users can integrate their own validation steps, and that is via the Inspector class (more on that in the next section). In short, this other method does not require overwriting the core of the library. Users can implement their validation steps wherever they want and then just pass it to the library via the Inspector's method **AddCustomStep(Step)** (see Code preview 13 on page 29).

```
public class Step {

        // Every step can have its own error code.
        public virtual int ErrorCode { get; set; } = Error.Generic;
        ...
        // Sets up validation step for execution.
        public virtual void Setup() {...}

        // Starts the step validation process.
        // Should be overwritten for custom behaviour.
        public virtual void Run() {...}

        // Prints the results of the step to user and returns
        // the appropriate response code.
        public virtual int Summary() {...}

        // Returns the step and it is essential variables to their
        // default state.
        public virtual void Cleanup() {...}
}
```

Code preview 11: Overview of variables and methods exposed by Step class

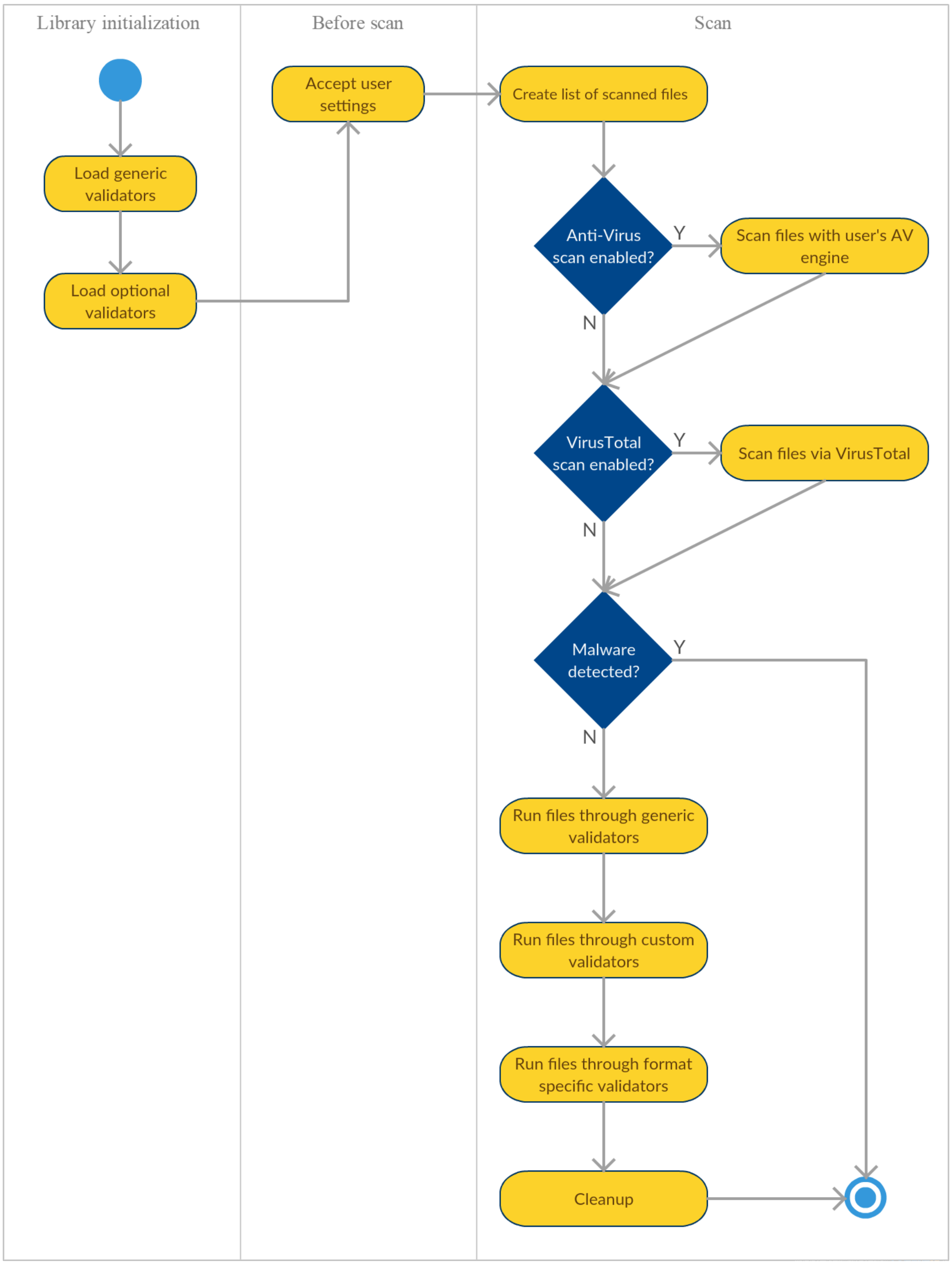
Library initialization
Before scan
Scan
Load generic validators
Load optional validators
Accept user settings
Create list of scanned files
Anti-Virus scan enabled?
Y
Scan files with user's AV engine
N
VirusTotal scan enabled?
Y
Scan files via VirusTotal
N
Malware detected?
Y
N
Run files through generic validators
Run files through custom validators
Run files through format specific validators
Cleanup

Figure 2: General overview of the library's scan process

3.2.2 Inspector

The Inspector will be the class users will use to set up and work with the library. It will offer a Fluent interface with all the functions that can enable or disable validation steps as the user will deem necessary.

3.2.2.1 Inspector's interface

The following list contains all the functions the library offers and which can be used to set it up. This list is also a part of the documentation found in the official GitHub repository **hanzik/verifiler**,[1] where it can help users quickly familiarize themselves with the library's interface. To understand how the interface should be used, a quick look at the Code preview 12 on page 27 should make everything clear.

```
string folder = "C:\path\to\directory"

Inspector = new Verifiler.Inspector();
Result result = Inspector.AddExtensionRestriction(".jpg")
                                 .MinSize(500)
                                 .MaxSize(2000)
                                 .EnableSignatureTest()
                                 .Scan(folder);

if (result.Code() == VerifilerCore.Result.Ok) {
    ...
}
```

Code preview 12: Example use of the Inspector class

- EnableAV(string pathToExecutable, string parameters)
- DisableAV()

Runs the anti-virus engine installed on your machine and scan the files before they are analyzed by the library. Most of the available anti-virus engines include a console application, which is the application you are looking for.

[1]Official Verifiler repository at https://github.com/hanzik/verifiler

- EnableVirusTotal(string apiKey)
- DisableVirusTotal()

Sends the files to VirusTotal and reports files with **Error.Fatal** if at least 10% of AV engines find the file suspicious. Should any of the files be recognized as the cause of this, the scan process will terminate.

- AddAllowedChecksum(string md5checksum)
- RemoveAllowedChecksum(string md5checksum)
- AddAllowedChecksums(string[] md5checksums)
- RemoveAllowedChecksums(string[] md5checksums)

Adds or removes MD5 file checksum from the whitelist. If the whitelist of checksums is empty, this step will be skipped. If it is not empty, files with checksums not on the whitelist will be reported with **Error.Checksum**.

- AddExtensionRestrictions(string[] types)
- AddExtensionRestriction(string type)
- RemoveExtensionRestrictions(string[] types)
- RemoveExtensionRestriction(string type)

Similar to checksum whitelist. Adds or removes extensions from the whitelist. If the whitelist is empty, this step will be skipped. If it is not empty, files with extensions that are not on the whitelist will be reported with **Error.Extension**.

- MinSize(int kilobytes)
- MaxSize(int kilobytes)

Sets up a minimum and/or maximum size in kilobytes of scanned files. Files that break this rule will be reported with **Error.Size**.

- EnableSignatureTest()
- DisableSignatureTest()

Compares a file's extension with file's magic number located in the header. Files that break this rule will be reported with **Error.Signature**.

- EnableFormatVerification()
- DisableFormatVerification()

Enables verification of a file's integrity. This requires some of the optional packages to be installed (depending on which file extensions you want to verify).

- AddCustomStep(Step)
- RemoveAllCustomSteps()

You can add your own validation steps. In the example below, we create a validation step which checks the name of every file and lets through only those that start with the string "abc". This means that "abctextfile.txt" will pass, but "defimage.jpg" will not. Failed tests will return **Error.Generic**, or if you override the ErrorCode, it can return whatever code you want. It is preferred that the error code is a positive integer.

```
inspector.AddCustomStep(new StartsWithStep("abc"))
         .Scan(GetTestFolderPath());

class StartsWithStep : VerifilerCore.Step {

    public override int ErrorCode { get; set; } = Error.Generic;
    private readonly string startString;

    public StartsWithStep(string startString) {
        this.startString = startString;
        Enable();
    }

    public override void Run() {
        foreach (var file in GetListOfFiles()) {
            string name = Path.GetFileName(file);
            if (name.StartsWith(startString)) {
                ReportAsValid(file);
            } else {
                ReportAsError(file, name + " does not start with " +
                    startString);
            }
        }
    }
}
```

Code preview 13: Implementing a custom validation step in runtime

3.2.3 Optional library loader

As we mentioned before, it is difficult to determine whether or not a file's data is corrupted and that the A file cannot be opened. Our format specific validation steps are using the libraries mentioned in the **Common office file formats** chapter (page 12). This can bring quite large libraries to the table, and it is necessary to minimize the final size of our library as much as possible.
By dividing the library into smaller libraries, we can enable users to always download only those they will truly need. For example, should a user only want to validate PDF files, they should not be required to download libraries related to OpenDocument file formats.
This is where the **Optional library loader** comes in. It is a class that can load optional libraries at runtime using the C# Assembly class. All it needs is an XML configuration file which contains names of all the optional libraries. The advantage is that if we create a new optional library supporting a new file format, we do not have to rewrite half of the core library, we only have to edit the configuration file.
All the user needs to do is download optional libraries he or she wants to use, make sure they are placed in the project's folder, and they are ready to go. By using the packaging manager NuGet, the library is put in the projects folder by default, which makes the whole process even more straightforward (although using NuGet is not required).

3.2.4 Result object

The initial idea was that our library would only return a status code in the form of an integer. It was quickly discovered that some additional information will be necessary. As such, the Result object returned by the **Inspector.Scan()** method contains the following information:

- Response code - A positive integer in case of an error, zero otherwise.
- List of executed steps - List of names of executed steps.
- List of valid files - List of files which passed all the validation steps.
- List of invalid files - List of files which did not pass at least one validation step accompanied by a status code to help identify where the failure occurred.

3.3 External dependencies

One of many goals of this thesis was to make the library as independent of other libraries as possible. This was not difficult, as the .NET framework provides many useful functions that we can use instead of importing them from external sources. There are, however, a few exceptions where using an external library is a necessary trade-off.

This chapter will exclude the libraries dedicated to verifying the integrity of file formats we work with as they are mentioned in the analysis of **Common office file formats** on page 12.

3.3.1 VirusTotal.NET

We have already gone through a good deal of Anti-Virus engines in our analysis of **Popular Anti-Virus engines** (page 14), but there is a use case which we have not covered. The user might not have an AV engine installed, yet he might want to use another popular alternative for detecting malware. VirusTotal allows users to upload files on their server which are then in turn evaluated by over 60 isolated AV engines including all the programs we discussed. One engine can always make a mistake, therefore a report from dozens of them is that much more important.
If a popular file, people can even find comments and vote for the given file (identified by SHA256 checksums). It offers a whole new perspective of malware detection.
The obvious disadvantage for us is that for a file to be analyzed, it must first be sent over to their server. This can be unacceptable for some of our users and is therefore only considered as one of many options as to how to approach malware detections of the file we are about to validate. The other disadvantage is that as of 1st of April 2017, the public API which they provide for free is limited to only four requests per minute, which might not be enough for some people. Additionally, the public API key can not be used for commercial purposes, but people can request a private key where some of these restrictions can be lifted (depending on what it will be used for). Some may say these rules are rather strict, but it is important to understand that this should serve only as a backup solution and as a demonstration of how easily new validation methods can be integrated into the rest of the library.
VirusTotal's API is also fully implemented in C# as one of their libraries and can be integrated within our library through the NuGet package manager. With Apache License 2.0, this is a viable library to be included.[23]

3.3.2 NLog

Based on the discussion in chapter **Logging** (page 20), we will be using the **NLog** library for logging. It offers both basic and advanced logging features, and while we do not need much from the library, users might need more options at their disposal. Using NLog is rather intuitive, as shown in example 14 (page 32).

```
using NLog;

namespace VerifilerCore {

    public class Step {

        private static Logger logger =
           LogManager.GetCurrentClassLogger();

        public void Enable() {
            logger.Debug("Step {0} enabled", Name);
            enabled = true;
        }

        ...

        protected void ReportAsError(string file = null, string msg =
           null) {
            logger.Error("Error reported by step {0} for file {1} with
               message {2}", Name, file, msg);
            ...
        }
    }
}
```

Code preview 14: Logging with NLog

Part 4
Testing

In this chapter we will describe how we have made sure our library works as intended. Unit tests will be implemented to provide at least a basic certainty that each and every module of our library validates files according to the rules we have set up in the previous chapter. Performance tests will also be conducted to verify that our library can run and be used in a reasonable time frame. These tests should also help us understand how the number of files affects the library's performance.
Thanks to unit tests we do not have to create an external application using the library in order to test all of its functions. The interface allows us, among other things, to test parts of our library independently.

4.1 Unit Testing in Visual Studio for C#

Since most of the development was done in Visual Studio, there are many integrated tools at our disposal, such as the Microsoft C# unit testing framework **Microsoft.VisualStudio.TestTools.UnitTesting**. While there are other options like the **NUnit** or **xUnit** frameworks, we will work with the UnitTesting framework provided by Microsoft as it offers all we need and will result in one less external dependency for our library to work with.
As you can see from the example 15 on page 34, test classes in the UnitTesting framework are denoted by the [**TestClass**] annotation and every unit test, technically a method of such class, is denoted by the [**TestMethod**] annotation. For people familiar with Java's JUnit framework, this is somewhat similar. Visual Studio can use the information from annotations to visualize testing via its Test Explorer, which makes execution of our tests extremely simple.
Our goal is to create a test class for every module our library provides - one for the file size module, one for the file signature module, one for the Microsoft Office module, etc. Each one of these test classes should cover most of the possible configurations the module offers. We understand "unit" as a single validation step described in section about **Validation steps** on page 24.
The tests will be designed as if a user was trying to validate files using the interface the library provides via the Inspector class. The tests will invoke it just as the user would, set it up to enable and trigger the module we are trying to test and try all possible inputs that might break the library.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace VerifileTests {

    [TestClass]
    public class SizeTests : Tests {

        [TestMethod]
        public void TestMinSize() {

            var testFile = FileCreator.CreateFile(DefaultFileSize,
               ".txt");

            /* MinSize requirement is met. */
            Inspector = new Verifile.Inspector();
            Result = Inspector.MinSize(DefaultFileSize -
               1).Scan(testFile);
            Assert.AreEqual(Verifile.Result.Ok, Result.Code());

            /* MinSize requirement is not met - file is smaller. */
            Inspector = new Verifile.Inspector();
            Result = Inspector.MinSize(DefaultFileSize +
               1).Scan(testFile);
            Assert.AreEqual(Verifile.Error.Size, Result.Code());
        }
          ...
    }
}
```

Code preview 15: Unit test of MinSize() function

We will also use the annotations [**TestInitialize**] and [**TestCleanup**] which denote methods that are run before and after every test method. This is important for us because we need to create a temporary folder in which we will work with the files that are to be validated. For every test method to be truly independent, this is a necessary function. While the TestInitialize method takes care of creating a temporary directory, the TestCleanup method deletes such a folder with all of its contents, so we do not waste indispensable space while repeatedly executing dozens of tests.
[**ClassInitialize**] and [**ClassCleanup**] annotations are available to us as well, but we will not be using them. In contrast to TestInitialize, ClassInitialize denotes a method that should be invoked when the test class is first initialized before any test from that test class is run. Similarly, the ClassCleanup denotes a method that is to be invoked after all the test methods have finished. There are many other annotations provided by the framework, but we will not need them and they are therefore left unmentioned.

4.2 Generating a set of test files

For our tests to have any meaningful results, we need to have a reliable set of input files. There are two obvious ways to approach this. The first method includes implementing functions which can generate files in runtime with requested extension, size, and other metadata. The other method includes using a prepared set of files which were created using standard tools. While the first option allows us to simply create files of arbitrary size and extension, the other method makes it possible to simulate more delicate defects a file can have. For example, if we want to test our PDF validation, it makes more sense to use a previously prepared real PDF file rather than generating it via a sophisticated function.
We will use both of these methods as there are situations where one is more useful than the other and vice versa.

4.2.1 Infected files

We must not forget about the integration of AV engines that our library supports. In order to test how such programs behave, we need a file that can trigger AV scans, while still being harmless to whoever is in possession of this file. Fortunately, there are very specific files that are universally recognized as malware while still being harmless. One of these is **EICAR Standard Anti-Virus Test File** which triggers most available AV engines.
The file is a legitimate DOS (Disk Operating System) program and produces sensible results when run (it prints the message "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!"). It is also short and simple – in fact, it consists entirely of printable ASCII characters, so it can be easily created with a regular text editor. Any anti-virus product that supports the EICAR test file should detect it in any file provided that the file starts with the following 68 characters, and is exactly 68 bytes long.[24]

4.2.2 Corrupted files

For format specific validations, we need a sample of both valid and invalid files of a given format. First we create valid files - images via MS Paint, MS Office files via MS Office 2010 (both the legacy formats and the Open Office XML formats), OpenOffice files via LibreOffice 5.3 and Access Database file via MS Access 2010.
To create a corrupted alternative of these files, we copy these files, open then in a hex editor and rewrite a very small portion of the file's contents to make these files unreadable by programs that originally created them. This was done through trial and error as certain parts of some files can be changed almost entirely without making the file unreadable. The file's signature was always left intact (the first few bytes of the file) as this would most likely not corrupt the file contents itself as much as confuse the

program trying to open it.

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Code preview 16: EICAR Test file contents

4.3 Summary of implemented unit tests

In the table **Overview of implemented unit tests** (page 37), you can find an overview with a list of names and short descriptions of implemented unit tests. If there was an obstacle in implementing a given unit test, an explanation will also be provided. The tests roughly follow the structure of the library. Every test is often linked to one specific validation module, testing every possible outcome this module might produce.

4.4 Performance tests

To verify that our library can be used in a standard production environment, performance tests are to be carried out. We will measure how the amount of files impacts the library's runtime.
While the AV scan plays a significant role in our validation, it is worth noting that the selected anti-virus engine and the platform may yield significantly different results for every user. We will leave it out of our two main performance tests. We will run one additional test with it, to get a clear picture of how our library might perform with everything turned on.
It is expected that the length of the validation will increase with additional files in a somewhat linear fashion.

4.4.1 Preparation

As our library provides us with both the validation of generic file formats and the validation of specific file formats we will run two separate tests:

1. Test of files where we trigger only generic validations.
2. Test of files where we trigger only format specific validations.
3. Bonus test where we enable our AV software Avast Internet Security and trigger all validations. We will not infect any of the files as that would terminate the scan and affect the total runtime.

Test module	Description	Note
AVTest	Tests the library's integration with a set AV engine. An EICAR test file is used to trigger a false positive alarm.	This test needs to be configured on a per system basis as the installed AV engine may vary.
CustomStepTest	Implements a custom validation module and tests whether the library uses it correctly.	
ExtensionTest	Tests all possible configurations of allowed extensions as restrictions can be added/removed one by one or in a single batch.	
ChecksumTest	Creates a few arbitrary files, calculates their checksum and passes these values to the library to check whether the validation will fail or not.	
InputTest	Tests the behavior of the library if an invalid path (or no path at all) to files is provided. This includes tests which provide files with path or a name that contains forbidden characters.	
SignatureTest	Tests whether the Signature module can read and validate magic numbers at the beginning of scanned files.	While the previous tests used generated empty files, this test needs real files with real file signatures.
SizeTest	Tests setting up the combination of both lower and upper bounds on an allowed file size.	
VirusTotalTest	Tests integration with VirusTotal. An EICAR test file is used to trigger a false positive.	
ACCDBTest	These tests take 2 files of a relevant file format - one valid and one corrupted. They first run the scan with only the valid file and expect the scan to come clean. They then add the corrupted counterpart and run the scan again.	
ImageTest		
LegacyOfficeTest		
OpenOfficeTest		
OpenXMLTest		
PDFTest		

Table 4: Overview of implemented unit tests

This will allow us to compare both of these independent parts of the library. Tests will be executed upon 10, 50, 100, 500, 1 000, 5 000, 10 000 and 20 000 files. We will run each test 5 times and calculate the average from that. We will also calculate the time spent per file for every test to get a clear understanding of the time complexity our library has. To achieve accurate results, we will work with same files we use in our unit tests as they mimic real files typically found in a production enviroment. Tests will be performed on a computer with the following specifications:

- Intel i7-3612QM 2.10GHz
- Kingston HyperX FURY SSD 240GB
- 8GB RAM
- Windows 7 (64-bit)
- Visual Studio 2015
- .NET Framework 4.5.2

Time will be measured from the moment the library is initialized via the **new Inspector()** call up until the **Scan()** method finishes executing. A built-in **Stopwatch** class will be used to measure the time. We will preheat cache by running the test twice, measuring only the time of the second scan. We will kill any test running for more than 100 seconds.
We hope to see the execution time increase in a linear fashion as we scan more and more files. The generic validations may take longer as all the files go through all of them compared to the format specific validations where each file goes through only one of them. However, we should account for the fact that the format specific validations are trying to open and read all the files with their respective library, which may take significantly longer. The generic validations are mostly just reading the file's metadata, which does not always require opening it.

```
var timer = new Stopwatch();
timer.Start();
Inspector = new Verifiler.Inspector();
Result = Inspector.MinSize(1)
                  .MaxSize(2000)
                  .AddExtensionRestriction(".odt")
                  .AddExtensionRestriction(".pdf")
                  .AddExtensionRestriction(".accdb")
                  .AddExtensionRestriction(".xlsx")
                  .EnableSignatureTest()
                  .Scan(GetTestFolderPath());
timer.Stop();
```

Code preview 17: Setup of the generic validations test

```
var timer = new Stopwatch();
timer.Start();

Inspector = new Verifiler.Inspector();
Result = Inspector.EnableFormatVerification()
                  .Scan(GetTestFolderPath());

timer.Stop();
```

Code preview 18: Setup of the format specific validations test

```
var timer = new Stopwatch();
timer.Start();

Inspector = new Verifiler.Inspector();
Result = Inspector.MinSize(1)
                  .MaxSize(2000)
                  .AddExtensionRestriction(".odt")
                  .AddExtensionRestriction(".pdf")
                  .AddExtensionRestriction(".accdb")
                  .AddExtensionRestriction(".xlsx")
                  .EnableAV("C:\\Program
                     Files\\AVASTSoftware\\Avast\\ashCmd.exe",
                     GetTestFolderPath() + " /_")
                  .EnableSignatureTest()
                  .EnableFormatVerification()
                  .Scan(GetTestFolderPath());

timer.Stop();
```

Code preview 19: Setup of the bonus test with AV scan

4.4.2 Results

The result of each test can be found in the tables located on the next page (page 40). They confirm what we suspected. For a small number of files (100 or less), the library's initialization takes most of the time, and the generic validations themselves do not significantly affect the total execution time. Going up (1000 or more files), we can see the time increase somewhat linearly in both of the tests. The format specific validations take significantly longer to run, for reasons we have already discussed. To

Files scanned	Total time [ms]	Average time per file [ms]
10	22	2.2
50	33	0.66
100	48	0.48
500	152	0.304
1000	267	0.267
5000	1411	0.282
10000	2780	0.278
20000	5798	0.29

Table 5: Result of the generic validations test

Files scanned	Total time [ms]	Average time per file [ms]
10	168	16.8
50	842	16.84
100	1656	16.56
500	8640	17.28
1000	16726	16.726
5000	86078	17.216
10000	-	-
20000	-	-

Table 6: Result of the format specific validations test

discover whether the file is corrupted or not, we have to try to open it. Opening files (and reading its contents) does indeed take much longer. We had to kill the tests of 10 000 and 20 000 files as they ran for far too long.
The bonus test, where we also used our installed AV software, confirms what we already learned in the first two tests. However, it is interesting to see that starting up the AV program is very time-consuming. This is seen most clearly in the plot of average times per file (page 41).
One of the most important things we have learned from these tests is that our library runs with a linear time complexity.

Files scanned	Total time [ms]	Average time per file [ms]
10	1802	180.2
50	2515	50.3
100	3263	32.63
500	10245	20.49
1000	19500	19.5
5000	96871	19.374
10000	-	-
20000	-	-

Table 7: Result of the bonus test with AV scan

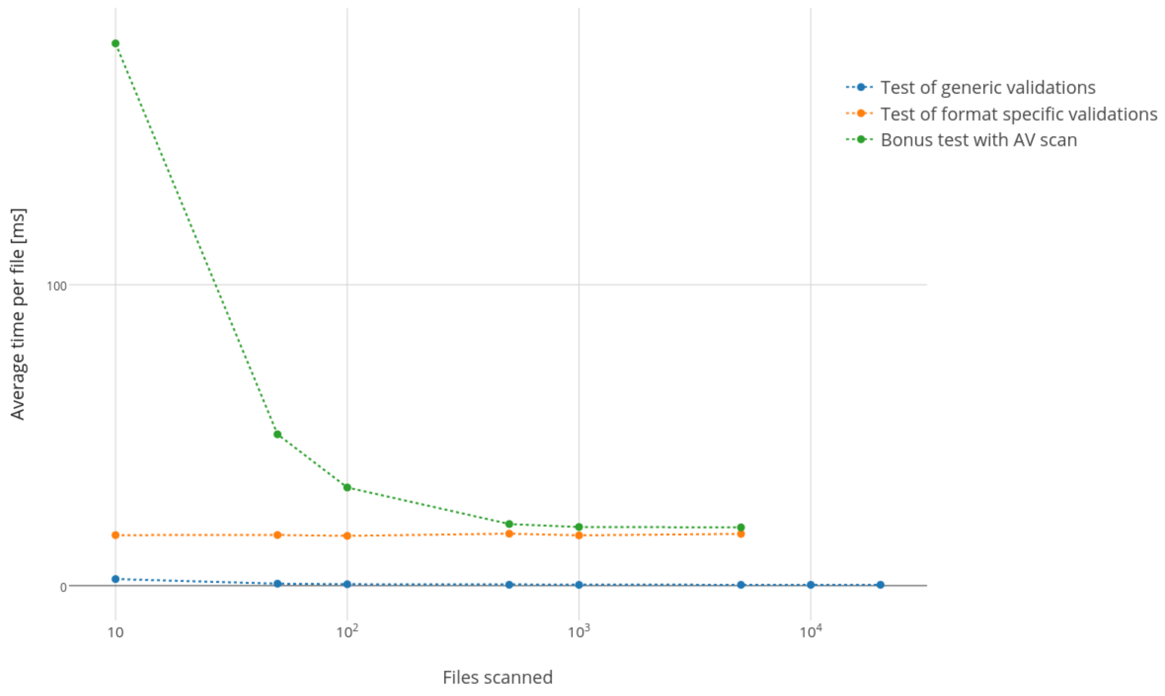

Figure 3: Plot of average time per file

Part 5
Conclusion

5.1 Future of the library

The library is already available on GitHub and package manager NuGet. It offers users two options how they can quickly integrate our library with their project. This presents us with a lot of possibilities when speaking about the future of this project. Should anyone be interested in extending the functions beyond its current capabilities, they are free to fork the project and create modifications as they see fit. Depending on the popularity, additional functions and validation modules can be created. Of the many possible options, the first few that come to mind are the following:

- Module validating the time of file's creation and modification.
- Format specific modules that can check additional popular file formats beyond what is already offered (e.g. audio formats, video formats).
- Format specific modules that can validate formats used for serialization of data (e.g. XML, JSON). These modules would only let through files with a valid data structure. For example, they would filter out a .json file which does not contain a valid JSON data structure.
- Similarly to the previous point, given a file containing a source code written in an arbitrary language, a module that would allow only a valid source code written in the selected language through could be an interesting addition. Indeed, we are getting quite specific, but our library does not care. Users can create their own validation modules that can be as specific as they need. There is no reason to restrict them.

Given that the source code is distributed under an open source license, there is no way of telling how many people might include it in their projects. It is important to remember that this library is useful only for a fraction of possible users, so we can hardly predict how widely it would be adopted.
The library is already being integrated into a project where it will assist with processing incoming requests. These requests contain a batch of files and it is crucial that these files meet certain requirements for the request to be processed correctly.

5.2 Summary

We have created a library which can validate important metadata like size, checksum or signature of all file types and additionally implemented modules that can detect corrupted files of a standard file formats by integrating existing libraries designed to manipulate a given file format. Our library can use installed anti-virus software as part of the validation procedure or an available online malware detection service VirusTotal. Users can choose to use only the core functions or download optional modules they might need for their project. As the library can load them dynamically, the user always has to download only what he will really be using. This limits the number of external libraries being downloaded and reduces the size of the library. Should a new optional library be created in the future, we have shown that it can be easily integrated into existing functions due to the library's modular nature. The library's interface offers a handful of commands which keep it straightforward and easy to use. Distributed under an open source license and available under GitHub and NuGet, anyone can use, modify and extend the result of our work. While it does provide functions that only a fraction of programmers might ever need, there are no similar libraries available for free, which can make our work a valuable tool for many.

References

[1] GAIGALAS, Gomes Alexandre. Respect/Validation. [online]. [cit. 2017-05-01]. Available from: https://github.com/Respect/Validation.

[2] SKINNER, Jeremy. JeremySkinner/FluentValidation. [online]. [cit. 2017-05-01]. Available from: https://github.com/JeremySkinner/FluentValidation.

[3] MSDN. Naming Files, Paths, and Namespaces. [online]. [cit. 2017-05-01]. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365247.

[4] KESSLER, Gary. File Signatures Table. Gary Kessler Associates. [online]. 23.7.2016 [cit. 2017-05-01]. Available from: http://www.garykessler.net/library/file_sigs.html.

[5] MATTHEWS, Ron. File signatures. [online]. 8.3.2012 [cit. 2017-05-01]. Available from: https://www.filesignatures.net/.

[6] MICROSOFT. Default cluster size for NTFS, FAT, and exFAT. [online]. 2017 [cit. 2017-05-01]. Available from: https://support.microsoft.com/cs-cz/help/140365/default-cluster-size-for-ntfs,-fat,-and-exfat.

[7] KOHNO, Tadayoshi., Niels. FERGUSON and Bruce SCHNEIER. Cryptography engineering: design principles and practical applications. Indianapolis, IN: Wiley Pub., c2010. ISBN 978-0470474242.

[8] WANG, Xiaoyun, et al. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. IACR Cryptology ePrint Archive. [online]. 2004 [cit. 2017-05-01]. Available from: http://eprint.iacr.org/2004/199.pdf.

[9] STEVENS Marc, BURSZTEIN Elie, KARPMAN Pierre, ALBERTINI Ange, MARKOV Yarik. The first collision for full SHA-1. [online]. 23.2.2017 [cit. 2017-05-01]. Available from: https://shattered.it/.

[10] DONIS, Peter. Windows vs. Linux File Timestamps. [online]. 2010 [cit. 2017-05-01]. Available from: http://www.peterdonis.net/computers/computersarticle3.html.

[11] ECMA International. OFFICE OPEN XML OVERVIEW. *Ecma International* [online]. 2010 [cit. 2017-05-01]. Available from: http://www.ecma-international.org/news/TC45_current_work/OpenXML%20White%20Paper.pdf.

[12] Microsoft Open Technologies. OfficeDev/Open-XML-SDK. [online]. [cit. 2017-05-01]. Available from: https://github.com/OfficeDev/Open-XML-SDK.

[13] The Apache Software Foundation. The Apache OpenOffice API Project. [online]. [cit. 2017-05-01]. Available from: http://www.openoffice.org/api.

[14] THOMAS, Kas. Portable Document Format: An Introduction for Programmers. MacTech [online]. [cit. 2017-05-01]. Available from: `http://www.mactech.com/articles/mactech/Vol.15/15.09/PDFIntro/index.html`.

[15] MSDN. System Error Codes. [online]. 2017 [cit. 2017-05-01]. Available from: `https://msdn.microsoft.com/en-us/library/ms681381.aspx`.

[16] COOPER, Mendel. Exit and Exit Status. [online]. 12.6.2003 [cit. 2017-05-01]. Available from: `http://www.faqs.org/docs/abs/HTML/exit-status.html`.

[17] MSDN. C# Reference. [online]. [cit. 2017-05-01]. Available from: `https://msdn.microsoft.com/en-us/library/618ayhy6`.

[18] The Apache Software Foundation. Apache License. [online]. 2017 [cit. 2017-05-01]. Available from: `https://www.apache.org/licenses/LICENSE-2.0.html`.

[19] The Apache Software Foundation. Apache License v2.0 and GPL Compatibility. [online]. 2017 [cit. 2017-05-01]. Available from: `https://www.apache.org/licenses/GPL-compatibility.html`.

[20] BALTER, Ben. Open source license usage on GitHub.com. GitHub. [online]. 9.3.2015 [cit. 2017-05-01]. Available from: `https://github.com/blog/1964-license-usage-on-github-com`.

[21] FOWLER, Martin. FluentInterface. MartinFowler.com. [online]. 20.12.2005 [cit. 2017-05-01]. Available from: `https://www.martinfowler.com/bliki/FluentInterface.html`.

[22] CMSC 311. Ascii vs. Binary Files. [online]. [cit. 2017-05-01]. Available from: `https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/asciiBin.html`.

[23] JONES, Keith. Genbox/VirusTotal.NET. [online]. [cit. 2017-05-01]. Available from: `https://github.com/Genbox/VirusTotal.NET`.

[24] EICAR. Intender use. [online]. 2017 [cit. 2017-05-01]. Available from: `http://www.eicar.org/86-0-Intended-use.html`.

Appendices

A List of CD contents

```
cd
├─ src
│  ├─ Verifiler .............................. Main library.
│  ├─ VerifilerCore ............ Library with core functionality.
│  ├─ VerifilerACCDB ...... Optional library verifying ACCDB format.
│  ├─ VerifilerImage ..... Optional library verifying image formats.
│  ├─ VerifilerMSLegacy .. Optional library verifying old MS office formats.
│  ├─ VerifilerODF .... Optional library verifying OpenDocument formats.
│  ├─ VerifilerOpenXML .... Optional library verifying OpenXML formats.
│  ├─ VerifilerPDF ........ Optional library verifying PDF format.
│  └─ VerifilerTest .......................... Unit test library.
├─ doc
│  ├─ mt ............................ Final version of the thesis.
│  └─ text_src ...................... Source code of the thesis.
│     └─ img ........................ Images used in the thesis
└─ readme ................................. List of CD contents
```

B List of used abbreviations

AODL	An Open Document Library
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AV	Anti-Virus
CLU	Command Line Utility
COM	Component Object Model
DOS	Disk Operating System
EOF	End Of File
exFAT	Extended File Allocation Table
FAT	File Allocation Table
JPEG	Joint Photographic Experts Group
MD5	Message Digest algorithm
MS	Microsoft
NTFS	New Technology File System
ODF	Open Document Format
OOXML	Open Office XML
PDF	Portable document format
PHP	Hypertext Preprocessor
SDK	Software Development Kit
SHA	Secure Hash Algorithm
UTF-8	UCS/Unicode Transformation Format
XLS	Microsoft Excel Spreadsheet
XML	eXtensible Markup Language

C Links to GitHub repositories

- https://github.com/Hanzik/verifiler
- https://github.com/Hanzik/verifiler-test
- https://github.com/Hanzik/verifiler-core
- https://github.com/Hanzik/verifiler-accdb
- https://github.com/Hanzik/verifiler-image
- https://github.com/Hanzik/verifiler-mslegacy
- https://github.com/Hanzik/verifiler-odf
- https://github.com/Hanzik/verifiler-openxml
- https://github.com/Hanzik/verifiler-pdf

D Links to NuGet repositories

- https://nuget.org/packages/Verifiler
- https://nuget.org/packages/VerifilerCore
- https://nuget.org/packages/VerifilerACCDB
- https://nuget.org/packages/VerifilerImage
- https://nuget.org/packages/VerifilerMSLegacy
- https://nuget.org/packages/VerifilerODF
- https://nuget.org/packages/VerifilerOpenXML
- https://nuget.org/packages/VerifilerPDF