Preprint version 1.1

Lafros MaCS Programmable Devices: case study in type-safe API design using Scala

Rob Dickens

Latterfrosken Software Development Limited, 32 Bradford St, Walsall, West Midlands, UK, WS1 3QA
rob.dickens@lafros.com

Abstract

Lafros MaCS is an experimental Scala API for building distributed monitoring and control systems, and features reusable software modules known as Programmable Devices (PDs). Each such module provides a type-safe API for doing such things as creating a device-interface instance or writing a program to control the device. Starting with the simplest of examples, and progressing to a case study of the MaCS PD framework, used for defining PD modules, the paper explains how this type-safety is achieved, and how the various language features which Scala has to offer can assist. It is concluded that these features have not only been shown to be effective in enabling the PD framework to be written, but also that they finally make the approach of using PDs for monitoring and control practical.

Keywords type-safety, API, Scala, monitoring and control, distributed systems

1. Introduction

This paper presents the design of the Lafros MaCS Programmable Device (PD) framework, used for defining PD modules, each having its own type-safe API. It is intended to complement an earlier, less computer-science-oriented one [1] which focused on comparing the resulting API of an example PD (steerable antenna) with the corresponding one obtained using the framework's predecessor (part of JMaCS [2]), written in Java rather than Scala.

The design is presented as a case study intended for consideration alongside those[1] carried out by Odersky and Zenger in their *Scalable Component Abstractions* paper [3], whose Scala idioms were the basis of those to be found here.

[1] Observer pattern component and Scala compiler structure

Following an introduction to what PDs are, the paper first explains what is meant by a 'type-safe' API, and how one may be constructed. It then introduces the programming language features offered by Scala which can assist. This is then followed by a case study of the PD framework, demonstrating their use.

The benefits of using Scala to write the PD framework are then reflected upon to conclude.

2. Lafros MaCS Programmable Devices

Lafros MaCS [1] is an experimental API for building distributed monitoring and control systems. It is a Scala version of a Java API [2] derived from experimental software developed to monitor and control a radar used for observing the Earth's ionosphere [4].

At each node of the distributed target is a device interface (DI) client having a domain.like.name, thereby resulting in a logical hierarchy. The core API provides facilities for creating DI clients, given a basic plug-in (DI Driver) that will execute commands received, and generate status samples on specified global boundaries.

Such DI-Driver plug-ins are rather low-level and not very reusable. Enter, the Programmable Device (PD) framework, providing facilities for defining high-level, reusable modules, each with its own built-in methods for creating a corresponding DI client.

These methods work by creating a DI-Driver adapter which uses, where appropriate, instances of command-interpreter and status-factory classes extending built-in abstract ones. The adapter is also capable of running programs, which are instances of classes extending a further built-in abstract one.

By 'built-in' is meant, specific to the PD module in question, and where the resulting API is *type-safe*.

3. Type-safety using inner classes

Type-safety is a property of programs written in statically-typed languages. Such programs have variables with static (fixed) types, so that the types of the values assigned to them may be checked by a compiler.

A type-safe API is one which does not require the types of values to be coerced (from more general to more specific), in either client or API code, since here the types involved cannot be checked by the compiler:

```
// client or API code
m(moreSpecialised)

// API or client code
def m(arg: MoreGeneral) {
  ...
  moreSpecialised =
    arg.asInstanceOf[MoreSpecialised]
  ...
}
```

One way to construct such APIs is to have required types appear to a ‘family’ of inner (nested) classes as either type parameters or abstract members of the container class:

```
class Container[T] {
  class Inner {
    def m(arg: T) {...}
    ...
  }
  ...
}
```

In this way, the API may be ‘instantiated’ for desired types by instantiating the container class:

```
val api4String = new Container[String]
val inner1 = new api4String.Inner
inner1.m("1") // must be a String

val api4Int = new Container[Int]
val inner2 = new api4Int.Inner
inner2.m(1) // must be an Int
```

Note also that, even though the same inner class is involved, `inner1` and `inner2` have different, so-called *path-dependent* types. This affords a further degree of type-safety where multiple APIs as created above appear in the same code:

```
var inner3 = inner1
inner3 = inner2
      // ^ compilation error: type mismatch
```

4. Benefits of Scala language features

4.1 Singleton objects

These can provide a globally accessible singleton instance of the container class with which to expose the inner classes, more conveniently than would be possible in Java for example:

```
package apis
...
object api4String extends Container[String]
// Java:
// public class Singletons {
//   public static Container<String>
//     api4String = new Container<String>();
// }

package other
import apis.api4String.Inner
// Java:
// import apis.Container;
// import static apis.Singletons.api4String;

val inner = new Inner
// Java:
// final Container<String>.Inner inner =
//   api4String.new Inner();

inner.m("1") // must be a String
```

4.2 Abstract types

These may be used as an alternative to container class type-parameters.

In the case of multiple types, abstract types avoid the need to remember the order in which the type parameters would need to be supplied when instantiating the container class:

```
class Container[H, L] {
...
object api extends Container[Low, High] // oops!

class Container {
  type H
  type L
...
object api extends Container {
  type L = Low  //
  type H = High // okay
}
```

Abstract types can also accept inner classes as type bounds, whereas they would not be in scope in the case of type parameters. Doing this addresses certain limitations that would otherwise be encountered if instances of one inner class need to be passed to methods of another:

```
class Container {
  class Inner1
  class Inner2 {
    def m(arg: Inner1) {}
  }
}

object api1 extends Container {
  class Inner1 extends super.Inner1 {
    def newMethod() {}
  }
  class Inner2 extends super.Inner2 {
    override def m(arg: Inner1) {
              // ^ compilation error:
              //   overrides nothing!
      arg.newMethod()
    }
  }
}
```

The error arises because Inner1 now refers to a different type to the one referred to in the method intended to be overridden. Using Inner1 as an upper bound to a corresponding abstract type provides a solution:

```
class Container {
  type I1 <: Inner1
  class Inner1
  class Inner2 {
    def m(arg: I1) {}
  }
}

object api extends Container {
  type I1 = Inner1
  class Inner1 extends super.Inner1 {
    def newMethod() {}
  }
  class Inner2 extends super.Inner2 {
    override def m(arg: I1) {
      arg.newMethod()
    }
  }
}
```

This technique, which removes such limitations on 'members of the family' referring to each other is known as *family polymorphism* [5].

4.3 Self-types and traits

One further limitation would be encountered if Inner1 wished to pass its self-reference (this) to another 'family member':

```
class Container {
  type I1 <: Inner1
  class Inner1 {
    def createInner3 = new Inner3(this)
    //                              ^
    // compilation error: type mismatch
  }
  class Inner2 {
    def m(arg: I1) {}
  }
  class Inner3(val arg: I1)
}
```

Here, Scala allows Inner1 to set the type of its self-reference, or self-type, to that of the abstract type, which provides the solution:

```
class Inner1 {
  this: I1 =>
  def createInner3 = new Inner3(this)
}
```

Where it would be inconvenient or impractical to put all the inner classes into a single container class, additional traits can be used, perhaps in separate files, provided they are mixed into its self type. The self types of those traits should also be set so as to accommodate any interdependencies, with circular ones being allowed.

5. Case study: Programmable Device framework

5.1 Requirements

Our objective is to design a container class having certain type parameters and/or abstract type members, such that an instance would expose a type-safe API to support doing the following:

- writing a controls-GUI class - to be instantiated and displayed by UI clients, to submit commands interactively (interactive control)
- writing a command-interpreter class - to be instantiated by the DI client, to interpret and execute the commands submitted
- writing a status-factory class - to be instantiated by the DI client, to generate status samples
- writing a monitor-GUI class - to be instantiated by UI clients, to display the status samples (interactive monitoring)
- creating a device-interface instance
- writing program classes - to be instantiated by UI or DI clients, and submitted as commands to run the program (programmatic monitoring and control)[2].

5.2 Types required as parameters or abstract members

In order that a PD's properties may be sampled efficiently, it is advantagous to consider those properties which are constant, separately. This suggests having two types, StatusType and ConstantsType, both having java.io.Serializable as an upper bound.

For control, we would also require a DriverType, this time using AnyRef as an upper bound.

With no intuitive way to order these as type parameters, it makes better sense to use abstract types in all cases:

```
abstract class Pd {
```

[2] Note that programmatic monitoring and control of one device-interface by another is also possible.

```
  type StatusType <: Serializable
  type ConstantsType <: Serializable
  type DriverType <: AnyRef
  ...
}
```

5.3 But not all PDs have status and constants

In order to support PDs which might not have any status or constants, it is possible to put StatusType and ConstantsType in their own respective traits, which may then be mixed-in as required:

```
abstract class AnyPd {
  type DriverType <: AnyRef
  ...
}

trait StatusEtc {
  type StatusType <: Serializable
  ...
}

trait ConstantsEtc {
  type ConstantsType <: Serializable
  ...
}

...
abstract class ConstantlessPd
  extends AnyPd with StatusEtc {
  ...
}
...
```

A PD module having no constant properties would therefore be defined by extending ConstantlessPd, which does not require a ConstantsType:

```
package org.myorg.macspd.cat.mydevice
import com.lafros.macs.pd.ConstantlessPd

object pd extends ConstantlessPd {
  type DriverType = Driver
  type StatusType = Status
}

trait Driver {...}
trait Status extends Serializable {...}
```

Note the convention of inserting .macspd.cat. into the package name, to indicate the base of a 'catalogue' of MaCS PDs, and of naming the instance of the PD container class simply, pd.

Six such PD container classes are provided in all, as shown in Table 1. Note that a DriverContainer trait, having an abstract val driver field, is required in certain cases, such as when the macs.OtherDiDependent trait is to be mixed in, which it would not be appropriate to mix into the driver itself.

Container class	Status	Constants
Pd	*	*
ConstantlessPd	*	
DriverOnlyPd DriverContainerOnlyPd		*
DriverOnlyConstantlessPd DriverContainerOnlyConstantlessPd		

Table 1. PD container classes

5.4 Controls GUI

Even though we now have a DriverType, which could be implemented as a proxy, it turns out not to be practical to call its methods directly, interactively. Rather, the driver is designed to be accessed from programs or a command-interpreter. Therefore, the nested trait provided simply extends the one provided by the core API (where any Serializable object may be submitted as a command), and serves only to make it easier to give the implementation class the name ControlsGui as well:

```
package com.lafros.macs.pd

abstract class AnyPd {
  ...
  trait ControlsGui extends macs.ControlsGui
}

package org.myorg.macspd.cat.mydevice
import com.lafros.macs.pd....Pd

object pd extends ...Pd {...}
...
class ControlsGui extends pd.ControlsGui {...}
```

Note that the core API first looks for GUI constructors having a single argument, in which case any constants object found will be supplied.

5.5 Command interpreter

This is modelled as a function taking, in addition to other arguments including the command, a context object whose type has a nested trait as an upper bound, therefore requiring an abstract type:

```
abstract class AnyPd {
  type CmdInterpreterContextType
    <: AnyCmdInterpreterContext
  ...
  protected trait AnyContext {
    val driver: DriverType
  }
  protected trait AnyCmdInterpreterContext
    extends AnyContext {...}
  ...
  trait CmdInterpreter extends
  Function4[Serializable, Boolean,
            Option[String],
            CmdInterpreterContextType,
            Option[Serializable]] {
    protected type Context =
      CmdInterpreterContextType
  }
}
```

The CmdInterpreterContextType is then supplied by the PD container class (see Table 1):

```
abstract class ConstantlessPd extends AnyPd
  with StatusEtc {
  type CmdInterpreterContextType =
    CmdInterpreterContextImp
  private[pd] abstract class
    CmdInterpreterContextImp(progRnr: ProgRnr)
  extends AnyCmdInterpreterContextImp(progRnr)
    with CmdInterpreterContext
  protected trait CmdInterpreterContext
    extends AnyCmdInterpreterContext
    with StatefulContext
  ...
}
```

Our constantless PD (as defined in §5.3) could then define its command interpreter as follows:

```
object cmds {
  val launch = "launch"
  ...
}
class CmdInterpreter
  extends pd.CmdInterpreter {
  def apply(cmd: Serializable,
            control: Boolean, // accept
            // commands that make changes?
            diName: Option[String],
            context: Context) = {
    val recognised =
      if (control) cmd match {
        case cmds.launch =>
          context.driver.launch(); true
        ...
      }
      else false
    if (recognised) None
    else throw
      new macs.CmdNotRecognisedException(cmd)
  }
}
```

5.6 Status factory

Here we must extend a non-nested trait used by the PD to DI-Driver adapter (mentioned in §2) so as to make it PD-specific:

```
private[pd] trait AnyStatusFactory {
  val driver: AnyRef
  def status: Serializable
}

abstract class AnyPd {
  type DriverType <: AnyRef
  ...
  trait DriverContainer {
    val driver: DriverType
  }
}

trait StatusEtc {
  this: AnyPd =>
  type StatusType <: Serializable
  ...
  trait StatusFactory extends AnyStatusFactory
    with DriverContainer {
    def status: StatusType
  }
}
```

Note how the self type of StatusEtc is set to the type of the generic PD, in order to satisfy the dependency on DriverContainer.

Our constantless PD could then define its status factory as follows:

```
class StatusImp(...) extends Status {...}
class StatusFactory extends pd.StatusFactory {
  private var _launched = false
  ...
  val driver = new Driver {
    def launch() {
      ...
      _launched = true
    }
  }
  def status = new StatusImp(_launched, ...)
}
```

5.7 Monitor GUI

Here we extend the trait specified by the core API so as to make it type-safe:

```
trait StatusEtc {
  type StatusType <: Serializable
  ...
  trait MonitorGui extends macs.MonitorGui {
    final def refresh(status: Any) =
      refresh(status.asInstanceOf[StatusType])
    def refresh(status: StatusType)
  }
}
```

Note that coercion is permissible here, since the value in question is being supplied by the MaCS implementation, and therefore known to be of the correct type. A given pd could then define its monitor GUI as follows:

```
class MonitorGui extends pd.MonitorGui {
  private val launched_lab = ...
  ...
  val component = new FlowPanel {
    ...
    contents += launched_lab
  }
  def refresh(status: Status) {
    launched_lab.text =
      if (status.launched) "true" else "false"
    ...
  }
}
```

5.8 Device-interface creation

Here we add methods to the PD container classes themselves:

```
abstract class ConstantlessPd
  extends AnyPd with StatusEtc {
  ...
  def createDi(name: String,
statusFactoryClass: Class[_ <: StatusFactory]
): Di = {...}
  def createDi(name: String,
               statusFactory: StatusFactory
): Di = {...}
}
```

Components such as the GUIs and the command interpreter, which are common to all device-interface instances, need not be supplied as arguments, and may be instantiated on demand using `java.lang.Class.forName` and a class naming convention.

Device interfaces may then be instantiated and registered with the monitoring and control system in a type-safe way as follows:

```
import org.myorg.macspd.{cat, sims}
val di = cat.mydevice.pd.createDi(
 "mysimulator.mydevice",
  classOf[sims.mydevice.StatusFactory])
di.register()
```

5.9 Programs

These extend a ‘tag’ class, so that they may be identified by the PD to DI-Driver adapter via pattern-matching:

```
object cmds {
  ...
  abstract class Program extends Serializable
}

class Adapter extends macs.Di.Driver ... {
  def interpretCmd(cmd: Serializable, ...) {
    if (control) cmd match {
      case program: cmds.Program =>
        if (isProgramForThisPd(program))
          programRunner.startProgram(program)
      ...
```

One consequence of introducing a type-safe inner class that is `Serializable` is that our container class must be made `Serializable` too:

```
trait AnyProgramContext {...}
abstract class AnyPd extends Serializable {
  type ProgramContextType <: AnyProgramContext
  abstract class Program
    extends cmds.Program {
    ...
    protected type Context = ProgramContextType
    def init(context: Context) {}
    def complete(context: Context): Boolean
    ...
  }
}
```

Once again, the PD-specific types are incorporated using a context object whose type is specified using an abstract type. Although the type’s upper bound is not this time a nested class itself, the values we assign to it in each container class must themselves be nested, so a type parameter could still not have been used:

```
abstract class ConstantlessPd
  extends AnyPd with StatusEtc {
  ...
  type ProgramContextType = ProgramContextImp
  ...
  protected trait ProgramContext
    extends AnyProgramContext
    with StatefulContext
  private[pd] abstract class
ProgramContextImp(progAlertsAcc: ProgAlertsAcc)
    extends AnyProgramContextImp(progAlertsAcc)
    with ProgramContext
}
```

A program for our constantless PD could then be written as follows:

```
package org.myorg.macspd
package progs.mydevice
import cat.mydevice.pd.Program

class LaunchMonitor extends Program {
  def complete(context: Context) =
    if (context.status.launched) {
      context.addAlert("launched!", true)
      true
    }
    else false
}
```

6. Conclusion

The Lafros MaCS monitoring and control system has been introduced, and its concept of Programmable Devices (PDs) explained. The type-safe aspect of the API resulting from each PD module definition was highlighted, and followed by an explanation of what this means and how it may be achieved using inner classes. The various language features which can be brought to bear if implementing a framework for defining modules having type-safe APIs in Scala were then enumerated, and this procedure was then demonstrated in the case of the MaCS PD framework. The procedure is based on the Scala idiom of creating a singleton instance of a class having inner classes referring to certain type parameters and/or abstract type members.

Not only has using Scala as the implementation language enabled the PD framework to be written so that the APIs of the modules defined are type-safe, but it has resulted in a framework which should finally be practical to use, in comparison [1] with an earlier version written in Java.

Acknowledgments

The explanation of family polymorphism (given in §4.2) was informed by that provided by Fredrik Skeel Løkke in §3.4.1 of his Masters thesis[3] on ‘Scala & Design Patterns’.

References

[1] R. Dickens. *Lafros MaCS: an experimental Scala monitoring and control API*, 14th International EISCAT Workshop poster (Aug, 2009)[4].

[2] R. Dickens. *JMaCS: a Java monitoring and control system*, Proc. of SPIE Vol. 7019, 7019W (2008)[5].

[3] Martin Odersky and Matthias Zenger. *Scalable Component Abstractions*, OOPSLA (Oct 2005)[6].

[4] Röttger, J., U.G. Wannberg and A.P. van Eyken. *The EISCAT Scientific Association and the EISCAT Svalbard Radar Project*, J. Geomag. Geoelectr., 47, 669-679 (1995).

[5] Erik Ernst. *Family polymorphism* In Jørgen Lindskov Knudsen, editor, Proceedings ECOOP 2001, LNCS 2072, pages 303–326, Heidelberg, Germany, Springer-Verlag (2001).

[3] `http://www.scala-lang.org/sites/default/files/FrederikThesis.pdf`

[4] `http://lafros.com/macs/preprint.pdf`

[5] `http://jmacs.org/jmacsSpie08pre.pdf`

[6] `http://http://www.scala-lang.org/sites/default/files/odersky/ScalableComponent.pdf`