# Runtime Class Updates using Modification Models

Toon Verwaest Niko Schwarz Erwann Wernli
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

## ABSTRACT

Dynamic updates in object-oriented languages require high-level changes to be translated to low-level changes. For example, removing an unused instance variable from a class may shift the indices of other instance variables. The shift needs to be translated to a change of the bytecodes accessing these instance variables. Current languages do not offer a bridge between the two levels of abstraction. We outline such a model, and demonstrate its usefulness by discussing a prototype implementation in Pharo Smalltalk. In addition to simplifying the implementation of dynamic updates, our model enables easy experiments in modifying the language semantics.

## 1. INTRODUCTION

Dynamically updating software, *e.g.*, for developing and debugging the system without restarting it, has long been common practice in dynamic object-oriented languages such as JavaScript, Ruby, Smalltalk, *etc.* Despite their popularity and long history, dynamic software updates remain challenging. Changes such as removing and adding an instance variable can be challenging because it is unclear what should happen to the instances of the changed class. Changing the ordering of instance variables can be challenging, because while the semantics are clear, a possibly large number of methods is affected. In both cases, a high-level, structural change entails a number of changes on a much lower level.

From the point of view of a programmer of an object-oriented programming language, objects are instances of classes. Those classes are subclasses of other classes and so on. From the point of view of a language implementor, objects are merely strips of raw memory with a known structure. This distinction is fine so long as developers stay within their own camp. However, one cannot always stay on one's side of the fence. For example, compilers implemented in the language they compile[1], require knowledge of the looks of those raw strips of memory, despite being written in a language that abstracts away from those details.

We propose abstractions that bridge from the high-level structure to the low-level details. We reify both the high and low level abstractions in the programming language. Then, we show how these abstractions can ease previously messy tasks, such as dealing with structural changes, and implementing appropriate responses to those changes. As a case study we discuss how layout objects simplify dealing with structural changes to classes in Pharo Smalltalk.

At the high level, we propose abstractions that reify the structure of objects and their modifications. We propose explicit *layout* objects that capture the semantics of a strip of raw memory of an object, and explicit *slot* objects that capture the semantics of fields. Modifications to the layout are captured in a *class modification model* and refined into two *field modification models*, a *method modification model* and an *instance modification model*.

Our abstractions can be used as a foundation to support more elaborate forms of run-time changes. We sketch how it would facilitate scoping changes to support side-by-side deployment of multiple versions of an application—as Changeboxes allow [4]. We sketch how our model can facilitate the implementation of different approaches to dynamic software updating.

This paper is structured as follows: in Section 2 we explain the necessity of higher-level abstractions for lower-level details, describe the modification model and the rationale behind its design; in Section 3 we show how the modification model is used to dynamically update the system and we discuss alternative update strategies that would be simplified if implemented on top of our model; in Section 4 we discuss the relevant literature and we conclude in Section 5.

## 2. MODIFICATION MODEL

In this section, we show how our model captures the structure of instances as well as changes to their structure. That includes cascading changes. For example, when an instance variable is added to a class, this may change the index of fields in subclasses, requiring the modification of many methods that access the instance variable.

[1]This is common case, as Abel and Sussman assert [1, Section 4.1].

## 2.1 Object structure

The structure of objects is represented using two main abstractions, *layouts* and *slots*. We use them to construct a modification model that simplifies dealing with structural changes caused by changes to their classes.

*Layouts.* Layouts mediate between the instance formats used by the VM and the high-level language tools. Since in class-based languages the format of an object is dictated by its class, the layout metaobject is stored as a class variable. To avoid runtime performance overhead the instance format required by the VM is provided by the metaobjects but cached directly in the class.

VMs often support a multitude of instance formats. We call the layouts that correspond to these instance formats *primitive layouts*, other layouts are *custom layouts*. Since our prototype runs on Smalltalk, we have primitive layouts for integer, byte, word, pointer, variable pointer, weak pointer and compiled method. Metaclasses in Smalltalk are required to have instance variables (at least for the `superclass` link and the `methodDictionary`) so they correspond to pointer layouts. Primitive layouts are the ones that are already provided by the language. Custom layouts can be specified by the user, and then immediately become part of the language.

*Slots.* Slots are links between instance variables and the tokens that refer to them in the source code. As such, slots know the field index of an instance variable. Slots are stored in the layout of the class that declares them. Layouts holding slots point to the layout of the superclass since they inherit the slots of the superclass. This makes the layout reference graph parallel to the class hierarchy.

We modified the compiler so that it delegates the handling of instance variable accesses to the slots that generate the low-level code for instance variable access. By letting the slot object interact with the compiler we gain the following. (1) The slots are in charge of the semantics of a instance variable access. (2) We avoid performance overhead at runtime, because we inline the access code at compile time. Therefore, the resulting code, compiled using slots, is exactly as fast as the code compiled without slots.

Layouts and slots are the mediators between the low-level VM details and the high-level concepts in programming languages and other tools that need to interact with the VM. By separating the reified view of the structure of instances from the VM-level view, the language is no longer restricted to using the primitive layouts and slots directly supported by the VM. Custom layouts can be constructed having modified object layouts and access semantics and thus enrich the semantics of the language without modifying the VM. They also allow us to easily find which high-level constructs are related to low-level instance formats and field accesses. Now that we have these abstractions, we rely on them throughout the rest of the paper without going back to the VM-level.

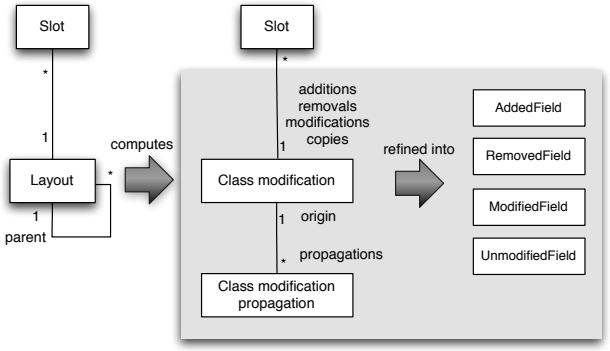


**Figure 1: Main classes in the modification model and how they relate to the other abstractions**

## 2.2 Modification model

Modification models capture changes to layouts and simplify the propagation of these changes to other impacted elements of the system. Figure 1 shows the main classes of the modification model and how they relate to layouts and slots. Following the previously argued necessity to split between high-level and low-level details, our model is comprised of class modifications (high-level) that can be refined into field modifications (low-level).

*Class modification.* A class modification captures the structural changes to a class at the slot level. A class modification is computed out of two versions of a class layout and contains separate lists of slots that have been added, removed, modified, and left untouched. Modifications to a class can impact its subclasses. As a consequence, a class modification can have so-called class modification *propagations*, which model changes performed to subclasses. Class modification propagations are themselves regular class modifications, and as such may have their own class modification propagations. Thus, recursively, every subclass can have its own class modification propagation.

*Low-level modifications.* Modifications to a high-level class have an impact on the related low-level structures. There are two modification models that transform the high-level model into concrete low-level modifications models, the *method modification model* and the *instance modification model*. Both models list for every field, whether it was added, removed, or shifted to a new position.

The instance modification model maps new positions onto old ones to initialize new instances from old ones.

The method modification model maps old positions to new ones to change accesses to field positions. In case the user has not provided custom slots or layouts, the system-provided slots act as follows. (1) If a field was shifted, all accesses to those fields are modified accordingly. (2) If a field was removed, all accesses to that field are replaced by a special native code sequence that the compiler does not otherwise create, to mark an illegal field access. If it is executed, it

displays an error message to the user.

# 3. SOFTWARE UPDATE

The specified modifications are carried out by informing the rest of the system of the change, and by applying the transformations in the instance and method modification model. In our approach, the responsibility to carry out the modifications lies with two main components: the class builder and the class installer.

*Class Builder.* The class builder is responsible for the structural part of modifying a class or creating a new class. It relies on the installer to fetch the old version of the class. It then uses the class modification model to compute the method modification and instance modification models. It then validates if these changes are semantically sound.

*Class Installer.* Once the class builder has correctly built and validated a modification model, the class installer is responsible for transactionally installing the change into the system. The class installer is the interface between the class building process and the rest of the live system.[2] It knows which subsystems to notify of changes, how to migrate live instances, and how to update existing methods. This allows different installers to implement different strategies to deal with the update of impacted method code, and with the migration of the instances of the impacted classes.

Let us describe how we reimplemented the default Pharo class installer using our model. We then sketch how more elaborate strategies could be implemented to support other forms of dynamic updates.

## 3.1 Pharo Class Installer

When a class is structurally changed in Pharo, the Pharo class installer migrates all instances and updates the methods of the class and its subclasses to reflect the structural change. It does not, however, update any running threads that might be affected by the change. This installer is greatly simplified by relying on our model since almost all the behavior is already captured by our generic class builder and class installer.

This specific installer first updates all methods of the old versions of the classes to adapt to the new versions of the classes. It relies on a *method field updater* to apply the changes using the method modification model. The method field updater decompiles[3] the bytecodes of the methods to a slightly higher-level IR (intermediate representation), updates the field accesses and compiles the IR back to bytecode. This saves a complete trip through all the phases of the compilation process, including parsing.

[2]Note that our model does not include a mechanism for migrating already running threads to the new version of the classes. This is the responsibility of more complex dynamic software update mechanisms that rely on our model—which are yet to be implemented.

[3]We built the decompiler for this paper, and made it available as part of the OPAL compiler package. `http://www.squeaksource.com/OpalCompiler.html`

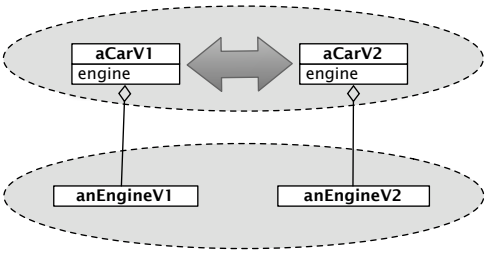


**Figure 2: The state in the two versions of the object is synchronized in a way to provide the illusion that only one object exists.**

During this process the field accesses occurring in the IR are linked back to the field modification models. These modification models know which slots are related to the particular field. There may be at most one slot of each the old version and the new version. This allows the slots to coordinate the updating of field accesses.

The installer then migrates live instances by creating new instances of the new versions of the modified classes from the existing instances. The installer implements instance migration by relying on instance modification models. After migrating the instances, it migrates the old versions of the classes to the new versions. Both instance migration and class migration happen in one single *stop-the-world* transaction.

## 3.2 Other installation strategies

Let us outline how, once our design is in place, other installation strategies can be installed in order to conduct experiments with modifying the semantics of the language.

*Side-by-side deployment.* Simultaneously running several versions of a module, service, or application in the same host is usually a complicated matter since programming languages lack dedicated mechanisms to scope changes [12] (even though they occasionally offer ways to muddle through, as is the case with the Java class loader [8]). ChangeBoxes [4], an approach that supports side by side deployment of applications in Smalltalk with first-class changes, required invasive changes in the system to intercept changes to classes and scope their visibility. In our model, the installer would be a natural fit for most of the adaptations. Rather than discarding the old class after a change, it could be modified to let different versions co-exist, and migrate the object instances accordingly.

*Passive partitioning.* Passive partitioning is an update strategy that lets existing threads run to termination with the old version of the system, while new threads use the newly updated code and data structures [9]. In previous work, we have explored such an approach using an experimental Smalltalk research platform for which we adapted the interpreter [6]. The challenge when old and new code co-exists, is to make sure that accesses to shared objects whose structure has changed is safe—we want for instance to prevent

that an old version of a method attempts to access a field that has been removed or shifted. To do so, the system keeps the old and new versions of the shared objects and synchronizes them when necessary. Two (or more) instances need to then transparently behave as one. This is similar to known problems that arise when using transparent wrappers and proxies [15]. In passive partitioning, unlike in generic proxies or wrappers, each object instance is local to exactly one thread. The state can be synchronized transparently upon field writes so as to keep the two representations consistent and restore the illusion of a single object. Figure 2 depicts such a situation where objects `aCarV1` and `aCarV2` are conceptually one single object. The same holds for `anEngineV1` and `anEngineV2`. When `anEngineV2` is written into the field `engine` of `aCarV2`, the system synchronizes the other representation accordingly and writes `anEngineV1` into the field `engine` of `aCarV1`. More complex synchronizations that address evolutions such as refactoring can also be supported. The design proposed in this paper would make an implementation of this approach in vanilla Smalltalk easy, as it exposes the necessary hooks in a convenient way: (1) slots could be extended to "weave" the necessary synchronization code during compilation[4] (2) the instance migration can be adapted to not discard the old version of an object after it has been migrated, but let the two versions co-exist. This is the strategy for side-by-side deployment pushed one step further: multiple software versions are not fully isolated any longer, and can also share objects.

## 3.3 Discussion

The original Pharo class builder takes a naive approach to updating bytecodes to a changed class structure: it recompiles all methods of the class. We replaced the original Pharo class builder by our own class builder. Our class builder makes use of the method modification model. This allowed us to experiment with language changes, and let us estimate the increase in code size that using method modification models entails.

The overall code size of our replacement of the class builder, including all models, is 2109 lines of code. Out of this, 1194 lines of code form the new class builder[5]. The size of the original Pharo class builder[6] is 1092 lines of code. Thus, the amount of code increased by a factor of 1.9, for the whole model, and for the mere class building, by a factor of 1.1.

The increase in complexity is compensated by a significant gain in flexibility. It also buys a significant performance gain in recompiling classes. The process of 10 times adding and immediately removing again an instance variable to a class with 14 subclasses was sped up from 31.2 to 4.6 seconds[7], leading to a speedup by factor 6.8. On a class with no subclasses, nor any installed methods, the same procedure took 1.2 seconds on both the default Pharo class builder and in our system. We conclude that our implementation combines higher flexibility and a clearer design with performance that is at worst as fast, and at times 6.8 times faster than the naive implementation.

[4]This step possibly requires adaptions of the run-time stacks as method size might increase after the synchronization has been woven in. Since stacks are first-class entities in Smalltalk, this can be done easily.

[5]In build 229 of our system on Squeaksource, `http://www.squeaksource.com/PlayOut.html`

[6]As ships in version 1.2.1 final of Pharo Smalltalk, `http://www.pharo-project.org/`.

[7]All measurements were performed on a 2011 MacBook pro at 2.3 GHz. We used the Cog virtual machine, build VM.r2378. The transformed class is RBProgramNode, as contained in Pharo 1.2.1 final.

# 4. RELATED WORK

Techniques to dynamically update production systems have been the subject of intensive research [19, 7, 9]. The emphasis in this case is on ensuring *program correctness* before and after the roll-out of the update, which requires well-timed migration of data [18]. Works in the field of dynamic updates focus on what constraints the migration should comply to and low-level implementation details, and not on the design of consistent and extensible language mechanisms to better support changes at run-time [12]. The problem of evolving object instances (sometimes called long-lived objects, or persistent objects) has been studied in the object-oriented database community [2, 14].

Change oriented development [17, 11, 5] aims at tracking and enabling changes in software with fine-grained change models. The goal of these models is to provide better insights in the nature of software development and provide better user experiences in IDEs. Our goal is different and we aim at supporting dynamic evolution with a modification model that abstracts low-level details and provides higher-level abstractions that can be extended. Penn *et al.* [14] provide a classification of all possible software changes. Our prototype supports all changes they list.

Dynamic software update is a cross-disciplinary research topic that covers software-engineering, programming language design, and operating systems [9, 13, 19, 18, 7, 3, 10]. The challenge to provide developers with a simple programming model that is practical—safe, efficient, and that allows developers to easily specify the necessary custom migration logic—is still open. Reflection has traditionally been used to provide means for run-time adaptations [16]. It is however orthogonal to safety, and it is then also a challenge to extend the reflective architecture so as to support safe dynamic updates.

# 5. CONCLUSIONS

Higher-level abstractions over VM level details are advantageous to language experiments, and overall system design. These abstractions should (1) abstract away from low-level details at the level of the virtual machines, and (2) capture changes to classes in a fine-grained manner. We proposed object layouts and slots as explicit higher-level abstractions to be used instead of implicit knowledge in the codebase. We proposed a modification model that captures changes to layouts and slots. We demonstrated the benefits of our design with a concrete implementation in Pharo Smalltalk. We show how at a moderate increase in code size (by factor 1.9), we gained flexibility for language experiments and sped up class reshaping by a factor of up to 6.8. We have sketched how one could build a dynamic update system on top of our model that could implement passive partitioning—old threads run old code, new threads run new code—of the

impacted entities; such an update scheme is safer than the regular update mechanism.

## Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012).

## 6. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT electrical engineering and computer science series. McGraw-Hill, 1991.

[2] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003.

[3] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.

[5] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 3–24, New York, NY, USA, 2007. ACM.

[6] D. Gurtner. Safe dynamic software updates in multi-threaded systems with ActiveContext. Master's thesis, University of Bern, Apr. 2011.

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

[8] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998.

[9] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.

[10] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[11] O. Nierstrasz. Putting change at the center of the software process. In I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, editors, *International Symposium on Component-Based Software Engineering (CBSE) 2004*, volume 3054 of *LNCS*, pages 1–4. Springer-Verlag, 2004. Extended abstract of an invited talk.

[12] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gaelli, and R. Wuyts. On the revival of dynamic languages. In T. Gschwind and U. Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.

[13] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002.

[14] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented DBMS. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 111–117, Dec. 1987.

[15] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

[16] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, Apr. 1996.

[17] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, Dec. 2008.

[18] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10(2):53–65, 1993.

[19] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.