A Module System for a General-Purpose Blocks Language

Yoshiki Ohshima
Yoshiki.Ohshima@acm.org
CDG

Jens Mönig
jens@moenig.org
Communications Design Group (CDG), SAP Labs

John Maloney
jmaloney@media.mit.edu
CDG

Abstract

Our team is developing GP, a new blocks language that aims to be beginner-friendly, like Scratch, yet capable of scaling up to support larger applications. We hope to allow a world-wide community of users to share projects, sprites, and code libraries and to create new ones using a mash-up development style.

To support such easy sharing and reuse, GP incorporates a strong notion of *modularity*. Modularity allows components created by different users at different times to interoperate without worrying about conflicts. However, we do not want modularity to add to the burden of the beginning programmer; we'd like GP's module features to stay out of the way until they are needed.

In this paper, we present the key ideas around the GP module system. A module in GP is a unit of encapsulated code and data. It exports a selected set of classes, functions, and variables, provides a namespace for the code inside, and may include private helper classes, functions, and variables. A module can also extend system classes with additional methods. Such extensions can be used freely within the module but are not visible outside it. Modules can be saved and re-loaded, allowing modules to store user-created projects, sprites, and code libraries.

I. Introduction

Our team is working on a new block-based language we are tentatively calling GP (Figure 1). In GP, as in Scratch, a beginner can start programming without having to learn non-essentials (i.e, it has a "low floor"). At the same time, as GP programmers learn and gain experience, they can create sophisticated applications or even extend the system itself ("high ceiling"). In other words, we hope to create a blocks programming language that is as beginner-friendly as Scratch while being as "general purpose" as languages like Python or Java.

There are three GP design goals that are especially relevant to the module system:

- **Blocks-Based:** We created a new language specifically designed for visual blocks. The user can write code in either blocks and text and can view, edit, and debug GP code as blocks, including GP library code and the GP programming environment itself.

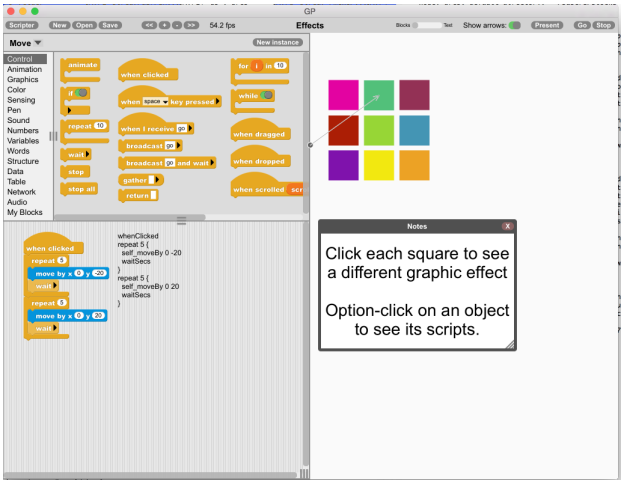

Fig. 1. A Screenshot of GP. On the left, the authoring tool is shown with a block palette and a scripting area, which is showing a stack of code in blocks and in textual code. The project being edited has some colored rectangles and an explanation in a text box.

- **Self-Sufficient:** To maximize the potential for learning about and extending GP, most of the core library and the entire programming environment is implemented in GP itself; only the GP virtual machine and low-level primitives for things such as graphics and file system operations are written in C. While not all users will want to explore GP's implementation, those of us who have used Smalltalk or Lisp systems know how much one can learn by "lifting the hood" and studying the workings of mathematical operations, collections, UI frameworks, and the programming environment.
- **Simple:** To encourage budding GP programmers as they explore and extend the GP system, GP has a carefully selected set of features. A conscious effort has been made to eliminate language features that could be obstacles for learners, while still retaining enough expressive power to allow GP to be used to build the GP system itself. For example, while GP is a class-based object-oriented language, it does not have an inheritance mechanism. GP system code uses explicit delegation, which we hope will be easier for learners to read and understand.

In short, GP is a class-based object oriented language

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

(somewhat influenced by Smalltalk) without inheritance and with a syntax that is easily represented as visual, puzzle-piece-shaped blocks.

A key idea around this new language is that a future programmer may want to make her applications in the *mash-up* style; that is, she would take sprites, projects, libraries and other kind of components created by her fellow programmers, and make her own by assembling them.

Imagine how cool it would be if the authoring tool itself were the product of this process, with modules contributed by various users! Also, a modular design, where the authoring tool itself can be controlled programmatically, would allow users to create tutorials or present a collection of interacting components written by different users.

Existing blocks languages offer various extension mechanisms. Snap supports libraries of user-defined blocks [1]. Android App Inventor's components add new blocks appear to the palette when dropped into the application being constructed, but as far as the authors can tell does not allow users to create and share new components [2]. The Scratch Extension Mechanism[3] comes closer to our vision. However, Scratch extensions cannot extend the Scratch editor itself and must be written in an entirely different language (JavaScript), not Scratch itself.

To facilitate this mode of collaboration, we need a strong assurance that someone else's component (or even your own previously created component) does not break any other components in the application. In other words, we would like to have good *modularity* in the language.

We looked at some ideas of existing module systems, most notably from the Newspeak [4] language and the Node.js Package Manager (npm) [5], and designed a module system for GP. A unique challenge for GP is to provide a module system that works, but at the same time does not get in the way of users, especially beginners.

The remainder of this paper focuses on the design and implementation of GP module system. In Section II, we briefly describe the GP language. In Section III, we explain the design of the module system design and show some examples.

II. GP in a Nutshell

GP is a class-based object-oriented language. Everything in the system is an object and has a class, which defines its instances' behavior. The system contains some classes that are considered to be system-defined, such as `Array` and `Morph`, and `Turtle`.

As the system-defined classes are also written in the same language, their definitions can serve as examples of the GP language:

```
defineClass Turtle morph x y direction

method forward Turtle n {
  x += (n * (cos direction))
  y += (n * (sin direction))
  ...}
```

In this code, a class called `Turtle` is defined with four instance variables (`morph`, `x`, `y` and `direction`). Then a method called `forward` is defined for the `Turtle` class. The method takes an argument and mutates some of the instance variables.

The following code creates an instance of `Turtle` and calls the method on it:

```
aTurtle = (new 'Turtle')
forward aTurtle 10
```

As you can see, the method name (`forward` comes) first, followed by the arguments. The method look up is single-dispatch; the method dictionary of the class of the first argument is searched to find the actual implementation of the method and invoked.

Besides methods, GP has functions that are not bound to a particular class. For example, you can define a function (that does not belong to any class) called `myAdd` and call the function with arguments 3 and 4:

```
to myAdd a b {
  return (a + b)
}
myAdd 3 4
```

There is no built-in inheritance mechanism between classes, thus there is no class hierarchy. However, it is desirable to be able to have one implementation of functionality that is polymorphic to the instances of different classes. The functions are used to support this. Namely, when a method look up fails, the system looks up the function that has the same name, regardless of the first argument's class.

As you can see, calling a method and calling a function have the same syntax: the function name, or the selector, comes first, followed by arguments. This syntax makes calling a method and calling a function polymorphic and allows the unbound functions to work as fallback code.

To have better extensibility, we feel that the system-defined classes should be *open*. That is, it should be possible for an end-user to add a method to a system class such as `Array` and extend its vocabulary We will return to this point in Section III-E.

Classes and functions are first class objects as well. For example, when the user creates a sprite in the authoring tool, the authoring tool creates a class for it programatically, and assigns a name such as `MyClass1`.

But here is the fundamental problem. If class names in user extensions were globally visible then one user's class names could conflict with those of another user when both users' modules are loaded into the same project. Likewise, without a good module system, one user's extensions to system-defined classes might conflict with another user's extensions or change the behavior of core classes in unexpected ways.

The module system proposed in the next section addresses this problem by providing a well-defined mechanism for namespace isolation between modules.

III. The Module System

A module in GP is a unit of encapsulated code and data. A module may have classes and functions inside, and their names are unique within the module and don't collide with classes and functions in other modules. Classes and functions that are for use by code outside the module can be *exported*, while the rest can be hidden.

The term "module" has been used in different ways in different systems. Compared to other module systems, GP's module system has the following characteristics:

- **Instance-based:** At runtime, a module is represented as an object (an instance of the `Module` class). It differs from being a compile-time name resolution mechanism, such as that of the Modula family of languages [6]. It is more similar to a module in Node.js package manager (npm), but with a distinct difference: loading a GP module multiple times creates a fresh instance for each time, whereas the npm returns the same cached instance. See below for the rationale for this distinction.
- **Local classes and functions are objects:** As GP represents classes and functions as objects (instances of `Class` and `Function` class), a runtime module instance stores the classes and functions local to it in its data structures.
- **Supports meta-operations:** The module system does not aim to make a module be a total black box. While local classes and functions cannot be used directly from the outside in normal circumstances, we feel it essential to be able to write introspection tool in GP itself that analyze and manipulate modules.
- **Objects can escape:** Instances of local classes are allowed to escape to the outside of the module. For example, an instance of a private helper class can be returned from a method call.
- **No nesting:** At the moment, we are not considering supporting statically nested module definitions. A module can use other modules by instantiating them dynamically, but there is no static hierarchy of modules.
- **A module is an object but an object is not a module:** This design is not pursuing a unification between objects and modules. The description of module as a unit of encapsulated code and data could also apply to an object, and some other languages, most notably Newspeak, unify these concepts.

A. A Module Example

A module definition for a paint editor is shown here to explain the proposed module system. Following the npm tradition, a module object behaves as a dictionary in which publicly visible items are stored. But it also has fixed instance variables to hold classes and functions and "Expanders" (see Section III-E) defined in this module. These variables are called `classes`, `variables`, `functions` and `expanders`.

```
module 'PaintEditor'
moduleVariables DefaultPenColor CurrentColor Palette
moduleExports Editor
moduleExports changePenColor

// the initializer of the module
to initializeModule {
  DefaultPenColor = (color 0 0 255) // blue
  CurrentColor = DefaultPenColor
  Palette = (readPNG 'palette.png')
}
// classes and functions
defineClass Editor a b c
method handMove Editor x y {...}
defineClass Pen a b c
method drawCircle Pen x y r {...}
to changePenColor newColor {
  tmpVar = CurrentColor
  ...
  CurrentColor = newColor
}
```

The `module` command indicates the start of a module definition, and the subsequent code up to the end of the compilation unit (or file) defines the content of the module.

When a module is loaded, an instance of `Module` class is created, and its instance variables (`classes`, `variables`, `functions`, and `expanders`) are filled with the items declared in the module definition. In this example, classes called `Editor` and `Pen` are defined, and a function called `changePenColor` is also defined in this module. Note that even a common name such as Editor can be used without fear of name clashes, because the module provides an isolated namespace.

The `moduleVariables` command declares module variables. In this example, `DefaultPenColor`, `CurrentColor`, and `Palette` are declared. These variables are visible from classes and functions defined in the module.

The `moduleExports` command specifies the items to be exported. The arguments are the names of classes, functions, or variables that are to be exported. Recall that the module object itself behaves as a dictionary; this dictionary is populated with items that are visible to the client of this module. In this example, the `Editor` class and the `changePenColor` function are exported, while the `Pen` class is not.

The module initializer may take arguments to initialize module variables:

```
module 'AnotherEditor'
moduleVariables MyPen

to initializeModule aPenModule {
  MyPen = (new (at aPenModule 'Pen'))
}
...
```

In this example, `aPenModule` is a module that exports a class named `Pen`, and the `MyPen` module variable is initialized with an instance of `Pen`.

Meta-language features to introspect classes and functions, such as `class` and `functionNamed`, accommodate the notion of modules. For example, there is a fucntion called `class` that takes a string as its argument and returns a class

object. It first searches the list of classes in the module where the call occurs. If the name does not appear in the module's list of classes, then the search is delegated to the top-level module (explained in section III-B). This allows all modules to use core classes such as Array.

With this name resolution rule in place, all code in a module can be written with "short names"; in other words, the creator of a module need not be concerned with how the module will be referred to and used.

Loading a module creates a new instance of the `Module` class and fresh copies of classes and functions that belong to the module. This is important as a user may want to create two copies of the same module and then modify one of them. Imagine that a user wants to improve the paint editor, using the existing paint editor to create and edit button icons. In that case, being able to have two independent copies is very helpful.

In npm, a `require()` function call returns the same cached instance when called with the same argument multiple times. This allows modules that have circular dependencies to load, and allows sharing. However, by passing in module parameters at module creation time (as arguments for the `loadModule` function call), or by setting module variables afterwards, we can build any network of modules. In other words, creating a fresh instance by default seem to be helpful for common use cases.

With this code snippet:

```
painterModule1 = (loadModule 'PaintEditor.gpm')
painterModule2 = (loadModule 'PaintEditor.gpm')
painter1  = (new (at painterModule1 'Editor'))
painter2  = (new (at painterModule2 'Editor'))
```

the variables `painterModule1` and `painterModule2` stores instances of Module initialized with the definition in `PaintEditor.gpm`, and `painter1` and `painter2` each have an instance of `Editor`, thus they can have distinct values for `CurrentColor`.

The following scenario illustrates how a project would evolve. Imagine that making improvements to the (current version of) paint editor requires to change the palette object as well as code. In that case, the authoring environment would load two instances of the paint editor module. One of these would be used as a stable working version to edit the new palette. The other would be used as the development version. The authoring environment is able to modify the development version on the fly so that the programmer could experience the edit-and-continue style work flow. Even if some change to the development version causes it to break, the stable version will be intact.

Note that the module instances don't need to be stored in variables. The above code could simply be written as:

```
painter1 = (new (at
                 (loadModule 'PaintEditor.gpm')
                 'Editor'))
painter2 = (new (at
                 (loadModule 'PaintEditor.gpm')
                 'Editor'))
```

On the other hand, sometimes it is useful to keep a reference to the module in a variable. For example, one could call a function from painterModule1 in the following manner:

```
// get the function
changeColor = (at painterModule1 'changePenColor')
// call it
call changeColor (blue)
```

or, more succinctly:

```
call (at painterModule1 'changePenColor') (blue)
```

B. The Top Level Module

GP comes with a set of core classes the define basic data types (Integer, String), collections (List, Dictionary), the Morphic UI framework, and the programming environment itself. These classes are stored in a module, called the top-level module, that is visible to the code in all other modules. Similar to the method look up rule, the look up rule for a class or function first checks the module where the look up starts. Then, if it does not find an entry there, it looks for it in the top-level module.

C. Sprites and Projects as Modules

The simple formulation of the module system allows us to represent user projects as modules. When a user enters the visual authoring environment and starts making a project, a module instance is created to store the work in the project.

A project is a set of sprites, each represented as a class, along with additional code (such as functions) and data. It is therefore a natural match for being represented as a module. The user may interactively add more sprites, variables, and classes. When a project is saved, these user classes and the serialized sprite data are stored into a module definition. As sprite properties can be modified by direct manipulation in the authoring environment, code alone may not be able to recreate the resulting state. So, keeping the Sprites' state as serialized data is necessary.

An open question is that how far such a user can go without needing to know about the module system. For for simple projects, the authoring tool can build module "behind the scenes", creating classes, variables, and functions inside that module as the user works and saving the entire project as a module. A more advanced user can import and use modules designed to extend GP by making new blocks available in the palette without needing to understand the module mechanism in detail.

Ideally, only when a user wants to create GP extensions that add features to the programming environment or export libraries of blocks meant to be used by other users would would they need to learn how modules work in more detail.

D. A Module Using Another Module

The fact that modules are just objects allows great flexibility in how one module uses others. Let us say the `readPNG` function used in the `PaintEditor` example was actually defined in a separate module called `PNGReader`. The line in the initializer section for the `Palette` image might load it like this:

```
Palette
  = (call (at (loadModule 'PNGReader.gpm')
              'readPNG') 'palette.png')
```

Once the job of reading the PNG file (`palette.png`) is done, the instance of `PNGReader` module is garbage collected without leaving a trace.

In other cases, an exported class or a function may be stored into a variable on the client side:

```
myReader = (at (loadModule 'PNGReader.gpm')
               'readPNG')
Palette = (call myReader 'palette.png')
```

In this example, a function from the `PNGReader.gpm` module is stored into the variable `myReader`, but the module itself does not get stored into any variable. Because the function has a pointer to the module object (as described in III-F), the module instance will be kept in memory as long as it is referenced from such a class or a function.

E. Expanders

As mentioned earlier, the core classes in the top-level module are visible in all modules. If a module defines a class with the same name as one of the core classes, then that definition will replace the core class within that module. However, there are times when a module may simply want to add a few additional methods to a system class.

The desired behavior is that a method added to a top-level class by one module should not effect any other modules, even when another module adds a method with the same name to the same class. In other words, we would like each module to be its own isolated universe with it's own private extensions to the core classes.

There have been research efforts in the past on this problem, including Expanders [7] and ClassBoxes [8]. We've adapted the Expander idea to GP.

The original Expander design (for Java) added a new language construct to specify the scope of expanders. However,the GP module mechanism already supports isolated code units, so by using a module as the scope for Expanders, no new construct is necessary. Any method definition for a class that is not defined in the module is only visible from within that module. In other words, such a method definition automatically becomes an expander.

For example, if you have the following definition in a module:

```
method f1 Array x y {..}
```

and a definition of `Array` does not exist in the module, `f1` becomes an expander. Code defined in the module can call this method on any instance of `Array`, but it does not change the behavior of Array for the code in other modules.

F. Implementation of Modules

In the current implementation, each function or method holds a pointer to the module it belongs to. When a call happens, the selector is looked up in the following order, and the first implementation found is invoked:

1) The expander data structure for the receiver's class in the module of the method where the call occurred (this is statically determined.)
2) The receiver's original class.
3) The `functions` of the module where the call occurred (this is statically determined.)
4) The `functions` of the top-level module.

As you can see, the look up rule is quite simple. The only complication over a language such as Smalltalk is that the method lookup needs to check the expanders for the module. Also, instead of looking up the superclass chain for an implementation, GP looks up the functions.

As the lookup result can be cached, the extra step of looking up in the expander data structure does not incur a significant performance penalty.

We are aware that the expander concept and unbound function could be unified. A function defined in a module could be thought as an expander for the class of `nil`, and the lookup mechanism can work accordingly.

IV. Future Work

Our work is still its initial state, and some features have yet to be designed, implemented, and tested. Some of the (obvious) missing pieces are described in this section.

A. Module Dependency

A module can depend on other modules. But since GP modules are created and loaded dynamically, we cannot statically determine the dependencies among modules. To detect such dependencies, when the serializer enumerates objects in the project (or a module), it needs to check which modules the objects' classes belong to. The other modules that the serializer detects are considered *dependencies* of the module, and those modules will be recorded.

B. The Externalized Form of Modules

When a module is saved to disk, the saved content would need to contain enough information to recreate it in memory without losing any information.

We will need a proper scheme to identify and store networks of modules. This is not yet implemented, but here are some goals we hope to achieve:

- **No Loss of Information:** Code may be written visually as well as textually. Data may be assembled by directly manipulating objects. The textual code with the `module` keyword in this memo represents code written textually and when that is what the user wrote, it should be

faithfully stored. At the same time, there is data (such as pictures, movies, sound, etc.) that also need to be saved.

- **Version Control and Dependency:** As noted above, a module may depend on other modules. These dependencies will need to include the identification of the modules' versions.
- **Self-contained "Timeless" module files:** Also, we would like to support the notion of "timeless" software, at least as good as Smalltalk's image files are capable of. We would like to be able to "bundle" all dependencies into one blob (file) and have a guarantee that that one blob can be run on a conformant virtual machine bit-identically, perhaps many, many, years later.
- **The Community Site:** Eventually, we envision a shared GP repository of projects, sprites and libraries with a complete history of past versions, a sort of hybrid of GitHub and the Scratch web site.

Toward these goals, one possible design of such a file is as follows: The file that represents a module would be a zip compressed directory structure. A sub-directory stores the resource data files, and the textual module definition. Another sub-directory stores a binary file that is the pre-processed file, in which resource files are internalized and code is parsed and converted to the runnable form. For a module to be properly stored and reloaded, some meta-information such as the version (perhaps the ancestor tree of versions in SHA1), and its dependencies will be needed.

The module might contain another optional sub-directory to contain other modules, thus the resulting zip file can be fully self-contained.

To ensure that a version of a module is uniquely identified globally, we would have a naming scheme. It may be assumed that a user (at least when publishing things) registers himself at the central server and obtains a unique user name. The user then is responsible for naming published projects uniquely. And we may assume that we don't support branches of a project but only a linear sequence of versions of a project. Thus, a project, (or a module), can be identified as a triple of (user name, project name, version) from the program. In the above, we used a simple file name, such as "PaintEditor.gpm", as the descriptor of a module; in the real system, we may specify a module with the fully qualified name.

V. Conclusion

This paper presented the design of the GP module system. We believe this design is simple and flexible, yet provides sufficient isolation to allow modules to be mixed together without fear of unexpected interactions.

The GP programming environment makes the concept of modules invisible to beginners by automating module creation. Our implementation shows that this module design does not impose significant complexity or performance penalties on the GP execution model.

In the future, we hope to use modules to support sharing of components, block libraries, and even programming environment extensions. We plan to facilitate sharing of such things within the GP community by creating a shared module repository.

Acknowledgments

The authors would like to thank the members of CDG, especially Todd Millstein, Alex Warth, Mahdi Eslamimehr, Aran Lunzer, Jonathan Edwards, and Alan Kay for valuable suggestions. The authors also thank the anonymous reviewers of the paper.

References

[1] Snap! `http://snap.berkeley.edu/`.
[2] App Inventor. `http://appinventor.mit.edu/`.
[3] Sayamindu Dasgupta, Shane M. Clements, Abdulrahman Y. idlbi, Chris Willis-Ford, and Mitchel Resnick. Extending Scratch: New Pathways into Programming. In *Visual Languages and Human-Centric Computing (to appear)*, 2015.
[4] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 405–428, 2010.
[5] npm: Node.js Package Manager. `http://www.npmjs.com`.
[6] Niklaus Wirth. MODULA. A Language for Modular Multiprogramming. Technical report, 1976.
[7] Alessandro Warth, Milan Stanojevic, and Todd D. Millstein. Statically Scoped Object Adaptation with Expanders. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 37–56, 2006.
[8] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Comput. Lang. Syst. Struct.*, 31(3-4):107–126, October 2005.