Complete Class System in Lua

YIQI CHEN, Stanford University, USA

Additional Key Words and Phrases: Lua, Object Oriented Programming, Class, Encapsulation, Polymorphism

ACM Reference Format:
Yiqi Chen. 2017. Complete Class System in Lua. 1, 1 (December 2017), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

This project aims to implement a complete class system in Lua. Comparing to the class system we were asked to implement in assignment 2 in this class, the complete class system implemented in this project will have stronger reliability, more functionality and cleaner API. In particular, the complete class system aims to tackle one of the major problem in class system in assignment 2, where private data members can be leaked to the outside world. The complete class system solves this problem by checking the caller when a private member is accessed or modified, and blocking the access or modification if the caller is not a function within the class. To evaluate the robustness of the class system, I wrote multiple unit tests that simulate different scenarios of maliciously access private class members from outside the class, and verified that the access is denied in all of those scenarios. Also, I implemented a graph library using the complete class system in order to demonstrate the additional features included in the complete class system.

2 BACKGROUND

2.1 Class System in Assignment 2

In assignment 2 in this class, we were asked to implement a class system which contains public methods and private data members. Specifically, the API of a class setup is in Code 1 [1].

Code 1. Class API in Assignment 2

```
local Class = class.class(
  ParentClass, {
    constructor = function(self, ...)
      -- to be called when an instance is created
    end,

    data = {
      -- table of initial values for private members
```

[1] http://cs242.stanford.edu/assignments/assign2/

Author's address: Yiqi Chen, Stanford University, Stanford, CA, USA.

XXXX-XXXX/2017/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
    -- e.g. x = 3
  },

  methods = {
    -- table of public methods that can be called on an instance
    -- e.g. foo = function(self) print(self.x) end
  },

  metamethods = {
    -- table of metamethods assigned to each instance
    -- e.g. __index = function(t, k) return nil end
  }
})
```

Although this attmept of building a class system in Lua indeed provides a workable class system with inheritance and encapsulation in the general use cases, it has some serious drawbacks:

(1) Private data members can very easily be leaked to outside world.
(2) This class system is missing some basic features in state-of-the-art class system in other languages. e.g. private methods, abstract classes.
(3) The API of this class system is too tedious. For example, a user needs to specify "data" and "method" blocks to sparate variables and functions, which does not quite align with the dynamically-typed property of Lua.

We can tolerate the above drawbacks in the assignment as the class system in the assignment is simply a prove of concept. In this project, I aim to build a complete class system that solves all the problems mentioned above. In particular, I want to prevent private data members from being accessed or modified by the outside world in all possible cases.

2.2 Other Alternatives

Since Lua does not have a built-in class system comparing to other dynamically-typed languages such as python, all class system implementations are unoffical. In the tutorial of Lua [1], there is a chapter that introduces object-oriented programming and provides examples on how a class system can be potentially implemented in Lua. The tutorial is simply a few code snippets and explanations, rather than a fully-functional module. On the other hand, there is a fully-functional class system implementation in GitHub [2] written by Roland Yonaba that has way more functionalities comparing to the class system in assignment 2, such as abstract class and static class/methods. However, the implementation does not distinguish public and private members and therefore lack the security enpowered by encapsulation, which the complete class system in this project is trying to tackle.

Encapsulation is indeed a critical concept in object oriented programming. From implementation level, it requires constraints on the language itself in order to hide data members. How to make it work with a very dynamic sripting language, Lua, is indeed a challenge towards degree of control in programming language in general.

3 APPROACH

3.1 Class API

In this project, I built a more sophisticated class system that has more features, simpler API and complete encapsulation enforcement comparing to class system in assignment 2. The full API is shown in Code 2.

Code 2. API of the Complete Class System

```
local Class = class.class(
  ParentClass, {
    abstract = true, -- Whether the class is abstract. Default is false.

    new = function(self, ...)
      -- to be called when an instance is created
    end,

    publicVariable = xxx,
    _privateVariable_ = xxx,
    publicFunction = function(self, ...) xxx end,
    _privateFunction_ = function(self, ...) xxx end,

    __metamethod = function(self, ...) xxx end
  })
```

From a higher level, the complete class system has a much cleaner API comparing to class system in assignment 2 since a user can declare all variables, methods and metamethods at the top level. Specifically, the complete class system has the following features:

- The system supports inheritance where a user can create subclasses of a particular class and inherits all its variables and functions.
- Each variable or function can be identified as 2 types:
  - Public: the member can be accessed or modified from whereever the class instance is accessible.
  - Private: the member can only be accessed or modified by functions defined within the same class or its subclasses.
- A class can be marked as abstract so that a user cannot instantiate the class.
- There is a *isInstance* function available for each class to identify whether a variable is an instance of a class or its subclasses.
- A user can declare new variables or functions to a class instance. This is an additional feature comparing to state-of-the-art class systems in other languages such as Java. Notice that the newly-declared functions does not have access to private data members in the class.
- A user can declare metamethods of each class. Notice that *__index* and *__newindex* are reserved for internal use.

3.2 Encapsulation Enforcement

While the implementation of most features mentioned above is rather straight-forward, making sure that private variables and functions cannot be accessed or modified by outside world presents a big challenge, especially in a dynamic scripting language like Lua.

The class system in assignment 2 handles this situation by exposing a public instance which only contains public functions, while keeping a private instance through lexical scoping. When a function is called, the *__index* metamethod is triggered and the class system replaces the public instance and sends the private instance into the actual function call. This approach works in the basic cases but can cause the private instance to be easily leaked to the outside world. For example, if a function returns *self*, the private instance will be returned since we pass

is_access_allowed
__index or __newindex
caller function to be evaluated
...

Table 1. Call Stack of *debug.getinfo*

the private instance into the function, and consequently all private data members can be accessible to the outside world.

We can fix this particular scenario by putting a wrapper checking return value. However, there are multiple other scenarios that private instance can be leaked (see Section 4.1). To tackle this problem, I take a different approach – instead of passing the entire private instance into a function, I use the same instance all along and do a permission check only when the actual data member is accessed or modified. Specifically, I do the following steps:

(1) Keep 2 instances for each class instance constructed by a user – a private instance which has all public and private members, hidden to the outside world through lexical scoping, as well as a public instance which is always empty, so that metamethods *__index* and *__newindex* can be triggered whenever a member of the instance is accessed or modified.
(2) During class initialization time, keep a list of references of all functions listed in the class, as well as in all parent classes.
(3) When an instance variable or function is accessed or modified, the *__index* and *__newindex* metamethods check whether the data member is public or private. If it's public, then directly return the value or modify the data. If it's private, the metamethods check whether the caller function is a member of the list mentioned in step 2, and allow access only if it is. The permission check function is shown in Code 3.

Code 3. Private Member Access Permission Check

```
local function is_access_allowed(var)
  -- Check if a variable/function is private
  if string.sub(var, 1, 1) == "_" and
      string.sub(var, var:len(), var:len()) == "_" then
    -- Fetch information about the caller function
    local info = debug.getinfo(3, "f")
    for k, v in pairs(init_hash) do
      -- Allow access only if caller function is one of the class functions
      if v == info.func then return true end
    end
    return false
  end
  return true
end
```

To explain what this function does, I need to introduce a Lua standard library called Debug [1]. The Debug library allows us to inspect the call stack of active functions through *debug.getinfo*, where parameter 3 means number of functions to trace up in the stack. The active function stack when *debug.getinfo* is called is shown in Table 1.

3.3 Iterations

While the design of the clean API and additional features of the complete class system is straight forward, coming up with the design of enforcing encapsulation while keeping the other features working does take a few iterations. The evolution of the final design is the following:

(1) Pass the private instance into function call (same as assignment 2), but put a wrapper on each place that the private instance can be leaked. This approach tries to tackle each leak case individually, but there are way more leak cases than I initially thought.
(2) After realizing that permission check in private variable access time is more effective, I tried to hash each function by it's source file name and defined line number. However, this won't work because the entire program can be written in the same line.
(3) I tried to categorize each variable into 3 types: public, protected and private, where protected variable is accessible to subclasses but private variable is not, same as what Java does. However, I ended up removing the "private" type because I need to keep the private variable in the private instance to keep the state, but I also need to know which class in the inheritant hierarchy defines it in order to implement permission check. Overriding the parent private variable in subclass by defining a variable with same name also makes things complicated. Therefore, the class system will be too complicated.

4 RESULTS

4.1 Unit Tests

In order to evaluate the class system project, I setup unit test for each feature of the complete class system to make sure it's working as intended, and private variables are hidden from the outside world under all circumstances. Testing a particular feature is rather straight forward, but to ensure private variables are hidden from the outside world, I have to simulate multiple scenarios of malicious private variable access in the unit test. Specifically, I simulate the following scenarios:

- A private variable of a class instance is accessed directly.
- A private variable is accessed by a function that is passed into a class function.
- A class function returns self and then a private variable is accessed through self.
- Self is assigned to a table inside a class function and then a private variable is accessed through self in the table.
- Self is passed into a function that is passed in a class function.
- Self is returned by a class function through coroutine and then a private variable is accessed through self.

Unit test shows that the private variable isn't available under all of the 6 scenarios above.

4.2 Application

In order to demonstrate the correctness and ease of use of the complete class system, I implemented a graph processing library using the complete class system. It has basic graph construction functions such as *addNode* and *addEdge*, as well as a few graph processing algorithms such as *getConnectedComponent* and *pageRank*. Internally, the graph is represented by adjacency list.

Object oriented programming indeed plays an important role in this library. With encapsulation, I'm able to hide the internal representation of the graph from the outside world and only reveal the API. I'm also able to hide the implementation of graph algorithms. For example, I take a recursive approach in *getConnectedComponent*, and I'm able to set the recursive helper function as a private function. Furthermore, with class inheritance, I'm able to create 2 classes, directed graph and undirected graph, with simply a few more lines of code than 1 class.

REFERENCES

[1] Ierusalimschy, R. *Programming in lua.* Roberto Ierusalimschy, 2006.
[2] Yonaba, R. Lua class system. https://github.com/Yonaba/Lua-Class-System, 2014.