Nested Trait Composition For Modular Software Development

Dependency injection is a common but verbose technique used to divide OO code into independently testable units. Here we propose extension to the trait model supporting independently testable units without the verbosity of dependency injection. We present our solution in the language L42.

Traits are units of code re+use, traditionally containing only abstract and implemented methods. When two traits `t1` and `t2` are composed, the resulting trait contains all the methods of `t1` and `t2`. It is an error if a method `m` is implemented in both `t1` and `t2`. On the other side, an abstract method can be composed with another abstract or implemented method. In former work we extended traits to include nested classes, with the intuitive idea that when two traits are composed, nested classes with the same name are recursively composed.

A common motivating example for Dependency Injection is the following: Alice and Bob are designing a videogame, where there is a `Map` containing `Item`s that are located in certain `Point`s. Various kinds of `Item`s exists: `Wall`s, `Rock`s, `Tree`s and many more. Alice will implement the `load` functionality: `load` will parse a formatted text file and use the information to fill a `Map`. Bob will implement the rest of the game, including all the kinds of `Item`s and the `Map`. Bob plans to make an efficient implementation of the `Map` as a tricky sparse matrix. Thus, bugs are likely to be present in the `Map` implementation. The various `Item`s will contains a lot of code related to the game logic.

In Java, we can use dependency injection to we write the code so that Alice can completely write and test `load` before Bob completes the rest of the game. This requires introducing a lot of boilerplate code, and to program in a very unnatural way: Alice can not write `new Map(..)` since the `Map` class may not be available yet. We need to introduce an `IMap` interface and an `IMapFactory` interface, and Alice need to provide a mock implementation designed for testing. This makes the code of Alice very involved and indirect: Alice `load` method needs to access an `IMapFactory mapFactory` and use it to create new maps, as in `mapFactory.makeMap()`. Same for `Item`s: Alice can not write `new Rock(...)`, but she needs to use an `IItemFactory`. When `load` is called in the context of a running game, the factories will create actual `Map`s, `Rock`s and so on, but in the testing environment, mock factories will create mock versions. While in Java this is very verbose and involved, it still has great advantages over using the possibly buggy `Map` of Bob: Alice can write a simple but inefficient version of map relying on a `HashMap<Point,Item>`. Professional programmers rely heavily on dependency injection: it seams like the benefits overcome the costs.

We would like to present the Java code for this example, but dependency injection make it so verbose that it would never fit into 1 page. On the other hand, our solution is quite compact:

```
Alice: Trait({//this is the code written by Alice.
  Map: {class method Map()   method Void set(Item that)}//Map has two abstract methods
  Item: {interface} //Alice just declares all the abstract signature she need.
  Rock: {implements Item     class method Rock(Num weight, Point point) }
  class method Map load(S fileName)={
    Map m=Map() //L42 do not use 'new' but instantiate objects calling
    ..//class methods with empty names, as it happens in Python.
    if line.startsWith(S"Rock") (map.set(Rock(weight:.., point: Point(x:..,y:..))))
    ..//Map, Item and Rock are local declaration in Alice code.
    }//Trait composition & merges the members with the same name.
  })//This will include nested classes Map, Item and Rock present in Bob trait.
Game: Alice & Bob & {}
```

And this is how Alice may test her code:

```
AliceMock: Alice & {
  Item: {interface     Point point}
  Rock: Data <>< {implements Item     Num weight}
  Map: Trait(Collections.hashmap(key: Point, val: Item)) & {
    method Void set(Item that)=this.put(key: that.point(),val: that)
    }
  class method Void test(S fileName,S expected)={
    map=this.load(fileName: fileName)
    Debug.test(map, expected: expected)
    }
Test1: AliceMock.test(S"justARock.txt",S"HashMap[Point(..)->Rock(..)]")
Test2: ..
..
```

Concluding, our solution is much more natural to use than conventional dependency injection, and allows writing code more naturally, without using factories.