A Framework for Automated Verification in Software Escrow

Elisabeth Weigl
SBA Research
Vienna, Austria
eweigl@sba-research.org

Johannes Binder
SBA Research
Vienna, Austria
jbinder@sba-research.org

Stephan Strodl
SBA Research
Vienna, Austria
sstrodl@sba-research.org

Barbara Kolany
ITM Münster
Münster, Germany
barbara.kolany@uni-muenster.de

Daniel Draws
SQS Research
Cologne, Germany
daniel.draws@sqs.com

Andreas Rauber
Vienna University of Technology
Vienna, Austria
rauber@ifs.tuwien.ac.at

ABSTRACT
If a business is in need of customized software it often orders it from a third party developer. This can lead to a dependency on this developer regarding maintainability and development of the product. Software Escrow offers a mitigation to this as it includes a trustable escrow agent in the business relationship. The agent is responsible for depositing all material that is needed to develop the software, like source code, documentation, and licenses for software artifacts. If a predefined trigger event occurs, the agent is obliged to hand out the objects to the customer. Thus the material needs to be of a quality that allows the customer to further maintain and develop the software. To guarantee this, all artifacts deposited are verified for their maintainability. As this verification is a time consuming and costly factor, we propose a Technical Software Escrow Framework that supports the reviewing process by highlighting parts of the software that can pose a problem regarding their maintainability. We also analyze an exemplary use-case software to show the applicability of our framework.

Keywords
Software Escrow, Software Quality, Evaluation, Verification Framework, Case Study

1. INTRODUCTION
For their daily work, businesses are in need of customized software. Thus they order the development and customization of software from other businesses, which commonly sell them a license for its usage. For the customer this then represents an asset of value, as he uses it for his day-to-day business. In order to adopt and adjust the software or to add new features, usually a service and maintenance contract is set up with the developer. This introduces a high dependency on the developer. In case he goes bankrupt or refuses to maintain the program, the customer will be negatively affected or in the worst case sustain severe financial effects. As common software licensing only includes the object code and not the sources of the software, the customer does not have access to the source code and thus will not be able to further develop or fix the software.

Software Escrow offers a mitigation to this scenario by placing a trustable party between the IT partner and his customer. The material relevant for the software development is deposited at the agent. To be able to further develop the software, it is important that the material gets checked. The agent is responsible for this verification and the subsequent storing for later re-use. During the depositing process, he has to ensure the physical security of the material. In case a trigger event occurs (e.g., bankruptcy), he is obliged to hand out the material to the customer, who wants to develop and maintain the software.

A successful escrow has several considerations to take into account. It has to be legally ensured that the future developing party has the rights for development, e.g., they have the right to use the source code and the libraries. These points are agreed on in the escrow contract, which is an extension to the commonly used license and maintenance contract and which has to be aligned with both of them. Other parts of the agreement involve decisions on the materials to be deposited, notification obligations, and trigger events that entail the release of the materials. With the trigger events clearly specified, the escrow contract helps to quickly release material and avoid delays in the procedure and legal uncertainties.

From a technical point of view, the escrow agent has to verify the completeness and evaluate the quality of the material relevant to the software project put into escrow, according to the agreements made in the contract. Completeness of material is needed because missing software artifacts can prevent a developer from maintaining the software [10]. Current escrow approaches only focus on material consisting of source code and documentation, without detailed verification (e.g., only virus check of the data). Software projects, however, consist of more than that. Different artifacts like compilers, test scripts, external resources like Web services, or databases are also part of a software development project.

Maintainability of the deposited software is an important indicator for future maintenance and development processes and thus also has to be evaluated. Up until now maintainability of the deposited material is not promoted to be considered in escrow agreements. Standards like [6] only propose quality checks that do not evaluate maintainability comprehensively, including tests for e.g., readability of the data, random samples of the documentation, virus-free data, or compilation. These tests do not check all artifacts relevant for software projects.

The check for completeness and quality of the deposited material has to be done by a reviewer. A manual review conducted by the escrow agent requires sustainable effort and causes high costs. To increase the efficiency of the reviewer we developed a framework to support the verification process with automated artifact analysis. Thus we propose a Technical Software Escrow Framework implemented in Java that supports a manual review with automatic checks of the deposited software development project. For this purpose it pre-screens all artifacts, performs automatic checks and assessments, and highlights parts of the software project that need further examination by the reviewer (e.g., complex classes with minimal documentation).

The presented framework extends Software Escrow by Digital Preservation aspects of software development projects, like identifying dependencies to external resources of software such as Web services, which is needed to ensure long term availability of the service and its functionality. With this we introduce a framework capable of supporting the execution of Software Escrow by supporting manual evaluation actions of the reviewer with automatically executable processes and thus reducing the time needed for verification.

In this paper we will explain the different steps of Software Escrow and the technical verification in detail. With an exemplary use case, based on a Java open source project, we will go through the evaluation process and show the applicability of our framework. We will start with the related work on software quality important for Software Escrow in Section 2. An overview of Software Escrow and the escrow process follows in Section 3. An evaluation of our use case can be found in Section 4. In Section 5 we summarize the lessons learned and give a conclusion.

2. RELATED WORK

From a technical point of view the CEN Workshop Agreement 13620-5 - *ESCROWGUIDE* [4] offers a comprehensive information on Source Code Escrow and will be the basis for our investigation of Software Escrow. It comprises of five different parts: introducing Software Escrow, the view for each of the participants (developer, customer, agent), and one focusing on the audit process. Concerning technical aspects, the guide for developers [5] is the most interesting part for all parties regarding setting up a proper escrow contract. It describes what material to deposit, which will be necessary for our completeness check, where to put the escrow process in the software life-cycle, and gives an overview of the legal considerations for the developer.

As a theoretical concept for verification of the deposited material, the Escrowguide dedicated to the escrow agent [6] mentions three different levels. A *standard verification* only verifies the readability of the data, its completeness, or random samples of documentation. The *full verification* involves practical verification methods, including a compilation of the program and a test for functionality of the software. A few checks require the assistance of the software owner or client as well, which increases the effort for the affected parties. The third verification, the *bespoke verification*, may include tests from the standard or full verification together with additionally agreed tests. As not everything needed for the development of the software project is checked in the full verification, maintainability related checks can be agreed on here. In practice, a verification is done in more detail, including for instance the comparison between the compiled deposit materials and the executables running at the customer's site [13], or simulating a release event scenario [19].

A general introduction to the difficulties that arise can be found in [10]. It argues that the benefits of escrow do not compensate the time, legal fees, and other resources spent. We focus on the statement mentioned there that the material escrowed often is not usable after releasing it and try to approach this by extending common quality measurement methods.

Over time different standards were developed to describe and classify software quality. Whereas the ISO 9126 [15] set six main quality objectives, its successor and the current standard ISO 25010 [14] defines two quality models: one for quality in use, with five characteristics that relate to the outcome of interaction when a product is used in a particular context; the other for product quality, comprising of eight characteristics that relate to static properties of software and dynamic properties of the computer system. Important for Software Escrow are the two quality in use attributes portability and maintainability, which are in the focus of our framework.

For the quality tests we therefore focused on metrics that indicate maintainability and portability. Cyclomatic Complexity can be used to determine maintainability of source code. It was developed by Thomas J. McCabe in 1976 [17], based on the idea that humans can understand source code only until a certain amount of complexity of the code is reached. Instead of looking only at the syntactic elements in it, source code is seen as a directed graph with nodes and edges, nodes representing commands and edges representing direct connections between commands. According to McCabe a “reasonable, but not magical” [17] upper limit for the cyclomatic complexity is ten. Our framework will not stick to this number but we will compare our results to other popular open source projects. Related measures are Halstead's software metrics [9], including metrics like Program Volume, Difficulty, and Effort-To-Implement. Contrary to the cyclomatic complexity proposed by McCabe these metrics are based on lexical measures. Our framework uses this measurement to give an indication of understandability of the source code.

Regarding the measurement of quality in the source code comments, there are different metrics we used in the framework. The first one is comment density, which calculates the percentage of comments compared to lines of code and which we will combine with Cyclomatic Complexity to propose a new measurement . Arafat and Riehle [1] found that the average comment density in over 5000 successful open source projects was 18.67%.

As a second option to evaluate documentation quality, our framework also checks language for consistency and grammatical errors, which can make text hard to understand. Determining the language of a text and thus categorizing comments can be done with the usage of n-grams, as described by Cavnar and Trenkle in [3]. To proof text for correct spelling and grammar can also be an essential task in text analysis. Part-of-speech syntactic patterns as mentioned in Heyer et al. [11] or word sequence patterns that are compared to entries in error corpora as described in [18] can support this process.

Regarding the legal perspective of Software Escrow, [20] describes types of escrow agreements and release conditions. In [21] a short overview of the legal and technical aspects is presented. A detailed discussion of legal aspects regarding Software Escrow can be found in [12]. In this work we will highlight the most important aspects that have to be taken into account when setting up an escrow agreement.

3. SOFTWARE ESCROW

The subject of Software Escrow is a software produced by a developer for a customer and thus it refers to contractual agreements about the deposit of materials relevant for said software at a neutral third party. In case a contractually recorded trigger event occurs, the third party is obliged to hand over all materials to the customer.

Software Escrow agreements involve three parties:

- the **customer**, who has a need for a software in his business, wants to ensure that he is able to use the software for a longer time, and secure his investments in it
- the **software developer**, who makes the compiled object code available to the customer and hands over the sources and all other necessary artifacts to the escrow agent
- the **escrow agent**, who is responsible for depositing the material and releasing it, and who has to verify that the submitted material meets the requirements as contracted, e.g., that all objects are available, accessible, and fulfill specified quality measurements

Figure 1 shows an illustration of the relationship between the parties.

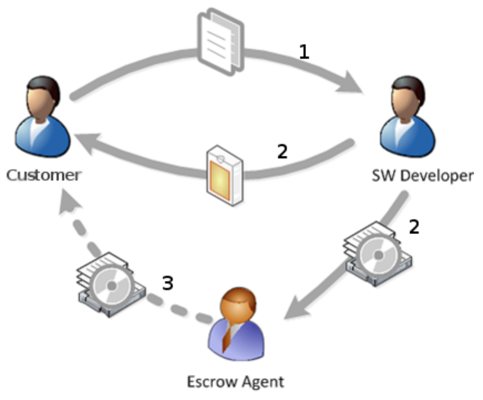


Figure 1: Relationship between escrow parties

There are technical and legal issues to be considered in a Software Escrow. From a legal point of view the contract needs to specify the obligations and rights of all three parties, the material to deposit at the agent, and the release events and procedure. It is important to exactly specify the events that entail the release of the deposited material to avoid legal uncertainties. The verification procedure and its success criteria also need to be stipulated. Licenses and rights to the material are as well part of the contract.

From a technical point of view, the completeness and quality of the deposited material have to be examined. The completeness of the material is crucial when the software has to be enhanced or maintained later. Thus all artifacts necessary for development have to be identified. This includes source code, libraries, compiler and compile instructions, test data, databases, and documentation amongst others. Availability of external dependencies of the software is necessary for preserving the functionality. Thus it is important to identify all external dependencies, like Web services or binaries used, in order to preserve them and therefore ensure the full functionality of the software over time.

Other technical considerations deal with the software's quality. Not mistakes or bugs are in the focus of Software Escrow, as they are part of the functional quality and thus part of an acceptance test by the customer. Maintenance aspects are important when depositing artifacts. All material has to be of a quality that ensures that it will be useable again. This also includes supplementary material needed for understanding certain artifacts, like documentation. These considerations are key drivers for our framework and will be explained in detail in Section 3.2, where the material is evaluated.

The escrow process can be divided into three phases: planning, execution, and redeployment. The focus of the Software Escrow planning phase (Section 3.1) lies on drafting the escrow agreement. The main task of the execution phase (Section 3.2) is the validation of the material against contractual requirements and its safe storage, as well as repeating those steps for each new version and update. The redeployment phase (Section 3.3) has to ensure the quick release of the deposited material once a contracted trigger event occurs.

3.1 Planning Phase

The planning phase is the first step in the escrow agreement and focuses on establishing an escrow contract. Preparations for an escrow contract should already be considered during the licensing contract negotiations. As certain costs are associated with setting up a Software Escrow, the first step should be an assessment of financial and business impacts if the software is unavailable. Then an appropriate escrow agent has to be selected. He has to be trustworthy for both parties. In the past a lawyer or notary was commonly chosen, however they often did not provide the necessary technical background needed. Nowadays specialized Software Escrow agents have been established, who are able to provide the knowledge needed for escrow as well as an appropriate technical infrastructure for evaluating and storing the deposited material.

In this phase the escrow agreement is composed and all parts of the contract are agreed on. The deposit material and its quality requirements have to be specified in the contract. As software can be a custom-made product, the artifacts needed for the deposit have to be specified for each project, including (based on [8]):

- Source code (including source code and libraries)
- Intellectual Property (especially licenses for different software components)
- Documentation (system and user documentation)
- Test environment (test cases, test scripts)
- Design environment (especially design models)
- Build environment (compilers, runtime environments, configuration files)
- Applications (databases or binary files that are used by the software)

Regarding the quality of these artifacts, the escrow agreement includes certain thresholds that have to be fulfilled. On the one hand this can be numerical boundaries, like compliance to a certain maximum source code complexity, and on the other hand the check for the fulfillment of requirements needed for immeasurable artifacts, like the requirements for a documentation of sufficient quality. The deposit procedure, including deadlines for the deposit, and the method of verification to fulfill the stipulated quality goals are agreed on as well [16].

The alignment of the licensing and maintenance contract with the escrow agreement (e.g., the specification of maintenance obligations) needs to be done in this phase as well. The escrow contract also needs to specify the transfer of rights, e.g., the allowance to use the source code or libraries needed for the maintenance of the software project. Rights exceeding the limitations of the original software contract, like commercial distribution of the program by the former customer, will have to be agreed on in the escrow contract separately.

3.2 Execution Phase

The execution phase is the second phase of the escrow process and depicted as the second step in Figure 1. It includes the deposit of the software, its verification, and the safe storage at the escrow agent, as well as the delivery of the program to the customer. As updates and new versions are released for the software, this process will be done repeatedly: With every update delivered to the customer, the material at the escrow agent has to be updated, verified, and deposited again. First the software, respectively a binary version of it, is delivered to the customer. At the same time the software development project and all its materials necessary for developing and maintaining the software are handed over to the escrow agent, where they get verified. If the verification is successful, the material gets stored safely, otherwise it gets rejected and the developer has to re-submit a revised version. These procedures have to be defined in the escrow contract.

The verification of the software has two main purposes: to ensure the completeness of the software development material and to verify the quality of each artifact. Both are necessary to guarantee the maintainability of the software once it gets handed out to the customer. The completeness check has to verify that each artifact agreed on and listed in the escrow agreement is part of the deposited materials. The quality evaluation includes verifications of the artifacts, like the quality of the documentation or that the sources do not exceed a predefined value for complexity. Each artifact has to be analyzed for its level of quality as specified in the escrow contract. To support this time-consuming process we developed a technical framework in Java that partly automates the verification process. It analyzes the artifacts and reports back to the reviewer those parts of the software that do not reach the required level of quality.

Our framework contains an extendable evaluation part with which various measurements can be conducted. It builds on the design of a Software Quality tool, which includes different static code analysis tools like Checkstyle[1], FindBugs[2] or PMD[3]. These support our maintainability evaluation because they are able to find source code sections that, e.g., contain code layout issues or flaws like unused variables that can make the source code difficult to understand. A tool combining these code analysis programs and different statistic code measurements is Sonar[4], an open source platform for continuous quality inspection, that forms the basis for our technical framework. Sonar supports the analysis of programs in several languages. With its client-server model the analysis can be run on a local system and the server provides different check modules for the client [2]. A project's quality is measured using metrics and rules, resulting in numerical values and violations, respectively. Sonar provides many measurements out of the box but can also be extended by integrating custom plugins. A description of the plugins developed for our Software Escrow scenario can be found below.

[1]http://checkstyle.sourceforge.net
[2]http://findbugs.sourceforge.net
[3]http://pmd.sourceforge.net
[4]http://www.sonarsource.org

In a Software Escrow scenario, the escrow agent first configures the framework according to the requirements agreed on in the escrow contract. The framework then processes the artifacts. Once it has finished it presents the reviewer with an overview of its findings, classifying the results according to their impact on the quality.

The following categories of quality checks related to maintainability, used to determine the quality of the deposit material, were implemented in our framework:

Completeness of artifacts. A reliable way to ensure completeness of the deposited source code material is to rebuild the software. Our prototype executes a build script and reports errors that may arise when doing so. The existence of other artifacts agreed on in the contract, such as additional documentation or specifications, can be assured either manually or using automatic checks as part of the build process.

Consistency of sources and released binary. The software put into escrow has to be the same as the one delivered to the customer. To verify this, the sources at the escrow agent have to be built and compared to the binaries delivered to the customer. Our implementation checks if the output generated by building the software matches a provided set of reference artifacts.

Quality of documentation. Documentation about the software development project is required to understand considerations and decisions made during the design and development phase in natural language. It is especially important if the software in question has to be maintained and possibly enhanced at an unknown time in the future because the programmer needs to understand the structure and design of the software. Thus the documentation has to be adequate, easily readable, and easily understandable. The following considerations apply to source code comments as well as additional documentation and specification, like architecture descriptions, requirement documents, manuals, etc.

One aspect that affects understandability is the language that has been used for the documentation. It needs to be ensured that all documentation is available in the agreed language. Our implementation detects the language of comments and reports if unexpected languages are found. To do so, the comments are extracted using SSLR[5] and analyzed using the Java Text Categorizing Library[6], which uses an algorithm based on n-grams [3]. To minimize the number of false positives, short comments can either be ignored or checked using a word list.

Also spelling, grammar, and other errors in documentation influence readability. Our implementation uses the text proof tool LanguageTool [18] to find issues of various categories like misspellings, wrong grammar, uncommon phrases, etc. in comments. It is possible to ignore specific issue types to filter frequently occurring mistakes that do not influence the readability, e.g., multiple whitespace characters, caused by specific formatting styles of comments. For each document the ratio of words compared to the number of issues detected in the comments is determined. This gives an overview of documents containing proportionally more errors than others.

[5]https://github.com/SonarSource/sslr
[6]http://textcat.sourceforge.net

Quality of source code. Software metrics can be used to assess the quality of the software, i.e., maintainability. An adequate level of quality is required for further development of the software. Sonar already implements a number of metrics for quality verification that can be used for our maintainability approach, such as the Cyclomatic Complexity. As mentioned in Section 2, Halstead's software metrics are a similar measurement method. As they are not implemented in Sonar, we provided a plugin for calculating the Halstead's software metrics Difficulty, Effort, Volume, Time to Program, and Bugs Delivered. For object oriented languages like Java there is no standardized way to calculate those metrics [7], therefore we implemented them to the best of our knowledge. Furthermore rule checks of Sonar can be used to verify the adherence to coding standards and best practices.

To support the verification of project specific requirements our implementation provides the possibility to calculate a measure using a custom defined formula that can make use of other measures and violation counts. The evaluation of this formula is done utilizing the Math Expression Parser of the Symja project[7]. For Software Escrow we propose *CCC*, a metric that sets Cyclomatic Complexity (CC) and comment lines density (C) in relation. Cyclomatic Complexity indicates the effort of an external developer to understand source code, documentation tends to ease understandability:

$$CCC = CC/(1 + (C/100))$$

Usage of third party resources. We further extended the framework by some digital preservation concerns that are also useful for escrow such as ensuring the availability of external sources. References to external third party resources, like libraries or Web services used in the software, can affect the functionality of the software. If the provider of the service is not available anymore, this can lead to a non-functioning program. Therefore external references have to be identified and properly inspected when verifying the source code. It has to be ensured that the service they are using is available in the long term. A potential strategy of the escrow agent is to deposit the library or materials needed for a Web service as well. If this is not possible, e.g., in the case of proprietary Web services, it has to be ensured that the executing source code sections are identified and reported as a risk to the customer. The use of external services should be specified in the licenses and escrow contract. Our implementation supports the escrow agent in identifying those external resources by reporting matches of a text-based *regex* search over source files which looks for Web service calls, system calls, etc.

[7]https://code.google.com/p/symja

A scenario that the escrow agent needs to be aware of is potential hiding of functionality in compiled libraries that limit the possibility to maintain the software. Instead of providing the source code, developers could supply compiled libraries to hide implementation details. Unknown libraries or such that are not available in public repositories are potential candidates for hiding code. To verify libraries our implementation performs a hash based lookup in the Maven Central Repository[8] of the JAR files that are part of the software. Other artifacts are looked up in the National Software Reference Library[9], a database containing hash values and other metadata of files that are part of software packages like Adobe Photoshop, Red Hat Linux, etc. Libraries which are not found in the corresponding database are reported and need to be checked against the agreements specified in the contract.

Legal certainty. Licenses are essential as they specify the legal foundation for the usage of the software. Thus the escrow agent needs to determine the licenses of the software's artifacts, as it has effects on the allowed usage in case of a release of the material. Our implementation extracts and identifies license information embedded in source files using the Perl script *licensecheck*[10]. For the licenses of the included libraries we use the License Maven Plugin[11] to determine the license information.

3.3 Redeployment Phase

The redeployment phase is the third phase of Software Escrow. Its main task is to ensure the quick release of the deposited material once a contracted trigger event occurs. The objective is to prevent a potential downtime of the customer's software. The events leading to the release of the software were agreed on in the *Planning* phase (cf. Section 3.1). If one of them occurs, the customer has to inform the escrow agent, who needs to check the contractual correctness, and proof the event. The agent is then obliged to release the material to the customer. Trigger events that lead to the release of the deposited materials to the customer can be the insolvency of the software developer, the liquidation of the developer's company, or an unjustified refusal of the developer to maintain the software.

4. EVALUATION

As an exemplary use case the review of aTunes[12], an open source audio player and media library, is used. The case study performs a verification by using the criteria described in Section 3.2, similar to one done by an escrow agent. Figure 2 shows the Sonar overview presentation of the results from the technical framework, presenting the metrics and checks of the software. aTunes 3.0.8 consists of 81,915 lines of code and 1,499 classes. As material to deposit we used the sources in the SCM repository[13]. As software binary release that has been handed over to the customer we used the official release package[14].

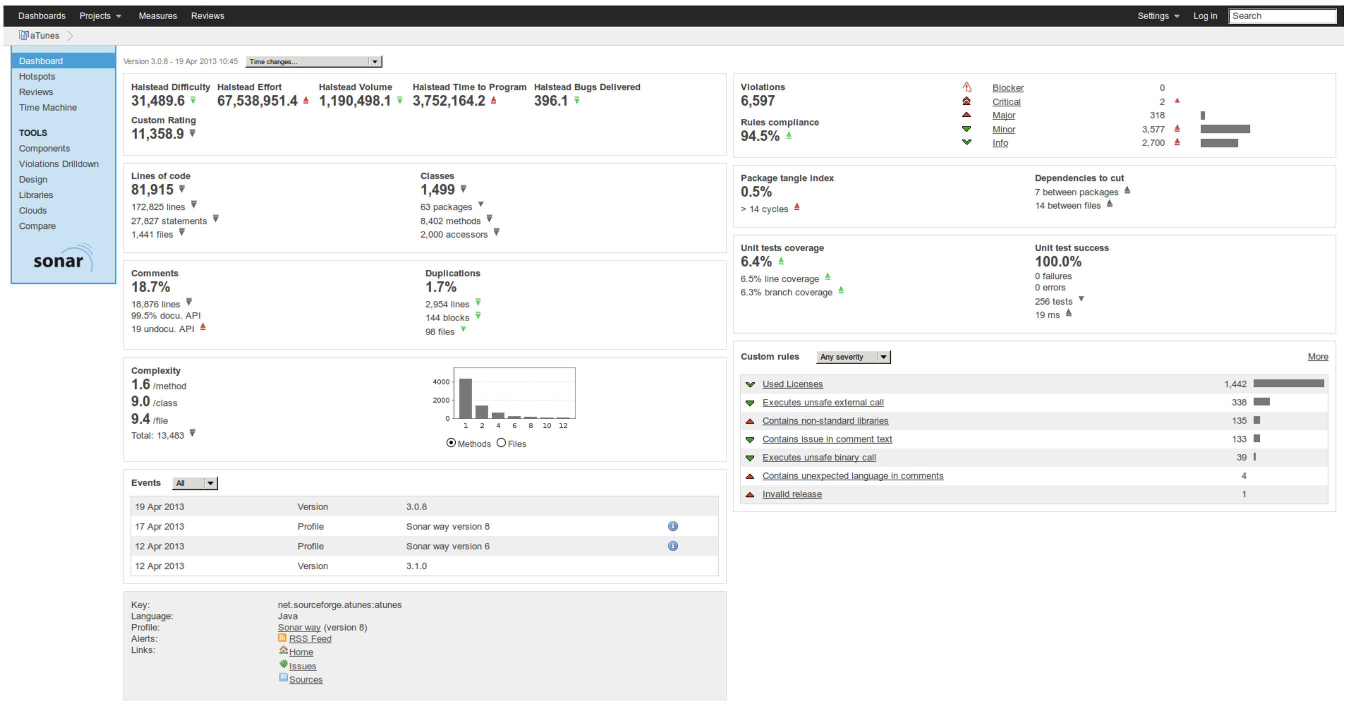
Completeness of artifacts. The check for completeness of artifacts was done by building the software. All essential artifacts needed for the development of the software were contained in the deposit. Some documentation like requirements documentation, coding guidelines, and user documentation was available in the aTunes Wiki[15]. Other documentation for developers, like architecture description, was not found in manual inspection. Depending on the requirements, all documentation should be available in the deposited materials.

Consistency of sources and released binary. In order to compare binaries it is important to use the same compiler version for all builds. Otherwise the resulting binaries can differ even if the same sources have been used. After using the correct compiler and ignoring files that hold metadata like build count, build time, etc., there are still some files missing. Those are related to builds for other platforms than Linux, which we did not execute, and are not required when running aTunes in Linux. Besides that, the rebuilded artifacts match the reference artifacts.

Quality of documentation. The documentation of aTunes only consists of the comments in the source files, thus only these were evaluated. The 18.7% comment line density nearly matches the average of comment line density found in open source projects as determined in [1]. Further inspection showed that there are two abstract classes and 17 other classes with public methods that do not have any documentation. Interfaces and enumerations seem to be commented the most, which is good common coding practice. From the 1,441 source files in aTunes, the Java Text Categorizing Library the framework uses reported four to contain comments in Hungarian when expecting comments in English only. Manual inspection showed that all of the reported files actually contain comments written in English, so those were false positives. Other comment quality issues have been identified by our framework using the text proof tool LanguageTool. To reduce issues with little impact, like warnings about duplicate whitespaces, a reduced set of LanguageTool categories has been used to check for readability issues. The framework reported 133 text proof issues, which indicates a good overall quality compared to the length of the documentation.

Quality of source code. aTunes showed a Cyclomatic Complexity of 1.6 per method, 9.0 per class, and 9.4 per file. We compared the complexity of aTunes to the average complexity of the projects listed in the public Sonar instance Nemo [22]. At the time of evaluation Nemo contained 204 projects, 177 of them were Java projects, amongst others

[8]http://search.maven.org
[9]http://www.nsrl.nist.gov
[10]http://www.beathovn.de/licensecheck
[11]http://mojo.codehaus.org/license-maven-plugin
[12]http://www.atunes.org
[13]http://sourceforge.net/p/atunes/code/HEAD/tree/tags/aTunes 3.0.8
[14]http://sourceforge.net/projects/atunes/files/atunes/aTunes 3.0.8
[15]http://www.atunes.org/wiki/



Version 3.0.8 - 19 Apr 2013 10:45

Halstead Difficulty	Halstead Effort	Halstead Volume	Halstead Time to Program	Halstead Bugs Delivered
31,489.6	67,538,951.4	1,190,498.1	3,752,164.2	396.1

Custom Rating
11,358.9

Lines of code
81,915
172,825 lines
27,827 statements
1,441 files

Classes
1,499
63 packages
8,402 methods
2,000 accessors

Comments
18.7%
18,876 lines
99.5% docu. API
19 undocu. API

Duplications
1.7%
2,954 lines
144 blocks
98 files

Complexity
1.6 /method
9.0 /class
9.4 /file
Total: 13,483

Events

19 Apr 2013	Version	3.0.8	
17 Apr 2013	Profile	Sonar way version 8	ⓘ
12 Apr 2013	Profile	Sonar way version 6	ⓘ
12 Apr 2013	Version	3.1.0	

Key:	net.sourceforge.atunes:atunes
Language:	Java
Profile:	Sonar way (version 8)
Alerts:	RSS Feed
Links:	Home Issues Sources

Violations
6,597

Rules compliance
94.5%

Blocker	0
Critical	2
Major	318
Minor	3,577
Info	2,700

Package tangle index
0.5%
> 14 cycles

Dependencies to cut
7 between packages
14 between files

Unit tests coverage
6.4%
6.5% line coverage
6.3% branch coverage

Unit test success
100.0%
0 failures
0 errors
256 tests
19 ms

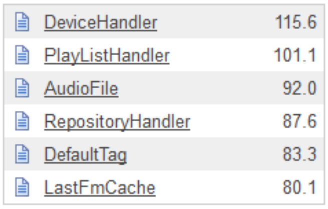
Custom rules

Used Licenses	1,442
Executes unsafe external call	338
Contains non-standard libraries	135
Contains issue in comment text	133
Executes unsafe binary call	39
Contains unexpected language in comments	4
Invalid release	1

Figure 2: The extended Sonar dashboard showing aTunes' results

including the JDK 7[16], JFreeChart[17], and several projects from the Apache Software Foundation[18]. The average Cyclomatic Complexity of those projects was 2.5 per method, 16.2 per class, and 19.5 per file. So considering other open source projects the results of this metric indicate a low complexity of aTunes. The Cyclomatic Complexity of aTunes shows similar results as its Halstead's Difficulty metric.

In our experiments we used a custom measure that takes into account the complexity and the comment density, with the idea that complex code should be easier to understand if it is commented properly (see Section 3.2). A list of the worst performing files gives a good starting point for a manual inspection of the software by the reviewer. Figure 3 shows the resulting poorest performing classes of this metric.



DeviceHandler	115.6
PlayListHandler	101.1
AudioFile	92.0
RepositoryHandler	87.6
DefaultTag	83.3
LastFmCache	80.1

Figure 3: The most incomprehensible classes of aTunes according to the CCC metric

Usage of third party resources. aTunes fetches most of its libraries through the build and dependency management tool Maven. Those libraries are considered trusted as they are provided from a central, public repository, which also provides the library's sources if available. Maven allows including additional repositories, which should be examined by the reviewer to verify their trustworthiness. From the nine libraries that are not obtained using Maven but already included in the deposited material, five have been reported as unknown, due to the fact that they were not available through the Maven repositories or that their hash value did not match one library there. All of them are used to create installation routines. The further investigation of the libraries depends on the agreements made in the escrow contract.

In our experiments our framework brought up 88 files that are assumed to contain external calls, 71 of them Web service calls and 17 binary calls. The number of Web service calls can be explained by further examination, which showed that the source of the calls are for instance modules that fetch additional meta data about the media from services like *Last.fm*. As we considered these modules optional, none of the Web service calls are an issue in this case. For a full functionality though, these Web service calls would pose a problem as they cannot be deposited as well. Inspection of the reported files showed further that many times operating system processes are spawned in order to execute external tools. One example of such a binary is *mplayer*[19], one of the supported audio playback engines. Other externally executed tools handle importing audio CDs to aTunes and encoding different audio formats. The deposited material

[16]http://openjdk.java.net/
[17]http://www.jfree.org/jfreechart/
[18]http://apache.org/

[19]http://www.mplayerhq.hu

contains binaries of the external tools for Windows and Mac OS. In Linux aTunes expects those tools to be installed in order to use the full functionality of the program. As mentioned in Section 3.2, externally called binaries are difficult to maintain so the reviewer should deposit those dependencies which are a necessity.

Legal certainty. All source files of aTunes contain license information (GPL in this case), but the framework could not find licenses in the JAR files that are part of the source distribution. In six cases the JAR files contained no licenses and in three cases the format of the license text could not be handled due to formatting issues. The licensing of those libraries need to be clarified in order to avoid legal consequences when releasing and further developing the software.

Summary. The applicability of the framework was shown in this Section. The framework supported review of aTunes showed that all artifacts required to build the software are available. External runtime dependencies are provided as artifacts, except those for Linux environments, which were missing. As discussed this could lead to problems in the future and the missing binaries should be put into escrow. Dependencies to Web services are used for optional features of the software. Depending on the requirements, these need to be put into escrow as well to preserve the full functionality of the software. The review also indicated that core parts of the software are well documented in general, which helps developers to familiarize with the software in a fine grained level. The lack of documentation of the architecture slows down understanding the big picture of the software design. Comparison to other software projects indicates that aTunes is not overly complex. There are no severe legal issues to be expected, as licensing of the artifacts is clearly specified with the exception of some installer tools that can be replaced without endangering the functionality of the software.

5. CONCLUSIONS

Software Escrow is a mitigation strategy when using a software developed by a third party. This paper aimed at presenting the necessary aspects needed for a successful Software Escrow, pointing out shortcomings of current practice, and presenting legal and technical considerations of this process. We also looked into the three different phases of Software Escrow, beginning with planning and setting up an agreement, executing the escrow by depositing the escrow material and verifying its quality with regard to maintainability, and finally redeploying the software. We further extended escrow by some Digital Preservation aspects, such as the use of external services that can be unavailable in the future. For the execution phase and its verification part we developed a Technical Software Escrow Framework by extending Sonar, an open source Software Quality tool, with escrow specific checks. This framework is able to check all kinds of material necessary for a successful deposit, from licenses over source code to documentation. By highlighting and reporting artifacts that have low quality it is able to support the verification of requirements agreed on in the escrow contract. We applied our framework for demonstration purposes to the open source software aTunes and analyzed the performance of our tool. It can be shown that Software Escrow critical parts of the software are found and reported back. These reports can then be used to easily find potentially problematic sections that need further improvement. Our framework thus achieves the objective to support a reviewer in analyzing the deposited material by partly automating the search for common software project issues.

6. ACKNOWLEDGMENTS

This work has been co-funded by COMET K1, FFG - Austrian Research Promotion Agency and by the TIMBUS project, co-funded by the European Union under the 7th Framework Programme for research and technological development and demonstration activities (FP7/2007-2013) under grant agreement no. 269940. The authors are solely responsible for the content of this paper.

7. REFERENCES

[1] O. Arafat and D. Riehle. The commenting practice of open source. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 857–864, New York, NY, USA, 2009. ACM.

[2] C. Arapidis. *Sonar Code Quality Testing Essentials.* Community experience distilled. Packt Publishing, Limited, 2012.

[3] W. B. Cavnar and J. M. Trenkle. N-Gram-Based Text Categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[4] CEN Workshop Agreement. *ESCROWGUIDE - Source Code Escrow - Guidelines for Acquirers, Developers, Escrow Agents and Quality Assessors.* European Committee for Standardization (CEN), 1999.

[5] CEN Workshop Agreement. *ESCROWGUIDE - Source Code Escrow - Guidelines for Acquirers, Developers, Escrow Agents and Quality Assessors - Part 3: A developer's guide to taking part in source code escrow.* European Committee for Standardization (CEN), 1999.

[6] CEN Workshop Agreement. *ESCROWGUIDE - Source Code Escrow - Guidelines for Acquirers, Developers, Escrow Agents and Quality Assessors - Part 4: A guide to providing a reliable escrow service.* European Committee for Standardization (CEN), 1999.

[7] D. De Silva, N. Kodagoda, and H. Perera. Applicability of three complexity metrics. In *Advances in ICT for Emerging Regions (ICTer), 2012 International Conference on*, pages 82–88, 2012.

[8] D. Draws, S. Euteneuer, D. Simon, and F. Simon. Short term preservation for software industry. In *Proceedings of the 8th International Conference on Preservation of Digital Objects (iPres 2011)*, pages 130–139, 2011.

[9] M. H. Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, USA, 1977.

[10] S. Helms and A. Cheng. Source Code Escrow: Are You Just Following the Herd?

`http://www.cio.com/article/187450/Source_Code_Escrow_Are_You_Just_Following_the_Herd_?page=1&taxonomyId=3000`, 2008. [Online; accessed 13-June-2013].

[11] G. Heyer, U. Quasthoff, and T. Wittig. *Text Mining: Wissensrohstoff Text.* W3L Verlag, Bochum, 2005.

[12] T. Hoeren, B. Kolany, S. Yankova, M. Hecheltjen, and K. Hobel. *Legal Aspects Of Digital Preservation.* Edward Eldgar Publishing, 2013.

[13] Iron Mountain Incorporated. Comprehensive Asset Verification and Testing. `http://www.ironmountain.com/Services/Technology-Escrow-Services/Escrow-Verification-Services.aspx`. [Online; accessed 17-June-2013].

[14] ISO 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[15] ISO 9126:2001. International Standard ISO/IEC 9126, Part 1, Software engineering - Product quality - Quality model, 2001.

[16] M. Karger. Software-Hinterlegungsverträge. In *Computerrechts-Handbuch.* Kilian/Heussen, 2011.

[17] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[18] M. Miłkowski. Developing an open-source, rule-based proofreading tool. *Software - Practice & Experience*, 40(7):543–566, June 2010.

[19] NCC Group. Types of Verification. `http://www.nccgroup.com/en/our-services/software-escrow-verification/software-verification/types-of-verification/`. [Online; accessed 17-June-2013].

[20] M. R. Overly. *A Guide to IT Contracting: Checklists, Tools, and Techniques.* Auerbach Publications, Har/Cdr edition, Dec. 2012.

[21] V. Siegel. Software-Escrow. *Informatik-Spektrum*, 28(5):403–406, 2005.

[22] SONARSOURCE SA. Nemo - Sonar. `http://nemo.sonarsource.org/`, 2013. [Online; accessed 12-April-2013].