Serializable Classes for Lively Kernel

Roland Lux and Willy Scheibel

Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam
Germany

Abstract. This paper proposes an extension of the Parts-based development process for the web-based IDE Lively Kernel, which is written in JavaScript. Parts are serialized object graphs consisting of Morphs and custom scripts for additional behavior. The current approach to add reusable behavior to different objects of a Part is to either copy it or create classes, which are versioned independently in system modules. Both of these approaches have undesirable characteristics which we aim to avoid. The proposed solution still uses classes to abstract common behavior but serializes them together with the Part instead of storing them in independent modules. In addition, an exemplary approach for appropriate tooling support to work with those classes is given.

Keywords: JavaScript, Lively Kernel, Locally Scoped Classes, Serializable Classes

1 Introduction

As part of the seminar "Web-based Software Development", we explored the development process of a web based development environment by implementing a small application in order to find potential for improvement. The environment we chose was Lively Kernel [1], which is written in JavaScript [2].

The application was a simple Tower Defense game [3] and started out as a Part, which is the common approach in Lively. Unlike traditional applications with a start up routine, Parts are programmed simply by adding state to an existing object graph, which then can be versioned in the PartsBin [4] by way of serialization and deserialization respectively to restore the state at a later time. The object graph's root is a morph, a graphical component from Morphic [5], usually with submorphs, which means that all Parts are morphs. Behavior is added by writing custom scripts, that is functions which are serialized as well.

While implementing the Tower Defense game, we soon encountered limitations of the Parts system. The scripts are useful for adding behavior to single objects like a callback for a button, but do not provide a good way to add common behavior to many objects of the same kind. That is done by writing classes. However, classes are versioned independently in modules saved in JavaScript files, which means they do not actually belong to the Part. Developing the Part and the classes independently can cause inconsistencies, for example a serialized Part does not work anymore after loading it, because the classes in the system have changed. Another approach would be to only simulate classes by copying behavior to new objects. This requires a prototype object to be stored within the Part. However, the tool support for that is missing. You can only add scripts with the *Object Editor* and only to morphs of the Part. Visible prototype morphs would be esthetically unpleasing, while non-morph objects are not even accessible through the editor. Invisible morphs would work, but it feels like a hack to use an invisible graphical component as just a code container.

We identified these problems and propose a solution which still allows common behavior to be abstracted in classes while keeping the versioning of Parts consistent, as well as tool support to access these classes through the *Object Editor*.

2 Concept

This section covers the different approaches to implement common behavior on Parts and introduces an alternative, which has some advantages over the previous ones.

The current approaches to implement abstract or common behavior on Parts are to either copy it to the other objects which use it or to use a class which is versioned in a system module. The disadvantage of the former is that the behavior is copied multiple times, which makes it hard to change, whereas the latter has the disadvantage of being versioned independently from the Part which can lead to inconsistencies.

A further approach is to create prototypes, keep them somewhere in the object graph of the Part and place them in the prototype chain of the objects with the common behavior after the Part is loaded. This approach seems eligible but it has disadvantages, too. First, only morphs have hooks to handle additional behavior while saving and restoring and second, it is impracticable to manipulate the prototype chain manually every time again.

But having a prototype is not far off from using classes in JavaScript [2, Chapter 9: Classes, Constructors, and Prototypes]. JavaScript classes use a prototype which is placed in the prototype chain of a newly created instance. This means, with the use of classes the prototype chain does not have to be manipulated manually and a prototype object with the common behavior can still be used. Instead of versioning classes in system modules these classes could be placed in the Part object graph and saved and restored together with the Part. Since multiple versions of a Part can exist in a world at the same time, the classes also have to be able exist in multiple versions at the same time in the system. This means those classes cannot not be referred to by a global name in a global namespace, so instead of using a global namespace, a namespace distinctive to the Part should be made available. Because of this characteristic, we refer to this kind of classes as *private classes*.

Serialized morphs are saved and restored with the website or as Parts in the PartsBin. A morph is serialized as JSON [6] which is then sent to the server. That requires the classes to be serializable to JSON as well.

To sum up, for our approach serializable classes without a global name are needed to implement reusable behavior on morphs. This means the concept and implementation of system classes can be adapted with only a minor change, namely that when a class is created it must not register itself in a global namespace. Instead, the morph can register the class on itself. Furthermore, the class must be serialized and deserialized together with the morph it is registered on.

An overview of the features of private classes is shown in Figure 1. A morph can have attached classes - that is private classes which are registered on the morph. Any object of the object graph of this morph can be an instance of a private class, e.g. in this image one of the submorphs of the main morph is an instance of `PrivateClass`. Inheritance among private classes is also supported. As a conclusion, even the root morph itself can be an instance of a private class which is attached to it.

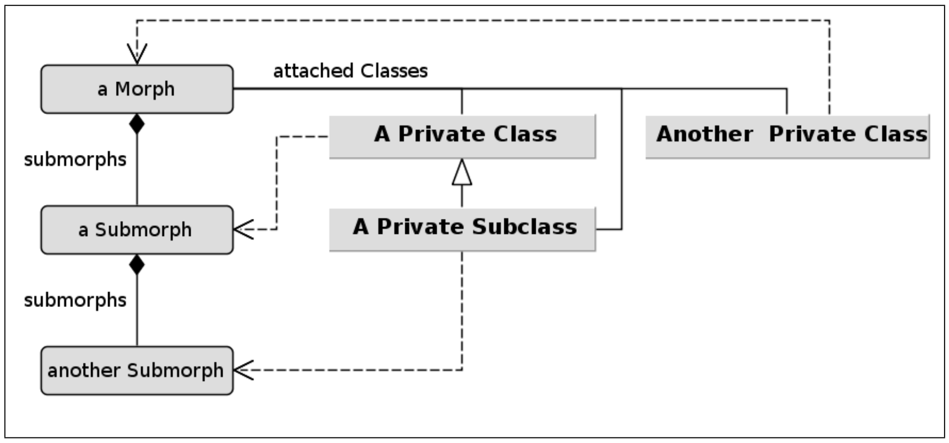


Fig. 1: Features of private classes

3 Implementation

This section contains the details of how the classes belonging to a morph are implemented. There are basically three different steps when storing a class on a morph. First the class has to be created without a global name, then it has to be registered in the local namespace of the morph, and finally the class has to be serialized with the morph.

3.1 Class Creation

Lively Kernel already comes with a sophisticated class system for Javascript, which is based on Prototype [7]. Normally a new class is created with the `subclass` method on `Function.prototype`. However, it automatically registers the newly created class in the global namespace. So it was necessary to prevent that. We extended `Function.prototype` with a method called `privateSubclass` which takes the same arguments and basically has the same behavior, but without registering the class in the global namespace. Unlike `subclass` it also does not check if a class with that name already exists. This enables us to create arbitrary classes which are only accessible through the return value of that method. The class itself knows its name, but there cannot be any conflicts, as the global namespace is not used. The slot `isPrivateClass` is set to true, so that it can be identified.

3.2 Class Registration

Once a non-global class is created, it needs to be stored somewhere other than the global namespace in order to be accessible. This is done by simply putting the class in a slot on the morph which is named after the class. That way, a class with the name `TestClass` can be accessed with `this.TestClass`, where `this` is the morph. Consequently, the class names are bound only to the morph; other morphs or different versions of the same morph from a Part can still use the same class names for different classes. With this approach it is easily possible to instantiate these private classes from scripts on the morph, but it does not allow the classes to instantiate another class which is also stored on the morph. We addressed this problem by binding the morph, which functions as the local namespace, to the `namespace` slot on the class's prototype. To access another class from within a private class, the code looks like `this.namespace.OtherClass`, where `this` is an instance of a private class.

For creating and registering a private class on a morph, we implemented a convenience method `declarePrivateSubclass` on `Morph`. The method creates the class, registers it on the morph's slot named after the class, and sets the `namespace` slot on the class to point back to the morph.

3.3 Class Serialization

Morphs in Lively Kernel are stored by serializing its current object state. They are saved either in the current world or as a Part in the PartsBin. Serialization works with the `ObjectGraphLinearizer`, which can be extended with plugins. Its usual behavior is to ignore classes and not serialize them. Instead, instances only store a reference to the name of their system class, which can then be restored after deserialization. That approach would not do for our private classes, because they are designed on purpose to not be among the system classes.

To serialize the classes, a plugin for the `ObjectGraphLinearizer` needed to be written. The linearizer works by using an identity map for all the objects in the object graph, and uses the id's instead of references. This makes it possible to cope with circular references.

The plugin we wrote handles two types of objects: private classes and instances thereof. In case of private classes, we add an object which contains all the members of the class's prototype under the slot `__privateClass__`. Members which are objects are given back to the `ObjectGraphLinearizer` to be treated normally. Methods, that is members which are functions, are converted to a serialized version. The serialized version is an object with a slot `__privateClassMethod__` containing the source code string of the function. Attributes on the constructor object are not serialized yet, but that behavior could be added with a small addition to the code. The superclass can be either a system class or another private class and is stored in `__privateClassSuperclass__`. System classes are only referenced by their name, but if the superclass is another private class, the actual class is referenced.

Instances of the classes are simply extended with a slot `__privateClassRef__` with a reference to the private class. That means, it would be possible to drag an instance of a private class onto another morph, where it would keep its current behavior, even though the particular version of the class is not registered on the other morph. That also means that the first morph is still referenced by the `namespace` slot and thus would be introduced into the object graph of the other morph, meaning it would be serialized as well, if the other morph is serialized. That is the price for the object keeping its behavior and access to the others classes in its namespace. This may or may not be the desired behavior.

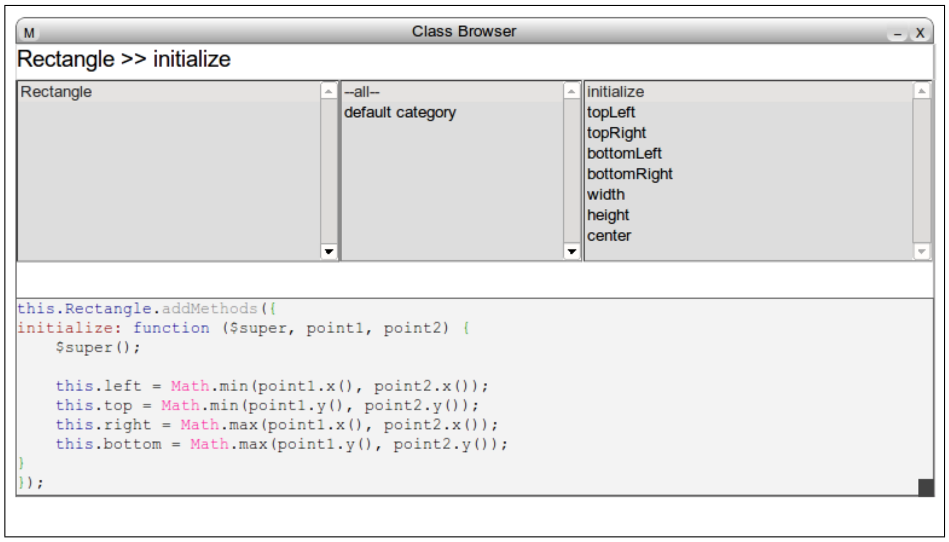
Another approach would be to only store the class name, instead of a real reference. Then the instance would be restored to the class with the same name on the morph it was dropped on. However, this would only be done after a subsequent serialization and deserialization. At runtime, it would still be bound to its original class, even after having been dropped on another morph. This kind of inconsistency was the reason we decided against this approach, though it might be valid in some scenarios.

4 Tool Support

This section covers the planned and currently implemented tool support for private classes in Lively Kernel. At first, a simple prototypical class browser is presented; then a more sophisticated design for a tool for working with private classes is introduced, which is based on the current *Object Editor*.

The simple class browser in Figure 2 offers support for creating private classes, changing their superclasses, adding methods and modifying them. It can be opened for a morph with the command `morph.openClassesInBrowser()`. The browser then shows all classes attached to the morph in the left pane. Protocols are listed in the middle pane. The methods of a selected class and protocol are listed in the right pane. To manipulate a class or a method, the lower code pane can be used to edit and save the code, using *Ctrl+S* for saving and thus executing all the code in the pane or *Ctrl+D* for executing the selected code directly. The browser aims to behave in a similar way as the existing *System Class Browser*, which operates on JavaScript module files instead of objects.

Since this simple class browser was only written as a prototype and not meant to be used productively to manipulate private classes, it lacks some very basic operations. For example, this browser does not support deleting private classes or methods, although this can be achieved by manually executing the according code. Likewise, there are no sophisticated features accessible through menus, toolbars or context menus.



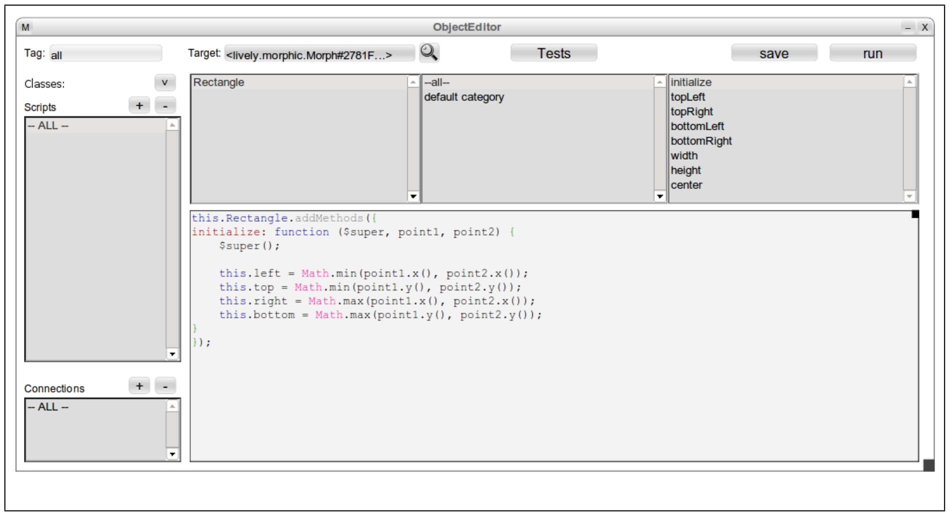
```
this.Rectangle.addMethods({
initialize: function ($super, point1, point2) {
    $super();

    this.left = Math.min(point1.x(), point2.x());
    this.top = Math.min(point1.y(), point2.y());
    this.right = Math.max(point1.x(), point2.x());
    this.bottom = Math.max(point1.y(), point2.y());
}
});
```

Fig. 2: A simple class browser to manipulate private classes

To develop GUI applications in Lively Kernel the main tool to use is the *Object Editor*. The editor allows the programmer to edit morphs in the current world by manipulating the attached scripts and attribute connections. In the normal development process only a few other tools – e.g. the *Style Editor* and the *PartsBin Browser* – are needed to create an application. Therefore, we propose an extension to the *Object Editor* to handle private classes as well.

Figure 3 shows how the integration of the class browser into the *Object Editor* could look like. Since not every application needs private classes, the panes for classes, protocols and methods should be collapsible. A button on the left side of the editor indicates that those panes can be expanded to reveal the classes that are attached to the current morph. The current script editing text pane serves as class and method editing pane as well. Its scope must be set when the user clicks on either a class, a method or an added script. The three panes for classes, protocols and methods should also provide context menus to provide additional behavior which cannot be supported otherwise with only these small changes to the user interface. Lastly, the panes need to be updated if a different morph is selected in the editor.



```
this.Rectangle.addMethods({
initialize: function ($super, point1, point2) {
    $super();

    this.left = Math.min(point1.x(), point2.x());
    this.top = Math.min(point1.y(), point2.y());
    this.right = Math.max(point1.x(), point2.x());
    this.bottom = Math.max(point1.y(), point2.y());
}
});
```

Fig. 3: Design Proposal: Object Editor Integration for Private Classes

5 Future Work

Although the serializable classes for morphs are implemented and operational, there are a few areas which are in need of improvement.

The first shortcoming is that only members of the prototype are serialized, class side attributes are not supported yet. It was not necessary for our game implementation, but we are sure that this feature is useful and should be added to the serialization plugin.

Another area with room for improvement is the syntax of how private classes currently access other private classes with `this.namespace`. Perhaps it is possible to bind that object to a variable like `$namespace` scoped to instances of the class. A completely different syntax might be deemed more suitable as well. This is only syntactic sugar though, and does not add more functionality.

As it is now, all private classes are stored directly on the morph, and thus have no subnamespaces. This can however be done manually by adding objects to the morph which serve as such subnamespaces. No convenience methods for this exist yet, so that is another feature to be added.

Last, the tool support for editing the private classes is not finished yet. We do have a prototypical class browser with a limited set of features, but the goal is to integrate the browser into the *Object Editor* as described in section 4.

6 Related Work

In this section, we compare our private classes to similar projects, with emphasis on the main aspects of our implementation, namely the access of classes from a limited scope and the serialization of object graphs including classes.

One project which is similar to the way Lively Kernel serializes object graphs is the STON format [8]. STON's approach to linearize arbitrary (and possibly circular) object graphs with the help of identity maps is very similar to the way Lively Kernel's `ObjectGraphLinearizer` works. However STON cannot serialize classes yet because they do not support serialization of code. In JavaScript we did not even have a problem with serializing functions, as we can simply convert it to its source code string.

Another similarity can be found in the local classes in C++ [9] and Java [10, Chapter 3.11 : Local Classes]. In C++, there can be classes which are defined inside of functions or anonymous namespaces. In Java, local classes can only be defined inside of functions. Such classes cannot be accessed from outside their scope.

Finally, parts of Newspeak [11] can be compared to our implementation, since Newspeak has no global namespace at all and all classes need to be accessed through slots. In fact, our implementation was strongly inspired by the Newspeak class access.

7 Conclusion

During the course of the seminar we have found out that using custom classes in conjunction with Parts poses a problem insofar as both are independently versioned and hence are prone to inconsistencies. The proposed solution consists of serializable classes which can be versioned together with a Part as well as tooling support in the form of integration into the existing *Object Editor*.

We implemented a prototypical class browser for the serializable classes, however witout integration into the *Object Editor* as of yet. The class browser allows to create and modify classes which are stored on a morph.

The class serialization is functioning as a plugin to the existing Lively Kernel Serializer. Class side attributes are not serialized yet, but is possible as future work. The class prototype and all its members, including functions, are fully serializable and thus allow morphs to store classes in their slots.

We extended the class `Morph` with convenience methods to create and access the serializable classes. The classes provide an easy way to access other classes of the same morph via the `namespace` slot. The serialization works in such a way that it is possible for the morph to have submorphs which are instances of the classes stored within the morph, and it is even possible for the morph itself to be an instance of its own classes. If the latter may ever be useful though is anyone's guess.

As a consequence of having the classes in slots, they are locally scoped and without a global name. That means, that multiple versions of a morph in the same world do also have different versions of the classes, which was impossible beforehand.

In conclusion, we successfully extended the workflow with Parts in Lively Kernel to include Part dependent classes without any restrictions to the previous set of features of Parts.

References

1. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The lively kernel a self-supporting system on a web page. In Hirschfeld, R., Rose, K., eds.: Self-Sustaining Systems. Volume 5146 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 31–50
2. Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly Media (2008)
3. Wikipedia: Tower Defense (`http://en.wikipedia.org/wiki/Tower_Defense`) (2012)
4. Lincke, J., Krahn, R., Ingalls, D., Röder, M., Hirschfeld, R.: The lively partsbin: A cloud-based repository for collaborative development of active web content. In: Collaboration Systems and Technology Track at the Hawaii International Conference on System Sciences. (2012)
5. Maloney, J.: Morphic: The Self User Interface Framework. Technical report (1995)
6. D. Crockford: RFC 4627 (`http://tools.ietf.org/html/rfc4627`) (2006)
7. Prototype Core Team: Prototype (`http://www.prototypejs.org/`) (2012)
8. Sven Van Caekenberghe: Smalltalk Object Notation (`https://github.com/svenvc/ston/blob/master/ston-paper.md`) (2012)
9. IBM Coorperation: Local Classes (`http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/topic/com.ibm.xlcpp8a.doc/language/ref/cplr062.htm`) (2004)
10. Flanagan, D.: Java In A Nutshell, 5th Edition. O'Reilly Media (2005)
11. Bracha, G., Ahe, P., Bykov, V., Miranda, E., Kashai, Y.: The Newspeak Programming Platform. Technical report (2008)