Evolving Tools in a Collaborative Self-supporting Development Environment

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Doktor-Ingenieur)
vorgelegt
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.-Ing. Jens Lincke

betreut von
Prof. Dr. Robert Hirschfeld
Fachgebiet Software-Architekturen
Hasso-Plattner-Institut für Softwaresystemtechnik
Universität Potsdam

September 2014

Abstract

Adapting and developing software tools, to fit them to the task at hand while being used, is the domain of end-user development (EUD). Unlike EUD, the programming tools of a general-purpose development environment cannot usually be adapted in a live and interactive way. However, there are exceptions. The combined development and run-time environments of languages, such as Lisp and Smalltalk, allow modifying applications, but also adapting the development environments themselves at run-time. This *self-supporting development* approach shortens feedback loops and makes the development style interactive and explorative. *Lively Kernel* is a Smalltalk-like self-supporting development environment in the Web-browser, which allows Web-applications to be developed in such a lively way. Since the environment is shared, the feedback cycles among collaborating users are also shortened. Therefore, a Lively Kernel-based wiki allows the development, sharing, and reuse of each other's creations in an environment which evolves while being used.

Because of computational reflection and meta-circularity, developing in a self-supporting environment inherently has the danger of breaking one's own tools, which is more severe in a shared environment since changes can and often do affect more than a single user. Therefore, developers should have the means to change their system at run-time, to try out changes, and share new features or tool adaptations with others in a controlled way.

To address this, we developed two approaches: first, we propose to develop tools as run-time-modifiable *parts* that can be cloned to safely change them. Adapted tools can then be shared via a *parts bin*, so that they can be collaboratively developed. Second, we propose to modularize changes to the base system via *layers* that can be scoped, depending on the execution context, to make the development of the base system safer at

run-time by reducing the risk of tools breaking themselves. Furthermore, layers can be used to share such changes in a wiki-like collaboration setting.

We implemented our approach in *Lively Webwerkstatt*, a collaborative, self-supporting development environment based on Lively Kernel. Its programming tools are developed as malleable *parts* that are shared via a *parts bin*. *Deep cloning* makes evolving tools safe and also allows alternative ideas to be tried out side by side in the running system. *Development layers* are implemented using *ContextJS*, our context-oriented JavaScript language extension, which supports domain-specific scoping strategies, that allow restraining behavioral adaptations not only to the dynamic extent of an execution, but also structurally to object composition hierarchies.

We present example artifacts and analyze the repository data of Webwerkstatt, in order to discuss and evaluate our approach. Webwerkstatt has been actively used for more than three years now—not only by a small group of core developers, but also by external users, including our students, for whom it has served as a shared development environment. During this time, its users successfully worked on their projects, adapted tools, and also helped to evolve Lively Webwerkstatt itself.

Acknowledgements

First and foremost, I want to thank Robert Hirschfeld for convincing me to pursue a PhD! He continuously provided the right balance between pushing, supporting, and giving me the freedom to explore, so that I had the chance to learn so many things on so many levels.

I also want to thank my colleagues and friends Malte Appeltauer, Robert Krahn, Tim Felgentreff, Marcel Taeumel, Bert Freudenberg, Bastian Steinert, Carl Friedrich Bolz, Michael Perscheid, and Tobias Pape. Working with them was always inspiring, fruitful, and fun and made the last years here at the Hasso-Plattner-Institute fly bye too fast.

I am thankful to Dan Ingalls for introducing Robert Krahn and me to Lively Kernel and providing us with a seemingly never-ending source of inspiration that lit our fires.

I also want to thank our graduate students, who helped evolve Lively Webwerkstatt: Julius Dannert, Marko Roeder, Lauritz Thamsen, Fabio Bornhofen, Christopher Schuster, and Astrid Thomschke.

I want to thank my friends Philipp Engelhard, Stephanie Platz, Anna Filippova, Jenny Rackwitz, Patricia Gerasch, and Philipp Tessenow for tolerating my continuous explanations of various aspects of my thesis while still giving me valuable feedback from so many perspectives.

I thank my family for their continuous support and encouragement even though there where other things that occupied our minds over the last year.

I would also like to acknowledge the financial support of the Hasso Plattner Design Thinking Research Program for our project.

Contents

List of Figures

List of Technical Terms

AOP Aspect-oriented Programming

COP Context-oriented Programming

HTML Hypertext Markup Language

IDE Integrated Development Environment

JSON JavaScript Object Notation

Layer Modularity construct that can be dynamically scoped

Morph Graphical object in Morphic framework

OOP Object-oriented Programming

Part Published morphic object structure that can be (re-)used across worlds

Parts Bin Shared repository of parts

Prototype Object that serves as a template for the creation of other objects

REPL Read-Eval-Print Loop

Run-time Adaptation Changing of behavior during execution

SSDE Self-supporting Development Environment

Submorph Child of a morph in a graphical object composition

TDD Test-driven Development

Tool Interactive object used to modify other objects

URL Uniform Resource Locator

UUID Universally Unique Identifier

Web-based Software that runs on Web-infrastructure

Wiki Web-based collaborative authoring environment that permits all users to read, edit, and create Web-content

World Root of all morphs in the display hierarchy of graphical objects

Part I

Motivation

1 Introduction

The Web has become an important platform for a wide spectrum of applications. However, the workflow for creating those applications usually involves long edit-and-reload cycles. One way to shorten the feedback cycle is to integrate the development and run-time environments, so that applications can be changed while they are used. Programming environments for dynamic languages, such as Lisp [116] and Smalltalk [35, 49], are developed in such a self-supporting way. *Self-supporting development environments* (SSDE) are a subclass of self-sustaining systems (S3) [44], which also include all kinds of systems, tools, or languages that somehow share the notion of being implemented in themselves.

SSDEs keep development tools and applications together, so that both can be changed from within the same environment while the applications and tools are being used (as shown in (Figure 1.1). In contrast, traditional development environments separate the tools from the applications that are built using those tools and, therefore, changing the tools requires a second environment, making it more complicated to adapt the tools while they are being used.

Continuously and interactively developing a running application has benefits, such as shortening the feedback loops. However, adapting a run-time environment from within itself also has some drawbacks. Rather like locking yourself out, in developing a system using the tools that system provides, one can easily reach a state that does not allow any subsequent adaptations. This, of course, can happen on purpose or by accident. In Smalltalk, this feature is sometimes used to build an application by stripping all the development tools before shipping the system image to customers [15]. So, being able to change every aspect of a system also brings with it the power to completely break the system itself.

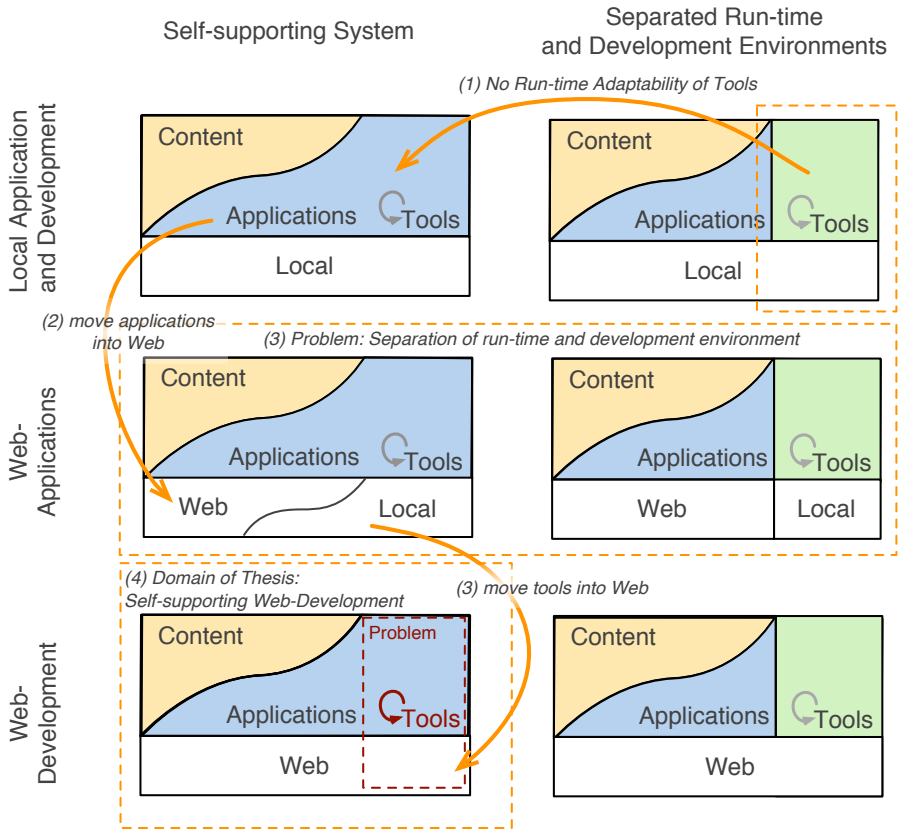


Figure 1.1: Domain of self-supporting Web-development

Once tools that are needed to perform changes are disabled or broken, such tools will not be useable until they are restored by external actions. Like some "Deus ex machina", another set of tools or external system is needed. Smalltalk, for example, provides an emergency evaluator, which is an interactive shell that allows the system to be repaired when the standard tools are broken [86].

This situation requires users to be careful when making changes to important parts of a self-supporting system. This problem is not a bad thing per se, as experienced users often prefer having simple, yet powerful tools such as Unix' command line tools, even though they are hard to learn [55, 111]. However, programming is already hard enough, without it being made even harder by having to think about how to perform a change safely. As we want to allow developers to stay in their self-supporting development environment, we have to provide means that allow developers to cope with this problem.

We want to support programmers in interactively evolving their tools and applying those changes in a safe way. The table in Figure 1.1 shows that developing Web-applications (second row) in self-supporting development environments (first column) is similar to a traditional setup (second column) where the tools are separated from the applications. When accidentally breaking the tools, no other users are affected and developers can fall back on other means, such as using another system to repair the original one.

In Web-based development environments (third row in Figure 1.1) the tools themselves run in the Web and are shared with others. The traditional setup (right column) separates the tools that are used to develop the applications. The good thing is that they cannot be broken this way, but the bad thing is that the system cannot be adapted or evolved while you are using it, either. In contrast, users can change the system from within when using self-supporting systems via the Web. Problematic is that users can break the system not only for themselves, but since the environment is shared with other users, it can be broken for others, too. This is a hard choice [19], since there are upsides and downsides for both alternatives. Similar to wiki-like open collaboration platforms, reverting changes to restore the functionality of a system is a general option to mitigate the problem in the left column of Figure 1.1. However, it forces you to step outside of your

working environment, while at that time the system might not be usable for other users.

The topic of this thesis is, how to provide users with the means to work more safely in a collaborative, specifically Web-based, self-supporting development environment without having to leave it.

1.1 Challenges in Evolving a Collaborative Self-Supporting Development Environment

In this thesis, we explore new ways of evolving a Web-based development environment in a self-supporting manner. Our motivation originates in our work on the Lively Kernel project [50]. The Lively Kernel is an environment based on the idea of combining Smalltalk-like development with a wiki-like way of collaboration [59]. The goal of the project is to be able to directly create active content and applications in the Web and allow others not only to use or play with your creations, but to explore them by deconstructing them and reusing parts for their own creative programming work.

Further, since Lively Kernel is a shared environment, developers are not only users of their programming tools but also the tools' potential developers. That is why, besides being a general programming environment, it incorporates elements of *end-user development* (EUD) [63]. In this way, while designing, implementing, and evolving the system, we were guided by the following general goals and challenges:

Run-time Adaptable Tools Developers can be regarded as (end-)users of their tools so, when adapting their environment, they perform end-user-like programming activities. This might seem strange, since developers should be, by definition, professional programmers. However, they usually have to work on domain-related tasks, instead of tooling related IDE adaptations. In order to make the environment developer-as-end-user-friendly, the tools should lend themselves to run-time customization. By making tools explorable and composable, users can, when they are dissatisfied with their tools, have a look at the inner workings, decompose them, reuse parts, and adapt them to better address their individual problems. "... it is by

fixing things that we often get to understand how they work" (Richard Sennet) [96]. So, it is only by being able to explore, repair, and adapt our tools that we are given full control over them and are able to use them in new and, from the point of view of the original developer, unanticipated ways.

Run-time Adaptations of Base System Depending on the programming technology used for creating a system, the means to adapt them may vary. For example, due to its origins in Smalltalk, the base system of Lively Kernel is built using a classical class-based approach. Since the base system is not built using graphical objects, cloning and direct manipulation of objects cannot be applied here. To safely work on core classes that provide base functionality in a self-supporting environment, we have to devise means that allow the system to be adapted from within, but in way that does not impair the development environment.

Direct but Controllable Sharing of Adaptations The benefits of being able to collaboratively evolve a shared environment from within itself balance the drawbacks of an environment that can be broken by everyone using it. To mitigate these drawbacks, we have to find means by which everyone can change the environment in a way that does not affect others.

Newly created tools and adaptations should co-exist and not necessarily pollute the base system. The new tools and adaptations should be available for all users to draw ideas from and reuse them—or parts of them—in their own applications.

1.2 Contributions

In this thesis, we propose two novel approaches for adapting tools in a collaborative, self-supporting development environment. The approaches address the problems at different abstraction levels (as shown in Figure 1.2): the cloning of *lively parts* at an end-user scripting level and the scoping of changes in context-dependent *development layers* at a system programmer's

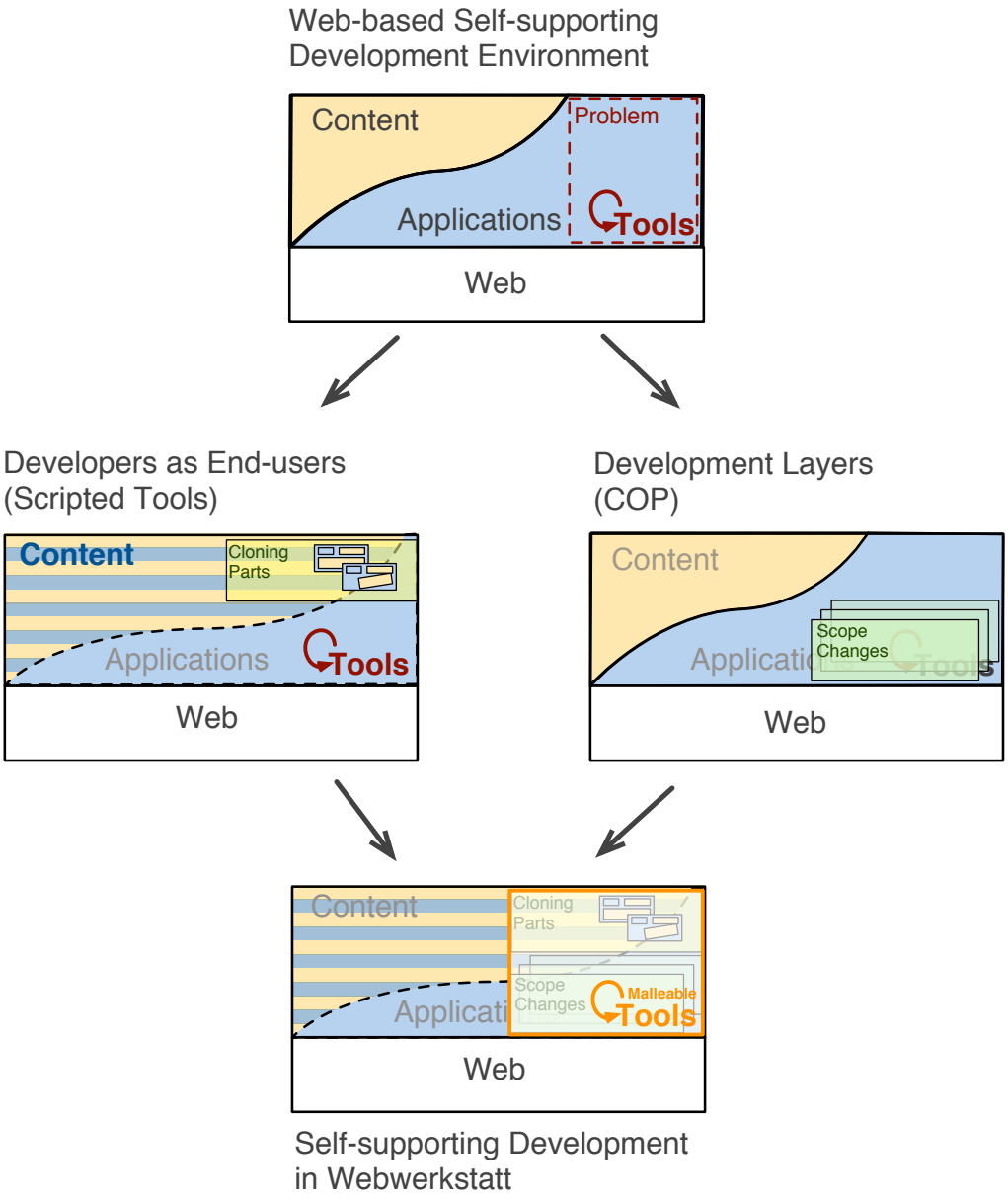


Figure 1.2: Overview of core problem and contributions

level. They were implemented and evaluated in Webwerkstatt, a wiki-like Web-development environment, based on the Lively Kernel.

1. Directly Modifiable Lively Parts Content, applications, and tools are built from the same, user-modifiable objects (*lively parts*). Based on this, tools can be cloned and their clones can be modified in a safe way. This means, for instance, that editing the editor tool does not yield the problems of meta-circular dependencies, caused by reflection. A general objects serialization mechanism allows both: to clone objects at run-time and to publish objects in a shared *parts bin* repository, therefore, allowing users to directly share their modified parts with others in the wiki.

2. Context-depended Development Layers Context-oriented programming (COP) provides dedicated support for defining and composing variations to basic program behavior. In addition to solely using COP to separate a system into dynamically composable features, we propose to express changes to the base system during development as layers. By using dynamic layer activation and composition, behavioral adaptations can be applied at run-time, without necessarily affecting and, therefore, potentially breaking the tools of the environment. Having modularized changes to the bases system into *development layers* allows a group users to experiment with new features, without changing the base system for all users.

3. ContextJS: an Open Implementation for Layer Composition We propose *instance- and structure-specific scoping strategies* to address the need to control the scope of adaptations dynamically and in domain-specific ways. An *open implementation for layer composition* allows the customization of such domain-specific adaptation rules. In the domain of user interface (UI) programming, for example, layers can be activated for a hierarchy of graphical elements.

4. Preserving the Derivation History of Objects When an object is cloned, the new clone and each cloned subobject are assigned a new unique identity, so they can coexist with the original object and its subobjects side

by side in the world. This renders comparing two objects difficult, because the identifiers of the objects involved cannot be used. By preserving the *derivation history* as a list of previous identities, we can leverage that information later when comparing objects. This is especially useful to automatically resolving conflicts when merging objects.

1.3 Outline

This thesis is organized into four parts:

- Part I introduces self-supporting development environments and outlines issues that arise when evolving a collaborative Web-based environment in a self-supporting way. Chapter 2 presents several object-oriented approaches for adapting tools in self-supporting development environments. It further introduces Lively Kernel as a collaborative Web-based authoring environment and its challenges in evolving the wiki-like system from within itself.

- Part II presents our two approaches for the safer adaptation of tools in a collaborative, specifically Web-based, self-supporting development environment, together with a novel approach of COP layer composition that allows for new domain-specific scoping strategies. Chapter 3 presents *Lively Webwerkstatt* and its two approaches, which can prevent the breaking of tools during their development. When tools are built and adapted using *lively parts*, they can safely be modified by direct manipulation after they have been deep cloned. Context-dependent *development layers* allow the adaptation of the base system in a scoped manner, so that tools do not break themselves. Both approaches make evolving tools safer at run-time and allow the adaptations to be shared with others. Chapter 4 presents the open implementation for scoping behavioral adaptations with *ContextJS*. New scoping mechanisms are needed to isolate tools from potentially dangerous reflective changes at run-time.

- Part III presents implementation details and evaluates and discusses the development of tools with lively parts and development layers,

based on examples and data gathered in Lively Webwerkstatt's development history. Chapter 5 discusses implementation details, such as the object serialization that serves for the similar sharing and cloning of parts likewise and it demonstrates how the derivation history is put to use when comparing and merging objects. Chapter 6 details case-studies of tool adaptations and their evolution in Webwerkstatt using parts and development layers and discusses some limitations, such as the overhead of storing meta-information and persistent garbage collection issues.

- Part IV discusses related and future work. Chapter 7 relates the results of this thesis to other self-supporting and collaborative programming environments. It discusses approaches to adapt tools and techniques for scoping dynamic behavioral adaptations. Chapter 8 provides an overview of future work and concludes the thesis.

2 Tool Adaptation in Collaborative Self-supporting Development Environments

A development environment is a software system that is used to create and evolve software. Tools in such an environment include source code editors, compilers, object explorers, and debuggers. When the tools are tuned to work together, such systems are also called integrated development environments (IDEs). Among development environments, *self-supporting* development environments are special, because they can be used to evolve themselves at run-time. Wikis are collaborative authoring environments that allow working asynchronously on shared content from within the Web-browser. Combining *self-supporting* development with *wiki-like* collaboration allows for an open software development approach, where the whole environment can be customized at run-time and collaboratively evolved.

2.1 Self-supporting Development

Creating and evolving tools while they are being used is common practice among programmers: writing shell scripts or extending text editors to optimize a programmer's workflow are the most common examples. Self-supporting development environments—such as Smalltalk [35], Self [122], Emacs [104], Common Lisp [106], Squeak [49], and Lively Kernel [50]—are systems where developers can evolve their environment while they are using it. Common to all of these environments is that they keep the software development tools—such as editors, debuggers, or code browser—in the

same environment as the objects and meta-objects they are working on. This allows for a direct development style with short feedback loops. A good motivation for such interactive programming styles is presented in *Inventing on Principle* [124]. It can be seen here how changes to the source code can be immediately observed in the running program.

2.1.1 Programming at Run-time

Although it has recently become very popular in the form of *live programming* [124, 78, 38], programming at run-time dates back to the first interactive computer usages. Being able to incrementally edit a running program dramatically shortened feedback loops. Before computers became interactive a typical programming workflow included handing over a stack of punch cards to operators and coming back the next day to get the results. Lisp programmers were the first, in 1961, who demoed live programming a computer [77], after being able to regularly use the expensive computers for themselves, at night, in a more explorative and playful way [61]. For some time, programming at run-time continued to be the preferred way of creating software for some schools of programming, such as the Smalltalk programmers who pioneered object-oriented programming (OOP) and personal computing in Xerox PARC [40, 53].

Both Lisp and Smalltalk programmers continued to prefer working in combined run-time and development environments. These environments are each deeply integrated with a dynamic programming language that allows working at very high levels of abstraction, but also directly deals with the concrete data and the behavior of running programs. Their programming approach is abstract, but also concrete and specific at the same time. They can describe their problems in abstract, often domain-specific ways, but at the same time interact with their running representation. As a result, the feedback loops and, therefore, the length of iterations can be kept very short. This is possible because it is easy to switch between using and developing an application or even to do both at the same time.

In this thesis, we use the attribute “self-supporting” to distinguish between development environments that can be used to adapt themselves at run-time and those that cannot. The extent to which a system can be

adapted or evolved may vary. Some environments provide the tools and mechanisms to completely evolve themselves. Other environments just allow the customization of specific parts at run-time.

Self-supporting development environments, such as Smalltalk, rely on reflection and self-modification capabilities of the underlying programming language. Smalltalk, for example, reifies meta-structures and concepts of the underlying programming language as objects. Classes are themselves objects and can be changed by tools or code at run-time. Maes describes this as a causal connection between the system and its meta-level [73]. The meta-level reifies its internal state as objects that are accessible to programs running in the system. Those reified objects can provide access to both static meta-structures, such as classes or methods, and dynamic meta-information, such as access to variables on the execution stack in processes. Since this connection between system and meta-level is bidirectional, changing the state of these objects will also affect the meta-level, e.g. replacing a method in a Smalltalk class object will result in an actual behavior change in all instances of that class.

2.1.2 Meta-circular Dependencies

Having such tight interaction between tools and code in one system can make the development of the core behavior difficult, as tools are changing the classes and functions that they are dependent on. Errors or debug statements in some core part of the system may break the whole development environment and force a restart. Developers can get used to such behavior, create workarounds, or become overly careful when changing core parts of the system.

Such self-supporting development environments often come in different flavors, depending on which kind of language they are built with and what the preferred way is for programming in that language. For example, as the comparison of Figure 2.1 and Figure 2.3 shows on an abstract level, the benefits and drawbacks are similar in file-based and object-centered development environments.

The workflow of tool adaptation in file-based environments like Emacs starts by loading the environment (as shown in Figure 2.1 1). The tools in

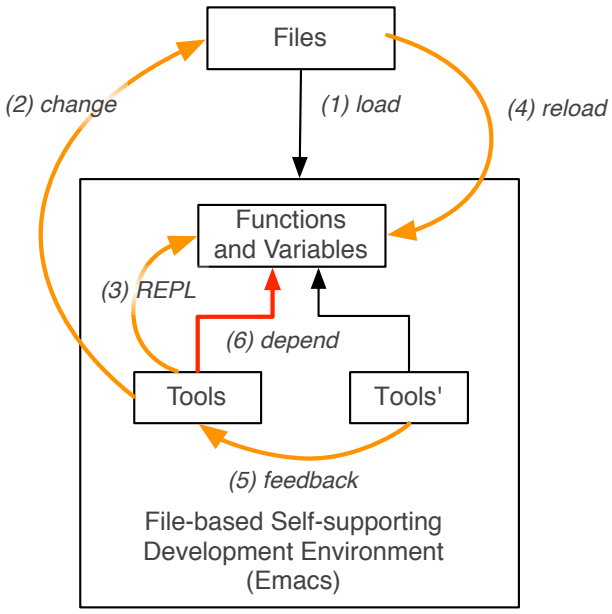


Figure 2.1: Workflow of adapting tools in file-based self-supporting development environments like Emacs.

the environment can now be used to change behavior, which, in the case of Emacs, is achieved by editing function definitions in files (2).

We call Emacs self-supporting, because it includes a *read-eval-print-loop* (REPL) which allows developers to directly apply changes to the running system itself by executing statements (3) or reloading entire files (2). Both approaches produce much quicker feedback (4) than reloading the entire system every time a change is made. However, having such a powerful development approach that allows for very quick iterations between making small changes (3) and getting feedback (5) also comes with a price. As the Figure 2.1 further shows, being able to change its own code introduces a meta-circular dependency (5) between the tools and their abstract representation. This dependency is inherently essential for the ability to evolve all parts of the system. However, not all changes are good and some of them may break the system in such a way that it cannot longer be fixed

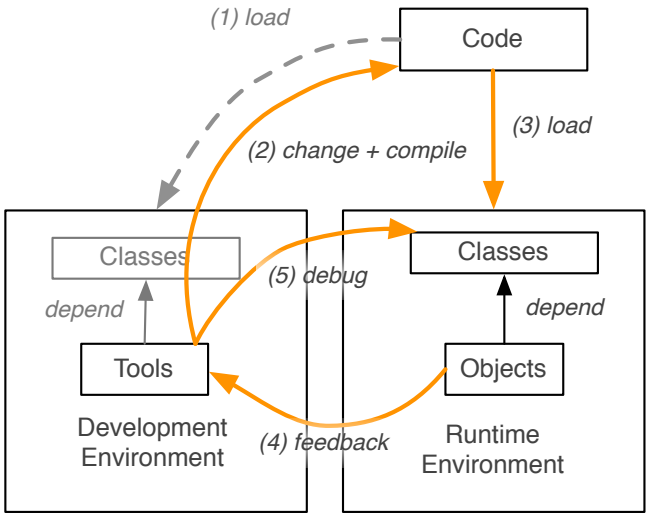


Figure 2.2: Separated tool environment

from within itself. So, a mechanism is needed to safeguard against such problems.

2.1.3 Separating Run-time from Developing Environment

A drastic solution for getting rid of such potential problems is to separate the run-time from the development environment. The more conservative approach is to reload the system under development after every change and this is the standard approach to development in many systems.

As it is difficult to make changes at run-time in statically typed languages such as Java, development environments like Eclipse [33] have to use just such an approach. Eclipse is a development environment for Java and it is written in Java. It is used to develop plugins to customize itself and create the next version of it. The difference from the approach in Emacs is that this is not a lightweight process. Making a quick customization is easy and fast in systems that can be changed at run-time, but it is much more indirect in systems like Eclipse.

As shown in Figure 2.2, the workflow of adapting tools starts by first loading the development system (1), performing the change by editing files

and, if necessary, compiling them (2), then loading the system a second time (3), to get some feedback (4). Since standard development tools, such as a debugger, allow the interaction with the run-time system to a certain degree (5), it is also possible to get immediate feedback while developing. The difference here is that the development-environment always comes first. It is not possible to adapt an environment while it is being used, so the context that motivated the change is still preserved. To achieve this, the development environment needs to be loadable on demand so that the running system can be changed when needed.

Another benefit of this edit-compile-run cycle is that developers are used to developing their systems in such a way, so they do not have to change their workflow when having to adapt their own tools. Fortunately, the performance of computers has increased so much that the actual time needed to compile a system is no longer such a big problem as it once was.

2.1.4 Automating the Feedback loop

Bringing the system into a state in which the new behavior can be tried out can still make the feedback loops much longer. One way to compensate this is to automate it using unit and acceptance tests, which allow getting feedback without any manual user interaction.

Getting such automated feedback is a valuable feature of Test-driven Development (TDD) [10]. TDD always starts by creating a program that tests for the desired behavior or absence of a bug. This has a lot of positive side effects, but it also raises the barrier of making the change. While programming at run-time, we might be able to see and fix the problem directly, having a context in which it is easy to come up with a solution. TDD is one way to get this kind of feedback and context in a reproducible fashion. The advocates of TDD argue that the additional work for writing the tests will pay off in the short term. The reasons are that it makes the development of the actual functionality easier and it keeps the system evolvable. Hence, developers can be sure that they did not accidentally break anything after successfully running all tests in the system.

When the programmer does not have control over the full system, recreating and testing some situations programmatically can become more taxing

than actually fixing the bug or implementing the desired behavior. This can also, sometimes, be much more difficult to achieve. In some cases it might also be unclear what the actual result should be. An example for this is fine tuning parameters or exploring possibilities in a prototype. On the other hand, if the test is not written first it might never be written at all and the code quality will deteriorate. So, ideally, programmers should be capable of knowing, when they use interactive programming to instantly try out new ideas, or when to do more rigorous system development. Ideally, a development environment should allow the ability to continuously go from one style of programming to the other, starting with trying out new ideas in an interactive fashion, but allowing them to evolve into the maintainable part of the system.

2.1.5 Object-centered Development Environments

Programming Environments are not always built around the idea of modifying text in source code files. Object-centered systems are an extreme case of run-time programming, since in this approach, changing behavior in the running system is the default and not an additional feature.

A workflow in a Smalltalk-like development environment is shown in Figure 2.3. At some point, the system has to be loaded (1). Traditionally, this simply consists of deserializing an object-space and then starting to run some processes. The system is then programmed by directly modifying classes (2). A behavior change is directly reflected in objects using the classes, though developers can get immediate feedback (3). Making these changes persistent by saving the complete state of the system, automatically preserving changes, or exporting some classes as source code (4) happens after the edit. Similar to Emacs-like environments from Figure 2.3, this workflow shows the benefits and the problem. The benefits are: being able to adapt tools at run-time and having short feedback loops between modifying some meta-structures (2) and observing effects in the running application (3). The problem that is introduced here is the danger of accidentally breaking the system while using it (5).

Unlike traditional file-based systems, programming in object-centered systems is not done by editing plain text files, but by directly manipulating

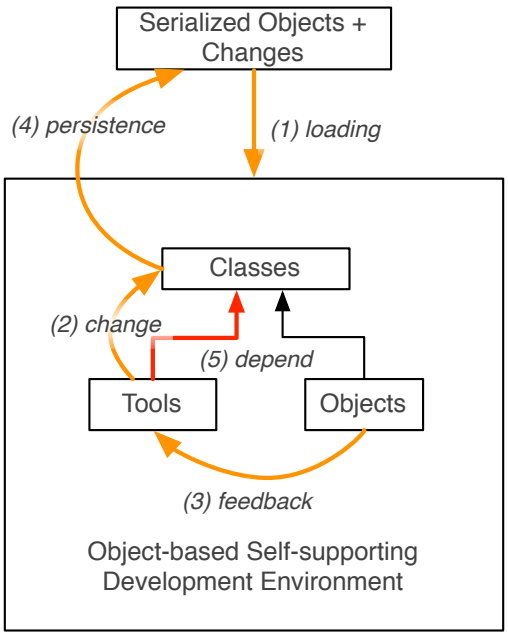


Figure 2.3: Abstract flow of changes and feedback in a self-supporting development environment

objects in the run-time environment. What those objects are and how they are used to create tools and applications depends on the actual system. Systems like the Smalltalk-80 [35], Self [122], Squeak [49], and Lively Kernel [48] are, from a certain perspective, very different systems. However, their approach of being able to directly inspect and change the state and the behavior of the running system and, with that, of being able to evolve it from within itself make them, in a sense, very similar.

2.2 Behavioral Adaptation in Object-centered Systems

Adapting tools or applications in a self-supporting environment, not only allows us to try to make sense of source code, but to actually work with live

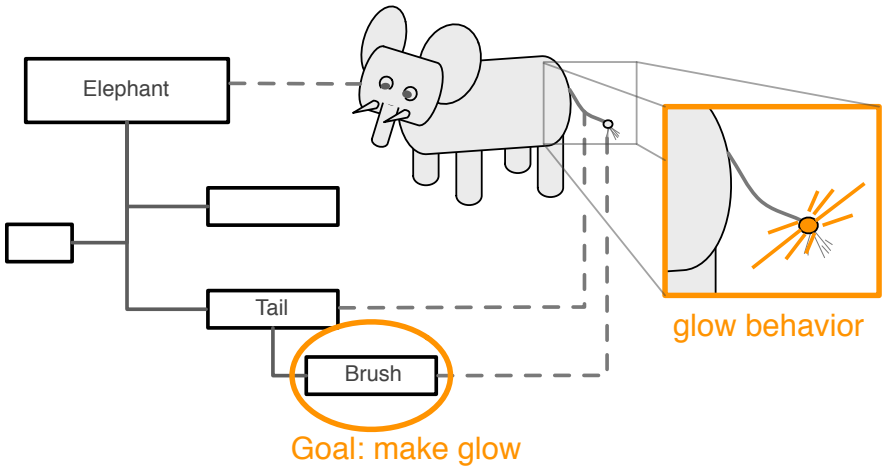


Figure 2.4: Example of a behavioral adaptation in an object composition: The motivation is to make the end tail of an elephant glow.

objects that form the application at run-time. However, how such objects are programmed depends on the language constructs and object technology used in the system. When interactively exploring the object structures, one can play around with setting properties or calling methods to see what they are doing, but going from that explorative stage to actually starting to refine the behavior is more difficult and depends on where the behavior is usually defined in the system and how such behavior can be changed, adapted, or reused.

To compare the various approaches, we use a common simple example of adapting an object in a composition hierarchy. Examples for such composition hierarchies are common in parse trees, business objects, or user interfaces. Developers in object-centered systems often work with such a composed objects. Since it is a composed object, it cannot always be directly adapted, but the actual behavioral adaptation may have to happen in a sub-object. Instead of using an actual example of the domains of user interface design or tool construction, we use an explicitly invented example. In this way, the distinction between domain and technology becomes clearer.

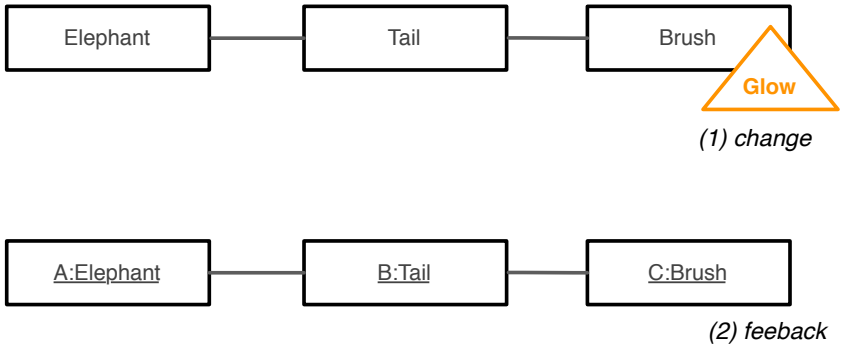


Figure 2.5: Behavioral adaptation in the elephant example: the most direct approach is simply to change the original implementation to glow.

Figure 2.4 illustrates this artificial example: the main object is an `Elephant` which has a `Tail` and the `Brush` of the tail should change its behavior so that it glows. How the actual glowing behavior is performed does not matter at the point, we assume here that it involves changing code in a method of `Brush`. The `Elephant` is composed of various subobjects, which, by themselves, are composed of subobjects. Interesting to the example is the relationship of the `Elephant`, its `Tail`, and the tail's `Brush`. Other subobjects, such as the feet, head, and ears are not important and have been omitted here.

2.2.1 Change Original Code

The simplest approach to make the tail glow is just by changing the original implementation, as shown in Figure 2.5. In run-time programmable systems, changing the abstract behavior in the class (1) is immediately reflected in the instance (2). As long as the instance somehow actually uses this behavior, the developer can get feedback while programming.

In self-supporting systems, this approach is problematic when the `Elephant`, or `Tail`, or `Brush` are an important part of the base system. In our example, the `Brush` was reused by the original `Elephant`'s creator. Besides

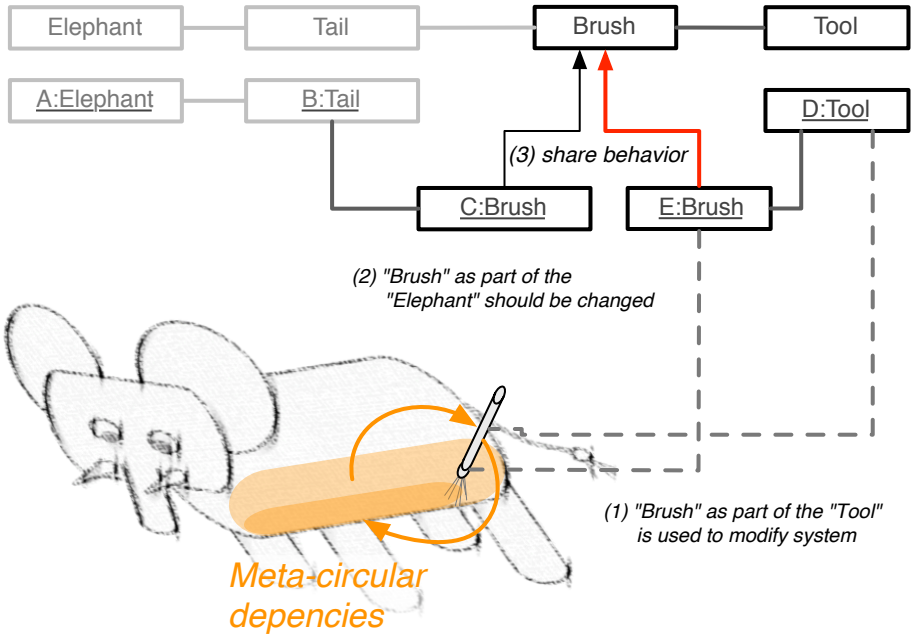


Figure 2.6: Problem of adapting the core behavior of a system. The `Brush` is used in the `Elephant` and in the `Tool`. So, when modifying it, meta-circular dependencies can become a problem. Such Escher-like meta-circular [46] loops are inherent in self-supporting systems.

being used as the tip of an `Elephant`'s tail, it is also part of a tool needed to paint objects, making the development environment dependent on it.

As shown in Figure 2.6, a potentially problematic meta-circular (reflective) dependency is introduced when the functioning of the environment and its tools depend on the meta objects, in this case, important classes, which the developer is working on. The `Brush` `C` of `Elephant` `A` is modified by `Brush` `E` of `Tool` `D`. It is a meta-circular dependency. The class `Brush` is modified and needed for modification at the same time. Since classes are meta-objects, their reflective meta-circular modification from within the system can break the system.

A similar problem occurs when one of these classes is reused by other applications or other parts of the same application which are unfamiliar

to the developer. Making the tail glow, then, will not only produce the expected glowing elephant, but it also makes the tails of other animals glow, something which was not intended by the developer. When, for example, the class `Brush` belongs not to the Elephant's application, but is reused itself, this problem is an instance of the *fragile base class problem* [79].

Introducing a new and potentially highly experimental feature, by directly changing the original code also has the downside that the original is no longer available. Such conflicts with the original version can be solved at various levels. In file-based systems, there is always the option of coping the original file, when files are directly executable, such as in scripting languages, and they are not part of a bigger framework, where their path or name is important. In theses cases, this is a viable option. However, this is often not the case. Having some form of version control system solves these issues. By allowing files to stay in their place and preserving the original versions, they can later be restored or used for computing what has been changed.

Both approaches address the problem outside the environment and, therefore, do not help in the domain of run-time programming, because, in that way, the original and the modified version cannot coexist side by side in the same environment.

The software engineering solution is to make the behavioral variation optional using various programming language constructs. The most basic approach is to use control structures such as *if-statements* at the right places. This would also require adding state somewhere so that the glowing can be enabled or disabled. This blows up the original code and scatters the new glow feature over many entities and places in the code, making it difficult to remove the feature afterwards.

2.2.2 Class-based Programming

Instead of adding the new glow features directly in the class `Brush`, developers learned that they can refine the original class by subclassing it. As shown in Figure 2.7 the new class `GlowBrush` inherits from `Brush` (1) and adds glowing by refining some exiting behavior (2). The classic object-oriented approach has the downside that subclassing is not enough in this

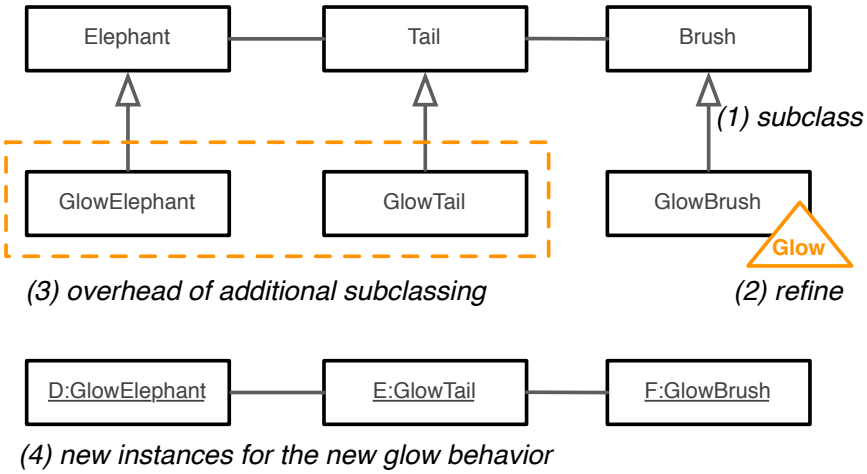


Figure 2.7: Behavioral adaptation in the elephant example, implemented with subclassing.

scenario. Due to the object composition, the elephant has, somehow, to use the new `GlowBrush` class. This can be achieved by additionally subclassing `Elephant` and `Tail` (3). The new glowing feature does not affect normal elephant objects, but only affects instances of `GlowElephant` (4).

Subclassing is more elegant than directly changing the code, because it separates the glow feature in a new entity, which can be easily dismissed in the case of the glowing elephant's tails will no longer being useful. The downside is that, in the case of adapting an object composition, subclassing one class is not enough but requires the subclassing of other classes or adapting them in other ways in order to manage the new dependencies, making it cumbersome to use.

2.2.3 Changing Objects at Run-time

When the behavioral adaptation is only needed while experimenting during development, instance-specific behavior might be an option. This means modifying just one elephant might be an option, if the developer wants to know how an elephant looks and behaves with a glowing tail. Some

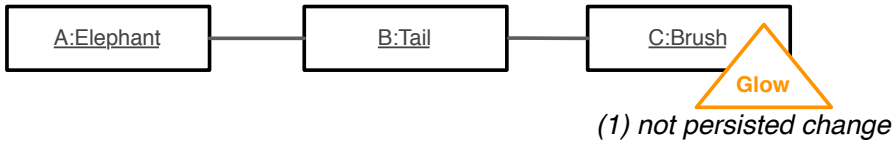


Figure 2.8: Behavioral adaptation in the elephant example with instance specific behavior

programming languages allow objects not only to have state, but instance-specific behavior. How this is implemented in different programming languages varies but, in principle, it allows us to add behavior directly to an object. Our running example in Figure 2.8 shows that the *glowing* behavior can now be added directly to the instance `C` of the class `Brush`. Such instance-specific behavior has to be created somewhere. Normally it is used to customize objects in the code where the objects are created, using some kind of meta-programming. In interactive systems developers can use this language feature for experimentation, by just adding the behavior using a REPL or other tools at run-time. Since in most systems behavior is stored separately from objects, e.g. source code files vs. data base, their instance-specific behavior cannot be persisted. Even though many programming languages support it instance-specific behavior is, therefore, often transient. With the exception of end-user programming approaches like Etoys [54, 2], usually, objects are not exchanged directly between developers or used as a means to package applications or behavioral adaptations.

2.2.4 Prototype-based Programming

Prototype-based programming [62], as implemented by Self [122], builds on the idea of programming objects directly and allowing objects to reuse the behavior of other objects through delegation. This is achieved by unifying Smalltalk's object and class dualism, so that state and behavior are treated equally in objects. In prototype-based programming, different from typical class-based programming, objects can also have state in addition to behavior. Some objects can serve as prototypes and share behavior and state among a group of objects. Each object can receive messages that are used to access

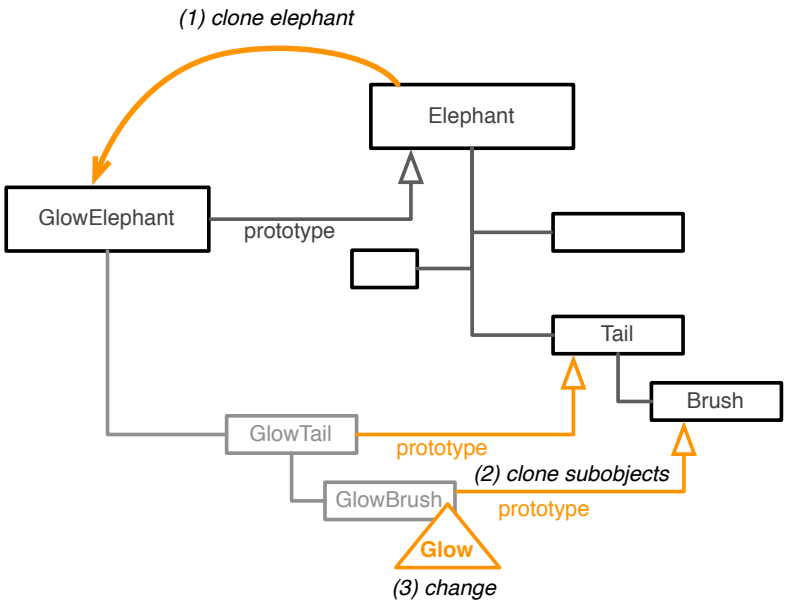


Figure 2.9: Behavioral adaptation in the elephant example, implemented with Self-like prototypes.

state or trigger behavior and can decide either to handle the messages by themselves or to delegate them to its prototypes. This feature makes Self's prototypical inheritances more powerful than Smalltalk's class-based approach. It further allows experimenting with different development styles and organizations of code. But programming in Self can still be very similar to Smalltalk's class-based programming, because prototypes can be used like classes [123].

The idea of prototype-based programming and its implementation as prototype-based inheritance by unified behavior and state lookup [122] are not the same, even though the latter can be used to achieve the first in many cases. However, it does not matter if shared behavior is represented as class or as prototype. When a prototype only serves as an abstract container for shared behavior, it cannot be applied to applications or tools that are not represented as a single object, but are composed of many subobjects. In Self, prototype-based inheritance cannot deal with complex object structures, they have to be separately copied or remain shared.

Figure 2.9 illustrates this problem by showing how the adaptation of `Elephant` can be done with prototypes. Theoretically, such adaptation of objects at run-time is one of the strengths of prototype-based programming as advertised in Self. The problem with Self-like prototypes is that object composition cannot be delegated. Nested object composition is not represented as messages that can be refined and delegated. So, as in the case of subclassing the classes `Tail` and `Brush` as shown in Figure 2.7, the objects `Tail` and `Brush` have to be copied (2) before they can be adapted (3). To make use of prototype-based inheritance, a copy can be made by creating an empty object that can delegate messages to the original object so that behavior can be reused.

2.2.5 Morphic: Directly Manipulatable User Interface at Run-time

Self also served as a platform for researching novel user interfaces. It introduced Morphic [76, 75], a new way for constructing user interfaces with directly manipulatable graphical objects. In Morphic, all graphical elements in the user interface are represented as *morphs*. Morphs are recursively composed of sub-morphs, react to events, and can draw themselves. Morphs are typically composed in trees, where a morph has one parent and can have many children. Morphs compute their position on the screen from their own *position*, and that of their parents and apply various styles, such as *fill color* and *border width*. What makes Morphic special is that the user interface is constructed and can be adapted at run-time.

The promise that prototype-based programming enables a philosophically different way of programming [100] cannot be held when it comes to working with morph hierarchies, because prototype-based inheritance in Self does not extend to object composition. Object composition is essential to constructing rich graphical applications that cannot be represented by a single graphical object, which only draws itself. For this primitive object case, the prototype metaphor holds. A typical example in Self is the simulation of a gas tank [123]. As shown in the screenshot in Figure 2.10, the gas tank contains many individual objects that move around and bounce when colliding. Figure 2.11 shows how these individual atoms are directly

Figure 2.10: Screenshot of Self's gas tank example, showing the running simulation (1), the behavior definition (2), and an exploration of the simulation state (3) side by side.

represented by objects that inherit from a common prototype. Changing the state or methods of the prototype immediately affects the individual objects, producing a very short feedback loop. As long as the appearance of objects is determined by properties, such as color or extent, and by a method that draws it, a developer can prototypically modify one object

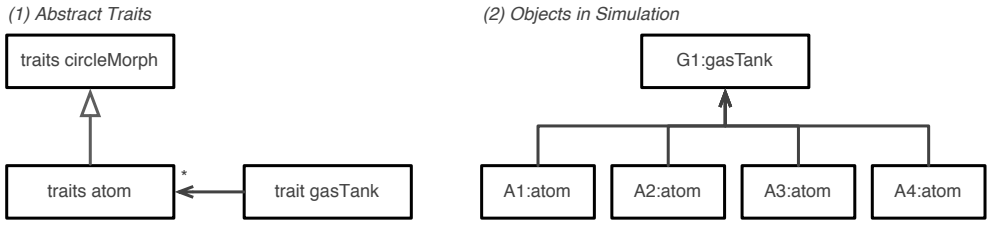


Figure 2.11: Abstract objects (1) and their instances (2) in Self's gas tank

and see the changes immediately applied in the other objects as well. The metaphor breaks down when the objects in the tank become more complex, e.g. by adding another graphical object as subobject to our atoms making them molecules. Then prototypical inheritance conflicts with the Morphic metaphor. In Morphic, all graphics on the screen are directly and bidirectionally represented by graphical objects (morphs). If the subobject were somehow prototypically inherited, it would be displayed several times on the screen and then conflict with the Morphic metaphor.

Rendering an object multiple times, is not a general problem. Scene graphs used for rendering do this all the time. One solution could be that the subobjects are immediately rendered multiple times on the screen. When graphics are procedurally produced, as in the drawing loop of computer games, the pixels on the screen have no knowledge of which drawing method produced them. This draw method typically displays the complete scene graph in a frame. But in Morphic, this approach breaks the one-to-one relationships with identifiable graphical objects on the screen and their corresponding morph objects. If only the graphical appearance were displayed, this would not be a problem. But since the graphical interface should also be interactive, it has to respond to events and propagate them to their parents. This is why the objects on the screen need to have just a single parent, forming a tree and not a graph, so that it is clear which object was actually clicked on. For this reason, the same graphical object should not be on the screen several times. This means that the graphical composition of objects in Morphic cannot be reused through prototypical inheritance. Subobjects have to be deeply copied and, in doing so, the

metaphor of working on a prototype where all changes are immediately propagated to inheriting objects is broken.

The graphical model of a Web-browser follows the same rules as Morphic here, since the graphics are represented through a composition of objects and not procedurally generated by calling a draw function in each frame. Therefore, in a Web-based system, we cannot avoid the object composition problem by falling back on frame-based drawing.

2.3 Collaborative Self-supporting Development

Since collaborative Web-applications are inherently distributed to some degree, they cannot be built in a simple self-supporting way as presented in Section 2.1. Web-applications can be developed in object-oriented self-supporting development environments, such as Smalltalk, but the client side code running in the Web-browser cannot be explored or adapted at run-time. Since client side parts run in a separate environment, the benefits of simple run-time programmability are lost. As we have seen in Figure 1.1, in standard Web-development (second row) the tools run locally in the self-supporting system, as in the traditional setup. So, both approaches only indirectly allow working on the behavior of the Web-applications that runs on the client's side. The typical edit-and-reload cycle of a Web-application is not changed if the JavaScript parts on the client's side are delivered by a traditional system or a Smalltalk-based system.

Moving the development environment completely into the Web does not solve this problem at all, since a typical Web-application still runs half on the server and half on the client. What happened is that the problem flipped. Now the client's side can be developed more directly, but for developing the server-side part, the client has to indirectly reach into the server-side. A solution that solves this problem is to get rid of the application server part and build the whole application in the client. When the development environment is capable of creating such client-side applications, it can be self-supporting.

2.3.1 Lively Kernel—A Collaborative Self-supporting Development Environment in the Web-browser

A development environment that follows this idea is *Lively Kernel* [48], a self-supporting, Web-based, and combined run-time and development environment. Lively Kernel (sometimes abbreviated as Lively) recreates a Smalltalk-like development environment in the Web-browser. With Lively, interactive Web-content can be directly created in the Web-browser and Web-applications can be changed while they are being used. Lively Kernel is built using Web technology and it derives from a Smalltalk background. It, therefore, combines object-centered development and run-time programming with rich media and the collaboration possibilities of the Web. Lively serves as a platform to experiment with novel programming language concepts and tool development approaches.

Unlike Smalltalk, it provides hybrid file-, class-based, and object-centered development. Figure 2.12 shows both Lively Kernels development workflows: a) changing the base system by developing classes and b) using and editing objects in worlds.

Similar to other Web-based applications, the Lively Kernel environment is loaded by visiting special pages. Theses pages are called *worlds*, contain user content and applications, and are stored in *HTML* files. The object serialization is discussed in Section 5.2.

In Lively Kernel, individual Web-pages, as shown in Figure 2.13, contain each one *world* which, as the root object, contains all other objects. All visible graphical objects (*morphs*) are direct or indirect children (*submorphs*) of the world. Non-graphical objects can belong to this world, by being directly or indirectly referenced by a graphical object. Objects that are not referenced are not persisted.

Lively Kernel bootstraps itself every time a world (Lively Kernel page) is loaded as shown in Figure 2.12 (1). The base system is represented as file-based modules that contain plain JavaScript source code. An example Lively Kernel module is presented in the following listing [1]:

[1] For better readability module prefixes are omitted.

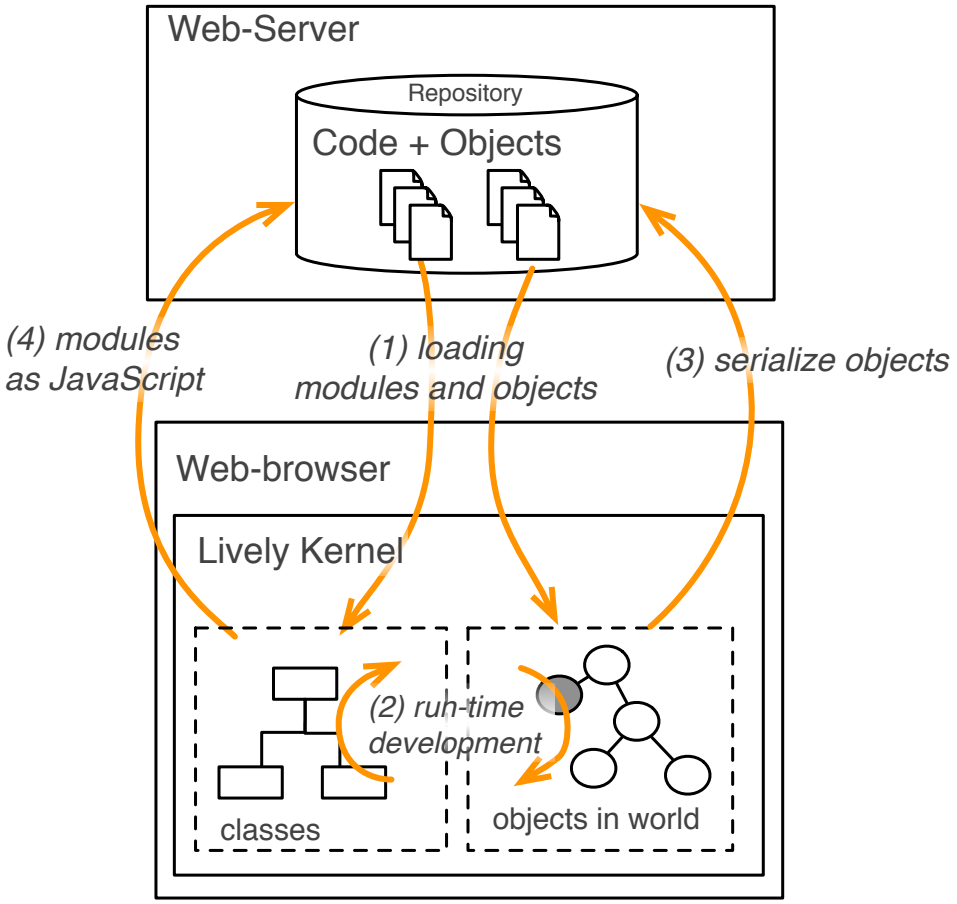


Figure 2.12: Web-based self-supporting development in Lively Kernel

```
module('animals').requires('morphic').toRun(function() {
  BoxMorph.subclass("ElephantMorph", {
      initialize: function($super) {
          $super(new Rectangle(0,0,100,100))
          this.addMorph(new TailMorph())
          // ...
      },
      walkRight: function() {
          this.moveBy(pt(10,0))
      }
      // ...
  })
  BoxMorph.subclass("TailMorph", {
      // ..
  })
  // ...
})
```

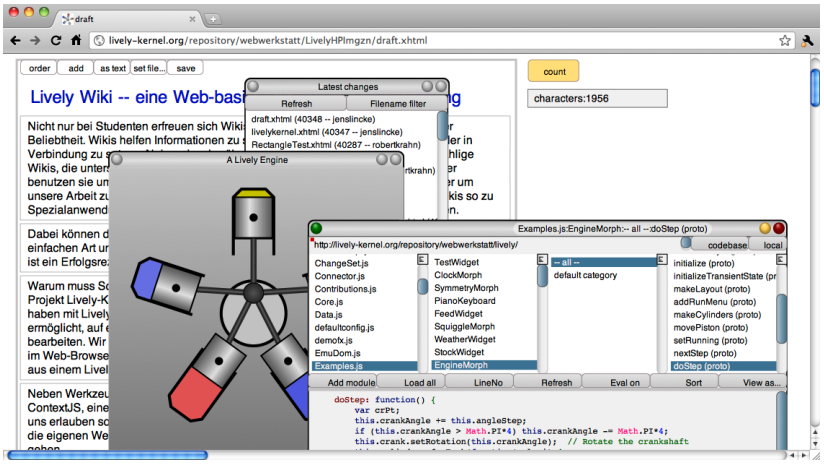


Figure 2.13: A Lively Wiki page with a simulation of a radial engine and a System Code Browser viewing its underlying class.

The module `animals` requires the `morphic` modules to be loaded (Line 1) before it will itself continue defining its own classes `ElephantMorph` (Line 2) and `TailMorph` (Line 13) that inherit from a general `BoxMorph`. The base system is loaded by dynamically resolving those module dependencies. After the object graph is deserialized, the `World` is started, and the system can be used. A Lively Kernel page as shown in Figure 2.13 can be used, like any other Web-page, to present information but, similar to a wiki, the text and graphical content on a page can be edited. Unlike a traditional wiki, this happens through WYSIWYG[2] direct manipulation. After editing a page, the result can be saved and sent back to the server (3). The Lively Kernel is developed with a Smalltalk-like approach. The system and all its applications are changed from within itself. However, unlike Smalltalk, even though classes can be edited at run-time, they are not persisted in the same object space as user objects, but are stored in JavaScript files that are also changed on the server (4). In this way, both user-editable content and the base system can be changed from within the wiki.

[2]What You See Is What You Get

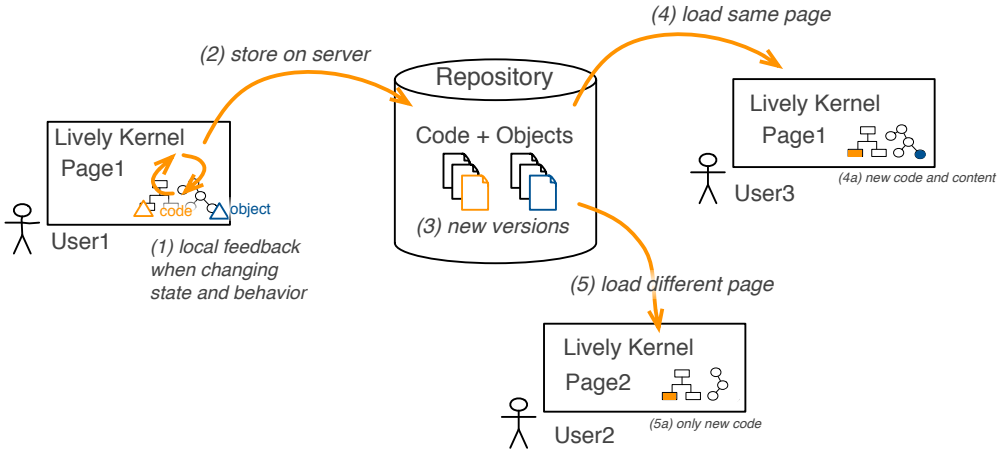


Figure 2.14: Editing content in Lively is propagated asynchronously like in a wiki. Similar to a wiki, content changes stay local to pages (4), but changes to the base system affect all users (5).

2.3.2 Wiki-like Collaboration for Content and Code

Since a repository in such a *Lively Wiki* [59] is shared, it serves an open collaboration environment. The flow of changes in the wiki is shown in Figure 2.14. Users can edit the content of a page directly in the Web-browser (1) and share the changes with other users in the central repository (2). The repository automatically versions all changes to code and content stored as serialized objects (3). Editing wiki content affects individual pages (4), but changing the base system behavior affects all users of the system likewise (5). This wiki-like collaboration happens asynchronously, a base system change affects other users' worlds only after they reload them.

Editing pages and changing the base system are two sides of development in Lively Wiki. Since a wiki-like collaboration process is very open and puts review after and not before publishing, such an approach to develop software is very different from normal *open source* development approaches [80]. There is a different entry barrier between open source and wiki-like collaboration. The entry barrier to make a simple edit in a wiki is much lower than making a successful contribution to an open source project [83]. This can

be partly attributed to the different skills necessary: being able to program vs. basic computer literacy and language skills. However, the contribution process is much more complicated and full of thresholds on the open source side, making small contributions—due to a bad overhead to programming ratio—not cost effective. Such minor contributions could be adding a better error message or fixing a layout bug. By allowing users to change not only content, but also to evolve the whole system, Lively Kernel is also a platform for researching open and direct collaboration approaches.

2.3.3 Problems of Self-supporting Development in a Wiki

Directly changing parts of the system in a self-supporting environment can be problematic as shown in Section 2.2.1. Such problems can occur when the part that should be changed belongs to the base system. The change could affect a feature of the environment or an essential tool that is needed to adapt the system itself further. Making a change there may deprive the system of it's own future—in other words: the tool breaks, the system dies, or it may still be running, but can no longer be changed.

In Lively Kernel, this self-supporting development cycle is extended from the run-time of the individual development session in a browser to the whole collaboration system. Since all changes are persisted and are immediately used for loading new pages, a problematic overall system state breaks the wiki development environment for everybody. Similarly to wikis, code and content are under version control, so all changes can be undone and the environment can be set to a working state again.

Even though, in theory, the system can break all the time, in our experience it used to be quickly restored or repaired. Since there are no technological barriers for contributing in such a system, people who where not used to such an environment can accidentally change the system without realizing it. As in every evolutionary process, some changes are good to the system, while others are bad. Multiple times we observed that users, who fixed a bug for their project, did not realize right away that they not only fixed this bug for themselves but for all users [67, 16]. On the other hand, many first time users broke the system on the attempt to change it. This seems to be a grave downside, but for the users it meant that they

received immediate help from other users. Users helped each other, so the system was restored for everybody. Additionally, it provided the occasion for collaboration and learning from each other, so that they will be able to help themselves better another time. This extremely optimistic approach of open source is inherited from the wiki-way of collaboration, but can also be traced back to the earliest Lisp programmers culture and their deliberate, non-existent rights management in their Incompatible-Time-Sharing System (ITS) [61, 77].

2.4 Summary

In this chapter, we have discussed self-supporting development environments and their inherent trade-off between short-feedback loops, self-adaptability, and the danger of breaking the system while adapting it. We have shown how object-oriented technology, such as subclassing and instance-specific behavior, can be used to adapt complex objects in an object composition, as this is a common requirement when working with graphical objects. We have presented Morphic as user interfaces framework that can be explored, decomposed, and modified at run-time by the user and discussed the limitations of prototype-based inheritance for that domain.

We have introduced Lively Kernel, a Web-based development environment that provides such a Morphic user interface. Lively Kernels explores how combining a self-supporting development approach with a Web-based authoring environment can shorten feedback loops, both, when adapting tools at run-time and when sharing adaptations and new tools with others. Lively Kernel-based wikis combine a Smalltalk development approach with an immediately usable open collaboration environment. As development experience in Lively Kernel has shown, the inherent danger of breaking a self-supporting environment while adapting it from within itself is amplified in a Web-based setting.

This thesis, therefore, explores how Web-based self-supporting development can be made safer. Users should be supported in adapting their tools at run-time and share their adaptations with others. Developing in a Web-based environment should still allow for an explorative and direct pro-

gramming style, and make it usable also for a wiki-like collaboration setting. Due to using a Smalltalk-like development approach with its traditional class-based development methodology, Lively Kernel lacked fine grained safety nets that help users avoid breaking the system while working on tools. Further, behavioral adaptations and new tools should also be shared with other users in a controlled way.

In the following chapters we will introduce *Lively Webwerkstatt* our approach to a self-supporting development environment. We show how tools can be developed as user-modifiable *parts*. With development *layers*, we demonstrate how to control the scope of changes in a way that it is safe to directly evolve a system at run-time. Further more, we present a way to share adaptations and new applications with other users in the wiki.

Part II

Run-time Tool Adaptation

3 Lively Parts and Development Layers

In this chapter we present *Lively Webwerkstatt*, our approach to tool adaptation in a self-supporting collaborative development environment. *Lively parts* [72, 68], are user-editable graphical objects that can be explored, cloned, composed, and shared through direct manipulation. Through deep cloning of parts, tools are prevented from breaking themselves when developed in a self-supporting setting. Further, Lively Webwerkstatt provides *development layers* [70, 66] as scaffolding mechanism that allows for context-dependent run-time adaptation of the underlying base system. Both approaches together enable the community to evolve their collaborative environment from within itself.

3.1 Lively Webwerkstatt

End-user adaptable applications, such as Lively Kernel, often exhibit a two-layer-architecture: the base system, usually written in a system programming language, and a scripting layer on top. In these applications, tools are part of the base system and are developed with a standard class-based approach using conventional object-oriented design and patterns that, for example, separate the data model from the user interface. In self-supporting systems, such as Smalltalk or Lively Kernel, the tools and the environment can be changed at run-time. In such reflective environments, users can also learn about the implementation of their tools by exploring concrete instances. They can even experiment by changing the behavior of objects at run-time through editing their class definitions. But as discussed in Section 2.2, direct adaptation can lead to undesired behavior and using

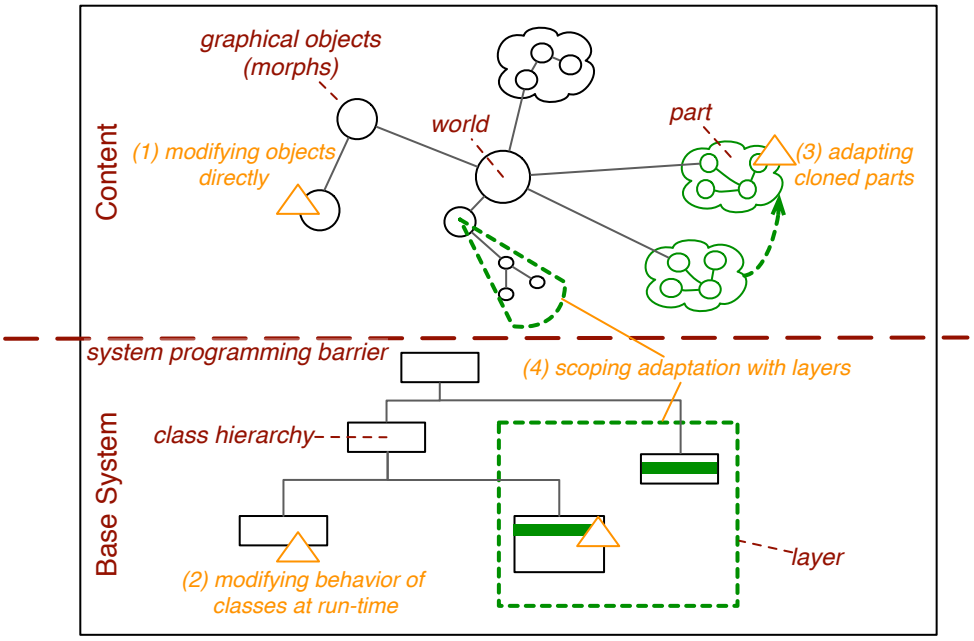


Figure 3.1: Overview of run-time adaptation in Lively Webwerkstatt

other object-oriented means like subclassing, instance-specific behavior or prototypical inheritance, development can become safer but at the same time also more complicated.

In Lively Webwerkstatt we provide two approaches—*lively parts* and *development layers*—that should mitigate that problem. Figure 3.1 shows an overview of object development in Lively Webwerkstatt. As it is based on Lively Kernel, it provides the same basic means of direct object modification (1) and programming with classes (2) as presented in Section 2.3.1. Both approaches can be used to develop in a Smalltalk-like style, making the workflow interactive and shortening the feedback loops. Being Web-based, Lively Webwerkstatt further allows users to collaboratively develop applications and evolve the environment in a wiki-like way. But as discussed in Section 2.3.3, being a Web-based collaborative environment amplifies the danger of breaking the development tools while changing them, by potentially breaking them for all users of the wiki, too.

Lively parts and *development layers* can serve as scaffolding during development and help users to adapt their tools more safely while using them at the same time. This is achieved by implementing tools as run-time modifiable *parts* which can be deeply cloned (3) so that exploratively adapting them gets safer. By publishing them in a *parts bin*, tools can be exchanged between worlds and users. Since some features of the environment are not expressible as graphical objects, *development layers* (4) allow for context-specific adaptations of the base system that make its evolution at run-time safer.

3.2 Lively Parts–Developing Tools as User-modifiable Objects

In contrast to the first versions of Lively Kernel (as presented in [50]), objects in Lively Webwerkstatt can have persistent instance-specific behavior. These *scripted objects* allow creating applications, games, tools and other active content without programming their behavior in classes.

All properties of graphical objects (morphs) like *style*, *position*, *extent* or *text content* can be changed directly using Lively's *halo* user interface. Tools like the *StyleEditor* or the *ObjectInspector*, which can be invoked from the halo (as described in Section 5.1.2), can also be used.

A morph in Lively usually consists of several other morphs, forming a tree structure which is also called a *scene graph*. The composition of morphs can be changed via drag and drop or by using the halo. Reusable *lively parts* are created by publishing named morphs in a shared repository called *parts bin*. Due to the JavaScript object model, a morph's scripts are only properties which happen to be functions. So, by overriding methods of an object's class, the behavior provided by the base system can be adapted. Section 5.2.2 shows how such persistent, instance-specific behavior can be implemented in JavaScript.

3.2.1 Cloning Parts before Adaptation

To illustrate how cloning parts makes the adaptation of composed objects safer, we come back to the elephant example from Section 2.2. In our

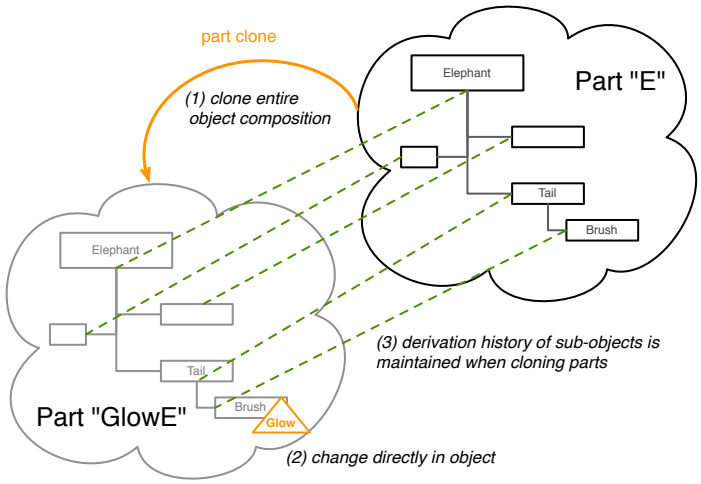


Figure 3.2: Behavioral adaptation in the elephant example using parts

scenario, we want to experimentally adapt the `Brush` of the elephant `E` without modifying `E`, but keeping the development itself direct and easy by giving immediate feedback. Doing something directly and not directly at the same time is a contradiction that we cannot solve. Therefore, we allow users to modify something directly that is very close to the original: its clone.

In object-oriented systems, cloning an object means creating a shallow copy of it. A shallow copy is produced by creating a new object and assigning it the same properties (attributes, instance variables) than the original object has. Since both objects reference the same objects afterwards, this kind of copy is called shallow. But complex objects, like morphs, cannot be meaningfully copied that way, since a morph cannot have two owners. When copying a morph, all its submorphs have to be copied as well. The process of copying not only one object but recursively all its subobjects that are referenced by it, is called deep cloning. As shown in in Figure 3.2, deep cloning the object `E` also produces a copy of all its submorphs. We call an object with all its subobjects that would be copied during cloning a *Lively Part* and use the deep cloning mechanism as means in interactive development and for sharing complex objects across worlds by publishing them in a shared repository as described in Section 3.2.2.

For the adaptation of graphical tools at run-time, we use the deep cloning of objects as a mechanism to produce disposable copies which can be modified safely (1). If experimental changes break the object or produce undesired effects, the copied object can be discarded and the original object stays unharmed. For our scenario this means that after cloning the elephant `E`, we can directly add the `Glow` behavior to the `Brush` in the cloned part `GlowE` (2). That way we do not interfere with the original part `E`.

Unlike copying text or saving and reloading the same world, the cloning will produce two coexisting exemplars. And similar to cloning animals in the real world, the clone will be very similar but not identical to the original. Since we want to preserve the invariant that all objects have a unique identity, we have to give every object that is created during the cloning process a new unique identity.

When working with objects, the ability to identify them even though their state has changed is important. An example application for using object identity is the comparison of complex objects as described in Section 5.3. When comparing objects, using the identity can help figuring out which subobjects have changed and which ones were added or deleted. Without being able to rely on the identity, objects can only be structurally compared, making it hard to provide users with meaningful diffs, especially when the composition of objects was changed.

To mitigate the problem of changing identities when cloning, lively parts keep a history of their past identities. This history in form of *derivation ids* is meta-information that is stored per object and updated in the cloning process. For an evaluation of the overhead needed to store such information see Section 6.4. Figure 3.2 shows how this derivation history is not only maintained for the root object that is cloned, but also for all its subobjects (3). The derivation history does not interfere with the functionality of the self-contained `GlowE`, because in contrast to prototypical inheritance, it does not share its behavior with part E.

To make development in a self-supporting system safer, tools can be deeply cloned before adapting them. This does not have to be a conscious decision of users but can be automatically provided by the system. Users do not need to remember to actively clone a tool each time, so they can safely modify it. In Lively Webwerkstatt the deep cloning of objects becomes

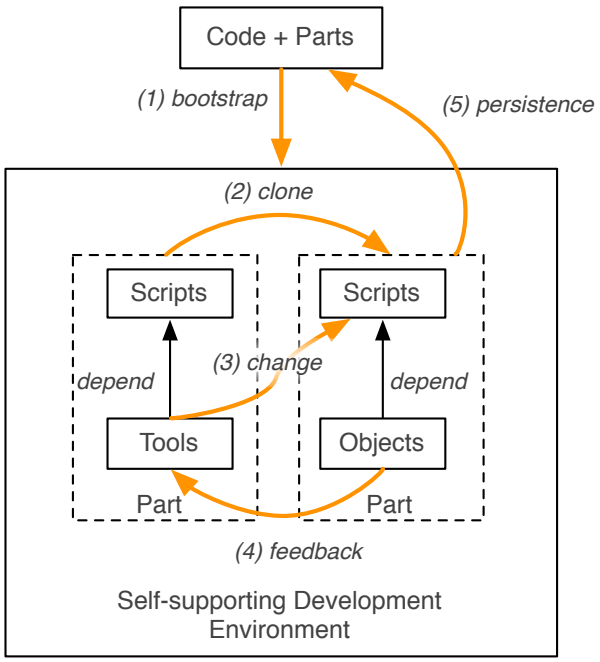


Figure 3.3: Flow of changes and feedback in a Self-supporting Scripting Environment

part of the normal user interaction: graphical objects are copied all the time and tools are copied by default when they are opened. So, editing the scripts of an `ObjectEditor` with another `ObjectEditor` is safe, because they automatically are independent clones.

Following that approach, the dangerous cycle of changing a behavior and accidentally breaking a tool while doing so is cut off by first cloning the part (as shown in Figure 3.3). Developers can change scripts and objects directly (3) in a cloned part tool (2), without breaking the tool itself, because every instance has its own set of scripts and state. But since the tools are still running in the same environment, the feedback loop (4) is kept short and tools can directly interact with the objects they manipulate. After the object is changed in the run-time environment, it can be (re-)published to the shared repository (5) and, therefore, replace the original version. In

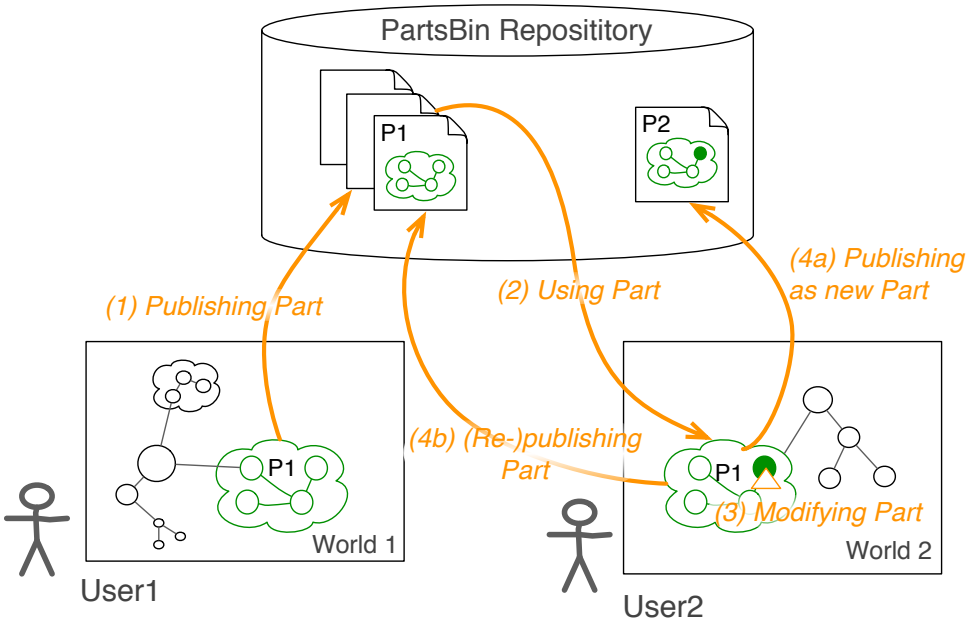


Figure 3.4: Workflow of sharing parts through the PartsBin

Lively Webwerkstatt, we combined the saving of content with immediate publication. This allows developers to directly fix a bug as it occurs, to fix a typo on a button label, rearrange morphs to produce a tighter layout, or just change colors to make the tools more appealing and share the changes immediately with others.

3.2.2 Sharing Parts in a Parts Bin

Parts are graphical objects that the original developer decided to extract from the world and publish separately so that they can be used and developed in other worlds, too. A part does not only contain a single object, but it also contains all the objects that belong to this object. The graphical objects, which an object consists of, are called `submorphs` and are serialized with the part, as well as all non-graphical objects that are reachable from these objects and can be serialized. Some objects are excluded or handled specially. The implementation of our serialization approach is discussed

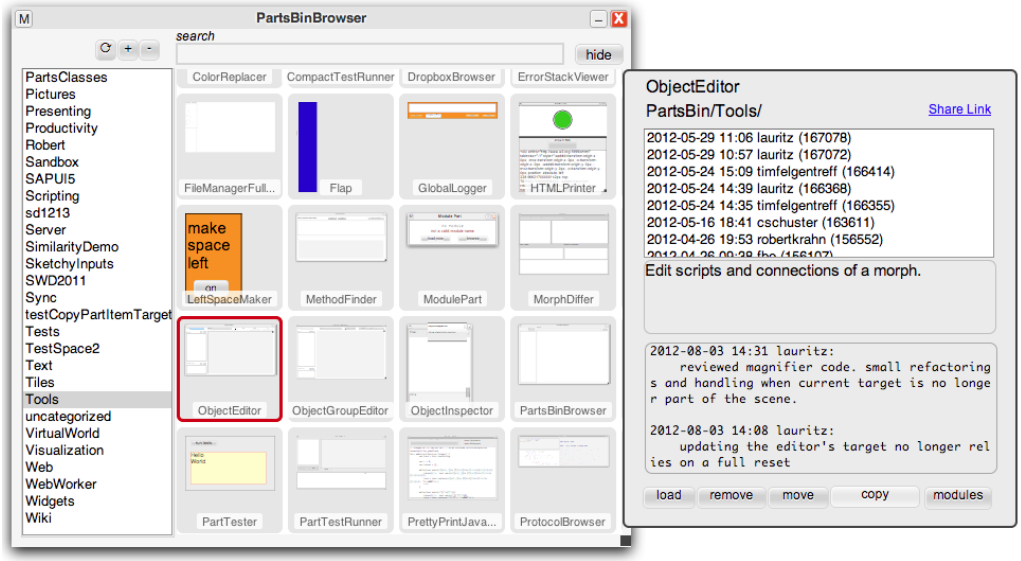


Figure 3.5: Screenshot of the `PartsBin` browser with `ObjectEditor` selected from the `Tool` category

in Section 5.2. A serialized *part* is published under a unique name in a *parts space*. Additional meta-information and a preview of that part are stored separately. By publishing a part, a new revision is created so users can revert changes as needed. All the *parts* are stored in a *parts bin* and are grouped in *parts spaces* which also provide a name-space and are used for categorization, too. The development tools provided by the environment are user-editable objects just as well and are stored in the *parts bin*, too. They can range from a simple `ColorChooser` to complex tools such as an `ObjectEditor`. The screenshot in Figure 3.5 shows the `Tools` part space in a parts bin with the `ObjectEditor` selected and its details.

Figure 3.4 presents the workflow of using a parts bin for collaboration. A user can decide to publish a part `P1` from *world 1* (1) in a central repository. The published part can then be cloned by a second user (2) and can be used and modified in *world 2* (3). When the second user decides to share the adaptation, the part can be published either under a new name `P2` (4a) or in the original location using the same name (4b). By creating a new version,

the repository ensures that old revisions are still accessible and changes can be undone.

3.3 Scoped Behavioral Adaptation with Development Layers

Not all features of the development environment can be represented as individual and exchangeable graphical parts that can be cloned to safely work on them (as discussed in the previously section). There is a base system in Lively Kernel that is implemented in a standard class-based way. The base behavior of all morphs, shapes, widgets, or the world are implemented as classes. To implement features that should affect not only an individual morph, but all morphs of a specific kind at once, the adaptation of individual graphical objects as presented in the previous sections is not sufficient.

To address this problem, we use context-oriented programming (COP) as a means to context-specifically adapt behavior at run-time. In this section we show how to employ COP in the process of evolving self-supporting development environments. Putting COP to work, programmers do not modify the core classes and methods directly, but use COP layers instead. Layers can be scoped to only affect the behavior of the objects under construction.

3.3.1 Context-specific Behavioral Adaptation in Object Compositions

Similar to lively parts, layers can be used to refine behavior in a complex object composition. Unlike the approach presented in the previous section, layers refine the behavior of classes in a context-specific way. This allows expressing experimental behavior in form of a layer that can be tested in a controlled way. A layer can be activated globally, during the extent of a specific computation (see dynamic scoping in Section 4.4), for individual objects, or for groups of objects that are part of a domain-specific structure (see structural-specific scoping in Section 4.5).

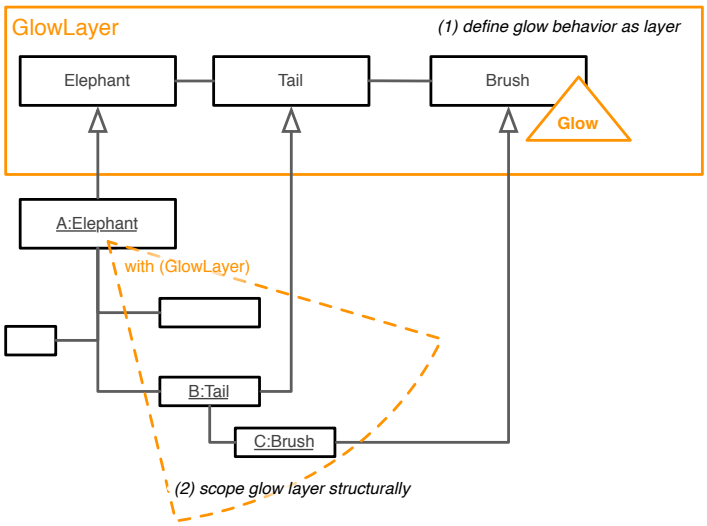


Figure 3.6: Behavioral adaptation in the elephant example implemented with COP.

Coming back to the elephant example from Section 2.2 and Section 3.2.1, Figure 3.6 shows how the elephant can be adapted with COP layers. The behavioral adaptation is implemented as a `GlowLayer` that refines the class `Brush` (1). That layer is then structurally scoped in the elephant object `A` and its domain-specific composition hierarchy consisting of the tail `B` and its brush `C` (2). The adaptation that only affects `C` can be scoped by (de-)activating the `GlowLayer` as needed.

The context-specific activation allows developers to experiment with the elephant's glowing feature on other elephants. One of the main advantages is that development layers allow experimenting without modifying the source code of classes or creating new instances. If needed, the glowing feature can be removed by deactivating the layer.

3.3.2 Separating Changes in Layers

We use ContextJS [64, 70, 60], our library-based COP extension to JavaScript. ContextJS is a JavaScript library and that uses method wrappers [15] for

its implementation. It allows defining behavioral variations to objects and scoping these in various ways. ContextJS and its open implementation of layer composition will be discussed in chapter 4. An overview of ContextJS syntax and semantics is provided in Section 4.3.

Below we look at an example of an `EventCounter` that should represent code from the base system. An anonymous object with the attribute `n` and a function `count`, which gets an event `evt` as argument, is created and assigned to the global variable `EventCounter`. JavaScript does not distinguish between state and behavior and represents both, attributes and functions, as named properties of objects.

```
EventCounter = {
  n: 0,
  count: function(evt) {
    this.n = this.n + 1;
  }
}
```

Due to its reflective capabilities, objects and behavior can be changed at any time in JavaScript. If this `EventCounter` is used by tools in a self-supporting development environment, changing it can be dangerous. If developers are interested in which events are counted, they can add an `alert` statement to the count method. The alert statement can be used to display values to the user:

```
EventCounter.count = function(evt) {
  alert("evt: " + evt);
  this.n = this.n + 1;
}
```

But this might instantly bring the system to a halt if the `EvenCounter` is used by tools, because of the potentially many alerts that might get sent on every user interface event. The problem here is not that the adapted code is erroneous, it just dynamically uses resources of the environment in a unexpected way. By using COP and defining the new behavior in a layer, the problem can be circumvented as follows:

```
cop.create("DevLayer").refineObject(EventCounter, {
  count: function(evt) {
    alert("evt: " + evt);
    this.n = this.n + 1;
  }
})
```

The layer overrides the original behavior without proceeding to the base behavior. Since we do not add a new crosscutting concern here, but work on the base method source, there is no need to proceed to other partial methods of the layers or the base system. A version that would use ContextJS's proceed looks as follows:

```
cop.create("DevLayer").refineObject(EventCounter, {
  count: function(evt) {
    alert("evt: " + evt);
    return cop.proceed(evt);
  }
})
```

That version ensures that original method can be adapted. Hence, the layer does not include an outdated version of the code, but calls the original code with its `proceed` statement. Nevertheless, the first version is useful for experimenting with the complete original source code.

3.3.3 Scoping Layer Activations

To actually affect the system, the `DevLayer`[1] can be activated in a certain scope in various ways, just as needed. The standard mechanism in COP is dynamic scoping:

```
cop.withLayers([DevLayer], function() {
  // a call of EventCounter.count() in this scope
  // will execute the new behavior in DevLayer
})
```

In interactive environments, most control flows are triggered by events. Other scoping mechanisms, such as instance-specific and structural scoping [64], are better suited to activate the new behavior. Figures 3.8 and 6.4 demonstrate such layer activations.

[1] cop.create creates the layer and assigns it to a global variable

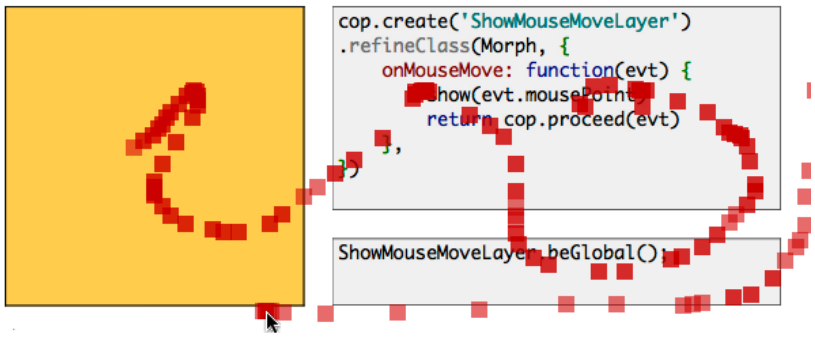


Figure 3.7: A layer that is used during the development and visualizes move events of the mouse. The layer is globally activated. When the mouse is moved over any object including the workspace with the code, the red rectangles are shown.

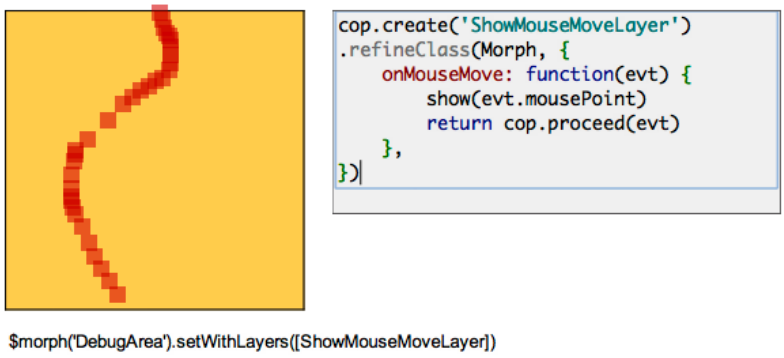


Figure 3.8: The ShowMouseMoveLayer, which is defined in the right workspace, is only activated for the yellow rectangle on the left. The layer is structurally scoped by activating the layer for the DebugArea.

If developers are satisfied with new behavior, they might want to apply it to the whole environment. They can do so by activating the layer globally. If they discover an error they can still deactivate the layer and fall back on the old behavior.

```
DevLayer.beGlobal();  // -> activate layer for all objects, even the tools
DevLayer.beNotGlobal();  // -> deactivate layer if necessary
```

In Lively Webwerkstatt the layers can be used during development to restrain the effects of debugging code. Figure 3.7 shows how code was added to the base system to see where mouse move events are fired. Adding such debugging code to core behavior can be problematic, as it also affects tools such as the workspace. Figure 3.8 shows how using the `ShowMouseMoveLayer` and structural scoping mechanisms such adaptations of the base system can be restrained, so that the debugging behavior is only active for some specific objects in the system (in that case the yellow rectangle called `DebugArea`). The layer is structurally scoped by using `setWithLayers` on a graphical object containing other objects. This scoping mechanism is not a general one, but a domain-specific layer composition introduced by the Morphic framework. The implementation of such domain-specific scoping is discussed in Section 4.2. Once the layer is activated structurally for the `DebugArea` morph, it is active for all its submorphs, too.

For the general workflow in self-supporting systems this means that COP layers can be used as a scaffolding during development, so that tools are temporally separated from the things they work on. Figure 3.9 shows how *development layers* can break up the reflective dependencies in an self-supporting development environment (1). Even though the tools needed to adapt the system depended on base system behavior like classes, they are not affected by their own changes (2), because the `DevLayer` that refines class behavior is only active for the objects under development. The new behavior can be tested in a scoped way (3) and, if desired, it can be activated immediately for the entire system. Finally, the new layer can be stored back to the server (4) so that other users can also experiment with the new feature.

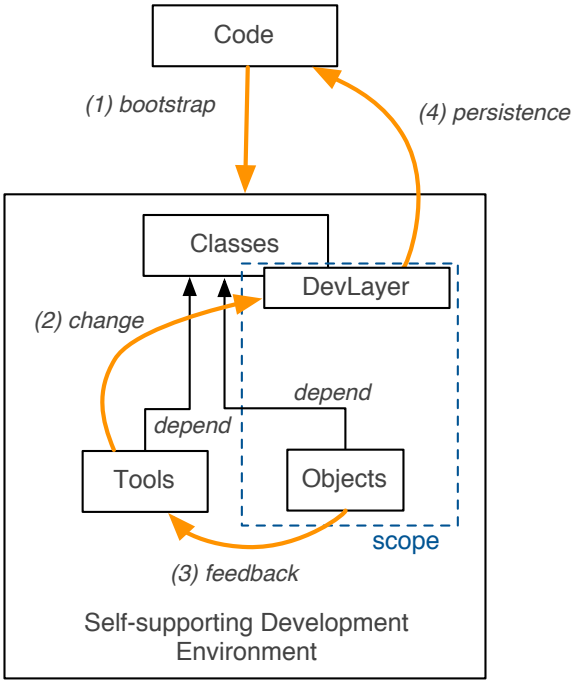


Figure 3.9: Development with scoped behavioral adaptations

By composing behavioral adaptations at run-time, the source code of the base system is not permanently changed. If necessary, the layer can be refactored and merged into the classes of the base system [66].

3.4 Combining Parts and Layers

This sections discusses how parts and layers can be shared across different Lively Wikis and how both approaches can be combined, but are not unified yet.

3.4.1 Sharing Parts and Layers across Lively Wikis

The development with *lively parts* and *development layers* is not limited to one shared repository. Even though, *Lively Webwerkstatt* uses one central repository for storing modules, worlds, parts, layers and other content, the development of code and objects is not limited to one Lively Wiki instance. Since classes and layers are stored as modules in plain text files, they can be copied and merged and handled with standard source control tools such as GitHub [2].

Lively worlds and parts are stored in a special serialization format that will be described in Section 5.2. Because of that they cannot be meaningfully managed using standard text-based tools. But as every Lively Wiki can deserialize and handle them on an object level, worlds and parts can be exchanged across wikis. Since object identities are unique and their derivation history is stored in the object, worlds and parts are not bound to one repository. Parts can, for example, be directly copied via the Web-browser and pasted in another World running on a different server via the Web-browser's clipboard paste command. They require a compatible base system that provides the classes needed for deserializing the objects. Since parts contain their own behavior, complete applications or tools can be transfered using that technique.

[2] `https://github.com/LivelyKernel/LivelyKernel/`

3.4.2 Unifying Parts and Layers

COP layers are very powerful and general meta-programming tools. Besides being useful for adapting the base system, they can also be used to adapt graphical objects. But unlike the direct manipulation and free editing with lively parts, COP layers refine only behavior and cannot be used to interactively modify object state. Since cloning of tools cannot be used to adapt the base system, a technique like development layers is inevitable for a safe and stable development environment.

Lively parts and development layers serve similar purposes in Webwerkstatt: they make self-supporting development safer and allow exchanging objects and features between different worlds and users. But they remain different approaches. Finding a unified approach for programming in a self-supporting, collaborative development environment that spans both end-user-like programming and system development remains future work.

3.5 Summary

In this chapter we have presented Lively Webwerkstatt and two approaches that help to make tool adaptation in a self-supporting, collaborative development environment safer. In Webwerkstatt, all graphical tools used for development are implemented as user-modifiable lively parts. Cloning tools before adapting them will prevent them from breaking themselves and the environment around them.

We further have shown how COP layers can be applied as development technique to isolate changes to the base system during development. Instead of modifying classes directly at run-time, changes are applied to layers that can then be scoped to specific parts of the system. With these layers new features can be tested with less risk of breaking the system during this test phase. Once they meet the developers' needs, layers can be applied to other parts of the system, activated globally and shared with other users in the collaborative development environment.

4 ContextJS

Scoped *behavioral variation* can serve as scaffolding that support a safer *tool adaptation* in *self-supporting development environments*. As discussed in Section 3.3, layers of *context oriented programming* [42, 21] (COP) are suited to capture and scope changes as needed to prevent breaking the development environment while evolving it from within itself. For the development in Lively Webwerkstatt, we developed ContextJS [4, 64], our JavaScript language extension for COP and an open implementation of COP layer composition.

In this chapter we introduce COP and present dynamic layer scoping as its default means to restrict behavioral variations to the dynamic scope of method executions. To exemplify this, we present use cases that needed alternative scoping mechanisms. Since new scoping mechanisms, like structural-specific scoping, are domain-specific, we propose to allow framework developers to specify them with an open implementation. We discuss global, dynamic, and structural-specific scoping and apply them in the earlier presented examples.

4.1 Scoping of Layer Activations

COP extends object-oriented programming by providing dedicated language abstractions for defining and composing variations to basic program behavior. Behavioral variations are encapsulated by *layers*, modules that can crosscut classes (or depending on the implementation also objects). Layers can be dynamically de-/activated—and composed with other layers—for the dynamic extent of a code block. This mechanism allows scoping behavioral variations to specific control flows. They are an encapsulation mechanism orthogonal to object-oriented decomposition. This meets the

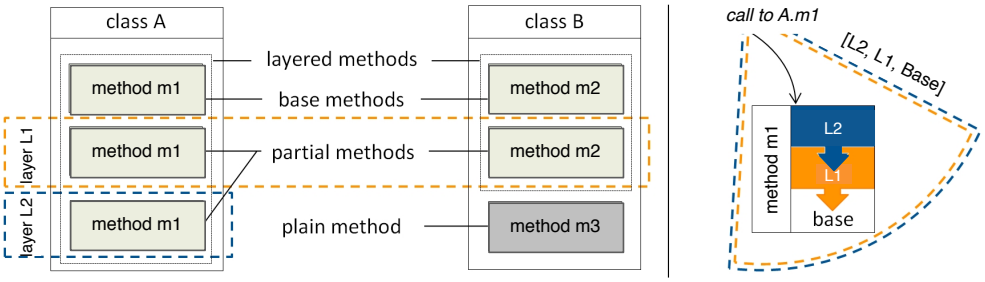


Figure 4.1: Method kinds and sideways composition in COP

nature of behavioral variations, whose implementation often affects various parts of a system and cannot be encapsulated by a single object.

In this section, we present examples for which we employed COP and discuss the benefits of layer-based adaptation. While COP is helpful for control flow driven use cases, existing dynamic scoping mechanisms are not applicable to interactions not centered on control flow. We discuss the inapplicability of existing layer activation mechanisms and the need for alternative specifications.

In the COP execution model, a statement's semantics depends upon the context in which it is evaluated. Behavioral variations, such as variations of method executions, become explicit concepts in COP. Typically, context-specific behavior requires adaptations at several points in a system, constituting its implementation as a crosscutting concern [84]. Behavioral variations are defined within a *layer* allowing for the modularization of functionality that would otherwise be scattered over an object-oriented decomposition. As discussed in Section 3.3, we use layers as a means to make development in a self-supporting system more safe.

Depending on the language's features, behavioral variations are implemented by *partial method*, *function*, and/or *class definitions* that encapsulate context-specific functionality; we will focus on partial method definitions. Figure 4.1 (left) illustrates two classes defining partial methods. In COP, layered methods consist of a *base method definition* and at least one *partial method definition*. The base method is executed when no active layer pro-

vides a corresponding partial method. Layers do not affect the execution of *plain methods*, which are methods that have no partial definition.

Layers can be activated and composed with others at run-time. Therefore, the object-oriented method lookup, which is based on a method's signature, its object, and inheritance rules, is extended with a sideways lookup that considers partial method definitions of active layers. This layer-aware method lookup is also denoted as *sideways* or *layer composition*.

In a layer composition, multiple layers may provide partial definitions of the same method. In that case, a partial method can *proceed* to the next partial definition in the composition, or, if none exists, to the base method definition. When activated, layered method calls are dispatched to the partial method provided by the layer. Partial methods can be executed before, after, or around the base method definition. Figure 4.1 (right) shows a method dispatch of `A.m1` while layers `L2` and `L1` are active. Given that the partial methods proceed with the call, the partial method `L2` is called before `L1` and the base definition.

Although COP does not prescribe a certain implementation strategy, most of the COP implementations described in literature scope layer activations to the dynamic extent of a block of statements [21, 41, 5, 4]. For many control-flow driven applications, dynamic extent-based layer composition is an appropriate mechanism. However, behavioral adaptations in general can also depend on scopes other than the control flow, such as a dynamic object structure or object state. Having conducted several case studies in which we applied COP to various projects [71, 59] to Lively Kernel [50] and Lively Webwerkstatt as discussed in Section 3.3 and Section 6.2, we identified needs for scoping strategies different from what has been proposed and implemented so far.

4.1.1 Programming with Dynamically Scoped Layers

An example of programming with dynamically scoped layers is an adaptation of an xUnit-like test runner [10] in Lively Webwerkstatt. The test runner as shown in Figure 4.2 *(A)* only displays the execution time of entire xUnit test cases, but no fine-grained execution information of individual test methods within test cases. A straightforward object-oriented imple-

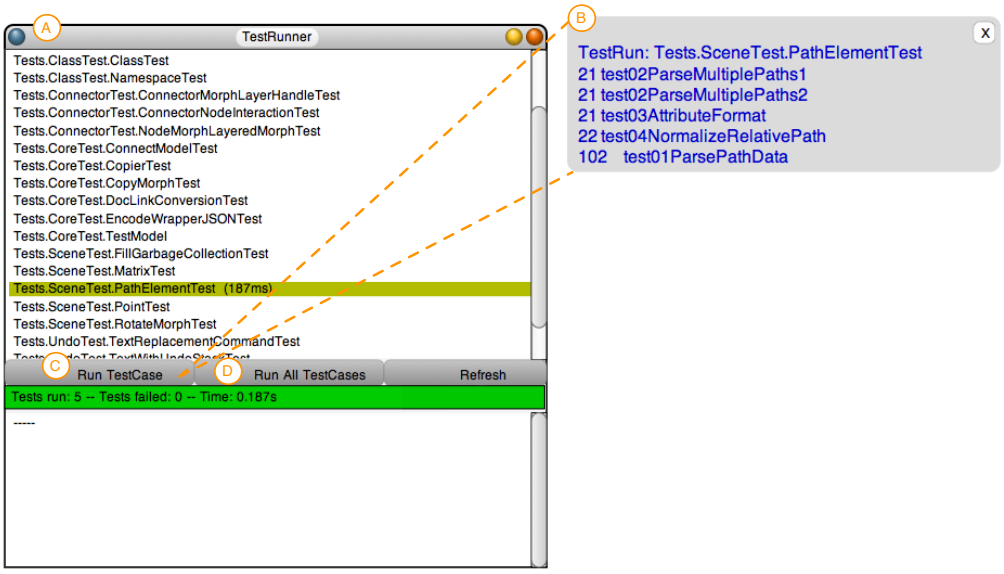


Figure 4.2: Test framework adaptation. *(A)* The test runner only shows the execution time of whole test cases. *(B)* A user adapted the test runner's behavior by measuring and displaying the execution time of individual test methods when a selected test case is run.

mentation would measure execution time whenever a test method is run. Therefore, the class `TestCase`, which is responsible for the execution of test methods, has to be adapted. Since our system contains a large number of tests, this static solution would decrease performance.

Instead, measuring and displaying results should only be active for the execution of an explicitly selected test. Figure 4.2 presents a screenshot of a JavaScript-based unit test tool. Whenever the button *Run TestCase* (Figure 4.2 *C*) of the test runner is pressed, the measurement adaptation should be activated, displaying the result in a separate window (Figure 4.2 *B*). Scoping of this feature is needed, because the displaying of individual test-runs times should be disabled when the test runner executes a batch of tests (Figure 4.2 *D*) or when test cases are used for other purposes, e.g. when executed in the context of other tools.

To address this adaptation of the test runner, COP layers can be employed. The core behavior of the test framework is implemented in three

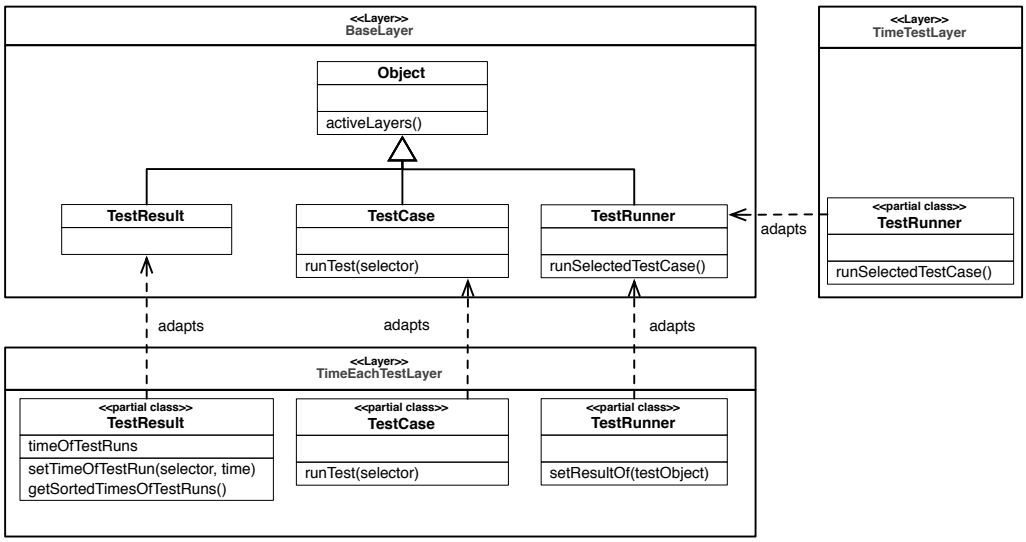


Figure 4.3: UML Class Diagram showing the adaptation of Lively Kernel's xUnit test framework measuring and displaying the execution of each test method.

classes: `TestRunner`, `TestCase`, and `TestResult`. As shown in Figure 4.3, the behavioral adaptation is implemented in two layers: `TimeTestLayer` and `TimeEachTestLayer`. Since time measurement should be implemented as a separate concern, we do not modify the base method definition of `runSelectedTestCase` in the class `TestRunner`, but define a partial method in the layer `TimeTestLayer` as a suitable entry point for the adaptation. To adapt the test runner behavior for any execution of the *Run TestCase* button, we activate `TimeTestLayer` globally.

Measuring execution times of individual methods is the responsibility of the class `TestCase`. Its adaptation is defined in the layer `TimeEachTestLayer`, which is dynamically activated in the `runSelectedTestCase` method. Figure 4.4 shows how the `withLayers` statement activates `TimeEachTestLayer` during the execution of an anonymous function, which is used as a scoping construct. The `withLayers` statement allows implicitly passing context information and changing the layered method composition at the moment the method `runTest` of the class `TestCase` is executed. The time each test

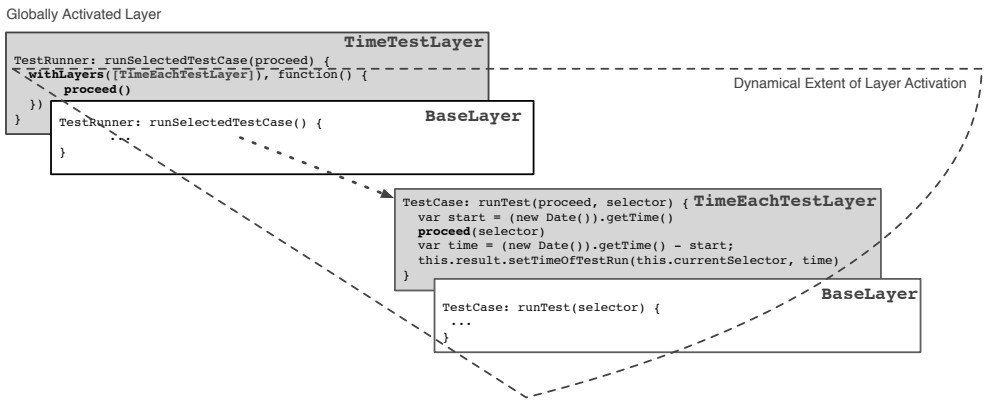


Figure 4.4: The refined `runSelectedTestCase` method activates `TimeEachTestLayer` and thus refines `runTest` of the class `TestCase` for the dynamic extent of this execution.

method execution takes is stored in an instance of the class `TestResult`, which was adapted by adding new behavior and new state as shown in Figure 4.3.

Our test case adaptation emphasizes an issue of object-oriented adaptation techniques. It requires extensions and refinements of several methods and fields of the abstract class `TestCase`. Using plain object inheritance, the adaptations could be either specified within `TestCase` itself or in a new subclass. The former is not desirable with regard to separation of concerns, since context-specific behavior should not be defined within an abstract superclass that handles core concerns. The latter requires changes to the inheritance chains of all concrete test cases, letting them inherit from our new class. However, we cannot assume to have access to the source code of all `TestCase` subclasses; thus, this strategy is fragile. Layers allow more fine-gained modularization using sideways composition, thereby supporting the definition of method refinements and of new methods and state. Sideways composition extends object-oriented adaptation techniques; it can be used as a delegation technique preserving object identities, and it can be applied where subclassing is not suitable.

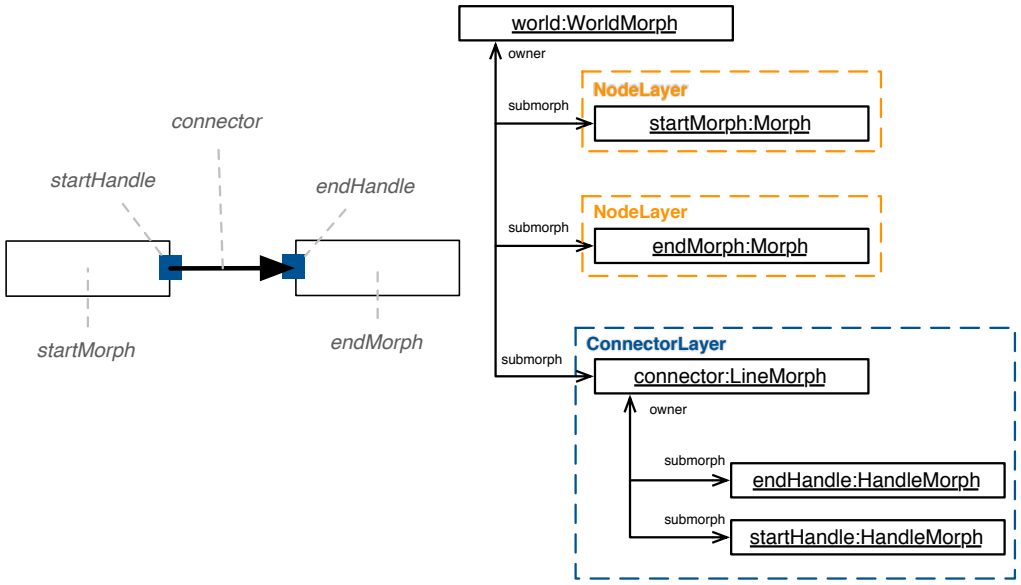


Figure 4.5: A Morphic scene graph of the connector example. Both `startMorph` and `endMorph` have instance-specific `NodeLayer` activations. `startHandle` and `endHandle` are in the structural scope of the `ConnectorLayer` which is explicitly activated only in the `connector`.

4.1.2 Lack of Alternative Scoping Mechanisms

While layers are useful for defining behavioral variations, the existing dynamically scoped layer activation is insufficient in situations that require alternative ways of describing scope. A situation that demonstrates this issue comes up while programming graphical objects using a *Morphic* [76, 99, 75] implementation as described in Section 2.2.5.

We use the example of developing a *connector* to demonstrate the need for new instance-specific and structural scoping of layer activations. One example of a simple graphical object in Morphic is a *line*. Lines can be moved by dragging their handles, which are small rectangles appearing at their ends. These handles—children of their corresponding line in the Morphic scene graph—are instances of the class `HandleMorph`. This parent-child relationship, as depicted in Figure 4.5, is an example of how context is constituted from object structure. In the Morphic domain, the scene graph

Figure 4.6: First requirement in connector example: connectors should update themselves when connected morphs change their position.

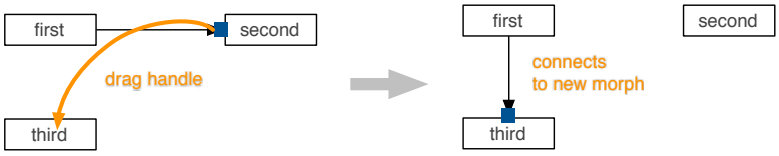


Figure 4.7: Second requirement in connector example: dragging a handle (small rectangle) onto a third morph connects the line to that morph.

structure is embodied by a bidirectional `submorphs/owner` relationship. There are other domains that have similar object structures, e. g., parse trees that may define their object structure in different ways.

Based on Morphic lines, we want to implement connectors. A connector is a line that graphically connects two morphs.

Our implementation has to consider two requirements. First, when one of the connected objects moves, the connector should automatically update its position, as shown in Figure 4.6. Second, a simple line can be moved by dragging its *handle*, but a connector line should not only be moved but also be reconnected when its handle is dragged onto a new morph, as shown in Figure 4.7.

We considered modeling and implementing this scenario by providing special kinds of morphs applying basic object-oriented techniques. We could either have included the new behavior into `Morph`, which would have bloated this already large class. Alternatively, we could have subclassed `Morph` and restricted the potentially connectable graphical objects to instances of this subclass hierarchy.

Instead, we want to use a layer-based sideways composition of context-oriented programming [42]. Layers allow us to encapsulate our behavioral variation without tangling the `Morph` class declaration and at the same time avoid unwieldy subclassing. We are convinced that layers are a good way to express such class adaptations and to separate them as dedicated concerns.

However, COP's standard scoping mechanisms are not applicable to our scenario. First, morphs need to dynamically adapt node behavior when they are attached to a connector line. Second, handle behavior needs to be adapted when they belong to a connector; i. e., when they are used in the context of a line playing the role of a connector.

With standard COP techniques, the respective layers would be activated upon initiation of the connector's control flow. However, user interface events can also influence the handle's behavior but run in separate control flows that will not touch the connector and its composition statements. This behavioral variation is not control flow centric, but rather depends on specific objects and the structure of the morph object tree (*scene graph*).

We identified the following new layer activation scopes. First, there is a need for layer activations depending on a *specific object*. Second, *structural object hierarchies* should be taken into account. Such structural context information should be used in combination with instance-specific layer activation to extend scoping to object structures. Our intended behavioral adaptation should be defined in `HandleMorph` and activated for handles when they are submorphs of a connectors, as shown in Figure 4.5 [1].

The new scoping strategies require domain specific information about structural hierarchies and instance-specific activation algorithms. Hence, we need a flexible implementation of layer activation to suit application- and domain-specific needs.

4.2 Open Implementation of Layer Composition

As a result of our analysis, we identify new scoping strategies that must be accessible for application-specific customization. We propose a holis-

[1] In Lively Kernel, handles are submorphs of the morphs they adjust so that they are automatically transformed with them. This makes their implementation much easier.

tic approach that integrates our new strategies with existing mechanisms. Our solution is based on an open implementation [56, 58] in which layer composition strategies are encapsulated in objects. Objects can add other scoping mechanisms or disable layers completely, by overriding the default layer composition behavior. This enables developers to implement domain-specific scoping strategies. The definition of a strategy is optional; a default implementation provides the original control flow-specific scoping mechanism.

Moving responsibility for adaptations to objects has some implications. Layer composition can be late-bound, being computed upon layered method execution (as opposed to the start of a dynamic extent), which allows for object-specific adaptation within a control flow. However, providing more control can also lead to bad design. Theoretically, one could specify a different composition strategy for each object in a system, ending up with an unwieldy application design. Nevertheless, we found a set of use-cases for which we believe the benefits of such fine-grained and open adaptation strategies outweigh this drawback. For example, nodes in a tree can provide behavioral variations depending on their position, i. e., structural context, affecting (only) their children; objects can be adapted to play a role independent from their control flow; or specific objects can be closed against any adaptation by providing an empty layer composition. Our experiences with our approach so far suggest that implementations of new scoping strategies require modifications of relatively few classes or objects. Moreover, few applications will require new adaptation scopes and not resort to existing ones. The implementation of new layer activation and composition mechanisms is meta-programming and, therefore, should only be used where actually needed—specifying scoping strategies should not be part of a standard development process.

4.3 ContextJS Syntax by Example

ContextJS [64], our COP language extension for JavaScript, is implemented as a JavaScript library. It allows defining behavioral variations of objects as

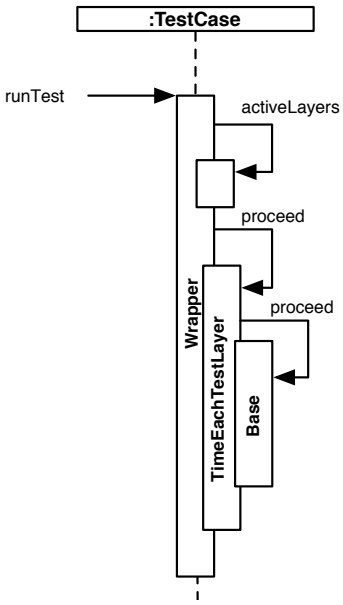


Figure 4.8: Sequence diagram for the layered execution of `runTest` in a `TestCase`

partial methods. In addition, it supports the definition of partial classes for a library-based class system.

ContextJS allows defining layers that refine methods of objects and classes. Since JavaScript is based on prototypical inheritance between objects, classes are just a convention. The class refinements are, therefore, only syntactic sugar for refining the class prototype object. ContextJS implements the class-in-layer strategy [4], in which partial method definitions are stored inside a layer. Layers are first-class objects and instances of `Layer`. Defining partial methods and classes is realized by calling library functions. The methods `refineObject` and `refineClass` of the class `Layer` take an anonymous object containing these partial methods as an argument.

The following listing presents a simple example for the usage of ContextJS:

```
1  Object.subclass("MyObject", {
2      m: function(a) {
3         return a * 3
4      }
5  })
6
7  cop.create('LayerA').refineClass(MyObject, {
8    m: function(a) {
9      return cop.proceed(a) + 4
10   }
11 })
12
13 cop.create('LayerB').refineClass(MyObject, {
14   m: function(a) {
15     return cop.proceed(a * 2)
16   }
17 })
18
19 var o = new MyObject()
20 o.m(2) // -> 6
21
22 // Dynamically Scoped Layer Activation
23 cop.withLayers([LayerA], function() {
24     o.m(2) // -> 10
25     withLayers([LayerB], function() {
26         o.m(2) // -> 16
27   })
28 })
29
30 // Global Layer Activation
31 LayerB.beGlobal()
32 o.m(2) // -> 12
33
34 cop.withLayers([LayerA], function() {
35         o.m(2) // -> 16
36 })
```

The layers `LayerA` and `LayerB` provide partial methods for `m` in class `MyObject`. Layers are created using a library function (Lines 7, 13). Their partial method definitions for `m` (Lines 8–10, 14–16) make use of the `proceed` function to traverse a partial method list at run-time. When `m` is invoked with the argument `2` without any layer composition, the call is dispatched to the plain method definition that returns `6` (Line 19). The execution of `m` in the dynamic extent of an activation of `LayerA` (Lines 22–23) is first dispatched to `LayerA`'s partial definition of `m`. The `proceed` expression (Line 9) delegates the call to the next partial method—in this case, to `m`'s default definition—and adds `4` to its result. If several layers are activated, for instance `LayerB` within the dynamic extent of `LayerA` (Line 24), the call is first sent to the

innermost layer (`LayerB`) and then (using `proceed`) passed to the next one. Besides dynamically scoped layer activation, ContextJS supports global activation using the `beGlobal()` method (Line 30). Globally activated layers are active until they are explicitly deactivated using `beNotGlobal()`.

During execution of `refineClass`, the corresponding plain method definition is made layer-aware by replacing it with another function performing layer composition for that method execution and holding a reference to the base method definition. This transformation is done by ContextJS automatically. State (properties) in JavaScript objects can also be layered by defining special JavaScript getter and setter methods in layers[2].

4.4 Globally and Dynamically Scoped Layer Activation

In our library, objects can respond to the message `activeLayers` to compose layers and return a list of active layers. Layer composition is only performed upon method lookup for layered method definitions, so `activeLayers` is only called for a subset of all method invocations. Plain method definitions are not affected. Layering of the `activeLayers` method itself is prohibited, since it would lead to infinite regress. The sequence diagram in Figure 4.8 illustrates a call of `activeLayers` in the invocation process of an adapted `runTest` method.

In the following, we present the implementation of COP's original scoping mechanisms. Our approach covers standard context-oriented scoping mechanisms such as global and dynamically scoped layer activations. The listing below presents an implementation of dynamically scoped layer activation. We use the variable `LayerActivationStack` to keep track of dynamically scoped layer activations[3]. The second variable, `GlobalLayerActivations`, stores the list of globally active layers. The implementation of `activeLayers` uses the `composeLayers` algorithm that recursively composes the layer activation stack and the global activations into an ordered list.

[2] see 11.1.5 Object Initialiser in ECMA-262 [29]

[3] In multi threading environments such as Smalltalk or Java, this variable has to be thread-local, so that different threads will not interfere with each other's layer activations.

```
LayerActivationStack = []
GlobalLayerActivations = []

Object.addMethods({
  activeLayers: function() {
    return composeLayers(LayerActivationStack.length - 1)
  }
})

function composeLayers(index) {
  // 1. Global Layer Activations
  if (index < 0) {
    return GlobalLayerActivations
  }
  var activation = LayerActivationStack[index]
  var layerComposition = composeLayers(index - 1)
  // 2. Dynamic Layer Activations
  if (activation.withLayers) {
    // reject duplicate layer activations
    layerComposition = layerComposition.reject(function(eachLayer) {
      return activation.withLayers.include(eachLayer)})
    return layerComposition.concat(activation.withLayers)
  }
  // 3. Dynamic Layer Deactivations
  if (activation.withoutLayers) {
    layerComposition = layerComposition.reject(function(eachLayer) {
      return activation.withoutLayers.include(eachLayer)})
    return layerComposition
  }
}
```

The layer activation stack is created by the two functions `withLayers` and `withoutLayers` presented in the following listing. The list `layers` is explicitly (de-)activated for the dynamic extent of the closure `func`. Globally and dynamically scoped layer activations are available in ContextJS by default for all objects.

```
function withLayers(layers, func) {
  LayerActivationStack.push(
    {withLayers: layers})
  try {
    func()
  }
  finally {
    LayerActivationStack.pop()
  }
}

function withoutLayers(layers, func) {
  LayerActivationStack.push(
    {withoutLayers: layers})
  try {
```

```
46      func()
47    }
48    finally {
49      LayerActivationStack.pop()
50    }
51  }
```

The list of globally active layers stored in the global variable `GlobalLayerActivations` can be managed by the two methods `beGlobal` and `beNotGlobal`. These functions ensure that a layer cannot be globally activated twice.

```
52  Layer.addMethods({
53      beGlobal: function() {
54          if (cop.GlobalLayers.include(this)) return;
55          cop.GlobalLayers.push(this);
56      },
57      beNotGlobal: function() {
58          var idx = cop.GlobalLayers.indexOf(this)
59          if (idx < 0) return;
60          cop.GlobalLayers.removeAt(idx);
61      }
62  })
```

4.5 Instance-specific and Structural Layer Activation

In Section 4.1.2 we motivated the need for new scoping mechanisms to leverage COP to new application domains, such as graphical object structures in Morphic. Contrary to the two scoping strategies mentioned above, structural and instance-specific scoping cannot be implemented in a generic way but require domain specific modifications. To demonstrate the flexibility of our open implementation, we implement these new scoping strategies for our Morphic scenario.

To change layer scoping for all graphical objects, we implement a new composition mechanism in class `Morph`. The implementation of instance-specific layer activation is straightforward: morphs provide a fixed layer composition that is returned when `activeLayers` is called. The list of layers is managed using accessors `(get|set|add|remove)WithLayers`. The following listing shows the implementation of `activeLayers` and two

accessor methods for the `Morph` class.

```
1  Morph.addMethods({
2    // object-specific layer composition
3    activeLayers: function () {
4      return this.getWithLayers()
5    },
6    // accessor methods
7    getWithLayers: function() {
8      if (! this.withLayers) return []
9      return this.withLayers
10   },
11   setWithLayers: function(layers) {
12     this.withLayers = layers
13   },
14   // other accessor methods such as addWithLayer, removeWithLayer are omitted here
15   // ...
16 })
```

Instance-specific layer activation can be extended to consider structural information of object graphs; in our case, the *submorphs-owner* relationship within a scene graph. The method `structuralLayers` of the next listing recursively walks up the owner hierarchy (as shown in Figure 4.5) and collects all instance-specific layer activations. To prevent multiple execution of a partial method, the algorithm also ensures that—like in the dynamically scoped layer activation—a layer is included only once in the result (Line 7). Since this implementation of structural layer activation subsumes instance-specific layer activations, we implement `activeLayers` for all graphical objects that allow using both strategies.

```
1  Morph.addMethods({
2    structuralLayers: function () {
3      var layers = this.getWithLayers()
4      if (this.owner) {
5        var ownerLayers = this.owner.structuralLayers()
6        // reject duplicate layer activations
7        return layers.concat(ownerLayers.reject(
8          function(eachLayer){return layers.include(eachLayer)}))
9      }
10     return layers
11   },
12
13   activeLayers: function () {
14     return this.structuralLayers()
15   }
16 })
```

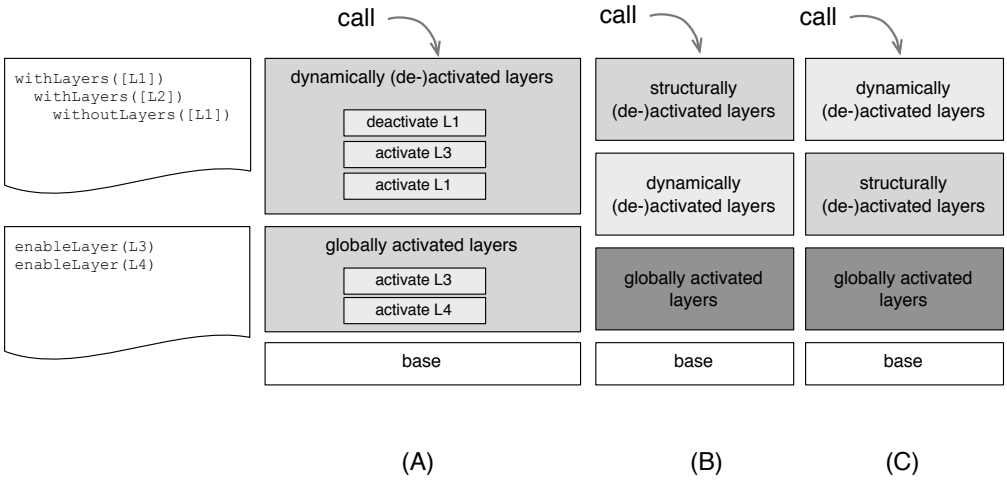


Figure 4.9: Order strategies for different layer composition kinds.

The scoping strategies presented above allow us to implement the connector example as described in Section 4.8.2. To fully support the default and new scoping mechanisms we have to combine them during layer composition.

4.6 Composition of Layer Activation Strategies

Each layer activation strategy can produce a layer specific composition. When these are used in combination, we need precedence rules. These rules can also be implemented in the `activeLayers` method, which we will exemplify in the following. The next listing shows a combination of (the default) dynamic extent-based scoping with structural (and therefore also instance-specific) scoping. It includes dynamic and global layer activation mechanisms into one composition.

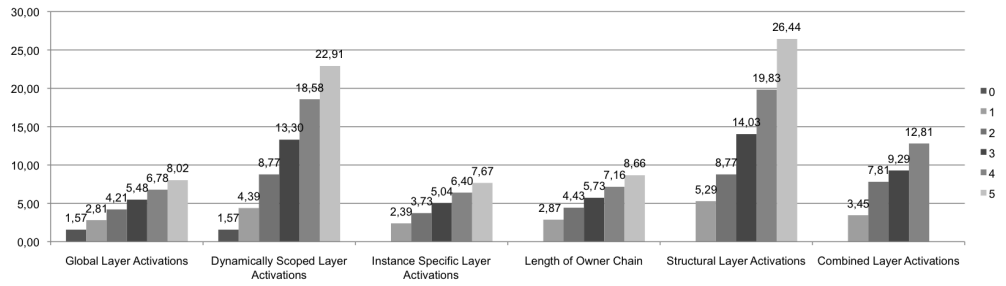
```
Morph.addMethods({
  activeLayers: function () {
    var defaultLayerActivations =
      composeLayers(LayerActivationStack.length - 1)
    var structuralWithoutDefaultLayers = this.structuralLayers().reject(
      function(eachLayer){
        return defaultLayerActivations.include(eachLayer)})
    return defaultLayerActivations.concat(structuralWithoutDefaultLayers)
  }
})
```

If differently scoped layers do not refine the same method, the execution order does not need any further modification. Semantic conflicts occur if a method is layered by dynamically and globally activated layers at the same time. In this case, we need precedence rules avoiding ambiguities. The precedence rule applied in our example proceeds from structurally activated layers to dynamically and globally activated layers, as depicted in Figure 4.9 (B).

The implemented layer composition should provide developers with a coherent metaphor to better understand the layering behavior. Dynamically scoped layer activation follows a stack-like discipline [21]; the stack metaphor as shown in Figure 4.9 *(A)* can be stretched to include global layer activations as well:

1. The base layer (every class and method not in a layer) lies at the bottom of the stack,
2. global layer activations come next,
3. dynamically scoped layer (de-)activations are pushed onto the top.

In this layer composition strategy, dynamically scoped layer deactivations can override global layer activations; i. e., globally activated layers can be deactivated by the dynamically scoped `withoutLayers`. However, depending on the application, other ordering rules may be required as well. Depending on the domain, it may make sense to change the layer composition so that dynamic layer (de-)activations come after the instance specific and structural layer (de-)activations as shown in Figure 4.9 (C). Thus, we found no general domain-independent solution for composition



	ops	time	ops / ms	overhead
plain method	104,857,600	1,522	68,895	0.03
wrapped method	819,200	1,567	523	3.94
activeLayers	3,276,800	1,589	2,062	1.00

Figure 4.10: Micro-benchmark results of various ContextJS `activeLayer` implementations. The chart displays the execution time of the various layer scoping mechanisms relative to the execution time of an empty `activeLayer` method; higher numbers are worse.

ordering. Instead, we allow developers to change the default ordering for their application-specific needs.

By default, classes in ContextJS are open to layered adaptations. ContextL [21], the first COP language, prohibits default adaptation but requires classes to be declared adaptable. ContextJS provides the inverse feature and uses the open implementation to explicitly declare a class as non-adaptable. This encapsulation can be achieved by implementing an empty layer composition. Therefore, a class or object can override the `activeLayers` method to return an empty list, with the consequence that its methods are executed without any adaptations. The decision to move the layer composition into objects hands back control over their method dispatch to them: a context can still change its behavior, but only if the class or object controls its own adaptation.

4.7 Performance Observations

Language support for dynamic adaptation requires additional lookups at run-time, which affect performance. To measure the actual overhead of

layered method definitions, we adopted a set of COP micro-benchmarks [4]. We ran the benchmarks on a MacBook Pro equipped with MacOS X 10.6.4, 4 GB RAM, and 3.06 GHz Intel Core 2 Duo. We used Google Chrome (Dev 5.0.375.99) since it supported the fastest JavaScript engine at the time of measurement[4].

In the benchmark, we activate zero to five layers with the different layer activation mechanisms and measure their execution time. All layers provide a partial definition for a method `m` and proceed to the next layer.

The execution of `m` is applied in a loop for two seconds to avoid startup noise within the measurement. Based on these data, we compute the ratio of executions per millisecond. The results are shown in Figure 4.10 as a chart of performance overheads of various layer activation mechanisms relative to an empty `activeLayer` method. The absolute results of each benchmark are the layered method executions per time (ops/ms), which are compared relative to the method execution of an empty `activeLayers` method which produced approximately 2,000 ops/ms. As shown in the table in Figure 4.10, a method with an empty method body executes approximately 70,000 ops/ms. This relatively high number of operations is a result of aggressive optimization techniques applied to the JavaScript compiler.

In addition to the naïve plain-method-based implementation, we implemented the benchmark using *method wrappers* provided by the *prototype.js*[5] library. It supports the definition of methods that can be wrapped around any other method, much like partial methods. Since method wrappers make similar use of meta-programming and are a commonly used for run-time behavioral adaptation they should be a fair reference point. The wrapper implementation produces approximately 500 ops/ms and with that is four times slower than an empty `activeLayers` but still faster than our layer activation mechanisms.

We can see that there is a considerable performance overhead in methods adapted by ContextJS. The performance of these methods decreased when no layer is activated and further decreases with each additional layer

[4]Since the micro-benchmark results depend on the performance of JavaScript virtual machines, our results might vary on different Web-browsers.

[5]`http://www.prototypejs.org/` (version 1.6)

activation. The performance of structural layer activation depends not only on the number of active layers as shown in the 5th group of the chart in Figure 4.10. It is also affected by the depth of the owner hierarchy (see the 4th group) that has to be traversed even if no layer activations are eventually found. Our benchmarking results show the need for performance improvements in future work. They can be realized by refactoring the layer activation methods to use less expensive expressions. Replacing recursion with iteration, collection functions with `for` loops and introducing caching techniques would probably speed up our implementation.

However, these performance overheads of using ContextJS only affect the performance of refined methods. This differs from other approaches to adaptation such as library-based AOP implementations for JavaScript [119] that wrap every method. These approaches slow down the whole system by a factor of five even when AOP features are not used. In our approach, plain method definitions are executed at full speed, so it depends on the usage of ContextJS how the performance of real programs is affected.

4.8 Scoping Behavioral Adaptations in Lively Webwerkstatt

This section shows how the examples from Section 4.1.2—the test runner adaptation and connectors—can be implemented using the various scoping mechanisms provided by ContextJS.

4.8.1 Test Runner Adaptation

In the following, we present a ContextJS-based adaptation of Lively Kernel's test runner. Its base implementation does not measure the execution time of test cases and individual test runs. This execution time should not be logged every time a test case is executed, but only when it is part of a single test run. The execution of each test is the responsibility of the class `TestCase`; without COP, context information would have to be passed (as parameters or in instance variables) to many test cases and their test methods. With

ContextJS, the desired behavior can be modeled as dynamically scoped layer activation.

For the implementation, we first separate the time measurement and logging concern from the remaining test runner implementation into a layer.

Second, we scope the new layer that should only be active when a user explicitly selects and executes a single test.

The actual behavioral adaptation is defined in layer `TimeEachTestLayer`. As shown in the (extended) class diagram in Figure 4.3, the layer refines the classes `TestCase`, `TestResult`, and `TestRunner`,

- adapting existing behavior such as the `runTest` method in class `TestCase`,
- adding new methods such as `getSortedTimesOfTestRuns` in the class `TestResult`, and
- adding new state such as the property `timeOfTestRuns` also in the class `TestResult`.

```
1  cop.create("TimeEachTestLayer");
2
3  // we do not adapt existing behavior in TestResult
4  // but store our interim results there
5  TimeEachTestLayer.refineClass(TestResult, {
6
7    setTimeOfTestRun: function(selector, time) {
8      if (!this.timeOfTestRuns)
9        this.timeOfTestRuns = {};
10     this.timeOfTestRuns[selector] = time;
11   },
12
13   getSortedTimesOfTestRuns: function() {
14     var times = this.timeOfTestRuns
15     if(!times) return;
16     var sortedTimes = Object.keys(times).collect(function(eaSelector) {
17       return [times[eaSelector], eaSelector]
18     }).sort(function(a, b) {return a[0] - b[0]});
19     return sortedTimes.collect(function(ea) {return ea.join("\t")}).join("\n")
20   }
21 });
22
23 TimeEachTestLayer.refineClass(TestCase, {
24   runTest: function(selector) {
25     var start = (new Date()).getTime();
```

```
26      cop.proceed(selector);
27      var time = (new Date()).getTime() - start;
28      this.result.setTimeOfTestRun(this.currentSelector, time)
29    },
30  });
31
32  // after executing all test methods the test runner sets its final result
33  // we use this hook to display our result
34  TimeEachTestLayer.refineClass(TestRunner, {
35    setResultOf: function(testObject) {
36      cop.proceed(testObject);
37      var msg = "TestRun: " + testObject.constructor.type + "\n" +
38        testObject.result.getSortedTimesOfTestRuns();
39      WorldMorph.current().setStatusMessage(msg, Color.blue, 10);
40    },
41  })
```

Since `TimeTestLayer` refines only the method `runSelectedTestCase` and that adaptation should be active for every execution of `runSelectedTestCase`, we can safely activate this layer globally. The `runSelectedTestCase` adaptation's purpose is to activate the layer `TimeEachTestLayer` in the dynamic extent of the `proceed` statement (Line 8 in the following listing).

```
1  cop.create("TimeTestLayer")
2  TimeTestLayer.beGlobal()
3
4  TimeTestLayer.refineClass(TestRunner, {
5
6    runSelectedTestCase: function() {
7      cop.withLayers( [TimeEachTestLayer], function() {
8        cop.proceed()
9      })
10   }
11 })
```

The layered method execution of `runTest` is shown in the sequence diagram of Figure 4.8:

1. Only those methods that have behavioral adaptations are instrumented, so the execution of most methods in a system is not affected.
2. Before executing the actual method, the object computes the active layers for that message send `(TimeEachTestLayer, TimeTestLayer, BaseLayer)`.
3. Partial methods can be traversed with `proceed` statements.
4. The globally activated `TimeTestLayer` does not define a partial method for `runTest` (see Figure 4.3), so it is ignored and the partial method proceeds directly to the base implementation of `runTest`.

The test runner example demonstrates the usage of two default layer activation strategies: global activation and dynamically scoped activation. The following example shows how the new scoping mechanisms can be used.

4.8.2 Connector

We have motivated the need for a new instance-specific layer scoping mechanism by developing a connector line for two graphical objects (morphs) that is updated when one of the morphs moves (as shown in Figure 4.6). We have shown an implementation of such scoping mechanisms in Section 4.5. To demonstrate the usage of instance-specific scoping mechanisms for modeling node and connector roles with layers, we implement the example in Lively Kernel.

Instance-specific Layer Activation First, we define `NodeLayer`, which adapts the method `change` of class `Morph` (see Figure 4.11). Each node knows its `connectors` and updates them when moved. The connector role is also modeled as a layer that adds a new `updateConnection` method used by the nodes.

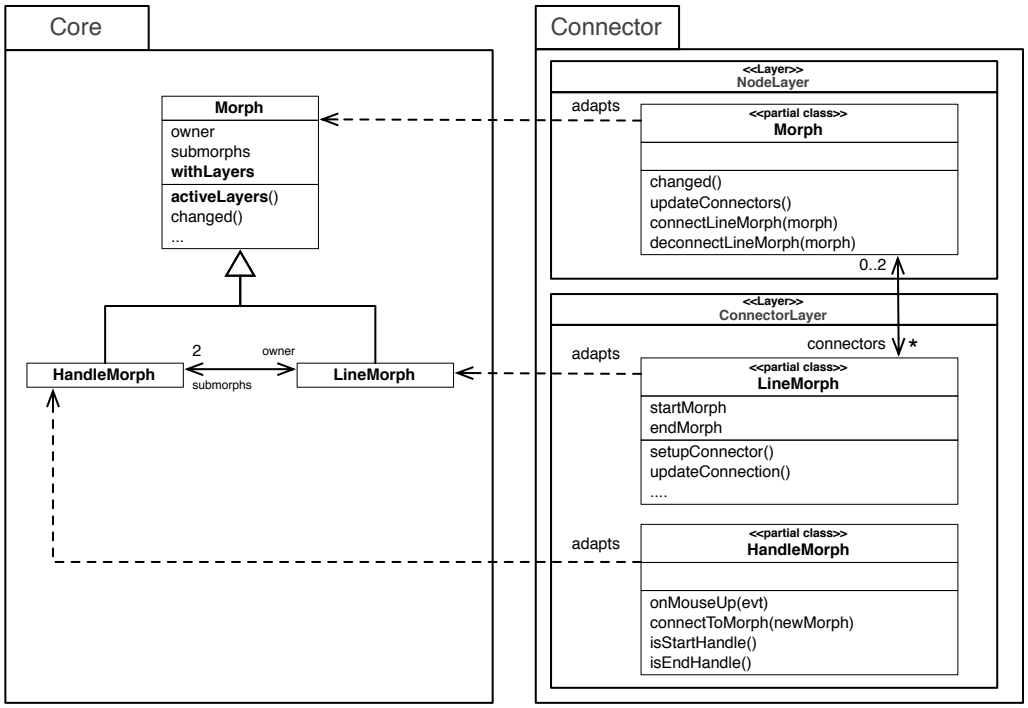


Figure 4.11: Adaptation of classes of the Lively Kernel base system in the connector module.

```
1  NodeLayer.refineClass(Morph, {
2    changed: function() {
3      cop.proceed()
4      this.updateConnectors()
5    },
6    updateConnectors: function() {
7      this.connectors.each(function(ea) {
8        ea.updateConnection()
9      })
10   },
11   ...
12 })
13
14 ConnectorMorphLayer.refineClass(LineMorph, {
15   updateConnection: function () {
16     // ... algorithm that computes
17     // new start and end position
18   },
19   ...
20 })
```

The actual instance-specific layer activation, which lets a `LineMorph` dynamically play the role of a connector, is activated by using the `setWithLayers` method, which associates a list of layers with an instance. The same construct is used to let other existing instances of `Morph`, such as rectangles, ellipses, or text fields, play the orthogonal role of a node. This instance-specific layer activation is used in the method `connectToMorph` (Line 33) when the connector is linked to a new morph that takes on the role of a node.

```
21 var connector = Morph.makeLine([pt(0,0), pt(100,0)], 1, Color.black);
22 connector.setWithLayers([ConnectorLayer]);
23 connector.setupConnector();
```

Structural Layer Activation The second problem is the behavioral variation that should be active when handles are part of a connector: when dragged onto a new morph, handles should reconnect the connector to that morph (as shown in Figure 4.7). The adaptation of the class `HandleMorph` consists of an adaptation of the `onMouseUp` event handler and the addition of helper methods:

```
24 ConnectorLayer.refineClass(HandleMorph, {
25   onMouseUp: function(evt) {
26     this.connectToMorph(this.findMorphUnderMe())
27     return cop.proceed(evt)
28   },
29   connectToMorph: function(newMorph) {
30     var connector = this.owner
31     if (newMorph) {
32       newMorph.setWithLayers([NodeMorphLayer])
33       newMorph.connectLineMorph(this.owner)
34     }
35     // ... update startMorph and endMorph
36   },
37   ...
38 })
```

Since handles are a part of the graphical structure of the `LineMorph` connector, as shown in Figure 4.5, we can make use of the new structural scoping mechanisms defined in Section 4.5. The behavioral variation of the class `HandleMorph` can be defined in the layer `ConnectorLayer`. The handle's `owner` is a `LineMorph` object; thus, the handle is in the connector's structural scope.

4.9 Summary

In this chapter, we have motivated the need for additional scoping strategies—instance-specific and structural scoping—and have proposed an open implementation for COP layer composition. Often behavioral variations cannot be adequately represented using plain object-oriented language features. Their crosscutting and dynamic nature demands alternative encapsulation and scoping mechanisms. COP meets these requirements by providing layers as an encapsulation mechanism orthogonal to objects, and as a control flow-specific scoping strategy. ContextJS, our COP extension to JavaScript offers an open implementation allowing developers to define domain-specific scoping strategies. We have shown how we can apply ContextJS's layer definition and novel scoping mechanisms in Lively Kernel to adapt tools.

Part III

Evaluation of Lively Webwerkstatt

5 Implementation

This chapter presents selected aspects of the implementation of Webwerkstatt [72]. It shows how tools are implemented at the same level as user objects. Since JavaScript does not natively support full object persistence, we developed our own serialization approach and discuss it by analyzing object representation at different levels. When users change and share objects directly without working at a textual source code level, collaboration tasks like comparing and merging have to be implemented at the object level.

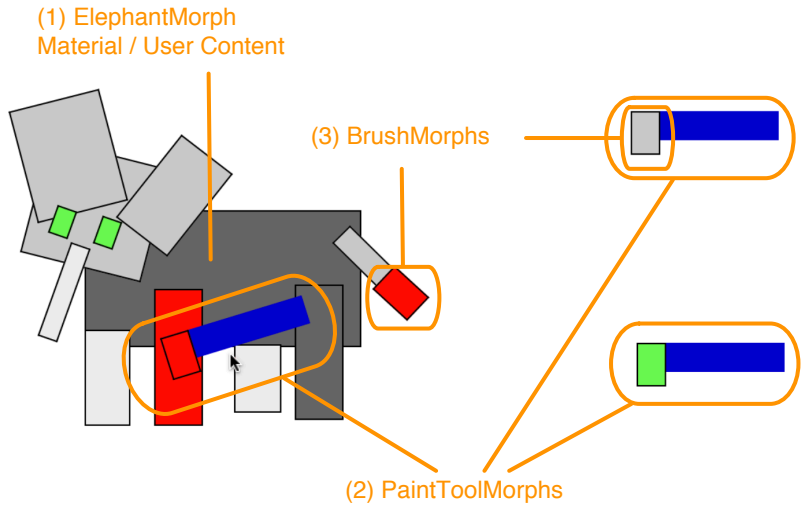


Figure 5.1: Screenshot of the `PaintTheElephant` application based on the `Elephant` and `PaintTool` metaphor from Section 2.2

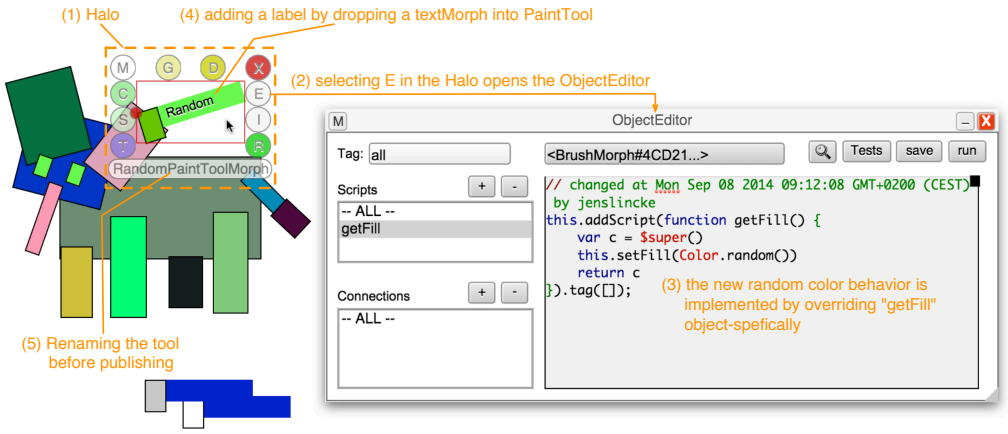


Figure 5.2: Adapting a `PaintTool` while using the PaintTheElephantApp

5.1 Implementation of Tools in Webwerkstatt

All development tools in Lively Webwerkstatt are built in a self-supporting way. Standard tools like the `ObjectEditor`, `PartsBin`, `Inspector`, `Workspace` and more Lively-specific tools like the `SerializationInspector` are all built in similar ways and share both the benefits and problems of being developed in a self-supporting development environment.

5.1.1 Tools and Materials

Since all these tools are too complex for a detailed discussion, we use the *Elephant* object from Section 2.2 to discuss implementation details of using morphs for representing user content and tools alike. Figure 5.1 shows the `Elephant` being modified by a `PaintTool` in a simple but complete application: the user can become creative in coloring the individual body parts of an elephant. The application consists of an `ElephantMorph` (1) and several `PaintToolMorphs` (2). When the paint tool is dragged over a morph, this morph takes the color of the tool's brush. Both the elephant, as user-

created content, and the paint tool are implemented as morphs, allowing users to adapt application content and tools the same way.

This scenario also illustrates that the distinction between tools and materials can be relative: the paint tool's *BrushMorph* (3) can paint every morph and, therefore, it can also paint other *BrushMorphs*, since the tool and elephant both use brushes (as discussed in Section 2.2). The *BrushMorph* stands only for the many other objects in the system that are used by the user application and by system tools likewise. Like many tools in self-supporting development environments, where such reflective manipulation is possible, the programmer of the tool has taken care that it does not accidentally paint itself. But it can paint other brushes without a problem. This example just changes the color of objects and a badly chosen color rarely prevents developers from continuing to work in the system. This danger becomes more apparent when the tool changes object state and behavior, as, for example, the `ObjectEditor` does.

5.1.2 Workflow of Adapting a Tool

Figure 5.2 shows the workflow of adapting a tool in Webwerkstatt. The adaptation of objects starts by opening a *halo* (1). A *halo*, as known from the Squeak Etoys [39] Morphic user interface, allows selecting deeply nested objects and to modify them directly. Individual halo buttons allow users to *grab* (G), *drag* (D), *resize* (R), *transform* (T), *copy* (C), and *delete* (X) the selected object. The halo further allows opening tools like the *object editor* (E), *inspector* (I), and *style editor* (S) for the selected object. Less often used functionality, like publishing an object in the *parts bin*, is available through the *menu* (M). The halo includes also a *name input field* that can be used to identify and rename the object. By clicking on the (E) halo button (2), a fresh `ObjectEditor` for that object will be opened. The object editor can be used to create a new script for that object (3). In this example the `getFill` method will be overridden, so that every time the object is asked for a `fill` color, it will color itself randomly. Now, when the tool asks the `Brush` for its color to paint an object, the brush changes its color, resulting in painting each object with a new random color. To distinguish the new tool from the standard painting tools, the user first adds a label object to it (4) and

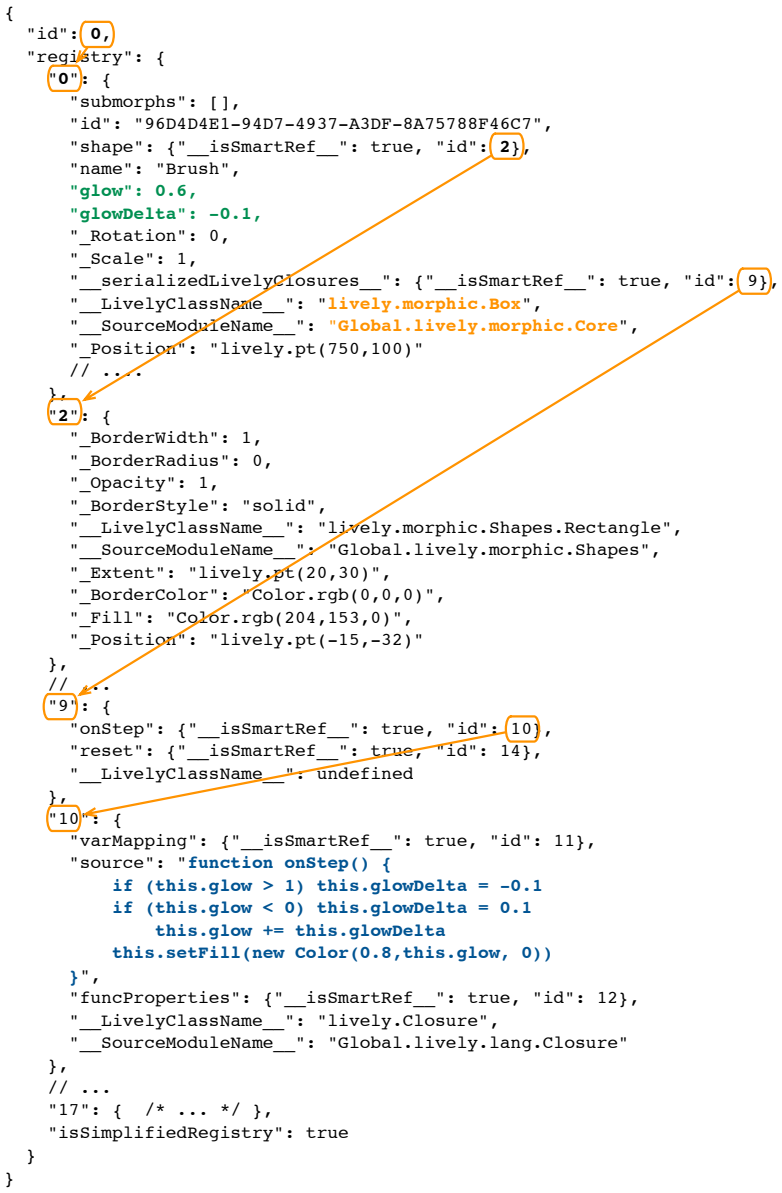


Figure 5.3: Truncated pretty print of a serialized `Elephant` morph showing 10% of the file's contents, in a format that is not meant for users to read or edit directly.

	Object	Name	ClassName	Refs	Size	FullSize
1	root	GlowElephant	Box	456	382	16679
2	submorphs		object	396	25	14490
3	| 1	Head	Box	185	376	6614
4	| | submorphs		object	153	21	5499
5	| | 0	LeftEar	Box	31	363	1120
6	| | 1	Nose	Box	31	363	1114
7	| | 2	RightEar	Box	31	363	1119
8	| | 3	RightEye	Box	30	341	1073
9	| | 4	LeftEye	Box	29	328	1052
10	| 2	Tail	Box	91	393	3555
11	| | submorphs		object	59	5	2406
12	| | 0	Brush	Box	58	413	2401
13	| | __serial...Closures		object	21	58	1099
14	| | onStep		Closure	10	103	552
15	| 0	Foot4	Box	30	346	1065
16	| 4	Foot1	Box	30	363	1085
17	| 5	Foot2	Box	30	363	1084
18	| 3	Foot3	Box	29	350	1062
19	partsBinMetaInfo		PartsBin.PartsBinMetaInfo	31	183	1106
20	changes		object	22	13	760

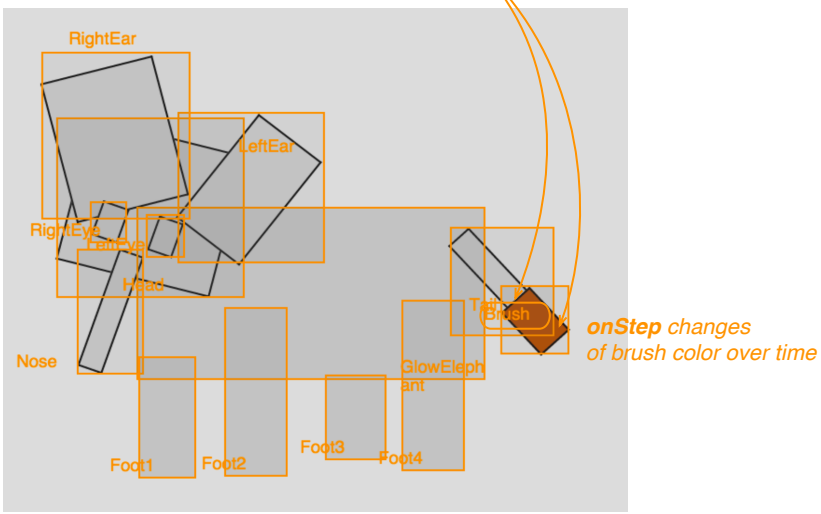


Figure 5.4: Object Serialization of `elephant` morph

gives the object also a more suitable name by typing into the name field halo item (5). The new tool will be automatically persisted when the whole Lively world is saved or when the painting application is (re-)published in the `PartsBin`. A developer can also decide to share it individually by publishing the painting tool alone.

	Object	Name	ClassName	Refs	Size	FullSize	Content
1	root	Brush	Box	57	405	2381	[object Object]
2	moved		boolean	1	4	4	true
3	submorphs		object	1	2	2	[]
4	id		string	1	38	38	58F99AE1-BE19-4D9D-A5E4-163..
5	shape		Shapes.Rectangle	11	274	420	[object Object]
6	| _BorderWidth		number	1	1	1	1
7	| _ClipMode		string	1	9	9	visible
8	| _Opacity		number	1	1	1	1
9	| _BorderStyle		string	1	7	7	solid
10	| position		string	1	20	20	lively.pt(0.0,0.0)
11	| _Extent		string	1	22	22	lively.pt(21.8,31.7)
12	| _BorderColor		string	1	18	18	Color.rgb(0,0,0)
13	| _Fill		string	1	22	22	Color.rgb(204,153,0)
14	| _Padding		string	1	22	22	lively.rect(0,0,0,0)
15	| _Position		string	1	24	24	lively.pt(-15.2,-32.1)
16	halosEnabled		boolean	1	4	4	true
17	registeredForMouseEvents		boolean	1	4	4	true
18	name		string	1	7	7	Brush
19	eventHandler		EventHandler	2	72	79	[object Object]
20	| morph		string	1	7	7	ref 0
21	_ClipMode		string	1	9	9	visible
22	scripts		object	6	5	149	[[object Object]]
23	| 0		TargetScript	5	124	144	[object Object]
24	| target		string	1	7	7	ref 0
25	| selector		string	1	8	8	onStep
26	| args		object	1	2	2	[]
27	| tickTime		number	1	3	3	100
28	derivationIds		object	3	9	85	[526A278A-C7E5-4960-87DD-BF..
29	| 0		string	1	38	38	526A278A-C7E5-4960-87DD-BF7..
30	| 1		string	1	38	38	0F2EB418-99C8-439D-8BB7-0C8..
31	glow		number	1	3	3	0.6
32	glowDelta		number	1	3	3	0.1
33	prevScroll		object	1	9	9	[0, 0]
34	_Scale		number	1	18	18	0.9999999999999984
35	__serializedLivelyClosures__		object	21	58	1097	[object Object]
36	| onStep		Closure	10	103	551	[object Object]
37	| | funcProperties		object	6	67	196	[object Object]
38	| | | timestamp		object	3	69	115	[object Object]
39	| | | | isSerializedDate		boolean	1	4	4	true
40	| | | | string		string	1	42	42	Sun Sep 07 2014 13:06:46 GM..
41	| | | user		string	1	12	12	jenslincke
42	| | | tags		object	1	2	2	[]
43	| | varMapping		object	2	48	55	[object Object]
44	| | | this		string	1	7	7	ref 0
45	| | source		string	1	197	197	function onStep() {\n if..
46	| reset		Closure	10	103	488	[object Object]
47	| funcProperties		object	6	67	196	[object Object]
48	| | timestamp		object	3	69	115	[object Object]
49	| | | isSerializedDate		boolean	1	4	4	true
50	| | | string		string	1	42	42	Sun Sep 07 2014 13:06:46 GM..
51	| | user		string	1	12	12	jenslincke
52	| | tags		object	1	2	2	[]
53	| varMapping		object	2	48	55	[object Object]
54	| | this		string	1	7	7	ref 0
55	| source		string	1	134	134	function reset() {\n thi..
56	distanceToDragEvent		string	1	23	23	lively.pt(34.0,-31.4)
57	_Position		string	1	22	22	lively.pt(-3.7,50.2)

Figure 5.5: Object Serialization of Elephant's Brush Morph

5.2 Object Persistence in Webwerkstatt

When saving a world or individual objects as parts, we employ our own custom serialization approach, because JavaScript's native *JavaScript Object Notation* (JSON) serialization cannot serialize general object graphs.

All objects in Lively have to be serialized for copying or persisting them. The first versions of Lively Kernel (as presented in [50]) serialized all user content as *Scalable Vector Graphics* (SVG) that was directly available through the Web-browser *document object model* (DOM). With this approach the source code of Lively Kernel worlds was dominated by the user interface. Non-visual objects were discarded on serialization, if not explicitly stored in an attribute of a graphical object's DOM object before serialization. Therefore, users had to explicitly take care of serializing their data, if it was not represented as graphical objects, such as text or graphical shapes. This approach produced a very readable and still comprehensible source code representation of a Lively Kernel world. If a world could not be loaded any more, it could still be repaired by manually editing it. However, user data had to be stored explicitly and more complicated object structures could not be serialized at all.

For practical reasons, we replaced this serialization approach with a general, more Smalltalk-like serialization of the whole object graph, producing an object table that could later be deserialized. As the pretty-printed extract of the serialization of the `Elephant` morph in Figure 5.3 shows, this source code representation is not intended for users to edit directly any more. Even though we choose to store the object table in JSON, we as developers treat it most of the time as a binary blob but do not edit it directly. The object table has one root with id "0". Every JavaScript object is represented by plain JSON in the table. Value objects, such as strings, numbers, points, colors and similar objects, are directly stored as attributes. References to other objects are represented by special objects that store the serialization id of the referenced object. The ids in the serialization table are different from object's persistent unique object ids morphs as described in Section 5.2.3. Serialization ids are necessary, because JavaScript objects do not have object ids we could use. Further we did not want to add persisted unique

object ids to all objects in Lively. To speed up serialization we add such ids temporally to objects during serialization and remove them afterwards.

5.2.1 Serialization Example

Getting a grasp of the Elephant's serialization by looking at the produced source code alone is difficult because of its size and its lost structure. The excerpt in Figure 5.3 shows only about 10% of the elephant's source code representation. By using tools we can get this structure and overview back. Figure 5.4 shows a more general view of on the serialization of an `elephant` morph. It was generated by Lively Webwerkstatt's `SerializationInspector` tool. The `SerializationInspector` displays the object graph in form of a tree and computes an object's own size `(Size)` and the number `(Refs)` and accumulated size `(FullSize)` of all subobjects. By filtering the entries below a given `FullSize` threshold, the table shows the general object structure, even though some subobjects might normally not be displayed in the world. As we can see in that visualization, the `Brush` is the only object with some instance-specific behavior `(onStep)`, which doubles its serialization size, compared to body parts of the Elephant. How this tool can be applied to help users cleaning up their worlds and parts is discussed in Section 6.5.

Figure 5.5 shows the unfiltered serialization table for the Elephant's `Brush`. We can see here that Lively serializes the full state of objects, even though some properties do not differ from their default values. Because of this implementation decision, objects' style and behavior are not automatically updated. On one hand this can lead to outdated tools or applications when they are serialized together with a world, but on the other hand it ensures that experiments and draft objects also do not change their behavior.

5.2.2 Serializing Object-specific Behavior in JavaScript

Allowing users to create active content as scriptable objects is a form of unification of state and behavior. Self [122] explored this unification many years ago and our implementation language JavaScript builds on the same underlying concepts.

One difference between the unification of behavior and state in JavaScript and Self is that behavior is not persisted in JavaScript objects. One reason for this is that JavaScript closures cannot be fully serialized, because as security mechanism there is no reflection interface for their variable bindings. So, by design, a JavaScript program cannot introspect the bound variables in a JavaScript function. Further, the JavaScript Object Notation (JSON), which is often used for serializing JavaScript objects, does only serialize plain objects without their functions or a reference to their prototype. JSON has no concept for serializing arbitrary references in an object graph. The JSON serialization mechanism can only serialize dictionaries (plain objects), arrays, numbers, and strings. Objects cannot be referenced but are serialized each time and cycles are not allowed. Pure JSON is, therefore, not suitable for serializing arbitrary object graphs.

We solved these limitations by implementing our own serialization mechanisms as described in Section 5.2. Because current Web-browser technology does not allow us to access the full state of JavaScript functions, we make sure that we define our own persistent object-specific behavior using our "addScript" method:

```
1    addScript: function(funcOrString, optName, optMapping) {
2        if (!funcOrString) return null;
3        var func        = Function.fromString(funcOrString),
4            oldFunction = this[func.name],
5            changed     = oldFunction && oldFunction.toString() !== func.toString(),
6            timestamp   = oldFunction && !changed ? oldFunction.timestamp : undefined,
7            user        = oldFunction && !changed ? oldFunction.user : undefined,
8            result      = func.asScriptOf(this, optName, optMapping);
9        result.setTimestampAndUser(timestamp, user);
10       return result;
11   }
```

We basically do not allow instance-specific behavior to be regular JavaScript closures by making sure (line 3) that the function source code is reevaluated without its bound variables. The `addScript` method ensures that scripts behave the same before and after (de-)serialization.

Without full reflection capabilities, snapshotting of a running system can become complicated. Interrupting and continuing a method execution requires advanced reflection capabilities JavaScript in comparison to Smalltalk is missing. But since the execution model of JavaScript is simple,

e.g. it does not support multiple threads, we do not have to serialize any running code at all. JavaScript is only called by events that always return to the browser-controlled main loop. There is no continuously running parallel method execution in the background. Stepping scripts, a form of green threading used in Lively Kernel, always return regularly so the user interface can be updated. That way they can be stopped and restarted for (de-)serialization as needed.

5.2.3 Cloning and Derivation History

To preserve the invariant in object-oriented systems that all objects have a unique identity, we need to make sure that cloning does not lead to identity conflicts. We implemented this by generating a universally unique identifier (UUID) for the cloned object and its subobjects. This ensures that two objects in the system do not have the same id. The initial version of Lively Kernel used an object counter which conflicted with shared objects (in form of parts) across world boundaries. Using UUIDs in Lively Webwerkstatt solved that problem. However, assigning all objects new ids while cloning deprives comparing and merging algorithms of the possibility to detect object movements after object structure changes [118].

As a contribution of this thesis, we propose to remember object cloning relationships by preserving previous identities. Old identifiers are remembered in a derivation history of each object. This is done for every subobject of that object, too. So, cloning in Lively does not produce an exact duplicate in every aspect. The identifiers and the derivation history of the object and its clone differ. This information is meta-information that should ideally be stored separately [74], e.g. in separate meta-objects like Self's Mirrors [14], so that it does not interfere with domain code. But since JavaScript has no concept of distinguishing properties from meta-information, conventions had to be established and tools made aware of them. Similar to other JavaScript frameworks, we use special naming conventions, e.g. prefixing property names to mark meta-data, making such separation easier to recognize.

When using object cloning for improving robustness in live coding sessions, it can become hard to manually deal with additional redundancy. But

by tracking the derivation history of objects, we can mitigate this problem. For example, when a tool is developed by two different users at the same time, their changes to the tool have to be merged. The objects have new identifiers, because they were copied from the `PartsBin`, so the id cannot be used for identifying corresponding subobjects any more. By capturing the derivation history of whole object structures, we can find matching old and new objects in an objects graph so comparing and merging such complex graphical objects becomes easier, especially when the structure of objects change, e.g. which is a common result when dragging and dropping graphical objects.

5.3 Object Merging as Collaboration Support

Objects can be compared at different levels of abstraction. As shown in Figure 5.6, standard tools can point out the difference in text files, e.g. as in the source code representation of serialized JavaScript objects (4). But since users are not expected to be familiar with this low-level representation of objects, in Lively, object merging can compare JavaScript objects directly (2 and 3), and resolve conflicts based on the visual representation (1). The merging tools utilize the object's derivation history to identify corresponding subobjects in an object graph when comparing objects that were cloned from each other.

5.3.1 Simple Comparison of Objects

When comparing different versions of the same object as shown in Figure 5.8, the identity of object `P1` does not change when modifying it (1). `P1'` is still the same object, but with different state. When comparing these two versions to produce a *diff* (2), the object ids can be used to detect that `S1` changed, `S2` was deleted, and a new object `S3` was added. Without knowing the object identities, the algorithm would not be able to figure this out. Relying only on object structure, a diff algorithm would be able to detect only a state change in the second object of `P1`, which would not help the user in the case of very different states in `S2` and `S3`.

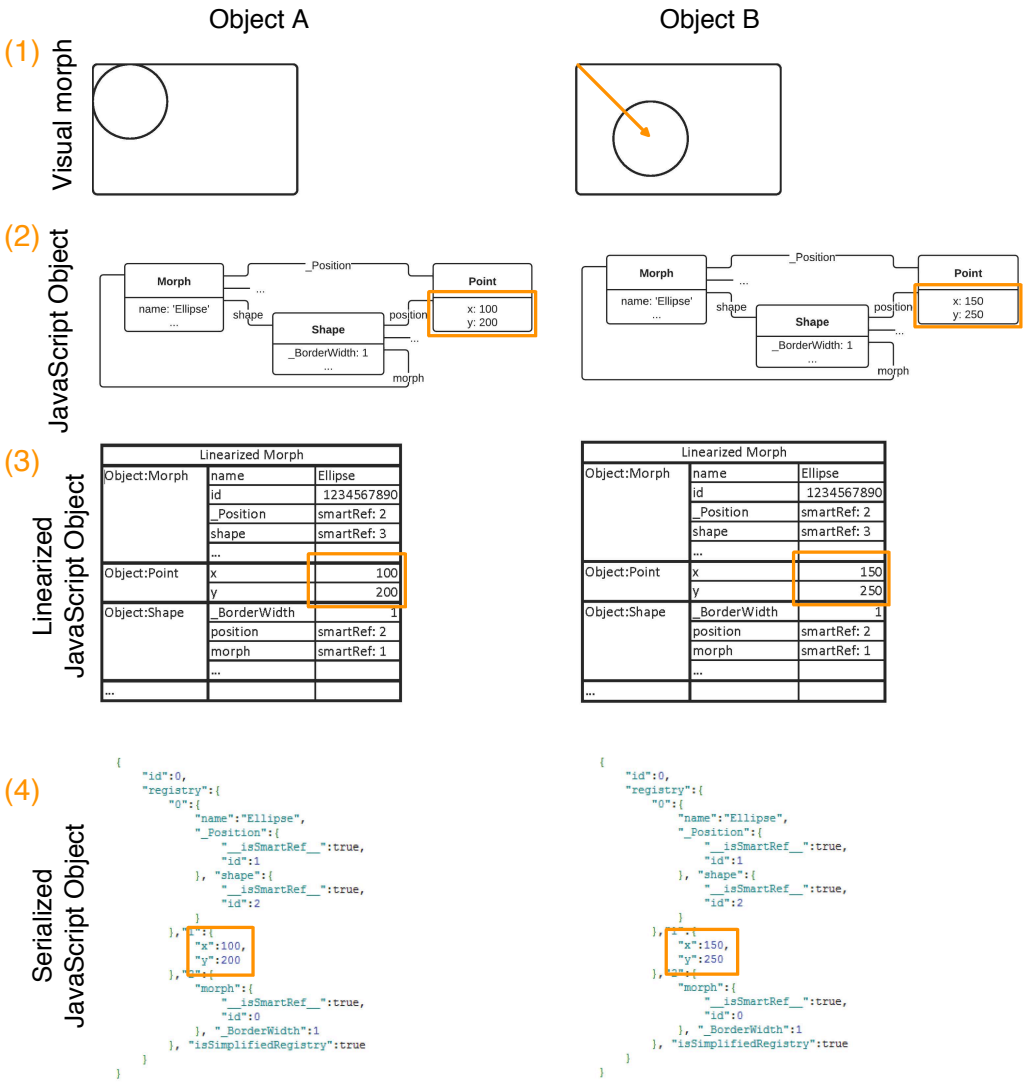


Figure 5.6: Comparing objects at different levels of abstraction [16]

5.3.2 Comparison with Cloned Objects

When an object (`P1`) is cloned as shown in Figure 5.9, the new object `P2` received a new identity, so that both objects can coexist in the same world

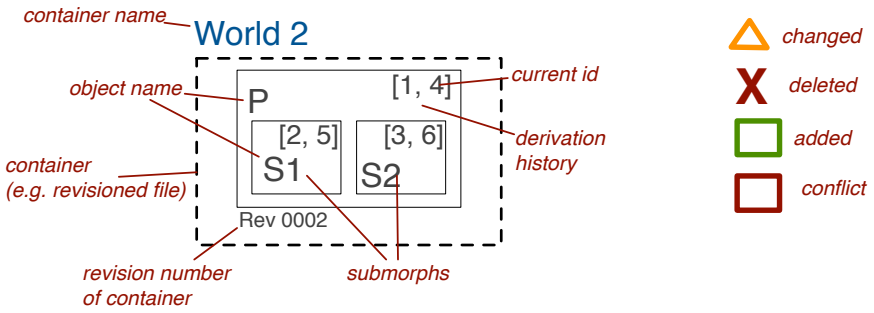


Figure 5.7: Notation used in Figure 5.8

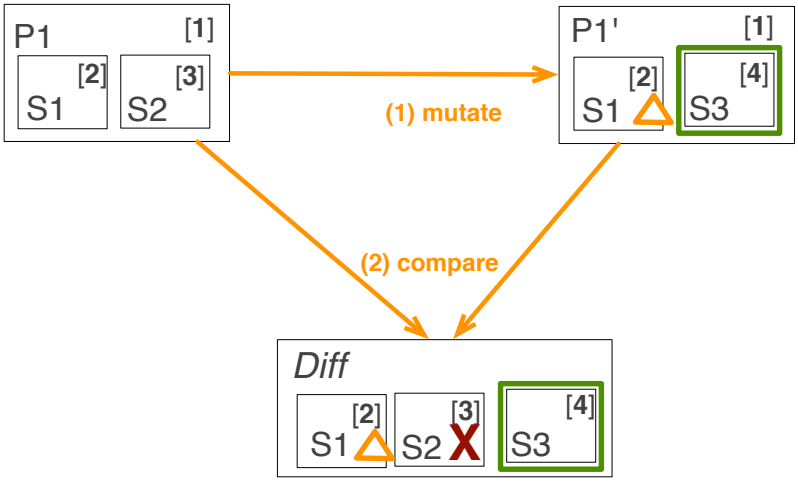


Figure 5.8: Comparing two versions of the same object `P1`. (Notation is explained Figure 5.7)

without identity conflicts. Unlike the scenario in Figure 5.8, the object identities changed and cannot be used for comparing objects any more. But since each object still knows where it was copied from, we can deduce that `P2`'s `S1` was copied from `P1`'s `S1` and changed afterwards. Further it becomes clear that `S2` was deleted, and `S3` added.

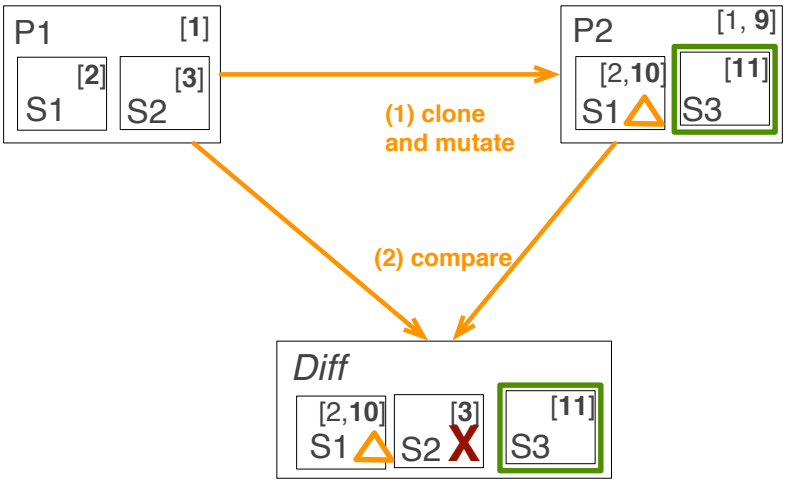


Figure 5.9: Comparing an object `P1` with its mutated clone `P2`. (Notation is explained Figure 5.7)

5.3.3 Merging Objects using the Derivation History

Comparing two objects like `P1` and `P2` can only show where they differ but cannot automatically merge them. If both objects where derived from a shared ancestor, the derivation history can be employed to find that common ancestor and perform a so called three-way diff [103]. As a three-way diff on text files shows which lines were removed, added, changed or are in conflict with each other, a three-way diff on objects recursively shows which properties were removed, added, changed or where there are conflicts. This information can then be used to perform a three-way merge.

Figure 5.10 shows a typical example where such a three-way diff of objects can be used. In that scenario, a part `P1` was created in one world and published in the `PartsBin` (1). It was then copied from the `PartsBin` into a second world (2), where it was edited (3) and published again (4). Meanwhile, the development of `P1` continued in the first world (5), but it comes to a conflict, when trying to publish the modified `P1'` (6). The first world could ignore that conflict and resolve it by replacing the `P2` with its own version, or it could decide to merge object `P1'` with object `P2`.

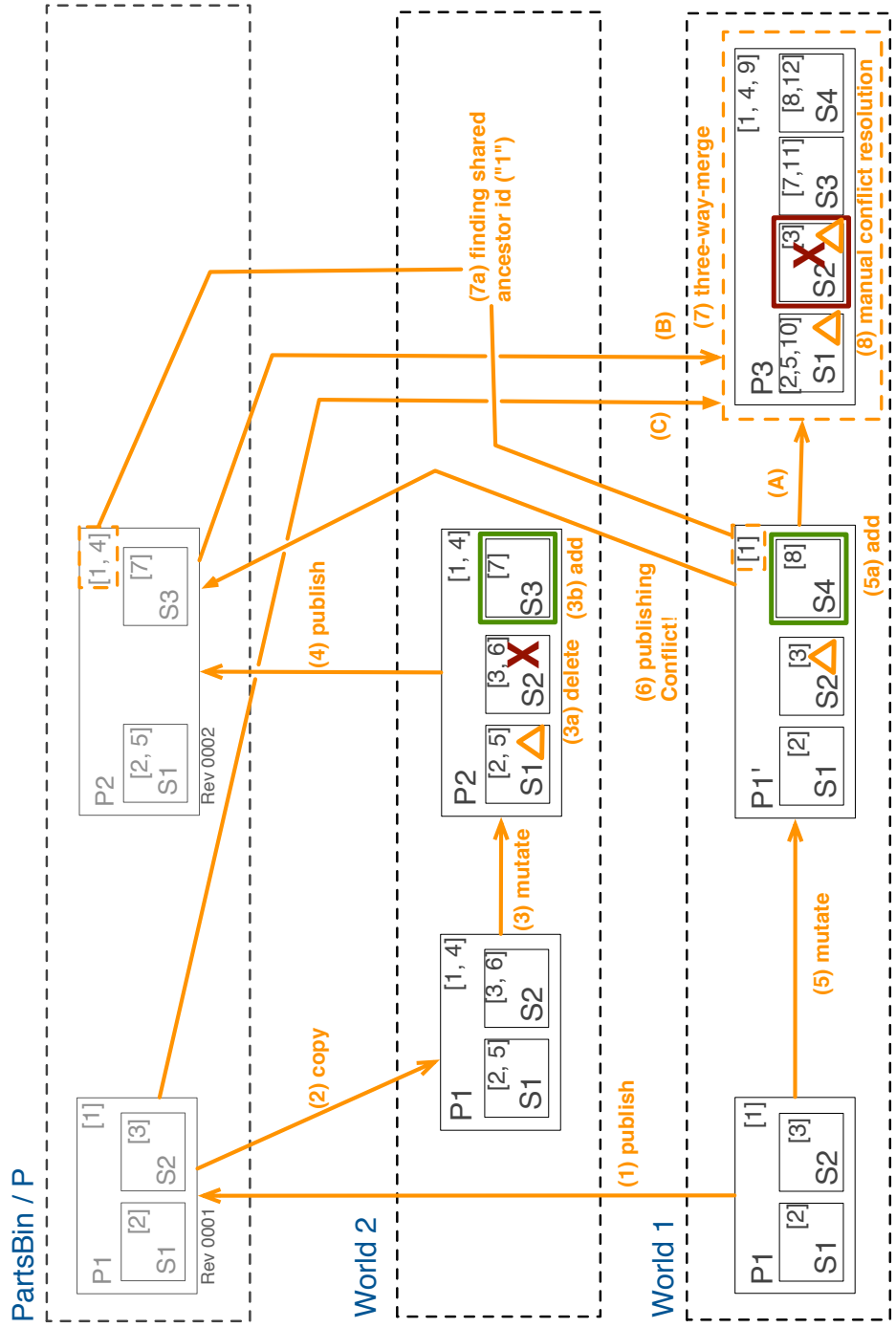


Figure 5.10: three-way merge of a part, development in two worlds and shared via the `PartsBin`

For finding the shared ancestor (C) of object `P1'` (A) and object `P2` (B), we can look at their derivation history. The id of the common ancestor can be found by comparing the derivation history of the two objects. But just having the id does not automatically yield an object (C). By looking at the revision history of the part `P`, we can find the original published version of object `P1` (C) and use it to perform a three-way merge (7).

Similar to a textual three-way merge, independent changes can be automatically merged, e.g., we keep `B`'s changed `S1`, because (A) did not touch it. We further can add `S3` and `S4`, because they do not conflict either. But the change `S2` (A) does conflict with (B) deleting `S2`. Like in textual diffing, such a conflict has to be propagated to the user and solved manually.

5.4 Summary

In this chapter, we have shown how tools in Lively Webwerkstatt are implemented as normal objects and how they can be modified in the same way. Tools are built as user-editable graphical objects (morphs), that can be adapted at run-time and shared with others by publishing them in a parts bin. Since objects are changed in a prototypical way, users do not have to fall back on a source code level any more. This source code is produced by our serialization mechanism. Unlike the standard JavaScript object serialization (JSON), this allows us to serialize complete object graphs. We further also serialize instance-specific behavior, which is the foundation of our prototypical workflow. Since sharing objects in a parts bin allows users to collaboratively work on objects, we have to provide means of object comparison and merging. The object's and all its subobject's identity has to change after cloning, so that there are not two objects with the same identity in the system. We have shown how tracking the derivation history of all objects can help while directly comparing two objects to produce a more meaningful diff. Finally, we have discussed how the derivation history can be used to find a shared ancestor that is needed to perform a three-way merge, which can combine two separately changed objects automatically when there are no conflicts.

6 Evaluation and Discussion

This chapter discusses how *parts* and *layers* were applied.

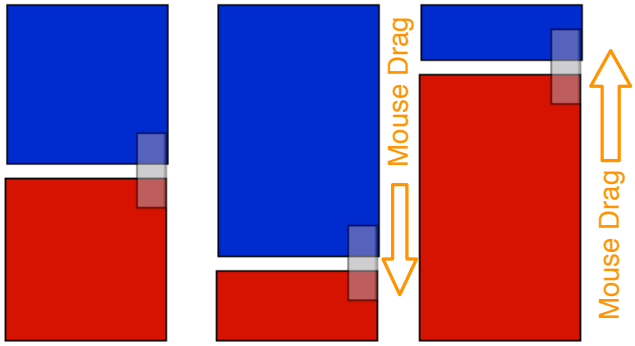


Figure 6.1: A SplitterMorph adjusts the extent of two adjacent morphs when dragged

6.1 Splitter Morph as an Example of Adaptation through Object Composition

This first example is a real use case from Webwerkstatt. It shows what kind of simple morphs can be created using the scripting approach. The `SplitterMorph` was developed to adjust the extent of two objects. As shown in Figure 6.1, when the splitter is dragged down, the upper blue morph gets larger and the red morph gets smaller. When it is dragged up again, the process is reversed. The interesting aspect of the `SplitterMorph` is that it

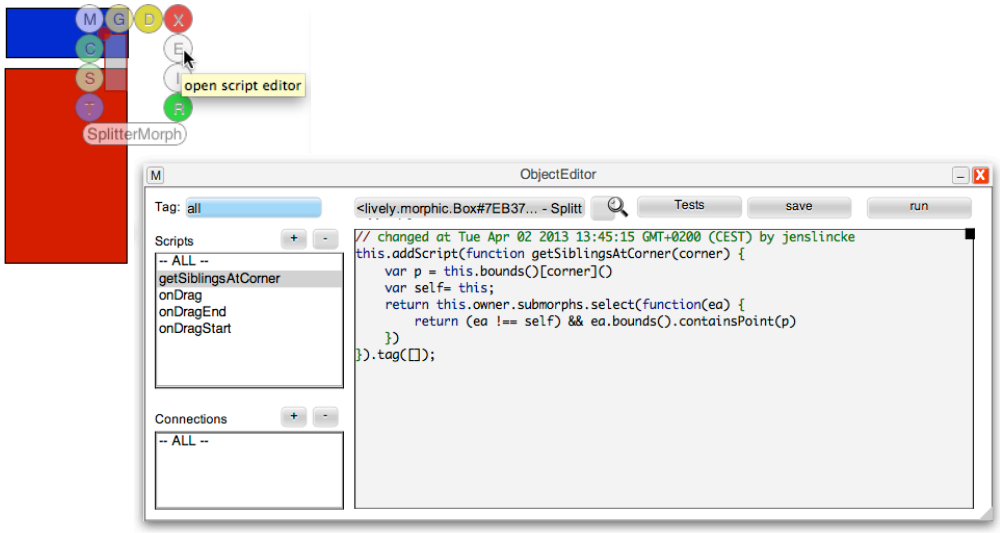


Figure 6.2: Editing the SplitterMorph with Halo and ObjectEditor

does not need an explicit configuration. It works by automatically detecting the objects next to them and connects them like a patch. By looking at the bounds of the object, it can detect them and adjust them appropriately. The `getSiblingsAtCorner` script in the *ObjectEditor* in Figure 6.2 shows some of its implementation. The developers realized that they built something useful and published their morph in the `PartsBin` for others to experiment with and reuse.

6.1.1 Evolving the Inspector by Adding the Splitter Morph

At a later stage, some users noticed the new `SplitterMorph` and realize that the inspector, a core development tool, misses this feature and decide to change it. Figure 6.3 shows the work flow: (1) they open a standard inspector and drag the `SplitterMorph` to the correct position. (2) They test the new behavior by interacting with the Inspector and (3) publish it again to the `PartsBin`. From then on, when users open an inspector, they can adjust it using the `SplitterMorph`.

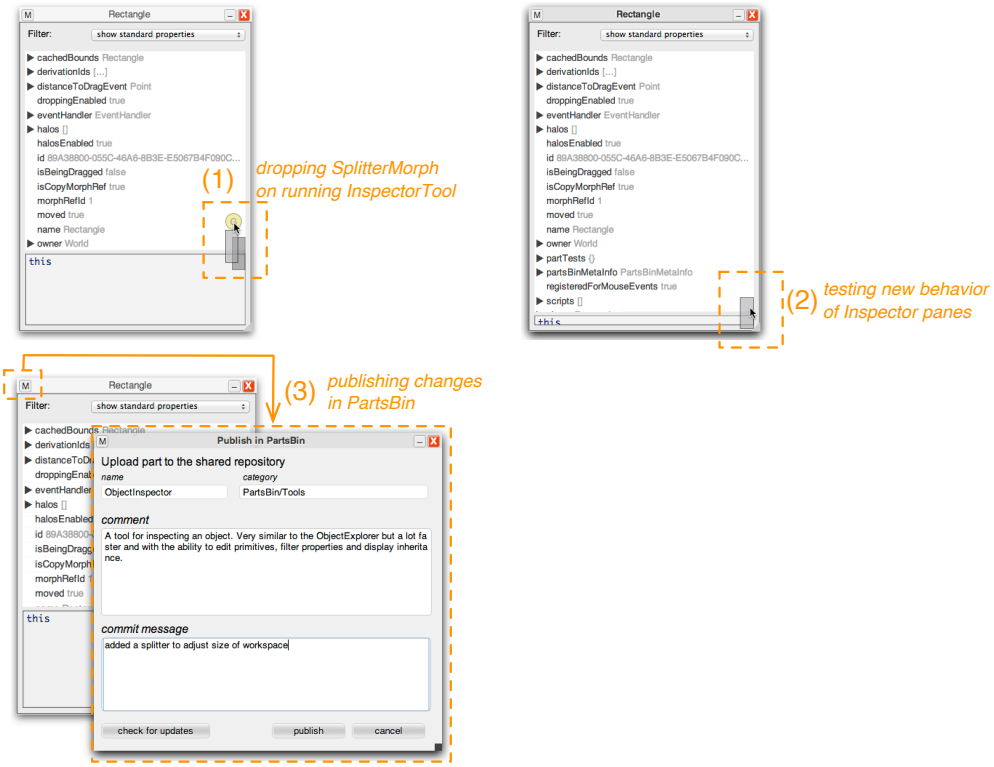


Figure 6.3: Adopting the inspector tool using a SplitterMorph

This tool adaptation did not involve programming. It demonstrates that publishing operations does not always require to actually type code in Lively. Users can contribute in a meaningful way by changing the style or layout of objects or editing the text of labels without having to touch source code. It also demonstrates that object composition can be a powerful mechanism to develop objects.

6.2 Developing with ContextJS in Webwerkstatt

We have gained experience using development layers when evolving Lively Kernel in our self-supporting development environment Webwerkstatt [72]. This section shows three examples that illustrate such usages.

Figure 6.4: A workspace with the text coloring source code and an example instance.

6.2.1 Text Coloring

An example of a behavioral adaptation of the base system is text coloring via keyboard shortcuts. Figure 6.4 shows the complete source in a workspace and the text object, on which the new feature was interactively tried out. The layer refines the keyboard processing method and will color the current selection if specific shortcuts are pressed.

This is an example where shaping the tools happens directly while they are used. The example looks simple at first, but it shows how easy it is to produce meta-circular loops between tools that change their own code at run-time as described in Section 2.1.2. The new text coloring behavioral adaptation affects all `Text` morphs in the system and the `Workspace` tool uses also a `Text` morph. During development, the activation of the `TextColorLayer` should be restrained in a way that prevents it from changing the behavior of the `Workspace`. That way making an error while developing the code of `processCommandKeys` will not affect the workspace and the developer can continue editing.

As a result, layers allow developing features in a controlled way and using them later in the entire system.

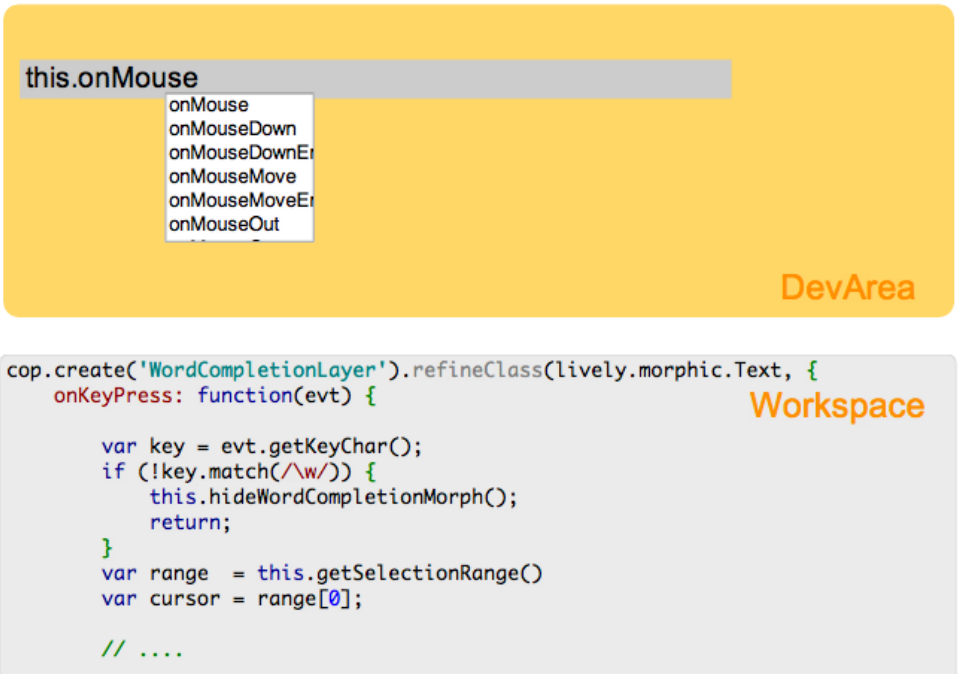


Figure 6.5: A snapshot of developing auto-completion of source code, showing a workspace and an area where the new code is activated.

6.2.2 Developing Auto-completion

A more complex example is the development of source code completion for the development environment as shown in Figure 6.5. It adapts over 10 methods in two classes, although the figure shows only the area where the new behavior can be tried out and the workspace with the code under development. By using layers, the feature that could interfere with the workspace itself can be developed locally and tried out in other places by changing the scope of the layer.

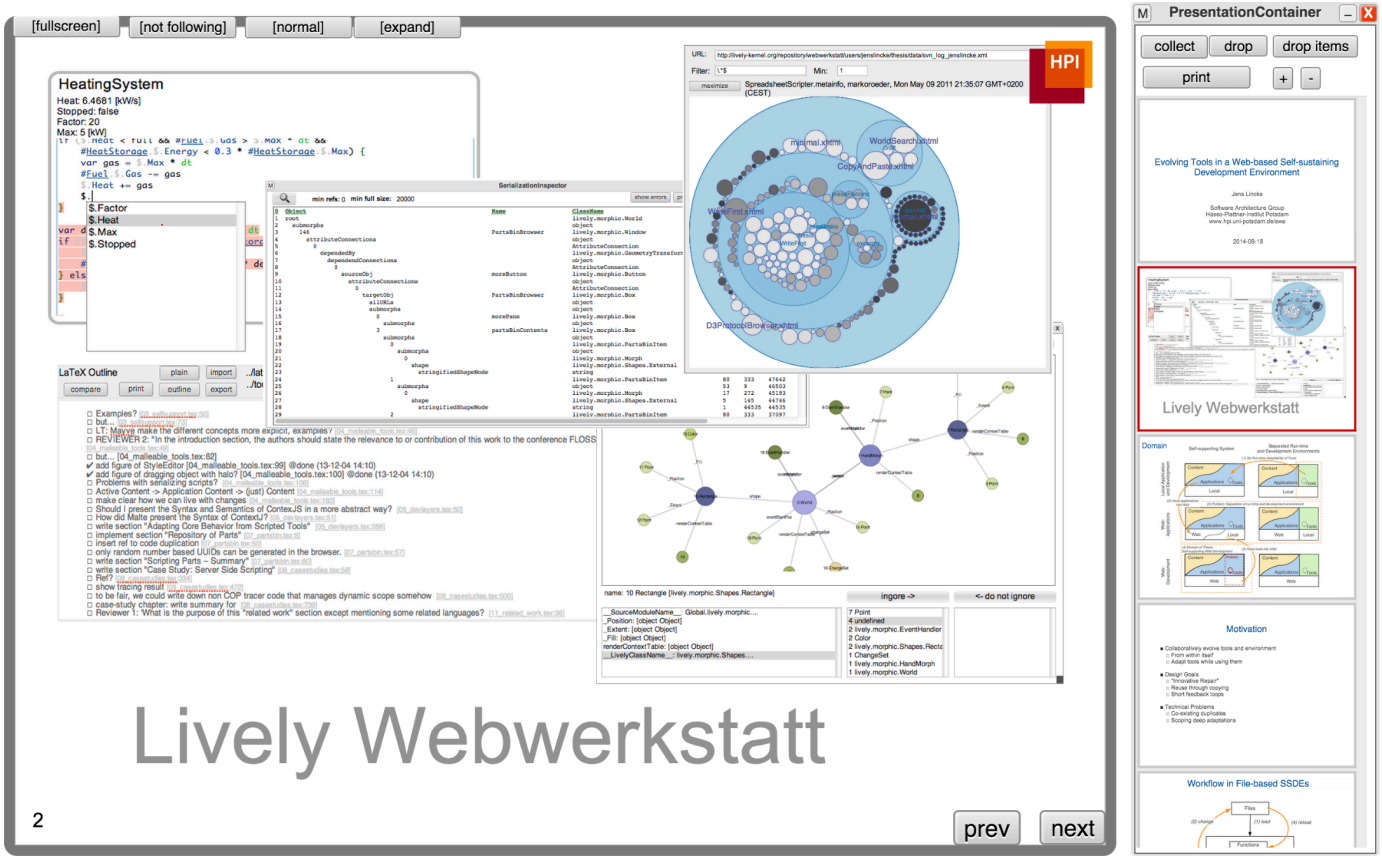


Figure 6.6: ProjectorMorph showing a slide of a presentation about Lively Webwerkstatt.

6.3 Run-time Evolution of Tools: URLLister Example

We use Lively Webwerkstatt as an authoring environment for all kinds of media, not just JavaScript programs. A project `ProjectorMorph` allows us, for example, to give presentations directly from within Lively (as shown in Figure 6.6). After interactively creating an overview slide with various screenshots, we were interested in a list of URLs of the images being used, because they should be reused in another place. Instead of going through each morph and manually copying the image URL into a list, we decided to write a little script in a workspace that does exactly this:

```
1 var urls = []
2 this.withAllSubmorphsDo(function(eaMorph) {
3     if (!eaMorph.getImageURL) return;
4     urls.push(eaMorph.getImageURL())
5 });
6 urls.join("\n")
7 // ->
8 // ../screenshots/131204_LaTeXOutliner_ToDo_Extraction.png
9 // ../screenshots/110818_WorldSerializationGraph1.png
10 // ../screenshots/140130_SimulationPart_CustomAutoCompletion.png
11 // ../screenshots/130817_SerializationInspector_03.png
12 // ../screenshots/130128_ChangeLogVisualization_05.png
```

The variable `this` (Line 2) is bound to the slide that contained the images. Not every morph is an image and understands `getImageURL`, so we ignore them (Line 3). We are interested in a textual list, so we convert the `urls` array into a string (Line 6).

6.3.1 From Script to Graphical Tool

After writing this script, we realized that it is useful and that we wanted to publish it so we (and others) can reuse it later. So, we published the workspace with the script directly in the parts bin. Then two things happened: first, we wanted to make this script a little more user friendly and second, we generated new requirements while using the tool.

As shown in Figure 6.7, we added a `Log` text field (1) to the workspace (2) and also added a `Splitter` morph (3), so the size can be adjusted (as dis-

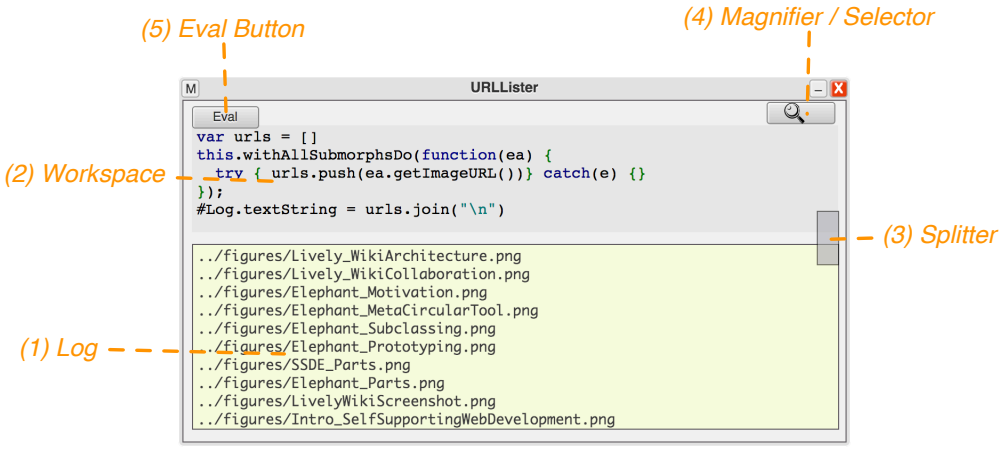


Figure 6.7: Example of an URLLister tool.

cussed in Section 6.1.1). At this point the script slowly changes its character. First it was a pure text snippet without a specific user interface. Then it evolves gradually into an interactive graphical tool. The textual script is still visible, but the `Magnifier` (4) allows it to target any visible morph and the `Eval` button (5) executes the workspace without keyboard interaction. We do not need the script in the workspace to be directly executed by the user any more. Therefore, we could go on refactoring and hide the workspace by making the script instance-specific behavior of the tool. But by not hiding that code, the user is still invited to experiment with it.

At a later point, we wanted to see the URLs not in a list, but directly printed on the image morphs. So, we modified the `URLLister` to `show` (highlight with red corners) all images and display their URL as a label (as shown in Figure 6.8).

6.3.2 Refining Base System Behavior

While using the `show` function in our `URLLister` script, we wondered why the highlighting corners disappear after just 3 seconds. This is not long enough for reading the URLs on the labels. Since the `show` function could

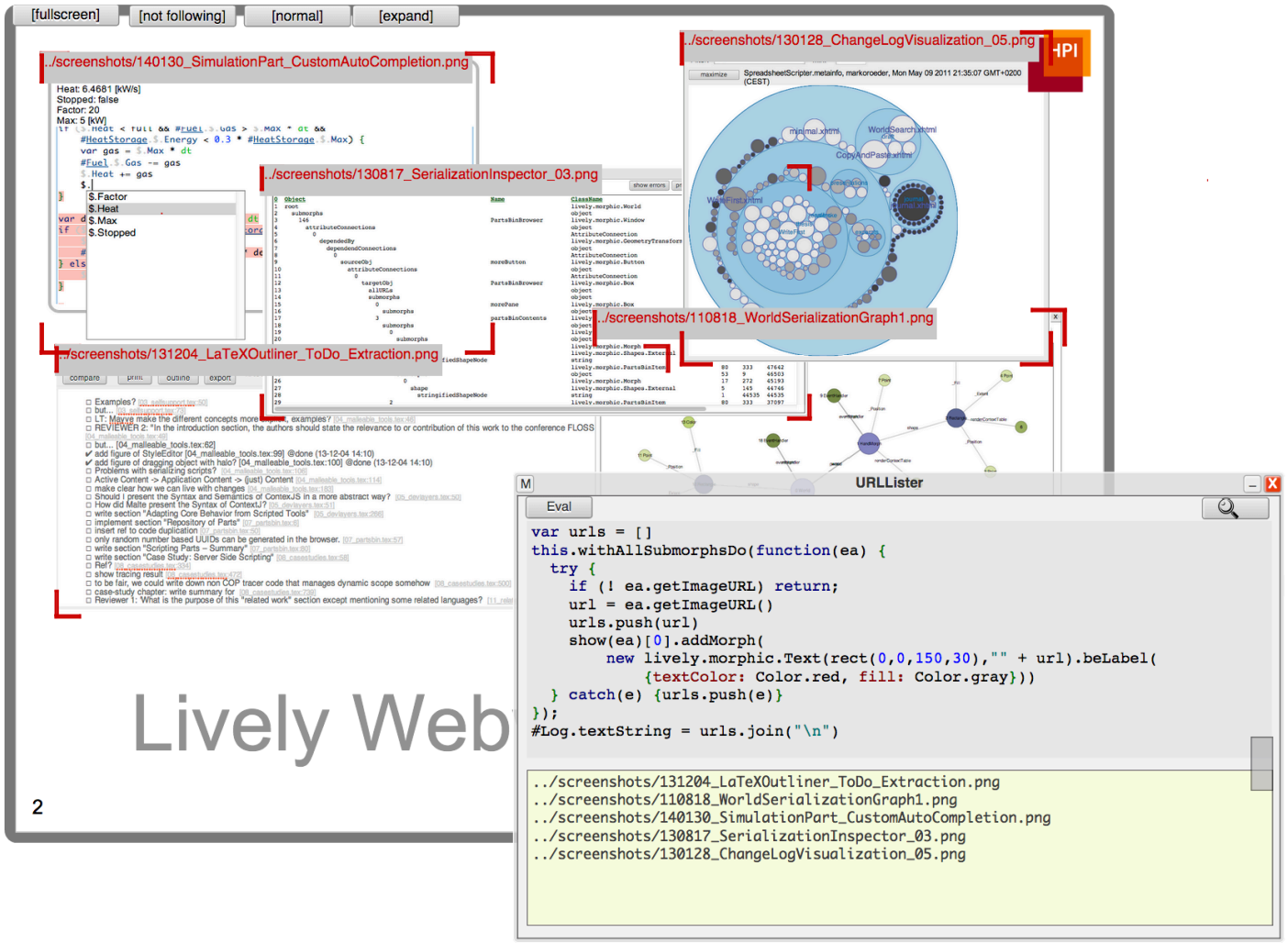


Figure 6.8: The URLLister tool is used on the ProjectMorph. The URLs are displayed directly on the ImagesMorphs.

not be directly customized, we could either implement our own highlighting function or we refine (or fix) the base system behavior.

By looking at its API, we learned that `show` called a `showMorph`, which then called `showThenHide`. The function `showThenHide` accepts a `duration` parameter, which is omitted by `showMorph`. To fix this generally, we could add the parameter to all `show` functions, but we could not oversee what would be the impact of this. So, we decided to context-specifically adapt the `showThenHide` function in the `lively.morphic` object with a `LongerShowDelayLayer`:

```
cop.create("LongerShowDelayLayer").refineObject(lively.morphic, {
    showThenHide: function(r, duration) {
        if (duration === undefined) duration = 10; // new default
        return cop.proceed(r, duration)
    }
})
```

We could either activate it now globally, but this would change other calls to `show`, too. So, we scope the new behavior by overriding the `evalAll` method of our URLListers's workspace, that is triggered by the `Eval` button (5), with an instance-specific script in our `URLLister`:

```
this.addScript(function evalAll() {
    return cop.withLayers([LongerShowDelayLayer], function() {
        return $super()
    })
});
```

This script activates the layer for its dynamic extend and then calls itself `$super`, which will delegate to the default behavior of the class. All calls to `show` from `URLLister` will highlight the corners and display the attached label for 10 seconds instead of 3 seconds. Since the `URLLister` requires the `LongerShowDelayLayer` layer, we add it to the `onload` script of the object, so it is bundled with the `URLLister`.

6.3.3 Tools in Lively Webwerkstatt

Compared to other tools in Lively Webwerkstatt, the `URLLister` is very small, both in terms of time needed for development and object size. But it demonstrates well how we built many tools in Lively Webwerkstatt. Tools

consist of parts that are copied out of the repository or are extracted directly from other tools. The individual parts are then glued together with scripting them. And if necessary, the base system can be adapted with layers. By sharing the result again in a parts bin, others can get inspired, start using the new tool, or adapt it for their own needs.

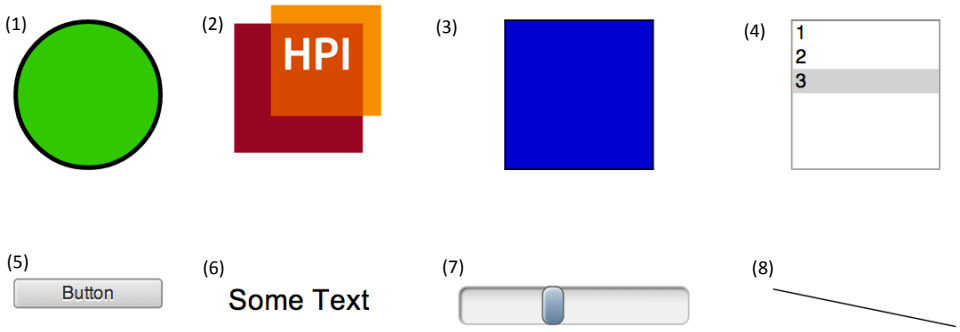
6.4 Overhead of Storing Meta-information

Due to their decentralized nature, objects in Lively Webwerkstatt carry a lot of meta-information. Some meta-information is added to all objects that are or were once published in the `PartsBin` (`partsBinMetaInfo` property). That meta-information can grow in size, because it also stores all change messages typed in by the user when publishing this object. That way an object can be copied from one repository to another and keep some meta-information of its own change history. Other meta-information keeps track of each object's and its subobjects' identity and its history of identity changes when cloning (`id` and `derivationIds` properties). This meta-information increases the size of objects, both in memory and when having to transport them. But since this information is only added on a coarsely granular level to objects actually published in the `PartsBin` it still stays relatively small.

To measure the overhead, we compare the size of objects with and without the meta-information. The size is measured by the length of their serialized string representation. The size is measured once with all meta-information present and once with all information stripped.

For a larger lively part like the `ObjectEditor` tool, the stripping of all `partsBinMetaInfo` and the `derivationIds` saved 38kB from the original 453kB (8.5%). This low percentage can be attributed to the relatively large size of the object. Since the meta-information includes a changelog, the size depends on how often the object is edited and not how large the object is in general.

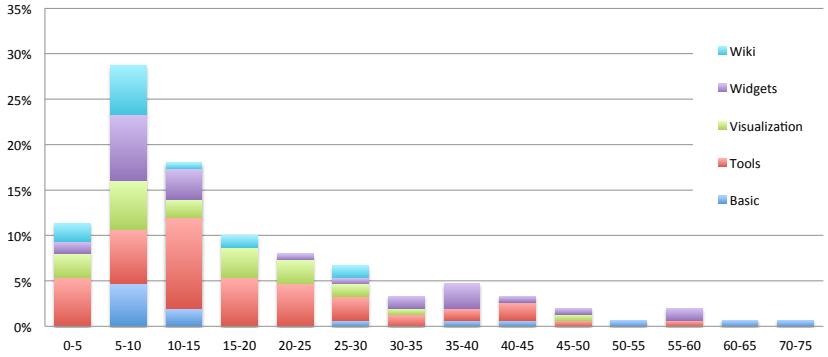
Figure 6.9 shows 8 lively parts from the `Basic` and `Widgets` category in a table. It shows measurements of the serialization size (total and percentage) of the whole object, its parts bin related meta-information, and the addition of derivation ids. It shows that smaller objects have a



name	(1) Ellipse		(2) Image		(3) Rectangle		(4) List	
original	1999	100.00%	1517	100.00%	8240	100.00%	4972	100.00%
partsBinMetaInfo	740	37.02%	423	27.88%	3248	39.42%	706	14.20%
derivationIds	61	3.05%	22	1.45%	373	4.53%	101	2.03%
total	801	40.07%	445	29.33%	3621	43.94%	807	16.23%

name	(5) Button		(6) Text		(7) Slider		(8) Path	
original	5252	100.00%	1851	100.00%	4904	100.00%	2476	100.00%
partsBinMetaInfo	966	18.39%	0	0%	775	15.80%	404	16.32%
derivationIds	114	2.17%	57	3.08%	124	2.53%	22	0.89%
total	1080	20.56%	57	3.08%	899	18.33%	426	17.21%

Figure 6.9: Meta-information memory footprint of basic lively parts. Unlike bigger objects such as the ObjectEditor or other tools, these small objects from the `Basic` category of the `Lively PartsBin` have high ratio between total size and size of the meta-information. (based on data from http://lively-kernel.org/repository/webwerkstatt/PartsBin/Basic/ Rev. 193298, 2013-03-08)



	0-5	5-10	10-15	15-20	20-25	25-30	30-35	35-40	40-45	45-50	50-55	55-60	60-65	70-75	Sum
Basic	0	7	3	0	0	1	0	1	1	0	1	0	1	1	16
Tools	8	9	15	8	7	4	2	2	3	1	0	1	0	0	60
Visualization	4	8	3	5	4	2	1	0	0	1	0	0	0	0	28
Widgets	2	11	5	0	1	1	2	4	1	1	0	2	0	0	30
Wiki	3	8	1	2	0	2	0	0	0	0	0	0	0	0	16
Total	17	43	27	15	12	10	5	7	5	3	1	3	1	1	150

Histogram normalized for each category

	5	10	15	20	25	30	35	40	45	50	55	60	65	70	
Basic	0.0%	43.8%	18.8%	0.0%	0.0%	6.3%	0.0%	6.3%	6.3%	0.0%	6.3%	0.0%	6.3%	6.3%	100%
Tools	13.3%	15.0%	25.0%	13.3%	11.7%	6.7%	3.3%	3.3%	5.0%	1.7%	0.0%	1.7%	0.0%	0.0%	100%
Visualization	14.3%	28.6%	10.7%	17.9%	14.3%	7.1%	3.6%	0.0%	0.0%	3.6%	0.0%	0.0%	0.0%	0.0%	100%
Widgets	6.7%	36.7%	16.7%	0.0%	3.3%	3.3%	6.7%	13.3%	3.3%	3.3%	0.0%	6.7%	0.0%	0.0%	100%
Wiki	18.8%	50.0%	6.3%	12.5%	0.0%	12.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100%
Total	11.3%	28.7%	18.0%	10.0%	8.0%	6.7%	3.3%	4.7%	3.3%	2.0%	0.7%	2.0%	0.7%	0.7%	100%

Figure 6.10: Memory overhead of meta-information stored directly in parts (Webwerkstatt PartsBin Rev. 200311, 2013-10-01))

higher percentage of meta-information data, as shown in an extreme case of the `Rectangle` morph. The `Rectangle` morph has a large percentage of `partsBinMetaInfo` and a small overall serialization size with 40% taken up by meta-information.

This meta-information also accumulates, since it increases every time an object gets cloned or published. It was important for us to know which effect the decentralized storing of such meta-information has on the size of our tools and other user developed parts. Especially since there are some tools that were cloned and published many times.

For that we analyzed several categories of parts in Webwerkstatt's `PartsBin`. We measured the impact the additional meta-information has on the serialization size of each part. As shown in Figure 6.10, in most objects

the percentage the additional meta-information takes up is 5-20%, but there are also some objects where the storing of additional meta-information takes up to 70%.

We learned that the additional storing of meta-information in objects has a high impact on the size of objects: in memory, when sending them over the network, and when storing them on hard-disk. But all these resources where available abundantly in our development scenarios, so we addressed this issue only when needed by providing tools that can strip all kinds of meta-information from objects. Generally, it is not important to keep the complete derivation history and other meta-information stored in the objects at all times. By providing a central information system that can query the history of individual objects, we should be able to keep the overhead of meta-information constant.

Replacing our file based repository of worlds, parts, and modules with a finer granular database that knows about individual objects and their development history is part of our future work.

6.5 Manual Garbage Collection in User Content

When manipulating objects directly, users tend to work only on the things they see. And naturally, there are things they are not aware of: property entries that are not used any more, whole scripts, or references to objects. This garbage will not be removed if the developer does not clean it up. In systems that are bootstrapped regularly, such unused code and content gets automatically removed because the objects are transient and only the abstract code is persisted.

Working with source code is more abstract than working with objects directly. Since developers do not interact with objects, objects can be discarded if they are not valuable user data, to clean up a system. This is, for example, a side effect when serializing the state of applications into relational databases. When editing plain source code, developers have to keep the code clean, because it is their only representation and they have to see it all the time. When working with objects everything is opaque by

default and developers have to actively inspect objects individually or use tools that visualize their state.

This leads to the problem that a new skill is required when working with objects directly: the object space has to be kept clean. This problem is shared by all systems with persisted objects. For example, in the Smalltalk community developers also have to keep their images clean. Since many developers are not used to this, some developers make it a habit to regularly trash their images and bootstrap their projects from fresh images. When there is no source representation available, as in our approach, the objects cannot be trashed automatically but have to be cleaned up by the developers.

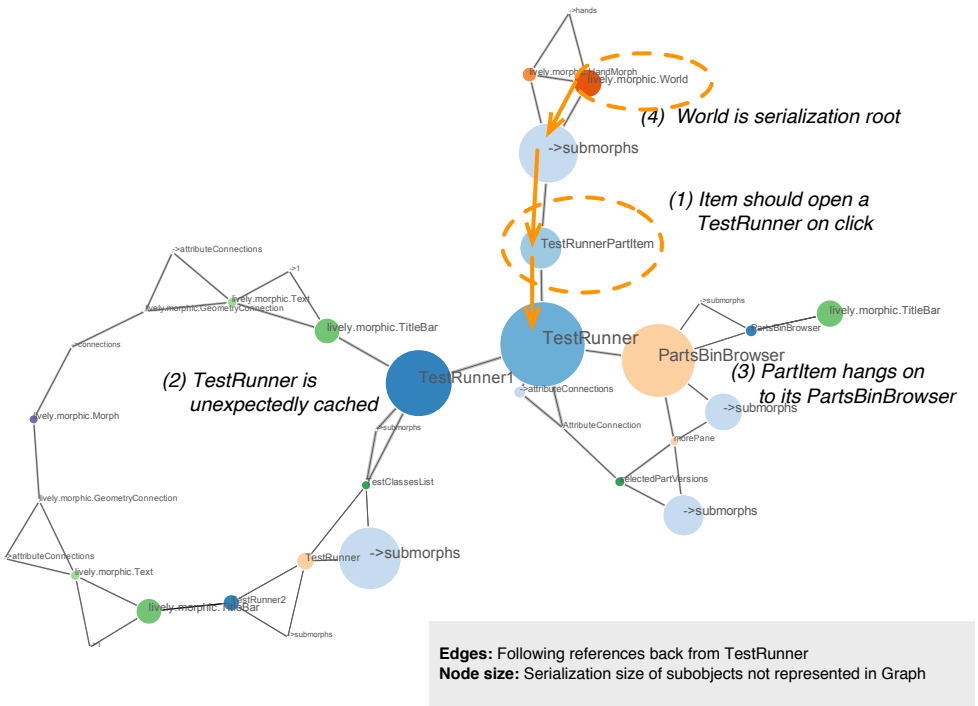


Figure 6.11: Visualization of a world in Lively Webwerkstatt that contains garbage. (created with our `WorldAnalysis` tool)

An example for this is shown in Figure 6.11, which shows a visualization of exploring the references to instances of `TestRunner` and the `PartsBinBrowser` tools, which are not visible in the user interface, but were unintentionally serialized due to other object holding references to them.

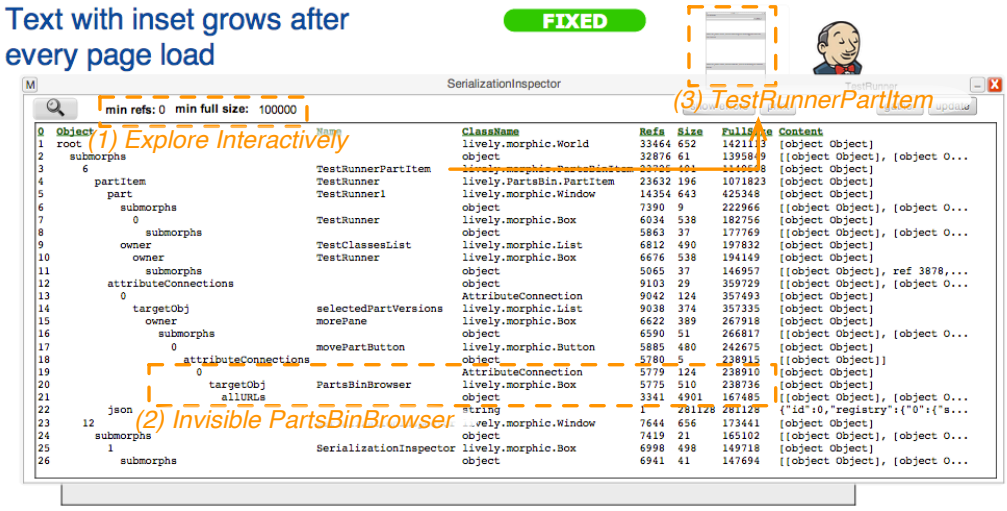


Figure 6.12: Screenshot of SerializationInspector

During our own work in Lively Webwerkstatt we learned to avoid using fixed references as much as possible and replacing them with dynamic name-based lookups. We further implemented tools like a `SerializationInspector` that help us detect such serialization issues. Figure 6.12 shows a `SerializationInspector` was applied (1) to detect the objects (2) that were responsible for the unusual overall size of the world. The issue on this page was that a `PartsBinBrowser` (3) was accidentally serialized together with a `TestRunner` `PartItem`. The users that created the template for this page added such a `PartItem` to the world, to quickly launch a `TestRunner`.

This case exemplifies that like in Smalltalk, having a full-featured object-oriented persistence (as presented in Section 5.2) can be powerful, since it allows serializing not only data but complete instances of running applications. But it also requires additional care to not pollute serialized user content. To address these issue in Webwerkstatt, we a) provide tools

that help keeping the object space clean and b) provide programming techniques that avoid using unnecessary hard references by looking up objects dynamically by names. Having tools to show and remove garbage, which are available in Lively Webwerkstatt, can be helpful in such occasions. But developers are usually not aware of the problem until they realized that somehow, their worlds get larger than they should be. The serialized `TestRunner` and `PartsBinBrowser` where not detected, because they did not harm the user experience and, therefore, nobody looked explicitly for such garbage.

This problem is mainly caused by a design decision: we try to preserve all data and, therefore, like in Smalltalk images, developers have to keep track of the reference they create. An alternative way would be to identify all valuable user data and throw away the rest, but since this may result in accidentally discarding user data we did not pursue that direction.

6.6 Summary

In this chapter, we have presented examples of tool development and adaptation in Lively Webwerkstatt. We have shown how tools can be adapted through direct manipulation and scripting and how layers can be used to context-specifically adapt behavior of the base system at run-time. The `URLLister` example has shown how parts and layers can complement each other. It demonstrates the evolution of a workspace script to a graphical tool. We further have evaluated how the additional meta-information that we have added impacts the serialization size. We have explored a case of how the general serialization kept invisible objects that bloated the world size and how we can detect those with tooling support developed in Lively Webwerkstatt.

Part IV

Related Work and Conclusion

7 Related Work

This chapter relates the results of this thesis to other self-supporting and Wiki-like collaborative programming environments. We discuss then related work that is more implementation-specific, such as repositories of scripted objects, other implementations of context-oriented programming, and techniques for scoping dynamic behavioral adaptations.

7.1 Self-supporting Development Environments

In principle all file-based editors can be part of a self-supporting environment when their source code and the necessary compiler or virtual machine to run the editor is available. In that sense, operating systems like *Unix* [55] are self-supporting. They provide all the tools needed to change themselves at run-time. Not at run-time of the individual application, but at run-time of the operating system. *GNU Linux* [105] goes further than Unix by providing the source code of nearly all applications and tools in the system. Hence, knowledgeable users can adapt all aspects of their system, ranging from shell scripts to the kernel. But programming the Linux kernel from within a running Linux kernel cannot happen at run-time. The system has to boot after every change, resulting in relatively long feedback loops. As stated in Section 2.1 we call systems only self-supporting when they can be changed at run-time from within and without having to restart themselves.

Adapting a Unix-like system on a high level by editing *shell scripts* provides fast feedback loops. Shell scripts by design lack a graphical user interface and can usually be executed rapidly and without user intervention. Since they can be used both as command in the command line interface and for programming they allow users to smoothly transition from using a tool to adapting it or combining it with other tools. This power of adapting

your own tools or creating new ones specifically for the problems at hand are part of our motivation.

A system that takes the run-time adaptability further is *Smalltalk* [35], which is also discussed in Section 2.1. Smalltalk and especially Squeak [49] with its Morphic [75] user interface, provide malleable tools by allowing to change the definition of classes and methods at run-time. However, directly adapting the behavior of individual instances can sometimes be difficult, since methods are only defined in classes. So, experimenting by directly changing important methods can break the system. Since Smalltalk is a single user environment this problems can be mitigated through tool support that helps to recover the system. Typical tools that help Smalltalk developers in recovering after changes that broke the system are the debugger, the emergency evaluator, and the changes file. The debugger is opened automatically on every unhandled error. The emergency evaluator provides a simple REPL and appears if the debugger should fail to open. The changes file can be used to replay changes selectively after the system crashed, so that the change responsible for the crash is not executed again. Because Lively Webwerkstatt is Web-based, using the same techniques as Smalltalk for recovering from failures in run-time development was not possible. The Web-browser's debugger cannot edit methods, so it cannot be used to recover from failures. Using the JavaScript console of the Web-browser feels very similar to the emergency evaluator and can be used in the same way. Since all source code in Lively is under revision control, reverting source code files to a previous version has the same effect as replaying selected changes.

An important aspect of the Smalltalk programming language is its ability to allow programs to reflect upon and modify themselves at run-time. The ability of self-modification in Smalltalk was not added to the language to create programs that can automatically adapt and evolve themselves [92], but it is a prerequisite for building such an interactive development environment in the first place. The reflection mechanisms provided by Smalltalk are directly integrated into its object protocol. More sophisticated object-oriented reflection architectures like *3-KRS* [74], *Mirrors* [14], *Reflectivity* [26], and *Albedo* [88] separate the reflection from the domain computation by allowing only some meta-objects to reflect over normal domain objects. These

approaches deal with meta-circularity problems on a programming language level, but the meta-circular dependencies described in Section 2.1.2 are between tools and the objects those tools work on. From the perspective of the programming language, tools are domain objects, e.g. tools in Lively Webwerkstatt are regular graphical objects the user interacts with, and they are designed that way so that users can adapt them. Similar to Smalltalk, our implementation language JavaScript also provides only "limited, add-hoc reflective facilities" [74]. The implementation of self-supporting development environment despite of these limitations is one of the contributions of Lively Kernel [50] and part of our research.

A general approach to circumvent the problem of reflective dependencies is to step out of the environment. *Virtual Smalltalk Images* [17, 86], for example, allow performing low level changes to core Smalltalk objects that would normally break the system by using a second image to perform those changes. This technique can be useful to deal with corner cases of reflection. Unlike in Smalltalk, the Lively Kernel is loaded from JavaScript source code files, so developers can fall back on text editors, which Lively also provides, in case of an emergency where "Image surgery" would be needed in Smalltalk.

Self [122] is an object-oriented programming environment that allows programming objects directly. Objects in Self contain and access data and behavior in a uniform way. Objects in Self can be transported from one Self world to another [121], but Self does not come with shared repositories of objects. Instead it approaches the problem of collaborative development by using a synchronously shared world. All developers work on one machine in one running Self-image with a 2D world of directly manipulatable graphical objects (Morphs). Collaboration is enabled by using the networking capabilities of *X-Windows* to distribute the user-interface. Since there is only one running system, it can be adapted at run-time without having to replicate it for synchronous collaboration as, for example, the Smalltalk-based virtual environment *Croquet* [98] does. Smaller problems can be directly solved in the world, similar to the single user usage of the system. When bigger problems occur, the normal development world is halted, all collaborators join a debugging world to fix more serious problems from

there [102]. Here, like in our approach, the redundancy and indirection is only introduced when necessary and not active by default.

Modern text editors that follow the Emacs-like self-supporting design to some extent, like the *Sublime Text* editor [97], made the workflow of customizing the system easier. Sublime observes all source code files that define user extensions and automatically reloads them when changed. This allows customizing the system by editing some parts of its source code, and immediately see the change affect all currently active instances of the text-editor. This does not go as far as live programming approaches [124, 78, 37] but customizing the editor becomes very immediate. Emacs and Sublime show that the range of how much a system can be adapted or even evolved may vary. Both systems are built around a statically compiled core, which is then heavily extended using a dynamic programming language. The scope of what can be changed is much smaller in Sublime than in Emacs since many parts of the system are not changeable at all. E.g. the side bar which shows all files in a project can only be customized through declaratively specifying filters, but cannot arbitrary be sorted or extended in other ways. But having such limitations, the environment tends to be, even though it is still possible, much more difficult to break. As most end-user development approaches also show, allowing a system to evolve only in anticipated ways is a practical approach to prevent the users from breaking their tools [81, 87].

Built on top of Smalltalk, *CoExist* [107] addresses the problem of automatic version tracking during run-time development. By allowing multiple versions of the same class to coexist in one environment, different versions of code can be independently tried out and compared side by side in one system. CoExist solves the problem of making small variations to code and being able to quickly jump between them back and forth without having to restart the system. To implement it efficiently in Smalltalk, the virtual machine had to be adapted to make the class lookup late-bound. CoExists can serve as a domain-specific undo-and-redo mechanism for reverting class and method changes. This allows developers to go back to a previously working version of the system. In comparison, developers in Lively Webwerkstatt have to actively employ techniques such as using cloning tools and scoping changes in layers. Such scaffolding can decouple the tools used for development from the tools under development so that adapting

tools at run-time becomes safer or possible at all. CoExist does not support developers with such a scaffolding that prevents breaking the tools in the first place, it provides a safety net, similar to Smalltalk's changelog, that helps recovering the system quickly if tools were broken. Ideally developers would benefit from both approaches. CoExist's versioning of class-objects was experimentally extended to the general versioning of all objects in the system [117, 9]. The approach was implemented in JavaScript and evaluated with Lively Kernel. It showed that having such a general undo-mechanism in a freely programmable object-world can provide a valuable safety net, but due to the bad performance of JavaScript's proxies, it was not fast enough for productive usage yet.

Worlds [126] is a general approach to controlling the scope of side-effects. Implemented for Smalltalk and JavaScript, worlds are a language construct that allow capturing all changes to object state while executing arbitrary code. Since JavaScript does not distinguish between state and behavior, Worlds can also be used to change object behavior in a scoped way. An example use case for this approach is a general undo-and-redo mechanism that can capture all changes to objects and revert them as needed. Worlds can be understood as perspectives through which the same object can look differently depending on the currently active world. Similar to *Us*' layers [101], which are discussed in Section 7.5, worlds can only have one fixed parent. This means unlike COP's layer composition, worlds cannot dynamically combine the effects of several worlds without statically merging them first. Similar to our development layers in ContextJS, worlds can serve as scaffolding that has to be applied before actually performing a potentially tool breaking adaptation. But different to development layers, captured changes in worlds cannot be treated as features and be activated on demand. Our development layers serve both purposes, first they can be used as scaffolding that provides more safety during run-time adaptation in a self-supporting development environment, and second development layers can be used to share adaptations with other users who can try out new features without having to statically merge them into their system.

Vivide [112] is a scriptable data-flow development environment in Smalltalk. Development tools themselves are implemented as scripts that select, transform, and present meta-objects such as classes and methods

to the user. Vivide supports to adapt those scripts at run-time, making the tools evolvable while they are used. Since Vivide is implemented in Smalltalk and can edit Smalltalk code, it can be used in a self-supporting way. But unlike Webwerkstatt it is not collaborative and does not need to provide additional means for preventing queries to break Smalltalk.

Edit-and-Continue [28], as implemented for Microsoft's .Net framework, observes changes to source code files, automatically compiles them, computes the difference to the currently running system and replaces existing methods at run-time. This allows, for example, experimenting with parameters or interactively developing a new algorithm in a game. Since the program is updated automatically, the system produces a short feedback loop. But the scope of possible changes is narrow, e.g. it does not allow creating completely new behavior or objects, it is not suited to develop the whole system continuously that way. Such changes force developers to restart their application.

Brackets [1] and *Atom* [34] are examples of new development environments that allow for run-time development of Web-applications by controlling not only the server-side, but by also running the client side in custom Web-browsers. Because the environment is aware of the development context, the running application can automatically be updated while the source is edited. The standard example here is changing the color in a *Cascading Style Sheets* (CSS) file which is immediately shown in the Web-application. Even though both tools are written with JavaScript, HTML, and CSS, they are not self-supporting, because they cannot be used to adapt themselves at run-time.

7.2 Collaborative Web-based Development

Version control systems, such as *Subversion* (SVN) [85] and *Git* [120], can be used for collaboration and as fallback to revert source code if needed, two features which are very useful in self-supporting systems. For example, Lively Webwerkstatt uses SVN as a back-end to store all source code and user content and, therefore, can provide a full change history of the system.

GitHub [23] is a social code sharing environment. It combines source code repositories with issue management. They provide a better user interface and automation around the distributed version control system Git, which was built for the management of Linux source code [120]. Unlike SVN and similar version control tools, Git does not require a central repository. Developers can commit fine-grained changes, which can later be merged with other developers or a central repository like GitHub. The decentralized approach makes forking and merging whole projects easier, since the complete development history can be preserved. Our approach of keeping the cloning history per object is more fine grained than the forking history of projects in GitHub. In Webwerkstatt and its parts bin we can trace the history of subobjects after they have become part of another object. Copying files with their history from one Git repository into another is not directly possible, but requires merging and filtering of the entire projects history. [18]

Similar to Lively Webwerkstatt, *Web-based development environments* like *Cloud9* [24] allow developers to start working without installing tools or setting up a project first. Hence, initial overhead of contributing to a software project can be reduced to visiting a Web-page. But different from Lively, general-purpose Web-based developments separate the development from the run-time environment as discussed in Section 2.1.3. The programming workflow in such environments typically involves an edit-run-test loop. Depending on the starting time of the Web-application under development and the availability of an automated test suite, the iteration time can be relatively fast. Similar to all file-based development environments, systems like Cloud9 can also be used to change their own source code. This does not allow evolving the system while it is being used, since for adapting the environment and its tools, a second instance of environment has to be opened. There the source code can be edited and the changes can be tested in an edit-run-test loop with a third instance. When the developer is satisfied, the first environment has then to be restarted to get the new adaptation. In Lively Webwerkstatt the tools can be adapted and evolved while they are used.

WeScheme [128] is a Web-based development environment for the functional programming languages Scheme and Racket. WeScheme uses a

client-side byte-code interpreter on top of JavaScript to support event-driven functional programming. The byte-code is created by a server-side Racket compiler. Similar to Lively Webwerkstatt WeScheme provides an interactive development environment, that allows users to start programming without having first to install software. Since WeScheme does not use JavaScript directly, as Lively does, it has more control over its execution, allowing to interrupt and inspect the running coroutines as needed. Getting this kind of control over the execution without using a full JavaScript interpreter inside of JavaScript is part of our ongoing research [95]. Unlike in Webwerkstatt, users in WeScheme cannot adapt and evolve their tools from within the environment.

MikiWiki [129] is a Web-based wiki-like development environment that allows its users to create, share, and clone little JavaScript widgets and to use them in their wiki pages. The environment runs mainly on the client side to allow its users to collaboratively design and evolve both wiki text and JavaScript without running user code on the server. MikiWiki is used as an example for a meta-design process that involves the collaboration and communication of developers with their users through the development environment. Even though the development of the JavaScript widgets happens at the use time of the Wiki, the development of the widgets happens in a typical edit-run-test cycle. Even though its users can to some degree customize their wiki pages via user-created widgets, the environment itself cannot be changed.

7.3 Repositories of Objects with Instance-specific Behavior

Fabrik [51] is a visual programming environment that allows visual programming by connecting components dragged out of a parts bin. A composition could then be put back into the parts bin for use elsewhere. The parts bin is not shared and the environment not collaborative. We explored in previous work how we can transfer concepts of Fabrik to the Lively Kernel and implemented with Lively Fabrik [71] a Web-based version as an application on top of it. Lively Fabrik is an end-user programming environment allowing

to create user interface and functional components and connect them using wires to program in a data-flow style. The environment can be used to interactively create mashups [113] and Web-widgets, but different to the scripting facilities in Lively Webwerkstatt, it cannot evolve or extend the underlying Lively Kernel.

Second Life [91] is a virtual 3D world where most content is generated by users. Users can create and modify graphical objects, which are made of graphical primitives and scripts. These objects can then be shared with other users by selling them or giving them away for free. Granting editing rights allows users to work together on the same objects. Since the creation of active content is domain-specific, the Second Life user interface and tools cannot be extended from within the system as it is possible with the Lively approach.

Besides being a self-supporting system as discussed in Section 7.1, *Squeak//Smalltalk* [49] can be used as a personal multimedia authoring environment [39]. It has also a parts bin, where objects can be dragged out. Squeak is a class-based system and programming objects directly is not the intended way to extend the system, except in the case of the Etoys framework.

The *SuperSwiki* [90] allows users to share Squeak Etoys [54, 82] projects over the Web. Projects are containers, similar to Lively worlds, for user-created content built from objects and scripts. Unlike Lively Werkwerkstatt it is not Web-based and self-supporting, since the environment is implemented with Smalltalk and cannot be evolved with pure Etoys.

Many end-user programming environments have built-in repositories to allow their users to share content. A representative example is *Scratch* [87], a Squeak-based multimedia tile scripting programming environment for children that evolved from the Squeak Etoys work. In Scratch, sharing and remixing of projects plays an important part for being a "more social" programming environment. Children are encouraged to publish their projects from Scratch directly to the Scratch community website, where children can directly play with the projects (via a Java plug-in) and download them. As in Etoys, the unit of sharing is the project. Individual scripts and sprites can be copied from one project into another, but Scratch has no central repository of reusable parts.

Yahoo! Pipes[1] is a Web-based end-user programming environment that allows users to graphically wire together components to mashup Web-resources and produce RSS feeds. Those pipes can be shared and copied, modified, and reused by other users. Before a pipe can be modified, the user has to copy the pipe to its own set of pipes. Yahoo Pipes is domain-specific and, therefore, does not allow the creation of general active Web content.

The main difference between these restricted end-user development environments [63] and Lively Webwerkstatt is that they do not allow its users to create objects and tools that evolve the system itself. All of these environments distinguish between tool and material level [89, 47]: they put the editor as a tool on a different level than the script that is edited. This separation prevents tools from breaking tools, but also makes the environments inherently not self-supporting. It is clear that not all users want to have such a level of freedom and power, but building such freedom into a system allows its users to adapt it in unanticipated and hopefully interesting and useful ways.

7.4 Context-oriented Programming

As discussed in Section 3.3, dynamically scoped behavioral variations can be used as scaffolding means for development in self-supporting environments. ContextJS as introduced in chapter 4 is a Context-oriented programming implementation for JavaScript. Context-oriented programming as an approach for dynamically scoping behavioral adaptations at runtime was initially developed for Lisp [21] and Smalltalk [42], but later also implemented in various languages [4].

Original Scoping Mechanism. Most COP language implementations provide control flow-specific scoping as introduced in Section 4.1.1. *ContextL* [21, 22], based on Lisp and the Common Lisp Object System (CLOS), was one of the first COP extensions to a programming language. Layers can be defined for classes, functions, and methods.

Following ContextL and *ContextS* for Smalltalk [41], several meta-level libraries for dynamic programming languages were developed, namely

[1] `http://pipes.yahoo.com/` (visited 2014-09-13)

ContextR [94] for Ruby, *ContextPy* [43] for Python, and *ContextG* for Groovy. For the statically typed language Java, some COP prototypes [42, 8] and the compiler-based extension *ContextJ* [5] have been developed. A minimal subset of ContextJ, *cj*, was implemented for the *delMDSOC* kernel [93].

Implicit Layer Activation. The Python language extension *PyContext* [125], supports a variant of *implicit layer activation* where layers can determine if they are active. Layers can provide a method evaluating an activation condition before layered method invocations. This approach allows implementing activation mechanisms for specific layers, but it cannot change the entire layer composition. With our approach such implicit layer activations could be achieved by implementing an object-specific layer composition that delegates the layer activation to registered layers.

Event-specific Scoping. In some application domains, such as adaptive user interfaces, behavioral variations should be active depending on events rather than to control flows. Events can occur at various points in an execution that may not be obvious in the source code. The Java-based languages *JCop* [7] and *EventCJ* [52] address this problem by providing declarative composition statements adopted from aspect-oriented programming [57]. These declarations describe join points at which certain layers should be composed. Using AOP for implementing domain-specific scoping of layer activations is an alternative to using our ContextJS's open implementation. Both use some form of additional computation to let developers express scope in their domains.

7.5 Dynamically Scoped Behavioral Adaptation

With *scoping strategies* approach [114], variables are not either statically or dynamically scoped; instead, developers can parameterize variable bindings. A scoping strategy is specified with functions implementing the propagation and activation of a binding. While that approach opens the scoping implementation of variable bindings, ContextJS opens the scoping implementation of layer activation.

The *Ambient Object System* [36] (AmOS) supports context-orientation. AmOS is a prototype-based object system built on top of Common Lisp that

supports behavioral adaptations with partial method definitions and context objects, which correspond to COP layers. At any method call in AmOS, receiver methods are first looked up in the current activation and then in enclosing lexical scopes. If no appropriate method is found in the lexical scope, the lookup continues in a graph of context objects delegating to each other. The delegation chain between these context objects can be modified dynamically, achieving context-specific behavior. Unlike ContextJS, AmOS does not support the (open) implementation of domain-specific scoping strategies like structural-specific scoping of behavioral adaptation in a hierarchy of graphical objects.

Aspect-oriented programming (AOP) [57] aims to tame crosscutting concerns by introducing pointcut-based quantification. The main distinction between AOP and COP is that the former allows for a joint specification of *when* in the execution flow *what* kind of functionality should be used, while COP separates *when* (using `with` statements) from *what* (using layers and partial methods). For a comparison of AOP and COP as appropriate representation of behavioral variations, we distinguish between *homogeneous* and *heterogeneous* crosscutting concerns [3]. Homogeneous crosscuts execute the same functionality at multiple locations in a control flow, for which AOP provides well suited abstractions. AOP implementations of heterogeneous crosscuts tend to be less understandable to developers than layer-based implementations [3], since they have to mimic COP behavior using pointcuts with dynamic conditions and advice that are complex and fragile for changes.

There are various approaches to deal with AOP advice code triggering its own pointcuts directly or indirectly. In AspectJ the cflow pointcut designator allows (de-)activating an aspect in a specific control flow. Another approach to avoid the endless regression caused by aspects directly or indirectly calling themselves, are execution levels for AOP [115]. In this approach, the problem of conflation [20] of base and advice code is solved by executing them in different levels. By default aspects only observe the execution of specific levels, thus avoiding endless regressions.

These execution levels are also implemented in *AspectScript* [119], an AOP implementation for JavaScript. AspectScript aims for expressibility and, therefore, has to rewrite all JavaScript code. This makes the code several

times slower, even if no aspects are active. ContextJS does only instrument methods that are refined by layers, which lets JavaScript code that is not refined by any layer run at full speed.

Perspectives in the *subjective programming* language *Us* [101] are a way to describe the scope of behavioral variations. Us changes message passing of *Self* [122] to incorporate perspectives on layers. Perspectives define a fixed layer composition by statically connecting a layer with its parent layer. This implies that layers in subjective programming—unlike COP—cannot be part of different compositions at the same time. The problem of breaking tools during the development of interface code was explicitly mentioned as an application of perspectives. One of the examples shows how perspectives are used to try out and combine different changes. The authors describe a fallback approach for a debugger to a safer perspective. Us was never implemented [123].

Classboxes [12, 11] support static scoping of behavioral adaptations. A classbox is an explicitly named scope in which classes and their members can be defined. Besides common subclassing, Classboxes support local refinement of imported classes by adding or modifying their features without affecting the originating classbox, much like layers and partial methods. Since scoping is only controlled by the import and use of objects in a module, structural and instance-specific scopes cannot be expressed by Classboxes.

ChangeBoxes [25] are a mechanism that allows managing multiple development branches of an application within a single, running environment. It allows, for example, modifying a deployed and running application by doing the development in a change box. Inside that change box the new behavior can be safely developed and tested before it is merged into the running application. Since change boxes work on a Smalltalk project level individual tools cannot be separated so they can be safely adapted. But similar to CoExist, ChangeBoxes could in principle be employed as an additional safety net for developing in Smalltalk, providing a fallback in case the tools were broken.

Alternative approaches to inheritance, such as *traits* [100, 27] and *mixins* [13], allow for an additional inheritance relationship orthogonal to the class hierarchy, but do not offer dynamic adaptation like layers.

Refinements as introduced with Ruby 2.0 are an example of restricting the scope of library adaptation, known as *monkey patches* to the Ruby community. By default such monkey patches are globally scoped and replace the original behavior. This leads to problems when two libraries patch the same behavior in the base library in conflicting ways. As a solution Ruby 2.0 introduced refinements as an experimental feature to localize monkey patches. Refinements allow replacing methods of a class in the lexical context of ruby files. This has the effect that they are only partially useful, since they can only adapt the direct usage of a method in a library, but not indirect ones. Unlike behavioral adaptation through COP, refinements are only available in the lexical scope, e.g. when directly called from a file that uses that refinement. If the method would be indirectly called by a method defined in the base system or other libraries, the refinement would not be active. This is not very powerful, since the method invocations are already in control of the adaptation writers, they could have called helper objects of their own. The method invocations that programmers do not have direct control over are the interesting ones and the hardest to adapt.

8 Summary

This thesis has discussed self-supporting development environments and how they allow adapting applications and the system with its programming tools at run-time. Lively Kernel combines this approach with collaborative Web-based development. The resulting wiki-like environment supports both user-level collaboration and shorter feedback-loops during development at run-time.

Because of reflection and meta-circularity, developing in a self-supporting environment inherently has the danger of breaking your own tools. In a shared environment like a Lively Wiki the impact can be more severe since dangers can and often do affect more than one user. In this thesis we developed approaches that mitigate this problem by allowing for a controlled but still immediate adaptation of tools at run-time and in the collaboratively shared environment.

8.1 Contributions

The key contributions of our approach to self-supporting evolution of a collaborative Web-based development environment are:

Lively Parts By building tools as user-modifiable graphical objects, they can be adapted directly while being used. The deep cloning of an entire object composition as a *lively part* serves two purposes: first, the development at run-time is made more robust, because modifying a cloned tool does not interfere with other similar tools of the same kind including the one used to perform these modifications. Second, publishing parts in a shared *parts bin* allows for wiki-like collaboration that involves both programming and

direct manipulation of objects since not all shareable modifications require editing of source code.

Development Layers The base system provides behavior that affects all objects in the environment. To adapt the base system at run-time, *context-dependent layers* (COP) allow controlling the scope of changes during development as needed. Further, separated changes allow sharing experimental features that can be safely explored by other users. In our approach, layers serve as scaffolding during development. The scaffolding can later be removed by merging the changes into the base system.

Open Implementation of Layer Composition COP allows adapting system behavior by composing layers at run-time. Our work with Lively Kernel helped identify the need for more flexible domain-specific scoping strategies. In *ContextJS*, our COP language extension for JavaScript, we provide an *open implementation* of layer composition that allows for new scoping strategies such as *structural-scope* that takes the composition hierarchy of graphical objects into account.

Object Derivation History When different clones of objects (*parts*) have to be merged back together, because they where independently developed, a merging algorithm cannot rely on object identifiers any more. The object identifiers have changed, because a clone has to coexist with its original object in the same environment, so the clone and all its cloned subobjects, received new identities. Maintaining an object-specific history of former identities (*derivation history*) as meta-information helps identifying moved subobjects and finding a common ancestor for automatic merging.

8.2 Future Work

The self-supporting development of Lively Webwerkstatt spawned interesting questions in broad a field.

Reification of JavaScript Execution When building a development environment, we need full control over the code we are running at development time, for example, to slow down execution or to visualize its execution state like in *live programming* [124]. But due to security limitations, the Web-browser limits the reflectional capabilities of JavaScript. For example, it is currently not possible to interrupt out-of-control programs [127]. This limitation of the JavaScript virtual machine can be circumvented by executing all user code in an interpreter or rewriting the code appropriately. We currently experiment with approaches for implementing a debugger for JavaScript [95] that can debug code inside a browser without support of the surrounding JavaScript virtual machine.

Script Derivation History Copying source code is considered a bad programming style. It increases redundancy that makes it harder for developers to deal with the amount of code. For example, if a bug is fixed or a feature is added in the source, the bug remains in the copy (or vice versa), if they are not changed simultaneously. If the copied code should be changed, developers have to locate every copy and apply the change manually – to the extent that they still have control over them. Unlike the cloning of objects, when copying text, this derivation information is lost, since copying can only be traced by matching the text. In Lively Webwerkstatt, we only trace the cloning of objects, but scripting objects also involves the copy and paste of individual functions or other source code snippets. Preserving and using this information should provide a derivation history on the level of functions.

Coping with Redundancy The redundancy introduced by cloning objects can become a problem: when a new feature is prototyped in an object and this object is duplicated and used in multiple locations, changing that behavior in all copies may become difficult. As long as the new objects are still only in one world, we can migrate new features or bug fixes manually or by using scripts. We experimented with a special version of the object editor that allowed to edit multiple instances at once [32], which mitigated this problem when the objects with the cloned behavior are in one world. But what if the object was already copied to different worlds? Developers have

to search through all worlds and migrate all copies of the object individually. Some objects may even not reachable because they are in private space or in different repository. Therefore, automating this process and replacing it with smart object migration is part of our future work.

Publications

Journals

- Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. Declarative Layer Composition with the JCop Programming Language. *JOT: Journal of Object Technology*, 2013
- Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Elsevier Journal on Science of Computer Programming, Special Issue on Software Evolution*, 2011

Conferences

- Tim Felgentreff, Alan Borning, Jens Lincke, Robert Hirschfeld, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/js - a browser-based implementation of an object constraint language. In *ECOOP 2014: European Conference on Object-Oriented Programming*. ACM, 2014
- Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. Interleaving of Modification and Use in Data-driven Tool Development. In *Onward! '14: Symposium for New Ideas, New Paradigms, and Reflections on Everything to do with Programming and Software*. ACM, 2014
- Jens Lincke and Robert Hirschfeld. User-evolvable Tools in the Web. In *OpenSym 2013: International Symposium on Open Collaboration*. ACM, 2013

- Robert Krahn, Jens Lincke, and Robert Hirschfeld. Efficient Layer Activation in ContextJS. In *C5 '12: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2012
- 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 *HICSS 2012: Hawaii International Conference on System Sciences*. IEEE, 2012
- Bastian Steinert, Marcel Taeumel, Jens Lincke, Tobias Pape, and Robert Hirschfeld. CodeTalk Conversations about Code. In *C5 '10: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2010
- Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09: International Symposium on Wikis and Open Collaboration*. ACM, 2009
- Norman Holz, Robert Hirschfeld, Jens Lincke, Michael Rüger, and Michael Haupt. Sophie - Tools and Materials in Multimedia Book Creation. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009
- Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. Lively Fabrik - A Web-based End-user Programming Environment. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009
- Philipp Engelhard, Robert Hirschfeld, and Jens Lincke. Pitsupai - Collaborative Scripting in a Distributed, Persistent 3D World. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009
- Bastian Steinert, Michael Grünewald, Stefan Richter, Jens Lincke, and Robert Hirschfeld. Multi-user Multi-account Interaction in Groupware Supporting Single-display Collaboration. In *CollaborateCom '09: International Conference on Collaborative Computing, Networking, Applications and Worksharing*. IEEE, 2009

- Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into Examples - Leveraging Tests for Program Comprehension. In *TESTCOM 2013: Conference on Testing of Communicating Systems*. Springer, 2009

- Jens Lincke, Robert Hirschfeld, Michael Rüger, and Maic Masuch. SophieScript - Active Content in Multimedia Documents. In *C5 '08: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2008

Workshops

- Jens Lincke and Robert Hirschfeld. Programming in the Cloud: Context-oriented Programming for Self-supporting Development Environments. In *C-HPC '12: Sino-German Workshop on Cloud-based High Performance Computing*. Hasso Plattner Institute, 2012

- Jens Lincke and Robert Hirschfeld. Scoping Changes in Self-supporting Development Environments using Context-oriented Programming. In *COP '12: Workshop on Context-Oriented Programming, co-located with ECOOP*. ACM, 2012

- Jens Lincke, Robert Krahn, and Robert Hirschfeld. Implementing Scoped Method Tracing with ContextJS. In *COP '11: Workshop on Context-oriented Programming, co-located with ECOOP*. ACM, 2011

- Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-oriented Programming Languages. In *COP '09: Workshop on Context-oriented Programming, co-located with ECOOP*. ACM, 2009

Book Chapters

- Robert Hirschfeld, Bastian Steinert, and Jens Lincke. Agile Software Development in Virtual Collaboration Environments. In Christoph Meinel, Larry Leifer, and Hasso Plattner, editors, *Design Thinking: Understand-Improve-Apply*. Springer, 2011

Technical Reports

- Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, and Jens Lincke. Explorative Authoring of Active Web Content in a Mobile Environment. Technical report, Hasso Plattner Institute, 2013
- Jens Lincke and Robert Hirschfeld. Web-based Development in the Lively Kernel. Technical report, Hasso Plattner Institute, 2012

Bibliography

[1] Adobe. Brackets. Software, `http://brackets.io/`, 2014.

[2] BJ Allen-Conn and Kim Rose. *Powerful Ideas in the Classroom Using Squeak to Enhance Math and Science Learning*. Viewpoints Research, 2003.

[3] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *Transactions on Software Engineering*, 2008.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-oriented Programming Languages. In *COP '09: Workshop on Context-oriented Programming, co-located with ECOOP*. ACM, 2009.

[5] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ - Context-oriented Programming for Java. *JSSST: Computer Software of The Japan Society for Software Science and Technology*, 2010.

[6] Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. Declarative Layer Composition with the JCop Programming Language. *JOT: Journal of Object Technology*, 2013.

[7] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-Specific Software Composition in Context-Oriented Programming. In *SC 2010: International Conference on Software Composition*. Springer, 2010.

[8] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. Dedicated Programming Support for Context-aware Ubiquitous Applications.

In *UBICOMM 2008: 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE, 2008.

[9] Robert Hirschfeld Bastian Steinert, Lauritz Thamsen. Object Versioning to Support Recovery Needs: Using Proxies to Preserve Previous Development States in Lively. In *DLS '05: Dynamic Languages Symposium*. ACM, 2014.

[10] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[11] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Computer Languages, Systems & Structures*, 2005.

[12] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In *JMLC 2003: Joint Modular Languages Conference*. Springer, 2003.

[13] Gilad Bracha and William Cook. Mixin-based Inheritance. In *OOPSLA/ECOOP '90: European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*. ACM, 1990.

[14] Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages. In *OOPSLA '04: Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2004.

[15] John Brant, Brian Foote, Ralph Johnson, and Donald Roberts. Wrappers to the Rescue. In Eric Jul, editor, *ECOOP'98: European Conference on Object-Oriented Programming*. Springer, 1998.

[16] Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, and Jens Lincke. Explorative Authoring of Active Web Content in a Mobile Environment. Technical report, Hasso Plattner Institute, 2013.

[17] Gwenaël Casaccio, Damien Pollet, Marcus Denker, and Stéphane Ducasse. Object Spaces for Safe Image Surgery. In *IWST 2009: International Workshop on Smalltalk Technology, co-located with ESUG 2013*, New York, NY, USA, 2009. ACM.

[18] Scott Chacon. *Pro Git*, volume 288. Springer, 2009.

[19] Ruth Chang. How to Make Hard Choices. TED Talk, `http://www.ted.com`, 2014.

[20] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding Confusion in Metacircularity: the Meta-Helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *JSST 1996: Object Technologies for Advanced Software*. Springer, 1996.

[21] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Dynamic Languages Symposium*. ACM, 2005.

[22] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In D. Lightfoot and C. Szyperski, editors, *JMLC 2006: Joint Modular Languages Conference 2006*. Springer, 2006.

[23] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *CSCW 2012: Conference on Computer Supported Cooperative Work*. ACM, 2012.

[24] Ruben Daniels. Cloud9 – Your Development Environment, in the Cloud. Software, `https://c9.io/`, 2011.

[25] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and Exploiting Change with Changeboxes. In *ICDL '07: International Conference on Dynamic Languages*. ACM, 2007.

[26] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The Meta in Meta-Object Architectures. In *TOOLS 2008: Objects, Components, Models and Patterns*. Springer, 2008.

[27] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A Mechanism for Fine-Grained Reuse. *TOPLAS: Transactions on Programming Languages and Systems*, March 2006.

[28] Marc Eaddy and Steven K Feiner. Multi-Language Edit-And-Continue for the Masses. Technical report, Department of Computer Science, Columbia University, 2005.

[29] ECMA. Standard ECMA-262 ECMAScript Language Specification, 2009. 5th Edition (December 2009).

[30] Philipp Engelhard, Robert Hirschfeld, and Jens Lincke. Pitsupai - Collaborative Scripting in a Distributed, Persistent 3D World. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009.

[31] Tim Felgentreff, Alan Borning, Jens Lincke, Robert Hirschfeld, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/js - a browser-based implementation of an object constraint language. In *ECOOP 2014: European Conference on Object-Oriented Programming*. ACM, 2014.

[32] Tim Felgentreff, Philipp Tessenow, and Lauritz Thamsen. Lively Groups — Shared Behavior in a World of Objects. Seminar Report, 2012.

[33] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003.

[34] Github. Atom. Software, `https://github.com/atom`, 2014.

[35] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[36] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 2008.

[37] Chris Granger. Light Table. Sofware, `http://www.lighttable.com/`, 2012.

[38] Chris Granger. The IDE as a value. Blog entry and video, May 2013.

[39] Mark Guzdial and Kim Rose. *Squeak, Open Personal Computing and Multimedia*. Prentice Hall, 2001.

[40] Michael A. Hiltzik and Rutkoff. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. HarperCollins, 1999.

[41] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-oriented Programming with ContextS. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE II: Generative and Transformational Techniques in Software Engineering*. Springer, 2008.

[42] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *JOT: Journal of Object Technology*, March - April 2008.

[43] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *SAC 2010: Symposium on Applied Computing*. ACM, 2010.

[44] Robert Hirschfeld and Kim Rose, editors. *Self-Sustaining Systems*. Springer, 2008.

[45] Robert Hirschfeld, Bastian Steinert, and Jens Lincke. Agile Software Development in Virtual Collaboration Environments. In Christoph Meinel, Larry Leifer, and Hasso Plattner, editors, *Design Thinking: Understand-Improve-Apply*. Springer, 2011.

[46] Douglas R Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

[47] Norman Holz, Robert Hirschfeld, Jens Lincke, Michael Rüger, and Michael Haupt. Sophie - Tools and Materials in Multimedia Book Creation. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009.

[48] Dan Ingalls. The Sun Labs Lively Kernel Technical Overview. Technical report, Sun Microsystems, Inc., 2008.

[49] 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 *OOPSLA '97: Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 1997.

[50] Dan Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The Lively Kernel A Self-Supporting System on a Web Page. In *S3 2008: Self-Sustaining Systems*. Springer, 2008.

[51] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: a Visual Programming Environment. In *OOPSLA '88: Conference on Object-oriented Programming, Systems, Languages and Applications*. ACM, 1988.

[52] Tetuso Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing Event-based Context Transition in Context-oriented Programming. In *COP '10: Workshop on Context-Oriented Programming, co-located with ECOOP*. ACM, 2010.

[53] Alan Kay. The Early History of Smalltalk. In *HOPL II: History of Programming Languages*. ACM, 1996.

[54] Alan Kay. Squeak Etoys Authoring and Media, 2005. as of Aug 01, 2005, `http://www.squeakland.org/pdf/etoys_n_authoring.pdf`.

[55] Brian W Kernighan and John R Mashey. Unix Programming Environment. *Computer*, 1984.

[56] Gregor Kiczales. Beyond the Black Box: Open Implementation. *Software*, 1996.

[57] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented Programming. In *ECOOP 1997: European Conference on Object-Oriented Programming*. Springer, 1997.

[58] Gregor Kiczales and Andreas Paepcke. Open Implementations and Metaobject Protocols. Technical report, Xerox PARC, 1995.

[59] Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09: International Symposium on Wikis and Open Collaboration*. ACM, 2009.

[60] Robert Krahn, Jens Lincke, and Robert Hirschfeld. Efficient Layer Activation in ContextJS. In *C5 '12: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2012.

[61] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Anchor Press, 1984.

[62] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86: Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 1986.

[63] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. End-User Development: An Emerging Paradigm. In *End User Development*, pages 1–8. Springer, 2006.

[64] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Elsevier Journal on Science of Computer Programming, Special Issue on Software Evolution*, 2011.

[65] Jens Lincke and Robert Hirschfeld. Programming in the Cloud: Context-oriented Programming for Self-supporting Development Environments. In *C-HPC '12: Sino-German Workshop on Cloud-based High Performance Computing*. Hasso Plattner Institute, 2012.

[66] Jens Lincke and Robert Hirschfeld. Scoping Changes in Self-supporting Development Environments using Context-oriented Programming. In *COP '12: Workshop on Context-Oriented Programming, co-located with ECOOP*. ACM, 2012.

[67] Jens Lincke and Robert Hirschfeld. Web-based Development in the Lively Kernel. Technical report, Hasso Plattner Institute, 2012.

[68] Jens Lincke and Robert Hirschfeld. User-evolvable Tools in the Web. In *OpenSym 2013: International Symposium on Open Collaboration*. ACM, 2013.

[69] Jens Lincke, Robert Hirschfeld, Michael Rüger, and Maic Masuch. SophieScript - Active Content in Multimedia Documents. In *C5 '08: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2008.

[70] Jens Lincke, Robert Krahn, and Robert Hirschfeld. Implementing Scoped Method Tracing with ContextJS. In *COP '11: Workshop on Context-oriented Programming, co-located with ECOOP*. ACM, 2011.

[71] Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. Lively Fabrik - A Web-based End-user Programming Environment. In *C5 '09: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2009.

[72] 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 *HICSS 2012: Hawaii International Conference on System Sciences*. IEEE, 2012.

[73] Pattie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA '87: Conference on Object-oriented Programming, Systems, Languages and Applications*. ACM, December 1987.

[74] Pattie Maes. Computational Reflection. *The Knowledge Engineering Review*, 3 1988.

[75] John Maloney. An Introduction to Morphic: The Squeak User Interface Framework. *Squeak: OpenPersonal Computing and Multimedia*, 2001.

[76] John Maloney and Randall B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: 8th annual ACM Symposium on User Interface and Software Technology*. ACM, 1995.

[77] John McCarthy. History of LISP. In *HOPL I: History of Programming Languages*. ACM, August 1978.

[78] Sean McDirmid. Usable Live Programming. In *Onward! '13: Symposium on New Ideas, New Paradigms, and Reflections on Programming*. ACM, 2013.

[79] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP '98: European Conference on Object-Oriented Programming*. Springer, 1998.

[80] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution Patterns of Open-source Software Systems and Communities. In *IWPSE 2013: International Workshop on Principles of Software Evolution, co-located with ESUG 2002*. ACM, 2002.

[81] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. The MIT Press, July 1993.

[82] Yoshiki Ohshima, Takashi Yamamiya, Scott Wallace, and Andreas Raab. TinLizzie WYSIWiki and WikiPhone: Alternative Approaches to Asynchronous and Synchronous Collaboration on the Web. In *C5 '07: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2007.

[83] Shaul Oreg and Oded Nov. Exploring Motivations for Contributing to Open Source Initiatives: The Roles of Contribution Context and Personal Values. *Computers in Human Behavior*, 2008.

[84] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In Mehmet Akşit, editor, *Symposium on Software Architectures and Component Technology*. Kluwer, 2000.

[85] C Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2 edition, 2008.

[86] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual Smalltalk Images: Model and Applications. In *IWST 2013: International Workshop on Smalltalk Technology, co-located with ESUG 2013*. ACM, 2013.

[87] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM*, November 2009.

[88] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-Time Evolution through Explicit Meta-Objects. In *MRT '10: Workshop on Models@run.time, co-located with MODELS*. ACM, 2010.

[89] D. Riehle and H. Züllighoven. A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. *Pattern Languages of Program Design*, 1995.

[90] Michael Rüger. SuperSwiki-Bringing Collaboration to the Class Room. In *C5 '03: Conference on Creating, Connecting and Collaborating Through Computing*. IEEE, 2003.

[91] Michael Rymaszewski, Wagner James Au, Mark Wallace, Catherine Winters, Cory Ondrejka, Benjamin Batstone-Cunningham, and Philip Rosedale. *Second Life: The Official Guide*. SYBEX Inc., 2006.

[92] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *Transactions on Autonomous and Adaptive Systems*, May 2009.

[93] Hans Schippers, Michael Haupt, Robert Hirschfeld, and Dirk Janssens. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *SAC 2009: Symposium on Applied Computing*. ACM, 2009.

[94] Gregor Schmidt. ContextR & ContextWiki. Master's thesis, Hasso Plattner Institute, 2008.

[95] Christopher Schuster. Reification of Execution State in JavaScript Implementing the Lively Debugger. Master's thesis, Hasso Plattner Institute, April 2012.

[96] Richard Sennett. *The Craftsman*. Yale University Press, 2008.

[97] Jon Skinner. Sublime Text. Sofware, `http://www.sublimetext.com/`, 2011.

[98] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet – A Collaboration System Architecture. In *C5 '03: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2003.

[99] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 User Interface: Manifesting a System-Wide Vision of Concreteness, Uniformity, and Flexibility. In *OOPSLA '95: Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1995.

[100] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In *COOP'95 - Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7-11, 1995, Proceedings*. Springer, 1995.

[101] Randall B. Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 1996.

[102] Randall B. Smith, Mario Wolczko, and David Ungar. From Kansas to Oz: Collaborative Debugging When a Shared World Breaks. *Communications of the ACM*, April 1997.

[103] Randy Smith. Diff3 - Compare Three Files Line by Line. Software, `http://www.gnu.org/software/hello/manual/diff.html`, 1988.

[104] Richard Stallman. EMACS the Extensible, Customizable Self-documenting Display Editor. In *SIGPLAN SIGOA Symposium on Text Manipulation*. ACM, 1981.

[105] Richard Stallman. The GNU Manifesto, 1985.

[106] Guy L. Steele, Jr. An Overview of COMMON LISP. In *Symposium on LISP and Functional Programming*. ACM, 1982.

[107] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. Coexist: Overcoming Aversion to Change. In *DLS '12: Dynamic Languages Symposium*. ACM, 2012.

[108] Bastian Steinert, Michael Grünewald, Stefan Richter, Jens Lincke, and Robert Hirschfeld. Multi-user Multi-account Interaction in Groupware Supporting Single-display Collaboration. In *CollaborateCom '09: International Conference on Collaborative Computing, Networking, Applications and Worksharing*. IEEE, 2009.

[109] Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into Examples - Leveraging Tests for Program Comprehension. In *TESTCOM 2013: Conference on Testing of Communicating Systems*. Springer, 2009.

[110] Bastian Steinert, Marcel Taeumel, Jens Lincke, Tobias Pape, and Robert Hirschfeld. CodeTalk Conversations about Code. In *C5 '10: Conference on Creating, Connecting and Collaborating through Computing*. IEEE, 2010.

[111] Neal Stephenson. *In the Beginning... was the Command Line*. Avon Books, 1999.

[112] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. Interleaving of Modification and Use in Data-driven Tool Development. In *Onward! '14: Symposium for New Ideas,*

New Paradigms, and Reflections on Everything to do with Programming and Software. ACM, 2014.

[113] Antero Taivalsaari. Mashware: The Future of Web Applications. Technical report, Sun Microsystems, Inc., 2009.

[114] Éric Tanter. Beyond Static and Dynamic Scope. In *DLS '09: Dynamic Languages Symposium*. ACM, 2009.

[115] Éric Tanter. Execution Levels for Aspect-Oriented Programming. In *AOSD 2010: Conference on Aspect-Oriented Software Development*. ACM, 2010.

[116] Warren Teitelman and Larry Masinter. Interlisp Programming Environment. *Computer*, 1981.

[117] Lauritz Thamsen. Object Versioning for the Lively Kernel Preserving Access to Previous System States in an Object-oriented Programming System. Master's thesis, Hasso Plattner Institute, 2014.

[118] Astrid Thomschke. Diffing and Merging of Lively Kernel Parts. Bachelor Thesis, Hasso Plattner Institute, June 2012.

[119] Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive Aspects for the Web. In *AOSD 2010: Conference on Aspect-Oriented Software Development*. ACM, 2010.

[120] Linus Torvalds. Linus Torvalds on Git. Google Tech Talk, 2007.

[121] David Ungar. Annotating Objects for Transport to Other Worlds. In *OOPSLA '95: Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 1995.

[122] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 1991.

[123] David Ungar and Randall B. Smith. Self. In *HOPL III: Conference on History of Programming Languages*. ACM, 2007.

[124] Bret Victor. Inventing on Principle. Invited Talk at Canadian University Software Engineering Conference (CUSEC), January 2012.

[125] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-Oriented Programming: Beyond Layers. In *ICDL '07: International Conference on Dynamic Languages*. ACM, 2007.

[126] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. Worlds: Controlling the Scope of Side Effects. In *ECOOP 2011: European Conference on Object-Oriented Programming*. Springer, 2011.

[127] Danny Yoo. *Building Web Based Programming Environments for Functional Programming*. PhD thesis, Worcester Polytechnic Institute, 2012.

[128] Danny Yoo, Emmanuel Schanzer, and Shriram Krishnamurthi. WeScheme: The Browser is Your Programming Environment. In *ITiCSE 2011: Innovation and Technology in Computer Science Education*. ACM, 2011.

[129] Li Zhu, Ivan Vaghi, and Barbara Rita Barricelli. A Meta-reflective Wiki for Collaborative Design. In *WikiSym '11: International Symposium on Wikis and Open Collaboration*. ACM, 2011.