Transaction Layers

Controlling Granularity of Change in Live Programming Environments

Toni Mattis* Patrick Rein* Robert Hirschfeld*,‡

*Hasso Plattner Institute, University of Potsdam, Germany

‡Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA

{firstname}.{lastname}@hpi.uni-potsdam.de

Abstract

Modifying source code in a *live programming environment* changes the behavior of currently running programs immediately. When complex changes affect multiple locations in the code before reaching a consistent state, running programs are in danger of "de-railing" when their control flow reaches the yet incomplete "construction site".

Context-oriented Programming provides *layers*, which encapsulate code that would otherwise be scattered over many modules and can be activated to jointly adapt program behavior at run-time.

We propose to transparently collect and group changes to the code in a COP layer and defer its activation until the programmer deems its change to be completed. Additionally, layer deactivation serves as immediate undo operation on the group of changes.

We present and discuss a Squeak/Smalltalk prototype consisting of a code editor, which provides control over when and where such a group of changes is active, and an extension of Squeak's COP implementation *ContextS/2* required for representing most code changes in a layer.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.6 [*Software Engineering*]: Programming Environments—Interactive environments

General Terms Context-oriented Programming, Live Programming, Tools, Self-sustaining Systems

Keywords Transaction Layer, Granularity of Change, Smalltalk, COP

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

COP'16, July 17-22 2016, Rome, Italy
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4440-1/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2951965.2951969

1. Introduction

Live programming environments support changing the source code of a program at run-time, such that changes quickly emerge in the running process. This enables programmers to obtain immediate feedback after every incremental change[14]. Typical examples include *Smalltalk*[3], *Lively*[8], and some educational environments like *Etoys*[2] and *Scratch*[11].

Transforming a source code modification into a change in the executable representation occurs at a predefined level of granularity. For example, in Squeak/Smalltalk[7], every time the programmer saves a method or class, it is being compiled and replaces the previous executable code, i.e., the meta-objects holding the bytecode or representing the class structure. In live programming environments, changing the executable representation also affects running computation immediately.

For groups of changes and refactorings, developers can often anticipate that multiple translation units will be affected. Executing a partially completed change can, however, cause running instances to "de-rail" into inconsistent state or incompletely implemented code paths. Having removed code from one method to insert it in another method later is a typical example of a temporary inconsistency which may yield undesired computation if executed. Currently, developers have to figure out an order in which modifications keep the running program on track, or stop and restart the program, thereby forfeiting the benefits of a live environment.

Context-oriented programming (COP)[5] is an approach to modularizing cross-cutting concerns. The particular family of COP implementations we are concerned with introduces *layers*, which contain code that can be activated or deactivated to jointly modify behavior across multiple modules. Layer activation is composable, i.e., multiple layers can be active at the same time and compose their behavioral modifications. Throughout this paper, we will refer to layer-based context-oriented programming when mentioning COP.

We argue that changing the executable representation in response to a code change (*translation*) and changing the behavior of a running program instance (*emergence*) can be treated as two separate steps. While the programmer may want a single modification, such as a method, to be syntax-checked, compiled and become testable, the whole running program should only commit to the new behavior once a particular group of more primitive changes has been completed.

By interpreting the set of changes as a COP *layer*, the activation of this layer at run-time can be deferred to a point where programmers have completed their coherent unit of modification (Figure 1).

In section 2, we give a small example to illustrate the problem. In section 3 we propose an approach to solve this problem. A prototypical implementation of this concept is outlined in section 4. Limitations and directions of future work are examined in section 5 and section 7 summarizes our results.

2. Example

The following example represents a simple refactoring scenario in Smalltalk to demonstrate the effects of method-level change granularity. Consider an application which continuously queries the Twitter API and indexes the resulting `Tweet` instances. The `Index` class responsible for maintaining the index currently runs the following code repetitively for each incoming tweet:

```
1 Index ⇒ indexTweet: aTweet
2   | hashtags |
3   hashtags := aTweet words
4     select: [ :word |
5       word beginsWith: '#'].
6   hashtags do: [ :hashtag |
7     self dict at: hashtag
8       put: (aTweet id)]
```

The programmer currently reviewing this method decides to move the part which selects hashtags (Lines 3 to 5) to the `Tweet` class, resulting in the following code:

```
1 Index ⇒ indexTweet: aTweet
2   aTweet hashtags do: [ :hashtag |
3     self dict at: hashtag
4       put: (aTweet id)]
5
6
7 Tweet ⇒ hashtags
8   ↑ self words select: [ :word |
9       word beginsWith: '#']
```

Keeping in mind that the `indexTweet` method may be called by the running indexer at any time, the programmer has to consider creating the `Tweet⇒hashtags` method before replacing code in `Index⇒indexTweet:`. Otherwise, the `hashtags` message send in line 2 would fail with a `MessageNotUnderstood` exception[1].

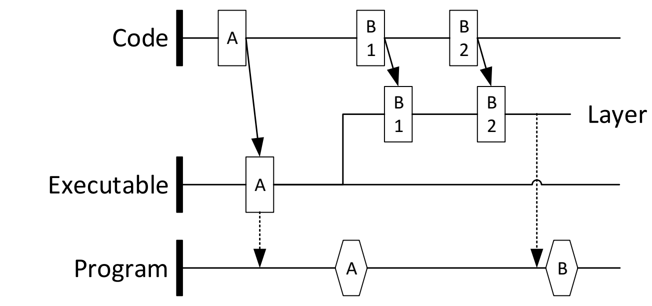


Figure 1. Interdependent changes are translated into a layer which gets activated when deemed ready. Joint behavior and state of both changes emerge atomically.

While this situation is manageable by experienced programmers without significant mental load, it serves as minimal example for the family of problems we aim to address.

3. Transaction Layers

For the scope of this paper, we focus on class-based single-inheritance OOP. The following types of fundamental editing operations are common in such environments:

- Changing the source code of a method
- Adding or removing a method
- Adding or removing an instance variable
- Changing the superclass of a class
- Adding or removing a class

We consider renaming operations being a composition of add and remove steps.

We aim to capture these editing operations in a COP layer that is not activated by default. We call this layer *transaction layer*, because it describes a set of changes to our system that is only consistent as a whole.

Operations on Transaction Layers The following workflow operations should be supported by code editing tools:

- *Open* a new transaction layer. From now on, each change is compiled into this initially empty layer, not into the base system. The code shown in the editor includes all changes in the currently open transaction layer.
- *Test / Work with* a layer. Changes in this layer are scoped to either a test runner or a workspace, but not all running instances of this code.

[1] A standard Smalltalk debugger would allow the programmer to correct the implementation and retry executing the exception site, however, exception-driven workflows are not always feasible or desired.

- *Activate* the open layer (*do*). The changes collected by this layer become globally effective but not yet part of the persistent code base.
- *Deactivate* an active layer (*undo*). This effectively undoes the behavioral modifications but does not discard code changes.
- *Commit* a layer, such that changes are permanently merged into the code base.
- *Abort* the open transaction layer. Changes now emerge immediately again, edits from the aborted layer are no longer visible in the editor. It is up to implementation details whether the layer is being preserved for later reuse.

We do not claim that this particular set of operations is complete or orthogonal, however, it yields a pragmatic set of operations that supports our proposed workflow.

Desired COP Features Basic COP implementations, such as *ContextS/2*, only support layered methods. This is already sufficient to collect all changes to method code. A tool should, as soon as there is an open transaction layer, encode all method changes as partial methods and display the partial method of the currently open layer instead of the base method.

In order to capture changes to classes themselves, we assume the underlying COP implementation to provide *layered classes*. The layer itself is required to support the following run-time adaptions:

- *Layer-local instance variables* as proposed in the *L* language[6]. Only partial methods in this layer can access them on instances of the specified class, thereby implementing transactional isolation. When the transaction layer is being committed, they are added to their respective classes in the base system.
- *Blocked methods and variables*. Other methods are forbidden access to these variables and methods. Upon commit, they are removed from their respective classes in the base system.
- *Dynamic super binding*. The method resolution order can be influenced by the layer, such that it can redirect super calls to a different class than in the base system.

4. A Smalltalk Prototype

To build a code editor in Squeak/Smalltalk that fulfills the requirements stated in section 3, we first extend a COP implementation for Squeak/Smalltalk, *ContextS/2*, to support layered classes. We then use the tool building environment *Vivide* to build a Smalltalk-style system browser that supports transaction layers.

Using *Vivide* and *ContextS/2* to build tools for context-oriented software development has been explored by Taeumel *et al.*[13], which is also the first public appearance of *ContextS/2*.

4.1 ContextS/2 and State Extension

Layered Methods *ContextS/2* originally implements layered methods by replacing the `CompiledMethod` object in the method dictionary of a class by a wrapper `LayeredMethod`, which contains the compiled base method and all partial methods indexed by layer name. Layers are identified only via their name, but a reified `Layer` object will be created lazily once a symbol is being used as layer name.

When a layered method is being called, it asks the currently running `Process` for active layers and invokes the partial method associated with the topmost layer.

If a class receives a new method and its source contains a `<layer: #myLayer>` pragma at the beginning, the method is compiled and placed in the wrapper's partial method dictionary at key `#myLayer`, otherwise it creates or replaces the base method. `#myLayer withLayerDo: [ ... ]` activates the layer by notifying the process that `#myLayer` should be added to the stack of active layers, executes the block, and finally removes the layer from the process.

Layered State Our extension adds a doubly nested mapping (`variable name` → `(instance` → `value))` to each layer, which holds dynamic state. In addition, layers carry a blacklist of blocked variables.

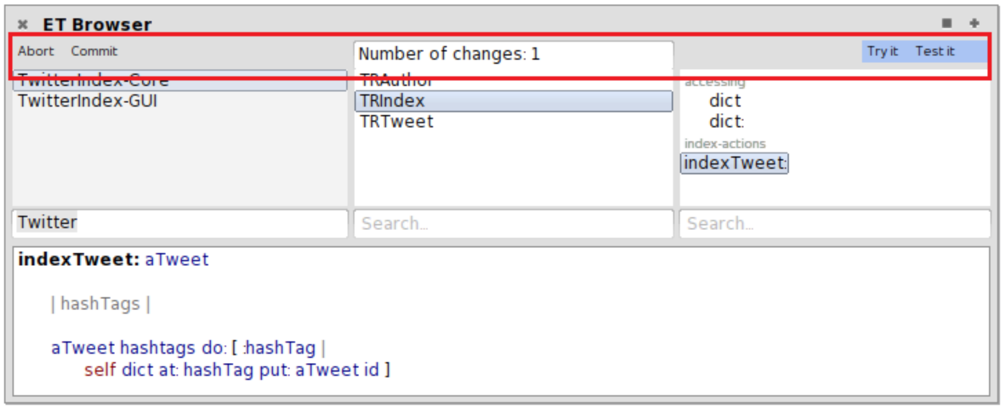
The `ClassDescription` class, of which regular classes are instances, stores a list of layered variable names specific to this class. They will be subject to a special lookup procedure upon compilation.

Resolution of these variables is realized in the compiler, but implemented as a COP layer itself to scope compiler modifications to our tools only. The compiler checks whether an instance variable is contained in the list of layered instance variables of the `ClassDescription` and, if so, compiles a specialized lookup into the method, that asks the active layers for a value. Blocked variables compile without error (in contrast to removed variables), but will signal a run-time error when accessed.

4.2 Tooling

Using *Vivide*, we created a three panel browser, which is typically seen in Smalltalk environments and depicted in Figure 2 with the example from section 2.

After clicking the Open button, which changes to Abort while a transaction layer is open, the programmer can start writing code in the transaction layer. The accumulated number of changes is displayed and acts as a reminder of how old the transaction layer is. Try it activates the transaction layer, while Test it opens a modified test runner, which executes the test suite inside the currently open transaction layer. Commit merges changes back into the system.



```
indexTweet: aTweet

    | hashTags |

    aTweet hashtags do: [ :hashTag |
        self dict at: hashTag put: aTweet id ]
```

Figure 2. Code browser with an active transaction and buttons for transaction control (highlighted).

Implementation of Changes Transaction layers receive a random identifier upon creation. As long as a transaction layer is open, a save action (CTRL + S) results in the following steps:

If a method has been edited, the method source code is rewritten to contain a pragma `<layer: #id>`, with `id` being the random ID of the current transaction layer. The rewriting is implemented using an *Ohm/S* attribute grammar[10]. Due to the pragma, the compiler translates the rewritten source to a partial method.

If the class is being edited, current instance variables will be compared to the last change. Added instance variables will be moved to a list of layered variables of the class, removed variables will be moved to the blacklist of the layer.

Limitations and Non-controllable Changes Changing the set of classes is not transactional, yet. This is not a major concern given the way Squeak works: First, a new class does not have any effect, except in code that explicitly monitors the global dictionary of classes, such as code browsers and editors. Second, since classes are bound during compilation, class deletion or renaming (effectively copying the class under a new name) has no immediate effect. Methods continue to use the old class and so do live instances. Introducing new classes, renaming classes, or removing references to an obsolete class require method changes and recompilation, which gets captured by the open transaction layer.

Changes to the superclass are not captured, as our extension of *ContextS/2* has no support for dynamic super binding yet. Also, only a single transaction layer can be opened per code editor at the moment.

5. Discussion and Future Work

Invasiveness of Our Tooling Transactional change management gives rise to a trade-off: On the one hand, being aware of the state of the transaction layer and remembering to activate and regularly commit the layer creates additional mental load. On the other hand, changes can be implemented in any order coming to mind, possibly reducing frustration by not being able to *just program* without running into failure states. Under which conditions our proposed workflow increases productivity remains to be shown.

As it stands, our tool requires careful manual interaction with the transaction layer. If a particular change crashes the application because no transaction layer was open, it is not yet possible to retroactively move the change to a layer and recover the application state.

Interaction with Tests From preliminary experience with our prototype we suspect that testing a set of changes before letting them emerge boosts confidence. This aligns well with test-driven workflows. A possible extension would be to always maintain an open transaction and automatically commit if tests are green under local layer activation, then immediately start a new transaction. This way, changes can emerge during test runs but not in a live instance until they are “safe”. If testing is not done upfront, the tool might prevent the transaction layer from committing if the change is not covered by tests.

Changes from External Sources Changes often originate from external sources, e.g. loading changes from version control or from external files. Especially in live programming environments, loading a faulty revision may render the whole environment unusable, or the update introduces new dependencies which also need to be loaded before it can be safely applied. Loading external changes into a transaction layer enables experimentation in a sandboxed, easily reversible manner before merging them.

This extends to dynamically loaded code as well, e.g. the program itself may employ a transaction layer to exercise tighter control over a third-party library or plug-in. However, this cannot be considered a sandbox, since a locally active layer can still cause global effects.

Capturing Changes to Layers As we use COP to build a tool for only class-based OOP, we relinquish further potential for COP usage within this tool. A transaction mechanism which captures changes to layers as well might be useful to circumvent this limitation.

This gives rise to explore *higher-order layers*, which are capable of runtime adaptions that jointly modify classes and layers. The exact semantics of this concept remain to be defined, yet this idea indicates that transactional editing tends to add a new dimension of dispatch to whatever programming model is being edited.

6. Related Work

CoExist Steinert *et al.*[12] proposed an environment which creates a new version for every individual change. The *CoExist* environment allows programmers to go back in time, run multiple versions simultaneously and compare them not only as code, but also as running instances. Granularity can be modified retroactively by grouping multiple correlated changes, which can even be used as change sets in a version control system. On the one hand, programmers are freed from remembering to explicitly start and commit a new version when using CoExist, on the other hand they cannot choose to defer changes from being applied instantly, only revert in case of failure. A combination of both scenarios gives rise to promising future work.

Development Layers Lincke *et al.*[9] addressed the problem of breaking development tools while modifying them within themselves. This particular problem of self-supporting live systems has been solved using the same idea of collecting changes in a COP layer. This layer is activated for another instance of the modified tool, such that the new version can be tried without breaking the old version used for editing. The size of a change captured in a development layer is intended to be much larger than our incremental layers and the presented implementation in a prototype-based language is largely different from a class-based, object-oriented environment. However, our proposed tool integration and merging of changes directly follows the direction of future work outlined in the *Development Layers* paper.

Changeboxes Changeboxes, proposed by Zumkehr *et al.*[1, 15], allow multiple versions of a set of classes to co-exist. A changebox is similar to a layer, as message lookup can be redirected to a particular version of a method depending on the active changebox. Similar to development layers, their scope extends to a larger change, and they are suited for on-the-fly deployment of new versions. A key feature is that changeboxes can be merged to add up changes from different sources. The same effect as merging changeboxes can be achieved by activating multiple transaction layers.

Transactional Contexts The work of Gonzlez *et al.*[4] on Transactional Contexts demonstrated that atomic, isolated, and abortable transactions can be managed using COP by layering the underlying computation model. While their approach is logging changes to the application state (objects in our terms), we apply this concept of logging changes to behavior expressed through meta-objects.

This points towards a solution that unifies both, treating objects and meta-objects as state that can be jointly adapted transactionally. Further work in this direction could bridge the gap between state and behavior with regard to transactional change management.

7. Conclusion

We presented an idea to decouple translation and emergence of changes in a live programming environment using layers as translation target. This mitigates the problem of incomplete changes emerging in running programs and thereby provoking errors. Moreover, we presume that the presence of a safety net and the option of trying out the change in a sandboxed way increase confidence in a larger change. The effectiveness of trading in careful anticipation for transaction layers and the tools associated with them remains to be shown.

References

[1] M. Denker, T. Grba, 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: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 25–49. ACM, 2007.

[2] B. Freudenberg, Y. Ohshima, and S. Wallace. Etoys for one laptop per child. In *Creating, Connecting and Collaborating through Computing (C5) 2009*, pages 57–64. IEEE, 2009.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1983.

[4] S. Gonzlez, M. Denker, and K. Mens. Transactional Contexts: Harnessing the Power of Context-oriented Reflection. In *International Workshop on Context-Oriented Programming*, COP '09, pages 3:1–3:6. ACM, 2009.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology, March-April 2008*, 7(3):125–151, 2008.

[6] R. Hirschfeld, H. Masuhara, A. Igarashi, and T. Felgentreff. Visibility of context-oriented behavior and state in L. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, 32(3):149–158, 2015.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM SIGPLAN Notices*, volume 32, pages 318–326. ACM, 1997.

[8] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel: A self-supporting system on a web page. In *Proceedings of the Workshop on Self-Sustaining Systems (S3) 2008*, pages 31–50. Springer, 2008.

[9] J. Lincke and R. Hirschfeld. Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming. In *Proceedings of the Interna-*

tional Workshop on Context-Oriented Programming, COP '12, pages 2:1–2:6. ACM, 2012.

[10] P. Rein, M. Taeumel, and R. Hirschfeld. Gramada: Immediacy in Programming Language Development. Submitted for Review to *ACM Symposium for New Ideas, New Paradigms, and Reflections on Everything to do with Programming and Software (Onward!) 2016*.

[11] M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[12] B. Steinert, D. Cassou, and R. Hirschfeld. CoExist: Overcoming Aversion to Change. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 107–118. ACM, 2012.

[13] M. Taeumel, T. Felgentreff, and R. Hirschfeld. Applying Data-driven Tool Development to Context-oriented Languages. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, pages 1:1–1:7. ACM, 2014.

[14] S. L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34. IEEE Press, 2013.

[15] P. Zumkehr. Changeboxes – modeling change as a first-class entity. Master's thesis, University of Bern, 2007.