

**JavaFX as a Domain-specific Language  
in Scala / Groovy**

**CS 297 Report**

Sadiya Hameed  
sadiya@gmail.com

Advisor: Dr. Cay Horstmann  
horstmann@cs.sjsu.edu

San José State University  
Department of Computer Science

December 18, 2007

## TABLE OF CONTENTS

Introduction.....	1
1. Domain Specific Languages .....	2
2. JavaFX .....	2
2.1. List of Important Features .....	2
2.1.1. Incremental dependency-based Evaluation .....	3
2.1.2. Update Triggers.....	3
2.1.3. List Comprehensions .....	4
2.1.4. dur operator.....	4
2.1.5. do and do later.....	5
3. Scala.....	5
3.1. List of Features .....	5
3.1.1. Operators as valid Identifiers.....	5
3.1.2. Single Parameter methods as infix operator .....	5
3.1.3. Parameterless methods .....	5
3.1.4. Properties .....	6
3.1.5. Functions .....	6
3.1.6. Mixin Class Composition .....	7
3.1.7. Views.....	8
4. Groovy .....	9
4.1. List of Features .....	9
4.1.1. Parenthesis less methods & named parameters .....	9
4.1.2. Closure.....	9
4.1.3. Categories .....	10
4.1.4. DelegatingMetaClass.....	10
5. Progress.....	12
5.1. Scala Prototype .....	12
5.2. Groovy Prototype.....	12
6. Conclusion & Future work.....	13
References.....	14
Appendix A: Code.....	i
1. Scala Code:.....	i
2. Groovy Code: .....	iv

## INTRODUCTION

Domain-specific programming languages (DSLs) are designed for a particular problem domain and promise substantial expressiveness and ease of use in their specialized area over general-purpose programming languages. JavaFX is such a domain-specific language aimed at speedy development of rich-client Java user interfaces. The core of JavaFX is JavaFX Script, a declarative scripting language with a high degree of interactivity with Java classes.

It seems unfortunate that JavaFX is yet another language. The modern trend is to provide DSLs inside a larger host language. The goal of the project is to examine the feasibility of mimicking the functionalities provided by JavaFX as a DSL in Groovy and Scala languages and to reason about the suitability of these languages as DSL hosts.

Scala claims to have been invented for just this purpose. “Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries:

- any method may be used as an infix or postfix operator, and
- closures are constructed automatically depending on the expected type (target typing).

A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.” [1]

Groovy has had practical success in providing DSLs for XML builders, ORM, etc. and its builders claim that it is particularly well suited for writing a DSL. Groovy provides various features to let you easily embed DSLs in your Groovy code. e.g.

- you can also create your own control structures by passing closures as the last argument of a method call
- it is possible to add dynamic methods or properties (methods or properties which don't really exist but that can be intercepted and acted upon) by implementing `GroovyObject` or creating a custom `MetaClass`, etc. [3]

The plan was to create prototypes in both languages to facilitate evaluation and comparison of their abilities to be DSL hosts. Incremental dependency-based evaluation, an important feature of JavaFX, was chosen for the initial implementation. The report gives an overview of JavaFX features. Next it provides a list of features for Scala and Groovy that facilitate addition of dynamic behavior in these languages. The report also gives details of the prototypes, of the selected JavaFX feature, created in Scala and Groovy.

## 1. DOMAIN SPECIFIC LANGUAGES

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. They can be formally defined as:

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” [9]

DSL development is hard, requiring both domain knowledge and language development expertise. According to Mernik et al., the easiest way to design a DSL is to base it on an existing language. Possible benefits are easier implementation and familiarity for users, but the latter only applies if users are also programmers in the existing language. One of the approaches to designing a DSL listed according to Mernik is to take an existing language and extend it with new features that address domain concepts. In most applications of this pattern, the existing language features remain available. The challenge is to integrate the domain specific features with the rest of the language in a seamless fashion. This is the approach we plan to employ in creating the JavaFX like DSL in existing languages of Scala and Groovy.

## 2. JAVA FX

JavaFX is a domain-specific language aimed at speedy development of rich-client Java user interfaces. The core of JavaFX is JavaFX Script, a declarative, statically typed scripting language with a high degree of interactivity with Java classes. It has First-class functions, list-comprehensions, and incremental dependency-based evaluation. It can make direct calls to Java APIs that are on the platform. Since JavaFX Script is statically typed, it has the same code structuring, reuse, and encapsulation features (such as packages, classes, inheritance, and separate compilation and deployment units) that make it possible to create and maintain very large programs using Java technology.

Although the official name of the scripting language is JavaFX Script it is generally referred to as JavaFX as it is the core of the JavaFX family. This report also uses JavaFX to refer to the scripting language.

### 2.1. *List of Important Features*

Following is a list of important JavaFX features. The selection is based on the most widely used features of JavaFX.

### 2.1.1. Incremental dependency-based Evaluation

Attribute values can be declared to be dependent on (bound to) expressions involving other attributes. Thus, when a referenced attribute changes its value all attributes directly or indirectly dependent on them will change their values accordingly with no intervening procedural logic. This is similar to the way formula cells in a spreadsheet change their values whenever one or more cells they depend upon change their values. This is especially useful for GUI development where maintaining model and view attributes in sync usually requires complex procedural logic.

Consider the following model/View type example, the value of the View's title is dependent upon the model's saying attribute. The dependency is bi-directional.

```
import javafx.ui.*;

class HelloWorldModel {
  attribute saying: String;
}
var model = HelloWorldModel {
  saying: "Hello World"
};
Frame {
  title: bind "{model.saying} JavaFX"
  width: 200
  content: TextField {
    value: bind model.saying
  }
  visible: true
};
```

Initially the title and TextField's value are set by model's saying attribute. When the user updates the TextField's value the model's saying attribute is automatically updated (which in turn updates the title).

### 2.1.2. Update Triggers

JavaFX provides SQL-like triggers for data modification since it does not have constructors, destructors and setters. A trigger consists of a header and a body. The header specifies the type of event the trigger applies to. The body of the trigger is a procedure that executes whenever the specified event occurs. Triggers also behave like member functions/operations, in that the context object is accessible inside the body via the `this` keyword. Triggers are available on the following functions:

1. new
2. insert
3. delete
4. replace

For example, replace triggers perform an action whenever the value of a single-valued attribute or an element of a multi-valued attribute is replaced, such as:

```

import java.lang.System;

class X {
  attribute nums: Number*;
  attribute num: Number?;
}

trigger on X.nums[oldValue] = newValue {
  System.out.println("replaced {oldValue} with {newValue} at position
  {indexof newValue} in X.nums");
}

trigger on X.num[oldValue] = newValue {
  System.out.println("X.num: just replaced {oldValue} with
  {newValue}");
}

var x = X {
  nums: [12, 13]
  num: 100
};

x.nums[1] = 5; //prints just replaced 13 with 5 at position 1 in X.nums
x.num = 3; // prints X.num: just replaced 100 with 3
x.num = null; // prints X.num: just replaced 3 with null

```

### 2.1.3. List Comprehensions

A list comprehension consists of one or more input lists, an optional filter and a generator expression. Each source list is associated with a variable. The result of the list comprehension is a new list which is the result of applying the generator to the subset of the Cartesian product of the source lists' elements that satisfy the filter. For example:

```

select n*n from n in [1..10] where (n%2 == 0); //yields [4,16,36,64,100]

var a:Integer* =
  foreach(n in [1..4], m in [100,200] where (n%2 == 0) )
    n*m; //yields [200, 400, 400, 800]

```

*select*, *foreach* and *where* are JavaFX operators.

### 2.1.4. dur operator

The *dur* operator (*dur* stands for duration) allows you set a variable to an entire range of values over the span of time. When assigning any value with an array of possible values and the operator *dur*, JavaFX will create a background thread (or, more likely, push events into the AWT event queue) which will assign the range of values into the variable. By binding the location, size or transformation of a graphical element, this can create a smooth animation. The documentation for the *dur* operator is sparse and the syntax is still being worked out.

### 2.1.5. do and do later

JavaFX allows you to execute a portion of code in a separate thread by placing that code into a `do` statement. Using this technique the AWT Event Dispatch Thread (EDT) will be able to process all the incoming events. The `do` statement has a second form, `do later`, that allows for asynchronous execution of its body in the EDT rather than synchronous execution in a background thread, similar to the functionality provided by `java.awt.EventQueue.invokeLater()`.

## 3. SCALA

Scala fuses object-oriented and functional programming in a statically typed programming language. It is aimed at the construction of components and component systems. Scala programs resemble Java programs in many ways and they can seamlessly interact with code written in Java.

### 3.1. List of Features

Below is a list of Scala features that might be useful for DSL creation.

#### 3.1.1. Operators as valid Identifiers

Scala treats operator names as ordinary identifiers. So, an identifier is either a sequence of letters and digits starting with a letter, or a sequence of operator characters. The precedence of a user-defined infix operator is based on the first character of the operator name. It coincides with the operator precedence of Java for those operators that start with an operator character used in the languages. e.g.:

#### 3.1.2. Single Parameter methods as infix operator

Any method which takes a single parameter can be used as an infix operator in Scala. This is good for syntactic sugar such as:

```
System exit 0 // call System.exit(int) using infix notation
Thread Sleep 10 // call Thread.sleep(int) using infix notation
```

#### 3.1.3. Parameterless methods

Such methods are invoked every time their name is selected. No arguments are passed.

```
class Complex(real: double, imaginary: double){
  def re = real //re and im are parameter-less methods
  def im = imaginary
}
```

### 3.1.4. Properties

For every definition of a variable `var x: T` in a class, Scala defines setter and getter methods. Every mention of the name `x` in an expression is then interpreted as a call to the parameterless method `x`. Furthermore, every assignment `x = e` is interpreted as a method invocation `x._=(e)`. The treatment of variable accesses as method calls makes it possible to define properties (in the C# sense) in Scala. For instance, the following class Celsius defines a property degree which can be set only to values greater or equal than 273.

```
Class Celsius {
  private var d: Int = 0
  def degree: Int = d
  def degree_=(x: Int): Unit = if (x >= 273) d = x
}
```

### 3.1.5. Functions

1. Scala allows anonymous, curried, nested and higher order functions. You can define functions, anonymous functions and closures anywhere.

```
//define anonymous function and call it
((i: Int, j: Int) => i+j)(3, 4)

//nested inner functions can access outer's locals and arguments
def outer(s: String) = {
  def inner() = {
    System.out.println("outer's 's': " + s);
  }
  inner();
}
```

2. Methods can be used as arguments to functions (similar to delegates concept in C#). For example function `forall` is defined to be true if the predicate passed holds true for all elements of the passed array. Then the expression `forall(row, 0 ==)` tests whether `row` consists only of zeros. Here, the `==` method of the number 0 is passed as argument. This illustrates that methods can themselves be used as values in Scala.
3. Functions are objects with apply methods so function `foo(x)` is actually `foo.apply(x)`. Special syntax exists for function applications appearing on the left-hand side of an assignment; these are interpreted as applications of an update method. So, `a(i)` being the array element at index `i`, the assignment `a(i) = a(i) + 1` is interpreted as `a.update(i, a.apply(i) + 1)`.

### 3.1.6. Mixin Class Composition

As opposed to languages that only support single inheritance, Scala has a more general notion of class reuse. Scala makes it possible to reuse the new member definitions of a class (i.e. the delta in relationship to the superclass) in the definition of a new class. This is expressed as a *mixin-class composition*. Consider the following abstraction for iterators.

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}
```

Next, consider a mixin class which extends **AbsIterator** with a method **foreach** which applies a given function to every element returned by the iterator. To define a class that can be used as a mixin we use the keyword **trait**.

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit =
    while (hasNext) f(next)
}
```

Here is a concrete iterator class, which returns successive characters of a given string:

```
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s.charAt(i); i += 1; ch }
}
```

We would like to combine the functionality of **StringIterator** and **RichIterator** into a single class. With single inheritance and interfaces alone this is impossible, as both classes contain member implementations with code. Scala comes to help with its mixin-class composition. It allows the programmers to reuse the delta of a class definition, i.e., all new definitions that are not inherited. This mechanism makes it possible to combine **StringIterator** with **RichIterator**, as is done in the following test program which prints a column of all the characters of a given string.

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator
    val iter = new Iter
    iter.foreach(System.out.println)
  }
}
```

The **Iter** class in function **main** is constructed from a mixin composition of the parents **StringIterator** and **RichIterator** with the keyword **with**. The first parent is called the superclass of **Iter**, whereas the second (and every other, if present) parent is called a *mixin*.

### 3.1.7. Views

Views are implicit conversions between types. They are typically defined to add some new functionality to a preexisting type. For instance, assume the following abstract generic list and a trait of simple generic sets:

```
abstract class GenList[+T] {
  def isEmpty: boolean
  def head: T
  def tail: GenList[T]
}

trait Set[T] {
  def include(x: T): Set[T]
  def contains(x: T): boolean
}
```

A view from class *GenList* to class *Set* is introduced by the following method definition.

```
implicit def listToSet[T](xs: GenList[T]): Set[T] =
  new Set[T] {
    def include(x: T): Set[T] = xs prepend x
    def contains(x: T): boolean =
      !isEmpty && (xs.head == x || (xs.tail contains x))
  }
```

Hence, if *xs* is a *GenList[T]*, then *listToSet(xs)* would return a *Set[T]*. The only difference with respect to a normal method definition is the *implicit* modifier. This modifier makes views candidate arguments for implicit parameters, and also causes them to be inserted automatically as implicit conversions. Say *e* is an expression of type *T*. A view is implicitly applied to *e* in one of two possible situations:

1. when the expected type of *e* is not (a supertype of) *T*, or
2. when a member selected from *e* is not a member of *T*.

For instance, assume a value *xs* of type *GenList[T]* which is used in the following two lines.

```
val s: Set[T] = xs;
xs contains x
```

The compiler would insert applications of the view defined above into these lines as follows:

```
val s: Set[T] = listToSet(xs);
listToSet(xs) contains x
```

## 4. GROOVY

### 4.1. List of Features

Groovy provides many dynamic features that can be useful for creating DSLs. This report only discusses a few that were used in the prototype creation. Groovy is an evolving language at the time being and many of these features have been added to the language very recently and are only available under the developer version of the language.

Groovy supports metaobject protocol (MOP) which facilitates DSL creation in Groovy. A metaobject protocol (MOP) is an interpreter of the semantics of a program that is open and extensible. Therefore, a MOP determines what a program *means* and what its behavior is, and it is extensible in that a programmer can alter program behavior by extending parts of the MOP. The MOP may manifest as a set of classes and methods that allow a program to inspect the state of the supporting system and alter its behavior. MOPs are implemented as object-oriented programs where all objects are metaobjects.[3]

#### 4.1.1. Parenthesis less methods & named parameters

Method calls in Groovy can omit the parenthesis if there is at least one parameter and there is no ambiguity.

```
println "Hello world"  
System.out.println "Nice cheese Gromit!"
```

It is also possible to omit parenthesis when using named arguments. This makes for nicer DSLs:

```
compare fund: "SuperInvestment", withBench: "NIKEI"  
monster.move from: [3,4], to: [4,5]
```

When calling a method you can pass in named parameters. Parameter names and values are separated by a colon (like the Map syntax) though the parameter names are identifiers rather than Strings. Currently this kind of method passing is only implemented for calling methods which take a Map or for constructing JavaBeans.

#### 4.1.2. Closure

A closure is like a "code block" or a method pointer. It is a piece of code that is executed at a later point. Groovy provides support for closures. For example, it is possible to define action listeners inline through closures.

```
//Add a property Change Listener to myProperty  
myProperty.propertyChange = { e -> println "property changed";}
```

The closure is passed to the method on the listener interface (*propertyChange*), and not to the interface itself (*PropertyChangeListener*).

### 4.1.3. Categories

Categories are used to add new methods and properties to an existing class. It is a feature borrowed from Objective-C. To add a method to a class T, simply define a new class with a static method whose first parameter is of type T and then pass this class to the `use` keyword. Here is a simple example:

```
class StringCategory {
    static String lower(String string) {
        return string.toLowerCase()
    }
}

use (StringCategory) {
    assert "test" == "TeSt".lower()
}
```

### 4.1.4. DelegatingMetaClass

Each groovy object has a meta class that is used to manage the dynamic nature of the language. This class intercepts calls to groovy objects and adds the groovy behavior. It is possible to replace a meta class to adjust the default behavior of a class using *DelegatingMetaClass*. There is more than one way to replace a meta class using *DelegatingMetaClass*. The following is the package name convention solution to replacing meta class.

#### Package Name Convention Solution

Any class can have a custom meta class loaded at startup time by placing the meta class into a well known package with a well known name.

```
groovy.runtime.metaclass.[YOURPACKAGE].[YOURCLASS]MetaClass
```

The following example shows how we can change the behavior of the String class. First we define the custom meta class.

```
package groovy.runtime.metaclass.java.lang

class StringMetaClass extends groovy.lang.DelegatingMetaClass{
    StringMetaClass(MetaClass delegate){
        super(delegate); //delegate contains the String instance
    }

    public Object invokeMethod(Object a_object, String a_methodName,
        Object[] a_arguments){
        return "changed ${super.invokeMethod(a_object, a_methodName,
            a_arguments)}"
    }
}
```

The actual class that uses the enhanced features is very simple. Notice that there are no extra imports or any mention of the meta class. The mere package and name of the class tells the groovy runtime to use the custom meta class.

```
class DelegatingMetaClassPackageImpliedTest extends GroovyTestCase{
    void testReplaceMetaClass() {
        assertEquals "changed hello world", "hello world".toString()
    }
}
```

## 5. PROGRESS

The prototypes were to mimic the functionality of JavaFX's incremental dependency-based evaluation feature (For details of this feature, consult section 3.1.1 of this report). The sample test case was to create two `JTextField` instances and bi-directionally bind their text property to each other, such that a change of value in one triggered a change in the other. The desired syntax was `textfield1.text bind textfield2.text`

### 5.1. Scala Prototype

The Scala prototype was accomplished using the Views feature of Scala. For the sample test case, the user needs to import the `BoundJTextField` class using the syntax given below. The text fields can then be bound on their text property using the `bind` operator with the desired syntax.

```
import boundUtilities.BoundJTextField._;

object BindTester {
  def main(args: Array[String]) : Unit = {
    val field1 = new JTextField;
    val field2 = new JTextField;

    //bi-directional binding between the text property of TextFields
    field1.text bind field2.text;
  }
}
```

For detailed Scala source code of `BoundJTextField` and the test case consult Appendix A section 1.

### 5.2. Groovy Prototype

The Groovy prototype was created using the `DelegatingMetaClass` feature of Groovy. For the sample test case, the user does not need any special imports, as long as the required package (with `BoundJTextField` etc.) is in the classpath. To bind, the user needs to invoke the `property` field on the text field and then the required property(`text`) that needs to be bound. The final syntax for bind in Groovy was slightly different than the desired syntax listed above as we had to add an extra field(`property`) in the bind call.

```
class BindTester {
  static void main(args) {
    JTextField field1 = new JTextField();
    JTextField field2 = new JTextField();

    //bi-directional binding between the text property of TextFields
    field1.property.text.bind field2.property.text;
  }
}
```

For complete Groovy source code of `BoundJTextField`, custom meta class and the test case, consult Appendix A section 2.

## 6. CONCLUSION & FUTURE WORK

Both Scala and Groovy come with a variety of features useful for constructing DSLs. Though many constructs exist to dynamically change or augment the behavior of a class, our prototype feature required adding behavior to a specified property of a class and not the class itself. We were able to implement the prototype with the required syntax in Scala but in Groovy we had to use a slightly longer syntax. In comparison to Groovy, Scala is less volatile, has support for generics, allows use of operator names as valid identifiers for methods and allows single parameter methods to be called using infix syntax. On the other hand, Groovy has a much richer and growing, feature set for adding dynamic behavior. Future work will concentrate on adding other features of JavaFX to our DSLs, such as the `dur` operator and list comprehensions etc. Work will also be done to evolve the bind to enable unidirectional binding between a property and an expression.

## REFERENCES

- [1] Henry, K. 2006. A crash overview of groovy. *Crossroads*, 12, 3, ACM Press, May 2006.
- [2] JavaFX. Sun Microsystems. (Accessed September 2007).  
<http://www.sun.com/software/javafx/index.jsp>.
- [3] Kiczales, G., des Rivieres, J., and Bobrow, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [4] Mernik, M., Heering, J., and Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)* 37, 4 (Dec. 2005), 316-344.
- [5] Odersky et al. A Tour of the Scala Programming Language. Programming methods laboratory EPFL, May 2007. Available from  
<http://www.scalalang.org/docu/files/ScalaTour.pdf>
- [6] Odersky, M. And Zenger, M. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 41-57.
- [7] Odersky, M. 2006. The Scala experiment: can we provide better language support for component systems?. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA, January 11 - 13, 2006). POPL '06. ACM, New York, NY, 166-167.
- [8] The JavaFX Script Programming Language. (Accessed September 2007).  
[https://openjfx.dev.java.net/JavaFX\\_Programming\\_Language.html](https://openjfx.dev.java.net/JavaFX_Programming_Language.html)
- [9] van Deursen, A., Klint, P., and Visser, J. 2000. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.* 35, 6 (Jun. 2000), 26-36.
- [10] Writing domain specific languages. (Accessed November 2007).  
<http://groovy.codehaus.org>, 2006

## APPENDIX A: CODE

### 1. Scala Code:

Implementation of the `BoundProp` class:

```
package boundUtilities;

import java.beans._;

class BoundProp[T] extends PropertyChangeSupport with
PropertyChangeListener {

  var value: T = _;

  //bi-directional binding done between this and other
  def bind(other: BoundProp[T]){
    this.addPropertyChangeListener(other);
    other.addPropertyChangeListener(this);
  }

  def propertyChange(evt: PropertyChangeEvent){
    if(!(evt.getNewValue().asInstanceOf[T]).equals(value)){
      this := evt.getNewValue().asInstanceOf[T];
    }
  }

  def :=(newValue : T): BoundProp[T] = {
    val oldValue = value;
    value = newValue;
    //hard-coded property name as it is the only property in class
    firePropertyChange("value", oldValue, newValue);
    this;
  }
}
```

Implementation of `BoundJTextField` class:

```
package boundUtilities;

import java.beans._;
import javax.swing.JTextField;
import java.awt.event._;

class BoundJTextField(textField: JTextField){
  var text = bind();

  //bi-directionally binds the textField's text to local variable text
  private def bind(): BoundProp[String] = {

    var temp = new BoundProp[String];

    //update text whenever the textField changes
    textField.addActionListener(new ActionListener{
      def actionPerformed(e:ActionEvent): Unit = {
        temp := textField.getText();
      }
    });

    //update textField when the text changes
    temp.addPropertyChangeListener(new PropertyChangeListener{
      def propertyChange(e: PropertyChangeEvent){
        //check if text has already been updated( to avoid infinite
        //loop caused by bi-directional binding )
        if(textField.getText() != e.getNewValue().asInstanceOf[String]){
          textField.setText(e.getNewValue().asInstanceOf[String]);
        }
      }
    });
    return temp;
  }
}

object BoundJTextField {
  implicit def JTextField2BoundJTextField(textField: JTextField) =
  new BoundJTextField(textField)
}
```

Code for the sample test case:

```
package test;
import boundUtilities.BoundProp;

class Person {
    var name = new BoundProp[String];
}

package test;

import javax.swing._;
import java.awt.FlowLayout;
import boundUtilities.BoundJTextField._;

object BindTester {
    def main(args: Array[String]) : Unit = {

        /*binding of JTextFields*/
        val frame = new JFrame("BoundTextFieldDemo");
        val container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        val p = new Person;
        val field1 = new JTextField;
        field1.setColumns(20);
        val field2 = new JTextField;
        field2.setColumns(20);

        container.add(field1);
        container.add(field2);

        //this should do the bi-directional binding between textfields
        field1.text bind p.name;
        field2.text bind p.name;

        //update value of p.name should also update value of all
        //properties bound to it
        p.name := "Curly";

        frame.setSize(325, 100);
        frame.setVisible(true);

    }
}
```

## 2. Groovy Code:

Implementation of **BoundProp** class

```
package bComponents

import java.beans.*;

class BoundProp extends PropertyChangeSupport implements
PropertyChangeListener {

    Object value;
    def BoundProp(Object source){
        super(source);
    }

    //bi-directional binding done between this object and other
    def bind(BoundProp other){
        this.addPropertyChangeListener(other);
        other.addPropertyChangeListener(this);
    }

    void propertyChange(PropertyChangeEvent evt){
        if(!evt.getNewValue().equals(value)){
            this.set(evt.getNewValue())
        }
    }

    BoundProp set(Object newValue){
        def oldValue = value;
        value = newValue;
        //hard-coded property name as it is the only property in class
        firePropertyChange("value", oldValue, newValue);
        this;
    }
}
```

Implementation of `BoundJTextField` class

```

package bComponents

import javax.swing.*;
import java.awt.*;

class BoundJTextField {
    JTextField component;
    String propertyName;
    String text;

    public BoundJTextField(JTextField c){
        component = c;
    }

    BoundJTextField getText(){
        //saved for future if more than one property to be bound in
        //JTextField
        propertyName = "text";
        this;
    }

    void bind(BoundJTextField bfield){
        //adding an action listener to each of the JTextFields.
        component.actionPerformed =
            { e -> bfield.component.setText(component.getText());

        bfield.component.actionPerformed =
            { e -> component.setText(bfield.component.getText());
    }

    void bind(BoundProp bproperty){
        component.actionPerformed =
            { e ->
                if(!component.getText().equals(bproperty.value)){
                    bproperty.set(component.getText());
                }
            };

        bproperty.propertyChange =
            { e ->
                if(!e.getNewValue().equals(component.getText())){
                    component.setText(e.getNewValue());
                }
            };
    }
}

```

Implementation of `JTextFieldMetaClass`

```
package groovy.runtime.metaclass.javafx.swing;

import bComponents.BoundJTextField;
import groovy.lang.*;

//replaces the getProperty method of JTextField with the given closure
class JTextFieldMetaClass extends DelegatingMetaClass{
    JTextFieldMetaClass(MetaClass metaClass)
    {
        super(metaClass);
    }

    public Object getProperty(Object object, String propName){
        if(propName == "property"){
            BoundJTextField bField = new BoundJTextField(object);
            return bField;
        }
        //for all other properties call the regular getProperty for the
        //JTextField instance.
        else{
            return super.getProperty(object, propName);
        }
    }
}
```

Code for the sample test case:

```
package test;

import javax.swing.*;
import java.awt.*;

import bComponents.BoundProp;

class BindTester {
    static void main(args) {
        JFrame frame = new JFrame("Binding by Category Demo");
        Container container = frame.getContentPane();
        container.setLayout(new GridLayout(2,1));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField field1 = new JTextField("Default1");
        field1.setColumns(20);
        JTextField field2 = new JTextField("Default2");
        field2.setColumns(20);

        container.add(field1);
        container.add(field2);

        //JTextField.property returns a BoundJTextField object
        field1.property.text.bind field2.property.text;

        //Issue: ActionListener not getting invoked on setText. It does
        //get invoked on UI input
        field1.setText("Curly");

        frame.setSize(325, 100);
        frame.setVisible(true);
    }
}

class Book {
    BoundProp title = new BoundProp(this);
}
```