User-evolvable Tools in the Web

Jens Lincke Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam, Germany
{firstname.surname}@hpi.uni-potsdam.de

ABSTRACT

Self-supporting development environments like Smalltalk and Emacs can be used to directly evolve themselves, making their tools very malleable and adaptable. In Web-based software development environments users can collaborate in creating software without having to install the environment locally. Bringing these two together and making Web-based environments self-supportive is challenging, since users have to take care of to breaking the system, since there might be others using it also. Environments aimed at end-users usually provide a scripting level above the base system. Instead of providing users with a fixed set of tools, we propose to make the tools user-evolvable by building them as scriptable objects in a shared user editable repository. In our system, the Lively Kernel, the core system is developed using modules and classes, and on top of it users create active content by directly manipulating and scripting objects. By leveraging the scripting level for the development of tools themselves, we allow users to adapt their tools in a self-supporting way, without the need to invasively change the system's core. In this paper we show how development tools in Lively are collaboratively evolved. Tools can be directly explored, adapted, and published in a shared manner while they are being used.

1. INTRODUCTION

Authoring content in the Web has become increasingly popular and people often use Web-based applications when they have to collaboratively write a text or create a presentation. Sometimes they even use the Web to develop programs directly within a Web-browser [9, 17]. These Web-based development environments have to be created and evolved just as other programs too. Some development environments like Emacs and Smalltalk [6, 8] allow developers to directly change the code of the system and experience those changes at runtime. Developers get immediate feedback and the system evolves as they are using it. As an example, this allows developers to directly fix bugs as they occur.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

We use our Web-based development environment Lively Kernel [9, 13] to evolve itself. After an initial time of bootstrapping using external tools, we started programming with the Lively Kernel running on a Web-page to develop the system using only itself and the tools we created on top of it. As changing the system is open to all registered users, they can directly evolve the system while using it. However, this can be very dangerous: when changes of the users target the base code of the system, they can break the whole environment for everyone. But changes can be reverted, so it is not fatal when this accidentally happens, even though it may break the development flow of other people.

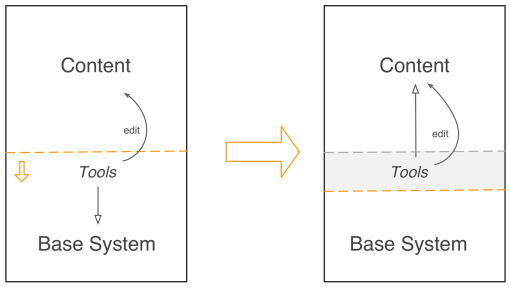

Figure 1: Lowering the barrier for end-users to adapt and evolve their tools.

In this paper we show how we moved the development tools from the base system into the realm of scriptable objects. This way, the tools can be cloned and developers can work on the clones, without breaking the tools they are currently using. By doing so, we lowered the barrier for end-users to adapt their tools to evolve the environment (as shown in Figure 1). By using the scripting level, they do not have to learn a second set of tools and concepts to adapt the environment for their needs.

The remainder of the paper is structured as followed. Section 2 motivates self-supporting development environments. Section 3 gives a general overview of a self-supporting Web-based development environment. Section 4 describes the approach of developing tools by scripting objects and publishing them in a shared repository. Section 5 discusses the approach. Section 6 evalutes how the approach was used in one collaborative environment and gives an illustrating example. Section 7 discusses related work and section 8 concludes.

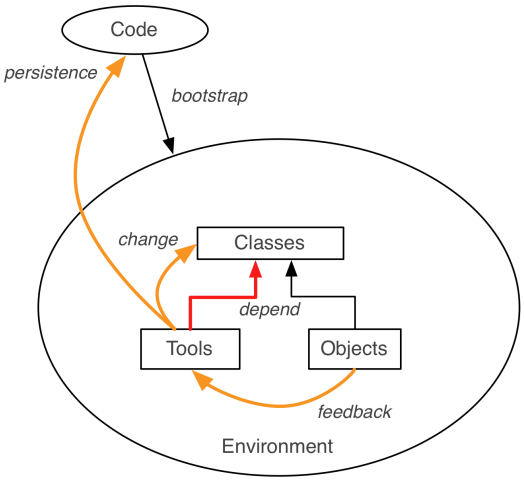

Figure 2: Abstract flow of changes and feedback in a self-supporting development environment

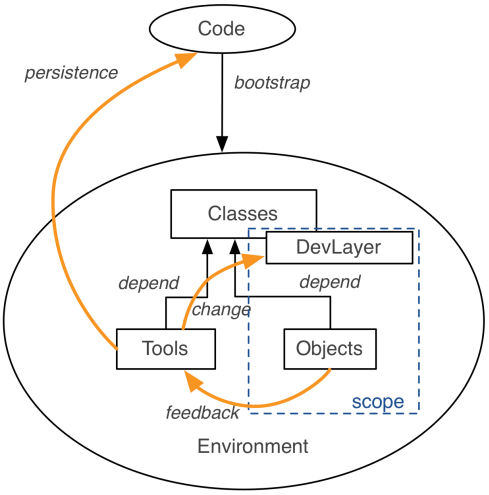

Figure 3: Flow of changes and feedback in a system were changes are encapsulated in scoped layers

2. MALLEABLE TOOLS IN SELF-SUPPORTING DEVELOPMENT ENVIRONMENTS

Beeing able to change programming tools directly in the same environment as they are used is common in dynamic programming environments such as Lisp [18] or Smalltalk [6]. A good description of adapting tools at runtime gives Martin Fowler in his personal summary of the goto-Arhus2011 conference about a presentation of the Moose Software visualization tool [2] by Tudor Gîrba [5], in which he adapted the visualization tool at runtime:

> "This philosophy sees the user of a tool as someone who is expected to tinker with the very guts of the tool in order to mould it to her needs. To achieve this, the tool not just opens up its source code for inspection and modification, but also makes it easy to make enhancements with rapid feedback. The most common example that springs to my mind is emacs - which encourages a user to treat it as a malleable tool." [4]

The concept of tool development in self-supporting development environments such as Squeak [8] is shown in Figure 2. The environment has to be bootstrapped at some point. The developer can then start creating and modifying meta-objects such as classes. Because this happens at run-time, developers will get feedback from the objects while they are programming. The changes to the meta-objects are typically also persisted outside of the development environment in some kind of code repository. Since tools, meta-objects, and objects are in the same environment, there are no extra levels of indirection: the tools can be simple and powerful. But at the same time having no levels of indirection can be problematic and even fatal if the developers want to work on objects such as core classes that directly or indirectly affect the tools themselves. The tools depend on those core classes. Introducing an error or having an intermediate broken state is fatal for the whole environment. That is why developers have to be careful, when they are modifying those shared objects.

In collaborative Wiki-like development environment, such as the Lively Kernel [9], programming mechanisms are needed, that do not break the whole system for everyone when users are adapting tools. We showed in previous work [12] how layers of context-oriented programming (COP) [7, 1] can be used to separate and scope changes as needed. This allows new features to adapt base code directly but safely and to be gradually deployed [11]. As shown in Figure 3, the shared behavior in classes is not directly changed, but encapsulated in a *DevLayer* that can be scoped to objects and object structures. This way, the tools that are needed for development can be excluded from possible malicious changes during development.

3. CREATING ACTIVE CONTENT IN LIVELY KERNEL

This section gives an overview of Lively Kernel, a self-supporting Web-based development environment. The Lively Kernel (often just abbreviated as Lively) is built using Web technology, but it is deriving from a Smalltalk background, the terminology is different. Lively serves as a platform to experiment with novel programming language concepts and tool development approaches. For an introduction to Lively Kernel see [9, 10, 13].

3.1 Morphs

Graphical objects in the Lively Kernel are called *morphs*. Morphs are hierarchical structures that are composed of other morphs. Applications and tools in the Lively Kernel are published as serializations of morphs. These objects can be directly manipulated and contain their own behavior. Cloning objects allows users to experiment and develop

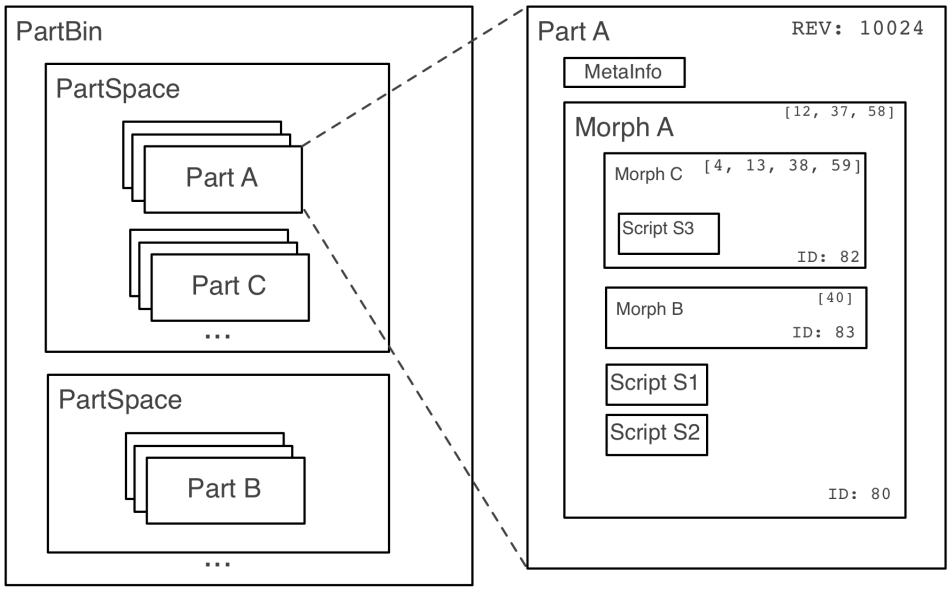

Figure 4: Morphs are serialized as Parts, grouped in PartSpaces inside the PartsBin

safely on copies — publishing them when satisfied with their changes. Morphs are graphical objects users can directly interact with. In Morphic [15, 16, 14] based architectures everything can be taken apart, inspected, and put together in a new way. Morphs can have different shapes and other morphs as submorphs. A morph can only have one parent. In Lively Kernel, morphs can have a name and they get unique identifiers. Duplicating a morph creates a new morph with a new unique identifier. The root of the scene graph in the Lively Kernel is called *world*. The world itself is also a morph and can be inspected and changed by the user in the same way every other morph can.

3.2 Parts

Parts are graphical objects 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 contains not only a single object but also all the objects that belong to this object, including all the submorphs. A serialized part is published under a name in a PartSpace as shown in Figure 4. Some meta-information and a preview of that part are stored separately. By publishing a part, a new revision is created and users can revert changes as needed. All the parts are stored in a *PartsBin* and are grouped in *PartsSpaces* which also provide a name-space. Morphs that are used to work on other morphs are called tools. They can range from a simple *ColorChooser* or to complex tools such as an *ObjectEditor*. The screenshot in Figure 5 shows the *Tools PartSpace* in an PartsBin with a selected ObjectEditor and its details.

3.3 Direct Manipulation and Scripting

A morph's properties like *style*, *position*, *extent* or *text content* can be directly changed using Lively's *halo* user interface and tools like the *StyleEditor* that can be invoked from the halo. See Figure 8 for a morph with halo. Parts in Lively consist usually of several other morphs forming a tree structure also called *scene graph*. The composition of morphs can be changed via drag and drop or by using the halo. Since scripting in the Lively Kernel relies on *dynamic name lookup* in the scene graph, changing the composition can not only change the appearance but also the behavior of morphs. Due to the JavaScript object model, a morphs's scripts are also just properties, which happen to be functions. So, by overriding methods of an object's class, an object can directly adapt behavior of the base system.

4. DEVELOPING TOOLS AS SCRIPTED OBJECTS

The Lively Kernel provides two levels of editing:

1. The base level, which allows editing modules and classes that indirectly change the behavior of objects [10].

2. The scripting level, which allows editing object state such as properties, style, and composition as well as editing object behavior such as scripts and connections [13].

The first generation of Lively Kernel tools were developed with the first level. Using classical object-oriented design, employing classes and patterns, and separating the data model from the user interface. With this approach developers can get immediate feedback and it allowed to evolve the system as they are using it. But as shown in Figure 2, it is easy to break the system accidentally while doing this, because the changes also affect the tools that are used to make the changes.

Developing tools as scripted objects allows users to directly change the graphical objects, their style, the composi-

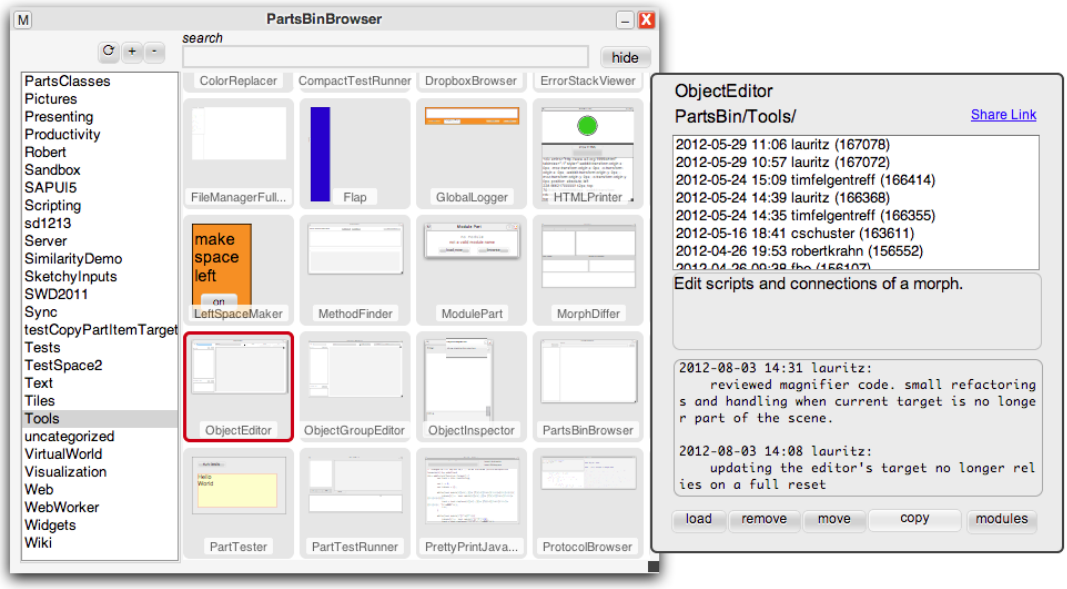

Figure 5: PartsBin with open ObjectEditor in Tool category showing details

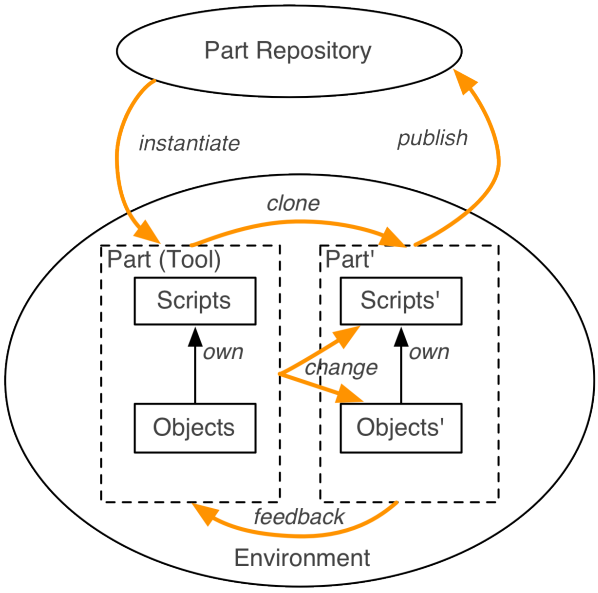

Figure 6: Flow of changes and feedback in a system were changes are encapsulated in scoped layers

tion and their scripted behavior. Instead of having to adapt the behavior either in the class or with development layers, the users can shape the tools using direct manipulation too. Figure 6 shows that the dangerous cycle of changing a behavior and breaking a tool is broken by cloning the part first. Developers can change scripts and objects directly in a cloned part tool, without breaking the tool itself because it has its own set of scripts and state. But the feedback loop is still as short and tools can directly interact with the objects they manipulate, because they stay in the same environment. For example 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. Thus, adapting tools does not necessarily mean programming.

4.1 Persistent Object-specific Behavior

Object specific scripts combine both material and procedural forms of authoring content [13]. Self [21] explored this unification of state and behavior many years ago and our implementation language JavaScript builds on the same underlying concepts. One difference of the unification of behavior and state in JavaScript and Self is that the behavior is transient in JavaScript since normal JavaScript closures cannot be fully serialized. The standard JSON serialization mechanism does not support objects inheriting behavior (classes, prototypes). It has no concept for serializing arbitrary references in an object graph. It only serializes a tree formed of nested objects, arrays, numbers, and strings. We solved this problem by using our own version of closures when adding scripts to objects. JavaScript functions are closures that are bound to variables in the execution stack of the virtual machine. By design, a JavaScript program cannot introspect such state of Function objects. We solved this limitations by defining persistent object specific behavior using our own “addScript” method. This method ensures that functions can be serialized and do not contain bound

variables that are not accessible after deserialization any more. JavaScript does also not support multiple threads. Therefore we do not have to serialize any running code in the background. Stepping scripts, a form of pseudo parallel programming used in Lively Kernel can be stopped and restarted for (de-)serialization as needed.

4.2 Forward References From Base System Into PartsBin

Some core tools have to be integrated into the base system. The "StyleEditor", "Inspector" and "ObjectEditor" are tools that can be opened through the *halo* of an object. Other parts can be opened directly through the world's context menu. However, the base system cannot guarantee that these parts will be available or working correctly as they do not belong to the base system. If someone breaks a tools, a previous version can always be restored the same way as a bad commit in a Wiki can be reverted. We thought of referencing only known stable versions of a tool from within the base system, but we would then have to update these stable links, slowing down the evolution. If we will run into problems with unstable tools, we might consider falling back to this more conservative approach for core tools.

4.3 Cloning and Derivation History

When a morph gets duplicated, a new universally unique identifier (UUID) is generated. The old identifier is remembered in the derivation history of that morph. When using object cloning for gaining safety in live coding sessions, redundancy is generated which is hard to deal with manually. But by tracking the copy history of objects, we can mitigate this problem. For example, when a tool is developed by two different users at the same time and their changes to the tool have to be merged. The objects got new identifiers, because they were copied from the PartsBin, so comparing two instances is difficult. But by capturing the copy history of whole object structures, we can diff and merge such complex graphical objects again.

4.4 Duplicated Source vs. Copied Objects

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 together. If the copied piece of code should be changed, developers have to locate every copy and apply the change manually – if they still have control over them.

Objects consist of identity, state and behavior, while source code is just text. When we clone objects, we can record this process in the objects themselves. In Lively, all graphical objects know the identifiers of all the objects they were copied from. When copying text, this information can get lost, since the copying can only be traced by matching the text.

5. DISCUSSION

This section discusses features and problems or our approach.

5.1 Problem of Object Clones

The redundancy introduced by cloning object can become a problem: When a new feature is prototyped in an object and this object is duplicated and used in many places, 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 of bug fixes manually or using scripts. We experimented with a special version of the object editor, that allowed to edit many instances at once [3], 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. Automating this process or replacing it with smart object migrations is part of our future work.

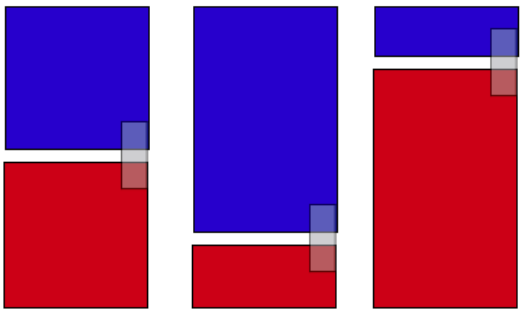

Figure 7: A SplitterMorph adjusts the extent of two adjacent morphs when dragged

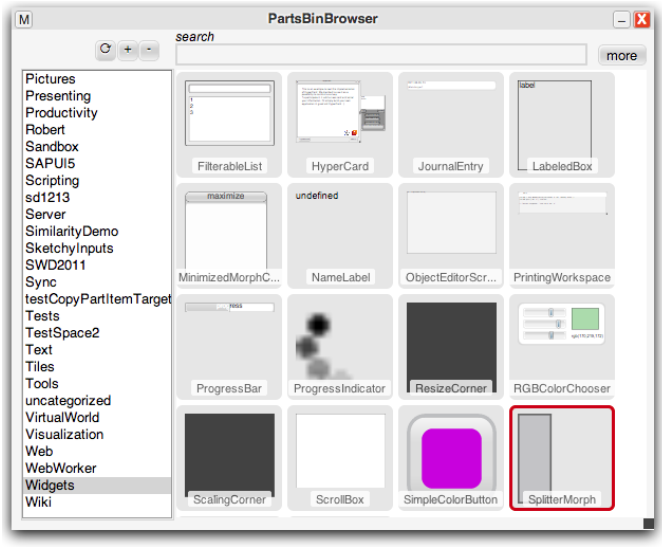

Figure 9: Published SplitterMorph in the PartsBin

5.2 Scripts in Nested Objects

Our current ObjectEditor allows only to directly edit one object at a time. The scripts of submorphs have to be edited with separate editors each. This has lead to a style of programming where a lot of scripts are attached to the first level object and not distributed over all the involved objects as in a typical Object-oriented design.

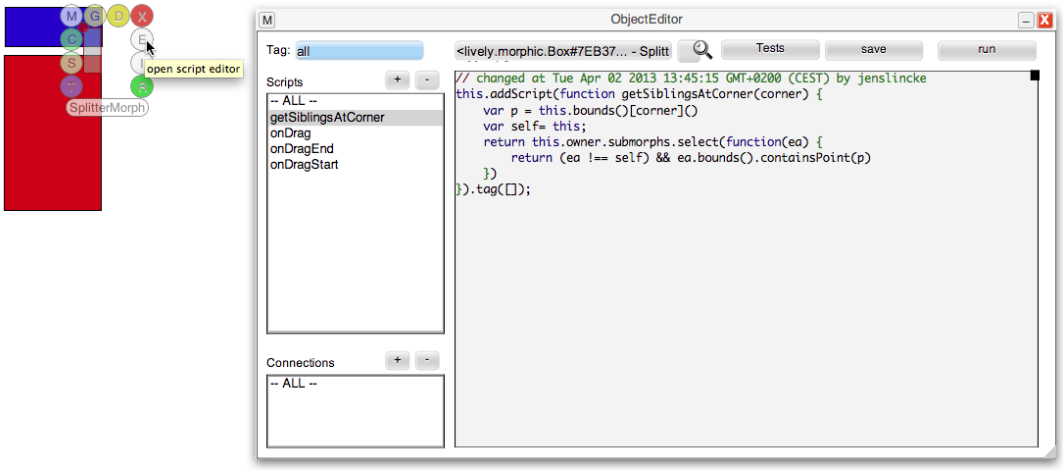

Figure 8: Editing the SplitterMorph with Halo and ObjectEditor

5.3 Source Code vs. Objects

Manipulating and scripting objects directly – in contrast to making a plan to automatically build them – may result in a lot of cruft in the objects that remains from the construction process. When manipulating objects directly, we tend to only work on the things we see. And naturally, there are things we are not as aware of as others: Property entries that are not used any more, whole scripts, or unused objects that are still referenced.

Working with source code is more abstract than working with objects directly. Since developers do not interact with the objects, they can be thrown away, if they are not valuable user data, to clean up a system. When editing only source code, it is much easier to keep the code clean, because the representation is more condensed and everything is visible. 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 that persists 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 throw away their images and bootstrap their projects from fresh images. When there is no source representation as in our approach the objects can not be thrown but have to be cleaned up by the developers.

5.4 Prototyping

In Webwerkstatt, tools evolve regularly from of code snippets and one-liners that were used even before the first draft was finished. The feedback of using the tools on real user data drives the development of the tools right from the beginning. If the tool seems not to be useful it is easy to abandon since all scripting parts in Lively are self-contained and not part of the core system. Discontinued prototypes and unfinished ideas do not have to be immediately discarded in Lively, as they do not pollute the base system anymore. The fast changing tool prototypes are separated from the more stable code of the core system. Prototypes are different from sketches and specifications. They are usable and users can get insights from working with them, playing around with them, and changing them. Further, prototypes in Lively often look and feel like prototypes. They feel rough and still malleable and are therefore less likely to be mistaken for the finished product.

5.5 Immediate Feedback

Change should be immediate in the sense that it should reflect to the user. Changing the color of an object should be visible to the user. Changing the behavior of an object should also be perceptible. If users make mistakes, an error should be displayed as soon as possible. It is not always easy to provide immediate feedback. It gets harder if what is changed is more abstract and more dynamic. When the change is only a modification of a property such as the color or position, it is very easy to reflect the change in the system. Bret Victor demonstrated ideas on how such immediate feedback could be given in easy to visualize systems [22], but it is very hard to integrate such feedback in general purpose development environments. In the Lively Kernel we are relatively fortunate because we are working with graphical objects that lend themselves naturally to such immediate feedback. But we also want that kind of feedback for tool development. When following best practices in software engineering such as test-driven development, running tests is the immediate feedback that drives developers. When we let users create their tools and they are working on their data, they can and should use their own data as testbeds for their tools.

5.6 Meta-circularity in Tool Development

Meta-circularity is a double-edged sword. Editing each other's worlds and parts can break the system for all. What if someone accidentally publishes a broken version of the ObjectEditor to the PartsBin? This can happen and as a result nobody can edit scripts of objects any more. Our approach does not limit these changes and parts can break, but working versions will still be available in the repository. We follow the wiki principle that everyone can change anything but malicious changes can be undone [13].

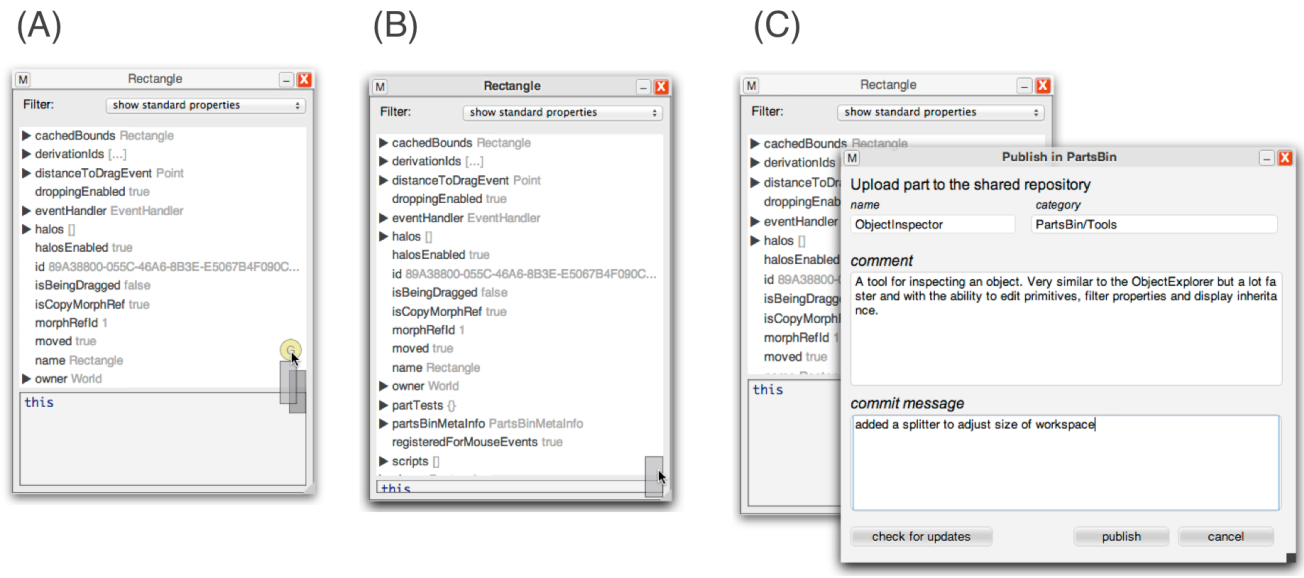

Figure 10: Adopting the Inspector using a SplitterMorph

6. EVALUATION: USER CREATED TOOLS IN WEBWERKSTATT

This section gives a short overview of how much the PartsBin was used in terms of absolute numbers and reports two examples from how Parts are used to adapt tools in Webwerkstatt.

Starting in April 2011 we used parts as our main approach of developing tools for Lively Kernel in Webwerkstatt. Our PartsBin is located in the Webwerkstatt *SVN* repository [1] and in March 2013 it contained more than 667 parts stored in 60 PartsSpaces, which serve as name-spaces and categories. Each time a user publishes a part from a World, a new revision is created. The users created 5690 revisions uploading parts. The granularity of increments per upload varies usually a lot and depends also on the user. But usually many revisions correlate with much development effort, for example the ObjectEditor (158 revisions) and the PartsBinBrowser (100 revisions). They are important tools and also the parts with the most revisions. These two parts belong to the *Tools* PartSpace, which contains 50 other tools with in total 683 revisions. As a result of our openness, 21 different authors uploaded at least one revision into that PartSpace. From the total 941 revisions, 761 were produced by 5 authors, who also happen to be core developers of the Lively Kernel project.

In the future we want to look at individual parts and analyze their derivation history and how they are composed of other parts. We expect to find both, outdated parts that are still in use and parts that were reused and evolved in interesting ways.

6.1 Example: Developing a Splitter Morph

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 extend of two objects as shown in Figure 7. When the splitter is dragged down, the upper blue morph gets bigger and the red morph gets smaller. When it is dragged up again, the process is reversed. The interesting approach of the SplitterMorph is that it does not need an explicit configuration. It is only required to be put besides the two objects like a patch that connects them. By looking at the bounds of the object, it can detect them and adjust them appropriately. See the script in the ObjectEditor in Figure8. There is no explicit configuration required. The developers, realized that they build something useful and published their morph in the PartsBin for others to play with and reuse as shown in Figure 9.

6.2 Example: Evolving the Inspector by Adding the Splitter Morph

Later, some users see the new SplitterMorph and realize that the Inspector, a core development tool, misses this feature and decide to change it. They open a normal inspector and drag the SplitterMorph to the right position (see Figure 10) (A). They test the new behavior (B) by interacting with the Inspector and publish it again to the PartsBin (C). From then on when users open an inspector they can adjust it using the splitter morph. This demonstrates that publishing operations do not alway require to actually type code in Lively. Sometimes object composition can be a powerful mechanism to develop objects.

7. RELATED WORK

Smalltalk [6] and especially Squeak [8] with its Morphic [15] user interface, provide extremely malleable tools since everything can be changed at runtime. Hence, directly adapting the behavior of objects can sometimes be difficult, since the behavior is defined in classes. Further, these environments are single user environments and breaking a system with a bad change affects only one person.

Self [21] is an object-oriented programming environment

[1] http://lively-kernel.org/repository/webwerkstatt

that allows for 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 [20], but Self does not come with shared repositories of objects. Instead it approaches the problem of collaborative development in a shared world. They propose to debug small problems directly in the world and when bigger problems occur, they debug the default world from a special debugging world [19]. Their concern is more on real-time collaboration compared to the asynchronous wiki-like collaboration in our approach.

Our approach of cloning objects to directly make changes and the usage of an online repository of objects to collaborate is somewhat similar to GitHub[2]. GitHub provides a very direct way of changing things. People can fork a project and make changes as they like, since they have full control over the source code. The difference is that GitHub does only source code and revision management, where Lively is a development and runtime environment.

Web-based development environments like Cloud9[3] can be used to develop themselves. But the process is not direct: one instance modifies the other instance. This does not allow to evolve the system while it is being used resulting in longer feedback loops.

8. CONCLUSION

By lifting the development process of our self-sustaining system one level up from programmer concepts like modules and classes to direct editing of end-user-accessible objects, we make it easy to prototype and adapt tools without accidentally breaking the system.

We value the robustness of the introduced redundancy higher than the problems. We deliberately increase the redundancy to make it harder to break something, even if this means that we loose the power of abstraction that established software engineering practices give to us.

In future work we want to address how bugs in scripts can be fixed or new features added to already distributed objects. A solution can be to trace objects using their copy history and migrate them as needed.

9. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. In *Proceedings of the Workshop on Context-oriented Programming (COP), co-located with ECOOP 2009, Genoa, Italy*. ACM, 2009.

[2] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an Agile Reengineering Environment. *SIGSOFT Softw. Eng. Notes*, 30(5):99–102, Sept. 2005.

[3] T. Felgentreff, P. Tessenow, and L. Thamsen. Lively Groups — Shared Behavior in a World of Objects. Seminar Report, 2012.

[4] M. Fowler. Blog Entry: gotoAarhus2011, 26 October 2011.

[5] T. Girba. Humane Assessment with Moose, 2011.

[6] A. Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

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

[8] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices*, 32(10):318–326, 1997.

[9] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel A Self-Supporting System on a Web Page. In *S3 2008*, LNCS 5146. Springer-Verlag Berlin Heidelberg, 2008.

[10] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09*. ACM, 2009.

[11] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, 2011.

[12] J. Lincke and R. Hirschfeld. Scoping changes in self-supporting development environments using context-oriented programming. In *Proceedings of the International Workshop on Context-Oriented Programming*, COP '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[13] J. Lincke, R. Krahn, D. Ingalls, M. Röder, and R. Hirschfeld. The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content. In *Hawaii International Conference on System Sciences*, volume 0, pages 693–701, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[14] J. Maloney. *Morphic: The Self User Interface Framework*. SUN Microsystems, 1995.

[15] J. Maloney. An introduction to morphic: The squeak user interface framework. *Squeak: OpenPersonal Computing and Multimedia*, 2001.

[16] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28. ACM, 1995.

[17] M. Nordio, H. Estler, C. A. Furia, B. Meyer, et al. Collaborative software development on the web. *arXiv preprint arXiv:1105.0768*, 2011.

[18] E. Sandewall. Programming in an Interactive Environment: the "Lisp" Experience. *ACM Comput. Surv.*, 10(1):35–71, Mar. 1978.

[19] R. B. Smith, M. Wolczko, and D. Ungar. From kansas to oz: collaborative debugging when a shared world breaks. *Commun. ACM*, 40(4):72–78, Apr. 1997.

[20] D. Ungar. Annotating Objects for Transport to Other Worlds. *SIGPLAN Not.*, 30(10):73–87, 1995.

[21] D. Ungar and R. B. Smith. Self: The Power of Simplicity. *Lisp and symbolic computation*, 4(3):187–205, 1991.

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

[2] https://github.com/

[3] https://c9.io/