J(i)azzing up Java Collections

Josh Baratz

May 22, 2002

1 Abstract

This paper explores the effects of Units and Mixins when applied to core Java utilities. It gives an analysis of the changes necessary, and benefits from implementing the collections api using a framework of Jiazzi. Attention is paid to the affects of Units on subclass based polymorphism, static type checking, and module interface and name dependency.

2 Background

While looking around for a good problem with which to play around with Jiazzi, it was suggested that I look at Java's collection API. In the tradition of research everywhere, I blindly dove in, hoping to find something interesting which I could explore further.

The normal candidates for program revision weren't available - Bloch clearly put a lot of thought into his design of the API. The normal tricks of reducing dependency through the use of interfaces was already applied. There were a few areas however, where Jiazzi is able to bring about improvements over the Java implementation. I focused on bringing name dependencies from the java source file to the unit file, a standard Jiazzi trick. In addition, I looked at possible improvements that could be made on the collections polymorphism. Java implements subtype polymorphism, and in theory, Jiazzi allows parametric polymorphism.

3 Summary and Evaluation

To see what was possible with Jiazzi, I took a subset of the collections api and pared it down to a size that readily allowed proof of concept programming, yet wasn't so trivial as to make it completely different from a real implementation. I then put together a test framework that allowed me to see what effect Jiazzi has on actually code.

As per my previous experience, the java source code swapped name dependencies for interface dependencies, and the name dependence moved to the linking compound. More interesting was the effect on polymorphism. By using the Jiazzi constructs of lexemes and generics, I was able to implement a type of parametric polymorphism. The endless type casts when using the collections api was always a source of frustration, and replacing that by introducing one small module is well worth it for me. There are times when subtype polymorphism may be useful (mostly for supporting legacy code IMHO), and the way Jiazzi introduces parameters makes it possible to bind Object as the parameter - turning the class back to subtype polymorphic.

4 Lessons Learned

I was pleasantly surprised by this experience - I thought I was going to learn about modules and dependencies, but instead gained a better insight into methods of polymorphism. The complicated nature of the work gave me a better appreciation for what Jiazzi is capable of, and a better understanding of separate compilation and linking. It also reminded me just how painful working with a research programming tool can be.

5 Appendix A: MDD

6 Appendix B: Code

jwb.finalprog.unit

```
1     compound jwb.finalprog {
2       export main: program;
3
4     } {
5       local
```

```
6           objects: obj.impl ,
7           lmain : main,
8           lobjc: objc,
9           lobj2c: obj2c;
10
11      link package
12
13          objects@basicimpl to *@testobjs,
14          objects@basicimpl to objects@testobjects,
15          lobjc@outc to lobjc@container,
16          lobjc@outc to lmain@container1,
17          lobj2c@outc to lobj2c@container,
18          lobj2c@outc to lmain@container2,
19          lmain@out to *@main;
20
21
22  }
```

Main.java

```
1   package out;
2
3   import java.io.*;
4   public class Main extends java.lang.Object {
5
6       public static void main(java.lang.String[] args) {
7
8
9           container1.UnarySet obj_1_set = new container1.UnarySet();
10          testobjs.Object1 obj1 = new testobjs.Object1();
11          container2.UnarySet obj_2_set = new container2.UnarySet();
12          testobjs.Object2 obj2 = new testobjs.Object2();
13
14          obj_1_set.add(obj1);
15          obj_2_set.add(obj2);
16
17          testobjs.Object1 retobj1 = obj_1_set.get();
18          testobjs.Object2 retobj2 = obj_2_set.get();
19
20
21          System.out.println("call␣returned␣something␣with␣value:␣" + retobj1.getName());
22          System.out.println("call␣returned␣something␣with␣value:␣" + retobj2.getString());
23
24      }
25  }
26
```

main.unit

```
atom main {


  import container1: objcontainer;
  import container2: obj2container;
  import testobjs: testobjs;
  export out: program;
  bind package
       container1 to container1@container,
       container2 to container2@container,
       testobjs to *@testobjs;
}
```

program.sig

```
signature program = {

  class Main extends Object {
    public static void main(String args [] );
  }
}
```

objc.unit

```
compound objc {

export outc: objcontainer;
import container extends outc;
import testobjs : testobjs;
bind package container to *@container, testobjs to *@testobjs;

} {
local cont : set.impl;
link package cont@basicimpl to *@container, cont@basicimpl to outc;
link generic testobjs$Object1 to cont@TARGET;
}
```

objcontainer.sig

```
signature objcontainer = a: container + {

package container, testobjs;

bind generic testobjs.Object1 to a@TARGET;

bind package
       container to *@container;


}
```

obj2c.unit

```
compound obj2c {

export outc: obj2container;
import container extends outc;
import testobjs : testobjs;
bind package container to *@container, testobjs to *@testobjs;

} {
local cont : set.impl;
link package cont@basicimpl to *@container, cont@basicimpl to outc;
link generic testobjs$Object2 to cont@TARGET;
}
```

obj2container.sig

```
signature obj2container = a: container + {

package container, testobjs;

bind generic testobjs.Object2 to a@TARGET;

bind package
       container to *@container;


}
```

set.impl.unit

```
atom set.impl {

generic TARGET;
```

```
4
5       export basicimpl: container;
6       import container extends basicimpl;
7
8       bind package container to *@container;
9       bind generic TARGET to *@TARGET;
10      }
```

UnarySet.java

```
1       package basicimpl;
2
3       public class UnarySet extends java.lang.Object {
4
5           protected var.TARGET state;
6
7           public UnarySet() {};
8
9           public void add(var.TARGET t) {
10              state = t;
11          }
12
13          public var.TARGET get(){
14              return(state);
15              /* Danger - Rep Exposure. it's okay for proof of concept though */
16          }
17      }
```

container.sig

```
1       signature container = g:generic + {
2
3       package container;
4       generic TARGET;
5
6         bind package container to g@generic;
7         bind lexeme UnarySet to g@Generic;
8
9       class UnarySet extends Object {
10              UnarySet();
11              void add(TARGET t);
12              TARGET get();
13      }
14      }
```

generic.sig

```
signature generic = {
  lexeme Generic;
  package generic;
  class [Generic] extends Object { (); }
}
```

obj.impl.unit

```
atom obj.impl {

export basicimpl: testobjs;
import testobjects extends basicimpl;

bind package testobjects to *@testobjs;

}
```

Object1.java

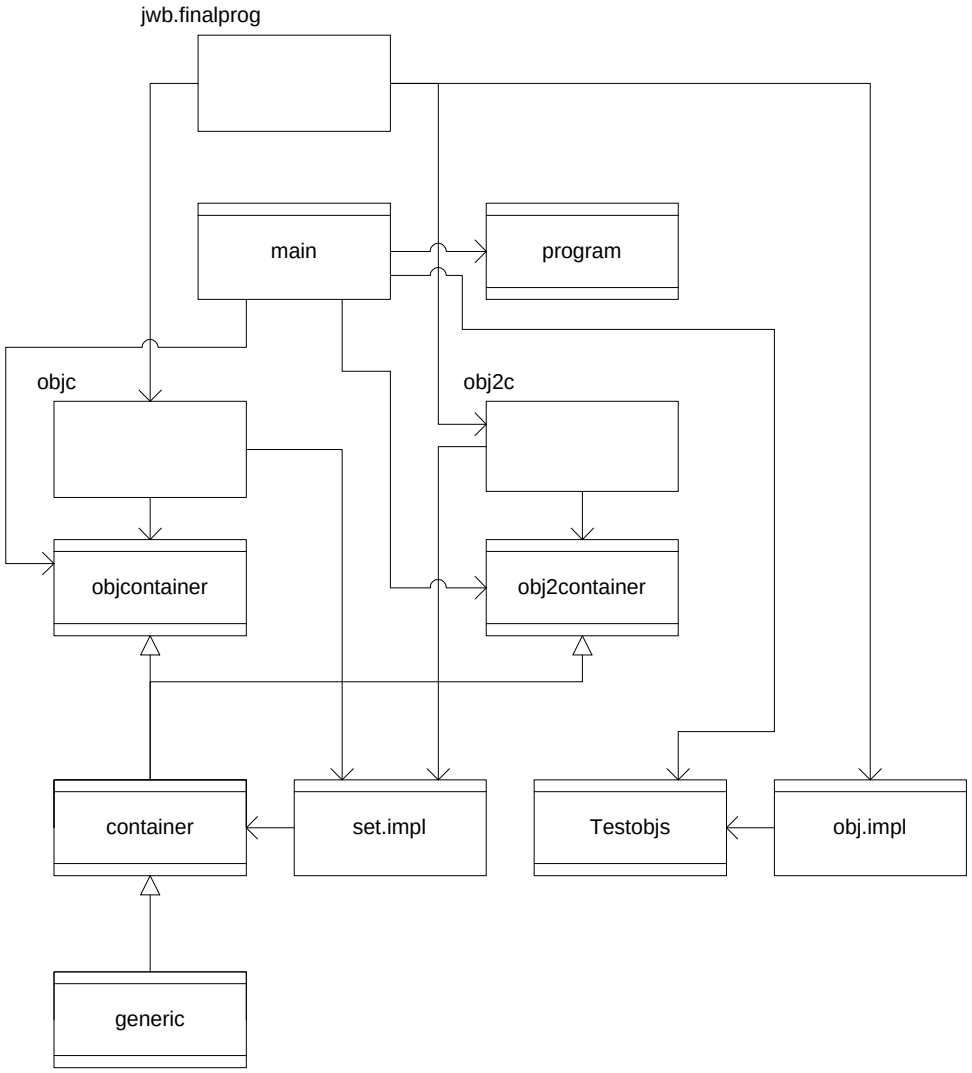
```
package basicimpl;

public class Object1 extends java.lang.Object {

    public Object1() {};

    public String getName() {
        return("this␣is␣a␣type␣1␣object");
    }
}

```

Object2.java

```
package basicimpl;

public class Object2 extends java.lang.Object {

    public Object2() {};

    public String getString() {
        return("this␣is␣a␣type␣2␣object");
    }
}

```

testobjs.sig

```
1   signature testobjs = {
2
3   package testobjs;
4
5   class Object1 extends Object {
6           Object1();
7           String getName();
8   }
9
10  class Object2 extends Object {
11          Object2();
12          String getString();
13  }
14
15  }
```


jwb.finalprog
main
program
objc
obj2c
objcontainer
obj2container
container
set.impl
Testobjs
obj.impl
generic