# Nested Trait Composition For Modular Software Development

Marco Servetto (marco.servetto@ecs.vuw.ac.nz)

*What if traits have nested classes?*

*Better dependency-injection and mocking*

# Traits composition

```
class{
  String hello();
  String helloWorld()=hello()+" World";
  }
+
class{ String hello()="Hi"; }
=
class{
  String hello()="Hi";
  String helloWorld()=hello()+" World";
  }
```

- The result of the sum contains the methods of both arguments.
- If a method is present in both arguments, they need to have the same type signature, and at least one of the two needs to be abstract.

# Traits and nested classes

```
class{
  Foo=class{
    String hello();
    String helloWorld()=hello()+" World";
  }}
+
class{
  Foo=class{ String hello()="Hi";}
  }
=
class{
  Foo=class{
    String hello()="Hi";
    String helloWorld()=hello()+" World";}
  }
```

- Nested classes with the same name are recursively composed.

# Traits and state

```
trait geometryPoint = class{
  Point = class{
    Int x();
    Int y();
    static Point of(Int x,Int y);
    static Point double()=Point.of(x()*2,y()*2);
  }}

Geometry1=geometryPoint //declaring class Geometry1
... Geometry1.Point.of(3,4).double()//example usage
```

- Static methods can also be abstract.
- A class with an abstract static method returning its type, and abstract methods looking like getters/setters is a **coherent** class. Such abstract methods work like factories, getters and setters.

# Interactions between nested classes

```
trait geometryPoint = class{ //Same as before
  Point = class{Int x(); Int y();
    static Point of(Int x,Int y);
    Point double()=Point.of(x()*2,y()*2);}}

trait geometryRectangle = class{
  Point=class{Point double();} // Declare only the necessary
methods
  Rectangle=class{
    Point upLeft(); Point downRight();
    Rectangle of(Point upLeft, Point downRight);
    Rectangle double()=Rectangle.of(upLeft().double(),
      downRight().double());}
    }
  }
...
Geometry2=geometryPoint+geometryRectangle
```

```
Game = class{ //example game code, NOT MODULARISED
  Item = interface{ Point point(); Item hit();}
  Rock = class implements Item{
    Num weight(); static Rock of(Point point, Num weight);
    Item hit()=Rock.of(..);}
  Wall = class implements Item{
    Num height(); static Wall of(Point point, Num height);
    Item hit()=Rock.of(..);}
  Map = class{..//map implementation by Bob
    static Map empty()=..
    Item get(Point that)=..
    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide in lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
Main ={... Game.run()..}
```

# Alice and Bob

- Can we split the code of Alice and Bob into two traits?
- It would be nice to test Alice's code in isolation from Bob's code.
- Alice needs to create instances of `Map`, `Wall` and `Rock`.
- Bob needs to be able to call `load(String)`

```
trait alice = class{//start: all code copied. What can we remove?
  Item = interface{ Point point(); Item hit();}
  Rock = class implements Item{
    Num weight(); static Rock of(Point point, Num weight);
    Item hit()=Rock.of(..);}
  Wall = class implements Item{
    Num height(); static Wall of(Point point, Num height);
    Item hit()=Rock.of(..);}
  Map = class{..//map implementation by Bob
    static Map empty()=..
    Item get(Point that)=..
    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//alice do not use hit, is part of game logic
  Item = interface{ Point point(); Item hit();}
  Rock = class implements Item{
    Num weight(); static Rock of(Point point, Num weight);
    Item hit()=Rock.of(..);}
  Wall = class implements Item{
    Num height(); static Wall of(Point point, Num height);
    Item hit()=Rock.of(..);}
  Map = class{..//map implementation by Bob
    static Map empty()=..
    Item get(Point that)=..
    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//alice do not use hit, is part of game logic
  Item = interface{ Point point();}
  Rock = class implements Item{
    Num weight(); static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    Num height(); static Wall of(Point point, Num height);
    }
  Map = class{..//map implementation by Bob
    static Map empty()=..
    Item get(Point that)=..
    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//alice do not use getters/points
  Item = interface{ Point point();}
  Rock = class implements Item{
    Num weight(); static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    Num height(); static Wall of(Point point, Num height);
    }
  Map = class{..//map implementation by Bob
    static Map empty()=..
    Item get(Point that)=..
    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//alice do not use getters/points
  Item = interface{ }
  Rock = class implements Item{
    static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    static Wall of(Point point, Num height);
    }
  Map = class{..//map implementation by Bob
    static Map empty()=..

    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//the implementation of map methods is not needed
  Item = interface{ }
  Rock = class implements Item{
    static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    static Wall of(Point point, Num height);
    }
  Map = class{..//map implementation by Bob
    static Map empty()=..

    Void set(Item that)=..}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//the implementation of map methods is not needed
  Item = interface{ }
  Rock = class implements Item{
    static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    static Wall of(Point point, Num height);
    }
  Map = class{
    static Map empty();

    Void set(Item that);}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//finally, run is not needed.
  Item = interface{ }
  Rock = class implements Item{
    static Rock of(Point point, Num weight);
    }
  Wall = class implements Item{
    static Wall of(Point point, Num height);
    }
  Map = class{
    static Map empty();

    Void set(Item that);}
  static Void run()=..this.load(..).. //implemented by Bob
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

```
trait alice = class{//We only keep the abstract signatures she uses!
  Item = interface{ }
  Rock = class implements Item{static Rock of(Point point, Num weight);}
  Wall = class implements Item{static Wall of(Point point, Num height);}
  Map = class{static Map empty(); Void set(Item that);}
  //-------------------------------------
  static Map load(String fileName)={//Alice writes load(_)
    Map map=Map.empty();
    //read from file and divide into lines,
    for(String line: lines) {load(map,line);}}
  static Void load(Map map,String line)={
    //example line: S"Rock 23 in 12, 7"
    ListNum ns=line.readNums();
    if (line.startsWith("Rock"))
      map.set(Rock.of(ns.get(0),Point.of(ns.get(1),ns.get(2))));
    if (line.startsWith("Wall")) ..
    ..
    }}
```

# Alice Trait

- Alice can write all its code in a single trait with nested classes. She can declare all the dependencies she needs by just declaring classes with abstract methods.
- The code of Alice is untouched, no need to insert new interfaces/factories or other programming patterns.
- Alice can now easily test her code in isolation!

```
AliceMock = alice + class{
  Item = interface{String info();}
  Rock = class implements Item{
    static Rock of(Point point, Num weight)=
      Rock.of("Rock:"+point+"->"+weight);
    static Rock of(String info);
    }
  Wall =..;
  Map ={
    String info();
    Void info(String that);
    static Map of(String that);
    static Map empty()=Map.of("");
    Void set(Item i)=info(info()+i.info()+"\n");
    }
  static Void test(String fileName, String expected)={
    Map map = load(fileName);
    assert map.info().equals(expected);
    }
  }
...
AliceMock.test("justARock.txt", "Rock:Point(5,6)->35\n");
AliceMock.test(...);
```

# Traits do dependency injection

- Thanks to declaring the abstract requirements, including factories, the code can be written
- In Java, in order to use DI components never “new” other components, but create them with factory objects, and they will always refer to each others using interfaces. This requires a very unnatural and involved way of coding
- The proposed language solves this issue and allows different modules of code to be independently developed
- Just declare your abstract requirement

# Arbitrary splitting any code base

- Any program can be split in multiple independent pieces in this way
- Any arbitrary split is possible
- Good design = less abstract declarations?
- Any programmer can just work in its own traits

# Concluding

- I'm experimenting with traits with nested classes
- Typical example: Expression problem
- However, it is not the only one.
  Dependency injection and Mocking also can be easily supported
- Part of the bigger language 42 project (https://L42.is)

```
//common code:
class Point{..}//most of these require their own file
interface Map{..}
interface Item{..}
interface Rock extends Item {..}
interface ItemFactory{Rock makeRock(Point point, int weight); ..}
interface MapFactory{Map makeMap();}

class MapLoader{//Alice code
  ItemFactory items; MapFactory maps;
  MapLoader(ItemFactory i,MapFactory m){items=i;maps=m;}
  Map load(String fileName){..maps.makeMap()..}
  Void load(Map map,String fileName){..items.makeRock(..)..}
  }

class MockMap implements Map{..}//Alice mocking code
class MockMapFactory implements MapFactory{
  public Map makeMap(){return new MockMap();} }
class MockItemFactory implements ItemFactory{
  public Rock makeRock(..){return new MockRock(..);}
  public Wall makeWall(..){return new MockWall(..);} }
class MockRock implements Rock{..}
class MockWall implements Wall{..}

class Tester{
  static void test(String fileName,String expected){
    MapLoader m=new MapLoader(new MockMapFactory(),new MockItemFactory());
    Map map=m.load(fileName)
    assert map.toString().equals(expected);
  }}
```