Instance Migration in Dynamic Software Update

Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse

Instance Migration in Dynamic Software Update

Pablo Tesone
Inria Lille–Nord Europe, France
Mines Douai, IA, Univ. Lille, France
pablo-adrian.tesone@mines-douai.fr

Guillermo Polito
Inria Lille–Nord Europe, France
guillermo.polito@inria.fr

Luc Fabresse
Mines Douai, IA, Univ. Lille, France
luc.fabresse@mines-douai.fr

Noury Bouraqadi
Mines Douai, IA, Univ. Lille, France
noury.bouraqadi@mines-douai.fr

Stéphane Ducasse
Inria Lille–Nord Europe, France
stephane.ducasse@inria.fr

Abstract

Nowadays, there are more and more applications that need to run uninterruptedly. This need requires minimizing the downtime to add new features or fix bugs. Dynamic Software Update (DSU) solutions allow updating applications while they are executing. A common concern in all DSU solutions is the migration of the application's state. This migration should guarantee the coherence of the state between versions by either adding, removing, preserving or transforming state. In an object-oriented environment, this state is represented by instances.

In this paper, we analyse the features that a DSU solution should have with the objective to understand the operations a DSU should provide. Our analysis focuses on the migration of instances. Then, we identify the Meta-Object Protocol (MOP) that a programming language should expose to support these operations. We scope our proposal to DSU solutions for class-based programming languages with eager and atomic instance migration.

Finally, we validate our proposal with a prototype DSU implemented in the Pharo programming language using the identified operations.

Categories and Subject Descriptors D-3.2 [*Language Classifications*]: Object-oriented Languages

General Terms Dynamic Software Update, State Migration, Object Oriented Programming

Keywords DSU, OOP, Pharo, Meta-programming.

1. Introduction

Software must constantly evolve, otherwise they become obsolete [1, 7]. Adding new features, improving performance or just fixing bugs and security failures are common daily tasks. Keeping software up-to-date is a truly challenging task. This challenging task is usually addressed by different techniques that take place during the development of the system but also during its execution [13]. In this latter case, the classical process is: stop, install and restart implying a downtime of the application and often the loss of the execution state. But, for an 7days/24hours online service, long time scale simulations or a life support computer, restarting the system may either have dramatic side effects or be very expensive.

Instead of restarting the whole system, updates may be applied gradually. The basic idea it to turn the *stop, install, restart* cycle into a simple *update* action [14]. This usually requires more complex manual scheme of updates and special handling of running instances and processes. Such systems are commonly called Dynamic Software Update (DSU) [4, 13]. A DSU is meant to manage the migration from version 1 to version 2 of a software while it is running. Of course, the software should work *correctly* after the migration.

In the context of object-oriented systems, one of the main challenges with dynamic software update is to handle objects migration. When changing the software from version 1 to version 2, you might need your objects to adopt a new structure. This problem typically arises in live development environments in which programs are modified while they are running. Having a proper DSU integration in such environments will help developers minimize the times the application has to be restarted after a change and improve the interactive programming experience. This integration should look for *programmer transparency* [9], allowing the programmer to have all the benefits, but minimizing the extra work.

Traditional dynamic languages such Smalltalk [3] or CLOS [5] offer API to support a simple migration of instances. However, the support is often minimal, for example

Meta'16 October 30, 2016, Amsterdam, Netherlands

an instance drops any field that is not available in the target class or the new fields are left uninitialized.

Meta-operations are often the building blocks of DSU solutions because they provide introspection and intercession of objects that constitute a running application. The availability of Meta-operations constrains the DSU. So that, the programming language should expose a Meta-Object Protocol (MOP) [6] to support DSU capabilities.

In this paper we focus on the migration of instances. We first provide an overview of DSU solutions (Section 2) from the instance migration point of view. Then in Section 3, based on the previous analysis we propose a DSU architecture with an identified required MOP. Our proposal is general, since the DSU can be easily implemented in any object-oriented language providing the same MOP. The proposed DSU is designed for class-based programming languages with eager and atomic instance migration. After that, we present the validation of our solution. We validated our solution through a prototype developed in the Pharo programming language (Section 4).

The remainder of this paper is organized as follows. Section 5 discusses this solution regarding constraints and properties previously explained. Then, Section 6 compares our proposition with existing solutions. Section 7 concludes this paper and presents some future work.

2. DSU Overview

Updating a running system, minimizing the impact on provided services is the core objective of a DSU approach. Updating an application consists in: changes in the behaviour *i.e.,* removing, adding, and modifying code and changes in the data *i.e.,* modifying the structure and the values of objects.

In this section, we analyse the steps performed in the use of a DSU solution (Subsection 2.1). Then we describe the structure of a patch (Subsection 2.2), and we present a general DSU architecture (Subsection 2.3).

Next, as this paper is centred in the migration of instances we describe the possible changes in the instances during an update (Subsection 2.4) and the properties a DSU should guarantee regarding the migration of instances (Subsection 2.5).

2.1 Main Steps of a DSU Process

The following steps describe the complete life-cycle of an update. The development process can be performed in a separated environment *i.e.,* one environment for production and other for development, or in the same environment *i.e.,* using a live programming environment.

1. The programmer generates a new version in the development environment. It is desirable that the development is performed without thinking the steps needed to apply the update.
2. After having the running version the programmer generates a patch. The way to generate the patch and the patch content is analysed in Subsection 2.2.
3. The DSU engine loads the patch, without applying the changes into the target environment.
4. The DSU engine decides which moment is best to install the patch and hence perform the update.
5. The DSU engine applies the changes. Depending on the DSU implementation, the whole application is suspended or just the modified part. In this step the data, metadata and code of the application has to be updated.
6. The DSU performs any clean up required.

2.2 Patch Content and Generation

A patch is a piece of software describing the changes from a version to another. A patch is generated automatically or it is written by the developer. A general definition of a patch consist of the following elements:

- A set of changes to the application code and to the structure of the instances.
- Scripts to execute before and after the changes are applied. The former prepares the environment for the execution of the changes. And the later cleans up and initializes the environment, including a set of operations needed to migrate the instances from the previous version to the new version.

2.3 General DSU Architecture

DSU implementations [2] provide different operations to allow the update of running software. These operations are constructed using the meta-level operations present in the language and a number of virtual machine primitive operations (Figure 1).

Although most of these meta-operations may be present in the platform, a useful DSU solution should add value providing abstractions and high level operations to ease the update process.

The design of this API is usually focused on the expression of updates. Allowing the programmer to express all the changes needed in the update, hiding the details of the low level operations and primitives used.

The availability of these basic meta-operations determines the selection of strategies the DSU provides or even if some properties of the DSU can be achieved.

2.4 Instance Migration Changes

A full support for object migration in an object-oriented environment has to be able to manage arbitrary combinations of the following elementary changes. These changes are composed in different ways, allowing the creation of complex changes. Including changes in many classes at the same time, for example complex automatic refactorings.

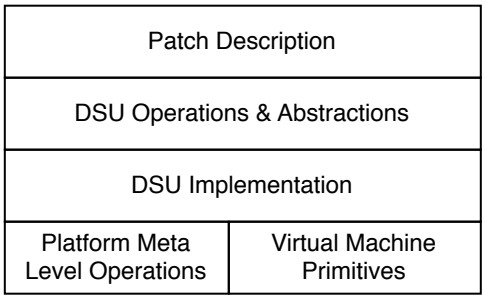

Figure 1. General Architecture of a DSU

Adding new instance variable. Adding a new instance variable on an existing instance means extending the structure of the object and filling the room with a value. Neither initializing the new variable with the null pointer nor using the value assigned in the construction are useful for existent objects. The decision of the value for the new instance variable depends on the update performed.

Removing instance variable. When removing an instance variable there are two options. Either the object is resized to fit its new layout and the value is dropped or the object keeps the value but the instance variable is made obsolete as it is not a property of its class any more.

Renaming instance variable. From the structure point of view this is equivalent to removing an instance variable and adding another one. But doing so, we loose the value held in the old instance variable. Therefore, the system needs to be able to transfer state from an instance variable about to be removed to a newly added instance variable.

Value Change. When the object structure is updated, it might not be a structural change. Changes are application specific and only affect the value stored in an instance variable without modifying the structure of the object. These migrations may need to compute the new state of the object in various manners that involve all the data available in the object. The way the new values are calculated are application dependent. For example when in an object representing a product we store a price as a number. In a later version we store the price as an object (containing the number and the currency). The structure of the instances are not modified (both versions have the *price* variable), but the instances should be migrated and the migration is application-dependant *i.e.,* what is the default currency.

Class Hierarchy Changes. A class hierarchy change can generate or not changes in the structure of the instances. The DSU should detect when these changes affect the structure of the instances and migrate the affected instances.

2.5 Instance Migration Properties

Instance migration is handled in various ways in the different solutions. However, all of them share a set of properties. Hicks [4] gives a nice set of properties that a DSU should exhibit. We provide here a more specific set of properties focused on instance migration:

Application Dependent Migration Policies. The migration process needs to know what the steps to perform are to get a valid new instance from an existing old instance, we call it migration policies. These policies are generated automatically or provided by the developer. General migration policies are not useful because each version of the updated software requires migrations that are application dependent.

System and Application Level Validations. DSU should provide a way of expressing invariants that should guarantee before and after the update process. These invariants should be represent the healthy state of the environment and the application running on this. Most of these invariants are valid in the different updates of the application. There are application dependent invariants and there are also invariants that should guaranteed for the health of the system.

Atomicity. It is desirable that a DSU guarantees that all the operations are correctly performed or none of them is performed. In that way the DSU prevents a partial migration of the instances.

Eager / Lazy. The instances can be migrated all at once, performing an eager migration, or it can be migrated as soon as it are used by the new version of the updated application.

Coherence. During the migration, both old and new objects are alive at the same moment and represent the same piece of data; but each process only knows one of those. If a process tends to affect one of these objects while it is under migration, the system might get to an invalid state. The system has to ensure that there is no possibility of error by using old instances

Causality Guarantees. A DSU mechanism has to ensure that the relationship between the objects and the meta objects are ensured. Ensuring that the dependencies between live instances is intact after the application of the update.

Composability. Some migrations are generic and recurring (e.g. refactoring actions) and we want to reuse them. Furthermore, more complex and complete migrations are composed of elementary operations, first-class updates that, aggregated one by one, may allow you to define more complex migrations.

Use Cases We have detected two main situations where the DSU should be used to guarantee the correct execution of the application. When updating a running application and when using it to support a live programming development. It is interesting to analyse in which of the two use cases (or maybe both) a tool can be applied.

3. Our Reflection-based DSU

In this Section we present a general implementation of a Instance Migration solution based on a limited set of meta-operations.

Our proposed solution updates software transactionally. To achieve this the proposed solution performs all the changes in a new namespace. The new namespace is a copy of the running environment including all the classes, objects and global state of the application.

Objects are modified following the operations described in a patch. A patch interacts with the DSU through an API. This API is designed to express all the changes in the instances (Subsection 3.2).

A patch delimits the transaction through the use of special operations in the DSU. Until a patch invokes the *commit()* operation no change is performed in the running application. When a patch invokes *commit*, the DSU performs all the changes and replaces the old namespace with the new one.

The following steps describes how the proposed DSU works after the *commit* operation:

1. The DSU creates a copy of the namespace.
2. The DSU applies all the changes in this new namespace.
3. The DSU migrates the modified objects. Using migration policies (Subsection 3.4).
4. The DSU runs all the validations, if any problem is detected, the DSU discards the new namespace and aborts the update process.
5. The DSU swaps all the old instances with the new instances, replacing the current namespace with the new one.

After the DSU performs all the steps the normal execution of the updated application is resumed.

The replacement of the old namespace with the new one is performed through the execution of a bulk swap operation. This operation replaces all the references to the old objects with references to the new objects. Replacing all the old objects at once provides the atomic property to the DSU.

The validations and migrations of modified objects are application dependent. So that, the programmer needs to write them. We have detected that these implementations are reused in many updates. Subsection 3.4 presents our proposed way to reuse them.

3.1 Copied Namespace Contents

A running application can have global state. In a Object Oriented environment the global state of an application is stored in globally accessible objects. These objects have a binding in a namespace.

By just migrating the namespace and the objects referenced by it, all the global state of the application is migrated. So that, the global state can be migrated without affecting the running application. These global state includes all the accessible objects and classes.

The amount of objects to copy to get a new namespace is analysed in the Section 5.

3.2 DSU Metalevel Operation Protocol

Our DSU solution provides the following operations to express the changes from a version to an updated version. Figure 2 decribes the operations. The following subsections explains the different operations.

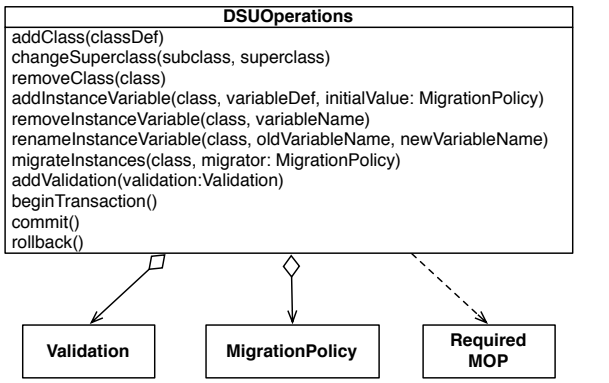

Figure 2. Operations provided by the DSU

3.2.1 Instance Structure Operations

Changing the structure of a class impacts all its instances. As this change also impacts all the subclasses of the modified class, the DSU should propagate this change down the hierarchy of the modified class. Any time one of these changes occur, the DSU should migrate all the instances of the modified class and the instances of all the subclasses. The operations in this category are listed below. These same operations apply to class side variables.

Add Instance Variable. When an instance variable is added to a class, an initial value should be given. Most of the solutions initialize this value with *Null* or using the constructor, but this is not possible for already created objects. So the developer should give a migration policy to calculate the value of the new variable.

Remove Instance Variable. When an instance variable is removed, there are no particular operation to perform, just the common behaviour of all the operations in this category.

Rename Instance Variable. When an instance variable is renamed the DSU should preserve the value of the variable. It is the combination of the Add and Remove Instance Variable, but keeping the value unchanged. In this scenario, a common migration policy can be applied to keep the value of the renamed variable.

3.2.2 Class Hierarchy Operations

Changing class hierarchy triggers a number of instance changes. For example if the superclass of an existing class is changed, the instance structure of the class is changed also and a migration of the instances should be performed. Also these changes should be propagated through the hierarchy.

All these changes are expressed in the patch by the developer using the instance migration operations. But we decided

to include some high-level operations to express changes in the class hierarchy. The DSU can identify these changes and detect if there is need to migrate instances.

Add Class. When a new class is added to the running system, there is no instances of this class to be migrated. This is true because as the class is new there is no existing objects of this class. However, the class side instance of the new class should be handled correctly and initialized to reflect the current state of the application. If it is needed, the DSU should let the developer to provide a migration policy to initialise the new class in the context of the running application.

Change Superclass. When it is needed to change the superclass of an existing class all the instances are affected. These changes can include adding new instance variables, removing instance variables and renaming instance variables. How the values for the new variables are calculated should be provided by the developer through a migration policy.

Remove Class. When a class should be removed, the system should validate that there are no instances of the removed class after the migration of all the instances.

3.2.3 Value Change Operations

During an update process, it is normal to have modifications in the values contained in instance variables of live objects. We have called this change *Value Changes*. To address this problem the DSU provides a way of registering a migration policy to use on the instances of an unmodified class.

3.2.4 Transaction Handling Operations

The proposed operations include a set of operation to delimit the duration of a transaction. The DSU process includes the operations *beginTransacction()*, *commit()* and *rollback()*.

All the operations invoked by a patch after a *beginTransaction()* are only effective when the *commit()* operation is invoked. If *rollback()* is invoked all the pending operations since *beginTransaction()* are discarded, without affecting the updated application.

3.3 Meta-Object Protocol needed

To be able to implement the DSU the underlying platform should provide a set of basic meta-level operations. Figure 3 presents the required MOP. These operations can be described as:

Class Manipulation. The DSU needs to read the definition of a class and also it needs to define a new class.

Object State Manipulation. The DSU needs to read and write instance variables.

Instance Access. The DSU needs an operation to retrieve all the instances of a given class.

Namespace Manipulation. The DSU needs to be able of creating a copy of the namespace and replace the current namespace with a new namespace.

Object Swap. The DSU needs the ability to swap a group of objects by another group. Replacing all the references to the objects in the first group with references in the second group. It needs these operations to be performed atomically, the operations should change all the objects at once.

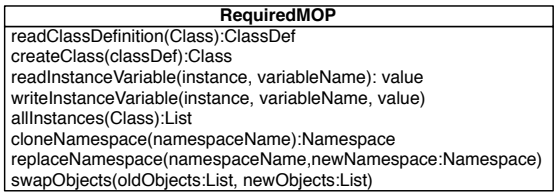

Figure 3. Required MOP

3.4 Proposed Abstractions

We see that some of the code implemented by the programmer for the update is repeated from one update to another. To reuse these elements, we propose some abstractions. Materializing these concepts as objects not only allows the reuse of code but also makes the creation of patches an easier task.

3.4.1 Migration Policies

During the migration of instances, there is the task of migrating the values of the instance variables from the old instance to the new instance. As said before, this depends of the application and the update. This problem is also happening when there is value change.

However, the Migration Policies are very dependent of the application, the updated class and the version itself. We have detected that most of them are repeated and can be extracted. So the developer can reuse migration policies in the migration of different instances and versions of the application. We give to the migration policies the format in Figure 4

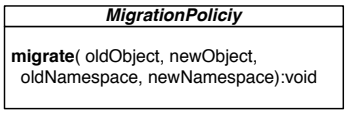

Figure 4. Structure of the migration policies

The migration policies should not modify the old objects or the old namespace, all the changes has to be performed in the new objects and namespace.

In our solution, there already implemented general Migration Policies and the developer can implement their own migration policies, enlarging the library of available migration policies.

3.4.2 Validations

Figure 5 shows another abstraction that we have identified in our solution. The idea is that the developer can express validations to guarantee that the invariants of the application are still valid.

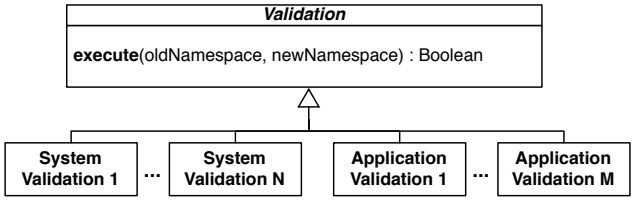

Figure 5. Hierarchy of the Validations, mixing system and application dependent validations

The DSU runs the validations after the new objects are created, to guarantee that the system is consistent. If all the validations succeed the changes are committed. In this way, the invariants of the application are guaranteed by the DSU.

Some validations are provided with the implementation of the DSU, as there are validations that guarantee the consistence of the platform. But most of the validations are application dependent, so the developer should implement them.

Having the validations as first order citizens of the DSU allows the reuse of them and ease the creation of patches.

4. Implementation

We implemented a prototype in the Pharo programming language. This prototype is available in a Git repository[1]. It can be loaded and used in the latest version of the platform, with the warning that it is in an early state of stabilization. It is intended to be included in future versions of the platform.

The required operations in the MOP are already present in Pharo. We only need to expose them through the interface the DSU is expecting.

The resulting DSU implementation can be used not only for updating a running application, but also in the daily developing. We validated it by updating a running REST web service. The web service provides a login operation, we have updated the representation of the user, while the server was running.

Also we validated by creating atomic refactors in the environment. Even updating objects used by the DSU process.

A DSU is clearly important in Pharo as it is an image based where most of the runtime is implemented in itself; modifying the base classes is not trivial because a partial applied change can lead to an unusable environment.

We are working in integrate the DSU with the existing tools for Pharo *e.g.,* allowing the creation of complex atomic automatic refactors.

[1] https://github.com/tesonep/pharo-AtomicClassInstaller

5. Discussion

Guaranteeing the correctness of an update is one important goal in any DSU. To achieve this, our solution is using two mechanisms: the first one the ability to copy the entire namespace and the second the ability to swap atomically objects.

Minimizing the copied and swapped objects is a good approach to improve the solution. In our validation implementation, we follow this schema. Only copying the modified objects. However, determining the exact objects to copy is dependent of the language and the environment. And it is possible that the detection process is even more resource consuming than the copy itself. One interesting alternative to explore is the use of a write barrier to detect the changed objects.

The proposed solution handles all the changes in the instances, including non structural changes. The first ones are supported by the detection of class changes and the later by the use of migration policies.

The use of different strategies in the moment of migrating instances is one of the top points of the solution. It gives an equilibrium between the possibility of highly configurable updates and the reusable characteristics of the migration policies. The idea is to provide a set of reusable migration policies and a flexible API to express new ones.

The coherence of the DSU process is achieved by the use of an alternative namespace. The proposed solution creates a copy of the namespace, and the running application is using the older one, so the mix of the old and new instances is controlled in a single point. This point is the migration of the instances by the migration policies.

The causality guarantee is assured by the implementation of validations.

One of the high points of the proposed solutions is that it can be implemented in any dynamic language providing the required operations.

We have decided to use eager migration of instances. Because having lazy migration of instances requires the use of proxies. It can be an extension to the proposed solution, but it will require the presence of meta-level operations to support dynamic proxies generation.

This design decision also allows that the penalty during the normal execution of the application is zero. During the execution of the application there is no need to execute code of the DSU.

The proposed solution is intended to be used not only in the update of a running application, but in the development process. The key points where the application can be used in the day-by-day development in Pharo is in the ability of perform atomic complex refactors, and in the modification of the system classes in the namespace.

For example if you want to modify a system class (that is used in user level code and in system code), *i.e.,* a Collection implementation. Usually, you have to implement this change

in carefully thought steps taking the order in account. With the proposed DSU, the changes are performed in a copy of the namespace, allowing the system to run with the old version until the whole change is finished and the become is performed in a single operation. This allows the user to concentrate more in the changes than in the order of the operations.

6. Related Work

As said before there are different solutions [2] for having Dynamic Software Update in an Object Oriented environment. However, there are some differences with our proposed approach.

DCEVM [16] and JRebel [17] are intended to be used during the development of the application. DCEVM requires a modification in the Java VM to run. JRebel does not requires modification in the VM, but the amount of handled changes is limited. It does not allow hierarchy changes and does not include a mechanism to migrate instances. Both solutions are not intended to update a running application.

Jvolve [15] implements a Virtual machine level DSU, needing to have an special version of the Java Virtual Machine to run. The DSU related code is running all the time impacting the global performance of the application.

DuSTM [11] and DUSC [10] implement a Java DSU without the need of modification of the Java VM. They change the code by the use of bytecode rewriting on the loading process, and can use lazy proxys to handle migration of data. However, they generate an impact in size of the program and time of execution.

Rubah [12] is a Java DSU implementation, it does not require to modify the VM, but the updateable program is modified at loading time through the use of bytecode rewriting. Rubah uses lazy proxy for the transformation of the instances. This solution is running during the whole execution of the program affecting the performance of the system outside the update process.

Pymoult [8] implements an API for the update of a program. The developer should express the changes implementing an update object. This object will be executed and it is the responsible of performing all the changes. The update objects can extend already implemented solutions. The main difference is that the patch, represented by the update object, has all the instance migration and validation logic.

None of the previous solutions is capable of updating the DSU process or any of the kernel parts of the environment.

All the solutions exposes an API for expressing the changes and the migrations to perform, but none of them describes the MOP required by the DSU.

7. Conclusion

This paper describes the different characteristics a DSU solution should have in an Object Oriented Environment to preserve the integrity of software state. After this analysis, we propose a solution that can achieve these goals. We identify the meta-level operations needed for the implementation of our solution. By implementing these basic operations, any object oriented language can use our proposed solution. We have validated our solution with a prototype in Pharo.

For future work, we aim at studying the relationship between the different properties and the required MOP. For example answering what the operations needed for lazy migration of instances are.

Nowadays, the solution can represent complex updates, but it would be nice to have tools to help the developer representing these updates in a easier way. In my opinion, this kind of tools are useful if they can be used easily by the developers and are integrated in the development environments, so those should be goals to bring the technology to everybody.

Also in the future, we aim at integrating the DSU with the day-to-day developing in Pharo. Allowing the tools to perform changes in the environment in atomically.

Finally, adding a nice model of changes can provide the DSU solution more information to take decisions, performing better results in the proposal of the migrations strategies.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. ISBN 1-55860-639-4. URL http://www.iam.unibe.ch/~scg/OORP.

[2] C. Giuffrida and A. S. Tanenbaum. A taxonomy of live updates, 2010.

[3] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984. ISBN 0-201-11372-4.

[4] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6): 1049–1096, nov 2005. doi: 10.1145/1108970.1108971.

[5] G. Kiczales and L. Rodriguez. Efficient method dispatch in pcl. In *Proceedings of ACM conference on Lisp and Functional Programming*, pages 99–105, Nice, 1990.

[6] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[7] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985. ISBN 0-12-442440-6. URL ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf.

[8] S. Martinez, F. DAGNAT, and J. Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015 : 10th International Conference on Software Engineering Advances*, pages 80 – 85, Barcelone, Spain, Nov. 2015. URL https://hal.archives-ouvertes.fr/hal-01247603.

[9] E. Miedes and F. D. Munoz-Escoı. Dynamic software update. Technical report, Technical Report ITI-SIDI-2012/004, 2012.

[10] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658. IEEE, 2002.

[11] L. Pina and J. Cachopo. Dust'm-dynamic upgrades using software transactional memory. 2011.

[12] L. Pina and M. Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. In *HotSWUp*, 2013.

[13] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *Proceedings of OOPSLA*, 2014.

[14] G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, and M. Mattone. Virtualization support for dynamic core library update. In *Onward! 2015*, 2015. URL http://rmod.inria.fr/archives/papers/Poli15b-Onward-CoreLibrariesHotUpdate.pdf.

[15] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A vm-centric approach. *SIGPLAN Not.*, 44 (6):1–12, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542478. URL http://doi.acm.org/10.1145/1543135.1542478.

[16] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852764. URL http://doi.acm.org/10.1145/1852761.1852764.

[17] ZeroTurnAround. What developers want: The end of application redeployes. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012.