Lively Groups: Shared Behavior in a World of Objects

Tim Felgentreff, Philipp Tessenow, Lauritz Thamsen

Hasso-Plattner-Institut
Jens Lincke, Robert Hirschfeld

Abstract. The Lively Kernel is a self-supporting, browser-based environment for explorative development of active Web content. In addition to object-oriented programming with modules and instances, Lively supports an object-centric approach for modifying visible objects directly. However, to share behavior between similar objects, Lively developers must choose to either abstract concrete objects into modules, to scatter code between objects, or to copy code to multiple objects. That is, they must choose between longer feedback loops, tight coupling, or reduced maintainability.
In this paper, we propose an extension to the object-centric development tools of Lively to work on multiple concrete objects. In our approach, developers may dynamically group live objects that share behavior and manipulate such groups as if they were single objects. Our enhancements scale Lively Kernel's explorative development approach from one to many objects, while preserving the maintainability of abstractions and the immediacy of concrete objects.

Keywords: Web Applications, Interactive Systems, Explorative 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. The Lively Kernel [4] is an interactive development environment for developing Web applications inside the Web browser. Lively's development tools allow programmers to change applications from within the same Web page and immediately see the results. Developers can either change the modules—as, for example, classes—or specific objects. The direct interaction with objects allows short feedback cycles [9] when working on the objects that make up the application.

Lively's object-centric tools only work on one object at a time. To implement behavior common to multiple objects, developers have three choices: either, abstract common functionality into modules that define these objects, scatter code across collaborating objects,or manually repeat code among objects. The first option reduces feedback immediacy, the second option breaks encapsulation [13], while the third reduces maintainability [7].

To overcome this challenge, we propose an extension of Lively's object-centric tools to work on groups as if they were single objects. Developers group objects by clicking their visual representations, by selecting nodes in the scene-graph, or by evaluating program queries. Developers may label such groups according to the role they share and edit all group members simultaneously. That is, programmers can evaluate statements on and add functions to all group members. Our approach manages shared behavior without dictacting the program decomposition, while maintaining the immediacy of live objects.

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 in 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 run-time, object-centric development tools, the implementation of the Morphic User Interface Construction Environment [11], 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 [8]. 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 [16], this directness and livelyness shortens the development cycle [15].

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 [10], 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.

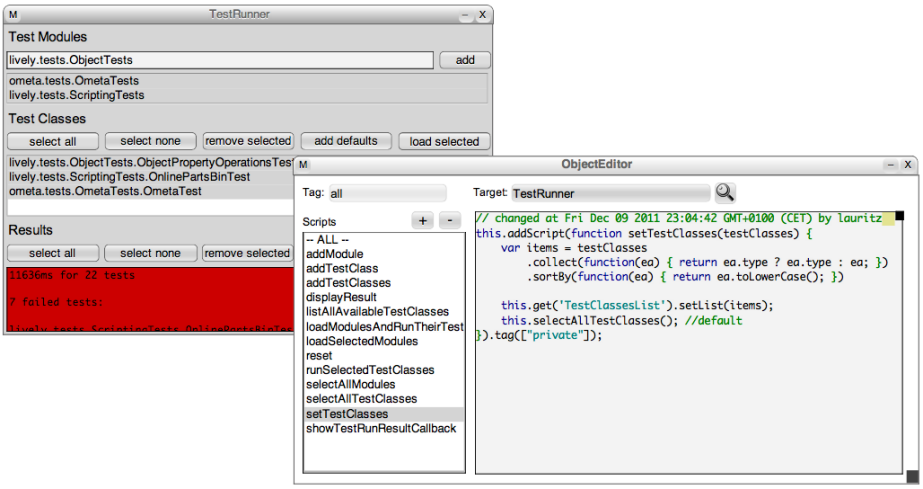

Fig. 1: Lively's Object Editor modifies a test runner built from Parts.

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.

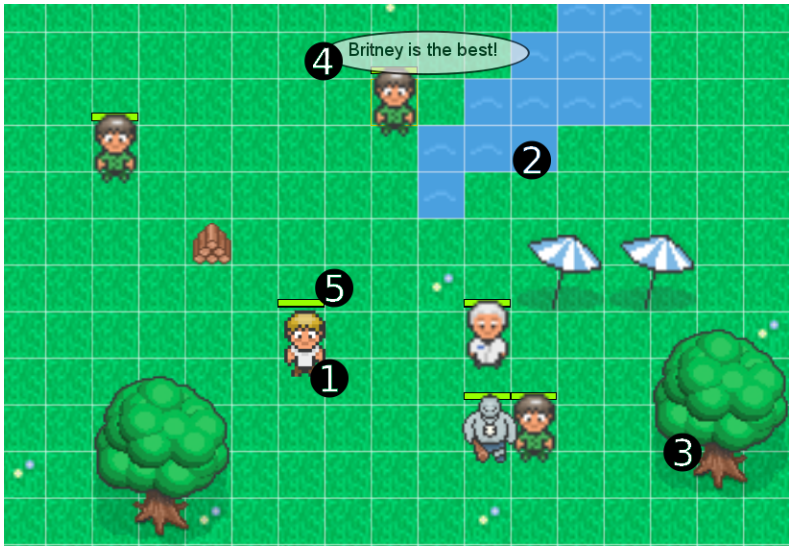

Fig. 2: *Freedom of Speech* — A debating adventure game built from Parts.

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 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 [5]. 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. 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.

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.

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 [12] 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.

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.

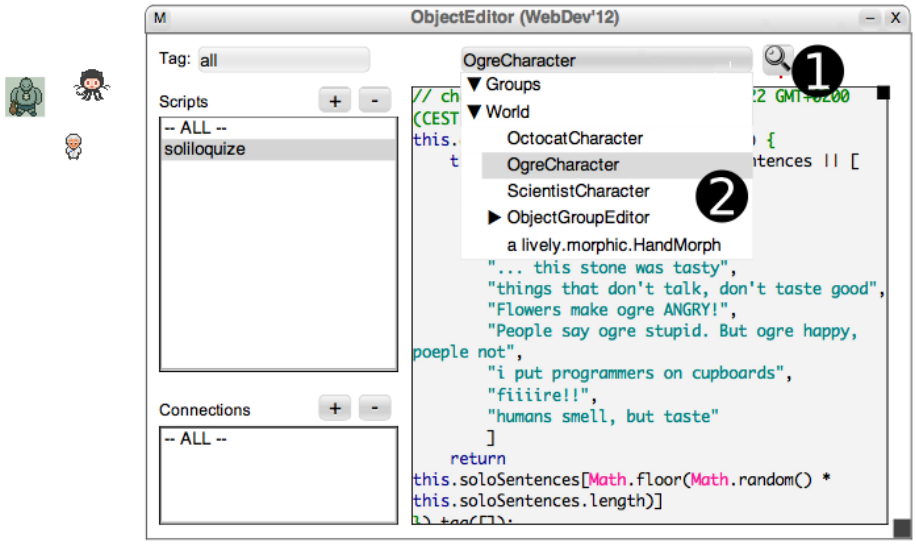

Fig. 3: Extended Object Editor includes a scene-graph browser and an object selection tool.

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.

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.

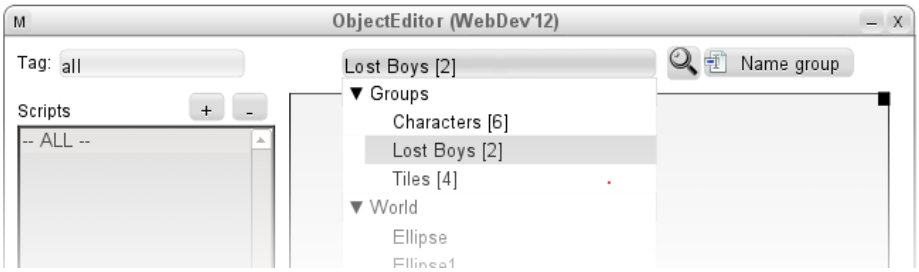

Fig. 4: Our Object Editor shows a list of available groups.

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 Future Work

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.

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.

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 that implement group functions directly, so their members can dispatch to them. Such group dispatch may only happen if an object has no object-specific implementation of a method. Given such first-class groups, it is possible to visually associate shared functions with their groups in Lively tools.

Similarity Based Groups In the current implementation, developers create groups explicitly. Alternatively, grouping could happen automatically based on the similarities between objects. Such automatically detected groups would allow developers to recognize emerging groups and to identify diverging groups.

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 Multi-dimensional Separation of Concerns

Decomposition along multiple dimensions, as in Aspect-oriented Programming [6], 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.

6.2 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.

6.3 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 [6], 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.

6.4 Prototype-based Inheritance

Prototype-based Inheritance [14] 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 propose to extend Lively's object-centric development tools to work on object groups. 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. Cardelli, L.: A semantics of multiple inheritance. Semantics of data types pp. 51–67 (1984)
2. Chiba, S., Horie, M., Kanazawa, K., Takeyama, F., Teramoto, Y.: Do We Really Need to Extend Syntax for Advanced Modularity? In: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development. pp. 95–106. AOSD '12, ACM (March 2012)
3. Curry, G., Baer, L., Lipkie, D., Lee, B.: Traits: An Approach to Multiple-inheritance Subclassing. In: Proceedings of the SIGOA Conference on Office Information Systems. pp. 1–9. ACM (June 1982)
4. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: Self-Sustaining Systems. chap. The Lively Kernel–A Self-supporting System on a Web Page, pp. 31–50. Springer (May 2008)
5. Kay, A.: Squeak Etoys, Children & Learning. Tech. rep. (Jan 2005)
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming. p. 220–242. ECOOP '97, ACM (December 1997)
7. Lague, B., Proulx, D., Mayrand, J., Merlo, E., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: Software Maintenance, 1997. Proceedings., International Conference on. pp. 314–321. IEEE (1997)
8. Lincke, J., Appeltauer, M., Steinert, B., Hirschfeld, R.: An Open Implementation for Context-oriented Layer Composition in ContextJS. Science of Computer Programming 76(12), 1194–1209 (December 2011)
9. Lincke, J., Hirschfeld, R.: Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming. In: Proceedings of the International Workshop on Context-Oriented Programming. pp. 2:1–2:6. COP '12, ACM (June 2012)
10. Lincke, J., Krahn, R., Ingalls, D., Roder, M., Hirschfeld, R.: 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. pp. 693–701. HICSS '12, IEEE (January 2012)
11. Maloney, J.H., Smith, R.B.: Directness and Liveness in the Morphic User Interface Construction Environment. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology. pp. 21–28. UIST '95, ACM (December 1995)
12. Meyer, B.: Object-oriented Software Construction. Prentice-Hall, second edn. (1997)
13. Micallef, J.: Encapsulation, reusability and extensibility in object-oriented programming languages. Journal of Object-Oriented Programming 1(1), 12–36 (1988)

14. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing Programs Without Classes. Lisp Symbolic Computing 4(3), 223–242 (July 1991)
15. Ungar, D., Smith, R.B.: Self: The Power of Simplicity. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. pp. 227–242. OOPSLA '87, ACM (December 1987)
16. Ungar, D., Smith, R.B.: Self. In: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages. pp. 9–1–9–50. HOPL III, ACM (June 2007)