BABEŞ-BOLYAI UNIVERSITY CLUJ–NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Masters Dissertation

Reactive programming in Scala for implementing interactive applications and easing functional testing of Swing applications

Abstract

Interactive applications are in an increasing demand, yet their development is a difficult and error-prone in any programming language. We present a file manager application developed using *Scala.React*, an interactive programming library based on delimited continuations in Scala representing a viable and practical alternative to existing solutions. The application was entirely developed in Scala, the object-oriented and functional programming language running on the JVM.

We also present FEST-Logging, an extension for the functional GUI testing system FEST-Swing that tests Swing based desktop Java applications or applets. FEST-Logging logs execution details of test suites, helping development, troubleshooting or optimization efforts, easily integrating into existing test systems requiring few code changes. We demonstrate FEST-Logging through the file manager application.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

JULY 2014 ZSOLT DONCA

ADVISOR:
ASSOC. PROF. DR. LEHEL CSATÓ

Babeş-Bolyai University Cluj–Napoca
Faculty of Mathematics and Informatics
Specialization: Computer Science

Masters Dissertation

Reactive programming in Scala for implementing interactive applications and easing functional testing of Swing applications

Scientific supervisor:
Assoc. prof. dr. Lehel Csató

MSc student:
Zsolt Donca

July 2014

UNIVERSITATEA BABEŞ-BOLYAI, CLUJ–NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de disertaţie

Programare reactivă în Scala pentru a implementa aplicaţii interactive şi facilitarea testării funcţionale a aplicaţiilor Swing

CONDUCĂTOR ŞTIINŢIFIC:
CONF. DR, LEHEL CSATÓ

ABSOLVENT:
ZSOLT DONCA

IULIE 2014

Contents

1. Chapter

Introduction

Interactive applications are in a continuously increasing demand, driven by the expansion of computing and mobile devices. A reactive application needs to be "readily responsive to a stimulus", thus needs to:

1. **react to events** — the application needs to have an *event-driven* design;
2. **react to load** — the application needs to scale well as the amount of interaction and data increases;
3. **react to failure** — the application must be resilient to error and needs to recover at all levels;
4. **react to users** — the application must be responsive to the user, regardless of load.

There is a growing trend in the programming community over the importance of reactive programming. [23]

Development of interactive applications is difficult engineering task: the continuous user input and output, the high number of possible internal states makes it difficult to achieve the above desired properties. The event-driven nature of the applications tend to have a negative effect its complexity: the state machines and the inversion of control that is needed to deal with events increase the application's complexity, resulting in code that is possibly difficult to maintain, develop, and is error-prone. Also, the control flow of the application is interleaved with reactions to events, leading to interactions that can be hard to foresee.

The Scala library *Scala.React*, developed for the paper [14] offers a solution to the above problem. Its goal is to introduce multiple reactive programming abstractions enabling an event-driven design using declarative implementations, and together with a high-order data-flow DSL[1] embedded into *Scala* makes it possible to implement complex multi-event logic in a direct imperative style, without the need for inversion of control or to implement state machines.

1.1 Scala Commander

We implement a simple desktop application with the goal of demonstrating the capabilities of *Scala.React* in a practical example. The application is *Scala Commander*, a file manager application with a user interface that constantly reacts to user interaction. The reactive abstractions in *Scala.React* enabled

1. Domain-Specific Language

the design of the application to consist of simple “building blocks” called *signals* to be wired together representing the internal state of the application. The relationship between signals is declarative, and there is no development effort needed for change propagation or to maintain consistency. The logic of the application is implemented using *reactors*, with a data-flow DSL that lets us write logic dealing with multiple events in a direct pseudo-code-like style, significantly reducing the complexity of the code.

1.2 FEST-Logging

Testing of applications with user interfaces is also important. The library *FEST-Swing* offers an easy and intuitive way to write tests for *Swing* user interfaces that are compact, easy to write and read like a specification. However, dealing with large number of tests can be problematic: shared state between individual tests can affect the stability of test suites.

We also implement *FEST-Logging*, the library extending *FEST-Swing* with the goal of easing the testing of Swing user interfaces. FEST-Logging is an agent running along test suites, gathering information while test run providing insight into what happens in the test suites, helping maintenance and troubleshooting of failed tests. FEST-Logging gathers information on the tests such as total duration and what actions did the test execute (along with the actual *arguments*). It also takes screenshots of failed tests, and optionally can create a “movie” of the tests by taking screenshot after each test action.

To ease development of tests, *FEST-Logging* is integrated with *Cacio-tta*, a graphics stack that enables the tests to run in background. There is also a simple annotation-based feature that provides automatic delegation of UI-related operations to the Swing EDT thread.

1.3 The structure of this thesis

In chapter 2, we present the theoretical background necessary to understand the applications *Scala Commander* and *FEST-Logging*:

- In section 2.1 we give a generic overview of Scala, the object-functional programming language. Our goal is to give a quick overview of the language constructs in Scala, and to present the most important features that the concepts in *Scala.React* or in *Scala Commander* rely on.
- In section 2.2 we present *Scala-Swing*[13], a Scala library built on *Java Swing*, the primary Java GUI widget toolkit. [19] Scala Commander uses Scala-Swing to implement the user interface.
- In section 2.3, we present the *Scala Continuations plugin*, a compiler plugin providing delimited continuations using a continuation passing style transformation. These delimited continuations make it possible for the Scala.React data-flow DSL to suspend and resume the control flow in the reactors.
- In section 2.4, we present the library *Scala.React*, its main concepts and some implementation details.

- In section 2.5, we present the library *FEST-Swing* that provides functional testing for Swing user interfaces.
- In section 2.6, we present the Model-View-Controller design pattern that Scala Commander is based on.
- In section 2.7, we present a design pattern commonly associated with testing of user interfaces.

In the next two chapters, we present the two main applications developed for this paper:

- In chapter 3, we present *Scala Commander*, the file manager application demonstrating the capabilities of *Scala.React*.
- In chapter 4, we present *FEST-Logging*, the library extending *FEST-Swing* with the goal of easing the testing of Swing user interfaces.

Finally, in appendices A and B we present some interesting Scala delimited continuation examples and Scala.React examples, respectively.

2. Chapter

Theoretical background

2.1 Scala

Scala is a general-purpose object-oriented and functional programming language that runs on the JVM. Besides supporting all the standard OOP concepts (classes, inheritance, encapsulation etc.), Scala has full support for functional programming (including currying, pattern matching, algebraic data types, tail recursion, lazy initialization, immutability etc.) Scala has a unified type system (there is no distinction between primitives and classes), functions are“first-class citizens”, supports anonymous types, operator overloading, optional parameters, named parameters, string interpolation, and so on. All these put together, Scala enables a programmer to develop very concise and small programs.

Scala and Java have commons roots. Scala's main designer and developer, Martin Odersky, is also known for developing the programming language *Generic Java* (which, with the addition of wildcards, was integrated into *Java 5*), and for the development of the current generation of *javac*, the Java Compiler. Many of Scala's syntax elements were inspired by Java (e.g. the imperative control structures, operators). Many of Scala's design decisions were inspired by criticism over the shortcomings of Java. [17]

The Scala source code is compiled to Java bytecode, so the resulting executable runs on the Java virtual machine. Existing Java libraries can be directly used in Scala code, and vice versa.

2.1.1 General syntax

Scala uses the curly-brace syntax reminiscent of the C programming language. A simple hello-world program looks like the following:

```
object HelloWorldApp extends App {
  println("Hello, world!")
}
```

Note that contrary to a hello-world application in Java, there is no class declaration and no static main method declaration; instead, a singleton object is created with the keyword `object` whose body consists of the application. [1].

A local variable or a field is declared with the keyword `val` or `var`. The keyword `val` is used for variables that are initialized only once and never change their value, and the keyword `var` is used for

1. In order to be compatible with the JVM, the method `static void main(String[])` is generated for the class

variables whose value changes during the execution. For example:

```
val end = 100 // constant
var sum = 0 // variable, changed in the loop below
for (i <- 1 to end) {
  sum += i
}
println(s"The sum of numbers between 1 and $end is $sum")
```

Of course, the above example can be written in a much more concise way in Scala, while avoiding looping and mutable state, using functional-style programming:

```
val sum = (0 to 100).sum
```

This works because the method `sum` is defined on *numeric* sequences.
The following example defines a method that calculates the below expression $\frac{1}{x^2}$ for $x \neq 0$ and otherwise returns 0:

```
def f(x: Double): Double = {
  if (x != 0) {
    val square: Double = x*x
    return 1 / square
  } else {
    return 0
  }
}
```

Note that the type *follows* a name (as in Pascal) rather than *preceeds* a name (as in C or Java). This, together with the mandatory initialization of variables, enables *type inference* to take place, making type declarations optional for variable declarations, fields, and even (non-abstract) methods. In the above example, all the type declarations, with the exception of the parameter `x`, can be ommited. The following code is equivalent:

```
def f(x: Double) = {
  if (x != 0) {
    val square = x*x
    1 / square
  } else 0
}
```

It is also possible to omit the return statement. This is because in Scala there is no distinction between expressions and statement (such as `if`, `while`, `try ...catch` and so on). In Scala, everything is an expression and thus has a value. The value of a block (delimited by the curly braces), is equal to the value of the *last* expression inside the block, and value of the statement `if (condition) <then-block> else <else-block>` statement is equal to either the value of the "then-block"

block or the “else-block” block, depending on the condition. In Scala code, return statements are typically used only to control the execution flow of a method.

2.1.2 Type system

Scala uses a unified type system that includes value types and reference types (see figure 2.1). Scala defines the following types:

- `scala.Any`: any object implicitly extends `Any`, including the primitive types (int, long etc.) and reference types (classes);
- `scala.AnyVal`: base class for value type, including the standard primitive types and `Unit`, Scala’s equivalent of the Java type `void`;
- `scala.AnyRef`: base class of reference types, the equivalent of `java.lang.Object`;
- `scala.ScalaObject`: any class defined in Scala code implicitly extends this trait;

There are two special-purpose types at the button of the hierarchy:

- `scala.Null`: this type implicitly extends *any reference type*, and there is only one instance of it, the `null` reference. Note that `Null` is not a subtype of the value types; for instance, it’s not possible to assign `null` to a variable of type `scala.Int`.
- `scala.Nothing`: this type implicitly extends *any type*, and *no instances* of this type exist. This type is typically used for two purposes: in covariant types (e.g. `Nil` is instance of `List[Nothing]`, which is a subtype of `List[T]` for any `[T]`), or as a return type of methods that never return normally.

Note that there are also conversions defined between primitive types, as figure 2.1 shows using dashed arrows.

Classes

A class is declared in the following way:

```
class C[typeParams](classParams) extends Superclass with Trait {
  definitions
}
```

With the exception of the class name, everything can be omitted. A class may extend (or *mix in*) multiple traits (similarly to interfaces and abstract classes in Java). The class parameters are Scala’s equivalent of Java’s constructor parameters.

A class can be instantiated with the `new` operator, similar to Java.

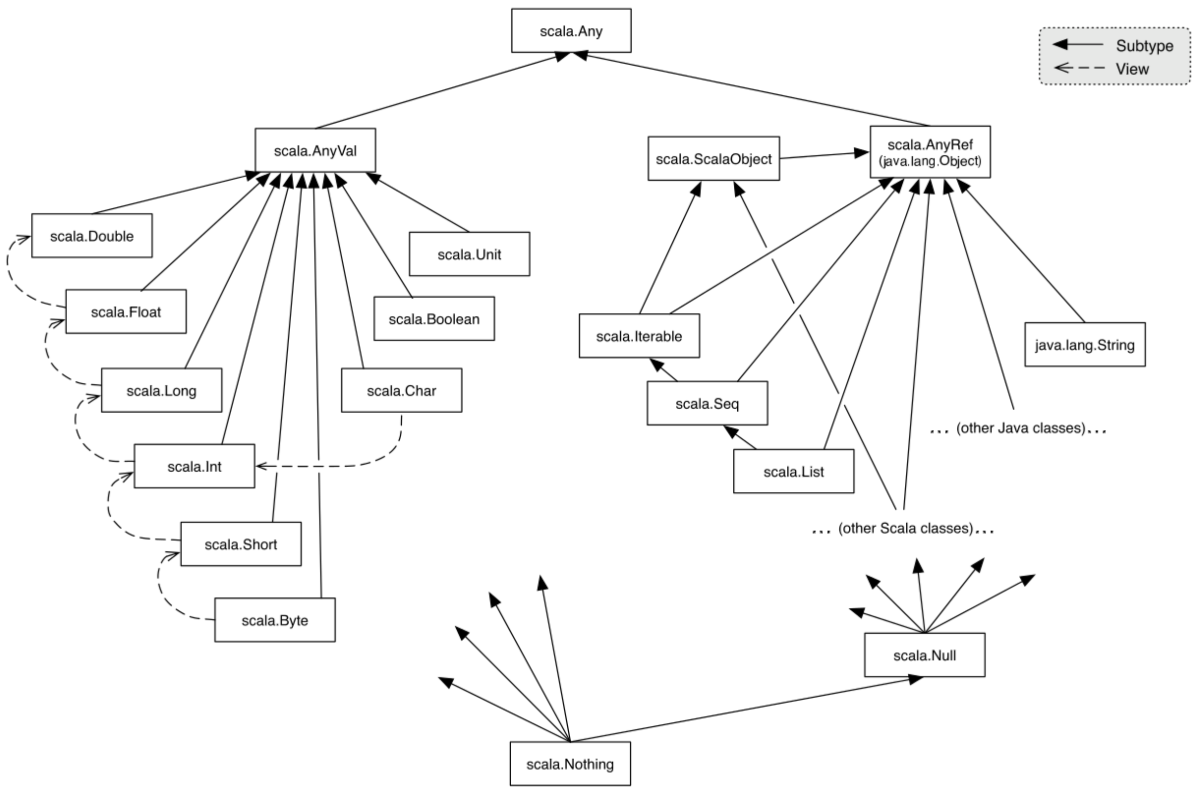


Figure 2.1: Scala-Commander.

Singleton objects and companion objects

With the use of the `object` keyword, it is possible to declare a singleton object:

```
object O extends Superclass with Trait {
  definitions
}
```

Package-level singleton objects can define fields and methods that are accessible throughout the entire application, and thus are similar to Java's static fields and methods. In addition to that, a singleton object can have a base class or can extend traits, which is not directly possible in Java.

A *companion object* is a singleton object that has the same name as a class defined in the same scope. For example:

```
class Complex(val r: Double, val i: Double) {
  def +(c: Complex): Complex = \dots
}
object Complex {
  val zero = new Complex(0, 0)
  val one = new Complex(1, 0)
  val i = new Complex(0, 1)
  def apply(r: Double = 0, i: Double = 0) = new Complex(r, i)
}

val c1 = Complex(2) // calls Complex.apply(2)
val c2 = Complex(2, -1) // calls Complex.apply(2, 3)
val c3 = Complex() // calls Complex.apply(0, 0)
val c4 = Complex(i = 1) // calls Complex.apply(0, 1)
val c5 = new Complex(2, 3)
val c6 = Complex.one + Complex.i
```

This is in direct parallel with a class in Java having both static and instance members: the members defined in the companion object correspond to the the static members, and the members defined in the class correspond to the intance members. Note that this is also actually how classes with companion objects are implemented in Scala (the compiler generates only one class with static and instance members).

The above example uses Scala's equivalent of overloading the method call operator: the expression `a(x)` is equivalent to `a.apply(x)` where `a` is an object of a type that defines the method `apply`.

2.1.3 Traits

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method definitions (abstract and concrete) and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits. Mixing traits with fields and concrete methods is similar to multiple inheritance present in other languages. Traits are similar to abstract classes, but traits cannot have parameters (constructors).

Let us consider the following example:

```
trait Named {
  def name: String
}
```

This defines the abstract method `name`. The method is implicitly abstract (without having to use the keyword `abstract`) as no method body follows the declaration. This trait can now be mixed in into a class:

```
class Cat(nickname: String) extends Named {
  override def name: String = nickname
}
```

Just as with ordinary inheritance, any `Cat` instance can be treated as an instance of the trait `Named`:

```
def printName(o: Named) { println("name is: " + o.name) }
...
val myCat = new Cat("Bobby")
```

```
printName(bobby)
```

A class can mix in any number of traits, and can even have a subclass:

```
class Animal { ... }
trait FourLegged { ... }
trait Mammal { ... }

class Cat(nickname: String) extends Animal with FourLegged with Mammal
    with Named { ... }
```

Thin versus rich interface

Traits are Scala's replacement for Java interfaces. In Java versions prior to 8, interfaces are highly restricted, able to contain only abstract method declarations. This leads to difficulties with convenience methods (the same methods must be reimplemented in each implementing class), and to the inability to add methods to an interface without breaking compatibility (issue for published libraries).

One major use of traits is to automatically add methods to a class in terms of methods the class already has. This way, traits can enrich a *thin* interface making it into a *rich* interface.

Thinness versus richness is a common trade-off in object-oriented design. The trade-off is between the implementers and the clients of an interface. The richer an interface is, the more convenient it is for a client to pick a method that exactly suits his needs. A thin interface, on the other hand, contains less methods and is easier to implement, however the clients might need to write extra code to use it.

Scala traits makes it possible to have interfaces without this trade-off: it is possible to have interfaces with few abstract methods that *need* to be defined by an implementation — the methods of a thin interface — and at the same time to have concrete methods that rely on those abstract methods — the extra methods of a rich interface. Implementations can choose to override any of the methods implemented in the trait (e.g. for performance reasons).

Consider the following example:

```
trait TwoNames extends Named {
  def firstName: String
  def lastName: String

  def fullName: String = s"$firstName $lastName"
  def lexicalFullName: String = s"$lastName, $firstName"

  def name = fullName // implements from Named
}
```

Here, the trait `TwoNames` extends the trait `Named`, implementing the method `name` with the full name, while declaring the methods `firstName` and `lastName` abstract. It also defines the convenience method `lexicalFullName`. To implement a trait like this in Java prior to Java 8, one could use an abstract class in Java, eliminating the possibility of using other base classes in extending classes (maybe of similar design). Another possibility would be to make `TwoNames` an interface, to the move the implementation of the methods `fullName`, `lexicalFullName` and `name` to a utility class, and to reference that utility class in all classes extending the interface. While this workaround is technically possible, it is cumbersome and rarely preferred.

Linearization

Traits are a way to inherit from multiple class-like constructs, but they differ in important ways from the multiple inheritance present in many languages. One difference is especially important: the interpretation of *super*. With multiple inheritance, the method called by a super call can be determined right where the call appears. With traits, the method called is determined by a linearization of the classes and traits that are mixed into a class, enabling a kind of stacked modifications similar to the *decorator* pattern.

Consider the following example:

```
class Animal {
  def description: String = "animal"
}
trait FourLegged extends Animal {
  override def description: String = "four-legged " + super.description
}
trait Furry extends Animal {
  override def description: String = "furry " + super.description
}
trait Carnivorous extends Animal {
  override def description: String = "carnivorous " + super.description
}
trait Feline extends FourLegged with Furry with Carnivorous {
  override def description: String = "feline, that is, a " +
      super.description
}
trait Domesticated extends Animal {
  override def description: String = "domesticated " + super.description
}
class Cat extends Feline with Domesticated {
  override def description: String = "I am a cat, a " + super.description
}
println(new Cat().description)
// prints: I am a cat, a domesticated feline, that is, a carnivorous furry
    four-legged animal
```

The linearization goes from right to left, and from the bottom to the top of the hierarchy:

`Cat` → `Domesticated` → `Feline` → `Carnivorous` → `Furry` → `FourLegged` → `Animal`

In Scala, classes and traits are always are always inherited only once, meaning that there won't be four different instances of the class `Animal` in a `Cat` instance [2]. Practically, the `extends Animal` part in the above traits make the class `Animal` a requirement of the traits, and the class `Cat` automatically extends the class `Animal` through the traits. Mixing in the trait `Feline` to a class already having a different base class results in compilation error:

```
class Plant
class PottedPlant extends Plant with Feline
// Error: illegal inheritance; superclass Plant is not a subclass of the
   superclass Animal of the mixin trait Feline
```

2.1.4 Syntactic flexibility

Scala has significantly more flexible syntax when compared to Java. This includes:

- Semicolons are unnecessary for statements that are otherwise delimited by new lines. Semicolons are used only when more than one statement is present on a line.
- Any method can be used as an infix operator. For example, `"%d apples".format(count)` is equivalent to `"% apples" format count`. This is in fact how operator overloading is possible: since method names can consist of any characters, it is possible to define a method with a symbol in its name, and then use the method as an infix operator. For example:

  ```
  class Complex(val r: Double, val i: Double) {
    def +(c: Complex) = new Complex(r + c.r, i + c.i)
    def multiplyBy(c: Complex) = \dots
  }
  val a = new Complex(1, 2)
  val b = new Complex(3, -1)
  val sum1 = a + b // sum1 and sum2 are equivalent
  val sum2 = a.+(b)
  val prod1 = a multiplyBy b // prod1 and prod2 are equivalent
  val prod2 = a.multiplyBy(b)
  ```

- Possible to overload the function call operator — methods `apply` and `update` have shorts forms similar to function calls. For example, where `a` is a class instance or singleton object, `a()` is equivalent to `a.apply()`. The method can parameters as well, e.g. `a(42)` is equivalent to

2. contrary to the (non-virtual) multiple inheritance of C++, one of the most widely used languages supporting multiple inheritance

`a.apply(42)`. The method `update` has an equivalent assignment-like syntax: `a() = 42` is equivalent to `a.update(42)`, and `a(4) = 2` is equivalent to `a.update(4, 2)`.

– Scala supports the Uniform Access Principle: it is transparent to the user of a class whether a property of an object is implemented by directly using a field or by using getter-setter methods. In Scala, public `val` or `var` members can be overridden by getter-setter methods and vice-versa. This is in contrary to Java, where explicit getter-setter methods are required and the public fields exposed by a class cannot be overridden by getter-setter methods in subclasses. For example:

```
trait Person {
  var name: String // abstract
}

class Student extends Person {
  private var _name: String
  def name = _name
  def name_=(newName: String) { _name = newName }
}

val p: Person = new Student
p.name = "Foo Bar"
println(p.name)
```

– Scala supports the definition of constructs that are similar to language structures. For example:

```
def safe(f: => Unit) {
  try {
    f()
  } catch {
    case e: Throwable => e.printStackTrace()
  }
}

safe {
  println("This block never throws exceptions")
}
```

In the above example, the method call parenthesis were omitted, and the block was automatically converted to an argument of type `=> Unit` which represents nullary functions returning Unit (equivalent of the Java type `void`).

This syntactic flexibility allows domain-specific languages to be defined in Scala without the need to extend the compiler. For example, the library *Scala-react* defines a custom control-flow language with the use of higher-order functions, by using extensively using `apply` and `update` methods, and by using infix method call syntax for operator methods.

2.1.5 Tail recursion

Tail call optimization is an optimization technique in which method invocation instructions are replaced with jump instructions when the method invocation is immediately followed by a return from the method, effectively meaning that the method call is the last expression in the method body. The purpose if this implementation is to avoid growing the stack unnecessarily, allowing long call chains that would otherwise run out of stack space. A simple example:

```
def a(x: Int): Int = {
  val y = b*4
  b(y) // calling b is the returned expression
}
def b(x: Int): Int = { x + 2 }
```

A special case of tail call optimization is *tail recursion*, where the method being called might lead to invoking the same method again: it's either a direct call to the very same method (direct tail recursion), or a call to an other method which will eventually result in the current method being called again (mutual tail recursion). Tail recursion plays an important part of functional programming as it allows the using of recursive method calls instead of loops.

Scala only offers direct tail recursion, which is realized in compiler-time. The reason is that the JVM does not support mutual tail recursion or tail call optimization - a feat that needs runtime support.[3]

```
def factorial(n: Int): Int = {
  @tailrec def factorialAcc(acc: Int, n: Int): Int = {
    if (n <= 1) acc
    else factorialAcc(n * acc, n - 1)
  }
  factorialAcc(1, n)
}
```

Figure 2.2: Calculating factorial using tail recursion. Notice the annotation `@tailrec`.

The Scala compiler will automatically optimize any truly tail-recursive method. If you annotate a method that you believe is tail-recursive with the `@tailrec` annotation, then the compiler will fail to compile you if the method is actually not tail-recursive. This makes the `@tailrec` annotation a good idea, both to ensure that a method is currently optimizable and that it remains optimizable as it is modified.

Note that Scala does not consider a method to be tail-recursive if it can be overridden. Thus the method must either be private, final, on an object (as opposed to a class or trait), or inside another method to be optimized.

3. Note that this is particular to the JVM only — the language F# running on top of the .NET CLR *does* have tail call optimization

2.1.6 Functions as types

In Scala, functions are first-class citizens. For example, one can declare a variable that references a function:

```
val plusOne: (Int => Int) = x => x + 1
println(plusOne(2)) // prints 3
```

In this example, the type of the variable `plusOne` is `Int => Int`, which means that it's a function taking an integer and returns an integer. The form `Int => Int` is practically a syntactic sugar for the generic type `Function1[Int, Int]`.

Partial functions

A partial function in Scala is a function that is only defined for a subset of its input set. For example:

```
val fibo: PartialFunction[Int, String] = {
  case 1 => "one"
  case 2 => "two"
  case 3 => "three"
  case 5 => "five"
  case 8 => "eight"
}

println(fibo.isDefinedAt(2) + " " + fibo(2)) // prints: true two
println(fibo.isDefinedAt(4)) // prints "false"

// prints "one two three five eight"
(0 to 10).collect(fibo).foreach(x => print(x + " "))
```

The function `fibonacci` is a partial function taking an `Int` value and returning `String`, as is defined only for the first five Fibonacci values. A partial function has two important methods:

- `isDefinedAt(x)`: returns true if the partial function is defined for `x`, and returns false otherwise;
- `apply(x)`: return the value of the function for the given `x` if the function is defined for `x`, and throws an exception otherwise.

Partial functions in Scala are used for pattern matching, or as a programmable *switch-case*-like structure. For example, the event handling mechanism of *Scala-Swing* uses partial functions (see section 2.2.1).

2.1.7 Currying

Currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application).

Consider the following function:

```
def sum(a: Int)(b: Int): Int = { a + b }
```

This is a curried function, which takes an integer `a` and returns a function that takes an integer `b` and returns the value `a + b`. This is just syntactic sugar for the form:

```
def sum(a: Int): (Int => Int) = b => a + b
```

Both forms can be used in the same way:

```
println(sum(1)(2)) // prints 3

val plusOne: (Int => Int) = sum(1)
println(plusOne(2)) // prints 3
```

Typical use cases of currying in Scala include helping type inference, providing fluent API, and performance improvements in multi-stage computation.

For type inference

In Scala, type inference of expressions on a single line always works form left to right: the leftmost expression is evaluated first, then followed by the next expression to the right. Chained function applications of curried functions are considered separate expression, and therefore type inference on a proceeding parameter can affect the type inference of a latter parameter. Consider the library function `foldLeft` defined on sequences of base type `A`:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

This allows the following code:

```
val s = List("Joe", "Jack").foldLeft(0)((len, name) => len + name.length)
println(s) // prints 7
```

If `foldLeft` was defined as a simple non-curried function such as the following:

```
def foldLeft[B](z: B, op: (B, A) => B): B
```

The one would have to provide more explicit types:

```
List("Joe", "Jack").foldLeft(0, (len: Int, name: String) => len +
    name.length)
List("Joe", "Jack").foldLeft[Int](0, (len, name) => len + name.length)
```

For fluent API

Currying (together with other language features) allows to create an API that looks like a language construct:

```
def loop[A](n: Int)(body: => A): Unit = (0 until n) foreach (n => body)

loop(2) {
  println("hello!")
}
```

Multi-stage computation

Consider a problem where we want to decide whether given point is inside a given circle, identified by its center point and radius:

```
class Point(val x: Double, val y: Double)
class Circle(val center: Point, val radius: Double)

def isInside(c: Circle, p: Point): Boolean = {
  import Math.pow
  val radiusSquare = pow(c.radius, 2)
  val distSquare = pow(p.x - c.center.x, 2) + pow(p.y - c.center.y, 2)
  distSquare <= radiusSquare
}

val myCircle = new Circle(new Point(1, 2), 2)
val points = Seq(new Point(2, 3), new Point(4, 2), new Point(1, 1))
val inside = points.map(p => isInside(myCircle, p))
println(inside) // List(true, false, true)
```

In the above example when we calculate whether a *series* of points are inside the *same* circle, we are calculating the value `radiusSquare` over and over again for each point, even though calculating it only once and then using it for all the points would suffice. Currying offers a simple and elegant solution:

```
def isInside(c: Circle): (Point => Boolean) = {
  import Math.pow
  val radiusSquare = pow(c.radius, 2)

  (p: Point) => {
    val distSquare = pow(p.x - c.center.x, 2) + pow(p.y - c.center.y, 2)
    radiusSquare >= distSquare
  }
}
```

```
val myCircle = new Circle(new Point(1, 2), 2)
val points = Seq(new Point(2, 3), new Point(4, 2), new Point(1, 1))
val isInsideMyCircle = isInside(myCircle)
val inside = points.map(p => isInsideMyCircle(p))
println(inside) // List(true, false, true)
```

Now the function `isInsideMyCircle` already has the value `radiusSquare` pre-calculated. Note that in this case using the simple multi-parameter list version of currying does not offer the same performance benefits, because Scala cannot automatically transform the function body to extract computation that is dependent only on a subset of the parameters.

Partially applied functions

Partial function application and currying are related. One of the significant differences between the two is that a call to a partially applied function returns the result right away, not another function down the currying chain; this distinction can be illustrated clearly for functions whose arity is greater than two.

Let us consider the following example:

```
def sum3(a: Int, b: Int, c: Int) = a + b + c
val sum2: ((Int, Int) => Int) = sum3(1, _: Int, _: Int)
println(sum2(2, 3)) // prints 6
```

After fixing the first parameter to the value `1`, the function `sum2` is a binary function, taking two arguments. This is in contrast to what it would be if `sum3` were a curried function:

```
def sum3(a: Int)(b: Int)(c: Int) = a + b + c
val sum2: (Int => Int => Int) = sum3(1)
println(sum2(2)(3)) // prints 6
```

2.2 Scala-swing

Scala-swing is a UI library written in Scala that wraps the Java Swing components with the purpose of providing a simpler, more straightforward and typesafe interface that is more natural to use in Scala. It is meant to be sufficiently close to the Java Swing to appeal to Java programmers with existing Swing experience.[13] The library diverges significantly from Java Swing only on parts that are considered as bad design decisions of Java Swing.

The library provides lightweight wrappers around Swing components, maintaining their functionality and hierarchy, reimplementing them in Scala-like terms (using Scala properties instead of getter/setter methods, using enumerations instead of magic constants, and providing a typesafe API for objects like JList). The main divergences are the following:

– In Java Swing all components are also containers. This doesn't make sense for a number of components (e.g. TextField, CheckBox, Button etc.) This is likely to be because of the lack of multiple inheritance in Java. Scala-swing uses the *trait* `Container`.

- Layout managers and panels are coupled, though they are decoupled in Java Swing. Scala-swing offers a straightforward, typesafe API for the standard Swing layout managers in the form of a panel class for each layout (e.g. `BorderPanel` for a `JPanel` with `BorderLayout`).
- The event mechanism is completely reworked around the concept of publishers and reactors. This offers a more homogeneous API for event management than what Java Swing offers.

2.2.1 Event handling

Event handling is based around the concepts of *publishers* and *reactors*. Any `scala.swing` component is actually both a publisher that publishes events and a reactor that reacts to events. For efficiency reasons, events of a component are not published directly by the component itself but by members, grouping the events into categories (see below the member `mouse`).

As each event is a normal Scala object, it has a unique type, which determines how to pattern match on it. Contrast this to Java Swing, where some events are dispatched in two phases, according to their type plus an additional listener method. The following example shows how to listen to mouse click events using Java Swing:

```
new JComponent {
  addMouseListener(new MouseAdapter {
    @Override
    def mouseClicked(e: MouseEvent) {
      System.out.println("Mouse clicked at " + e.getPoint)
    }
  })
}
```

Note that in this case we are using the convenience class `MouseAdapter` that implements all the methods of the interface `MouseListener` with empty implementations. This interface is just one example where we first dispatch by the event's Java type (`MouseEvent`) and then we refine the match implementing the appropriate method (`mouseClicked`). The equivalent using `scala.swing` looks like the following:

```
new Component {
  listenTo(mouse.clicks)
  reactions += {
    case e: MouseClicked =>
      println("Mouse clicked at " + e.point)
  }
}
```

Here we rely on the event type (`MouseClicked`). Since the event classes form hierarchies, we can also very conveniently listen to *any* mouse event:

```
listenTo(mouse.clicks, mouse.moves, mouse.wheel)
reactions += {
  case e: MouseEvent =>
    println("Mouse event at point " + e.point)
}
```

The equivalent using Java Swing would require us to add three listeners (using `addMouseListener`, `addMouseMotionListener`, `addMouseWheelListener`) and to implement a total number of **eight listener methods**, all with the same implementation.

Note that the members `listenTo` and `reactions` are defined by the trait `Reactor`. The member `reactions` is a mutable list of *partial functions* of type type `PartialFunction[Event, Unit]`.

See figure B.2 in appendix B for a complete example of a `scala.swing` application listening to mouse events.

2.3 Scala Continuations Plugin

The concept of continuations are naturally present in programming: a conditional expression selects a continuation from two possible branches, or throwing an exception discards the rest of the continuation. Although continuations are implicitly built-in into every language, having control over them explicitly is not typical [1].

Delimited continuations are a versatile programming tool. Their most important use is based on their ability to suspend and resume sequential code paths in a controlled way. The Scala Continuations plugin is a compiler plugin that enables the programmer to have delimited continuations in direct-style code, without much syntactic overhead. This is implemented by the plugin selectively transforming the source code in compile time using a continuation passing style transformation, driven by annotations and other structures placed into the code by the programmer. Multiple variants of delimited continuations have been described in the literature, and the Scala Continuations plugin implements the so-called Danvy and Filinski model [4], with two primitive operations, `shift` and `reset` [20].

A delimited continuation embodies a certain part of the program. When invoking a delimited continuation, the control is returned to the caller, and it may also return a value. Inside the continuation, at any point the current fragment of the call stack starting from the delimiting block can be saved to be later reused (possibly multiple times), or discarded. This control over the execution of the control flow can is the essence of the delimited continuations.

Since the Scala Continuations plugin uses compile-time transformation, applications using it do not need a special VM or other specific runtime support.

2.3.1 `shift` and `reset`

There are two control operators: `shift` and `reset`. The `reset` operator is followed by a code block which is the continuation itself. The `shift` blocks inside the continuation play a dual role:

- they are in control of the rest of the continuation: the rest of the continuation is provided in the form of a function (denoted as `k` below), and the `shift` block can decide what to do with it: it may invoke it once, more than once, or never. Note that the rest of the continuation may include further `shift` blocks;
- the value returned in the `shift` block gets propagated back to the previous `shift` block (as the result of the function `k`), or, in case of the first `shift` block, back to the `reset` block itself.

The value of the `reset` block is always equal to the return result of the very first `shift` block that gets executed in the continuation, or, in case no `shift` block gets executed, equal to the body of the `reset` block (see figure 2.3).

```
val simple: Int = reset {
  val a = 40
  val b = 2
  a + b
}
println(simple) // 42
```

Figure 2.3: No `shift` block - the continuation is executed sequentially.

Figure 2.4 shows a simple `reset` block with a single `shift` block inside and also shows an equivalent manually transformed version of the same continuation. In this example, the shift block invokes the rest of the continuation, which in the $x + 4$ function, with the argument 1, modifies the result by adding 7 to it, and returns the result (12). Because this is the only `shift` block in the continuation, the returned value is the result of the `reset` block.

Answer type modification

The Scala Continuations plugin allows full answer type polymorphism in a statically type-safe manner. This allows the return type of the `reset` block (the type of the last expression in the block) and the return type of the individual `shift` blocks to be different. The transformation accurately tracks answer-type modification throughout the continuation and ensures, in compile time, that no type errors can occur in runtime (u.i. if the continuation can possibly run through path of conflicting answer types, the compilation fails with error.)

Figure 2.5 shows a simple an example where answer-type modification takes place. The type of the entire continuation is `String`, the answer type of the `reset` block is `Int`, and the answer type is modified in the `shift` block from `Int` to `String`. Note that even though the `shift` block's body returns a `String` (which the final answer type of the continuation), the actual type of the `shift` *expression* is a third type `Char`, which corresponds to the input type of the function `k`.

```
val result = reset {
  val s = shift {
    k: (Int => Int) =>
      k(1) + 7
  }
  s + 4
}
println(result) // prints 12
```

```
val reset = {
  val shift: ((Int => Int) => Int) = {
    k: (Int => Int) =>
      k(1) + 7
  }
  val rest: (Int => Int) = {
    s: Int =>
      s + 4
  }
  shift(rest)
}
println(reset) // prints 12
```

Figure 2.4: Left is a delimited continuation, and right is the equivalent manually-transformed continuation passing style versions.

```
def charToHexString(char: Char): String = {
  reset {
    val c: Char = shift {
      k: (Char => Int) =>
        val num: Int = k(char)
        "0x%02x".format(num) // Int to String (hex code)
    }
    c.toInt // converts Char to Int
  }
}
println(charToHexString('A'))
```

Figure 2.5: An example showing answer type modification: the continuation's type is `String` even though the `reset` block's body returns `Int`.

Dynamic control flow

A delimited continuation does not has to consist of a single block that the compiler can understand entirely. It is possible to have `shift` blocks defined in methods or functions, and thus the `shift` blocks to be associated only in runtime with a certain `reset` block. Since the final return type of the `reset` block is not necessarily reflected in a `shift` block at all (because of answer type polymorphism), the code containing such "detached" `shift` blocks **must** be annotated with the annotation `@cpsParam[B, C]` where `B` is the type of computation state after computation has executed, and before control is returned to the shift, and `C` is the eventual return type of this delimited computation.

Because the compiler needs to transform continuation passing style (CPS) code, non-CPS code cannot (neither directly nor indirectly) call CPS-code. This can lead to awkward limitations, such as the inability to iterate through a sequence with a simple for-loop (the reason is that a for-loop is desugareted by the compiler to a `foreach` method call on the collection with the loop as a closure body being the argument). Code that needs to interact with CPS-code thus needs to be marked to CPS-code itself (by the annotation @cpsParam), but then such code can no longer be used by non-CPS code (even if it doesn't contain `shift` blocks). A reason is that a method returning the type `A` annotated with `@cpsParam[B,C]` is effectively transformed by the compiler to return the internal type `ControlContext[A,B,C]`.

Appendix A Scala CPS plugin examples contains examples demonstrating the annotation @cpsParam.

2.3.2 Use cases

The paper [20] describes some academic use cases (Type-Safe printf, Direct-Style Monads, Concurrency Primitives, Actors, Functional Reactive Programming, Asynchronous IO). At the time of writing, the following widely available Scala libraries using delimited continuations were available:

- Akka Dataflow Concurrency — Akka implements Oz-style dataflow concurrency by using a special API for *Futures* that enables a complementary way of writing synchronous-looking code that in reality is asynchronous. [22]
- Scala.React — A reactive programming library for Scala developed based on the paper [14].
- Swarm — a distributed computing framework allowing the creation of web applications which can scale transparently through a novel portable continuation-based approach. [2]

There is an implementation of *nondeterministic evaluation* in Scala based on delimited continuations [16].

Generators were also implemented in Scala using the delimited continuations [15]. Generators are a form of special *iterables* where the series of elements are produced by a function one at a time, at the same time while a caller iterates over them. The delimited continuations in Scala allows such generators to be implemented directly in the form of an algorithm with *yield* statements (similarly to the *yield return* statement in C#), instead of having to encode the algorithm in the form of a *state machine* (which is required in e.g. Java).

2.4 Scala.React

Programming systems with interactive user interfaces requires a considerable amount of engineering to deal with continuous user input and output, yet the predominant programming models dealing with continuous state change in applications is the observer pattern [8]. The Scala library *Scala.React* offers a reactive data-flow programming model, and by that it solves numerous engineering problems observer pattern-based design faces. Amongst multiple Scala features, this is enabled by the higher order concepts of events and state, enabled by Scala's higher-order functions, and by a data-flow DSL embedded into Scala, enabled by continuation passing style transformation that *Scala* offers. Its design is also heavily based on Scala's *traits*, simplifying the binding of observer-based interfaces to the reactive abstractions.[14]

Object-oriented reactive applications traditionally implement reactivity by using the Observer pattern [14][21]. The observer pattern relies on the concept of *inversion of control* to decouple the observers from the subjects (observables), reversing the intuitive flow of the application and making the code harder to understand, analyze and maintain. Typical implementations often require a lot of boilerplate code, further complicating the code of the program. Separation of concern is often hard to achieve because the observer logic typically needs to be mixed with the application's business logic. It is difficult to maintain data consistency because the observer pattern does not incorporate the concept of transactionality or atomicity when notifying observers of changes. Also, the observer pattern enforces mutability and relying on side-effects, enforcing exclusively imperative an programming style. The paper [14] argues how Scala.React resolve these and many other problems with the observer pattern.

2.4.1 Event streams

The base of simplifying the event logic is a general interface, offering uniformity, abstraction and reusability. The class `EventSource[A]` represents a generic source of events of type `A`. It is possible to schedule an event source to emit a value at any time. Let us consider an example of an event source of type integer emitting two events:

```
val es = new EventSource[Int]
es emit 1; es emit 2
```

We can print all events from the event source to the console as follows:

```
val ob = observe(es) { x => println("Receiving " + x) }
```

In case of a UI application, we can have a button class that emits event each time the button is pressed. The values associated with the event might represent the click count:

```
class Button(label: String) {
  val clicks = new EventSource[Int]
  // for each event coming from the UI component, call "clicks emit x"
}
```

It is possible to listen to an event, and associate an action that needs to be executed when the event emits. In a UI application, we can have a "Quit" button that terminates the application.

```
val quitButton = new Button("Quit")
observe(quitButton.clicks) { _ => System.exit() }
```

2.4.2 Signals

Many interactive application needs to deal with continuously changing values. Usually, changes made to internal variables needs to be reflected on the UI, possibly in more than one place. We can represent time-varying values by the type `Signal`. A signal is an observable object that has an associated value in any point of time. There are two main kind of signals: variables and functions (either strictly or lazily evaluated, see below).

Let us consider a fragment of a file manager application that displays, in the form of a label, a text with the number of selected files and folders.

```
val selectedFiles: Var[Int]
val selectedFolders: Var[Int]
val selection: Strict[String] { "There are " + selectedFiles() + "
    files(s) and " + selectedFolders() + " folder(s) selected".

observe(selection) {
  selectionStr => selectionLabel.text = selectionStr
}
```

In the above example, whenever the number of selected files and folders changes, the signal `selection` is automatically evaluated, and the component on the UI is automatically updated.

In this example, the signal is `Strict`, which means that whenever the dependent signals (`selectedFiles` and `selectedFolders`) change their value, the signal is automatically re-evaluated. We need this because we always want to keep the UI updated. For cases where this is not necessary, we can use the type `Lazy` that represents a signal function that is evaluated only if needed. This means that an observer listening to such lazy signals might miss events.

2.4.3 Reactives

Events and signals are very tightly related concepts. They are both *reactives*. The widely used method `observe` expects a reactive, therefore it possible to observe an event source or a signal the same way. The difference between them is only subtle: while a signal always has a defined value, and event source only has a value while it is *emitting* (it does not store the value emitted last time). There are also operations that lets convert between the two: the trait `Signal[A]` defines the method `changes` that creates an event source for the current signal; and the trait `Events[A]` defines the method `hold` that creates a signal that always holds the values of current event (and has a given initial value).

2.4.4 Reactors: observers without inversion of control

One of the most important problem with having to deal with observer-based architecture is the *inversion of control*. When an operation consisting of multiple events needs to be modeled, the typical solution is to model it as a *state machine*.

Let us consider a simple drawing application that lets us draw a closed line using the mouse (variables `mouseDown`, `mouseMove` and `mouseUp` are event sources):

```
var moveObserver: Observer = null
var path: Path = null
observe(mouseDown) {
  downPos =>
    path = new Path(downPos) // new path with a starting position
    moveObserver = observe(mouseMove) {
      movePos =>
        path.lineTo(movePos) // add the current position to the path
        draw(path)
    }
}

observe(mouseUp) {
  _ =>
    moveObserver.dispose()
    path.close() // closes the path
    draw(path)
}
```

The above example has multiple fundamental problems. The drawing logic is inverted (scattered through three observer blocks), the observers needs to be manually disposed, and it is prone to errors (e.g. if a mouse up event is somehow triggered without a mouse down event, the code fails with a null-pointer runtime exceptions — a defensive code would increase complexity).

Ideally, we want to encode a state-machine that can be described informally as follows:

1. Once the mouse button is pressed, start a new path.
2. While the mouse isn't released, gather all mouse moves and draw the path.
3. Once the mouse is released, close and draw the path.

Scala.React's imperative data-flow language and the concept of *reactors* lets us turn the above steps into a straightforward implementation [4]:

```
Reactor.loop {
  self =>
    // step 1
    val path = new Path(self await mouseDown)
    self.loopUntil(mouseUp) {
      // step 2
      val m = self awaitNext mouseMove
      path.lineTo(m)
      draw(path)
    }
    path.close() // step 3
    draw(path)
}
```

4. See figure B.2 in appendix B for a complete example

The method `Reactor.loop` creates a reactor that repeatedly executes the given block. The variable `self` lets us control the reactor, which supports actions such as `await`, `awaitNext` and `loopUntil`. The actions `await` and `awaitNext` pauses the reactor until the given event source emits; whenever it does emit, the reactor is resumed from exactly where it was paused, with all its internal state (e.g. `path`) restored. This is enabled by the Scala Continuations plugin: the method `loop` contains a `reset` block, and the methods `await` and `awaitNext` contain a corresponding `shift` block that creates an observer on the given event source that resumes the continuation. For more details, see 2.4.6.

2.4.5 Data-flow language

Scala-React's data-flow language has a series of operations on reactors that offers control over the reactor:

- `flow`: creates a new reactor with a control flow that is executed once;
- `loop`: creates a new reactor with a control flow that is repeatedly executed, until the reactor is disposed or the reactor calls the `join` method on itself;
- `await`: waits for the given reactive to emit; if it is already emitting in the current cycle, this method immediately returns;
- `awaitNext`: waits for the given reactive to emit; if it is already emitting, it discards the current value and awaits for the next;
- `pause`: suspends the reactor which will be continued in the next iteration — useful to break infinite cycles in reactors or to run a have a long-running operation on the UI without completely blocking (freezing) it;
- `halt`: suspends and disposes the reactor;
- `par`: lets execute *two* branches in *parallel*, until they both finish or until or at least one calls the method `join`; useful in cases we want to have two related control flows that in control of each other;
- `join`: halts the reactor or controls the enclosing `par` expression;
- `abortOn`: discards a reactor block as soon as a given reactive emits; useful if we need a control flow that supports *cancelling* of whatever action it is doing;

2.4.6 Implementation details

Scala.React proceeds in discreet steps, called *turns*. Part of the implementation is a *scheduler* and a *propagator*. External updates, such as when an event source is triggered, are forwarded to the scheduler which schedules a revalidation request for the next turn. In the next turn, the propagator takes over, and revalidates all related events and signals, possibly creating other turns (e.g. when there is a dependent signal on a signal being updated). This turn-by-turn evaluation strategy makes it possible that the system is always in a consistent state (e.g. it is *glitch-free*). See figure B.1 in appendix B for an example demonstrating consistency.

Signal expressions

Scala.React propagates changes and resolves dependencies using an internal representation of *nodes* forming a *tree*. Event sources and signals are represented as such nodes. As part of the revalidation, subtrees are inspected and, in case of opaque nodes (see below), are rebuilt.

A signal can depend on other signals; whenever the *dependent* signals change their value, the *depending* signal needs to be notified (e.g. to be re-evaluated in case of strict signals). This means that there is an implicit observer-subject relationship between such signals. Maintaining such relationship is straightforward for derived signals such as `val s1 = s0.map(...)` where it is statically known that `s1` depends on `s0`. A signal expression of the form `Strict{ a() + b()}` however is completely opaque to the framework (the block is a closure), e.g. it has no prior knowledge that it depends on the signals `a` and `b` before actually evaluating it. The nodes of such signals are therefore called *opaque* nodes.

When defining a signal such as `Strict{ a() + b()}` we are in fact calling a factory method

```
def Signal[A](op: =>A): Signal[A]
```

with the argument `{ a() + b()}`. The compiler rewrites the expressions `a()` and `b()` to method calls `a.apply()` and `b.apply()`, which are important below. The parameter `op` of the factory method `Signal` with the type `=> A` is actually a “call-by-name” argument (a nullary function). This means that the expression in the closure body does not get evaluated when the signal is created, but instead it is passed to the newly created signal object. The actual body of the signal is evaluated during a revalidation turn, which uses an internal stack for establish dependencies: before evaluating the above signal expression, the signal is pushed to the stack; during the evaluation of the signal expression if the execution gets to an `apply` method of a signal, the topmost element on the stack is added to the dependents list of the signal. This way, dependencies are automatically discovered, without the need of explicitly stating them.

For some language expressions, dependencies can actually vary. Consider the signal

```
val s = Signal { if (c()) a() else b() }
```

Signal `s` should ideally ever depend on only two signal: on signal `c` and either `a` or `b` (depending on `c`’s value). Therefore, when a node notifies its dependencies of a change, it removes all opaque dependents (and expects them to re-add themselves as needed).

2.5 FEST-Swing

FEST (Fixtures for Easy Software Testing) is a collection of libraries whose mission is to simplify software testing. It is composed of various modules, which can be used with the unit testing frameworks JUnit [9] or TestNG. The most significant module is the FEST-Swing module.

The Swing module provides a simple and intuitive API for functional testing of Swing user interfaces, resulting in tests that are compact, easy to write, and read like a specification. Tests written using FEST-Swing are also robust. FEST simulates actual user gestures at the operating system level, ensuring that the application will behave correctly in front of the user. It also provides a reliable mechanism for GUI component lookup that ensures that changes in the GUI's layout or look-and-feel will not break the tests [5].

2.5.1 Introduction

A FEST-Swing test is testing either individual frames (the building blocks of a UI), or entire Swing applications or Applets. FEST simulates actual user gestures (mouse movements, keys presses) at the operating system level, ensuring that during the application will behave in the same way as in the front of the user. It uses AWT Robot, which generates events in the platform's native input queue. Thus, the test needs to create the components and make them visible on the screen, and the tests will actually moves the mouse cursor, and not just only generates mouse move events.

2.5.2 Architecture

FEST Swing's component fixture architecture is separated into several layers (from bottom to top):

1. BasicRobot: Simulates a user interacting with a mouse and keyboard. It uses the AWT Robot to generate native input events.

2. Component driver: This layer does all the "heavy lifting." All interaction with a GUI component is done in this layer. It knows how to simulate events and check the state of a specific GUI component. For example, JComboBoxDriver knows how to simulate a user using a JComboBox (selecting a particular element) and how to verify the state of it (which element should be selected.)

3. Component fixture: This layer sits on top of the driver. It provides a fluent interface to that and makes the API easier to write and read. Users of FEST write their GUI tests using fixtures, not drivers. There is one fixture per Swing component, and each fixture has the same name as the Swing component they can handle ending with "Fixture." For example, a JButtonFixture knows how to simulate user interaction and verify state of a JButton. Fixtures can be considered the "user interface" of the FEST-Swing library.

The architecture also supports extensions, so developers of application using custom Swing components can write their own fixtures and component drivers.

2.5.3 Example

Writers of GUI tests need to use the fixtures located in the package org.fest.swing.fixture. These fixtures provide specific methods to simulate user interaction with a GUI component and they provide assertion methods that verify the state of the GUI component. Although it is possible to work with the FEST BasicRobot directly, the BasicRobot is too low-level and requires considerably more code than the fixtures.

As a concrete example, let us consider a very simple JFrame that contains a JTextField, a JLabel and a JButton, and component has its unique name. The expected behavior of this GUI is the following: when user clicks on the JButton, the text of the JTextField should be copied to the JLabel. (Figure 2.5.3)

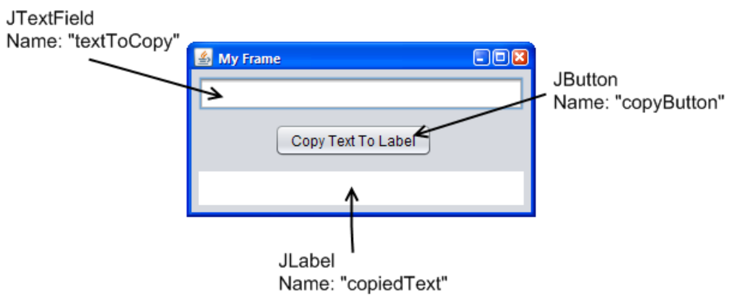


Figure 2.6: A very simple JFrame that contains a JTextField, a JLabel and a JButton.

To the frame, the test's setup method needs to create the frame (in the EDT, delegating it with GuiActionRunner), create a fixture for it, and make it visible (Figure 2.5.3).

```
protected def onSetUp() : Unit = {
   val frame : MyFrame = GuiActionRunner.execute(new GuiQuery[MyFrame] {
      protected def executeInEDT() : MyFrame = new MyFrame
   })
   window = new FrameFixture(robot, frame)
   window.show() // shows the frame to test
}
```

Figure 2.7: The setup method creating the frame and the fixture, and making it visible.

The test method can use the fixture to simulate a user interacting with a GUI in order to verify that such GUI behaves as we expect. The user interactions and the assertions in the test are fluent and simple. The method looks up the UI components by their unique names. (Figure 2.5.3)

```
@Test
def shouldCopyTextInLabelWhenClickingButton : Unit = {
   window.textBox("textToCopy").enterText("Some random text")
   window.button("copyButton").click()
   window.label("copiedText").requireText("Some random text")
}
```

Figure 2.8: The test method.

It is important to mention that besides testing individual components that build up an application, FEST-Swing also supports testing entire applications. In such cases, the first tests starts up the application with ApplicationLauncher that understands how to launch an application using its main method. Once the application is started, the test just needs to find the application's main window.

2.5.4 Threading model

The documentation of FEST-Swing strongly advises to respect Swing's threading rules [18] both in the tests and in the tested application (this remains only an advice because Swing itself does not enforce thread safety). In short, the cardinal rule is the following: creation and access (both read and write) of Swing components should be done in the Event Dispatch Thread (EDT.) Since JUnit and TestNG tests do not run on the EDT, creation and any direct access in the tests to Swing components should be delegated to EDT via the utility class GuiActionRunner (or via other tools).

To ensure that the threading rules are respected throughout the tests and the tested application, FEST Swing provides the class FailOnThreadViolationRepaintManager. It forces a test failure if access to Swing components is not performed on the EDT. However, it only detects component creation and writing, and is unable to detect read operations (component creation and writing triggers repaint events, but read does not).

2.5.5 Limitations

Because FEST-Swing is actually simulating user interaction, it needs an environment very similar to actual user environments: the tested application needs to be active and in focus, and needs to move the mouse. Therefore, that machine that the tests run on cannot be used for the entire duration of the tests. In addition, the tests probably fail whenever another applications unexpectedly gets the focus and covers the application's window (the mouse no longer clicks the correct application). This can become a burden because UI tests, by their nature, can take a significant amount of time to execute and especially when using agile development methods, it is preferable to run them frequently.

Because of this, it is common to delegate the tests to Continuous Integration (CI) systems (e.g. Hudson/Jenkins, TeamCity etc.) When the CI platform is based Linux, BSD or UNIX-style operating systems, it is common that the server does not even have the X Window system, running applications with UI impossible. For this case, Xvfb offers a simple and straightforward solution [6].

However, if the target platform is Windows, there are a whole set of issues, and the known solutions are problematic, and are very complicated and time consuming to setup. [7] [10]

To overcome all of these limitations, OpenJDK's project Caciocavallo can be used [10] to provide a graphics stack for the Java VM that is completely independent from the environment, eliminating any need for platform-dependent setup for CI systems. It renders everything into a virtual screen (which is simply a BufferedImage object), and is driven solely by AWT Robot events. However, this also has its own limitations; the most important being that drag-and-drop is unsupported.

2.6 Design patterns

2.6.1 Model-View-Controller

Model-View-Controller is an architectural pattern for implementing user interfaces[12]. It divides an application into three connected parts:

- **Model** — consists of the application data, business rules, logic and functions;
- **View** — consists of the visual representation of the data, along with components that a user can interact with (e.g. consists of the Java Swing components);
- **Controller** — accepts inputs from the view, and converts it to commands for the model or view.

The interaction between the three components is defined as follows:

- A **controller** knows the view and the model, it listens to events of the view triggered by the user and updates the view and the model accordingly.
- A **view** accepts user interaction, and notifies its observers (u.i. the controller). It also listens to the changes of the model, and updates its state accordingly.
- A **model** notifies its observers (u.i. the view) whenever its state changes.

Figure 2.9 shows the relation between the three components of the MVC pattern. The pattern relies on the *Observer* pattern: the view is the subject of the observing controller, and the model is the subject of the observing view (hence the indirect dependencies on figure 2.9).

The purpose of this separation is that the parts can be interchangeable (e.g. more than one view can exist for the same model, or more than behavior can be defined using different controllers), and are testable separately (e.g. it is possible to the test just the model without a UI, to test just the view with a mocked model, or to test the controller with mocked view and model).

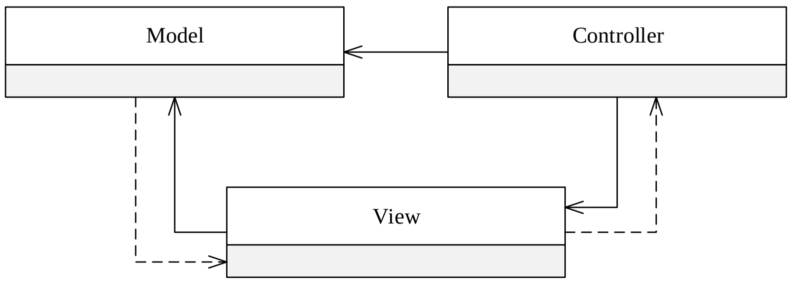


Figure 2.9: The Model-View-Controller pattern.

2.7 Functional testing of an application with a user interface

2.7.1 User Action and Adapter design pattern

The users using an application with a UI typically interact with it by executing a series of operations that change the application's state. These operations might themselves be consisting of multiple operations of the UI components, changing the UI's state and, indirectly, changing the application's internal state. UI test simulating the users can incorporate in their architecture the concept of separation: the tests can interact with a user action layer (consisting of the actions the user can do with the application), and the user actions can interact with an adapter layer that itself interacts with the UI components (e.g. by using FEST-Swing).

The same separation applies to the tests' assertions: the tests assert the application's state through the user actions, and the user actions assert the UI's state, and thus, indirectly, assert the application's internal state.

2.7.2 Adapters

The adapters understand how to execute operations on the application by using the UI components.

For example, to create a new text file in an editor application supporting multiple types of files, the user typically needs to click on the File menu, then on the menu item New, and then on the sub-menu-item Text file. The combinations are usually limited (the submenu-item cannot be clicked without clicking the menu item first), and sometimes require some checking (if the File menu is already visible for whatever reason, the test should not click it again because that would only close the menu).

These operations come from the functional specification of the application, and their implementation details are only dependent on the UI design. These operations can be structured into adapters, where each logically bound component group can have a corresponding Adapter class (e.g. FileMenuAdapter), where the methods are the operations (e.g. FileMenuAdapter#clickNewTextFile()). The adapter class can have a fixture for each Swing component, and the methods can directly interact with them.

This approach also leads to tests that are more resistant to UI changes, requiring less code change for each UI design change. For example, if the in the UI design of the above text editor the Text file menu item is moved directly into the File menu under the name New text file, only the appropriate adapter class needs to be modified, while all the tests using this operation are left unchanged. Whereas, if the code using the fixtures and clicking on the menu items was present in all the tests, such a change would need all the tests to be changed.

2.7.3 User actions

The user actions understand how to execute operations of the application by using the adapters.

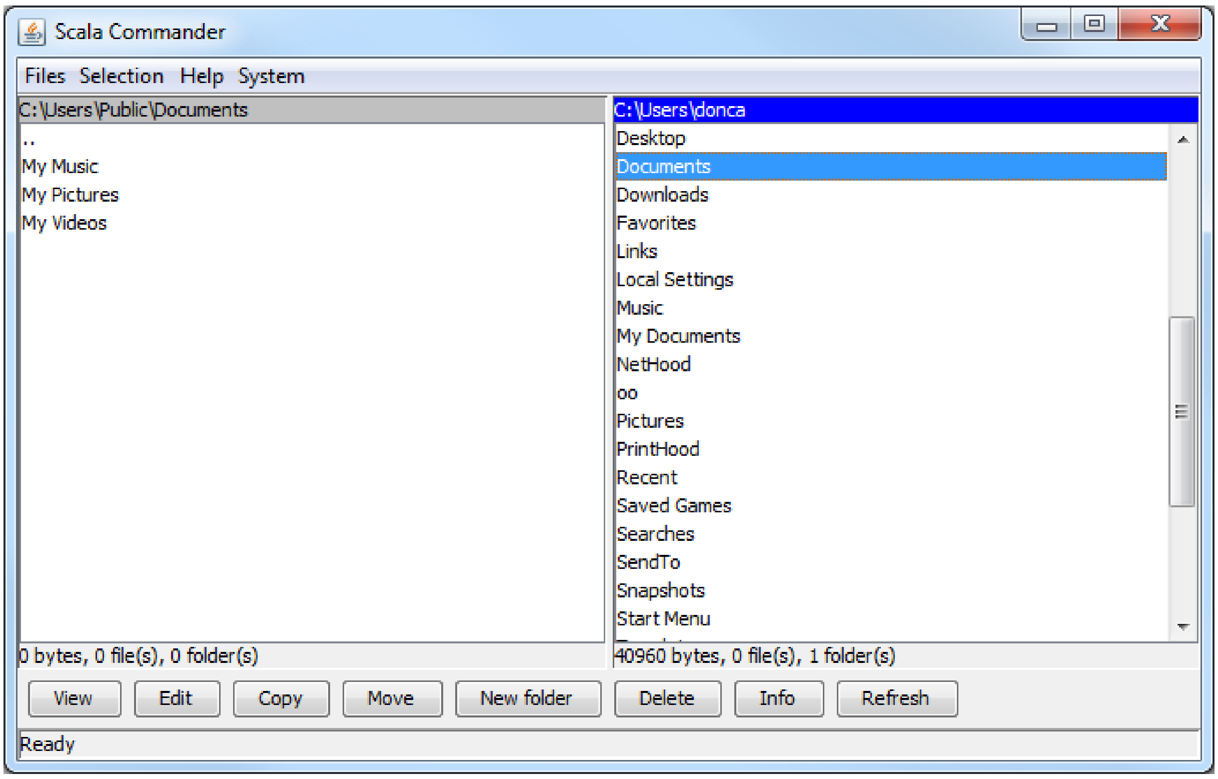
For example, let us consider a case where the user wants to save the file in the current editor of an editor application with a new name. The user needs to click the Save as menu item, resulting in the appearance of a browser dialog. After that, he needs to enter the new file name into a text box, and then needs to click the OK button of the dialog, and finally wait for the dialog to disappear. In this case, the file menu can have an adapter; the file browser dialog can have an adapter; and using both of them in the same time in a given way can represent a user action. For example, the user actions related to the current edited document can be grouped into a user action class (EditorUserAction), and where the methods are the user actions (EditorUserAction.saveAs(newFileName)).

3. Chapter

Scala Commander

Scala Commander is a simple desktop application built as a proof-of-concept for the libraries *Scala.React* and *Scala-Swing*, and for Scala as a language of choice for development of reactive applications.

Scala Commander is an *orthodox file manager* application (see figure 3.1). The center of the window consists of two panels (directory lists) each showing the contents of one directory. The user can navigate to an arbitrary directory, can select files and folders, and can execute operations on the selection such as copying, moving, or deletion. The bottom of the window is for file system operations: there is a toolbar with a list of buttons representing file system operations, and whenever a button is clicked, a helper panel appears that prompts the user to confirm the action.



```
C:\Users\Public\Documents
..
My Music
My Pictures
My Videos
0 bytes, 0 file(s), 0 folder(s)
```

```
C:\Users\donca
Desktop
Documents
Downloads
Favorites
Links
Local Settings
Music
My Documents
NetHood
oo
Pictures
PrintHood
Recent
Saved Games
Searches
SendTo
Snapshots
Start Menu
40960 bytes, 0 file(s), 1 folder(s)
```

```
Ready
```

Figure 3.1: Scala-Commander.

The user interface is **constantly reacting to the user input**: as the user selects files and folders, a label always summarizes the selection (shows the number of selected files and folders and the total size of files). File system operations are also reactive: after clicking the *Copy* button, the user can further change the selection (the files and folders to be copied) in the source panel, and can change the destination folder by navigating in the destination panel. Such changes are automatically reflected in the helper panel prompting the user at the bottom of the window. While executing a long-running operation, such as copying, the user interface is not blocked, and the user can always cancel the operation by pressing the *Cancel* button.

The application is built on *Scala.React* and on the the Model-View-Controller design pattern (see section 2.6.1): the model consists of *event streams*, *signals* and *reactors*, the view listens to the model signals, and the controller interacts with the model's event streams for simple operations such as navigation, and uses *reactors* for complex operations such as a file system operation.

To demonstrate the design choices, we describes two components in greater detail: the *directory list* and the file system operation *copy*.

3.1 The directory list

The main window consists of two directory list components: the *left* list and the *right* list. Both serve the same purpose: the user can see the contents of a directory, can navigate to a subdirectory or to the parent directory, and can select files and folders in the current directory. At any point of time, the directory list that the user last used is the *active* list, while the other list is the *inactive* list. The operations on the toolbar are always executed on the selection of the active list, and the destination of the operation (e.g. copy or move) is the directory of the inactive list.

The design of the implementation is according to the Model-View-Controller design pattern (see figure 3.2). The implementation is heavily based on Scala.React concepts, resulting in significantly reduced code length and complexity, as there is no need care for event propagation, dependencies between variables, consistency, and so on.

3.1.1 Model

The directory list model (class `DirectoryListModel`) holds the state of a navigable list and defines a set of operations. The operations alter the internal state of the model (e.g. change the current directory, or change selection). The model is entirely built around Scala.React concepts, such as signal variables, signal functions, event streams and reactors.

The state of the model is defined by the following signal variables:

- `currentDirectory: Var[Path]` — holds the current directory (as a `java.nio.Path` object);
- `selectedPaths: Var[Set[Path]]` — holds the set of currently selected paths (the selection is highlighted on the user interface);

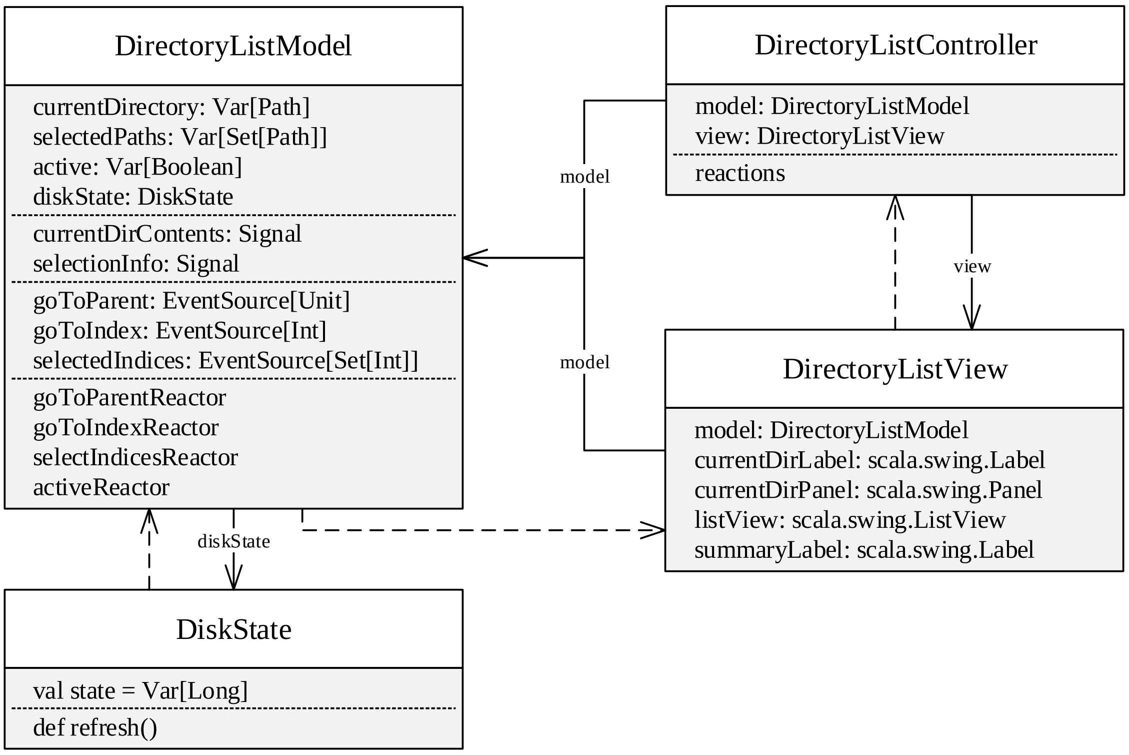

DirectoryListModel
currentDirectory: Var[Path]
selectedPaths: Var[Set[Path]]
active: Var[Boolean]
diskState: DiskState
currentDirContents: Signal
selectionInfo: Signal
goToParent: EventSource[Unit]
goToIndex: EventSource[Int]
selectedIndices: EventSource[Set[Int]]
goToParentReactor
goToIndexReactor
selectIndicesReactor
activeReactor
DirectoryListController
model: DirectoryListModel
view: DirectoryListView
reactions
DirectoryListView
model: DirectoryListModel
currentDirLabel: scala.swing.Label
currentDirPanel: scala.swing.Panel
listView: scala.swing.ListView
summaryLabel: scala.swing.Label
DiskState
val state = Var[Long]
def refresh()
model
model
view
diskState

Figure 3.2: The directory list components.

- `active: Var[Boolean]` — a flag representing whether the directory list is active or inactive;
- `diskState: DiskState` — an object with a counter signal that is incremented whenever the disk is modified and the model need refreshing (u.i. after copying/moving/renaming/deleting, or when the user presses *Refresh*).

There are also two derived signals:

- `currentDirContents` — a strict signal that builds the actual list of files and folders that are located inside the current directory; it is dependent on `currentDirectory` and `diskState` (the value is just queried and ignored);
- `selectionInfo` — a strict signal holding an aggregated summary of what the user selected, depending `selectedPaths` and `currentDirectory`.

The model also contains three event sources, each representing an *action* that is defined on the directory list:

- `goToParent: EventSource[Unit]` — go to the parent of the current directory (if it exists);

- `goToIndex: EventSource[Int]` — go to the directory identified by the given index (index of the list `currentDirContents`);
- `selectIndices: EventSource[Set[Int]]` — select paths identified by the given set of indexes.

Finally, the model consists of four reactors that glue together the model's state and the event sources: whenever a certain event source emits, they alter the model's state accordingly. The four reactors are the following:

- `goToParentReactor` — awaits the next `goToParent` event in a loop; when it emits, sets the variable `currentDirectory` to the parent of the current directory (if it exists);
- `goToIndexReactor` — awaits the next index of `goToIndex` in a loop; when it emits, sets current directory to the directory identified by the index emitted by the event based on the current value of `currentDirContents`;
- `selectIndicesReactor` — awaits the next index set of `selectIndices` in a loop; when it emits, sets the variable `selectedPaths` based on the current value of `currentDirContents`;
- `activeReactor` — awaits the next value of the signal `active`, and if it is set to false, clears the selection (sets the variable `selectedPaths` to the empty set).

3.1.2 View

The directory list view (class `DirectoryListView`) is responsible for visualizing the model. It is a `scala.swing.BorderPanel`, and contains four subcomponents (figure 3.3):

- `currentDirLabel`: a label displaying the full path of the current directory (marked as `1`);
- `currentDirPanel`: the panel containing the label `currentDirLabel`; the panel's background is blue for the active list and gray for the inactive list (marked as `2`);
- `listView`: a `scala.swing.ListView` component that displays the contents of the current directory, and consists of the *list data* and the *selection* (marked as `3`);
- `summaryLabel`: a label displaying a summary of the current selection (marked as `4`).

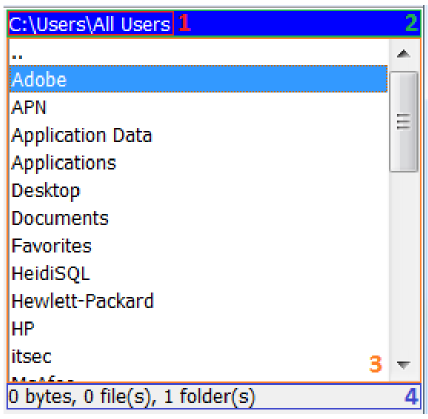


Figure 3.3: The directory list view and its subcomponents.

The view contains `observe` blocks listening to model signals, where each listener updates the appropriate Swing component. This ensures that the view is always up-to-date as the model changes.

```
observe(model.currentDirectory) {
  currentDir: Path => currentDirLabel.text = currentDir.toString
}
```

Figure 3.4: The `observe` block listening to the changes of the model signal `currentDirectory`.

The view listens to changes of the following model signals:

- `currentDirectory` — the view updates `currentDirLabel` (see figure 3.4);
- `currentDirContents` — updates the *list data* of `listView` and clears the *selection*;
- `selectedPaths` — updates the *selection* of `listView`;
- `selectionInfo` — updates `summaryLabel`;
- `active` — updates the background of `currentDirPanel`.

3.1.3 Controller

The controller (class `DirectoryListController`) listens to the events of the view's components (extends the trait `scala.swing.Reactor`), and triggers the appropriate event sources from the model.

The controller listens to the *mouse clicks*, *selection* and *key* events of the `listView` component:

```
listenTo(view.listView.mouse.clicks, view.listView.selection,
    view.listView.keys)
```

The controller reacts to the events generated by the above sources as follows:

```
reactions += {
  case MouseClicked(_, _, _, 2, _) => // triggered by 'mouse.clicks'
    val leadIndex = view.listView.selection.leadIndex
    model.goToIndex << leadIndex
  case ListSelectionChanged(_, _, false) => // triggered by 'selection'
    if (!view.listView.updating) {
      val selection = view.listView.selection.indices.toSet
      model.selectIndices << selection
    }
  case KeyPressed(_, Key.Enter, _, _) => // triggered by 'key'
    val leadIndex = view.listView.selection.leadIndex
    model.goToIndex << leadIndex
  case KeyPressed(_, Key.BackSpace, _, _) => // triggered by 'key'
    model.goToParent << Unit
}
```

Note that the event handling mechanism relies heavily on Scala's pattern matching. For example, in order to listen to double click events, the first case matches events typed of the case class `MouseClicked` with the fourth parameter `clicks` fixed to the value `2`.

3.1.4 Interaction between the components

Figure 3.5 shows the interaction between the components of a directory list when the user changes the current directory. It consists of four stages:

1. As the user interacts with the UI (double clicks on `listView`), the event is handled by the controller, and the controller makes event stream `goToIndex` to emit an index (the index of the line where the double clicked).

2. In the next turn, Scala.React resumes the reactor `goToIndexReactor`, which gets the index, and queries `currentDirContents` and `currentDirectory`, and updates `currentDirectory` and `selectedPaths`.

3. In the next turn, Scala.React revalidates the strict signal `currentDirContents` because its dependency `currentDirectory` was changed.

4. In the next turn, Scala.React notifies the observer blocks in the view listening to model signals changed in previous turns; this results in the view components being updated.

3.2 Copying files

Copying is according to the customs of orthodox file managers: the user selects files and folders in one panel and copies them to the directory where the other panel is navigated to. Traditionally, this is implemented using modal dialogs: once the user clicks a "Copy" button, a modal dialog appears prompting him to confirm the action. When confirmed, another modal dialog appears showing the progress of the copying, and until the copying is over, the user cannot interact with the application.

Our application, however, takes a slightly different approach: when clicking the "Copy" button, a panel at the bottom of the window appears. The panel shows the number of files and folders to be copied based on the selection in the active panel, and also contains a text box with the destination path which is the directory of the inactive panel. As the user subsequently navigates the panels, changes the selection, or switches activity, the contents of the panel are automatically updated. Once the user clicks the "OK" button in the panel, the selection and the destination directory is captured, and the system beings the copying: the status bar of the window shows the progress, and at any time, the user can abort the operation by pressing the "Cancel" button. Copying does not block the application, it remains usable. After copying is done, the directory lists are refreshed.

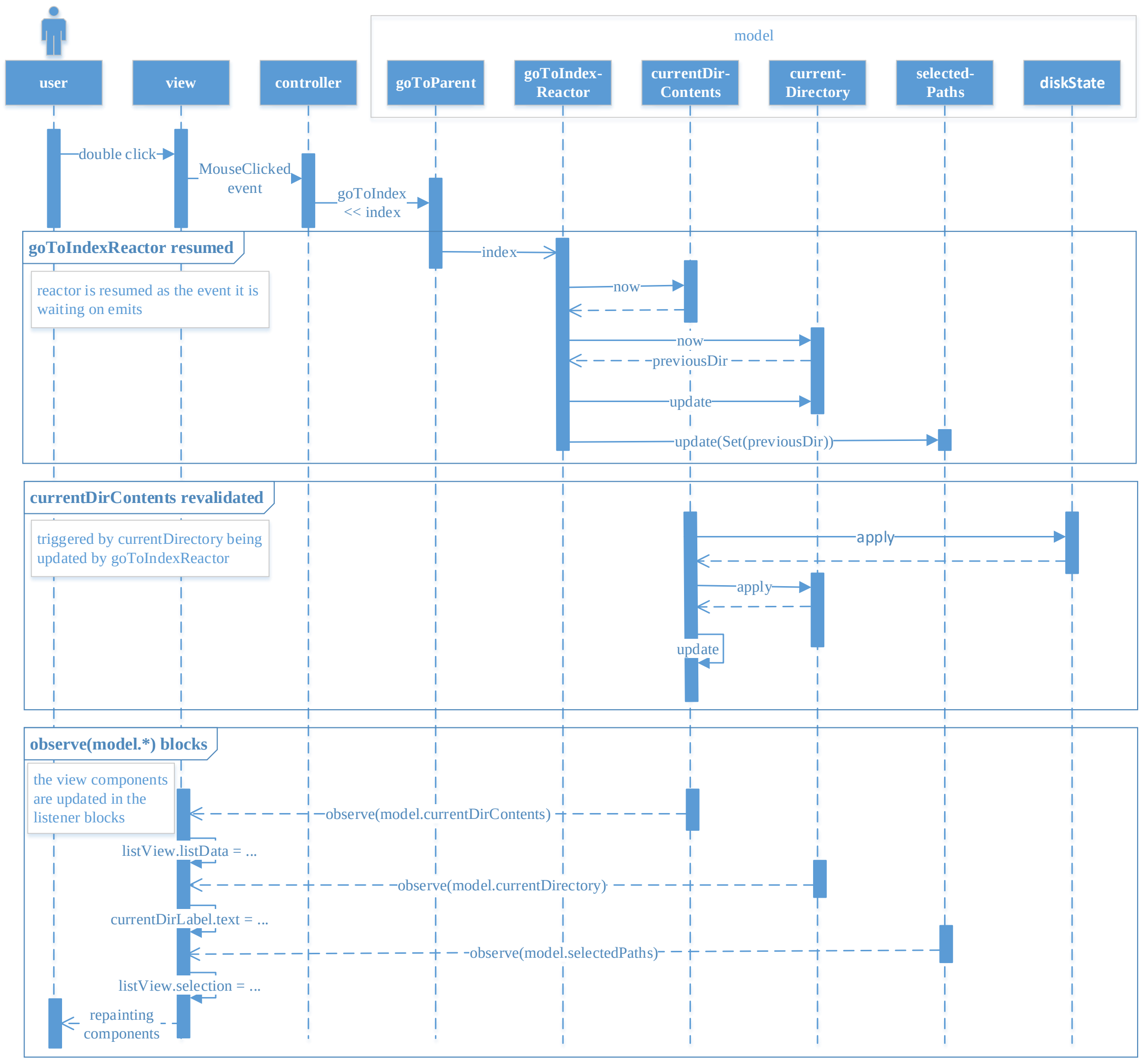


Figure 3.5: Sequence diagram showing the interaction between the components of a directory list when the user navigates in a directory list.

3.2.1 Directories pane

Because copying is defined such as to copy the selection of the *active* list to the directory of the *inactive* list, we need to be able to refer to the active and to the inactive directory list no matter if the left list is active and the right is inactive or vice-versa. This is where the *directories pane* comes into the picture: it consists of the two directory list panels, keeps track of activity based on UI events, and the model has signal functions that always reflect certain properties of the lists. For example, the signal `activeCurrentDir` always has the value of the active directory list's path, and it is updated when either the user navigates in the active list or switches to the other list.

The model is as follows:

```
class DirectoriesPaneModel(val left: DirectoryListModel,
                           val right: DirectoryListModel) {

  val activeList = Var[DirectoryListModel]
  val inactiveList = Var[DirectoryListModel]

  val activeCurrentDir = Strict { activeList().currentDirectory() }
  val inactiveCurrentDir = Strict { inactiveList().currentDirectory() }
  val activeSelection = Strict { activeList().selectionInfo() }
  val inactiveSelection = Strict { inactiveList().selectionInfo() }
}
```

3.2.2 Copy controller

Copying is a single reactor: it awaits the user pressing the *Copy* button on the toolbar, displays the panel, awaits the *OK* button, captures the current state of the directories pane and begins copying. The user can abort the copying by pressing the *Cancel* button at any time. After the copying is done or is aborted, the reactor refreshes the disk state and hides the copy panel.

A simplified version of the reactor code is as follows:

```
Reactor.loop {
  self =>
    self awaitNext mainWindowView.commandButtons.copyButton()

    displayPanel(Some(view.panel)) // display the copy panel
    self.abortOn(view.cancelButton()) { // abort on 'Cancel'
      self awaitNext view.okButton() // wait until 'OK'

      val sources: Set[Path] = directoriesPane.activeSelection.now.paths
      val destinationDir: Path = directoriesPane.inactiveCurrentDir
      sources.cps foreach { // do the copying
        ... self pause ... // for 'abortOn' to be able to abort the copying
      }
    }
    diskState.refresh()
    displayPanel(None) // hide the copy panel
}
```

Note that the `abortOn` operator executes the body and aborts it as soon as the given reactive (the cancel button) emits. This makes it possible to cancel while waiting for the *OK* button or while actually copying the files.

3.3 Conclusion

In this chapter, we presented the most important implementation details of Scala Commander, a design based on the reactive framework Scala.React. Through the example of the *directory lists*, we presented how an MVC-based application can take advantage of the reactive abstractions of Scala.React, and we have seen how the automatic change propagation between dependent signals effortlessly keeps the state of the application consistent and up-to-date. Through the example of copy, we have seen how a *reactor* can be used to implement a relatively complex event-driven logic in a direct, imperative style without inversion of control.

The source code of Scala Commander, along with the sources of FEST-Logging and Scala.React are accessible to the public at `https://github.com/zsolt-donca/scomm-dissertation`

4. Chapter

FEST-Logging

Test suites testing even relatively complex applications can easily contain some hundred tests. Because UI tests, by their nature, typically take much time, the total run time of an entire test suite is usually important. In order to keep the overall runtime at minimum, it is typical to avoid the creation, initialization and the destruction of the individual UI components for each test (which might also be difficult or impossible depending on the architecture of the application). A possible option is to test entire applications and to reuse the same components between the tests.

The problem with testing entire applications throughout entire test suites is that the tests can become dependent on the internal state of the application and on the state of the UI. Thus, it is possible for one test to affect the outcome of another, e.g. one test makes a subtle change in the state of the application and makes another test fail, or, in a more extreme case, one failing test can leave the application in such a state that no other further tests will pass. Having non-independent tests can lead to fragile and unstable test systems and thus should be avoided as much as possible.

Unstable test system can very difficult to cope with. Tests can seemingly fail randomly, and it can be very difficult to reproduce failures that originate from typically a subtle change in the application's state that was made by a previous test (possibly executed many tests before). FEST supports saving screenshots anytime during a test, but the test must explicitly save it and it is usually done only on errors. Since in the case of unstable tests, the errors typically manifest themselves only in the tests that are affected by the erroneous tests' side effects, saving screenshots for failing tests does not help much on its own.

Understanding the data flow of all the tests of an entire test suite, knowing all the operations that were made on the application's state, along with screenshots made after all important steps can greatly reduce the debugging efforts of an unstable test suite and can benefit the test development and maintenance process.

4.1 AspectJ-based auditing

We introduce *FEST-Logging*, an AspectJ-based solution that gathers information of the test methods and on annotated methods in the user code. The gathered information (method arguments, screenshots) are visualized in the form of a table and the entire execution tree can be inspected. AspectJ [3] is an implementation of the programming paradign Aspect-Oriented Programming [11]. We are integrating with *JUnit*, a unit-testing framework for Java. [9]

We introduce the JUnit runner `CacioFESTLoggingRunner`: enables FEST-Logging and executes the tests using the *Cacio-tta* graphics stack (running the tests in the background) [10]. Running the tests with this runner is a **requirement** of FEST-Logging.

We also introduce two annotations:

- `GUITestBean`: a *type* annotation
- `GUITestAction`: a *method* annotation.

FEST-Logging provides the following features:

- auditing tests — audits the *test*, *setup* and *teardown* methods (annotated with `@Test`, `@Before` and `@After`).
- auditing test actions — audits *all* methods of classes annotated with `@GUITestBean`;
- screenshots of test actions — the runtime system takes screenshots after methods annotated with `@GUITestAction` are executed;
- screenshots of failed tests — the runtime system takes screenshots of failing tests, failing either in their *test*, *setup* or *teardown* methods.

Tests using FEST-Logging generate reports as files in the folder *reports/xml*. The reports can be visualized using the HTML page *web/report.html*.

4.1.1 Implementation

Using a JUnit runner lets us add additional behavior to the execution of a test, similarly to advices in AspectJ. The runner `CacioFESTLoggingRunner` takes this opportunity to register the test in the object `MethodCallStack` and to execute the test. When the test is over, it captures any exceptions thrown by the test, unregisters the test from `MethodCallStack` and writes the audited data to the reports folder.

The test methods and test actions are altered by the aspect class `FESTLoggingAspect`. Here, we define a series of pointcuts, where each pointcut is to match a method annotated with a certain annotation (see figure 4.1). There are also *around advices* defined referring to the pointcuts, where each advice alters the execution of the method:

- `auditTestAndGUITestBeanMethods` — registers and gathers the result of the advised method;
- `takeScreenshotOfFailedTest` — executes test methods, and takes a screenshot if they throw exceptions;

- takeScreenshotsOfTestActions — executes the advised method and takes a screenshot of the desktop.

Each advice relies on the state of the object MethodCallStack, and stores the gathered data in it.

```
@Aspect
class FESTLoggingAspect {
  @Pointcut("execution(@org.junit.Test * *.*(..))")
  def testMethods() {}

  @Pointcut("execution(@org.junit.Before * *.*(..))")
  def beforeMethods() {}

  @Pointcut("execution(@org.junit.After * *.*(..))")
  def afterMethods() {}

  @Pointcut("execution(* (@edu.zsd.festlogging.GUITestBean *).*(..))")
  def guiTestBeanMethods() {}

  @Pointcut("execution(@edu.zsd.festlogging.GUITestAction * *.*(..))")
  def guiTestActionMethods() {}
  ...
}
```

Figure 4.1: The pointcuts matching to test methods (first three) and to test actions (last two).

Building the method stack

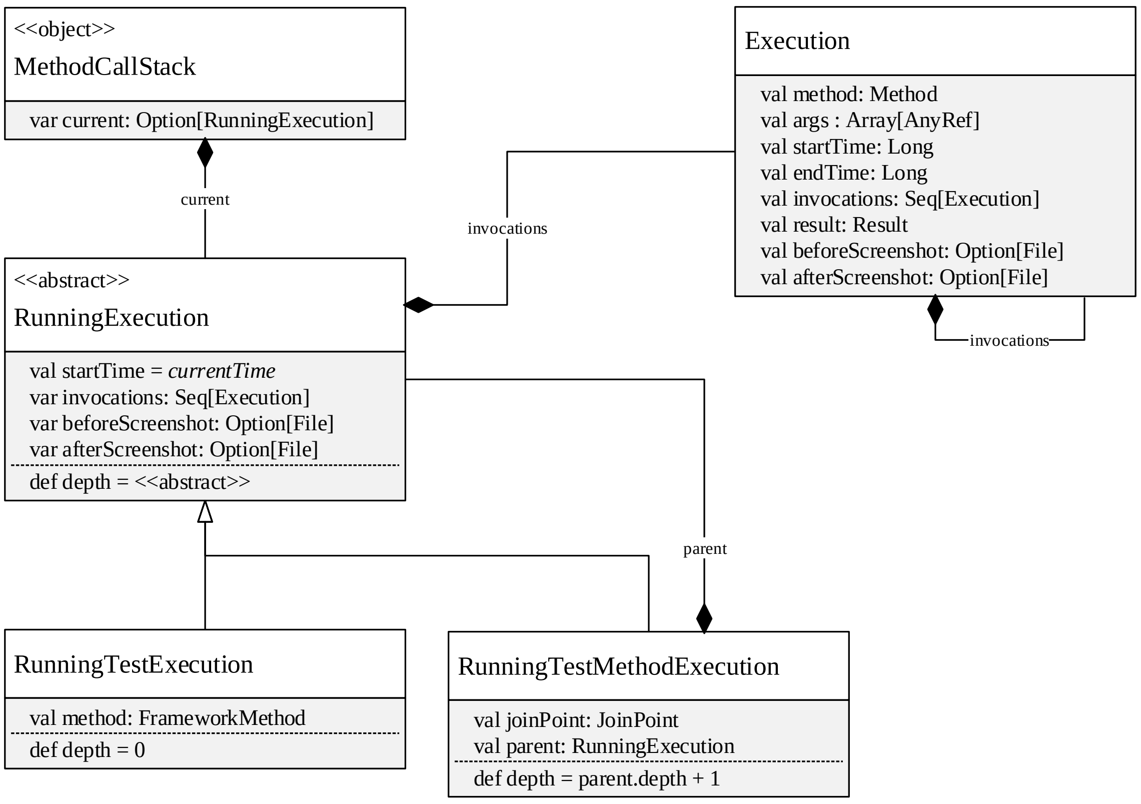
Figure 4.2 shows the advice responsible for auditing. It calls the method enterTestMethod, proceeds the join point, and finally calls the exitTestMethod. This is so the class MethodCallStack, as the name suggests, can keep a stack of objects representing the advised methods.

```
  @Around("testMethods() || beforeMethods() || afterMethods() ||
     guiTestBeanMethods()")
  def auditMethods(joinPoint: ProceedingJoinPoint): AnyRef = {
    MethodCallStack.enterTestMethod(joinPoint)
    try {
      val result: AnyRef = joinPoint.proceed()
      MethodCallStack.exitTestMethod(joinPoint, result)
      result
    } catch {
      case e: Throwable =>
        MethodCallStack.exitTestMethod(joinPoint, e)
        throw e
    }
  }
```

Figure 4.2: The pointcuts matching to test methods (first three) and to test actions (last two).

The class MethodCallStack holds a single reference named current of type RunningExecution (see figure 4.3). This represents the *top* of the stack. The very first running execution object is pushed into the stack by `CacioFESTLoggingRunner` in the form of a `RunningTestExecution` object. The class `RunningExecution` holds the advised method, the argument list, has a mutable list of child method invocations and mutable screenshot fields.

Subsequently, when the actual test methods and the test action methods are executed, the advice `auditMethods` pushes a new `RunningTestMethodExecution` object with the parent reference set to the top of the stack. When a method terminates, the stack is "popped": the top object, referenced by `current`, is converted into a new `Execution` object, this `Execution` object is added to the invocation list of its parent, and this parent is then assigned to the variable `current`.



<<object>>
MethodCallStack
var current: Option[RunningExecution]
current
Execution
val method: Method
val args : Array[AnyRef]
val startTime: Long
val endTime: Long
val invocations: Seq[Execution]
val result: Result
val beforeScreenshot: Option[File]
val afterScreenshot: Option[File]
invocations
invocations
<<abstract>>
RunningExecution
val startTime = *currentTime*
var invocations: Seq[Execution]
var beforeScreenshot: Option[File]
var afterScreenshot: Option[File]
def depth = <<abstract>>
parent
RunningTestExecution
val method: FrameworkMethod
def depth = 0
RunningTestMethodExecution
val joinPoint: JoinPoint
val parent: RunningExecution
def depth = parent.depth + 1

Figure 4.3: Class diagram of the classes internally used by the object `MethodCallStack`. Class `RunningExecution` represents an ongoing execution (with mutable fields), and the class `Execution` represents a terminated execution (with immutable fields).

When the execution reaches back the class `CacioFESTLoggingRunner`, the variable `current` **must be** referencing to a `RunningTestExecution` object. This also transformed to an `Execution` object, but instead of further pushing it on the stack, this object is transformed to XML format and is written to the reports folder.

Taking screenshots

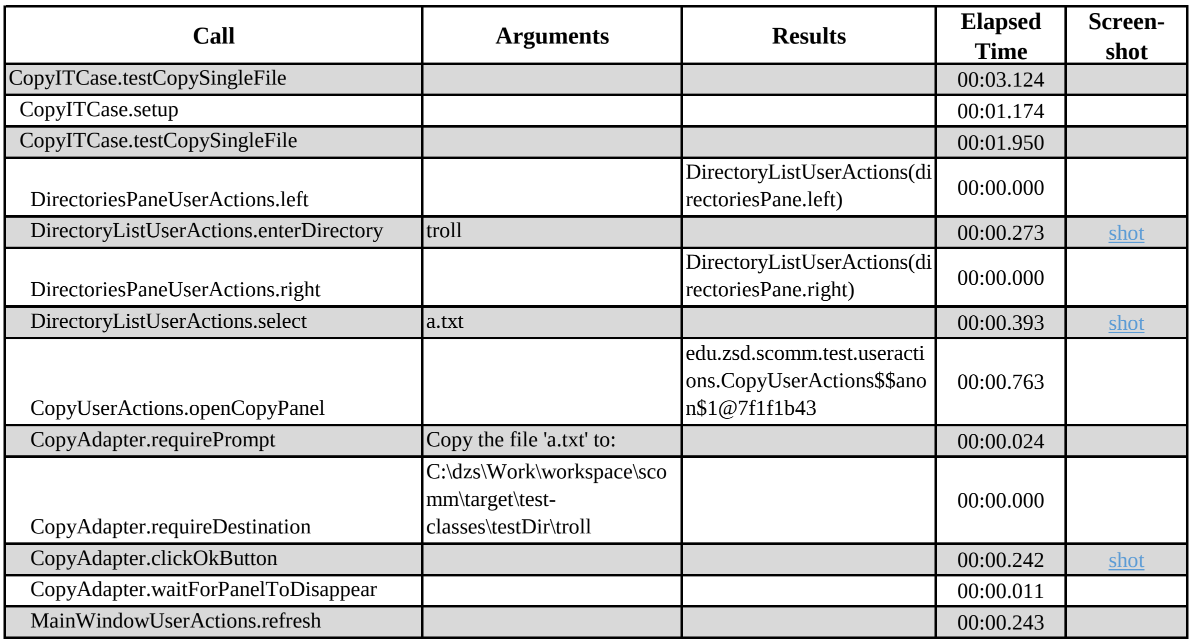
There are two advices taking screenshots: `takeScreenshotOfFailedTest` and `takeScreenshotsOfTestActions`. Both rely on the object `MethodCallStack`: they take the topmost `RunningExecution` object on the stack, take the screenshots into files and assign the appropriate fields in the `RunningExecution` object. This is based on AspectJ applying multiple advices on the same method using well-defined precedence rules (which is declaration precedence in this case).

For performance reasons, FEST-Logging takes the screenshots in two steps: first, it takes a screenshot of the desktop in the form of a `BufferedImage` object on the Swing EDT thread, and then it uses a *Swing worker* to saves the screenshot on a thread pool to a PNG file. This way, it is possible to gain a significant performance improvement when multiple CPU cores are available (see section 4.3 Case study: Scala Commander)

4.1.2 Formatting the audited data

Since the runtime data set can be huge for long tests runs, it is important to provide views that presents the data in an easily accessible, intuitive and adaptive form.

The XML report files generated during the tests can be visualized using the HTML page *web/report.html*. The page visualizes the report data in the form of a pivot table, with a tree axis showing the call hierarchy, and there are columns displaying the runtime data such as method arguments, results and elapsed time. The tree supports closing and expanding nodes (method calls) in the call hierarchy, and there is an additional column with links to the screenshots (see figure 4.4).



Call	Arguments	Results	Elapsed Time	Screen-shot
CopyITCase.testCopySingleFile			00:03.124	
CopyITCase.setup			00:01.174	
CopyITCase.testCopySingleFile			00:01.950	
DirectoriesPaneUserActions.left		DirectoryListUserActions(directoriesPane.left)	00:00.000	
DirectoryListUserActions.enterDirectory	troll		00:00.273	shot
DirectoriesPaneUserActions.right		DirectoryListUserActions(directoriesPane.right)	00:00.000	
DirectoryListUserActions.select	a.txt		00:00.393	shot
CopyUserActions.openCopyPanel		edu.zsd.scomm.test.useractions.CopyUserActions$$anon$1@7f1f1b43	00:00.763	
CopyAdapter.requirePrompt	Copy the file 'a.txt' to:		00:00.024	
CopyAdapter.requireDestination	C:\dzs\Work\workspace\scomm\target\test-classes\testDir\troll		00:00.000	
CopyAdapter.clickOkButton			00:00.242	shot
CopyAdapter.waitForPanelToDisappear			00:00.011	
MainWindowUserActions.refresh			00:00.243	

Figure 4.4: Report of a test.

4.2 Automatic EDT dispatch

The threading model of Swing recommends that all Swing-component-related operations are to be executed on the Swing EDT thread, including those of the tests. While it is always thread-safe to use the FEST Swing fixtures, there are some cases when the methods exposed in a fixture are not enough and one needs to access the underlying component. Direct interaction with Swing components with the purpose of testing must be delegated to the EDT thread (unless the tests can become unstable). Fortunately, FEST-Swing offers the class `GuiActionRunner` that provides a simple way to execute tasks or queries on the EDT, such as:

```
def requireCurrentDirBackground(color: Color) = {
  GuiActionRunner.execute(new GuiTask {
    override def executeInEDT(): Unit = {
      assertEquals(color, currentDirPanel.component().getBackground)
    }
  })
}
```

It is even possible to do assertion in a `GuiTask` because if the assertion fails, the `AssertionError` exception is propagated back the test thread with a merged stack trace that helps debugging. However, this is still much boilerplate code to write just for a simple assertion.

To offer a solution to this problem, we present the annotation `ExecuteInEDT` as part of *FEST-Logging* library. When an annotated method is called on the test thread, the method is seamlessly and transparently delegated to the Swing EDT thread, and the test thread awaits the result (the return value or the exception). This leads to less effort from the developer's side, and at the same time, increases security on the tests' correctness. The above example is equivalent to the following:

```
@ExecuteInEDT
def requireCurrentDirBackground(color: Color): Unit = {
  assertEquals(color, currentDirPanel.component().getBackground)
}
```

4.2.1 Implementation

We define a pointcut that matches methods annotated with `ExecuteInEDT`, and we use an *around advice* to delegate the invocation to Swing EDT:

```
@Pointcut("execution(@edu.zsd.festlogging.ExecuteInEDT * *.*(..))")
def executeInEDTMethods() {}

@Around("executeInEDTMethods()")
def executeInEDT(joinPoint: ProceedingJoinPoint) : AnyRef = {
  GuiActionRunner.execute(new GuiQuery[AnyRef] {
    override def executeInEDT(): AnyRef = joinPoint.proceed()
  })
}
```

The method `GuiActionRunner.execute` takes a single `query` argument and executes it on Swing EDT. If the current thread is already Swing EDT, the method directly executes the argument and returns the value, otherwise it uses `SwingUtilities.invokeLater` to execute the method and wait for the response. In case the method threw an exception, the stack trace is **merged** with the invocation stack of the `GuiActionRunner.execute` method, resulting in an exception that looks just as if the query was executed synchronously on the test thread.

4.3 Case study: Scala Commander

To demonstrate the performance implications of using FEST-Logging, we applied it over the UI tests of Scala Commander.

Scala Commander has 15 tests, grouped into four IT cases:

1. `CopyITCase` — contains four tests, copying a handful of small files and directories and checks the application's state and the disk state;
2. `DirectoriesPaneITCase` — contains seven tests, checking the behavior of the directories pane, ensuring that the user is able to make selections, enter directories, go back to the parents, and able to switching tabs;
3. `NewFolderITCase` — contains three tests, checking the behavior of the application when creating new folders;
4. `OperationsITCase` — contains two tests, checking the generic behavior of the file system operations panel.

The total number of test actions executed while running the test suite is 115, and the total number of audited method invocations is 1067 (see figure 4.5).

IT Cases	Tests	Test actions	Audited methods
4	15	115	1067

Figure 4.5: The number of items in the Scala Commander test suite.

To demonstrate the performance implications of using FEST-Logging, we compared the execution time of the tests running without FEST-Logging (but still using Cacio-tta) with the execution time of the tests running with FEST-Logging in two modes: with method auditing only, and with both auditing and screenshots of the test actions. In each case, we used the average execution time of three consecutive runs. All measurements were done on an HP EliteBook 8570w laptop with Intel Core i7 3720QM CPU 2.60 GHz and 8 GB RAM on Windows 7, with Scala 2.10.3 on Oracle's Java 1.7 update 55.

The results are in figure 4.6. The performance drawback of using auditing only is relatively low (2.11%). Using both auditing and taking screenshots of the 115 test actions takes a somewhat higher 5% of the total execution time, thanks to the parallel saving of the screenshots. In the case when it is not practically possible to save the screenshots in parallel to the tests, the total execution time grows to a significant 50.17 seconds, FEST-Logging taking 21.48% of the total time.

Execution type	**Time**	**Cost**
Without FEST-Logging	41.30 s	
FEST-Logging with auditing only	42.17 s	0.87 s (2.11%)
FEST-Logging with auditing and screenshots	43.36 s	2.06 s (5.00%)
FEST-Logging with auditing and screenshots (single-threaded)	50.17 s	8.87 s (21.48%)

Figure 4.6: The execution time of the Scala Commander test suite in different setups; in each case, the time is the average execution time of three executions.

4.4 Conclusion

In this chapter, we presented FEST-Logging, a framework that integrates into FEST-Swing UI tests, and offers a series of features such as running the tests in the background, detailed auditing of the tests' execution, measuring the executed time, taking screenshots after test actions and failures and easing delegation to Swing EDT. We also presented that obtaining such additional information comes with a reasonable performance cost.

The source code of FEST-Logging, along with the sources of Scala Commander and Scala.React are accessible to the public at `https://github.com/zsolt-donca/scomm-dissertation`

A. Appendix

Scala CPS plugin examples

```
val v1 = reset {
  shift { k: (Int=>Int) =>
    k(7)
  } + 1
} * 2
println(v1) // prints 16
```

Figure A.1: A simple numeric example. The value of the `reset` block is equal to the value of the `shift` block plus one. The `shift`, block gets the rest of the continuation in the form of the function `k`, which in this case is the function $x + 1$. The shift block invokes `k` with the argument 7 and returns the resulting value 8, which is then the value of the `reset` block. This value is then multiplied by 2 (outside the continuation), so the printed value is 16.

```
val v2 = reset {
  shift { k: (Int=>Int) =>
    k(k(k(7)))
  } + 1
} * 2
println(v2) // prints 20
```

Figure A.2: Example showing that the rest of the continuation may be called multiple times. In this case, the rest of the continuation is invoked three times. The final result of the continuation is multiplied by 2 outside the continuation (resulting in the value 20).

```
val result = reset {
 println("entering first shift")
 val firstShift = shift { k: (Int => Int) =>
    val res = k(0)
    println(s"exiting first shift, res = $res")
    res
 } + 1

 println(s"firstShift = $firstShift; entering second shift")
 val secondShift = shift { k: (Int => Int) =>
    val res: Int = k(firstShift)
    println(s"exiting second shift, res = $res")
    res
 } + 1
 println(s"secondShift = $secondShift; returning the reset")

 secondShift
}

println(s"result = $result")

// entering first shift
// firstShift = 1; entering second shift
// secondShift = 2; returning the reset
// exiting second shift, res = 2
// exiting first shift, res = 2
// result = 2
```

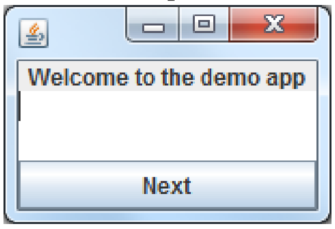
Figure A.3: Example showing the control flow throughout the reset block when there is more than one shift block. Notice the similarity with multiple function calls.

```
import scala.util.continuations._

object TwelveDaysOfChristmas extends App {
  val daysAndGifts = Seq(
    ("First", "a Patridge in a Pear Tree"),
    ("Second", "Two Turtle Doves"),
    ("Third", "Three French Hens")
  )
  val days = daysAndGifts.map(_._1)
  val gifts = daysAndGifts.map(_._2)

  val carol: List[String] = reset {
    val dayIndex: Int = shift {
      verse: (Int => List[String]) =>
        (0 to days.length - 1).foldRight(List.empty[String])((day, list) =>
            "" :: verse(day) ::: list)
    }

    // the continuation below this line calculated a single verse for the
        day identified by `dayIndex`
    val dayLine = s"On the ${days(dayIndex)} day of Christmas my true love
        sent to me"
    val giftIndex: Int = shift {
      line: (Int => String) =>
        dayLine :: (0 to dayIndex).foldLeft(List.empty[String])((list, gift)
            => line(gift) :: list)
    }

    // the continuation below this line calculates a single line of a
        verse, the line identified by `giftIndex`

    val gift = gifts(giftIndex)
    val line = if (dayIndex == 0) {
      s"$gift."
    } else if (giftIndex > 0) {
      s"$gift,"
    } else {
      s"and $gift."
    }
    line
  }
  carol.foreach(println(_))

  /*
  On the First day of Christmas my true love sent to me
  a Patridge in a Pear Tree.

  On the Second day of Christmas my true love sent to me
  Two Turtle Doves,
  and a Patridge in a Pear Tree.

  On the Third day of Christmas my true love sent to me
  Three French Hens,
  Two Turtle Doves,
  and a Patridge in a Pear Tree. */
}
```

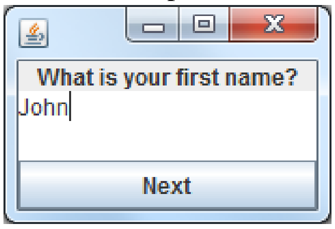
Figure A.4: An example printing the first three verses of the well-known Christmas carol “The Twelve Days of Christmas”. Notice the answer type modification.

```
import scala.util.continuations._
import scala.swing._

object ContinuationsSwingApp extends
    SimpleSwingApplication {

  var continue: (Unit => Unit) = { _ => run()}

  def run() {
    reset {
      val first = ask("What is your first name?")
      val last = ask("What is your last name?")
      label.text = s"Hello, $first $last!"
    }
  }

  def ask(prompt: String): String@cpsParam[Unit,
      Unit] = {
    label.text = prompt
    shift {
      k: (Unit => Unit) => {
        continue = k
      }
    }
    textField.text
  }

  val textField = new TextArea(10, 40)
  val label = new Label("Welcome to the demo app")
  val button = new Button(new Action("Next") {
    override def apply(): Unit = continue()
  })

  override def top: swing.Frame = new MainFrame {
    contents = new BorderPanel {
      add(label, BorderPanel.Position.North)
      add(textField, BorderPanel.Position.Center)
      add(button, BorderPanel.Position.South)
    }
  }
}
```

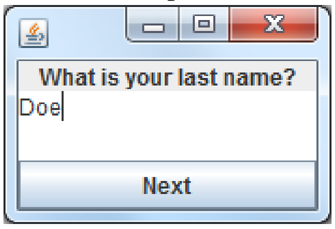
Step1:



Step2:



Step3:



Step4:

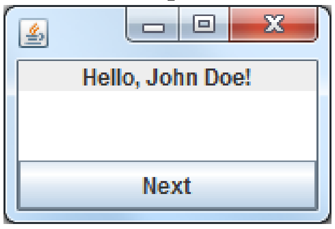


Figure A.5: A simple Swing application with an event-based control flow defined in imperative style (see method `run`). For each question, the flow is suspended and the `shift` blocks assigns the rest of the continuation to a variable. Pushing the button resumes the continuation.

B. Appendix

Scala.React examples

```scala
object person {
  val firstName = Var("John")
  val lastName = Var("Doe")
  val fullName = Strict { firstName() + " " + lastName() }
}

observe(person.firstName) {
  firstName =>
    println(s"The first name changed to $firstName, and last name is
        ${person.lastName.now}.")
}

observe(person.lastName) {
  lastName =>
    println(s"The last name changed to $lastName, and the first name is
        ${person.firstName.now}.")
}

observe(person.fullName) {
  fullName =>
    println(s"Full name changed to $fullName.")
}

def changeName() {
  person.firstName() = "Jane"
  person.lastName() = "Roe"
}
// calling 'changeName' prints the following:
//
// The first name changed to Jane, and last name is Roe.
// The last name changed to Roe, and the first name is Jane.
// Full name changed to Jane Roe.
```

Figure B.1: This example demonstrates that observers always see the state of the application in a consistent state. With using traditional observer pattern, the observers would see the person having the name “Jane Doe” before it finally changes to “Jane Roe”.

```scala
object MyDomain extends Domain {
  val scheduler = new SwingScheduler()
  val engine = new Engine
} import MyDomain._

object ScalaReactLineDrawing extends SimpleSwingApplication with Observing{
  override def main(args: Array[String]) {
    schedule { startup(args) }
    start() // starts the scala-react engine
  }
  override def top: Frame = new MainFrame() {
    contents = new FlowPanel() {
      val mouseDown = EventSource[Point]
      val mouseMove = EventSource[Point]
      val mouseUp = EventSource[Point]

      val mainProgramFlow = Reactor.loop {
        self =>
        // step 1
          val path = new Path(self await mouseDown)
          self.loopUntil(mouseUp) {
            // step 2
            val m = self awaitNext mouseMove
            path.lineTo(m)
            draw(path)
          }
          path.close() // step 3
          draw(path)
      }

      listenTo(mouse.clicks, mouse.moves) // scala.swing events
      reactions += {
        case MousePressed(_, point, _, _, _) => mouseDown << point
        case MouseReleased(_, point, _, _, _) => mouseUp << point
        case MouseDragged(_, point, _) => mouseMove << point
      }
      class Path(var positions: Seq[Point]) {
        def this(pos: Point) = this(Seq(pos))
        def lineTo(pos: Point) { positions = positions :+ pos }
        def close() { positions = positions :+ positions.head }
      }
      var pathDrawn = new Path(new Point(0, 0))
      def draw(path: Path) { pathDrawn = path; repaint() }
      override protected def paintComponent(g: swing.Graphics2D): Unit = {
        super.paintComponent(g)
        val xPoints = pathDrawn.positions.map(pos => pos.x).toArray
        val yPoints = pathDrawn.positions.map(pos => pos.y).toArray
        g.setColor(Color.BLACK)
        g.drawPolyline(xPoints, yPoints, pathDrawn.positions.length)
} } } }
```

Figure B.2: A sample application painting a closed line with the mouse using Scala.React *reactor* and *event streams*.

Bibliography

[1] K. Asai and O. Kiselyov. Introduction to programming with shift and reset, 2011. URL http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf.

[2] P. Bagwell. Swarm - Concurrency with Scala Continuations. URL http://jim-mcbeath.blogspot.ro/2010/09/scala-generators.html.

[3] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse Aspectj: Aspect-oriented Programming with Aspectj and the Eclipse Aspectj Development Tools*. Addison-Wesley Professional, first edition, 2004. ISBN 0321245873.

[4] O. Danvy and A. Filinski. Abstracting control. In *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[5] easytesting.org. Fest, . URL https://code.google.com/p/fest/.

[6] easytesting.org. Running fest under xvfb, . URL http://docs.codehaus.org/display/FEST/Running+FEST+under+Xvfb.

[7] easytesting.org. Hudson under windows, . URL http://docs.codehaus.org/display/FEST/Hudson+under+Windows.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[9] JUnit. Junit. URL http://junit.org/.

[10] R. Kennke. How to use Cacio-tta with FEST for Java GUI testing. URL http://docs.codehaus.org/display/FEST/Hudson+under+Windows.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[12] G. Krasner and S. Pope. A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. URL http://citeseer.ist.psu.edu/krasner88description.html.

[13] I. Maier. The scala.swing package. 2009. URL http://www.scala-lang.org/old/sites/default/files/sids/imaier/Mon,%202009-11-02,%2008:55/scala-swing-design.pdf.

[14] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.React. Technical report, 2012.

[15] J. McBeath. Scala Generators, . URL http://jim-mcbeath.blogspot.ro/2010/09/scala-generators.html.

[16] J. McBeath. Nondeterministic Evaluation in Scala, . URL http://jim-mcbeath.blogspot.ro/2010/11/nondeterministic-evaluation-in-scala.html.

[17] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[18] Oracle. Oracle. swing's threading policy. URL http://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html#threading.

[19] M. Robinson and P. Vorobiev. *Swing*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 1999. ISBN 1884777848.

[20] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44(9):317–328, Aug. 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596596. URL `http://doi.acm.org/10.1145/1631687.1596596`.

[21] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Modularity '14 - 13th International Conference on Modularity - 2014*. ACM Press, Apr. 2014.

[22] Typesafe. Akka — dataflow concurrency, . URL `http://doc.akka.io/docs/akka/2.3-M1/scala/dataflow.html`.

[23] Typesafe. The reactive manifesto, . URL `http://www.reactivemanifesto.org/`.