On Software Modernisation due to Library Obsolescence

Gerasimou, Simos; Kechagia, Maria; Kolovos, Dimitris; Paige, Richard; Gousios, Georgios

DOI
10.1145/3194793.3194798

Publication date
2018

Document Version
Accepted author manuscript

Published in
WAPI'18 Proceedings of the 2nd International Workshop on API Usage and Evolution (WAPI 2018)

Citation (APA)
https://doi.org/10.1145/3194793.3194798

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.
On Software Modernisation due to Library Obsolescence

Simos Gerasimou
Department of Computer Science
University of York
York, UK
simos.gerasimou@york.ac.uk

Maria Kechagia
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
m.kechagia@tudelft.nl

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Richard Paige
Department of Computer Science
University of York
York, UK
richard.paige@york.ac.uk

Giorgos Gousios
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

ABSTRACT

Software libraries, typically accessible through Application Programming Interfaces (APIs), enhance modularity and reduce development time. Nevertheless, their use reinforces system dependency on third-party software. When libraries become obsolete or their APIs change, performing the necessary modifications to dependent systems, can be time-consuming, labour intensive and error-prone. In this paper, we propose a methodology that reduces the effort developers must spend to mitigate library obsolescence. We describe the steps comprising the methodology, i.e., source code analysis, visualisation of hot areas, code-based transformation, and verification of the modified system. Also, we present some preliminary results and describe our plan for developing a fully automated software modernisation approach.

CCS CONCEPTS

• Software and its engineering → Software libraries and repositories; Software evolution; Maintaining software;

KEYWORDS

application programming interfaces; software libraries; library evolution; software modernisation; visualisation

1 INTRODUCTION

Modern software development practices advocate the use of third-party libraries as a means of improving maintainability, usability and dependability of software systems [7]. Many of these libraries aggregate functions and services, and expose endpoints, i.e., Application Programming Interfaces (APIs), that enable interacting with the logic of the libraries. An API is considered an implicit agreement between a software system (client) and a third-party library (provider). Based on this agreement, software developers can integrate the library within their client systems and use the provided functions without needing to be aware of the underlying implementation logic.

Despite the benefits accompanying the use of third-party libraries and their APIs, a strong dependency link is created with software systems using these libraries [2]. Since software systems and the libraries they are based on evolve independently, maintaining the system to a fully-functional state requires additional effort when changes to the libraries’ APIs violate the agreement. For instance, library evolution caused by new requirements or architectural changes can lead to API modifications that break compliance with client systems. Likewise, end-of-support of a software library can cause client systems to fall into “technology stagnation” unless they switch to another functionally-equivalent and actively maintained library. Similarly, the introduction of a competing library with improved functionality, better features and reduced overheads might urge software developers to consider adopting this new library in subsequent system versions.

In order to avoid the imminent risks that arise from using obsolete libraries (e.g., unresolved bugs, susceptibility to cyber attacks due to security issues), developers typically sustain the effort of modernising their systems [13]. Nevertheless, modifying the system to start using a new library is a time-consuming, error-prone and, to a great extent, developer-driven task [2, 5]. Performing correctly this task includes identifying system instructions that must change, doing the necessary code changes and, finally, checking that the modernised system preserves its original functionality. As the size of the client system and the legacy third-party library increases, the effort required for performing these steps increases significantly [7].

We argue that developers can reduce significantly the manual effort spent for mitigating library obsolescence by adapting concepts and approaches from the areas of software visualisation and bug finding [5, 12]. For instance, visualisation paradigms can help developers to understand better the architecture of their systems and perform impact analysis. Software visualisation is an active research area, but with rather limited adoption in software development [8, 12]. Also, code-based and pattern-based transformation approaches can assist with automating code refactoring [2].

In this paper, we propose an end-to-end methodology that not only helps developers to assess the required effort, but also automates a great part of the modernisation task. In particular, the methodology involves analysis of the software system’s source code to extract hotspots, i.e., source code elements that invoke the legacy library, visualisation of these hot areas using the city metaphor [12], automatic source code transformation based on templates (patterns), population of the generated templates with suitable code, and, finally, verification of the software system and the transformed code.
We motivate our work using two examples in which library evolu-
tion introduced compatibility-breaking changes. The first example
comes from TinyXML\textsuperscript{2}, the new version of the popular C++ XML
parsing library. Even though the evolved library is more CPU and
memory efficient than its predecessor, it is heavily redesigned and
does not maintain backward compatibility. Figure 1 shows a rel-
vant issue\textsuperscript{4} from the GitHub repository of the TinyXML project.
These obsolescence issues caused modernisation problems to de-
velopers of client systems. Some developers performed the migration
manually while others preferred to switch to another functionally-
equivalent library: for instance, the FileZilla client (https://filezilla-
project.org) started using pugiXML from version 12 onwards. Ir-
respective of the chosen solution, the effort required for updating the
system was substantial (e.g., see FileZilla development diary\textsuperscript{5}).
Second, we refer to the Lighttpd web-server library which is used
by several high-traffic websites (e.g., Wikipedia, YouTube). Research
in [4] found that an one-line change to one of the methods in the
updated library (Listing 1), broke support for HTTP compression and
crashed any client applications. Despite the minimal API changes,
this new bug issue could push developers to undertake the effort
and migrate their system to an alternative and more reliable library.
Clearly, library and API changes affect the stability of client
systems. We aim to assist developers by identifying the affected
source code in client systems and simplifying the migration process.

Listing 1: Lighttpd before and after update (as in [4])

// Lighttpd before update
\texttt{for (h=0, i=0; i < etag \rightarrow used; ++i)}
\texttt{\hspace{1cm} h = (h << 5) \& (h \gg 27) \& (etag \rightarrow ptr[i]);}

// Lighttpd after update
\texttt{for (h=0, i=0; i < etag \rightarrow used \rightarrow 1; ++i)}
\texttt{\hspace{1cm} h = (h << 5) \& (h \gg 27) \& (etag \rightarrow ptr[i]);}

\textsuperscript{4}http://www.grinninglizard.com/tinyxml2/index.html
\textsuperscript{5}https://github.com/leethomason/tinyxml2/issues/440
\textsuperscript{6}https://tinyurl.com/y7styrn4

3 SOFTWARE MODERNISATION APPROACH

3.1 Modernisation Methodology

The high-level overview of our methodology is shown in Figure 2. Given as inputs the source code of the client software system
and the API of the obsolete library, our methodology operates as follows. Initially, it starts a set of extraction transformations that enable
source code analysis, and extraction of suitable metrics and ab-
tract specifications from the legacy code. These metrics (e.g., total
number of API invocations) are very important as they provide
the means to establish and visualise the dependency level of the
software system on the obsolete library. Also, the specifications
are in a form suitable for reengineering. Then, through a code-
based transformation step, our methodology uses the specifications
to generate code based on templates (patterns). These templates
are subsequently populated with suitable code that permit to start
exercising the new library. Finally, verifying and validating the
transformed code and the software system as a whole (e.g., running
unit and integration tests) enables to check that both the function-
ality and properties of the modernised system are preserved.

Stage 1: Source code parsing and analysis

This automated stage focuses on parsing and analysing a set of
input files, i.e., the source code corresponding to the client system
and the API of the obsolete library. The parsing step comprises a
series of actions including source code scanning and preprocessing,
identification of language tokens, semantic analysis, name resolu-
tion and binding, and, finally, generation of various types of internal
models, indexes and Abstract Syntax Trees (ASTs).

Once parsing finishes, the extracted models and ASTs are anal-
ysed. This step enables to identify elements, i.e., files, classes, meth-
ods, and instructions, of the software system that access the obsolete
library. Orthogonal to this process is the extraction of a similar set
of elements from the library’s API that are used by the system.

By combining and analysing the generated sets of elements we
gain insight about the system and establish several dependency
metrics. These metrics capture information about the system ar-
chitecture including interconnections between software modules
and dependencies with external libraries and components. Also,
through impact analysis of the affected system elements we can
establish a coupling degree between the software system and the
obsolete library.

The effectiveness of this stage depends on using a suitable scan-
ning and parsing source code model. High-level programming
languages (e.g., C/C++, Java, C#) come with publicly available
and ready-to-use infrastructures that help with this task; see for
instance, Eclipse CDT for C/C++ and Roslyn for C#. We use Eclipse
CDT for our prototype implementation (Section 3.2).

Stage 2: Visualisation of software metrics
Manual source code inspection is challenging for complex systems
with multiple dependencies, packages, classes, and methods (e.g.,
FileZilla has more than 300 C/C++ files and more than 130KLoC).
To facilitate reasoning and effort estimation, we automatically gen-
erate interactive visual artifacts using the metrics produced during
analysis. These artifacts provide a pictorial view of the system and
help with system understanding.

The benefits of using suitable visual metaphors both for assisting
with software exploration and for reducing the cognitive load is
widely accepted [1, 8]. Several visual metaphors have been proposed
for representing static aspects of software (cf. Section 4). To the best
of our knowledge, however, there is no prior research on visualising
the dependencies of a software system on third-party libraries and
using this information to guide software modernisation.

We bridge the gap in existing research by adapting the semantics
of available visualisation metaphors (e.g., Treemap, Code City [12])
to this problem. In our preliminary realisation (Section 3.2), we use
Code City to represent software systems as three-dimensional cities
with districts and buildings, and associate the extracted dependency
metrics with visual properties of city components.

The development team can study the produce interactive visual
artifacts and carry out a risk analysis. The outcome of this study
indicates how to best address the obsolescence issues. An estimate
about the effort required for completing the modernisation can
be also made. For instance, if extensive coupling is identified, it
might be more sensible to investigate how to reduce its degree (e.g.,
by improving the system architecture) before proceeding with the
remaining modernisation stages.

Stage 3: Code-based transformation
During this stage, the software system undergoes a three-step trans-
formation in order to become compliant with the new library. The
first step includes (1) the automated generation of an abstraction
layer (e.g., using the adapter pattern) and its population with ele-
ments that delineate the usage of the obsolete library by the soft-
ware system; and (2) the automated modification of commands
in the affected system files to delegate the work to the generated
abstraction layer instead of the obsolete library. In this way, our
methodology minimises changes in the system’s source code.

The next step involves the inference of mappings between the
obsolete and new libraries [3]. A mapping rule comprises two sets
of API commands that perform the same task, one from the obsolete
library and the other from the new library. The outcome of this
inference step is a list of mappings.

Inferring likely API mappings is challenging and time-consuming,
and its automation has been the focus of recent research [11]. The
selection of a suitable technique depends on the characteristics
of the considered libraries and the expertise of developers. For in-
stance, static analysis and textual similarity techniques [9] can be
used when the APIs of the libraries are, to a degree, similar. Alter-
atively, developers can resort to inspecting the documentation of
these APIs and generating manually the list of mappings.

Listing 2: Code fragment that maps the get attribute method
in TinyXML to the equivalent methods in PugiXML

```c
const char* XMLElement::getAttribute(const char* name,
const char* value) const {
    return pugiXmlNode.attribute(name).value();
}
```

The final step involves populating the generated abstraction
layer with suitable code fragments that invoke the new library. To
achieve this, developers use the extracted list of mappings and write
code within the placeholders in the abstraction layer. For instance,
Listing 2 shows the TinyXML method (placeholder) for extracting
the attribute from an XML element and the corresponding code
fragment in PugiXML (https://pugixml.org); this relationship is one-
to-many. Once this is done, the functionality that was previously
done by the obsolete library is now undertaken by the new library.

The level of difficulty for writing the relevant code fragments
depends on the correspondence between the obsolete and the new
library. Investigating the time and effort required for completing
this task based on this correspondence is part of our future work.

Stage 4: Verification and validation
The next stage involves checking that both the functional and
non-functional requirements of the evolved system (transformed
and unaffected code) continue to hold. This includes running unit,
integration and system tests, or any other type of formal verification
(e.g., use model checking to verify the absence of concurrency bugs).
Special focus should be given to the amended system parts, i.e.,
the abstraction layer and affected components, as these parts are
most likely to have introduced erroneous behaviour. The visual
artifacts generated earlier can help developers to extract traceability
information and decide where to spend most of their efforts.

3.2 Preliminary Realisation
We present a prototype realisation of our methodology which is
currently under development as an Eclipse plugin. To evaluate our
approach, we use the FileZilla client v.11, a mature open-source
C/C++ application, with a non-trivial code base and several func-
tional and non-functional requirements. FileZilla uses a number of
third-party libraries, including TinyXML for reading and updating
XML files that keep server sites and interface-related properties.

At first, we obtained the ASTs of the FileZilla source code using
the parsing facilities provided by Eclipse CDT4. The automated
analysis of these ASTs enabled the identification of hotspots, i.e.,
FileZilla code elements that access the legacy XML library. We also
gained insight into the usage level of this library by the system.

We adapted the code city metaphor [12] to visualise the informa-
tion extracted during the analysis stage. Figure 3 shows a represen-
tation of FileZilla using our adapted city metaphor. We employ the
basic metaphor semantics, i.e., represent an application as a city, a
package/sub-package as a district/sub-district, and a class as a build-
ing. However, we introduce new semantics that help visualising
the dependency level between the software system and the obso-
lete library, and capturing the hotspots. First, we use red-coloured
buildings to show a class that invokes the obsolete library, i.e., an
affected class. Second, we set the height of a red-coloured building

4https://www.eclipse.org/cdt
Visualisation Techniques. Software visualisation techniques are widely used in program comprehension; see for instance the recent survey in [8] about existing software visualisation approaches. This survey highlights the number of visualisation tools for supporting developers’ needs on dependency management. A few studies have attempted to provide visualisations of evolving software systems and their library dependencies. Kula et al. developed a heat-map metaphor, which can help maintainers to navigate to library dependencies and gives an overview of the users across the different versions of a library [6]. Wettel et al. have used the city metaphor to represent software projects and they conducted a controlled experiment with developers to examine how the subjects comprehend the structure of a program [12]. In addition to the existing related work, we are using the city metaphor to highlight areas of software projects using third-party libraries.

5 CONCLUSIONS

The increasing dependency of software systems on third-party libraries challenges system maintainability. Developers need guidance on identifying and modernising client source code that can be affected due to library obsolescence. We proposed a modernisation methodology that can help developers of client systems to maintain their source code when these obsolescence issues occur. Also, we introduced a visualisation approach, using the city metaphor, for visualising source code areas that invoke the legacy library. We demonstrated a prototype realisation of our methodology and applied it on the FileZilla client. In the future, we plan to provide a mechanism that can automatically adapt client source code to new versions of third-party libraries. We will also explore the automated test generation for the changed parts of client source code.

REFERENCES