In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking Research: Making Design Thinking Foundational (pages 235-264)
Springer 2016 (doi:10.1007/978-3-319-19641-1_15)

Preserving Access to Previous System States in the Lively Kernel

Lauritz Thamsen, Bastian Steinert, and Robert Hirschfeld[1]

Abstract In programming systems such as the Lively Kernel, programmers construct applications from objects. Dedicated tools allow them to manipulate the state and behavior of objects at runtime. Programmers are encouraged to make changes directly and receive immediate feedback on their actions. However, when programmers make mistakes in such programming systems, they need to undo the effects of their actions. Programmers either have to edit objects manually or reload parts of their applications. Moreover, changes can spread across many objects. As a result, recovering previous states is often error-prone and time-consuming. This report presents an approach to object versioning for systems like the Lively Kernel. Access to previous versions of objects is preserved using version-aware references. These references can be resolved to multiple versions of objects and, thereby, allow re-establishing preserved states of the system. We present a design based on proxies and an implementation in JavaScript.

1 Introduction

Programming systems such as Squeak/Smalltalk [11, 9] and REPLs for LISP or Python allow adapting programs at runtime. Changes to programs in such environments are effective immediately and programmers can see or test right away what differences their actions make. Thus, these systems provide immediate feedback to programmers.

A subset of such systems, which includes, for example, Self [34, 33] and the Lively Kernel [13, 15], are those built around prototype-based object-oriented languages [17]. In prototype-based systems programmers create applications using objects and without having to define classes first. In Self and the Lively Kernel, programmers can inspect and change the state and behavior of objects at runtime. Programmers create actual objects, not source code that only abstractly describes potential objects.

The Lively Kernel was designed to support this kind of development [22]. It provides tools to directly manipulate the style, composition, and scripts of graphical objects. For example, programmers can change the positions and composition

[1] Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
e-mail: firstname.lastname@hpi.uni-potsdam.de

of objects directly using the mouse. They can use temporary workspaces to manipulate objects programmatically. They can edit and try methods directly in the context of graphical objects.

For example, to add new functionality to a graphical application, a Lively Kernel user might copy an existing button object and then modify the new button object: move the new button to a sensible position, resize it, set a new label, and add a script to be executed on mouse clicks. The user makes all changes directly to one button object. How this button fits into the application's interface is visible at all time. Clicking the button allows to directly test its functionality. This way, the Lively Kernel allows for fast feedback, especially during the development of graphical applications.

Programmers' changes to objects can turn out to be inappropriate. Programmers can, for example, accidentally change positions or connect the wrong objects when manipulating applications with mouse interactions. They might try a couple of different alternatives such as different colors and layouts, only to realize that an earlier state was most appealing. Similarly, programmers might learn in hindsight that making a change to an object's scripts introduced an error or impacts the application's performance. They might make a mistake in a code snippet, which then manipulates many objects. Moreover, problematic changes can be introduced when code is evaluated only to understand or test behavior, not to permanently change state.

However, when changes turn out to be problematic, programmers often need to undo the changes manually. The Lively Kernel does not provide an undo for changes to objects. This is especially at odds with the Lively Kernel's support for trying ideas right away: Developers are able to make changes directly and receive immediate feedback, but do not get support when such changes turn out to be inappropriate. Thus, to recover a previous development state, programmers often need to manually reset the state to how it previously was—probably using the same tools the changes were initially made with. Furthermore, this potentially involves multiple properties of multiple objects changed by multiple developer actions.

The Lively Kernel provides tools to commit and load versions of objects. In case such commits exist, programmers can load earlier versions of objects to re-establish previous states. Nevertheless, depending on how far the latest version is from the actually desired state, manual changes might still be necessary. To keep the effort to re-establish any previous state low, programmers would need to commit many versions. However, this contradicts the goal because committing many versions is also a significant effort. Some commits would be made only to protect intermediate states and not to share and document results. Especially when the preserved versions should be usable and documented, programmers would be required to test and describe many versions.

In summary, recovering previous states of objects in the Lively Kernel is currently a significant effort for programmers. They either have to manually reset changed state or need to take time-consuming precautionary actions.

A typical approach to implementing multi-level undo for the changes to application state is the Command pattern [8]. The Command pattern packages changes into actions. These actions can then be recorded to be able to subsequently undo them. This requires developers to implement undo operations for all possible actions. Therefore, an implementation of the Command pattern—even when limited to the Lively Kernel tools that manipulate objects—would be rather comprehensive. Furthermore, using the Command pattern requires developers to follow the pattern when implementing new tools. The Command pattern is entirely impractical for undoing the effects of evaluating arbitrary code from the Lively Kernel's workspaces and editors.

Worlds [37, 36], in contrast, is a more generic approach for controlling the scope of side effects. Code is executed in world objects, which capture all side effects. The worlds can then be used to run code with particular sets of changes. Developers could create new worlds for all their actions and discard worlds to return to previous states when necessary. Therefore, it still requires programmers to explicitly take precautionary actions, similar to version control systems. In addition, the implementation of Worlds in JavaScript is not yet practical. For example, it currently prevents garbage collection.

CoExist [28, 29] provides automatic recovery support without requiring developers to take precautionary actions. CoExist automatically records versions for every change and, thereby, provides a fine-grained history of intermediate development states. Programmers can review the changes chronological, examine the impact each change had, and re- establish previous versions. However, CoExist currently recognizes only changes made to the source code of classes. Its versions do not include the state of objects.

This thesis proposes an approach for versioning the entire state of programming systems as basis for automatic recovery support. In particular, this thesis introduces an approach to preserving and managing versions of all objects using alternative, *version-aware* references. Version-aware references are alternative references as they refer to multiple versions of objects. They resolve transparently to particular versions. Versions of objects are preserved together, so that version-aware references can be resolved transitively to the state of a particular moment. For this to be practical, versions of objects are kept in the application memory and the state of all versions is preserved incrementally on writes. To which versions the version-aware references resolve can be changed without significantly interrupting program execution: The version-aware references select the current versions dynamically instead of being hard-wired to specific versions.

We implemented our approach in JavaScript. The implementation does not require adaptions to established execution engines. Proxies [3, 6] are used to implement version-aware references: conventional references point to the proxies and the proxies delegate all object interactions transparently to particular versions of objects. Source transformations introduce proxies consistently for all objects. Therefore, programmers do not need to adapt their programs manually.

The approach supports fine-grained histories of development states. Not every state can be re-established, but versions that have been preserved. In this, the presented solution is a basis for recovery support that continuously preserves versions.

2 Background

This chapter describes prototype-based programming, the Lively Kernel, and CoExist. These works are the background of this thesis as we introduce an approach for providing CoExist-like recovery support in prototype-based programming systems, which we implemented for the Lively Kernel.

2.1 Prototype-based Programming

Prototype-based programming is object-oriented programming in which applications are created directly with objects, without requiring developers to define classes first.Self, JavaScript, and Kevo [32] are prototype-based programming languages. Many end- user programming systems such as Scratch [25], Etoys [14], and Fabrik [12] also enable users to express programs using objects.

Prototype-based programming allows building applications from particular objects. This is the fundamental difference to the class-based style of object-oriented programming, in which programs are expressed with classes. Each part of a prototype-based program has particular state.

There are different advantages associated with this kind of programming:

- [30] and [17] suggest that it might be easier for programmers to understand concrete examples than to grasp abstract classes. A concrete example provides particular values for its state and, in case of objects with a visual appearance, can be actually looked at.
- [34] and [2] describe how prototype-based programming makes it easier to introduce one-of-a-kind objects with their own structure or behavior.
- [2] and [24] argue that especially editing visual objects can be more concrete with prototypes. Instead of writing code that describes the appearance of objects, programmers can manipulate visual objects directly. Programmers could, for example, use the mouse to manipulate properties like the size, position, or to combine multiple elements. This way, programmers always see intermediate states instead of only receiving feedback on explicit test runs in-between edit-compile-load cycles.

Editing Graphical Objects at Runtime Many prototype-based programming systems, including the examples given in this section, allow manipulating objects at runtime. Scratch, Etoys, Fabrik, the Lively Kernel, and Self all provide tools dedicated to manipulating graphical objects directly. Such graphical objects range from basic objects like primitive shapes to complete applications like presentation software or programming tools. Prototype-based programming, programming at runtime, and direct manipulation of graphical objects seem to be properties that suit each other.

Similar Objects Without Classes Different prototype-based programming systems provide different approaches for creating similar objects. Self and JavaScript incorporate *delegation* to allow for prototypical inheritance. Objects can inherit state and behavior directly from other objects: each object has a *prototype* to which it delegates whenever looking up a property in the object itself yields no results. In Self, the prototype of an object is set when objects are cloned: The clone's prototype is the object it was cloned from.

In JavaScript, objects are created from constructor functions. The constructor function's prototype becomes the prototype of created objects. Kevo, in contrast, does not incorporate this notion of prototypical inheritance. It provides *concatenation* for incremental modification of objects [31]. Objects are copied to create objects with the same state and behavior as existing objects. These objects are self-contained. Changing an object only changes that particular object and a particular object can only be changed by directly changing it, not by changing any other object. To adapt many objects at once, programmers can use so-called *module operations* in Kevo. Module operations are evaluated for groups of objects.

2.2 The Lively Kernel

The Lively Kernel is a programming system in the tradition of Smalltalk and Self. Development happens at runtime. It incorporates tools and techniques to be completely self-sufficient. Thus, programmers can create versions of the Lively Kernel with the Lively Kernel. The Lively Kernel is a browser-based system. It is implemented in JavaScript and renders to Hyper Text Markup Language (HTML).

Programming with Prototypes and Classes

As the Lively Kernel is based in JavaScript, the system and applications are expressed in a prototype-based object-oriented language that provides prototypical inheritance. At the same time, the Lively Kernel also provides a class system and considerable parts of the system are expressed using classes.

The Lively Kernel implements Morphic [24], a framework for developing graphical applications. The graphical objects of this framework are called *Morphs*. Each morph has a class but can also have object-specific behavior. They can be created by instantiating a class or by copying an existing morph. Morphs are often edited directly and not through adapting existing or creating new classes. This way, the Lively Kernel mixes the class-based with the prototype-based style of object-oriented programming.

The Lively Kernel's copy operation does not establish a prototypical inheritance relationship between the copy and the original. Instead, it creates a full copy of the original morph's properties, including its class. Therefore, even though JavaScript incorporates prototypical inheritance, the Lively Kernel encourages programmers to use classes to share behavior among objects.

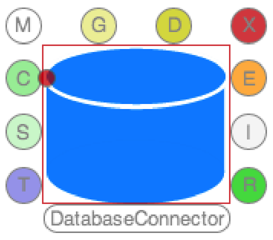
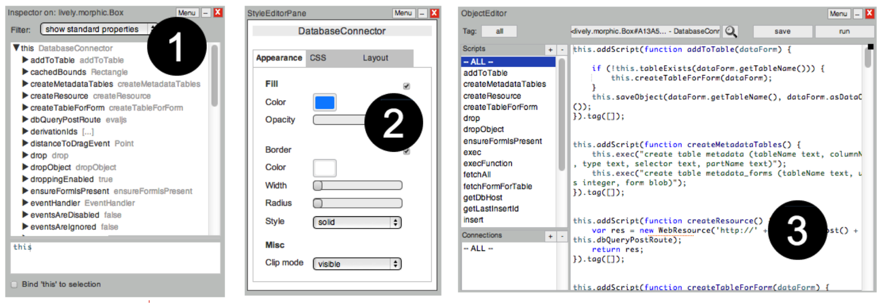


Fig. 2.1: The halo buttons of a basic morph.



▼ this DatabaseConnector
▶ addToTable addToTable
▶ cachedBounds Rectangle
▶ createMetadataTables createMetadataTables
▶ createResource createResource
▶ createTableForForm createTableForForm
▶ dbQueryPostRoute evaljs
▶ derivationIds [...]
▶ distanceToDragEvent Point
▶ drop drop
▶ dropObject dropObject
▶ droppingEnabled true
▶ ensureFormIsPresent ensureFormIsPresent
▶ eventHandler EventHandler
▶ eventsAreDisabled false
▶ eventsAreIgnored false

```
this
```

Bind 'this' to selection

Fill ✓
Color 🟦
Opacity
Border ✓
Color
Width
Radius
Style solid
Misc
Clip mode visible

-- ALL --
addToTable
createMetadataTables
createResource
createTableForForm
drop
dropObject
ensureFormIsPresent
exec
execFunction
fetchAll
fetchFormForTable
getDbHost
getLastInsertId
insert

-- ALL --

```
this.addScript(function addToTable(dataForm) {

    if (!this.tableExists(dataForm.getTableName())) {
        this.createTableForForm(dataForm);
    }
    this.saveObject(dataForm.getTableName(), dataForm.asDataC
());
}).tag([]);


this.addScript(function createMetadataTables() {
    this.exec("create table metadata (tableName text, columnN
, type text, selector text, partName text)");
    this.exec("create table metadata_forms (tableName text, u
s integer, form blob)");
}).tag([]);


this.addScript(function createResource()
    var res = new WebResource('http://' +                 st() +
this.dbQueryPostRoute);
    return res;
}).tag([]);


this.addScript(function createTableForForm(dataForm) {
```

Fig. 2.2: Three tools of Lively Kernel to manipulate morphs: the Inspector, the Style Editor, and the Object Editor.

Direct Manipulation of Morphs

Programmers can change the position of morphs by *dragging* and the composition by an alternative dragging, called *grabbing*. When a morph is grabbed, it can be added to another morph and becomes that morph's submorph. This way, a morph does not have to be a basic shapes or simple widgets, but can be the interface of any application.

The Lively Kernel provides a set of manipulation tools, called *Halos*, as shown in Figure 2.1. Developers can bring up these tools for each morph. The different buttons of a morph's halo allow, for example, resizing, rotating, and copying morphs. Other halo buttons open specific tools, which are shown in Figure 2.2:

1. The *Inspector* (1) presents all the values that make up a morph's current state. It also has a small code pane at the bottom that can be used to manipulate the morph's properties programmatically.
2. The *Style Editor* (2) allows manipulating certain aspects of a morph's visual appearance. Programmers can use it to change, for example, a morph's color, border width, or the layout of its submorphs.
3. The *Object Editor* (3) is a tool to edit the object-specific behavior of morphs. It shows all scripts of a particular morph and allows programmers to add, remove, and edit scripts.

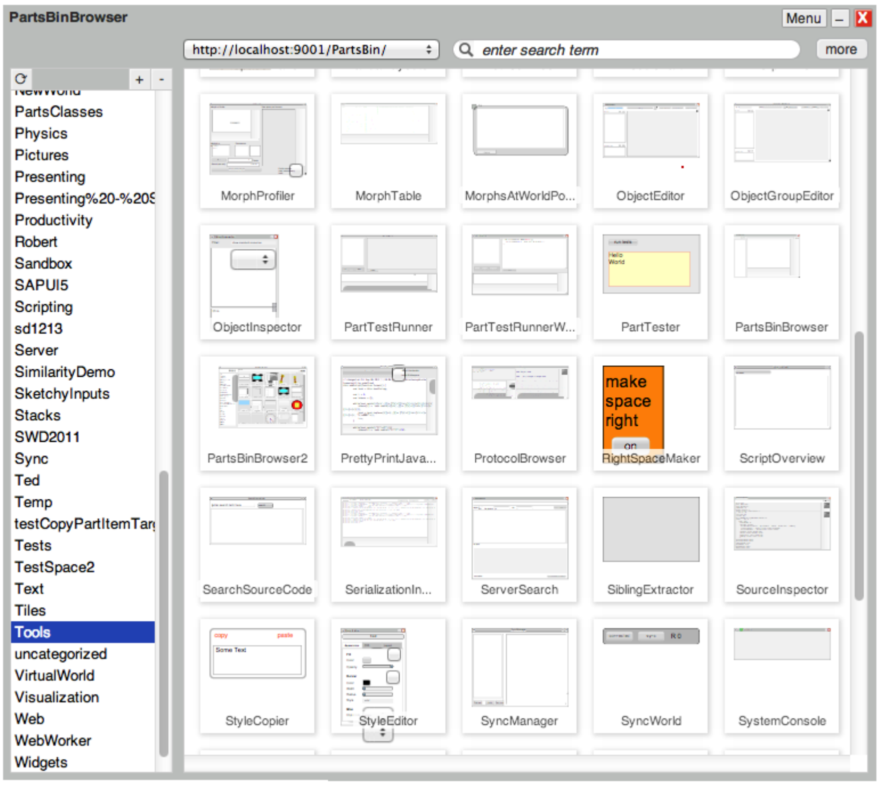


Fig. 2.3: The Lively Kernel's Parts Bin opened on the *Tools* category.

Saving Morphs to the Shared Parts Bin Repository

A related tool is the Lively Kernel's *Parts Bin* [23], an object repository to commit and load specific versions of morphs. Morphs saved to the Parts Bin are called *parts* to emphasize the ability to reuse any of the morphs in the Parts Bin for other Morphic applications. Figure 2.3 shows the Parts Bin, opened on the *Tools* category, which includes both the Style Editor and the Object Editor. Both these tools are examples for graphical applications developed from available parts. Their functionality is expressed in scripts and they are available to users through the Parts Bin.

The root of the scene graph of visible morphs is called *World*. Worlds are not shared via the Parts Bin, but can be saved as Web pages. A world stores the state of all visible morphs when saved and that state can be reloaded with the world.

2.3 CoExist

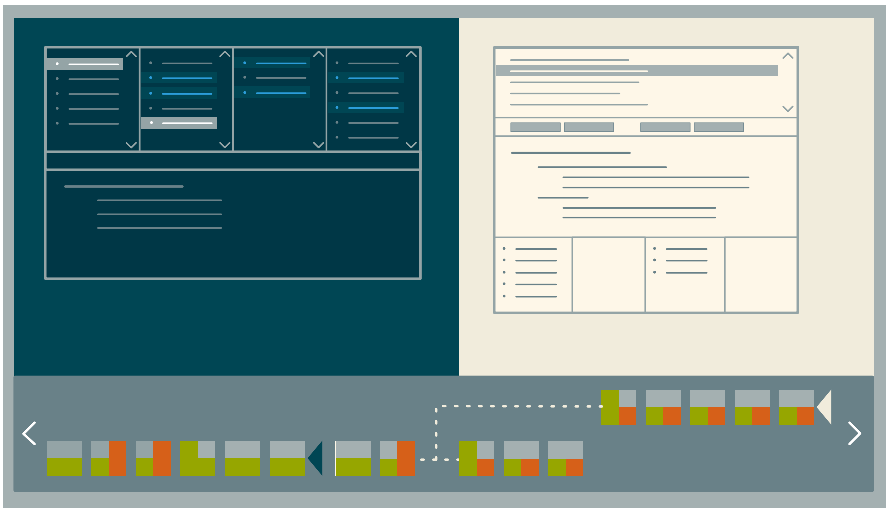


Fig. 2.4. Conceptual figure of CoExist featuring continuous versioning, running tests and recording the results in the background, and side by side exploring and editing of multiple versions.

CoExist preserves fast and easy access to previous development states [15]. It is based on the insight that the risk for tedious recovery is caused by the loss of immediate access to previous development states. With every change, the previous version is lost, unless it has been saved explicitly. This version, however, can be of value in future development states, when, for example, an idea turns out inappropriate. For that reason, CoExist creates a new version for every change to the code base. Users can rapidly switch versions or can access multiple versions next to each other. CoExist thus gives users the impression that development versions *co-exist*. Figure 2 illustrates some of the main user interfaces concepts of CoExist. It contributes the following concepts and tools:

- *Continuous Versioning* creates new versions in the background based on the structure of programs. It enables programmers to go back to a previous development state and to start over, which will implicitly create a new branch of versions.
- *User Interface Concepts* support browsing and exploring version information as well as identifying a version of interest fast. Two different tools are provided. First, the version bar highlights version items that match the currently selected source code element. Hovering the items will display additional information such as the kind of modification, the affected elements, or the actual change performed. Second, the version browser allows for exploring multiple versions at a glance. The version browser displays basic version information in a table view, which allows for scanning the history fast.
- *Additional Environments* to explore static and dynamic information of previous development states next to the current set of tools. Opening an addition environment is useful, when, for example, the programmer suddenly becomes curious about how certain parts of the source code looked previously or how certain effects were achieved. The additional environments also al- low for running and debugging programs. With that, users are capable of efficiently recovering knowledge from previous versions, which avoids the need for a precise understanding of every detail before making any changes.
- *Continuous and Back-in-time Analysis* for test cases and other computations. CoExist continuously runs analysis programs for newly created versions. As a default, it runs test cases to automatically assess the quality of the change made. The test result for a version is recorded and presented in the corresponding item of the version bar (left of Figure 2). The user can also run other analyses such as performance measurements. In addition to the continuous analysis features, CoExist provides full access to version objects and offers a programming interface to run code in the context of a particular version. So, whenever programmers become interested in the impact of their changes, they can easily analyze them in various respects. This allows programmers to ignore these aspects of programming at other times.
- *Re-assembling of Changes* for sharing independent improvements in separate commits. Users can extract selected changes to a new branch, test the result, and commit the achieved increment.

With CoExist, programmers can change source code without worrying about the possibility of making an error. They can rely on tools that will help with whatever their explorations will turn up. They no longer have to follow certain best practices in order to keep recovery costs low.

3 Motivation

In the Lively Kernel, programmers can create applications by manipulating and composing graphical parts. This chapter presents the development of such parts and related recovery needs by example.

3.1 Part Development by Example

To exemplify how developers work directly on objects in the Lively Kernel, we will outline how a Lively Kernel user adds a new feature to the Object Editor.

The editor has been developed by composing and editing graphical objects. Thus, the user does not adapt any source code modules to change the editor, but rather manipulates objects directly.

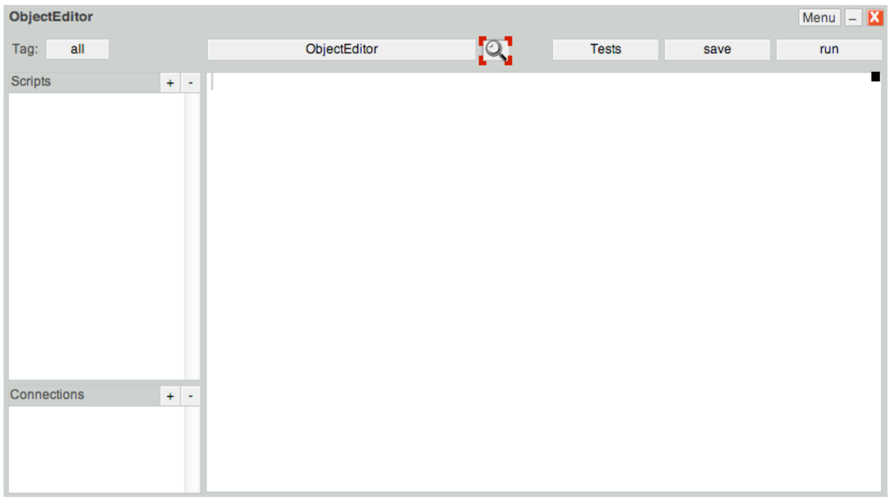


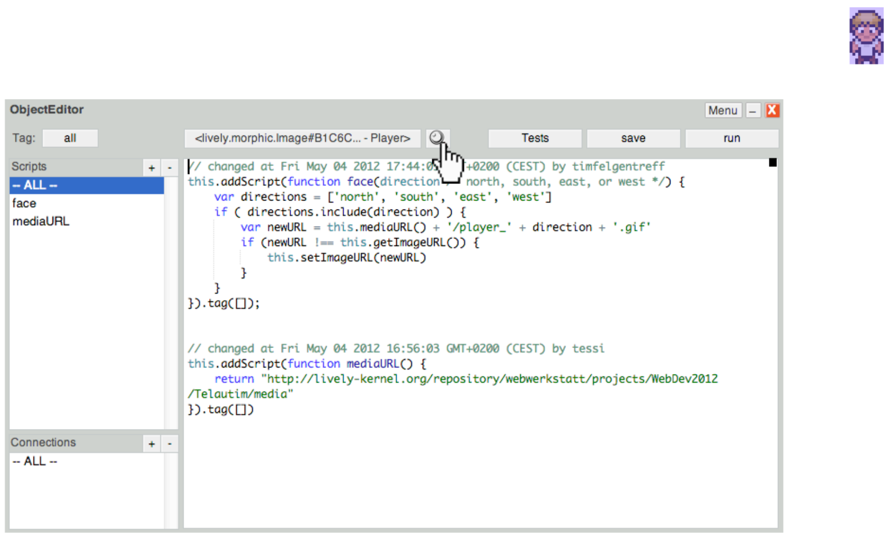
Fig. 3.1: The Object Editor's magnifier button highlighted with a red outline.

In this example, the user adds a magnifier tool to the Object Editor. The magnifier tool helps finding the editor's target, which is the object the editor currently presents scripts for. Implementing the new feature requires to create a new button morph and to add it to the editor, as shown in Figure 3.1. The magnifier button has two features:

1. When a programmer hovers over the button, the Object Editor's current target is highlighted with a rectangular overlay.

2. When a programmer clicks the button, the current target selection is revoked and the programmer can select the new target of the editor.

The following description covers the first of the two features, which is also shown in Figure 3.2 for an Object Editor currently targeting the character of a game.



```
// changed at Fri May 04 2012 17:44:0[illegible]+0200 (CEST) by timfelgentreff
this.addScript(function face(direction[illegible] north, south, east, or west */) {
    var directions = ['north', 'south', 'east', 'west']
    if ( directions.include(direction) ) {
        var newURL = this.mediaURL() + '/player_' + direction + '.gif'
        if (newURL !== this.getImageURL()) {
            this.setImageURL(newURL)
        }
    }
}).tag([]);


// changed at Fri May 04 2012 16:56:03 GMT+0200 (CEST) by tessi
this.addScript(function mediaURL() {
    return "http://lively-kernel.org/repository/webwerkstatt/projects/WebDev2012
/Telautim/media"
}).tag([])
```

Fig. 3.2: The Object Editor's magnifier button as it highlights the editor's target.

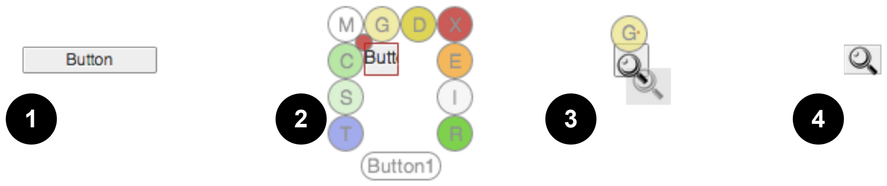


Fig. 3.3. Directly manipulating a button morph.

Manipulating the Button Morph Before implementing the button's behavior, the user first creates the button and manipulates its visual appearance. Figure 3.3 shows the steps in which the button is manipulated. A basic button, as visible in (1), can be found in the Parts Bin repository. In (2), the user resizes the button and gives it a square extent using the *Resize* halo button. Next, the user loads an image showing a magnifier icon. Using drag and drop, the image is added to the button in (3). Dropping a morph onto another connects the two morphs. Moving the button around will then move the image accordingly. Finally, the user adds the result of these manipulations, visible in (4), to the Object Editor.

All these changes are made directly to the state of objects: the button morph, the magnifier image morph, and the editor morph.When programmers edit parts in this way, they often see the effects of their actions immediately. For example, when adding the new button to the Object Editor, the button is visible at all times. Programmers do not need to run any code to see and test the button.

Scripting the Button Morph Now the user implements the button's behavior. The user adds scripts to the button that lay a translucent rectangle over the current target. In particular, the button receives two scripts: `onMouseMove` and `onMouseOut`. The implementation of the behavior includes the following:

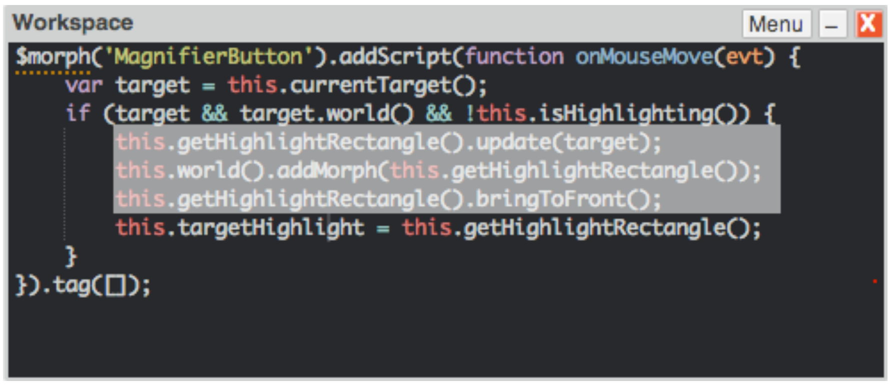
- The button holds a semitransparent rectangle morph.
- When the mouse enters the button (`onMouseMove`), the button resizes and adds the rectangle to the Lively Kernel world at the position of the target.
- When the mouse leaves the button (`onMouseOut`), the button removes the rectangle from the world again.

The Lively Kernel's scripting tools allow evaluating code in the context of their target objects. Hence, when programmers want to test a script or even just specific lines of code, they can try the behavior directly for the actual target.

3.2 Recovery Needs When Developing Parts

While manipulating objects directly, developers might make changes that they later want to undo. For example:

- **Accidental changes to state**: The user could accidentally move a morph and, thereby, change a carefully arranged layout. Similarly, meaningful state can be lost when a morph, for example the new button, is accidentally removed from the world.
- **Inappropriate changes through direct manipulation**: The user could make changes to the size, position, and colors of morphs to fine-tune the visual appearance of the editor's interface, only to decide later that a particular intermediate version was most appealing.
- **Accidental changes to scripts**: The user could introduce a typographical error to or accidentally remove a script. Moreover, editing a script could introduce a defect or a decrease in performance.
- **Inappropriate changes through scripts**: The user could make a mistake in a workspace snippet that is intended to manipulate morph properties programmatically. Such a snippet can change many properties of many objects.



```
$morph('MagnifierButton').addScript(function onMouseMove(evt) {
    var target = this.currentTarget();
    if (target && target.world() && !this.isHighlighting()) {
        this.getHighlightRectangle().update(target);
        this.world().addMorph(this.getHighlightRectangle());
        this.getHighlightRectangle().bringToFront();
        this.targetHighlight = this.getHighlightRectangle();
    }
}).tag([]);
```

Fig. 3.4. The button's `onMouseMove` script with a text selection.

Explorative Script Evaluation Undesirable changes can also be introduced when a programmer explores the behavior of objects by evaluating scripts. The Object Editor allows evaluating code directly for its target object. While such evaluation might help to understand the effects of particular code, it might also change the state of objects. For example, the user could be working on the button's `onMouseMove` script and could evaluate a few lines of code to quickly test them. These lines, as shown in Figure 3.4, would add the rectangle to the editor's current target. Only evaluating the selected lines would, however, neither check the conditions usually checked above nor set the state usually set below the selected lines. Therefore, evaluating this selection allows testing the highlighting behavior but leaves the system in a state it normally would not be in.

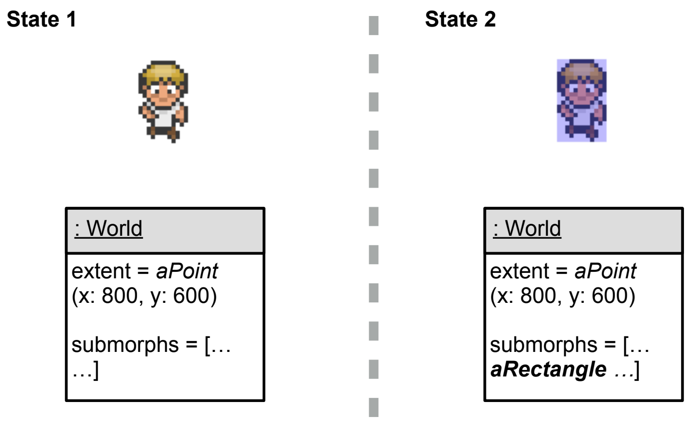


Fig. 3.5: Adding a submorph changes the state of a morph.

The examples show that there are many situations in which the user might want to undo previous actions. In programming systems like the Lively Kernel, where programmers work on objects, changes are always made to the state of objects. Functions are properties of objects. Even classes and modules are objects.

For example, evaluating the text selection in Figure 3.4 changes the world object's state. The world object has now one more submorph, as shown in Figure 3.5. Thus, the world's collection of submorphs is changed.

4 Object Versioning

This section introduces our approach to preserving access to previous states in systems like the Lively Kernel. The approach is based on alternative, version-aware references that manage versions of objects transparently. The section also presents a design that allows implementing version-aware references using proxies.

4.1 Version-aware References

In different versions of a system, objects have different states.

Versions of Objects

An object could represent an address. The state of such an *address* object could be as shown in Figure 4.1.

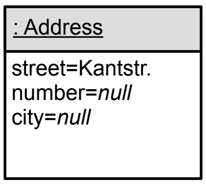


Fig. 4.1: An address object with three properties.

If values are assigned to the city and number fields of the address object, the object's state is changed. As the address object's state is part of the system state, changing the object's state changes the system state as well. If we call the initial state version *v1* and the state after making changes to the object version *v2*, the state of the address object is different in the two versions of the system, as shown in Figure 4.2.

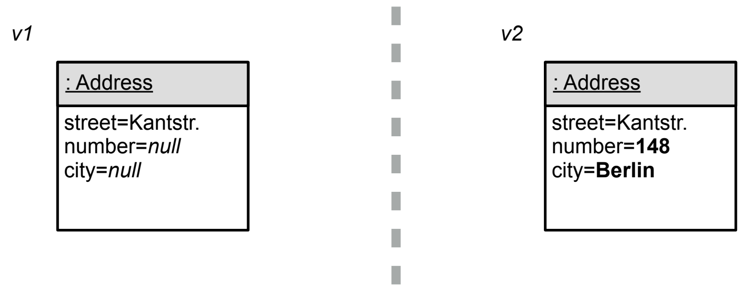


Fig. 4.2: Two versions of an address object in two versions of the system.

To be able to recover previous versions after making changes, the previous states of objects need to be accessible. For this reason, versions of objects are preserved and changes are made to new versions of the objects. A version of an object is, in the simplest case, a copy of an object. When the address object is changed in version *v2* of the system, the system does not change the original address object but the copy.

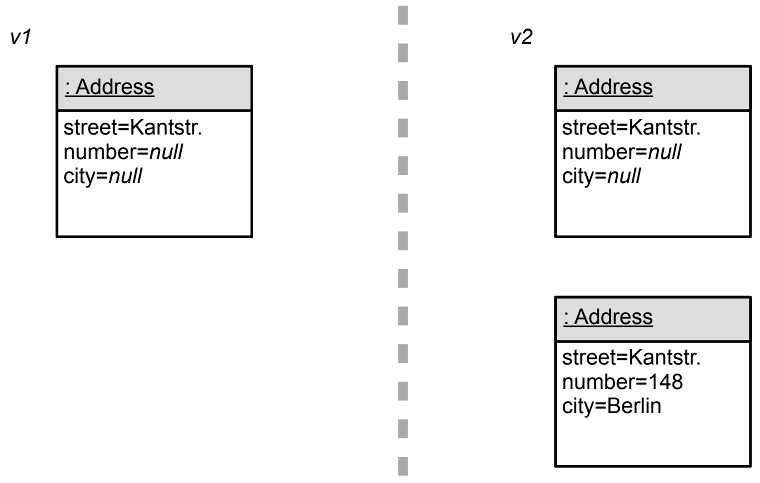


Fig. 4.3: Preserving the previous version of the address object.

As shown in Figure 4.3, there are now two versions of the address objects in version *v2* of the system. One of the objects holds the original state, while the other holds the state the object should have in version *v2* of the system. The two objects hold no information that indicates to which version of the system they belong. They also do not store any information showing that one object is a copy of the other.

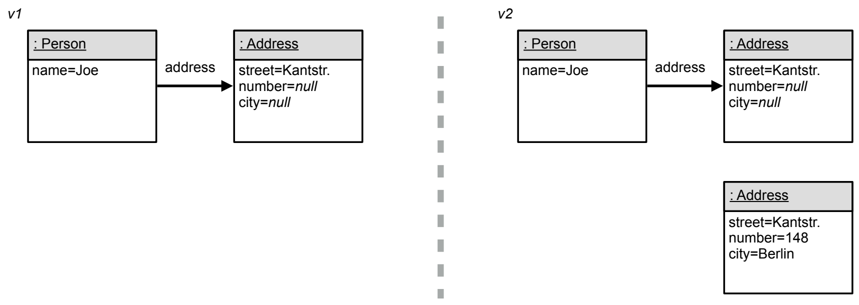


Fig. 4.4: A reference refers to the previous version of the address object.

At the same time, references to objects remain unchanged. For example, there could have been a person object referring to the address object. This reference would still be referring to the original address object, even in version *v2* of the system, as shown in Figure 4.4. Even after adding values to the fields of the address object, the following statement would still return true when `aPerson` refers to the person object:

```
aPerson.address.city === null
```

Version-aware References

Our approach uses *version-aware references*. Version-aware references know the available versions of an object and always resolve to one of those. Furthermore, version-aware references know which object version belongs to which system version. None of the versions is hard-wired to be the active version. Instead, the version-aware references resolve dynamically to the correct versions using context information.

Apart from that, the version-aware references behave like ordinary references. They can be assigned to variables and object fields, and are passed around.

When the person object uses a version-aware reference to refer to its address property, it can resolve to the versions of its address object. The version-aware reference knows both versions of the address object. In version *v2* of the system, it resolves to the second version of the object, as shown in Figure 4.5.

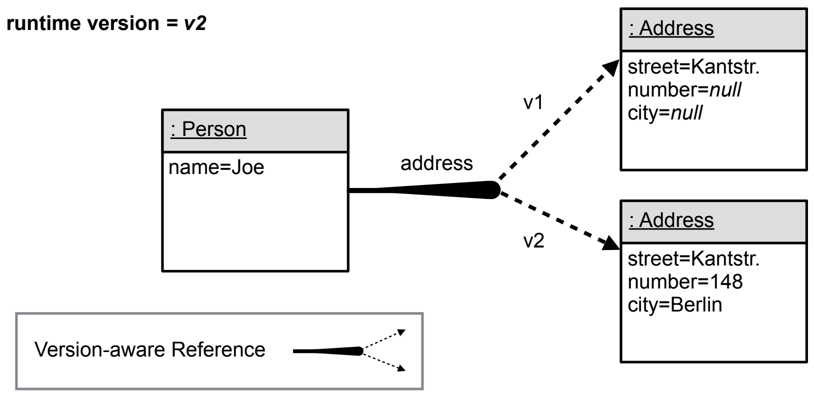


Fig. 4.5: A version-aware reference relates a person object to two versions of its address property.

In the same way, multiple version-aware references can be resolved as one path through a graph of versions. The version-aware references all choose versions of objects that belong to the same system state and, thereby, form the object graph of that state.

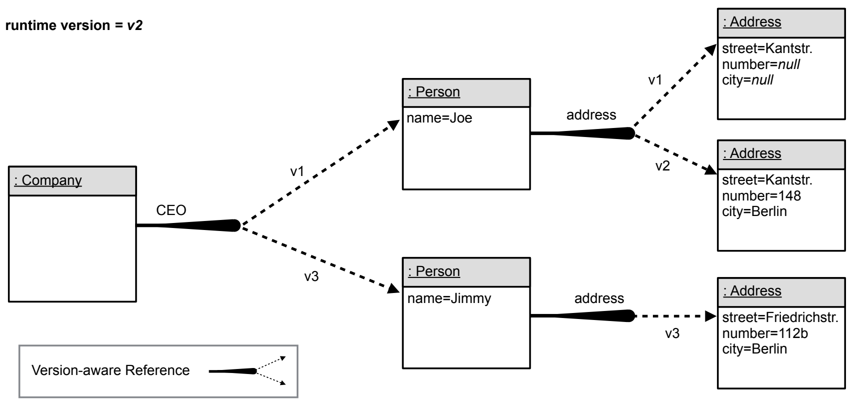


Fig. 4.6. An object graph with version-aware references.

Figure 4.6 shows an object graph that incorporates the previous example. The previously presented person object is a company object's CEO property. While the example shows that version *v2* is active, it also indicates a version *v1* and a version *v2* of the system. In version *v1*, the company's CEO has incomplete address information. In version *v3*, the company has a different CEO.

Versions of the System

To establish different versions of the system, the version-aware references have to resolve to different versions of objects. The version-aware references choose versions dynamically following a *version identifier*. Only this version identifier has to be changed to have version-aware references resolve to other versions of objects. For example, to undo the changes made with version *v2* of the system, the version identifier would need to be set to *v1* again.

Given the example situation from Figure 4.6 and given `aCompany` refers to the company object, the following statement would refer to three different values depending on the version identifier:

```
aCompany.CEO.address.number
```

Evaluating the statement in version *v1* would return the value null, in version *v2* the value 148, and in version *v3* the value 112b.

The information that one version is the predecessor of another version can be used to resolve to an earlier object version when no current version is available. This allows creating new versions of objects only when necessary.

The version identifier needs to be accessible to the version-aware references. It could be available globally, to have a single active version of the system, but could also be scoped more locally such as thread-local or in the dynamic scope of a code block. It should, however, not be changed while multiple version-aware references of an object graph are resolved transitively. Consequently, the version-aware references involved in evaluating the previous example statement should be resolved together for the same version identifier.

To be able to actually re-establish a particular version of the system with our approach, two requirements need to be fulfilled: First, all mutable objects of the programming runtime need to be accessed via version-aware references. Second, the particular version of the system needs to be available. Our approach does not allow re-establishing every state but specific states that have to been preserved. Programmers could preserve versions explicitly or the programming system could do so implicitly. When the programming system automatically preserves versions, each programmer action could implicitly yield a new version of the system. This way, programmers could undo and redo the changes of their actions regardless of whether or not they preserved a version in anticipation of recovery needs.

Discussion

The presented approach is incremental, not a stop-the-world approach. The version-aware references allow to preserve and re-establish versions of the system without completely halting the program execution. First, the version-aware references resolve dynamically to particular versions based on context information. Only this context information has to be changed to have all references resolve to another version. The version-aware references do not have to be re-configured individually.

Second, versions of the system are preserved incrementally. Instead of saving the state of all objects the moment a version is preserved, new versions of objects are created only when objects change. Before such writes, previous object versions continue to reflect the current state and can be read until they are changed.

4.2 Using Proxies as Version-aware References

We used proxies to implement version-aware references in JavaScript. Instead of actually requiring *alternative references*, proxies are referred to by *ordinary references* and transparently delegate to versions. This way, proxies allow a language-level implementation of version-aware references that works with existing JavaScript engines.

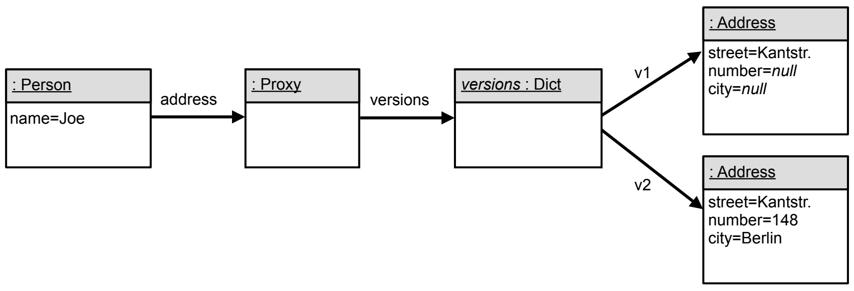


Fig. 4.7. Using a proxy as version-aware reference to connect a person object to two versions of an address object.

Proxies as Version-aware References

Figure 4.7 exemplifies how a proxy implements a version-aware reference in our solution. The proxy connects a person object to the two versions of its address property. The person holds an ordinary reference to the proxy in its address slot. The proxy in turn knows which versions are available for the address object.

When the `address` property of the `person` object is accessed, the proxy forwards the access transparently to a version. For example, in version *v2* of the system as indicated in Figure 4.7, even if the `address` property is a proxy, reading the proxy's city property returns the string Berlin. Given `aPerson` refers to the `person` object, evaluating the following statement returns true in version *v2*:

```
aPerson.address.city === Berlin
```

The statement does not include any version information. In particular, it does not read a specific version from a table of available versions. Instead, the proxies intercept all object interactions and forward to specific versions transparently. They fulfill three responsibilities:

1. They know which versions are available for a particular object.
2. They choose a particular version among all available dynamically using context information.
3. They forward all interactions transparently to a chosen version.

The proxies in this design are virtual objects [35]. They do not stand in for specific objects, but can forward intercepted interactions to any object.

Using Proxies Consistently

The proxies need to be used consistently for all mutable state. Ordinary references that usually refer to an object need to refer to the proxy that stands in for the object.

To use proxies consistently, we create and return proxies for all new objects. All expressions that create new objects return proxies for those objects instead. This is achieved by transforming code before it is executed. The source transformations wrap object literals and constructor functions into proxies. The proxies also always return proxies as return values. Thus, when proxied constructors are used, the constructors return proxies for the new objects.

The reference to the initial version of an object is only available to the proxy. The reference to the proxy is passed around instead. For this reason, all references that would usually point to the same object point to the same proxy. This way, proxies provide object identity. Checks that would usually compare an object to another objects now compare a proxy to another proxy.

As only the proxies hold references to the versions of objects, the versions get garbage collected with the proxies when the proxies are no longer reachable. For example, in the code of Listing 4.1, there would temporarily exist a version-aware reference—a proxy— connecting the person object to an address object, but the reference gets deleted before a version of the system is preserved. The address object is not required to re-establish either version *1* or version *2* of the system and nothing does prevent the garbage collector from reclaiming the proxy for the address object with the address object.

```
var person = {name: "Joe"};

\\ [preserve first version]
person.address = {street: "Kantstr.",
                  number: "148",
                  city: "Berlin"};

delete person.address;

\\ [preserve second version]
```

Listing 4.1: A newly created object is not preserved with any version.

Versions of the System

Proxies delegate to and create versions of an object using a *version of the system*. A version of the system is an object that has a *version identifier*, a *predecessor* version, and a *successor* version. Figure 4.8 shows three system versions. In the example, version *v2* is the current version of the system.

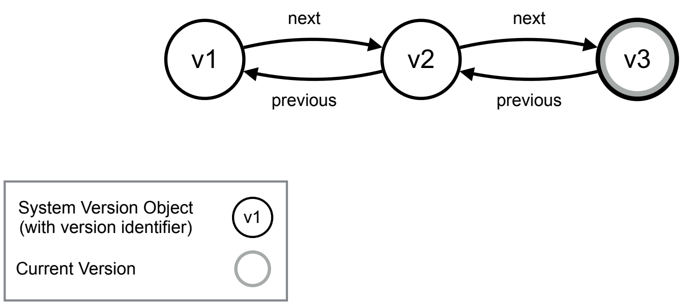


Fig. 4.8: Four versions of the system.

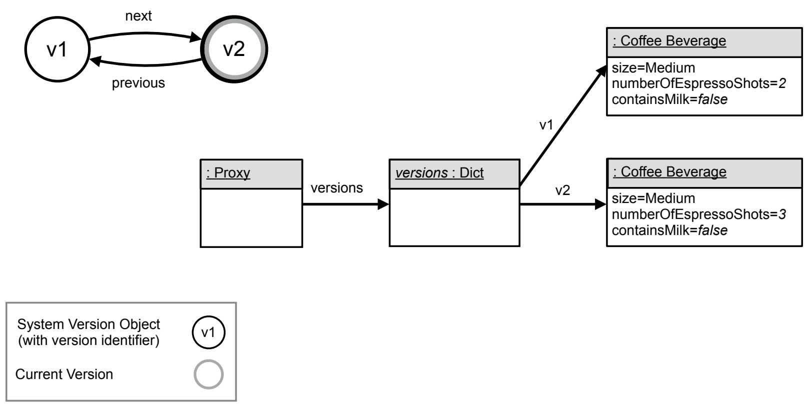


Fig. 4.9: A proxy with two object versions in context of the system versions.

The current version of the system is accessible to the proxies. Proxies use it to decide to which version of an object they currently should forward. Figure 4.9 shows a proxy, versions of an object, and versions of the system. In this example, there are two object versions that correspond to the two system versions. The current version of the system is *v2* and, therefore, version *v*2 of the object is the target that the proxy currently forwards to.

As long as the system version stays the same, the proxies forward to the same version of the object. Therefore, an object version is changed only as long as it matches the current system version. *To re-establish the previous version*, the system version has to be set to its predecessor. In that case, proxies forward interactions to previous versions of the objects. *To preserve the current version*, the system version has to be set to a different version. The proxies forward interactions to other object versions or, when no such version of the object exist, create new versions.

A situation in which a new version of an object is created is shown in Figure 4.10. In a new version *v3* of the system, the proxy intercepts a manipulation but has no object version it can forward to. It, therefore, copies the most recent version of the object and forwards to the copy.

New versions are only necessary when a proxy is about to delegate manipulations. As long as the state of an object is only read, the proxy reports values from a previous version as the old version of the object still reflects the current state. To create a new version, a proxy copies the most recent previous version of the object.

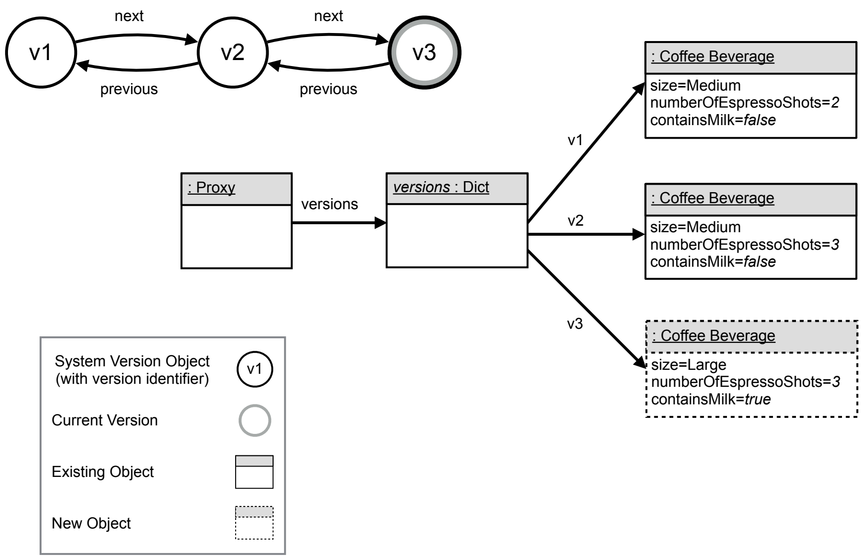


Fig. 4.10. A new version of an object is created for a new version of the system.

Limitations

The current design allows to preserve and re-establish versions of the system. Without further components, however, these versions only exist in memory and are not stored to disk. Our current design does not support multiple predecessors or successors. Another limitation of the current design is that the state of previous versions can be changed. New versions of objects are not affected by changes to previous versions, but changes to object versions that have not been copied shine through in subsequent versions of the system.

In the future, the versioning might allow for branches and merging. Changes to previous states could then be handled in branches that programmers may or may not merge into future versions.

5 Related Work

This sections presents two categories of related work:

1. Approaches related to our motivation and, thus, to providing access to previous versions of the system state.

2. Approaches related to our technical solution and, thus, to combining changes into first-class objects that can be used to scope changes.

7.1 Recovering Previous System States

The approaches presented in this section support programmers in recovering previous states without requiring programmers to create snapshots in advance.

CoExist

CoExist [28] provides recovery support through continuous versioning in Squeak/Smalltalk. For each change made to source code, CoExist creates a new version of the system sources, resulting in a fine-grained history of changes. CoExist presents this his- tory in a timeline tool and a dedicated browser. For each version, those tools show the changes, test results, and a screenshot. Developers can recover previous development states, even without taking precautionary actions beforehand. This way, developers can concentrate on implementing their ideas and let CoExist record the required versions to be able to recover when necessary.

Both CoExist and our approach to object versioning allow multiple versions of the development state to coexist. With both approaches, preserved versions are part of the program runtime and can be re-established easily. Currently only CoExist records versions continuously on the granularity of changes made by developers. CoExist provides much more tool support to find and recover changes from previous versions. However, CoExist recognizes only changes to the source code of classes, while our system preserves the state and behavior of objects.

Back-in-Time Debugging

Back-in-time Debuggers [16], also known as *Omniscient Debuggers*, allow developers to inspect previous program states and step backwards in the control flow to undo the side effects of statements. Approaches for this are either based on logging or replay: either the debugger records information to be able to recreate particular previous situations, requiring mainly space for the different states, or the debugger re-executes the program up to a particular previous situation, requiring mainly time to re-run the program. While many logging-based approaches introduce significant execution overheads, replay-based approaches have to ensure that the program is re-executed deterministically, which can be a difficult problem when, for example, programs can rely on state outside of the program runtime such as the content of files or the state of other programs.

Our approach is more related to logging-based back-in-time debugging. It allows re- establishing a previous state through preserving information. However, back-in-time debuggers need to be able to undo the effects of each statement separately, while our system's versioning granularity is arbitrarily and can, for example, correspond to programmer interactions with the system. In general, back-in-time debuggers support a particular development task—debugging—and, thus, are also often only active when a program is started in a separate debugging mode. In contrast, the purpose of object versioning is more comprehensive. We expect object versioning to be active at least during all development tasks, but possibly even be enabled at all times.

Software Transactional Memory

Software Transactional Memory (stm) [27] captures changes to values in transactions, analogous to database transactions. Each transaction has its own view of the memory, which is unaffected by other concurrently running transactions. Multiple versions of the system state can coexist. Which version is read and written to depends on the transaction. Transactions contain a number of program statements that are executed atomically. The changes of a transaction are only permanent when no conflicts occur with other transactions. On conflicts, all changes from the transaction are rolled back and undone.

stm and our approach are similar in that multiple versions of the system state can coexist and that a previous state can be re-established if necessary. However, stm provides concurrency control and an alternative to lock-based synchronization, while our approach provides recovery support to developers when changes turn out inappropriate. stm transactions are automatically rolled back when changes conflict with changes from other concurrently running transactions, while our versions are offered to programmers to undo changes when necessary. Programmers can actively decide to undo changes when these, for example, negatively impact the functionality, design, or performance of programs. Our versions of the runtime are also first-class objects, which can be stored in variables and be re-established at any time, while transactions are always created implicitly through particular control structures and committed immediately upon success.

6.2 Dynamically Scoping First-class Groups of Changes

The approaches presented in this section allow combining changes into first-class objects and running code with particular sets of changes.

Worlds

Worlds provide a language construct for controlling the scope of side effects: changes to the state of objects are by default only effective in the *world* in which the changes occurred. These worlds are first-class values and can be used to execute statements with particular side effects being active. A new world can be spawned from an existing world, which establishes a child-parent relationship between the two worlds. Developers can commit changes from a child world to its parent world, thereby extending the scope of the captured side effects. The Worlds approach includes conditions that prevent commits that would potentially introduce inconsistencies.

In comparison, Worlds provides a language construct for experimenting with different states of the system, while object versioning allows to preserve versions of the system to recover previous states: Our approach does not include extensions to the host programming languages and no conditions for combining versions with their predecessor versions, but provides a basis for CoExist-like continuous versioning and recovery tools.

Other differences between Worlds and our approach regard the implementations. Our implementation in JavaScript does not prevent garbage collection as Worlds does. Further, both use different libraries for source transformations. Our source transformations are faster and do not use JavaScript exceptions.

Object Graph Versioning

Object Graph Versioning [26] allows programmers to preserve access to previous states of objects. Fields of objects can be marked as *selected* fields. When a snapshot is created, the values of these selected fields are preserved. Therefore, not every state can be re- established, but states that are part of global snapshots. The approach, thus, provides fine-grained control to programmers regarding which fields of which objects should be preserved when.

The technical solution is similar to our design. Analogous to our proxy-based version- aware references, selected fields do not refer directly to their actual values, but to chained arrays that manage multiple versions of the state of a field and delegate access to the current version transparently. The chained arrays decide which version to retrieve and when to create new versions using a global version identifier. In contrast to our solution, individual fields are versioned and only when programmers explicitly mark them as selected.

Object Graph Versioning aims to support implementing application-specific undo/redo or tools like back-in-time debuggers. In contrast, our approach to object versioning aims to support recovery of previous system states during the development of arbitrary applications.

Context-oriented Programming

Context-oriented Programming (COP) [10, 1] adds dedicated language constructs for dynamic behavior variations. Depending on context information, cop allows enabling and disabling layers, which contain methods to be executed instead or around methods of the base programs. Context information can be any information that is computationally accessible. Layers can be enabled and disabled at runtime. Different implementations of cop provide different mechanisms to scope the activation of layers: for example, layers can be activated explicitly for a particular scope or globally for the entire runtime. ContextJS [20] it is possible to activate layers for specific objects.

In comparison to our approach, COP allows to activate combinations of layers, while our system executes code using a single active version. In cop layers are independent, while our versions are predecessors and successors of each other.

COP aims at supporting the separation of heterogeneous crosscutting concerns, while object versioning aims at supporting developers with the recovery of previous states. However, [21] showed that cop can also be used to experiment with changes to a system: developers can implement experimental changes to behavior in layers, not to modularize context-dependent adaptions, but to be able to scope changes dynamically and recover the original system behavior easily. However, this requires programmers to make experiments explicitly. They need to use layers for their adaptions, enable the layers for test runs, and move code from layers back to the base system when experiments are successes and they want to maintain the original modularization of the system. COP also allows only behavior variations, while our approach recognizes changes to both state and behavior.

ChangeBoxes

ChangeBoxes [5] is an approach to capturing and scoping changes to a system using first-class entities, called ChangeBoxes. A ChangeBox can contain changes to multiple elements of a software system such as adding a field, removing a method, or renaming a class. The approach does not constrain how changes get grouped into ChangeBoxes, but every change has to be encapsulated by a ChangeBox. Each ChangeBox can be used for setting the set of active changes for the scope of a running process. This way, multiple running processes can view the system differently by using different ChangeBoxes. ChangeBoxes can have ancestor relations and merge changes from multiple ancestors.

With the ancestor relations, ChangeBoxes can be used to review the evolution of systems and to undo changes.

The ChangeBoxes approach is similar to our approach as changes to the system are grouped into first-class objects and these can be used to run code in different versions of the runtime. Furthermore, with both solutions there is no definite notion in how changes are grouped into versions. Our object versioning approach is intended to be used to group changes associated with developer actions and a simple global undo/redo mechanism to undo inappropriate actions is built into our solution. To actually undo changes Change- Boxes, in contrast, is rather tedious [28]. Moreover, ChangeBoxes recognizes only changes to the static elements of a software system such as packages, the structure of classes, and methods. Object versioning, in contrast, preserves the state and behavior of objects.

Practical Object-oriented Back-in-Time Debugging

Practical Object-oriented Back-in-Time Debugging [19, 18] is a logging-based approach to back-in-time debugging that uses alternative references to preserve the history of objects. These alternative references, called *Aliases*, are actually objects and part of the application memory. These objects contain information about the history and origin of the values stored in fields. Aliases are not passed around, but instead are created for each read or write of a field and for each value passed as parameter. Each alias refers to an actual value, but also to another alias—its *predecessor*—representing the value previously stored by a field and to the alias that was used to create this new alias from—its *origin*. An alias and its origin both refer to the same value, but provide different information on their creation context, which is a particular method. The origin link can be used to follow the object's "flow" through the program. Each alias also records a timestamp on its creation and with this information the predecessor link can be followed to read a value as it was at a particular moment in time.

In comparison, with aliases it is possible to recreate all previous system state and also to retrace the flow of all values, while our system stores only particular versions. Such versions could, for example, correspond to programmer interactions, so that programmers can undo the effects of particular actions easily. Another difference between object versioning with version-aware references and reverse engineering with aliases is the existence of modes. The alias references are intended to be used in explicit debugging sessions, while our version-aware reference are intended to be used at all times.

6 Summary

This work introduced an approach to preserving access to previous states of programming systems such as the Lively Kernel. The approach is based on version-aware references. These references manage different versions of objects transparently. They resolve to one of multiple versions of objects; to which ones in particular can easily be changed. Thereby, different preserved states can be re-established.

We presented a design for our approach that uses proxies for version-aware references. Instead of actually using alternative references, ordinary references refer to proxies and proxies forward all interactions transparently to the right versions. The design allows implementing version-aware references without any adaptions to existing execution engines— neither for alternative references nor for the garbage collection of versions.

For each object that is created, a proxy is created and returned instead of the object. Thus, references to proxies are passed around and all access goes through the proxies. Moreover, only the proxies refer to the versions of an object. Consequently, the versions of an object are reclaimed together with their proxy by the ordinary garbage collector. Returning proxies for new objects is achieved using source transformations. The program sources are transformed when loaded and do not have to be adapted manually.

References

1. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. "A Comparison of Context-oriented Programming Languages". In: *International Workshop on Context-Oriented Programming*. COP '09. ACM, July 2009, 6:1–6:6.
2. Alan Borning. "Classes Versus Prototypes in Object-oriented Languages". In: *Proceedings of 1986 ACM Fall Joint Computer Conference*. ACM '86. IEEE, Nov. 1986, pp. 36–40.
3. Tom Cutsem and Mark S. Miller. "Trustworthy Proxies: Virtualizing Objects with Invariants". In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP '13. Springer, July 2013, pp. 154–178.
4. Martin A. Czuchra. "One Worlds: Automated Client–Side Persistence in Lively Kernel". Master Thesis. Hasso-Plattner-Institute, University of Potsdam, Jan. 2012.
5. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. "Encapsulating and Exploiting Change with Changeboxes". In: *Proceedings of the 2007 International Conference on Dynamic Languages*. ICDL '07. ACM, Aug. 2007, pp. 25–49.
6. Ecma/TS39. *ECMAScript Language Specification (Draft for 6th Edition)*. Published April 27, 2014 (Draft, Revision 24). Available at http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts. Accessed May 11, 2014. 2014.
7. Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. "An Incremental Constraint Solver". In: *Commununications of the ACM* 33.1 (Jan. 1990), pp. 54– 63.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Jan. 1995.
9. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Jan. 1983.
10. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (Mar. 2008), pp. 125–151.
11. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '97. ACM, Oct. 1997, pp. 318–326.
12. Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. "Fabrik: A Visual Programming Environment". In: *Conference Proceedings on Object- oriented Programming Systems, Languages and Applications*. OOPSLA '88. ACM, Jan. 1988, pp. 176–190.

13. Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The Lively Kernel–A Self-supporting System on a Web Page". In: *Self-Sustaining Systems*. S3. Springer, May 2008, pp. 31–50.
14. Alan Kay. *Squeak Etoys Authoring and Media*. Tech. rep. Published February 2005. Available at http://www.vpri.org/pdf/rn2005002_authoring.pdf. Accessed March 7, 2014. Feb. 2005.
15. Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. "Lively Wiki: A Development Environment for Creating and Sharing Active Web Content". In: *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*. WikiSym '09. ACM, Oct. 2009, 9:1–9:10.
16. Bill Lewis. "Debugging Backwards in Time". In: *Proceedings of the Fifth International Workshop on Automated Debugging*. AADEBUG'03. Springer, Sept. 2003, pp. 225–235.
17. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPLSA '86. ACM, June 1986, pp. 214–223.
18. Adrian Lienhard. "Dynamic Object Flow Analysis". PhD thesis. University of Bern, Dec. 2008.
19. Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming*. ECOOP '08. Springer, July 2008, pp. 592–615.
20. Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. "An Open Implementation for Context-oriented Layer Composition in ContextJS". In: *Science of Computer Programming* 76.12 (Dec. 2011), pp. 1194–1209.
21. Jens Lincke and Robert Hirschfeld. "Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming". In: *Proceedings of the International Workshop on Context-Oriented Programming*. COP '12. ACM, June 2012, 2:1–2:6.
22. Jens Lincke and Robert Hirschfeld. "User-evolvable Tools in the Web". In: *Proceedings of the 9th International Symposium on Open Collaboration*. WikiSym '13. ACM, Aug. 2013, 19:1–19:8.
23. Jens Lincke, Robert Krahn, Dan Ingalls, Marko Röder, and Robert Hirschfeld. "The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content". In: *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*. HICSS '12. IEEE, Jan. 2012, pp. 693–701.
24. John H. Maloney and Randall B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. ACM, Dec. 1995, pp. 21–28.
25. John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment". In: *Transactions on Computing Education* 10.4 (Nov. 2010), 16:1–16:15.
26. Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. "Executing Code in the Past: Ecient In-memory Object Graph Versioning". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. ACM, Oct. 2009, pp. 391–408.
27. Nir Shavit and Dan Touitou. "Software Transactional Memory". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. ACM, June 1995, pp. 204–213.
28. Bastian Steinert, Damien Cassou, and Robert Hirschfeld. "CoExist: Overcoming Aversion to Change". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. ACM, Jan. 2012, pp. 107–118.
29. Bastian Steinert and Robert Hirschfeld. "How to Compare Performance in Program Design Activities: Towards an Empirical Evaluation of CoExist". In: *Design Thinking Research*. Understanding Innovation. Springer, Jan. 2014, pp. 219–238.

30. Antero Taivalsaari. “Classes vs. Prototypes-Some Philosophical and Historical Observations”. In: *Journal of Object-Oriented Programming* 10.7 (Apr. 1996), pp. 44– 50.
31. Antero Taivalsaari. “Delegation Versus Concatenation or Cloning is Inheritance Too”. In: *SIGPLAN OOPS Messenger* 6.3 (July 1995), pp. 20–49. issn: 1055-6400.
32. Antero Taivalsaari. *Kevo, a Prototype-based Object-oriented Language Based on Concatenation and Modules Operations*. Tech. rep. Technical Report LACIR 92-02, University of Victoria, 1992.
33. David Ungar and Randall B. Smith. “Self”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, June 2007, 9:1–9:50.
34. David Ungar and Randall B. Smith. “Self: The Power of Simplicity”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA ’87. ACM, Dec. 1987, pp. 227–242.
35. Tom Van Cutsem and Mark S. Miller. “Proxies: Design Principles for Robust Object- oriented Intercession APIs”. In: *SIGPLAN Notices* 45.12 (Oct. 2010), pp. 59–72.
36. Alessandro Warth. “Experimenting with Programming Languages”. PhD thesis. University of California, Los Angeles, Dec. 2009.
37. Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. “Worlds: Controlling the Scope of Side Eects”. In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP’11. Springer, July 2011, pp. 179–203.
38. Alessandro Warth and Ian Piumarta. “OMeta: An Object-oriented Language for Pattern Matching”. In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS ’07. ACM, Oct. 2007, pp. 11–19.