Lively Groups: Shared Behavior in a World of Objects without Classes or Prototypes

Tim Felgentreff Jens Lincke
Robert Hirschfeld
Hasso Plattner Institute,
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

Lauritz Thamsen
Technische Universität Berlin, Germany
lauritz.thamsen@tu-berlin.de

Abstract

Development environments which aim to provide short feedback loops to developers must strike a balance between immediacy and the ability to abstract and reuse behavioral modules. The Lively Kernel, a self-supporting, browser-based environment for explorative development supports standard object-oriented programming with classes or prototypes, but also a more immediate, object-centric approach for modifying and programming visible objects directly. This allows users to quickly create graphical prototypes with concrete objects.

However, when developing with the object-centric approach, sharing behavior between similar objects becomes cumbersome. Developers must choose to either abstract behavior into classes, scatter code across collaborating objects, or to manually copy code between multiple objects. That is, they must choose between less concrete development, reduced maintainability, or code duplication.

In this paper, we propose Lively Groups, an extension to the object-centric development tools of Lively to work on multiple concrete objects. With Lively Groups, developers may dynamically organize live objects that share behavior using tags. They can then modify and program such groups as if they were single objects. Our approach scales the Lively Kernel's explorative development approach from one to many objects, while preserving the maintainability of abstractions and the immediacy of concrete objects.

FPW'15, October 26, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3905-6/15/10...$15.00
http://dx.doi.org/10.1145/2846656.2846659

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Programming Environments—Interactive environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

Keywords Web Applications, Interactive Systems, Exploratory Development, Lively Kernel

1. Introduction

A common goal of development environments is to alleviate development by providing feedback early. Such feedback mechanisms range from syntax checking and automatic test execution, to integration of the development environment and applications into a single runtime. Such *live programming environments* allow developers to experiment in and interact with the system while they are developing. Notable environments include Self [20], Squeak/Smalltalk [7], Lisp [18], and the Lively Kernel [6, 8], which provide varying degrees of support for immediately seeing ones actions and for developing the system from within itself.

The Lively Kernel is a collaborative web-based development environment. Lively's development tools allow programmers to change applications from within the same Web page and immediately see the results. Developers can either work on abstract behavior and classes or on concrete objects. Both approaches are live in the web page, but working on abstract behavior requires cognitive effort to map the abstractions to the concrete, visible objects and to makes it more difficult to explore while using the system. In contrast, the direct interaction with objects allows short feedback cycles [12], but currently only works one object at a time. It is, however, more suited to introduce inexperienced developers to programming interactive applications on the Web, especially such applications where reliability and correctness are subject to best-effort rather than strict requirements.

To implement behavior common to multiple objects, developers in Lively Kernel have three choices: either, abstract common functionality into classes that define shared behavior these objects, scatter code across collaborating objects, or manually copy code between objects. The first option requires mental effort to find the right decomposition [19], the second option breaks encapsulation [17], while the third reduces maintainability [11].

To overcome this challenge, but keep the benefits of immediate, object-centric development, we propose an extension of Lively's tools. Our extension is based around tagging objects to work on groups of objects as if they are one. Developers can tag objects by clicking their visual representations, by selecting nodes from the scene-graph, or by evaluating program queries. Tags are added as necessary when developers want to modify or add behavior to a group of objects, kept as long as such grouping seems useful, and removed when objects are no longer seen in the same context.

Our approach is motivated by physical construction, where similar parts are manufactured with common properties and maybe from a common template, but each part is self-contained and can, if needed, evolve independently. Thus, our approach manages shared behavior without dictating the program decomposition, while maintaining the immediacy of live objects. It provides an alternative to sharing behavior between multiple objects, without reverting to classes or common prototypes. We think the tooling we present here could be done for class-based or prototypical languages. However, we think that our tooling in conjunction with a flexible and immediate mechanism for directly working on multiple objects in a group, having objects shared across different groups for different contexts, and also for removing them again without having to change any code can be a boost to productivity. When a number of objects share a class, and we want to add specific behavior to just one of those objects, should we have to create a subclass? What if we are not sure the specific behavior will really stay? The cognitive overhead of adding a subclass just for one object, which may be removed again immediately inhibits experimentation and explorative development. We feel that the mental hurdle to adding or removing an object to or from a group is much less than what it may be when we consider classes or prototypes for sharing behavior.

Thus, the contributions of our work are:

- We present a lightweight approach to share behavior based on tagging that does not require committing to a specific decomposition, and that allows objects to enter and leave groups as needed. We explore our approach as an alternative way of sharing behavior using groups, where groups do not represent a meta-level object (like classes or prototypes), but are merely tags on an object that can be easily added or removed without immediately affecting the concrete behavior of the objects. This is done by going against the common wisdom of not copying code, but instead duplicating actions and behavior between objects in a group.
- We present tooling to visually and programmatically select and group live objects for development in a specific context, and which makes code that is duplicated between objects in a group manageable. Our tools are created with inexperienced developers in mind, and are meant to support the kind of small, interactive applications that are common on the Web today, and for which best-effort computing is sufficient.

The remainder of this paper is organized as follows. Section 2 gives a short overview of the Lively Kernel environment, demonstrates its object-centric development approach with an example, and identifies challenges that arise when concrete objects share behavior. Section 3 introduces our approach to sharing behavior between groups of objects, while Section 4 describes our implementation in Lively. Section 5 identifies current limitations and proposes future work. Section 6 presents related work, while Section 7 concludes this paper.

2. Object-centric Development: The Lively Kernel

The Lively Kernel allows browser-based, object-centric development of Web applications, including direct manipulation, object specific behavior and object serialization. To exemplify this object-centric development approach, we present a game built entirely with objects.

2.1 The Lively Kernel

Lively's main characteristics include the integration of design-time and runtime, object-centric development tools, the implementation of the Morphic User Interface Construction Environment [15], and a serialization mechanism to store objects persistently. It further supplies a module system that includes classes with single inheritance, traits and context-oriented layers [13]. The Kernel itself and applications are based on these modules, however, developers can create new applications as, for example, development tools, by composing and editing concrete objects without creating modules. As first explored in the Self programming language and environment [21], this directness and livelyness shortens the development cycle [20].

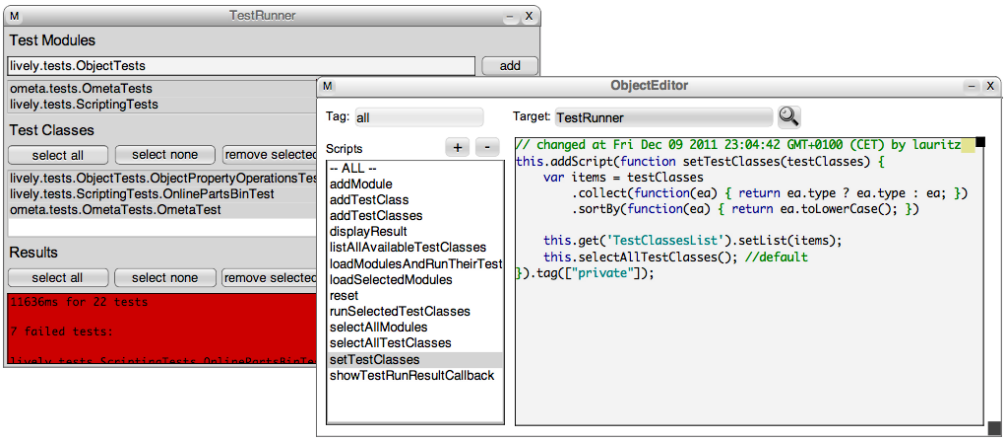

Figure 1. Lively's Object Editor modifies a test runner built from Parts.

The Morphic architecture allows programmers to directly manipulate and compose Morphs. It provides handles for basic graphic modification as resizing, repositioning, and rotating Morphs, but also ways to add them as children to other Morphs. Developers can add object-specific behavior to Morphs and try out changes to objects with immediate feedback. During development, the edited object provides a concrete context for the code. Lively's object-centric code editor is called *Object Editor*, shown in Figure 1. It shows all scripts of a certain object and allows developers to add and alter scripts. It enables developers to experiment with these scripts, as all statements that do not depend on parameters or temporaries can be evaluated directly on the editor's target object. Finally, Lively's object serialization enables a Web-based object repository [14], called *Parts Bin*, into which developers publish their Morphic creations. Such published *Parts* are available to other developers, effectively making the Parts Bin a library of visual components that developers can use and reuse.

2.2 Object-centric Development by Example

Our game, shown in Figure 2 and developed entirely with objects, features a two-dimensional map where a player character ① and several non-player characters (NPCs) can move about. It supports terrains and obstacles, some of which—like water ② or trees ③—are impassable. The goal of the game is to talk to all NPCs on the map ④ and defeat them in debates by choosing the best insults from multiple-choice menus and, thereby, bringing the morale meter ⑤ of the opponent down to zero.

Each feature of the game is implemented on basic Morphs in one of three ways: functionality that is required once, belongs to one object. If functionality is required on various different Morphs, however, we implemented it on a central

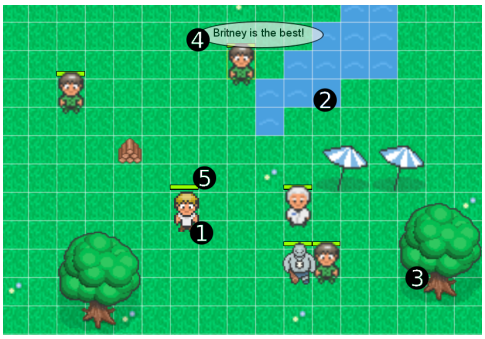

Figure 2. *Freedom of Speech* — A debating adventure game built from Parts.

component. For example, the game object implements an image loading function available to all objects of our game. If functionality is required by various similar Morphs, we implemented it by composing multiple Morphs, some of which are invisible and use visible Morphs as *costumes*, as in Squeak Etoys [9]. The invisible Morphs contain the shared behavior, while the visible Morphs implement distinct functionality and provide individual appearance. For example, each character is built from a transparent Morph that provides path-finding, user interaction, and debating, while three visible Morphs—a morale bar, a character picture, and a speech bubble—implement distinct behavior.

2.3 Problems Found

The implementation of the game's features exemplifies challenges in object-centric development of many objects that share behavior. Multiple of our objects require shared as well as distinct functionality. The object-centric development approach of Lively, derived from physical construction and easy to approach for inexperienced developers, suddenly breaks

down. In these situations, we recognize four different implementations:

Duplication Developers can copy the shared functions to all characters. While this approach maintains immediacy and concreteness, it duplicates code. This duplication impedes maintainability, as developers have to remember all occurrences when editing copied functions. Additionally, experimenting on functionality will only change one object with no convenient mechanism to propagate experiments to all similar objects. This approach, however, incrementally scales the development difficulty with the size of the program.

Abstraction Developers can abstract common functionality into modules that define shared behavior. This necessitates integrating existing objects into a module system and reduces feedback immediacy, as the code no longer has a concrete context in form of a specific instance. Abstraction also represents a point where the learning curve for inexperienced developers suddenly becomes very steep, since they might have to learn about new concepts like classes or prototypes.

Externalization Developers can implement procedures required by multiple objects at an external location. Objects call those routines and pass themselves as arguments. This impedes code comprehension as it trades Modular Understandability [16] for code re-use.

Scattering Finally, developers may choose the costume approach, in which they implement common functionality on invisible Morphs that compose visible Morphs to customize their appearance and functionality. However, this scatters the code belonging to one logical domain entities across multiple objects and the scene-graph. Additionally, costumes are highly coupled to their invisible base Morphs. That is, while Lively developers expect each object in the Parts Bin to be self-sufficient, costume Parts are not usable by themselves.

The identified implementations impose a choice between unmanaged duplication, reduced immediacy, diminished code comprehension, or scattered code. We need an approach to share behavior in a world objects that scales Lively Kernel's explorative development approach from one to many objects, while preserving the maintainability of abstractions and the immediacy of concrete objects.

3. Object Groups Approach

Different objects implement overlapping responsibilities. Developers modularize such responsibilities to share behavior between objects, however, multiple-inheritance, traits, and layers require either upfront planning or subsequent refactoring. Either way, developers loose the immediacy of concrete object development and reason on a more abstract level.

In our approach, developers can combine concrete objects into concrete groups. Each group represents a specific responsibility. Objects can be assigned to groups dynamically, allowing programmers to develop objects with immediate feedback, and modularize them on-demand to improve maintainability.

3.1 Lively Groups

We provide group operations for Lively's object-centric development operations, which include, first, evaluating code-snippets in the context of the target object, and second, adding functions to the target directly.

Direct evaluation for groups works differently depending on whether the evaluated code references `this` or not. Evaluations without a self-reference execute only once, but self-referential code snippets execute for each member of the group. The results of all evaluations are collected into a result set and is the return value for the developer. This enables developers to change properties of all members in one action, by referencing `this`.

Interactive evaluation of code snippets may throw errors. When editing a single object, uncaught errors abort the computation and the runtime unwinds the stack. However, in a group, the computation may fail for a subset of the objects. So, for group evaluations, we catch intermittent errors and return an exception object as part of the result set.

In Lively, developers use interactive evaluation to add functions to objects as well. To add a function to a group, each member of the group receives a copy of the same function. Even though this duplicates the code, the function can be edited and modified for all members at once in the context of the group.

3.2 Creating groups

Developers can create groups on-demand using one of the following three mechanisms:

Direct Selection Developers can explicitly point at visible objects to combine them into a group using multi-selection techniques, including clicking on multiple objects or dragging a selection rectangle around objects.

Scene-graph Selection As some members of a group may be off screen, invisible, obstructed or too small, direct selection is sometimes difficult or impossible. In such cases, developers may select objects in an alternative representation of the scene-graph, a textual tree-view.

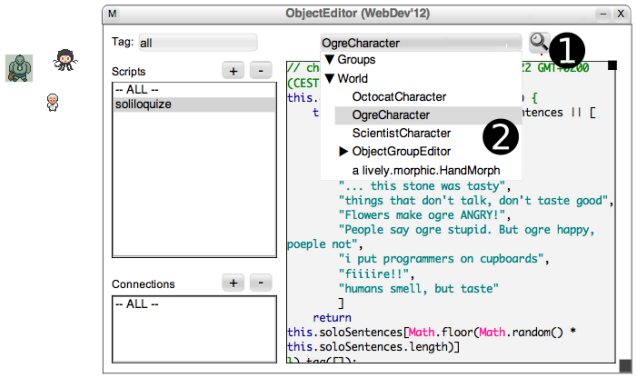

Figure 3. Extended Object Editor includes a scene-graph browser and an object selection tool.

Programmatic Selection Some groups may include a large number of objects, making it infeasible to select each member manually. Furthermore, some groups may be characterized by object properties that may not be visible. In such cases, developers can define groups through code snippets that yield lists of objects.

Programmers may use groups transiently, or assign labels to persist the connection from member objects to the group. To examine the parts of a group, developers can highlight all members from the editor, adding a colored overlay to their visual representation in world.

4. Implementation in the Lively Kernel

We evaluated our approach with a tool-based solution that extends the Object Editor of Lively. Our Object Editor allows to define groups visually and programatically. It stores the group as property on all group members and edits them simultaneously.

Defining Groups Our extended Object Editor, shown in Figure 3, supports all three selection mechanisms of our approach: direct, scene-graph, and programmatic selection.

For the direct selection, the editor provides a tool ① to enter *selection mode*, in which developers define groups by clicking on the visual representation of objects. Furthermore, we modified the edit button of Lively's selection tool to open our editor on the selected morphs. To select from the scene-graph, we added a tree-view ② that shows the composition hierarchy of the world. Developers can browse the scene and click to select. For programmatic selection, we modified the global function `edit`, which opens an editor on the argument, to treat collections of targets as a group. The global edit function is also invoked by Lively's shortcuts, providing convenient access. Furthermore, since group names are just properties of objects, any program can also filter objects with

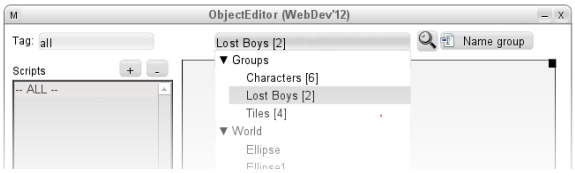

Figure 4. Our Object Editor shows a list of available groups.

respect to their group names to select custom slices from groups to edit. However, exposing this power in the tools could quickly make a program unmanageable, as we discuss briefly in Section 5.

Saving Groups Defined groups are initially anonymous and transient. Naming a group persists it. In our prototype groups are tags. When developers name a group, the editor attaches the name to the group members. Lively serializes this property as part of the object and, thereby, stores group membership, for example, in the PartsBin. The editor collect groups available in a world by iterating over visible objects and offers the set of unique group names Figure 4 to developers.

Editing Groups When developers open groups in the Object Editor, it determines the common group functions by comparing their code and omits all other functions from its script list. When developers evaluate code in the editor's script pane, we check the code for occurrences of `this`. If there is none, the code is evaluated once, otherwise for each object in the group. We guard such group evaluations with a `try/catch` statement to continue evaluations on subsequent group members even on intermittent errors.

Similarly, saving a function executes `addScript` on each object, effectively duplicating the function.

5. Directions for Future Work

Our approach, in its essence, is an attempt to smooth out the learning curve for inexperienced developers that started development with object-centric development tools like those available in Lively. It allows the step from editing single objects to multiple objects not to be abstract, keeping the metaphor of real objects that are manipulated directly. We propose to implement our approach for other Lively tools as well. Moreover, we want to explore first-class groups, functions owned by groups, and automatic group discovery.

Classes as "Fit and Finish" Provided we can scale our direct-manipulation approach to development using groups, we must ask ourselves if that makes abstraction mechanisms such as classes or prototypes irrelevant. Abstraction mechanisms give us structure (which can aid our thinking), and many patterns and idioms have been created around object-

oriented programs with traditional abstraction mechanisms that help communicate intent of code entities and structure working on programs in teams. Classes also make it clear how to get "clean" objects of a certain type, something notoriously difficult in object-centric development.

A possible approach could be to provide a path from groups to classes. As the system evolves and some groups settle down, the system could recognize groups that are rarely modified and suggest turning them into classes. The tools should guide this conversion to smooth the learning curve. The existence of classes in object-oriented programs can thus turn from a necessity to an indication of maturity, robustness, and quality. In terms of our metaphor of working with real objects, this would be akin to molding clay and, only when the final shape is settled, firing it to make it more structurally robust.

Groups as First-class Entities In our prototype implementation, there is no direct representation of a group. Instead, group membership is an attribute on member objects and available groups are derived from the available objects. These group attributes are serialized with their objects, but if a group is edited while one of its members is not available, that member will not receive the change. We want to investigate first-class groups as a (completely transparent) implementation approach. These would implement group-related meta-level function, and their members can relate to them and be updated through them. Given such first-class groups, it may also be easier to visually associate shared functions with their groups in Lively tools.

Similarity Based Groups In the current implementation, developers create groups explicitly. The mechanism we use is based on grouping as it is known from graphical editors, the user can select objects and declare them to be part of a group. Objects are either in a group or not, regardless of their properties. However, in large systems, objects can easily be forgotten, or similarities between objects might not be clear to the user. Thus, a mechanism could be added to help the user in finding "emergent" groups: grouping could happen automatically based on the similarities between objects. The interface of groups could also be based on distance, not on whether a property or function is shared (or not). Right now, any function that is not shared between all members of a group is not shown as part of the group editing process. However, if nine out of ten objects do share a method, it might be useful to show that method with some visual hint of that it is not ubiquitous. Such automatically detected and distance-based groups would allow developers to recognize emerging and diverging groups.

Additional Object-centric Tools Apart from the Object Editor, Lively offers more object-centric tools, e.g., an Object Inspector, a Style Editor, and a Text Editor, that all work on one object at a time. Developers would benefit if these can manipulate groups of objects as well. Furthermore, Lively's Parts Bin repository could be aware of groups. This would allow searching for groups and loading complete groups at once.

Advanced Query Tools for Groups In our current prototype, groups are just atomic strings — tags. These tags are used in a simple query mechanism based on equality, and a question arises about how useful a more elaborate query mechanism might be. Should we allow to select "all objects that are in groups that start with an 'a'?" How about "all objects that are in groups whose names are synonyms to 'player'?" While we, as experienced developers, might be tempted to allow such complex queries, the confusing group selections that can arise from such queries may prove to be unmanageable.

6. Related Work

Approaches to overlapping shared behavior range from abstract language concepts to tool support. Our approach primarily relates to approaches that allow injecting multiple distinct collections of methods into single objects. It further relates to module systems that emphasize concrete objects.

6.1 Stripetalk

The Stripetalk system [4], described as one of several theoretical systems that use different types of decomposition and language design principles, bears some similarity to our presented approach. Stripetalk is a shared-nothing system in that there is no mechanism to share behavior — if multiple objects need to be updated, they must be retrieved by means of some query, and the same edit must be applied to all of them. The authors proposed to attach one or multiple "stripes" to objects for easier retrieval, where stripes would simply be strings without any intrinsic meaning or structure. Stripetalk is very close to our system, but was never realized and evaluated in practice. The authors suggested, for example, very powerful retrieval methods (even a complete query language for stripes), but our practical experience with our approach so far indicates that such power may be detrimental to the maintainability of the system.

6.2 Multi-dimensional Separation of Concerns

Decomposition along multiple dimensions, as in Aspect- and Context-oriented Programming [5, 10], Traits [3], Mixins [3], and Multiple Inheritance [1] structure independent concerns

of an object independently. Developers compose objects from such inpedent decompositions. These modularize behavior and can be used by many objects similar to our groups. However, our groups are less abstract and enable direct feedback. In comparison to abstractions, the group approach does not separate concerns, but leaves the required information for execution and understanding in the object itself. Furthermore, traits require upfront planning or subsequent refactoring when implementing changes to a program, whereas object groups can be created on-demand.

6.3 Data, Context, and Interaction

Data, Context, Interaction (DCI) is a paradigm, in which objects can occur in different roles, depending on the execution context. Objects encapsulate only the domain knowledge, and shared behavior is added on-demand from roles. Similar to our approach, objects can have multiple roles at the same time. However, in DCI, object roles are added automatically at runtime, whereas groups are defined by the developer as they emerge.

6.4 Dynamic Text

Dynamic Text [2] is a tool-based approach that reduces the effects of duplication in scattered code. It tracks copies of code and allows developers to edit all instances of that code simultaneously. It is an alternative to Aspect-oriented Programming [10], but deliberately does not resolve tangling as concern implementations are sometimes more understandable in conjunction with base code. This approach relates to our object groups in that both do not resolve duplications, but rather provide tool support for managing duplications.

6.5 Prototype-based Inheritance

Prototype-based Inheritance [22] is an approach to shared behavior, in which objects inherit attributes and functions directly from other objects. Languages that offer prototypical inheritance—such as Self and JavaScript—allow dynamic replacement of an object's prototype, which is used in method lookup. They offer the concreteness of object-centric development. Compared to our groups, simple prototypical inheritance does not allow objects to share behavior across multiple prototypes.

7. Conclusion

We have presented Lively Groups as an extension to the Lively Kernel's object-centric development tools to group and work on multiple object simultaneously. Developers may group available objects that share behavior either visually, by clicking their graphical shapes or choosing them from a view of the Morphic scene-graph, or programatically, by evaluating program statements that return collections. The object-centric development tools present a group's shared functions and allow to specify behavior for all group members. Further, developers may evaluate statements on groups and, thereby, apply changes to all members simultaneously.

With our approach, tools manage duplicated code, while developers interact with live objects. That is, developers no longer choose between manual duplication of code to multiple objects, unnecessary indirection in form of delegation, or extended feedback cycles of traditional functional decomposition.

In the future, we want to implement our approach for more of Lively's tools. Further, the Object Editor should visually distinguish between object-specific and shared functions. Moreover, as our current implementations only considers available members while modifying groups, we want to explore first-class groups and automatic group discovery.

Nevertheless, our Object Editor already permits developers to work on groups of objects and, thereby, effectively scales the object-centric development approach from one to many objects.

References

[1] L. Cardelli. A semantics of multiple inheritance. *Semantics of data types*, pages 51–67, 1984.

[2] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto. Do We Really Need to Extend Syntax for Advanced Modularity? In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 95–106. ACM, March 2012.

[3] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An Approach to Multiple-inheritance Subclassing. In *Proceedings of the SIGOA Conference on Office Information Systems*, pages 1–9. ACM, June 1982.

[4] T. R. Green, A. Borning, T. O'Shea, M. Minoughan, and R. Smith. The stripetalk papers: Understandability as a language design issue in object-oriented programming systems. *Prototype-based Programming: Concepts, Languages and Applications*, 1998.

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

[6] D. Ingalls. The Lively Kernel: Just for Fun, Let's Take JavaScript Seriously. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 9:1–9:1. ACM, July 2008.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming Systems, Lan-*

guages and Applications, OOPSLA '97, pages 318–326. ACM, October 1997.

[8] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel–A Self-supporting System on a Web Page. In R. Hirschfeld and K. Rose, editors, *Self-Sustaining Systems*, pages 31–50. Springer, May 2008.

[9] A. Kay. Squeak Etoys, Children & Learning. Technical report, Viewpoints Research Institute, Jan. 2005.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '97, page 220–242. ACM, December 1997.

[11] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 314–321. IEEE, 1997.

[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. ACM, June 2012.

[13] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, 76(12): 1194–1209, December 2011.

[14] J. Lincke, R. Krahn, D. Ingalls, M. Roder, and R. Hirschfeld. The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content. In *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*, HICSS '12, pages 693–701. IEEE, January 2012.

[15] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28. ACM, December 1995.

[16] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, second edition, 1997.

[17] J. Micallef. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming*, 1(1):12–36, 1988.

[18] G. L. Steele Jr. An overview of common lisp. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107. ACM, 1982.

[19] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[20] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242. ACM, December 1987.

[21] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 9–1–9–50. ACM, June 2007.

[22] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp Symbolic Computing*, 4(3):223–242, July 1991.