EVALUATING A FRAMEWORK FOR THE DEVELOPMENT AND DEPLOYMENT OF EVOLVING APPLICATIONS AS A SOFTWARE MAINTENANCE TOOL

Georgios Voulalas and Georgios Evangelidis
Department of Applied Informatics, University of Macedonia, 156 Egnatia St., Thessaloniki, Greece

Keywords: Software Evolution, Runtime Evolution, Dynamic Applications, Runtime Compilation, Software Maintenance, Java ClassLoader.

Abstract: In our previous research we have presented a framework for the development and deployment of web-based applications. The framework enables the operation of multiple applications within a single installation and supports runtime evolution by dynamically recompiling classes based on the source code that is retrieved from the database. The feasibility of our solution has been successfully verified with the use of an architectural prototype. Given the importance of the maintenance activities in the software lifecycle, in this paper we are going to evaluate our framework as a software maintenance tool and position it in the domain of software evolution with a use of a related taxonomy.

1 INTRODUCTION

It is impossible to produce systems of any size which do not need to be changed. Once a software system is put into use, new requirements emerge and existing requirements change as the business running that software changes. Parts of the software may have to be modified to correct errors that are found during its operation, and/or improve its performance or other non-functional characteristics. All of this means that, after delivery, software systems evolve (Somerville, 2000).

A great part of the research in the area of software evolution has been carried out by Lehman and Belady (Lehman and Belady, 1985). Their research resulted in a set of 'laws' (Lehman's Laws) concerning system change that are regarded as being invariant and widely applicable. The proposed laws were derived from measurements conducted upon large software systems. The first two laws are the most important. They state that evolution is required in order to cope with the continuously changing requirements but inevitably makes the system more complex and degrades its structure.

In (Warren, 1998) three main types of software change are identified:

- Software maintenance: Changes to the software are made in response to errors or changed requirements but the core structure of the software is not modified.
- Architectural transformation: It involves significant modifications to the architecture of the software system.
- Software re-engineering: The system is changed in order to become easier to understand and evolve. System re-engineering may involve some structural modifications but does not usually involve major architectural changes.

From the three types of software change listed above, software maintenance is the most common. Software maintenance is defined in IEEE Standard 1219 (IEEE, 1993) as: "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." Note that the term software evolution lacks a standard definition and *it is usually used as a preferable substitute for maintenance*. In practice, there isn't always a clear distinction between these different types of maintenance. It is difficult to find up-to-date figures for the relative effort devoted to the different types of maintenance. A rather old survey by Lientz and Swanson (Lientz and Swanson, 1980) discovered that about 65% of maintenance was concerned with implementing new or modified

Voulalas G. and Evangelidis G. (2009).
EVALUATING A FRAMEWORK FOR THE DEVELOPMENT AND DEPLOYMENT OF EVOLVING APPLICATIONS AS A SOFTWARE MAINTENANCE TOOL.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 31-38

requirements, 18% with changing the system to adapt it to a new operating environment, and 17% to correct system faults. Similar figures were reported by Nosek and Palvia (Nosek and Palvia, 1990) ten years later. Updating the system in order to cope with new or changed requirements consumes most of the maintenance effort. The costs of system maintenance represent a large proportion of the budget of most organisations that use software systems. In the 1980s, Lientz and Swanson found that large organisations devoted at least 50% of their total programming effort to evolving existing systems. McKee (McKee, 1984) found that the amount of effort spent on maintenance is between 65% and 75% of total available effort.

Summarizing the above-mentioned findings we can state that *after the delivery of a software system significant effort is inevitably devoted to the implementation of new features, the modification of existing features and the correction of bugs***.**

In the traditional approach to software maintenance, the programmer edits or extends the source code of a software system, and re-compiles (possibly incrementally) the changes into a new executable system. The running software system has to be restarted for the change to become effective. However, in many cases it is not acceptable to frequently shut down the system in order to perform changes, therefore, it must be possible to modify it while at runtime.

Runtime adaptations are supported by programming languages, such as CLOS, Smalltalk, and Self (Zdun, 2004). These dynamically typed programming languages provide both a programming environment and a program execution environment, allowing one to influence the language behaviour from within a program. Similar features are provided by a number of scripting languages, including Tcl, Python, Perl, and Ruby. Those features are mostly used in an ad-hoc way and not as a distinct evolution technique. Modern statically-typed commercial programming languages such as Java and C#, through various concepts such as typing, encapsulation and polymorphism, encourage programmers to write code that should be easier to maintain and evolve (Evans, 2004). However, focus is placed on the non runtime issues of reusing program source code and trying to make it easier to manipulate the codebase of a particular application. Although it is possible to dynamically update both Java and C# programs neither of these languages directly address the issues of runtime evolution by defining an evolution model.

Toward this need, in (Voulalas and Evangelidis, 2008a), (Voulalas and Evangelidis, 2007) and (Voulalas and Evangelidis, 2008b) we introduced a development and deployment framework that targets to web-based business applications and *supports runtime adaptations*. The framework takes advantage of the options that the Java Programming Language provides for the creation of dynamic applications and operates as a runtime evolution infrastructure. In this paper we evaluate this framework and position it within the software evolution domain. For this reason we use one of the available taxonomies that are related to software evolution.

The paper is further structured as follows. In Section 2, we outline our framework. In Section 3, we present the selected taxonomy, and in Section 4 we apply it in our framework. We discuss the main conclusions in Section 5.

2 THE CORE CONCEPTS OF OUR FRAMEWORK

2.1 Database Model

The framework is *structured on the basis of a universal database model (meta-model).* As presented in Figure 1, the database model is divided into three regions.

- Region A holds the functional specifications of the modelled application and includes the following entities: Classes, Attributes, Methods, Arguments, Associations and Imports (class dependencies). For example, the method definition consists of a name, a return type, a set of arguments, and a body. For each class the entire source code is stored in the database.
- For each table of Region A a companion table using "_versions" as suffix is included in Region A′. This enables us to keep all different versions of the modelled applications.
- Region B holds data produced by the applications and consists of the following tables: Objects, AssociationInstances and AttributeValues. Those tables are structured in a way that is independent of the actual data structure of the applications. Thus, changing the database structure of a modelled application (e.g. adding a new field in an existing table or creating a new table) does not affect Region B.

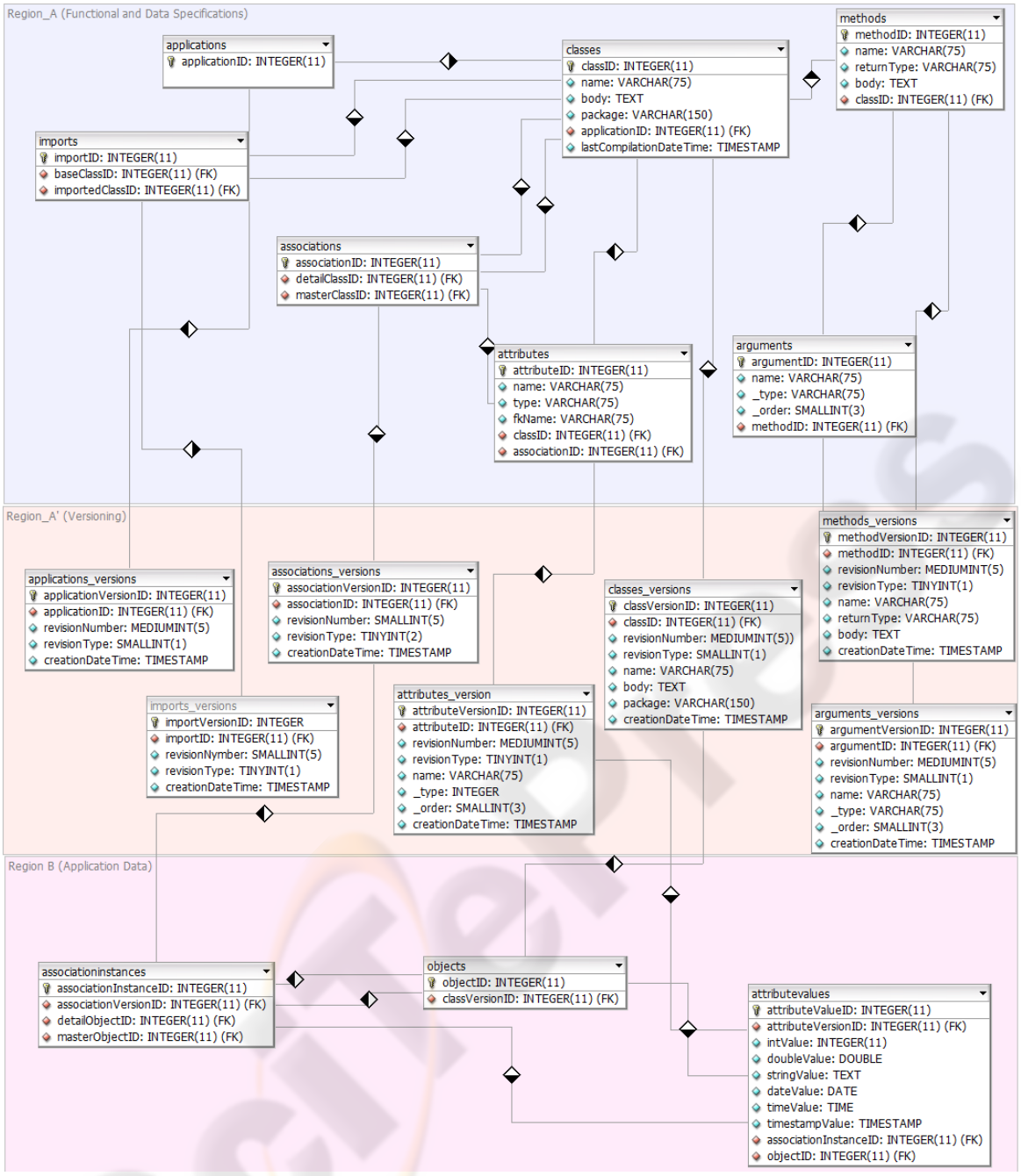

Figure 1: The Database Model.

2.2 Development Process

The framework facilitates the development process as follows:

- Since the Database tier is common for all applications, generic methods for the materialization and dematerialization of objects are provided. *The user does not have to write SQL code and interfere at all with the database layer.*
- For the development of the Application tier (i.e., business objects) the user should be provided with a custom editor implementing a moderated development environment. This editor should enable the user to take advantage

of all pre-build mechanisms that are supported by the framework.

- For the User Interface tier, a more flexible and less moderated approach is proposed in order for the user to be able to freely and creatively implement the user interface of the application.

2.3 Deployment

The framework operates as a deployment platform that hosts multiple applications within a single installation. There always exists one deployed application, independently of the actual number of running applications.

The running application constitutes of generic components and application-specific components that are produced by the runtime compilation of the application-specific source code. The generic components *operate as an abstraction layer that allows application-specific classes and their methods to be utilized.*

2.4 Runtime Evolution & Versioning

Since source code is retrieved from the database and compiled at runtime, we can deal with business logic changes at deployment time without interrupting the operation of the application.

What's more, we can anytime refer to a previous version of an application and examine old data in its real context (i.e., within the version of the application that created this data) by retrieving the corresponding data instances from the database, without the need for maintaining additional installations (one for each different application version).

Finally, we can easily support a policy for the management of active instances that allows existing threads to continue to call old code, whereas new threads to call new code.

2.5 Architectural Prototype

In order to verify the feasibility of our proposal, we have developed the core functional and data mechanisms. The underlying database schema resides in MySQL. For the functional components, Java was an obvious choice for us to consider since it supports two features that are essential for the implementation of our framework: reflection and runtime compilation of source code.

2.5.1 Reflection

Using the Java reflection API included in the Java Development Kit (JDK) version 1.1 or higher, a programmer can obtain meta information about the Java objects at runtime. That is, the programmer can look inside a Java object at runtime and see what variables it contains, what methods it supports, what interfaces it implements, what classes it extends—basically everything about the object that is known at compile time. The Class class supports getMethods, getMethod, getDeclaredMethods, getDeclaredFields, getFields, and getField for user code to call. User code can access the fields or the methods of an object via field objects or method objects. Similarly, the Java reflection API supports method invocations and accessing of field values.

2.5.2 Runtime Compilation of Source Code

The javax.tools package, added to Java SE 6 as a standard API for compiling Java source, enables the addition of dynamic capabilities that extend static applications (Biesack, 2007). It is an approved extension of Java SE, which means it is a standard API developed through the Java Community Process (as JSR 199). The main benefit is that the developer uses what he better knows: Java source, not bytecodes. He can create correct Java classes by generating valid Java source without needing to worry about learning the more intricate rules of valid bytecode or a new object model of classes, methods, statements, and expressions.

3 SELECTING A TAXONOMY

Several taxonomies related to software evolution exist (Lientz & Swanson, 1980), (Chapin & Hale, 2001), (Pukall & Kuhlemann, 2007). These taxonomies can be used for evaluating frameworks, tools and techniques within the domain of software evolution. Most of them focus on the purpose of the change. In order to evaluate our framework, we have selected a taxonomy that *focuses on technical aspects* and is based on the characterizing mechanisms of change and the factors that influence these mechanisms (Mens & Buckley, 2003). The selected taxonomy *is more comprehensive, in comparison to the others***,** as it includes several properties organized around four logical groups: temporal properties, object of change, system properties, and change support.

3.1 Temporal Properties (when)

"The 'when' question addresses temporal properties such as when a change should be made, and which mechanisms are needed to support this." (Mens & Buckley, 2003) (Table 1).

Table 1: Temporal Properties.

Dimension	Supported types
Time of change	▪ **Compile-time evolution (alternatively called static evolution):** the traditional approach to software maintenance, where the programmer edits the source code and re-compiles the changes into a new executable system. The running software system has to be shut down and restarted for the change to become effective. ▪ **Runtime evolution (also called dynamic evolution):** the software change occurs during execution of the software. The system evolves dynamically for instance by hot-swapping existing components or by integrating newly developed components without the need for stopping the system. ▪ **Load-time evolution:** changes are incorporated as software elements that are loaded into an executable system. It is well-suited for adapting statically compiled components dynamically on demand, so that they fit into a particular deployment context. Depending on whether load-time coincides with runtime or it coincides with a system's start-up time, load-time evolution is either dynamic or static.
Change history	▪ **Completely un-versioned systems:** changes are applied destructively so that new versions of a component override old ones. ▪ **Systems that support versioning statically:** new and old versions can physically coexist at compile- or load-time, but they are identified at runtime and therefore cannot be used simultaneously within the same context.

Table 1: Temporal Properties (cont.).

Change history	▪ **Fully versioned systems:** allow different versions of one component to coexist at runtime. This is particularly important for the dynamic evolution of systems, since safe updates of existing components often require that new clients of the component use the new version whereas existing clients of the old component continue to use the old one.
Change frequency	▪ Continuously ▪ Periodically ▪ Arbitrary

3.2 Object of Change (where)

The second group in the selected taxonomy addresses the 'where' question. "Where in the software can changes be made, and which supporting mechanisms are required?" (Mens & Buckley, 2003) (Table 2).

Table 2: Object of Change.

Dimension	Supported types
Artifact	Artifacts that are subject to changes can range from requirements through architecture and design, to source code, documentation and test suites. They can also be a combination of several or all of the above.
Impact	Very local to system-wide changes.

3.3 System Properties (what)

"A logical grouping of factors that influence the kinds of changes allowed as well as the mechanisms needed to support these changes has to do with the properties of the software system that is being changed, as well as the underlying platform, and the middleware in use." (Mens & Buckley, 2003) (Table 3).

Table 3: System Properties.

Dimension	Supported types
Availability	For some software systems it is not acceptable that their operation is interrupted occasionally in order for changes to be implemented by modifying or extending the source code.
Safety	▪ **Static safety** is provided if we are able to ensure, at compile-time, that the evolved system will not behave erroneously at runtime. ▪ **Dynamic safety** is provided if there are built-in provisions for preventing or restricting undesired behaviour at runtime.

3.4 Change Support (how)

"During a software change, various support mechanisms can be provided. These mechanisms facilitate the analysis, management, control, implementation and measurement of software changes." (Mens & Buckley, 2003) (Table 4).

Table 4: Change Support.

Dimension	Supported types
Degree of automation	▪ Manual ▪ Partially automated ▪ Automated
Degree of formality	▪ Implemented in an ad-hoc way ▪ Based on an underlying mathematical formalism
Change Type	▪ **Structural changes**: changes that alter the structure of the software. In many cases, these changes will alter the software behaviour as well. A distinction can be made between adding new elements to the software, removing elements from the software, and modifying (e.g., renaming) an existing element. ▪ **Semantic changes:** can either be semantics-modifying (such as refactoring) or semantics-preserving (such as the replacement of a 'for loop' by a 'while loop').

4 APPLYING THE SELECTED TAXONOMY

Having presented the core elements of the selected taxonomy, we are going to apply the taxonomy to our framework in order to evaluate its usability and identify missing properties or properties that can be improved. For each of the framework dimensions, we describe *the extent to which it is supported by our framework*. In Table 5, we summarize our evaluation.

4.1 Temporal Properties (when)

Using this group of properties we will identify the phase changes occur at, the frequency of changes and the way the different software versions (produced during evolution) are handled.

4.1.1 Time of Change

Technically our framework is based on Java's ClassLoader architecture that is a prominent example of load-time evolution mechanism. However, since classes are *loaded at runtime and changes become effective without the need of restarting the application*, it is clear that our framework supports *runtime changes*.

4.1.2 Change History

The database model upon which our platform is based, incorporates a simple data versioning technique (inspired by the Jboss Envers project) that allows us to keep all different versions of the modelled applications. What's more, it enables the identification of the objects that have been produced from a specific version of an application and the identification of the version that should be invoked in order for a specific object to be processed. In other words, multiple versions of the same application can *co-exist at runtime level.* Thus, our framework *provides a fully versioned environment.*

4.1.3 Change Frequency

Our framework *does not impose any restriction related to the frequency of changes*. The frequency of changes is arbitrary, since changes are triggered by the users.

4.2 Object of Change (where)

This group of properties will help us define the subject of changes and the granularity of supported changes.

4.2.1 Artifact

Our platform supports changes *directly (and only) to source code.*

4.2.2 Impact

Changes can range from *local to system-wide.*

4.3 System Properties (what)

We will examine the following two attributes: the availability of the deployed systems while maintenance takes place and the runtime safety.

4.3.1 Availability

Our framework can successfully support the evolution of software systems without interrupting their operation. In other words, in ensures *high availability* of the deployed applications throughout their lifecycle.

4.3.2 Safety

Since source code can be changed in an arbitrary way and classes are recompiled and loaded just before the execution of a method, *it is quite difficult to prevent undesired functionality at runtime level.* However, we should work on mechanisms that will restrict the runtime errors. One such method could be the provision of a test platform that will run in parallel with the main deployment platform and will allow new versions to be tested by the developers before becoming available to the end-users. Since the proposed platform already supports the parallel deployment of different versions of the same application, the implementation of a test platform will mainly require the distinction between production and test versions.

4.4 Change Support (how)

This group of properties will help us identify the supported degree of automation in the implementation of changes and the covered types of changes.

4.4.1 Degree of Automation

Our platform is a *semi-automated tool*. Taking as a basis the 3-tier architecture that is the most outstanding architectural paradigm, changes are implemented as follows:

- The Database tier is generic for all applications. The developer does not write SQL code, neither for the creation of the database, nor for the manipulation of data. A structured API that includes generic methods for inserting / updating / deleting objects, along with methods for retrieving objects using multiple filters is provided, facilitating the user during the development process. Changes in the database structure of an application result in data changes in the underlying meta-model and are transparently and *automatically* handled by the API.
- In the final prototype of the platform, the development of the Application tier should be supported by an editor supporting a list of custom features, such as code generation and auto-complete features. Thus, initial implementation and changes in the application tier *should be implemented in a semi-automated manner.*
- For the User Interface tier, we have selected a more flexible and less moderated approach. The developer should be able to freely implement the user interface of the application. *Changes in the user interface could be only manually supported.*

4.4.2 Degree of Formality

Our platform *has no underlying mathematical foundation.* It is very interesting to try to identify parts of the development process that could be formalized. The interaction of the functional layer with the database layer and the way changes in one level are propagated to the other seems to be such a domain.

4.4.3 Change Type

Our platform puts no constraints on the types of change that can be made to the software system. It can be a semantics-preserving or semantics-changing change. It can be an addition, subtraction, or alteration at functional or database level. However it doesn't support all types of changes with the same degree of automation.

Table 5: Evaluation of our platform based on the selected taxonomy.

Group	Dimension	Support
Temporal Properties	Time of change	Runtime
	Change history	Fully versioned
	Frequency	Arbitrary
Object of Change	Artifact	Source code
	Impact	Global changes
System Properties	Availability	No down-time
	Safety	Low
Change Support	Automaton	Semi-automatic
	Formality	No
	Change type	Any

5 CONCLUSIONS & FURTHER RESEARCH

By applying the taxonomy on the development and deployment platform that we have presented in our previous research efforts we are now able to evaluate it as a software maintenance mechanism and identify its strengths, along with its weaknesses. The most important strengths are:

- Our platform is a *run-time* change support mechanism since deployed platforms do not need to be restarted in order for changes to become effective. This is very important feature for systems that undertake frequent changes and / or for systems that are business critical and require *high availability*.
- Our platform is a *fully versioned* change support mechanism since it supports the runtime coexistence of multiple versions of the deployed applications (within a single installation of the platform).
- Our platform should be considered as a *semi-automated change support tool* as it will support the developer in the implementation of changes at the database and application (in a future version) level.

On the other hand, the most important weakness is that since source code can be changed in an arbitrary way and classes are recompiled and loaded just before the execution of a method, *the deployed applications seem to be vulnerable to runtime errors.* In order to limit the possibility of unwanted runtime scenarios we should elaborate on auxiliary mechanisms at two-levels: (a) at implementation level in order to assist the developer, (b) at test-level in order new versions to be thoroughly tested before there are delivered to end-users.

REFERENCES

Biesack, D., 2007. Create dynamic applications with javax.tools. http://www.ibm.com/developerworks/java/library/j-jcomp/index.html

Chapin, N., Hale, J., Khan, K., Ramil, J., Than, W.-G., 2001. Types of software evolution and software maintenance. *Journal of software maintenance and evolution*, pages 3–30.

Evans, H., 2004. DRASTIC and GRUMPS: design and implementation of two runtime evolution frameworks. *IEE Proceedings - Software* 151(2): 30—48.

IEEE, 1993. *IEEE Std. 1219: Standard for Software Maintenance*. Los Alamitos CA., USA. IEEE Computer Society Press.

Lehman, M.M., Belady, L.A., 1985. *Program Evolution – Process of Software Change*. Acad. Press, London.

Lientz, B. P., Swanson, E. B., 1980. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley.

McKee, J., 1984. Maintenance as a function of design. *Proceedings of the AFIPS National Computer Conference*, 187—193.

Mens, T., Buckley, J., Zenger, M., Rashid, A., 2003. Towards a taxonomy of software evolution. In *Proc. 2nd International Workshop on Unanticipated Software Evolution*, Warsaw, Poland.

Nosek, J., Palvia, P., 1980. Software maintenance management: changes in the last decade. *Journal of Software Maintenance: Research and Practice* 2 (3), 157—174.

Pukall, M., Kuhlemann, M., 2007. *Characteristics of Runtime Program Evolution*. RAM-SE, 51—58.

Sommerville, I., 2000. *Software Engineering*. 6th Edition. Addison-Wesley.

Voulalas, G., Evangelidis, G., 2007. A framework for the development and deployment of evolving applications: The domain model. In *2nd International Conference on Software and Data Technologies (ICSOFT)*, Barcelona, Spain.

Voulalas, G., Evangelidis, G., 2008. Introducing a Change-Resistant Framework for the Development and Deployment of Evolving Applications. In Filipe, J., Shishkov, B., and Helfert, M., editors, *Communications in Computer and Information Science*, Volume 10, 293–306. Springer Berlin Heidelberg.

Voulalas, G., Evangelidis, G., 2008. Developing and deploying dynamic applications: An architectural prototype. In *3rd International Conference on Software and Data Technologies (ICSOFT)*, Porto, Portugal.

Warren, H., 1988. Sharing the Wealth: Accumulating and Sharing Lessons Learned in Empirical Software Engineering Research. *Empirical Software Engineering* 3(1), 7—8.

Zdun, U., 2004. Supporting incremental and experimental software evolution by runtime method transformations. *Sci. Comput. Program*, 52, 131—163.