RESEARCH AND OPTIONS OF PRACTICAL SOLUTION OF SOFTWARE SYSTEMS COMPATIBILITY PROBLEM

D.S. Polikashin, V.O. Georgiev
Kazan Federal University, Kazan.

Received on 14-08-2016

Accepted on 20-09-2016

Abstract:

This paper provides the results of the study on definition of a software conceptual scheme designed for automation of a process of original application code migration into the current versions of libraries used there. The basic artifacts involved in this process were found. The architectural requirements for the software which supports an application source code migration between different versions of libraries are developed. Specifications, functions and interactions between the main components of the software considered are represented.

Keywords: Software systems compatibility, code generation, code transformation, computer word processing, backward software compatibility problem.

Terms, symbols and abbreviations

<table>
<thead>
<tr>
<th>Term / concept</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem (s) leading to the loss of backward compatibility between the software versions</td>
<td>LBC (loss of backward compatibility)</td>
</tr>
<tr>
<td>Software which code must be modified for migration to a new version of the software component used in it</td>
<td>Target software</td>
</tr>
<tr>
<td>Abstract syntax tree</td>
<td>AST</td>
</tr>
<tr>
<td>Programming language</td>
<td>PL</td>
</tr>
</tbody>
</table>

1. Introduction

Modern software development methodology is based on several key principles:

1. Code modularity and reuse;

2. Short development cycles;
Continuous implementation.

These practices are widespread and have a huge positive effect increasing the dynamics of software development and its qualitative characteristics. Open source ideology and projects like github, sourceforge, stackoverflow [Appendices 1, 8, 9] and its other analogues directly contribute to this beneficial effect. They allow solving the problem with attraction of not only thousands of individual developers from around the world, but also the experience of key IT companies and research laboratories, such as Google, Yandex, IBM, Intel through software components provided by them [Appendices 3, 4, 5, 7, 12]. Since the beginning of their appearance, the latest ideas and developments fall directly into hands of programming community and begin to be used by developers in projects around the world. All this leads to such positive effects as the increase in the dynamics of generation of a final software and components for its creation, and their adaptation to a changeable environment. However, appropriate application of the best practices of the present and the past does not guarantee the absence of problems in the production of software and its continued support and development.[1, 2, 3, 4, 6, 7, 8, 9] Historically, it becomes clear that, sooner or later, in developing software products architectural and ideological problems appear that can be solved only by a change in the architecture of these programs. In some cases, it is impossible to perform these changes with maintaining backward compatibility with the previous versions. In the case of software components at the time of occurrence of these problems, they are generally present in the composition a variety of other programs. That, in turn, leads to many negative effects. Obvious examples are the releases of popular software components such as OpenCV, Node.js, AngularJS, Ruby, Qt, DirectX [Appendices 14, 15, 16, 17, 18].

Depending on whether the library is distributed in a source code form, or in binary form, both concepts of interface compatibility relevant.

1. Compatibility of the program versions presented in the form of source code means that each program which is compiled with an older version of the library should be also compiled with its new implementation.

2. Binary compatibility ensures that every program which uses the binary form of the old library, can also use the binary form of its new version.

Further we consider only backward compatibility of libraries provided in source code.

Some studies provide methods, techniques and programs to test backward program compatibility or equivalences. [1-10]
In this context, it is actual to create tools for the automatic migration of the application source code into the actual version of the library used.

Therefore, the aim of this study is:

1. Identify the concepts and components involved in the migration of software systems between the different versions used in these libraries.

2. Identify the key architecture requirements to be met by the software performing automated solution to the problem of migration between versions of libraries that do not have backwards compatibility.

3. Identify the architectural solutions for the problem based on the identified requirements.

2. Study of the Subject Area

A migration of software systems between the different versions of libraries used in them affects a number of different elements. The first is programming languages. Consideration of a programming language, namely its syntax, is one of the main elements on which it should be paid a close attention, because it is necessary to directly interact with not only in finding the differences leading to the loss of backward compatibility between software component versions, but also in the initial migration of a target software code between the different library versions used in it. Along with the basic syntax of a programming language, not less important are frameworks expanding their syntax (examples: Qt, Groovy, Kotlin, etc.) and other specialized libraries (examples: OpenCV, OpenCL, IntelTBB, Boost) which build their ecosystem around themselves. Second, it is the set of possible changes to the source code of a software component because a specific subset of these changes leads to a lack of inter-version backward compatibility [1].

Third, it is the engine algorithms for searching backwards compatibility loss. It is necessary to take into account the peculiarities of each backward compatibility loss for their most optimal automatic detection [1-10].

Fourth, algorithms for eliminating loss of backward compatibility in the source code of a target software. These algorithms are designed to simplify the target code migration from one version involved in library development into its other version, both by providing developers of detailed instructions and tips for conducting this process, and automatic transformation of the target application code to remove the backwards compatibility loss there where it is possible. Fifth, it is the ways of obtaining baseline data to compare versions of the libraries. The data source can be both the source code library, if any, and alternative sources of documentation files in different formats, or direct dialogue with their developer.
3. Basic Requirements for Architecture

Given the fact that in the process of developing software aimed at addressing the loss of backward compatibility, it is necessary to take into account a large number of different components such as:

1. Syntax:
   a. Programming languages;
   b. Frameworks and specialized software libraries;
   c. Formats for documentation submission;

2. Classes of problems leading to the loss of compatibility between the versions of the libraries;


Therefore, application architecture must satisfy the following conditions:

- Be scalable - that is to have the ability to add new entities and functions to the system without disturbing its basic structure. And so that making the most probable changes should require the least efforts, that is, provide an easy way to add the functionality of the following types:
  ○ Support not only new programming languages, frameworks, and other syntactic structures, including new versions and standards of programming languages.
  ○ New ways to find and eliminate losses of backward compatibility.

- Be flexible - that is, allow parallel operation of the components responsible for different functionality. Also important aspects are the efficiency of the system, its reliability, performance, and scalability.

4. System Architecture

It follows from an article by Mark Richards "Software architecture patterns" [5] that a microkernel-based architecture of the application has been selected to meet the above-stated requirements.

Picture 1.

This architecture consists of two components: the basic system (kernel) and plug-ins (see Figure 2). The kernel contains a minimum of business logic, but controls loading, unloading and launching the necessary plug-ins. Thus, the plug-ins are unrelated to each other.

Figure 2.
Because plug-ins can be developed independently of each other, the developed system will provide maximum flexibility and scalability. Independent development of plug-ins allows for easy testing of each of them that increases overall reliability of the entire system. Productivity of an application built on the basis of this architecture is directly dependent on the number of connected and active modules. Also, when greater performance is necessary, this architecture can be easily modified in the micro-service one by separation of the main logic from the plug-in in a separate service. In this case, the plug-in is used only for communication between the micro-service and a program kernel.

Examples of successful microkernel-based architecture are such software development tools as Eclipse IDE and Sublime Text.

The kernel of a developed system provides a command line interface to configure the kernel, and action on searching and loading plug-ins from an external storage. Also, the main purpose of the kernel is to work with loaded plug-ins:

- Install, enable, disable, and delete.
- Configuration.
- Coordination between the plug-ins themselves.
- Interaction of plug-ins and user.

According to the tasks performed plug-ins are divided into the following types:

1. Plug-ins focused on identification of the loss of backward compatibility. Their input is different versions of the library between which it is necessary to implement the target software migration. This plug-in class is divided into:

   a. Plug-ins which depend on the syntax of a particular programming language (framework, types of configuration files, and the structure of the documentation file). The result of operation for this kind of plug-ins is the abstract representation of the program text reflecting the key features required for finding a loss of backward compatibility between versions. These plug-ins carry out the following:

      i. Lexical and syntactic analysis of the text and building an abstract syntax tree [4, 6].

      ii. Convert abstract syntax tree to the format of the abstract model suitable for finding a loss of backward compatibility.

   b. Plug-ins processing specially prepared abstract models. Plug-ins of this class are specified by type of backwards compatibility loss. Each plug-in of that class conducts a search of differences between the texts of different versions of
Further migration of the testing program and semantic analysis to find a strictly defined loss of backward compatibility. The result of operation for this type of plug-ins is a list of changes that lead to a loss of backward compatibility [3, 6, 10].

2. Plug-ins which are responsible for elimination of a backward compatibility loss in the target code. These plug-ins are dependent on the syntax of the processed structure and aimed at:

a. Bringing the recommendations and guidelines for implementation of the program migration between versions of the library by the programmer. Recommendations can be presented as a separate document, as well as be integrated into the source code of the target program.

b. Implementation of the automated transformation of the source code of the target program and configuration files of the software assembly system.

Plug-ins working with the source code, or with text documentation take a syntactically correct construction as their input. Plug-ins are not intended to perform the functions of static and dynamic code analysis to detect errors.

5. Conclusions

Since in the process of developing software aimed at addressing the problem of backward compatibility loss it is necessary to take into account a large number of different components such as a variety of programming languages and algorithms to find and eliminate the backward compatibility loss, it is reasonable to create an architecture having improved flexibility and extensibility. Consequently, the architectural solution to meet these conditions is a microkernel architecture which implies an application kernel providing a minimum set of functions for interacting with the user and the plug-ins that implement basic functionality.

6. Resume

This article provides the results of a study aimed at identifying the main concepts and components involved with the migration of source codes of applications from earlier into the more recent versions of libraries used in them.

The basic requirements for the tools' architecture necessary for programmers to implement an automatic transfer of a source code of an application to the actual version of the library used. The basic architectural solutions are represented. The constituent components of the developed software are classified, and the main areas for further work are identified. Further work in this direction is to test the functionality of the considered architectural solutions by creating working models of the components of the software system presented in this work.
Acknowledgements

The work is performed according to the Russian Government Program of Competitive Growth of Kazan Federal University.

References


**Attachment**

1. GitHub https://github.com /
5. Intel Developer Zone https://software.intel.com
7. Microsoft Developer technologies https://dev.windows.com
8. Sourceforge http://sourceforge.net /
13. Node.js API changes between v0.10 and v4 https://github.com/nodejs/node/wiki/API-changes-between-v0.10-and-v4
14. Angular JS Developer Guide. Migrating from 1.3 to 1.4 https://docs.angularjs.org/guide/migration
17. OpenCV 3.0 API changes http://docs.opencv.org/master/db/dfa/tutorial_transition_guide.html#gsc.tab=0