# Preserving Instance State during Refactorings in Live Environments

Pablo Tesone[a,b,*], Guillermo Polito[c], Luc Fabresse[b], Noury Bouraqadi[b], Stéphane Ducasse[a]

[a] *Inria Lille-Nord Europe, 40 Avenue Halley, Villeneuve d'Ascq, France*
[b] *Unité de Recherche Informatique et Automatique, IMT Lille Douai, 764 Boulevard Lahure, Douai, France*
[c] *Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France*

**Abstract**

An important activity of software evolution consists in applying refactorings to enhance the quality of the code without changing its behaviour. Having a proper refactoring tool is a must-to in any professional development environment. In addition, live programming allows faster development than the usual edit-compile-debug process. During live programming sessions, the developer can directly manipulate instances and modify the state of the running program. However, when a complex refactoring is performed, instances may be *corrupted* (i.e., their state is lost). For example, when pushing an instance variable to a superclass there is a moment where the superclass does not have yet acquired the new instance variable and the subclass does not have it any more. It means that the value assigned to this instance variable in existing instances is lost after the refactoring. This problem is not anecdotal since 36% of the refactorings described in Fowler's catalogue corrupt instances when used in a live programming context. There is a need to manually migrate, regenerate or reload instances from persistent sources. This manual fix lowers the usefulness of live programming.

In this context of live programming, we propose, *AtomicRefactoring*, a new solution based on Dynamic Software Update to preserve the state of the application after performing refactorings. We provide a working extension to the existing refactoring tool developed for the language Pharo (a new offspring inheriting from Smalltalk), allowing application developers to perform complex refactorings preserving the live state of the running program.

*Corresponding author
*Email addresses:* pablo-adrian.tesone@imt-lille-doaui.fr (Pablo Tesone), guillermo.polito@inria.fr (Guillermo Polito), luc.fabresse@imt-lille-doaui.fr (Luc Fabresse), noury.bouraqadi@imt-lille-doaui.fr (Noury Bouraqadi), stephane.ducasse@inria.fr (Stéphane Ducasse)

**Keywords**

automatic refactorings, live programming environments, dynamic software update, IDE

## 1. Introduction

Software needs to evolve to keep up with the requirements of a real-world application. During software lifetime most of the effort is spent during the maintenance phase which consists in adapting the existing software to new requirements [? ? ]. During this evolution, when new functionalities are added or existing functionalities are modified, the overall complexity of the program is increased and thus lowering the code quality [? ? ? ].

*Automatic Refactorings.* Refactorings are behaviour preserving operations that help developers to improve the design of the application [? ? ]. Refactorings modify the implementation of the application keeping its features. They improve the overal quality of the application. Nowadays, refactoring tools are present in the majority of Integrated Development Environments (IDE) used in the industry [? ], but with different degrees of refactoring supports. A refactoring is composed of pre and post-conditions as well as a number of ordered elementary steps. Each step modifies the classes and methods. The refactorings are designed to improve the quality of the code while keeping behavioural consistency [? ]. Automatic refactorings constitutes a daily tool used by programmers to improve the quality of its code [? ? ? ? ? ].

*Live Programming Environments.* Live programming environments [? ], such as Lisp [? ], and Smalltalk [? ] or Javascript [? ], allows developers to modify the code while the program is running. Live programming allows a faster development cycle if we compare it with the *edit-compile-debug* process. Live programming provides a continuous flow of interaction between the developer and the program [? ? ]. This continuous flow of interaction provides an excellent framework for the development of behaviour driven applications [? ? ? ? ]. Also, live programming environments allow the manipulation of running program's state, through the manipulation of live instances [? ]. Live instances represent the state of the application in an object-oriented programming language. Existing live programming tools allow hot update of running code, modifying the structure of live instances as classes changes. During code modification, the program is still running. The user of the running application is the programmer itself or other users (*e.g.,*. a web application is still serving content during the live programming session). The programmer is able to stop, debug, modify and re-execute the running program. The use of live programming is not limited to Smalltalk or Lisp environments. Nowadays, there are different efforts to integrate live programming features in professional programming environments in languages as Java [? ? ? ], Python [? ] and Javascript [? ? ? ? ]. This new attempt to integrate live programming in professional IDEs is a reflection of the benefits of live programming [? ? ? ? ].

*Instance Corruption.* However, depending on the performed change, the internal state of live instances may be corrupted, making them unusable for the running program. For example, adding an instance variable initializes the new instance variable to *null* for all live instances. Such bad initialization may break program execution. As explained later, there are other refactorings leading to such *instance corruption*. We use *instance corruption* to designate live instances that have an incoherent internal state making the system unusable.

*Automatic Refactorings as a source of corruption.* As a refactoring tool performs a number of modifications in a single operation, each of these steps can corrupt instances. The use of a refactoring tool in a live programming environment amplifies the *instance corruption* problem. As refactorings does not alter the invariants of the code [**?** ], when a refactoring only applies behavioural changes the instance state is preserved because the state structure is not modified. If the refactoring changes the state structure, the *instance corruption* issue should be addressed. We show that 36% of the refactorings described by Fowler *et al.* [**?** ] present this problem when they are applied in a live programming environment (cf. Section 2).

*How to avoid instance corruption.* Dynamic software update (DSU) [**?** ] provides the means to modify a running program while preserving its state. Traditionally Smalltalk IDEs are automatically updating instances to the new structure of a class when this one is modified during a live programming session, but the state of instances may not be correctly initialized and get corrupted. In DSU, the update process should not only perform a hot update of the running code but also it should handle all data migration needed to smoothly move from one version of the code to another. Some approaches support data migration although the user has to explicitly express the changes needed to migrate from one version to another [**? ?** ].

*Solution in a Nutshell.* We propose an implementation of automatic refactorings that takes advantages of using a DSU tool. Each refactoring runs in a transaction and it affects all instances and classes in an atomic fashion. For each refactoring requiring migration of instances, a migration strategy is provided. By doing so, the instance corruption is removed when applying automatic refactorings.

*Contribution.* The contributions of this article are: (1) an analysis of the impact of refactoring tools in a live programming environment. And (2) a new technique using atomic DSU for applying refactorings in live programming environments. This technique allows developers to perform refactorings while preserving the state of live objects and thus the correct behaviour of the running program.

*Paper Structure.* In Section 2, we analyse the consequences of refactorings in a live programming environment, and we explain the need of having an instance migration strategy for these refactorings. In Section 3, we present our solution based on atomicity to apply refactorings while preserving the coherence of instances. We then present in Section 4 how this solution successfully solves

the *instance corruption* problem. Section 5 describes the implementation of this solution in Pharo [**?** ] as an extension of the Refactoring Browser. Section 6 presents the validation of our solution, and Section 7 we compares our solution with alternative solutions. Finally, in Section 8 we present our final conclusions and possible future work.

## 2. Class Refactorings that break Instances

Although it is possible to perform refactorings by hand, tool support is crucial to increase productivity [**?** **?** ]. Refactoring tools guarantee software behaviour consistency while preserving its correctness [**?** ]. However, this guarantee is not extended to live instances that constitute the runtime environment. Nevertheless, live instance correctness is crucial when doing live-programming, as the program is executing while the modification is performed.

A refactoring operation involves a number of small modifications of the code and the structure of the objects. These operations are usually performed sequentially, modifying the classes one change after the other, without handling the refactoring as a complex atomic change. Since the scope of default refactoring tools is static (i.e., they manipulate models of the code not of the instances), they focus on preserving a correct behaviour. However, a problem arises when refactorings are applied in a live programming environment. Indeed, live objects whose classes were modified should be migrated from the previous structure to the new structure. This need of migrating instances is not addressed by existing refactoring tools as they are not intended to be used in an environment with live instances.

As an extreme example, in bootstrapped and reflective systems [**?** **?** ], applying a refactoring on system classes may result in an instability of the whole system if instances are not correctly handled by the refactoring tool. This is why developers end up with a carefully planned sequence of steps to preserve the internal state of "kernel" objects [**?** ]. This operation is very common in fully reflective languages such as Pharo, Self [**?** ], Newspeak [**?** ] or Strongtalk[**?** ]. These environments allow the developer to change all the elements without differencing application, core libraries or kernel classes.

### *2.1. Challenges in refactorings: Two examples of corrupting refactoring*

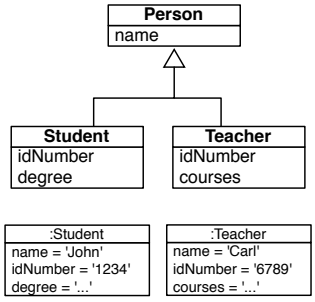
This section details two examples of refactorings that corrupt instances.

#### *2.1.1. Pull Up Instance Variable*

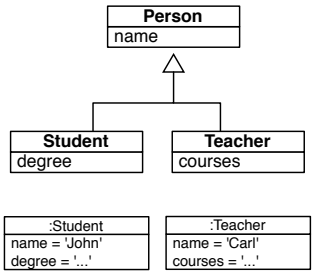
This refactoring removes the selected instance variable from all the subclasses and defines it in the selected class. Figure 1 shows the process of applying this refactoring to the *idNumber* instance variable. This instance variable is present in the *Student* and *Teacher* classes. Figure 1a shows the original state and Figure 1d shows the desired result of the refactoring.

To perform this, the refactoring does the following operations:

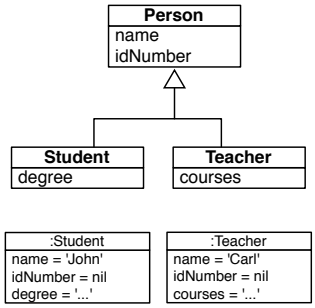
1. Iterate all the subclasses of the selected class. If the subclass has the instance variable, the instance variable is removed. Figure 1b shows the removal of the *idNumber* instance variable from *Student* and *Teacher* classes.
2. Add the instance variable to the selected class. Figure 1c shows the addition of the instance variable *idNumber* to the *Person* class.



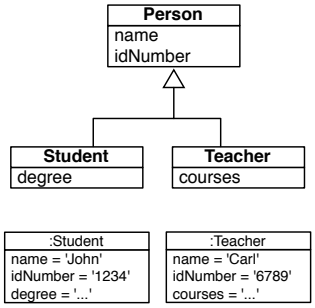

(a) Original State before the refactoring




(b) Step 1: Remove subclass instance variables




(c) Step 2: Add superclass instance variable




(d) Expected Result

Figure 1: Step by Step of applying the Pull Up Instance Variable refactoring to the *idNumber* instance variable present in *Student* and *Teacher* classes.

During each of the two operations, live instances are migrated due to the change in their structure. This migration is performed by the live programming environment each time a class is modified. During this process, the value of the pulled up instance variable is lost for live instances of subclasses (*e.g.,* Student, Teacher). These values are lost because the instance variables from subclasses are removed during the first migration step. Note that the order of these steps cannot be changed because instance variables of subclasses should be removed before adding the new instance variable to the superclass to avoid duplicated instance variables.

When we compare the result of applying the *Step 2* (Figure 1c) and the expected result (Figure 1d), we can see that the refactoring is not preserving live instance state. A state preserving refactoring must correctly keep the state

of *idNumber* in existing instances.

*2.1.2. Split Class Refactoring*

This refactoring extracts a selected subset of instance variables into a new object. It replaces all accesses to the selected instance variables by message sends to the new object. It also present the same problem described before.

This refactoring is more complex than the previous one. Indeed, to perform this refactoring the following changes are performed:

1. Create a new child class with the selected instance variables.
2. Add the accessor methods to the child class.
3. Add a new instance variable in the selected class to hold the extracted object.
4. Change all the uses of the mother instance variables with messages to the child object.
5. Remove the selected instance variables from the selected class.
6. Add initialization code creating the child objects when the mother objects are created.

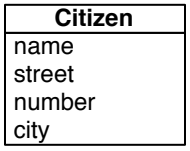
Figure 2a depicts the class structure of an example and Figure 2c shows some live instances in the environment before applying the refactoring. As a contrast, Figure 2b shows the expected result of applying the refactoring with the desired state of the live instances in Figure 2e.

Even though, the class structure and the methods are correctly created, live instance state is not preserved. Figure 2d shows the actual result of applying this refactoring. Since there is no special handling for migrating the extracted instance variables, this state is lost.

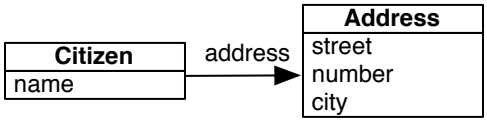
Although the refactoring operation is able to perform all the structural and behavioural changes needed, the instances are not migrated properly. The instances of the selected class are now useless because all the selected instance variables have been removed, replaced by an empty instance variable, and all the code has been modified to use this empty instance variable.

*2.2. Refactoring Impact Categories*

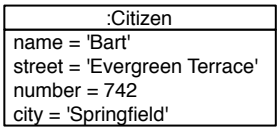
Instance corruption is not only present in the described examples. Instance corruption exists in a larger set of refactorings. Considering the 72 refactorings described in Fowler's book *Refactoring: Improving the Design of Existing Code* [**?** ] as a set of existing refactorings, we analyse the impact of applying the refactorings over live instances. This analysis shows that 36.11 % of these refactorings produce *instance corruption* when applied in presence of live instances. Preventing instance corruption is not just a matter of adding new pre/post-Conditions to refactorings. Indeed, refactorings have pre and/or post-conditions as part of their definition. These conditions help to guarantee consistency. Nevertheless, in the literature, these conditions only focus on structure and behaviour consistency without taking care of instances. Extending pre/post-conditions is not enough



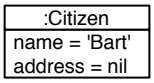

(a) Classes before Refactoring



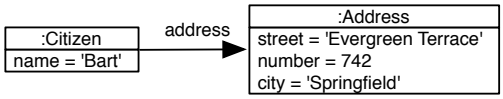

(b) Classes after Refactoring




(c) Objects before Refactoring




(d) Actual Result




(e) Expected Result

Figure 2: The Split Class refactoring corrupts its instances.

because instances must be correctly migrated according to the applied refactoring and the context.

We classified refactorings into 4 different categories related to *instance corruption*. For each category, we assess the amount of work to be able to preserve instance state.

*No Corruption.* The live instances are not affected at all because the refactoring does not modify the structure or the use of the state. All the changes are in the methods of the object. An example of this category is *Add Parameter refactoring*. This refactoring only adds a new parameter to an existing method. The method is modified in the class, but the structure of the live instances is not modified and no migration is required.

*Internal Corruption.* The structure of live instances is modified, but the state preservation can be computed using exclusively the modified instances. Client objects that have references to the modified instances don't need to be updated. An example of this is the *Instance variable rename* refactoring, where the value of the renamed instance variable should be preserved in a new instance variable with a new name. Another example is the *Extract class* refactoring. Here, the value of one or more instance variables is replaced with an object but the information to create this new object is taken from the original instance.

*Class Corruption.* When a refactoring changes the class of a set of instances, some or all selected instances should be migrated because their structure may also have changed which requires a data transformation. For example, the *Introduce Local Extension* refactoring may take some of the existent live instances and migrate them to a new class that is a subclass of the old class.

*Complex Corruption.* The changes performed by the refactoring either impact or require access to more instances than the ones from the modified classes to keep consistency. One example of refactoring corrupting more instances than the ones from the modified classes is *Change Value to Reference*. This refactoring impacts not only the instances becoming references but also all the users of those instances. Think about two clients referencing two equal value objects that when converted to references should be the same instance. This refactoring impacts both the client and the transformed value objects. On the other hand, an example of a refactoring that requires access to many instances is *Change Unidirectional Association to Bidirectional*. The refactoring creates a bidirectional association from a unidirectional. For example consider a *Course* with a collection of *Student*. This refactoring requires access to the *Course* instances to insert the back pointer from the *Student* instances.

In Appendix A we present the detailed classification and the justification of each of the problematic refactorings.

Applying this classification we discovered that 26 out of 72 (36.11%) refactorings corrupt instances and should take care of the migration of live instances to conserve their integrity. This means that 36.11% of these refactorings cannot be applied in a live programming environment without *corrupting* instances in the running program.

### *2.3. Ubiquity of the problem*

This analysis is not exclusive to the Fowler's catalogue and we also extended it to existing industrial tools. We wanted to check if potential problems are present in tools used daily by every programmer. To show this we extend the analysis to refactorings tools in *Rewrite Engine for Pharo (Smalltalk)*, *Eclipse JDT (Java)*, *Groovy/Grails Tool Suite (Groovy)*, *IntelliJ IDEA (Java)*, *Visual Studio 2015 with ReSharper (C#)* and *WebStorm (Javascript)*.

Table 1 presents the results of the tool analysis and how they compare with the results of Fowler's list of refactorings. These results show that if the refactoring tools are used in live programming environments, the consistency of the live instances is not preserved.

The analysed industrial tools are used in live programming environments. When doing so, they present the described problems. For example, live programming is performed in Java or Groovy using tools like DCEVM [**?** ], JRebel [**?** ] or Jvolve [**?** ].

Also, studies of the usage of automatic refactorings show that the problematic refactorings are used daily [**? ? ?** ]. Other studies show that the use of automatic refactorings is limited as it is not giving enough guarantee to developers [**?** ].

Although there are no specific studies applied to live programming, it is possible to extrapolate these results to live programming.

| | **No Corruption** | **Internal Corruption** | **Complex Corruption** | **Class Corruption** | **Total Corruption** |
|---|---|---|---|---|---|
| Fowler | 46 (63.89%) | 9 (12.50%) | 11 (15.28%) | 6 (8.33%) | 26 (36.11%) |
| Eclipse JDT | 24 (72.73%) | 5 (15.15%) | 3 (9.09%) | 1 (3.03%) | 9 (27.27%) |
| Resharper | 32 (69.57%) | 9 (19.57%) | 4 (8.70%) | 1 (2.17%) | 14 (30.43%) |
| IntelliJ IDEA | 28 (66.67%) | 5 (11.90%) | 8 (19.05%) | 1 (2.38%) | 14 (33.33%) |
| Pharo | 33 (66.00%) | 13 (26.00%) | 2 (4.00%) | 2 (4.00%) | 17 (34.00%) |
| WebStorm | 10 (90.91%) | 1 (9.09%) | 0 (0.00%) | 0 (0.00%) | 1 (9.09%) |
| Groovy | 7 (63.64%) | 3 (27.27%) | 0 (0.00%) | 1 (9.09%) | 4 (36.36%) |
| **Average** | **25.71 (67.92%)** | **6.43 (16.98%)** | **4.00 (10.57%)** | **1.71 (4.53%)** | **12.1 (32.08%)** |

Table 1: Results of the analysis of existing refactoring engines.

## 3. Our Solution: Atomic Refactorings for Live Programming

Traditionally, refactorings are applied in a *non-atomic way* and changes generated by the refactoring tool are applied one at a time, modifying live instances after each change. For example, the rename refactoring involves two operations: adding a new instance variable with the new name and removing the instance variable with the old name. These operations are performed one after the other and after each one, live instances are migrated and corrupted.

In a live programming environment, atomic refactorings are needed to prevent *instance corruption*. Since instances are accessed and modified concurrently in a multi-threading application, all the changes should be applied at once guaranteeing isolation and mutual-exclusion.

In Section 3.1, we present our solution to perform refactorings that preserve the state of live instances. This solution is based on using an Atomic Dynamic

Software Update (ADSU) mechanism [**?** ]. It manages the application of changes to methods and class structure as well as the migration of existing instances from an old class structure to a new one. The used ADSU mechanism performs all the changes atomically, preserving the state of the application and its behaviour while the application is running (cf. Section 3.2). The ADSU guarantee the atomicity of the modifications, providing isolation to the running threads, synchronizing the access and finding a safe update point.

Regarding refactorings, it means that all classes are modified at once and that live instances are directly migrated from the current version to the final one in an atomic way.

### *3.1. Description of the Atomic Refactoring Application*

Our proposed refactoring solution requires that the used ADSU provides ways of expressing the changes to be applied. These changes include modifications to class structure and method body as well as the process needed to migrate instances. The set of changes required and the migration logic needed is called *a patch* [**?** ].

We require that the selected ADSU includes a reification of the patch. A patch includes the details of all the modification to perform. It includes the changes in method and classes and the logic to migrate the instances. The proposed solution represents the modifications with instances of a hierarchy of classes as shown in Figure 3. These operations are later composed to produce the patch that is the entry point to the ADSU.

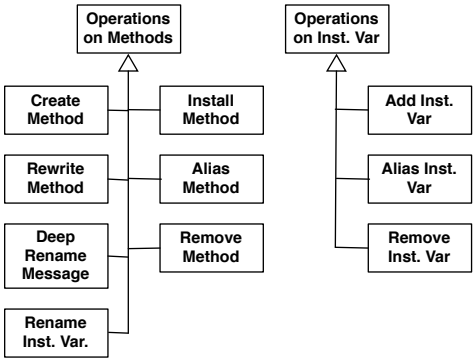



Figure 3: Patch Operations.

We call *migration policy* [**?** ], the behaviour needed to migrate an object from one version to another. Migration policies are special cases of *transform functions* [**? ?** ]. They apply only to the transformation of live instances. In the DSU literature, transform functions are more generic and transform not only live instances.

All the changes are scheduled and performed atomically, the ADSU mechanism determines the exact moment to execute the update. The ADSU solution finds a safe update point. The lookup of the safe update point is described in

Section 5. Basically, the solution looks for a point where there are no threads executing (or to execute) modified methods. This is performed through the inspection and manipulation of the thread call stacks.

The atomic refactoring engine generates patches automatically. As the changes required by an automatic refactoring are well known, the patch is generated with this changes. Instead of applying the changes in the methods and classes they are accumulated in a patch.

In our implementation for validation we are using *gDSU* [**?** ]. This ADSU solution presents the required elements to implement atomic refactorings. However, our proposed refactoring solution is adaptable to use any other ADSU that provides such requirements.

Figure 4 describes the overall process of applying a refactoring using a generic ADSU that provides the the requirements of our solution, the following section expands the details:

- In *Step 1*, the user selects and configures the refactoring, this is performed through the same user interface the user uses in non-atomic refactorings. When applying a refactoring, our solution does not need more information than the one provided in the non-atomic implementation of the refactorings. Indeed, all required information is already specified in the refactoring to apply or it is retrieved from the live environment. After having all the user input, the process is fully automatic.

- In *Step 2*, the atomic refactoring engine generates the *patch*. As described before, this patch includes all the modifications needed and the migration policies to migrate live instances. The creation of the patch is specified in the refactoring definition. Most of the time, existing refactoring engines compute the changes to be applied to classes and methods. With Atomic Refactoring, refactorings should also compute the migration policies that we will be applied to the instances.

- The *patch* is the entry parameter of the Atomic Dynamic Software Updater (ADSU). In *Step 3*, the ADSU is invoked to apply the patch.

- *In Step 4*, ADSU applies all the changes in an atomic process. Preserving the behavioural consistency [**?** ] and also the consistency of live instances. The ADSU can be executed while the system is running. The ADSU selects the best moment to execute the update process, checking that none of the running threads is using the affected instances. As the changes are applied atomically, in case of an error or problem during the execution the process is safely aborted.

After an update of the running environment by the ADSU, the atomic refactoring is completed and the user can continue using it.

### *3.2. Atomic Dynamic Software Update Mechanism*

The Atomic DSU mechanism provides a way of applying a patch in an atomic way. The goal of the ADSU is updating the state of an application, including

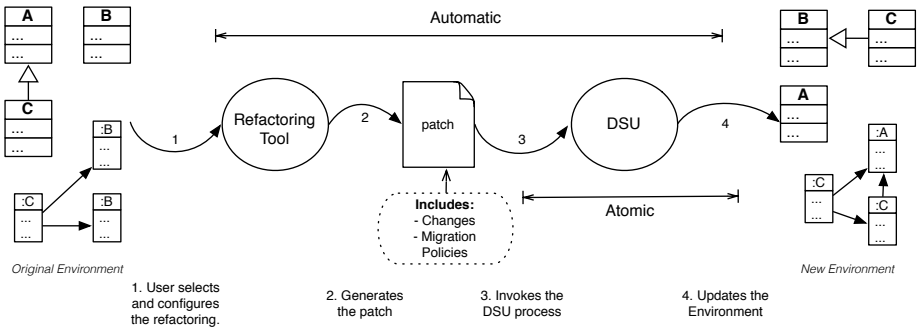



Figure 4: The Atomic Refactoring process.

classes, methods as well as live instances. However, this state should be correctly migrated to maintain consistency. The modified elements should take the new form, the unmodified elements should keep their state, the global state of the application should be preserved and the running threads should continue their execution normally.

To achieve this goal, all the modifications are performed in a *new environment*, without modifying the *original environment*, once all the changes are validated the original environment is replaced by the new environment. During the execution of the update all the other threads are suspended. After the update is completed, the threads are resumed.

In Object Oriented Programming, an environment is the set of all live instances and classes. In Smalltalk, all the elements in the system (instances, classes, methods, global variables) are represented as objects and are accessible inside the environment. An Object oriented environment is the space where all the instances are contained, and where they interact with each other sending messages [**?** **?** **?** ].

All the modifications, migration policies and validations are included in the patch. These elements are generated by the automatic refactoring engine. Each refactoring includes its required changes and a set of validations to check its effect.

Our solution can be implemented using any DSU mechanism that provides atomic execution, application of multiple modifications in a patch, creation of migration policies and a validation step before the actual commit of the atomic execution.

Our proposed refactoring solution in combination with the selected ADSU mechanism (*gDSU*) performs the following steps to apply a patch in an atomic way [**?** ]:

1. *gDSU* selects the moment in time to execute the update process. *gDSU* can only apply an update when there is no thread executing code or using objects to be modified. So, *gDSU* process waits for a safe point i.e. when the stacks of all the threads do not contain a method to be updated. This is performed through thread call stack reflection. When a safe point is

detected all threads are suspended. If some threads are using instances through other methods, *gDSU* will update the objects and references in the execution stack trace. All the instance references are updated, not only the ones in the heap but also the ones in thread stacks.

2. All classes to be modified are created in a *new environment*, and the changes are applied in the *new environment* without tampering the *original environment*. The classes in the *new environment* are created as they are needed by the changes in the patch. The new classes are not declared in the original environment, they are declared in the *new environment*. After this step, the new environment contains a copy of all the modified classes with the changes applied. The copying of classes produces an impact in time and memory, although as the number of modified classes and methods in a automatic refactoring is limited this impact is also limited. By definition each automatic refactoring has a limited impact in classes and methods.
3. When all the classes are modified in the new environment, the affected instances should be migrated. *gDSU* applies all the migration policies needed to migrate the objects. Each of the affected instances is copied to the new environment. During the copy, the migration policies are applied. So, the resulting instances have the migrated state from the ones in the original environment.
4. When all live instances are migrated, the results can be validated. The validation logic is included in the patch. The validations are run on the new environment. This validation logic is generated by the refactoring engine. Each refactoring has, by definition, a set of validations to execute. If the validation is unsuccessful the new environment is discarded without affecting the original environment. If the validations are successful the update process continues. However, if one of the validations fail, the process is aborted. When the process is aborted, the whole new environment is discarded without affecting the original environment. After the abortion, all the threads are resumed.
5. If the validation is successful the original environment is replaced with the new environment through the use of a bulk swap operation. This operation replaces all the instances in the original environment with the corresponding instances in the new environment. Performing this replacement will move to the original environment all the modified classes and migrated instances. It will move also all the new instances that are referenced by classes, global variables, or migrated instances.
6. Finally, *gDSU* process performs a clean-up. It frees all the structures and data used during the process and resumes all the threads, so the application can continue its normal execution.

Section 4 provides examples of this process through the application of two refactorings.

## 4. Preserving Instance State when Applying Refactorings with our ADSU

Our refactoring model and its tool are successfully used to correctly handle instances when applying refactorings presented in Section 2. In essence, our refactoring engine generates the corresponding migration policy for existing instances according to the currently applied refactoring.

We have applied this technique to implement the 17 instance corrupting refactorings existing in Pharo[1].

### *4.1. Pull Up Instance Variable*

In the case of *Pull Up Instance Variable* the migration policy should correctly copy the values from the instances before the application of the refactoring to the refactored instances. There is no need of transforming the values, just not losing them and correctly re-assign them into the refactored instances. To achieve this, the atomic refactoring engine generates a migration policy for the refactored class and its subclasses. This policy copies the state of all affected instances into newly created refactored instances. Then *gDSU* creates the refactored instances in the new environment. The refactoring engine correctly initializes new instances with the saved state even if refactored objects have a different layout from original ones (e.g. instance variable order has changed) by using the instance variable names. Relying on names is the default migration policy of *gDSU* and it is automatically generated. Here the automatically generated code for the migration policy of a pull up refactoring:

```
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
    new class instanceVariables do: [ :newIV |
      old class instanceVariableNamed: newIV name
        ifFound: [ :oldIV | newIV write: (oldIV read: old) to: new ] ].
```

This migration policies iterates all the instance variables defined in the new instance. In Smalltalk, the instance variables are reificated as objects. They can be accessed by the class of an object. The code iterates the instance variables of the new instance, and looks for the one with the new name in the old instance. If the instance variable is found in both instances, the object representing the instance variable is used to read it from the old instance and to write it in the new instance.

### *4.2. Split Class Refactoring*

For solving the live instance migration of Split Class, the atomic refactoring engine generates a more complex migration policy. Although this migration

[1] Atomic refactoring implementation is available at https://github.com/tesonep/pharo-atomic-refactors

policy is more complex because it has transformations of values, the migration policy is generated automatically.

We call mother objects the instances of the selected class for the refactoring and child objects the instances of the newly created class. We have two versions of the mother object, the old and the new one. This is shown in the Figure 2c and Figure 2e. The first figure shows the instance before the migration, with all the values of the instance even the ones to be migrated to the new class. And the second figure shows the two instances that are the desired result of the refactoring. This migration policy performs the following steps for each of the instances to migrate:

1. Create a new instance of the child class.
2. Store this new object into the instance variable of the new mother object.
3. Copy all the instance variables to extract to the new child object.

The creation of the new instances, the copy of all the instance variable values to the new created instance and building the relationship between the migrated mother object and the new child object are not performed by default by *gDSU*. It is the responsibility of the *migration policy* to perform the following tasks:

- It creates a new child instance.
- From the old mother instance, it copies all the instance variable values to the newly created object.
- It adds a reference to the child object in the new mother instance. Using for this the new instance variable in the mother object.

This migration policy is automatically generated by the atomic refactoring engine using the information it already has. A detailed explanation of the implementation of this refactoring is presented in Section 5.1.

## 5. Implementation

We validate our proposed solution by the implementation of an extension to the refactoring tool present in the *Pharo Programming Environment*. We selected Pharo because (1) it supports live programming with an advanced debugger allowing to define methods on the fly, dynamic recompilation of classes, and other features supporting life-programming such as instance migration, (2) the Refactoring engine available in Pharo is the direct evolution of the original Refactoring Browser [**?** ] implemented in Smalltalk, (3) the Pharo Refactoring engine propose one of the most complete refactoring implementation with 50 refactorings.

We implemented our Atomic Refactoring solution based on the use of *gDSU* [**?** ]. This ADSU solution has some characteristics that are required for the implementation of our proposed atomic refactorings. It supports the definition of patches. Each patch contains the set of changes, migration policies and

validations. Also, *gDSU* runs as a library without needing to modify the running Virtual Machine or expecting the target code to follow any guideline.

As required by our solution *gDSU* tool receives a patch describing the modifications. The patch contains not only all the changes to perform but also the migration policies to apply. This ADSU allows us to perform all the changes in an atomic way, which means all the modifications are applied at once. The implementation follows the description in Subsection 3.2.

Also, it allows the use of migration policies to describe how the objects should be migrated from the old version to the new version. The selected ADSU allows an easy reuse of the migration policies. This feature eases the development of the migration policies for the different refactorings. Moreover, *gDSU* provides a number of default migration policies that can be used. However, in our implementation, we need to implement our own migration policies to be able to express more complex instance migrations.

*gDSU* also allows us to validate the correct execution of the update. This task is performed by custom code implemented as validation objects. A validation object validates the result of the update process before committing the operation. The validations objects are added to the patch and they are generated by the refactoring tool. The validations are defined for each of the automatic refactorings. Each automatic refactoring includes the validations as it includes the required operations to do. For example, in the *Pull-Up* refactoring instances of the new classes are validated to have the proper instance variable value.

Using them we validate the correctness of the refactoring after all the changes are applied. Again the selected ADSU tool allows the reuse of different validations in different refactorings. All the validation objects receive the original and new environments and perform all the validations needed.

This feature is used when a refactoring operation needs to validate a post-execution condition. Including validations is not mandatory, but provides a way of validating the result of the refactoring execution, not only over the static model but also on live instances. For example, it can be used in the *Split Class Refactoring* to validate if the accessor methods in the original instances return the same values.

Our solution is handling each refactoring as an atomic operation. When a set of two or more refactorings modify the same classes, the required migration policy should be capable of performing the migration taking into account both refactorings. The required migration gets more and more complex. This required complexity is outside the scope of our solution. So, our proposed solution applies the set of refactorings as a sequence of atomic refactorings. By the refactorings nature of state and behaviour preserving operations this decision does not affect the final result of the operation.

As our solution generates the migration policies and validations automatically based in the definition of the refactorings. We implemented for each refactoring the code to generate them. This automation constraints our solution to apply each refactor atomically. We does not support to apply two or more refactorings at the same time. As they might modify the same class in many ways that cannot be analysed by the automatic generation of the migration policies.

We decided to implement the refactoring tool as an extension, so we can reuse all the user interface and the integration of existent tool with the IDE. As the required information to apply a refactoring is the same required by the previous implementation of the refactoring tool, it was not needed to implement modifications in the user interface. The implementation is available in Github[2] and can be easily downloaded and used in Pharo 6. *gDSU* [**?** ] is also available in Github[3], it is also intended to be used in Pharo 6.

*5.1. Application of the Refactoring step by step*

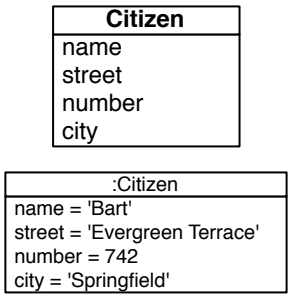



Figure 5: State before refactoring

To apply the changes in an atomic way a number of steps are executed. All these steps are performed during the atomic application of changes executed by *gDSU*. Some of the steps are executed by *gDSU* and other by our proposed solution. We will explain in detail the interaction of our solution with *gDSU*. For doing so, we will describe the steps following the example presented in Subsection 4.2.

Figure 5 shows the state before applying the changes. To perform the changes the atomic refactorings and *gDSU* apply the following steps.

1. *gDSU* creates a *new environment*. All the new and modified objects and class/methods will be stored in this new environment. The environment starts empty. (Figure 6)
2. *gDSU* executes the patch that have been created by our refactoring solution. This patch includes all the details to create and modify classes and methods affected by the refactoring. The modified versions of the classes *Citizen* and *Address* are created in the *new environment*. The refactoring operations modify the copied classes without tampering the *original environment*. For example, it is not needed to create the class at once, as the *Address* class can be created empty and then add the different instance variables (Figure 7).

[2]https://github.com/tesonep/pharo-atomic-refactors
[3]https://github.com/tesonep/pharo-AtomicClassInstaller

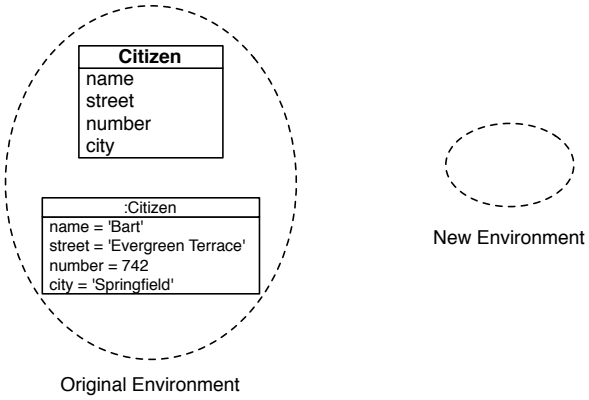



Figure 6: A new environment is created

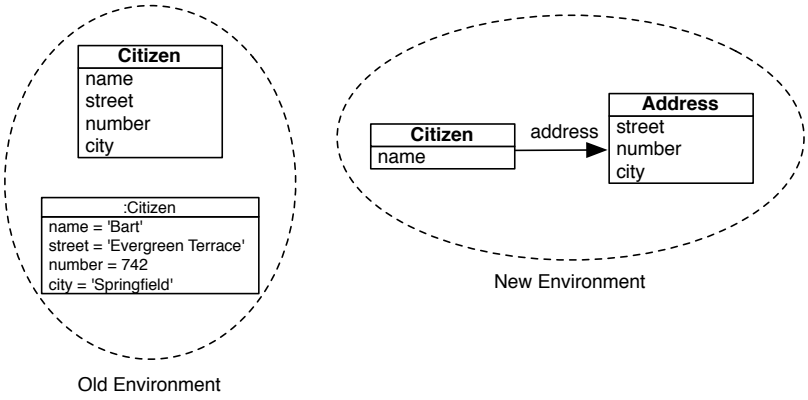



Figure 7: All the modifications to the classes are applied

3. Once the class modifications are completed, the *Citizen* instances should be migrated. The migration policy migrates the *Citizen* instances, creating the *Address* instance needed in each case (Figure 8). This migration is performed using the refactoring migration policy. This policy is included in the automatic refactoring engine, and it is used only for this refactoring. The following code is the migration policy. The migration policy is generated by our refactoring solution and it is applied atomically by *gDSU*.

```
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
    | child childClass targetClass oldClass oldValue |

"Step 1:uses the default migration for the mother object, that
    copy all the instance variables by name.
    This is implemented in the reusable part of the migration policies.
    It copies the instance variable values by name, copying the instance
```

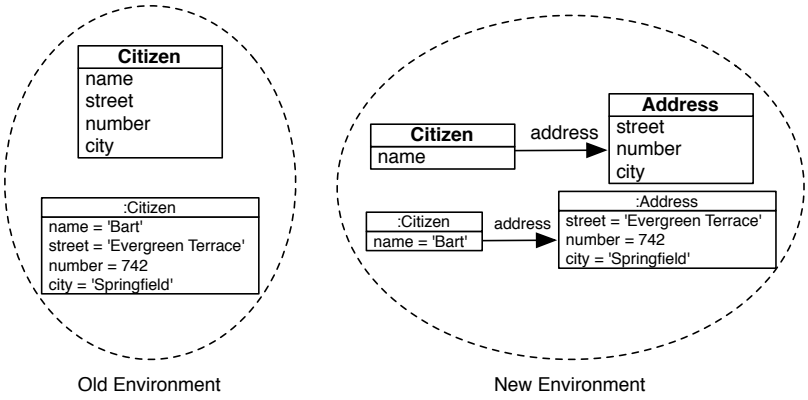



Figure 8: Live instances are migrated

variables existing in the old instances into the new instance."
*self* basicMigrateInstance: *new* from: *old*.

"Step 2: recovers the newly child class created from the environment.
The newClassName is a parameter of the refactoring."
*childClass* := *newEnv* at: *newClassName*.

"Step 3: creates a new instance of the child class.
Using the class recovered from the environment the new instance is created."
*child* := *childClass* new.
*targetClass* := *new* class.
*oldClass* := old class.

"Step 4: add the reference from the mother object to the child object."
(*targetClass* instanceVariableNamed: *referenceVariableName*)
write: *child* to: *new*.

"Step 5: copy all the extracted instance variables.
Fill up the child instances with the variables extracted from the mother instance.
These are the instance variables that are extracted from the mother class.
These variables are a parameter to the automatic refactoring."

*variablesNamesToExtract*
do: [ :*e* |
*oldValue* := ((*oldClass* instanceVariableNamed: *e*).

(*childClass* instanceVariableNamed: *e*)
write: *oldValue* read: *old*) to: *child* ].

4. When all live instances are migrated, *gDSU* validates the correctness of

the refactoring, by the execution of a validation object generated by the refactoring tool. In this case, it checks that all the *Citizen* instances have a corresponding *Address*. As the validation is successful, the modified objects in the old environment are replaced with the ones in the new environment. The bulk swap operation replaces all the instances updating the references to them. The logic to validate if the refactoring has been correctly applied is provided by the refactoring solution based on each of the refactorings. (Figure 9)

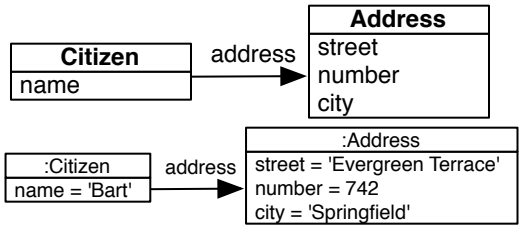



Figure 9: The *New environment* replaces the old environment

### *5.2. Implementation details of* gDSU*: Dynamic Software Update from Development to Production*

This section provides a glance of some implementation details of *gDSU*. These implementations details are key to understand the applicability of our proposed atomic refactoring solution and the requirements that should be present in any alternative Atomic DSU tool used by our solution.

Although the complete analysis of *gDSU* [**?** ] is published as an independent work and it is outside the scope of this work. For the sake of completeness of this paper we analyse some implementations details of *gDSU*.

*Update Window.* *gDSU* executes the refactoring in an atomic way. To achieve this objective, *gDSU* requires an update window. During the atomic update window *gDSU* is the only thread modifying the environment. Before starting to perform the update, *gDSU* tool suspends all running threads. The threads are suspended to guarantee that no new live instances of the old version are created during the application of the update. Running threads cannot be suspended in any moment, they should be in a safe quiescence point [**?** ].

*Safe Point Update.* An unsafe point is when there is a thread that has in its execution stack one or more of the methods to be modified by the update process. Performing an update during an unsafe point produces corruption in live instances or in the execution stack. The former is produced when the method code access or modifies live instances that have their structure modified. The later is produced when the thread tries to execute code that is no longer valid (*e.g.,* sending a message that does not exist or executing a method that has been changed).

*Safe Point Definition.* *gDSU* tool defines a safe quiescence point when none of the threads has methods to be modified in their execution stack. The threads using instances that should be migrated does not represent a problem to the ADSU. As all the instances are migrated using the *bulk swap* operation. This operation modifies the object stack of the running threads as the heap. This operation, that is natively implemented in the VM, traverses the heap and the stack replacing all references to a given instance with the references to the new instances. This operation is needed to already support live programming and replacing of live instances.

*Safe Point Detection.* *gDSU* detects the safe point to perform the update by monitoring the running threads. When the ADSU tool is invoked to perform the update process, it checks all the running threads to see if they are in a safe quiescence point. If the threads are in a safe point, the threads are suspended and the update process is performed. However, if one or more of the threads are in an unsafe point, *gDSU* will wait until all the threads are in a safe point. *gDSU* has to detect the moment when all the threads are at a safe point. To perform this, the ADSU tool modifies the execution stack of the threads that are not at a safe point. *gDSU* tool inserts a new method activation in the execution stack, just after the execution of the methods that should be modified. Each time a thread exits a method to be modified, the ADSU tool is activated to recheck all the threads. This process continues until a safe point is achieved, or after a number of retries, *gDSU* cancels the update because the threads are never getting to a safe point.

*Execution During the Update Window.* During the update window, the only thread running is the *gDSU* thread. So, the update window should be as short as possible. To minimize the time, *gDSU* only copies the classes that need to be modified. The set of classes and instances to be copied are calculated during the update process. It is calculated from the set of changes, as the changes includes the classes they affect. The old environment is only accessed in a single point. In this point, the classes and instances are copied to the new environment.

*Atomic Application of Changes.* *gDSU* applies all the changes in the new environment in a *bulk swap* operation, as said before this operation is implemented in the Smalltalk Virtual Machine. However, the existent implementation needs to update the whole memory [**?** ], extending the time required in the update window. So that, *gDSU* is using the Miranda [**?** ] et al. implementation that reduces the update window time.

### *5.3. Limitations*

Our proposed atomic refactoring solution does heavy usage of the underlying Atomic DSU tool. It generates all the information required to perform the refactoring and the the ADSU tool is the one that executes the update.

So, the selection of the ADSU to use limits the capabilities of our solution. Selecting *gDSU* allows us to perform updates in production and development

environments while this environments are executing. Moreover, it allows our solution to perform self updates and updates in the Pharo core libraries.

However, we are aware of the limitations of any existing ADSU solution. Even more, our proposed refactoring solution will never be able to execute refactorings that require updates not provided by the ADSU tool.

One of the advantages of selecting *gDSU* is that it uses a safe-first protocol to update the application. It is designed to discard any update that cannot pass the validations, allowing us to guarantee that the refactoring has been correctly applied. We consider this approach adds value to the solution, because we consider that a partial applied refactoring affects the stability of the application and we prefer to cancel the whole refactoring and notify the user.

One of the known limitations of selecting *gDSU* is that it still may not find suitable update-points or even may produce crashes due to an incorrect detection of safe-update points. This is a known overcome of the DSU solutions using activeness checking [**?** **?** ].

We consider, that the analysis and the solutions to the different problems and limitations of a given DSU solution is outside the scope of this paper. However, our proposed solution could be adapted to generate patches for other ADSU solution. Allowing us to use an alternative solution in case of finding that the limitations of *gDSU* make the atomic refactorings impractical.

## 6. Validation

To validate our proposed solution we have implemented atomic refactorings as an extension of the Pharo Refactoring Browser. Our validation implementation uses *gDSU* as its Atomic DSU tool. This implementation is intended to run in Pharo 6.1. We have stated in previous sections the motivations for choosing Pharo and *gDSU*.

This implementation presents an atomic variation for all the Pharo Refactoring Browser refactorings that introduce instance corruption. This set of refactoring includes 17 different refactorings.

As our proposed solution performs heavy use of an atomic DSU tool, we need to have a clear difference between the validations of our proposed solution and the validation of the underlying ADSU.

Our proposed solution generates all the information required by *gDSU* to correctly apply the changes. Once the patch is produced is the task of the ADSU to correctly execute it. The patch contains all the changes to execute, all the migration strategies needed and all the validations to perform in each refactoring.

The correct generation of the patch and the generation speed is under analysis. The correctness and speed of the atomic update process is outside the scope of this work.

### *6.1. Validation Methodology*

To validate our proposed solution we executed instance corrupting refactorings in three different scenarios. We used to set-up this scenario a test application

that mimics a REST Server application. This application implements a stateful chat between the connected users. The code of the application, instructions and scripts to set-up it are available in Github[4].

*Development Scenario.* We set-up a development environment with our application code. We run a simulation of requests to generate application objects. This simulation produces around 30 000 live instances. After the simulation, we have an environment with live instances that is useful to perform live programming refactorings. This environment replicates a common development environment where the developer has not only the code but also a set of data to try her changes.

*Production Scenario.* Using the same application, a HTTP server is launched. This HTTP server replicates a production server. This is designed to be deployed as a productive application, as it uses the production ready frameworks and technologies used in Pharo. We generated 10 concurrent requests to our server during 2 minutes, generating an average of 700 requests per second. This simulation generates the load expected in a production server. Then, we apply the refactoring at minute 1.

*Pharo.* Pharo is a fully reflective, self constrained language, tools and IDE. So, all the development tools and libraries are developed in Pharo, and these libraries are used to modify Pharo itself. When developing Pharo tools and libraries is frequent to get corrupted instances. As it is a live programming environment, corrupted instances produces the crash of the whole development environment. This self-modification nature constitutes a perfect test bench for validating our proposed solution.

All the validations and benchmarks have been executed using Pharo 6.1 32-bits, in a machine running OS X 10.12.6 having a 2,6 GHz Intel Core i7 and 8 Gb of 1600 MHz RAM memory.

### *6.2. Validation 1: Correctness of the implementation*

*Research Question.* Does our solution complies with the existing behaviour of Pharo Refactoring Browser?

*Scenario.* The Pharo implementation of the Refactoring Browser includes an extensive set of tests to validate their execution and the impact on the classes and methods each refactoring operation modify.

To validate our extension we executed all the tests included in the old Pharo implementation.

*Results.* All the tests execute correctly, the results of executing the old implementation or the new implementation is the same. We can guarantee that the atomic refactorings executes the same modifications in classes and methods that the reference implementation.

[4]https://github.com/tesonep/chatServer.git

### *6.3. Validation 2: Updating live instances*

*Research Question.* Is our solution capable of performing atomic refactorings in an application with a high number of instances?

*Scenario.* In this validation we have the objective to test the ability of our proposed solution of correctly applying refactorings while preserving the state of live instances.

This validation is performed in the Development Scenario.

For doing so, we executed the following refactorings on live instances of the application:

- Extract Class
- Pull-up instance variable
- Extract Subclass
- Introduce Local Extension
- Replace Array with Object

These refactorings where selected because presents different scenarios of migration policies, validations and changes.

We executed the refactorings in isolation, executing only one at the time. And all together, all the refactorings executed in sequence.

After each isolated execution and the combined execution we validate the live instances to guarantee that all the instances have preserved their state. By using a test application that we designed to present all the situations required to test the validity of the solution; we have performed the validation in two different ways. First, a manual analysis of some of the migrated instances, checking that the logic of the migrated application is still valid, the live instances correctly migrated and the test of some of the application features. And secondly, a set of tests to validate that all the instances have correctly state and the automatic validations performed during the application of the migration are covering all migration scenarios. The checks were performed on a representative set of objects.

This validation is complementary with *Validation 3*. This validation is taking a sample of instances to directly validate allowing us to correctly detect which of the refactorings is failing. Then in *Validation 3* we perform the analysis on all instances indirectly by executing the application.

*Results.* The migrated instances preserves correctly their state. All the validations executed on the objects are correct.

### *6.4. Validation 3: Updating live instances during execution*

*Research Question.* Is our solution capable of performing atomic refactorings in an application with a high number of instances while the application is executing and adding / modifying live instances?

*Scenario.* In this validation we have the objective to test the ability of our proposed solution of correctly applying refactorings while preserving the state of live instances while the application is under heavy usage.

This validation is performed in the Production Scenario.

For doing so, we executed the following refactorings on live instances of the application:

- Split Class
- Pull-up instance variable
- Extract Subclass
- Introduce Local Extension
- Replace Array with Object

These refactorings where selected because presents different scenarios of migration policies, validations and changes.

We executed the refactorings in isolation, executing only one at the time. And all together, all the refactorings executed in sequence.

The execution of the refactorings is performed while the application is receiving 10 concurrent request with an average of 700 request per second.

After each isolated execution and the combined execution we validate the live instances to guarantee that all the instances have preserved their state.

*Results.* The migrated instances preserves correctly their state. All the validations executed on the objects are correct.

We have validated the correct migration of the state by continuing with the normal execution of the application. As the load on the application is using the migrated instances, the correct execution of the second half of the test guarantee that all the migrated instances are correct.

This validation is complementary to *Validation 2*, as this validation performs an indirect check of all the migrated instances.

### *6.5. Validation 4: Self Modification*

*Research Question.* Is our solution capable of performing atomic refactorings in the executing Pharo libraries, including the atomic refactoring engine and the ADSU?

*Scenario.* For this validation we execute the following refactoring in Pharo library classes that are used in the system.

- Split Class
- Pull-up instance variable
- Extract Subclass

- Introduce Local Extension
- Replace Array with Object

We target the refactorings to the Collections framework, the UI framework and the atomic refactoring engine.

*Results.* The refactorings have been correctly applied without affecting the stability nor the state of the Pharo IDE. This validation checks indirectly the result of the migration. As the Pharo IDE and environment is continuously using the affected code and instances. This is guarantee as any error in the migration of instances or in the application of the refactoring results in an unusable system.

### *6.6. Validation 5: Benchmarks*

*Research Question.* Does the generation of the patch generates an impact in the duration of the update?

*Scenario.* This validation asserts that our solution does not introduce a noticeable impact in performance nor memory footprint in the generation of the patch.

For doing so we have compared the execution of a hand-crafted patch and the execution of the atomic refactoring tool.

The baseline time and memory consist of only the execution of the hand-crafted patch. This allows us to measure the memory and time impact of executing the atomic update.

The measures using our solution includes both the generation of the patch from the analysis of the state of the system and the execution of the update.

The execution have been performed on a running system using the Production Scenario, with 10 concurrent request and 700 request per second. This generates between 30.000 live instances to migrate.

The selected refactoring to test is Split Class.

*Results.* The execution of this validation does not present a noticeable impact in memory nor time in the execution of the baseline test and our solution test.

Executing both tests produce an average memory footprint of 10MB and an average execution time of 300 ms.

## 7. Related Works

*The Rewrite Engine* [**?** **?** ] provides a complete refactoring tool. Since 1996, it has been used in different Smalltalk implementations like Pharo, Dolphin Smalltalk, VisualWorks and VASmalltalk. However, this tool does not support migration of live instances. Our solution handles the correct migration of live instances.

Dynamic Software Update tool as DCEVM [**?** ], JRebel [**?** ], Gosh! [**?** ], Rubah [**?** ] and JavAdaptor [**?** ] provides the means to perform dynamic software update in Java. However, they are not integrated with refactoring tools. The

refactoring operations are performed on the source code. Even though having the support to migrate the instances, the developer is in charge of generating the migration patch and not the refactoring tools. The refactoring tools are not aware of the DSU tools, and the IDEs handle the changes statically only. Our solution introduces the integration of both tools. In the related works, the tools are not correctly integrated, depending on the developer to implement the changes performed by the refactoring by a DSU tool.

Moreover, in the work describing JavAdaptor, it have been used to implement automatic refactoring but the migration of instances have not been addressed. The lack of instance migration reduces the applicability of the solution. Our solution provides an integrated solution.

Working with a dynamic language does not change the disconnection between the ADSU tools and refactoring tools. An example of this is Pymoult [**?** ], that provides the support to migrate live instances after a refactoring operation. But none of the tools working on a live Python environment uses this support.

The dynamic nature of Javascript allows the change of the running code and modification of the instances. However, the refactoring tools present in Javascript development environments such as WebStorm [**?** ], Grasp [**?** ], Atom [**?** ] and Visual Studio [**?** ] handle all the changes in source code level, without caring about live programming.

Also the live programming tools for Javascript as Nodemon [**?** ], Firebug [**?** ] or Chrome Dev Tools [**?** ] do not handle the migration of live instances. They only migrate the code generating these instances. The live programming experience in these environments is not comparable, as they are not intended to implement complex automatic refactorings.

As we shown, the infrastructure to integrate DSU tools and automatic refactorings in IDE exist in the related work, although the transparent integration that we propose is not present in the related work.

## 8. Conclusion

In this paper, we identify the problem of refactorings corrupting instances when there are live instances of the modified classes. We propose a categorization of the *instance corruption*. And we show that 36.11% of refactorings described in the literature present this problem. Moreover, we proposed a refactoring implementation mechanism which solves this problem.

Our Atomic Refactorings mechanism preserves instances' state. It is suitable for live programming environments because it does not corrupt instances of refactored classes. We use an Atomic Dynamic Software Update engine (*gDSU*), this engine offers the possibility of modifying all the objects in a separated environment. When all the changes are performed, it replaces all the modified objects at once.

Our solution is integrated with a refactoring tool existing in an industrial platform. The use of this new extension is transparent to the developer. Developers perform the refactorings without thinking to regenerate the live instances.

As a future work, we can extend this atomic behaviour to all the changes performed in the live programming environment not only automatic refactorings. Providing ways to the developer of changing the code with the guarantee that the live instances are always consistent. Detecting the nature of the changes and the intention of the developer is not straight forward when the changes are constrained to automatic refactorings. So, one interesting future work is providing the developer a way of expressing the required missing information with the minimum bureaucracy.

**Acknowledgement**

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

## Appendix A. Detailed Analysis of Refactoring: Improving the Design of Existing Code

After analysing the impact to live instances of the refactorings described in *Refactoring: Improving the Design of Existing Code* [**?** ], we present here the detailed analysis of all the refactorings in the book.

There are 46 refactorings that does not affect live instances, 6 refactorings with complex corruption, 11 with class corruption, and 9 with internal corruption.

### *Appendix A.1. Refactoring without Corruption*

The following refactorings do not have any impact in live instances. The class structures are not affected.

| **Refactoring** | **Page** |
|---|---|
| Add Parameter | 275 |
| Change Reference to Value | 183 |
| Consolidate Conditional Expression | 240 |
| Consolidate Duplicate Conditional Fragments | 243 |
| Convert Procedural Design to Objects | 368 |
| Decompose Conditional | 238 |
| Encapsulate Collection | 208 |
| Encapsulate Downcast | 308 |
| Encapsulate Field | 206 |
| Extract Interface | 341 |
| Extract Method | 110 |
| Form Template Method | 345 |
| Hide Delegate | 157 |
| Hide Method | 303 |
| Inline Method | 117 |
| Inline Temp | 119 |
| Introduce Assertion | 267 |
| Introduce Explaining Variable | 124 |
| Introduce Foreign Method | 162 |
| Introduce Parameter Object | 295 |
| Move Method | 142 |
| Parameterize Method | 283 |
| Preserve Whole Object | 288 |
| Pull Up Constructor Body | 325 |
| Pull Up Method | 322 |
| Push Down Method | 328 |
| Remove Assignments to Parameters | 131 |
| Remove Control Flag | 245 |
| Remove Middle Man | 160 |
| Remove Parameter | 277 |
| Remove Setting Method | 300 |
| Rename Method | 273 |

*Appendix A.2. Refactoring with Complex Corruption*

| **Refactoring** | **Page** | **Explanation** |
|---|---|---|
| Change Bidirectional Association to Unidirectional | 200 | One of the sides of the bidirectional association should be dropped. One of the two classes involved in the association will drop one instance variable. |
| Change Unidirectional Association to Bidirectional | 197 | The modified instances need the objects referencing to them to construct the bidirectional association. This information is not present in the modified instances. |
| Change Value to Reference | 179 | All the client instances of this object should be updated to reference the same object. |
| Introduce Null Object | 260 | All the clients using *null* in this field should be updated to use the newly created object. If this object should be shared, it should be done in the migration process. |
| Move Field | 146 | The field can come from any class in the system, the original and target classes are not related. The logic to match from the original instances to the target instances should be in the migration process. |
| Replace Data Value with Object | 175 | It exposes the same problems of *Extract Class* but the new instances should be shared, it makes more complex the migration process. |
| Replace Type Code with Class | 218 | It exposes the same problems of *Extract Class* but the new instances should be shared, it makes more complex the migration process. |

*Appendix A.3. Refactoring with Class Corruption*

| **Refactoring** | **Page** | **Explanation** |
|---|---|---|
| Collapse Hierarchy | 344 | All the instances of the subclass should be migrated to the superclass. As the subclass does not exist any more. |
| Extract Hierarchy | 375 | Some of the modified instances should be migrated to the new subclasses. The migration process should manage the determination of which class to instantiate in each case. |
| Extract Subclass | 330 | Some instances should be migrated to the new subclass. Also, the structure of the main class is changed. |
| Extract Superclass | 336 | A new superclass is extracted from the common part in two classes. The live instances need a migration if the class structure changed. |
| Introduce Local Extension | 164 | Some instances should be migrated to the new class. Establishing which instances to migrate is a responsibility of the migration process. |
| Replace Conditional with Polymorphism | 255 | A new set of subclasses are created, the live instances should be migrated to these subclasses according to the values of the original instance variables. Also, the instance variables of the subclasses might be renamed. |
| Replace Delegation with Inheritance | 355 | A pair of collaborating objects is integrated into the same hierarchy. Making the client of the delegation a subclass of the delegate. The instances of the client should be migrated to the new subclass, and all the instance variables of the delegate should be migrated to the client. |

| **Refactoring** | **Page** | **Explanation** |
|---|---|---|
| Replace Inheritance with Delegation | 352 | As a subclassification is replaced with a delegation, all the internal state of the single instance should be migrated to the new collaboration. Also, the delegate object should be created. |
| Replace a Subclass with Fields | 232 | All the instances of the subclass should be migrated to the superclass. Preserving the subclass state and adding the needed fields to distinguish from the superclass instances. |
| Replace Type Code with Subclasses | 223 | The instances should be migrated to new subclasses depending on the value of the type code. |
| Tease Apart Inheritance | 362 | As the hierarchy is split, the live instances should be split in the same way, putting the original instance state to the corresponding instances. |

*Appendix A.4. Refactoring with Internal Corruption*

| **Refactoring** | **Page** | **Explanation** |
|---|---|---|
| Duplicate Observed Data | 189 | As *Extract Class*, with the addition that the extracted has a reference to the original instance it was extracted from. |
| Extract Class | 149 | The live instances of the original class should be split in the new version of the mother instances and the newly created child instance. |
| Inline Class | 154 | The internal state of the child object should be migrated to the mother instance. |
| Pull Up Field | 320 | The structure of the class has changed, all the fields should be migrated to their new position in the object. |
| Push Down Field | 329 | The structure of the class has changed, all the fields should be migrated to their new position in the object. |
| Replace Array with object | 186 | The same problematic of *Extract Class*. |
| Replace Type Code with State/Strategy | 227 | The same problematic of *Extract Class*, only having more possible subclasses. |
| Separate Domain From Presentation | 370 | The same problematic of *Extract Class*. |

**Vitae**

**Pablo Tesone** is a research engineer at Pharo Consortium [5]. He obtained the 17 of December 2018 his PhD entitled *Dynamic Software Update for Production and Live Programming Environments* under the direction of Stéphane Ducasse (Inria Rmod team) and Luc Fabresse(CAR team of IMT Lille Douai Douai). His research topics include Dynamic Software Update Solutions applied to Live programming environments, distributed systems and robotic applications. He is interested in improving the tools and the daily development process. He is an enthusiast of the object oriented programming and their tools. He collaborates with different open source projects like the ones in the Pharo Community [6] and the Uqbar Foundation [7].

**Guillermo Polito** is a coding enthusiast, software engineer and researcher. He is research engineer at CNRS working currently in the RMoD [8] and Emeraude [9] teams. His research targets programming language abstractions and tool support for modular long-lived systems. For this, he studies how reflective systems can evolve while maintaining these properties. He is interested in how these concepts combine with distribution and concurrency. He obtained the 13 of april 2015 his PhD entitled *Virtualization support for application runtime specialization and extension* under the direction of Stéphane Ducasse (Inria Rmod team) and the supervision of Noury Bouraqadi and Luc Fabresse(CAR team of Mines Douai). He loves coding and spends a lot of my free time helping the amazing community of Pharo. He also participates in several projects such as the Pharo's database driver suite (DBXTalk), its shortcut framework, or the static web page generator Ecstatic.

[5] http://pharo.org/
[6] http://pharo.org/
[7] http://www.uqbar.org/
[8] http://rmod.lille.inria.fr
[9] http://www.cristal.univ-lille.fr/emeraude/

**Luc Fabresse** is associate professor at IMT Lille Douai, France. His researches aims at easing the development of mobile and constrained software using dynamic and reflective languages such as Pharo. One of his goal is to support live programming of mobile and autonomous robots in an efficient way. He is the co-author of multiple research papers (http://car.mines-douai.fr/luc) and he concretizes all these ideas (models and tools) in Pharo to develop, debug, test, deploy, execute and benchmark robotics applications. Each year, Luc also gives computer science lectures, co-organizes events (technical days, conferences, ...) and promotes Smalltalk as an ESUG (European Smalltlak User Group) board member.

**Noury Bouraqadi** is a full professor at IMT Lille Douai, France since 2001. His research addresses mobile and autonomous robots from two complementary perspectives: Software Engineering (SE) and artificial intelligence (AI). From the SE perspective, he studies software architectures, languages and tools for controlling individual robots. He mainly focuses on reflective and dynamic languages, as well as component models, for a modular and agile development of robotic software architectures. From the AI perspective, he studies coordination and cooperation in robotic fleets. He mainly focuses on communication models as well as emerging or predefined organizations for multi-agent robotic systems.

**Stéphane Ducasse** is directeur de recherche at Inria. He leads the RMoD [10] team. He is expert in two domains: object-oriented language design and reengineering. He worked on traits, composable groups of methods. Traits have been

[10] http://rmod.lille.inria.fr

introduced in Pharo, Perl, PHP and under a variant into Scala and Fortress. He is also expert on software quality, program understanding, program visualisations, reengineering and metamodeling. He is one of the developer of Moose [11], an open-source software analysis platform. He created Synectique [12] a company building dedicated tools for advanced software analyses. He is one of the leader of Pharo [13] a dynamic reflective object-oriented language supporting live programming. The objective of Pharo is to create an ecosystem where innovation and business bloom. He wrote several books such as Functional Programming in Scheme, Pharo by Example, Deep into Pharo, Object-oriented Reengineering Patterns, Dynamic web development with Seaside. According to google his h-index is 51 for more than 11000 citations. He would like to thanks all the researchers referencing his work!

[11] http://www.moosetechnology.org/
[12] http://www.synectique.eu/
[13] http://www.pharo.org/