Roguelike++

Analyzing class features and expressiveness

Connie Xiao
Stanford University
coxiao@stanford.edu

William Jiang
Stanford University
ziranj@stanford.edu

1 SUMMARY

In this project, we explore class functionality under the course concept of expressiveness by extending our class implementation in Assignment 2: Roguelike. More specifically, we implemented the following class features: states and a combination of components and mixins. In order to evaluate the effectiveness, for each of the new class features, we developed an additional feature to the Roguelike game. By implementing new game features using our new class features, we were able to see how effective the class features were in terms of creating a new immortal state for the hero and adding game items that influence gameplay.

2 BACKGROUND

2.1 States

In games like Roguelike, to make the game more interesting, characters can often change their states or mode multiple times during the game. For example, the Hero can switch between the "immortal" mode and the "normal" mode back and forth during the game. In order to accommodate this feature, new functionalities need to be added in Class which enable the class to create and enter different states. In addition, states must be able to create new methods or override the original methods. Adding the state feature in the Class enriches the expressiveness of the program. State feature makes programmers more productive because they can more easily and flexibly create characters and methods in different states. Without state feature, programmers might need pre-define all the different state-dependent behaviors in methods, and pass in special parameters to the methods to tell them what states you are currently in.

2.2 Component-mixins

An alternative paradigm to classes in object oriented programming (OOP) are components in a component based design. Object oriented programming, a popular programming paradigm used by well-known languages like Java, Python, and C++, includes defining subclasses that can depend on or inherit attributes with some main class. Instances of a class in OOP include the data and the ability to use of the methods defined in its class and also classes it inherits.

On the other hand, a component based design expresses an object like an entity that is composed of components. Components are designed to be multiple-use, non-context-specific, composable, encapsulated, and a unit of independent deployment and versioning[4]. In comparison to classes in OOP, components in component based design are less dependent which could lead to more code reusability. For example, let all objects be defined as X, Y, or Z.
On top of that, some of these objects can be considered good or bad. In order to represent all possible objects through object oriented programming, good or bad could each inherit from X, Y or Z or visa versa. Regardless, this would require code to be duplicated as visualized in figure 1. Instead, we could use components to express this pattern as demonstrated in figure 2. Visualizing components lends to a more horizontal diagram with fewer vertical dependencies. Arrows here represent including a component. Figure 2 depicts an object instance that is Y and good. We can see that this model even allows programmers to describe a bad Y object without much additional code, which is something that would require code duplication and/or refactoring in a object oriented class.

A drawback of using components is long method calls. Since components are contained within an instance, a method call would need to go through the component resulting in a method calls like `object.component.method()`. The method calls get longer when components interact with other components, which makes code less readable and more tedious to write. This issue could be mitigated with mixins. A design with mixins is similar to one with components but instead of adding in components to the instance, the methods are directly added into the class. Consequently, we would be able to call a method with `object.method()`.

We wanted to see if using a component based design with mixins would make it easier for programmers to express certain programming concepts in practice. Through analyzing the vantages of component based design and OOP in an implementation of a game feature, we can gain insight into the expressive qualities of both paradigms.

3 STATES

3.1 Approach

We first studied the implementation of state features through an existing code here: https://github.com/kikito/stateful.lua [3]

This existing code, `stateful.lua` can be added to the original class to enable state features. We originally tried using the `stateful.lua` with our original Class function from the assignment. However the two are not compatible and after a long time of trying to make the `stateful.lua` compatible with the original Class, we were still not successful. We eventually changed plan, and tried implementing a simplified version of `stateful.lua` inside the original Class function, so that we do not need a separate `stateful.lua` anymore.

With the new state features, adding a new state and defining methods for that state become simple. Below are an example if you want to add a new "immortal" state to the Hero, and add a new `SetHealth()` function to the immortal state (which will override the original `SetHealth()` function). This code is implemented in the `Hero.lua`.

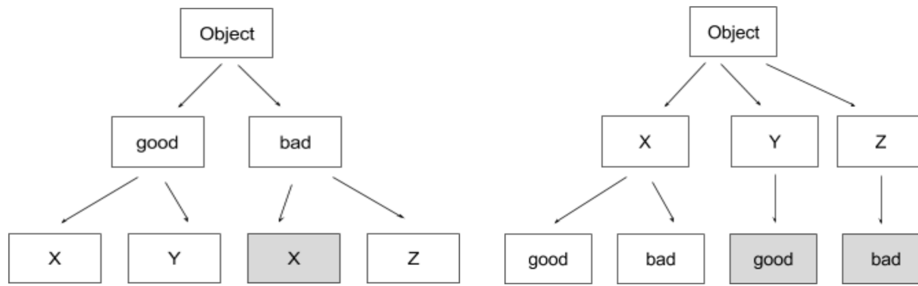

Figure 1: Object oriented designs (gray boxes represent duplicated code)

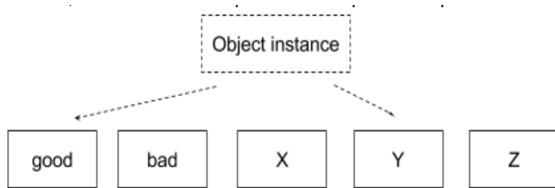

Figure 2: Component based design

```
Hero = class.class(Entity, {...})
local immortal = Hero:addState('immortal')
function immortal:SetHealth(h)
self.game:Log('I am UNBREAKABLE!!')
end
```

Similarly, you can add any number of additional states as well. Once the states and methods are created, in game.lua you can change the character's states as follows:

```
self.hero:gotoState('immortal')
```

You can also ask for the current state of an object. For example in `monster.lua`, you can say:

```
if self.game.hero.getState() == 'immortal' then ...
```

The way state features is implemented is that in the Class, we added two new values:

```
Class.states = {}
Class.currentState = nil
```

`Class.states` is a table which will store any new states that get created later. For example, after you add the immortal state and add the `immortal:SetHealth()` method, now the `Class.states` will become:

```
Class.states = {
"immortal" = {function SetHealth(h)...end}
}
```

Adding new states and methods will similarly add to this Class.states value.
In game.lua once you make the Hero go to the immortal state, then the Class.currentState will become:

```
Class.currentState = "immortal"
```

This variable is used to track the current state of the object, with `nil` being the default/original state. test test

3.2 Game feature

In the game, we placed some treasure items labeled `I` throughout the maze. The player can navigate the Hero to pick up these treasure items. Once Hero picks them up, it turns into the immortal state, and the character changes from `*` to `@` and color changes as well. In the immortal state, Hero does not lose health when being attacked by monsters, and can attack monsters just like before. Each treasure item enables Hero to become immortal for 30 steps, and after that, Hero automatically becomes normal state (mortal) again. Being immortal does not recover health, so if Hero had health of 2 before turning immortal, during immortal state Hero is invincible, but once Hero exits immortal state, it still only has a health of 2 left. The effect of treasure item is cumulative, so for example if Hero picks up 2 treasure items in a row, it will get almost 60 steps of immortality. However, when Hero hits a bomb `B`, then it dies even if it is in an immortal state.

3.3 Results

We implemented the state features in class, and used the state features to make Hero go back and forth between immortal and normal modes during the game. Therefore the success of our state feature implementation can be measured by the success of this new game feature as well as the ease of programming. As shown in the demo during our presentation, we achieved the goal of implementing these new game features. In addition, the new state features make the code very clean and intuitive. As mentioned previously, for example this was all we needed to add a new state and method to the Hero:

```
local immortal = Hero:addState('immortal')
function immortal:SetHealth(h)
self.game:Log('I am UNBREAKABLE!!')
end
```

The code is also flexible, so that in the future if you want to add some other new states, you can just add to the previous code, without having to edit the previous code.

4 COMPONENTS AND MIXINS

4.1 Approach

As a way of easing ourselves into these new designs, we first tried adding mixins into our class implementation. This helped us get a better idea of how mixins worked. In lua, this involves merging

Figure 3: Game with new features

Figure 4: Immortal state

Figure 5: Hero in immortal state

over methods from the mixin with the class's methods.

Usage:

```
local Mixin = {}
function Mixin:foo() return 'foo' end
function Mixin:bar() return 'bar' end
local ParentClass = ...
local ChildClass = ...
ParentClass:include(Mixin)
local c = ParentClass.new()
local d = ChildClass.new()
assert(c.foo() == "foo")
assert(d.bar() == "bar")
```

Next, we created a separate class file in order to implement a component based design. The component class is different from the original class in that there is a component list instead of a methods list and the metatables are not set for inheritance between child and parent classes. Each of the components is a table with methods. Components are described in the same way as classes are: with a constructor, data, and methods. Initially, we tried adding components to the class. This means that every instance of that class would have the same components. The component was initialized when it was added to the class. However, this prevent us from customizing each instance of the class because the values used to initialize the components were fixed once each component was added to each class.

Components:

```
local partOne = {
  constructor = function (self, name)
   self.name = name
  end,
  data = {
   name = '',
   var = 0
  },
  methods = {
    change = function(self, var)
     self.var = var
    end,
    getVar = function(self, var)
     return self.var
```

```
    end,
    getName = function(self)
     return self.name
    end
  }
}

local partTwo = {
  constructor = function (self, value)
   self.value = value
  end,
  data = {
   value = 0,
   two = 2
  },
  methods = {
   getTwo = function(self)
   return self.two
    end,
    change = function(self)
     self.two = 1
     return self.two
    end
  }
}
```

Usage:

```
-- easy to make multiple copies of same objects
local Entity1 = componentClass.class()
Entity1:addComponent("partOne1", partOne, "name")
Entity1:addComponent("partOne2", partOne, "other name")
Entity1:addComponent("partTwo", partTwo)
local entity1 = Entity1.new()
-- inconvenient to change initial values
local Entity2 = componentClass.class()
Entity2:addComponent("partOne1", partOne, "name")
local entity2 = Entity2.new()
```

In our second iteration we moved addComponent such that it became a method of the instance within the new method. This means that each instance is responsible for adding components as needed and initialization would occur for that specific instance. This gives us a more reasonable amount of freedom in customizing the instances.

Usage:

```
local Entity = componentClass.class()
local entity1 = Entity.new()
local entity2 = Entity.new()
entity1:addComponent("partOne1", partOne, "name")
entity1:addComponent("partOne2", partOne, "other name")
entity1:addComponent("partTwo", partTwo)
entity2:addComponent("partOne1", partOne, "new name")
entity2:addComponent("partTwo", partTwo)
```

Lastly, we combined mixins with our component implementation. We do this when adding components by adding the componentâĂŹs methods to our instance if they are unique.

```
for k, v in pairs(component.methods) do
  if type(v) == "function" and k ~= "new" then
    if self[k] == nil then
      self[k] = function(...)
       return component.methods[k](dataInst, ...)
      end
    elseif type(self[k]) == "function" then
      self[k] = function(...)
       return "duplicate method names:
         use components to call method"
      end
    end
  end
end
```

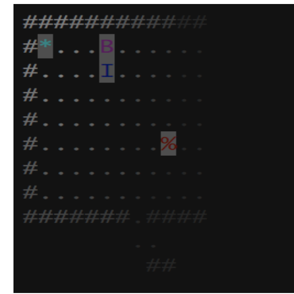

Figure 6: Items in the game (B and I)

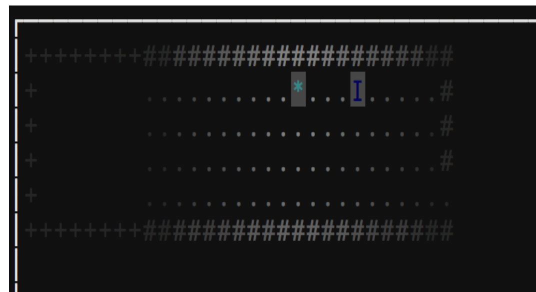

Figure 7: Immortal item

4.2 Game Feature

In order to see how useful our component-mixin class design was, we implemented an immortal item and bombs into our game. If the hero picks up the immortal item, the hero becomes immortal to the damage of the monsters for 30 steps. If the hero runs into a bomb, the bomb blows up the hero and the game automatically ends.

We have the following components: basicItem, bomb, immortal. It is constructed in a way that each item contains basicItem, which creates an item in the game that essentially does nothing (stays in one place, inconsequential to the game, contains default values). This is parallel to the entity.lua class. Then we add the other components to the relevant items we want to create.

4.3 Results

An component based design looks more readable when creating instances of the item.

```
local immortalItem = Item.new()
immortalItem:addComponent("basicItem", basicItem,
self, self:RandomFloor())
immortalItem:addComponent("immortal", immortal)
```

The component parts can serve as a description of an item and it is easy to infer the relevant initialized values chosen for particular component.

Our particular use case could have been implemented in an object oriented fashion by having the superclass be equivalent to basicItem and subclasses be bomb and immortal. This is most likely because we were simultaneously imagining how this could also be implemented with class.lua and we are more used to thinking in an object oriented manner. However, if we wanted to add more complex features later on, a component based design would be more robust and readable.

For example, if we converted all the code into a component based design and wanted to add the ability for immortal items to move, we would just create a moving component that would be included in hero, monster, and the immortal items.

It would be interesting to create treasure with a combination of object oriented programming and mixins. This would include the benefits of object oriented programing while being able to include methods or data that can be shared horizontally across objects. The drawbacks of this would include the drawbacks of using mixins, which include needing unique method names and potential mixin method conflicts in setting fields. However, we would assume that most of the functionality and attributes of the object are expressed within classes so we would not be dealing with many mixins for each class. With a minimal number of mixins, the aforementioned drawbacks are unlikely to occur. Mixins would come in handy for rarer situations to help reduce the the lines of duplicate code. While doing further researching, we found a popular OOP library for lua also has support for mixins [2], which affirms this reasoning. In addition, scala uses mixins with classes as well [1].

5 CODE

The code for this project can be found at:
`https://gitlab.com/conniexiao/cs242`

6 LIST OF WORK BY EACH STUDENT

Equal work was performed by both project members.

REFERENCES

[1] [n. d.]. Class Composition with Mixins. ([n. d.]). Retrieved December 13, 2017 from https://docs.scala-lang.org/tour/mixin-class-composition.html

[2] Enrique García Cota. 2015. Middleclass. (2015). Retrieved December 13, 2017 from https://github.com/kikito/middleclass

[3] Enrique García Cota. 2016. Stateful. (2016). Retrieved December 13, 2017 from https://github.com/kikito/stateful.lua

[4] David G Messerschmitt, Clemens Szyperski, et al. 2005. Software ecosystem: understanding an indispensable technology and industry. *MIT Press Books* 1 (2005).