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

Rob Dickens

// Latterfrosken
software.development(limited);

Walsall, West Midlands, UK

Table of contents

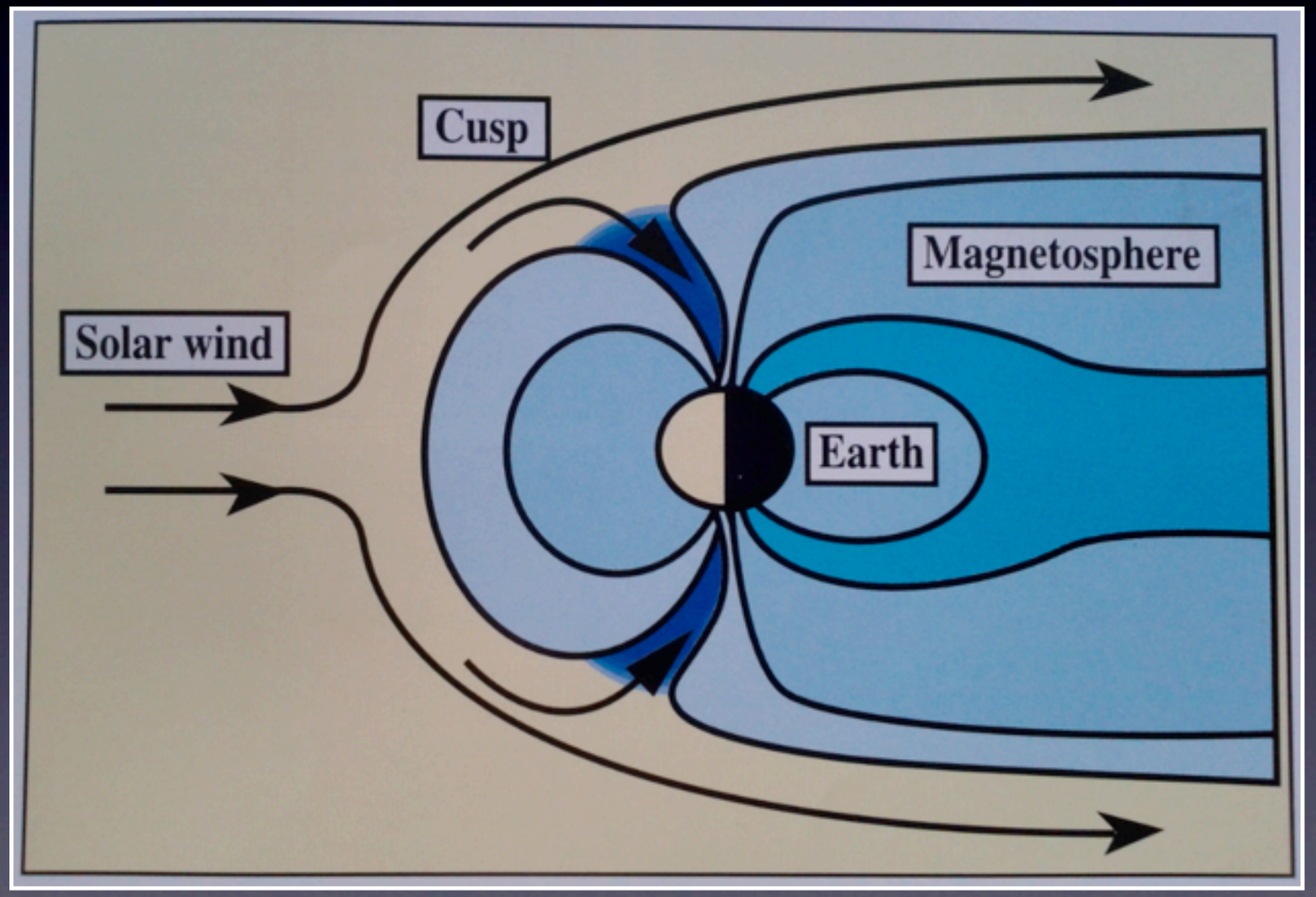
1/5 Lafros MaCS

- experimental Scala API for building distributed monitoring and control systems
  - JVM to JVM
    - transfer real Java objects
  - rewrite of existing Java API (JMaCS), in turn derived from software written for a research radar...

EISCAT Svalbard Radar (ESR)

- EISCAT = European Incoherent SCATter
- for observing Earth's ionosphere
- novel (1995) distributed architecture
- remote location...

Digression!



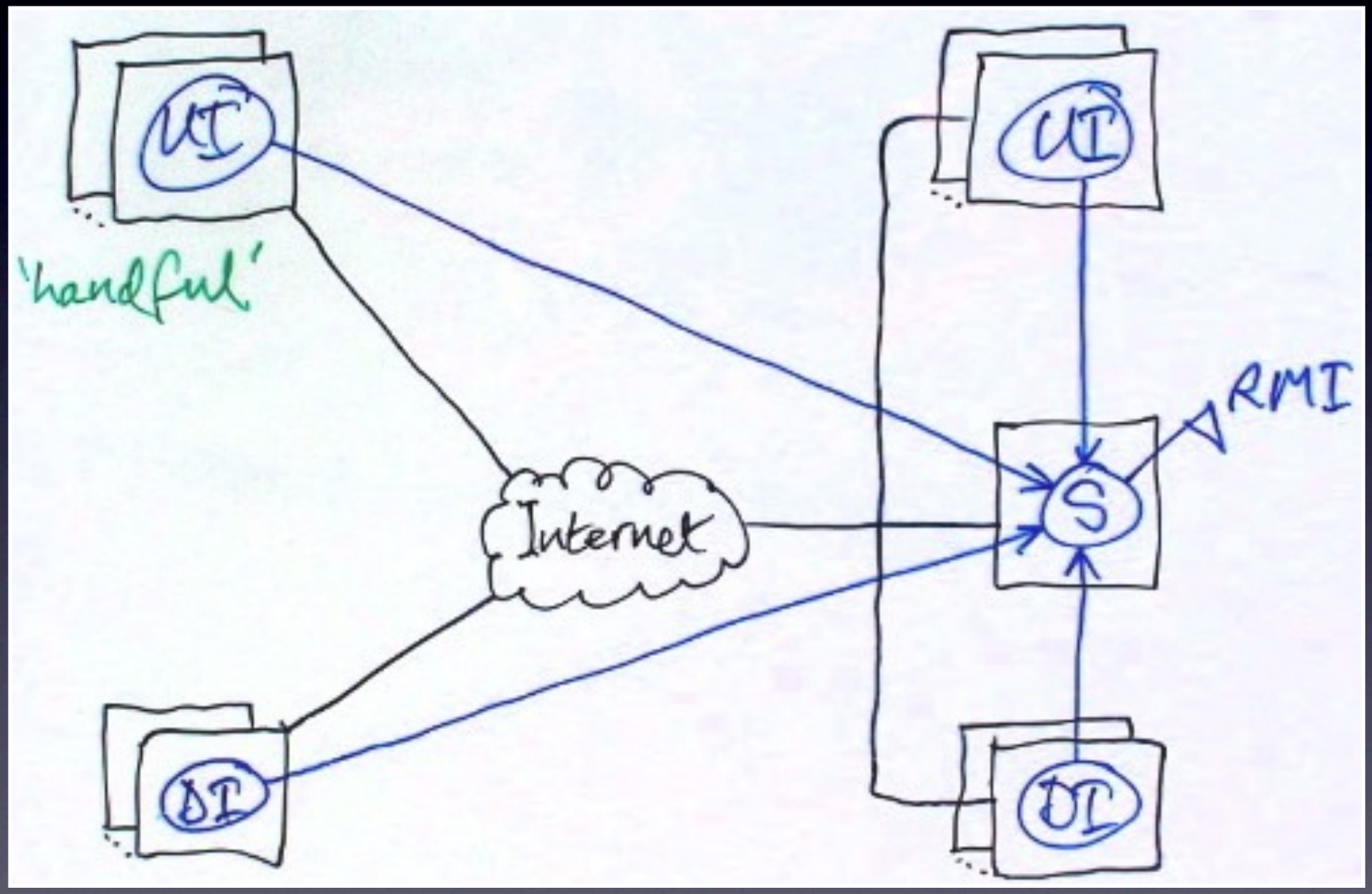
Original Java system in use Oct 00 - Mar 02

- problems implementing it on other radars
- scientists seemed to prefer scripts and C to Java
  - hence superseded by Tcl system
- JNI challenging!
- remote access not universally welcomed

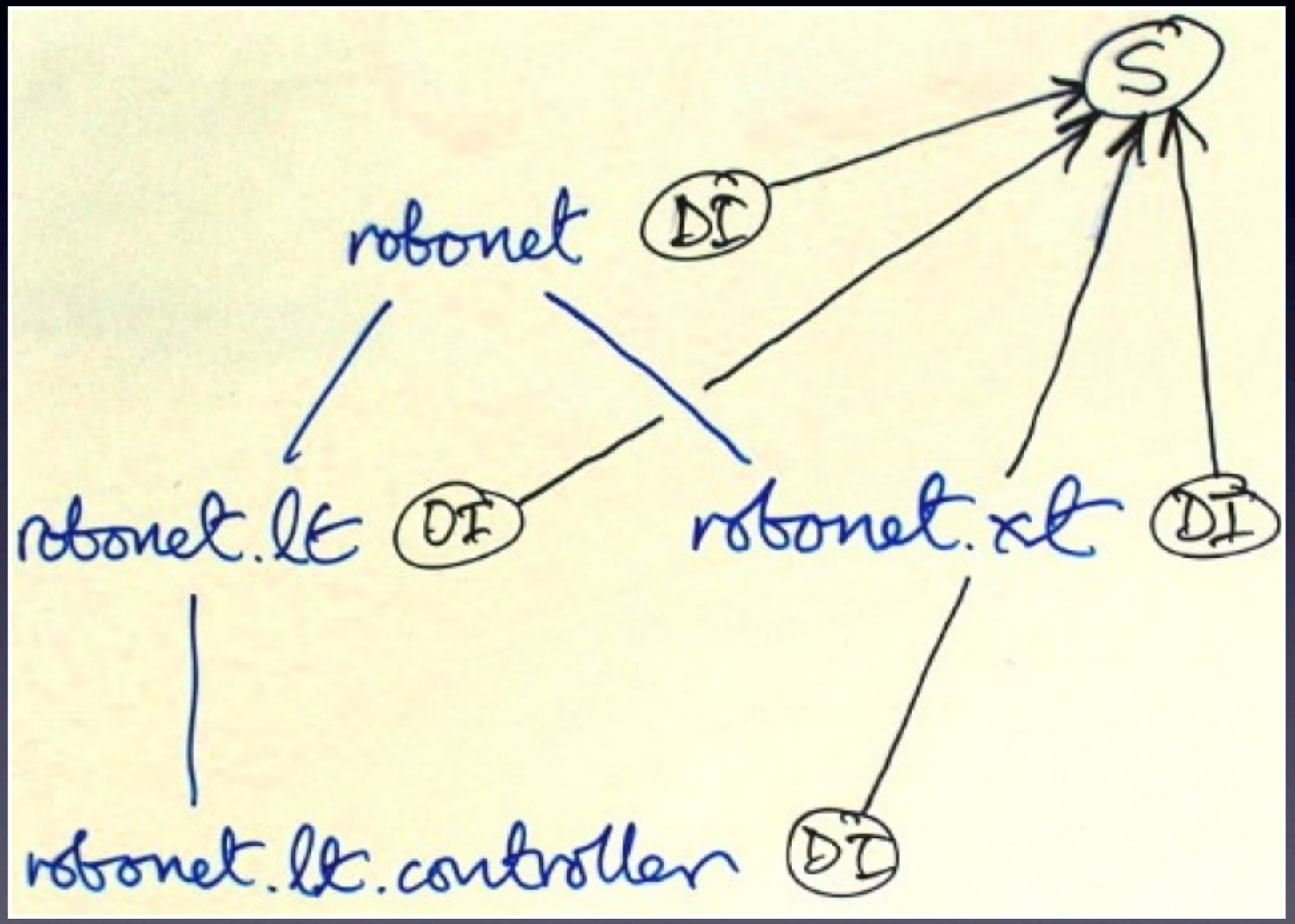
Should have given up...

- ...but foolishly carried on with the development independently
- Scala offers new hope!

Simple client-server architecture



Domain.like.names of DI clients establish logical hierarchy



DI clients require DI-Driver plug-ins

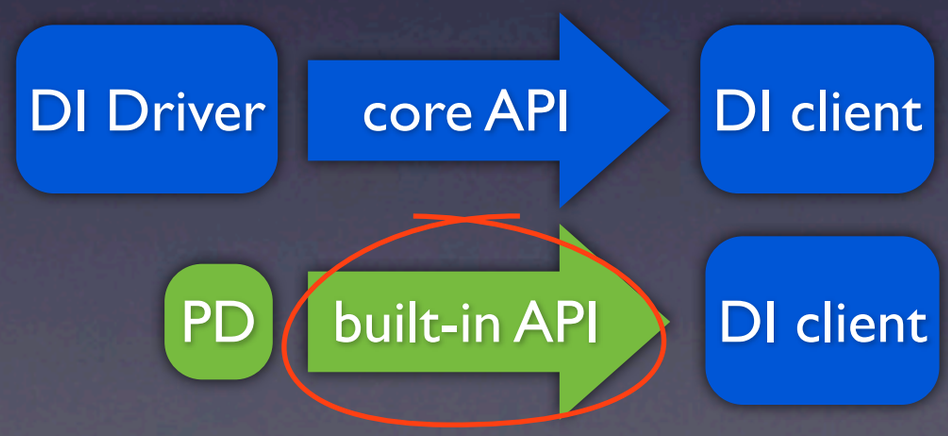
- simply execute commands received, and generate status samples on specified global boundaries

- low-level
- non-reusable
- non-programmable

```
object Di {
  ...
  trait NonPollableDriver {
    def interpretCmd(cmd: Serializable,
                     control: Boolean,
                     diName: Option[String]): Option[Serializable]
  }
  trait Driver extends NonPollableDriver {
    def status: Serializable
  }
  ...
```

2/5 Programmable Devices (PDs)

- high-level, reusable modules
  - ‘high-level’ as in e.g. steerable antenna
- defined using Lafros MaCS PD framework
- built-in method for creating the DI client



Programmable Devices also... programmable

- DI client creation method also creates a DI-Driver adapter
- this uses instances of command-interpreter and status-factory classes extending built-in abstract ones
- adapter capable of running programs - instances of classes extending a further built-in abstract class

And last but not least...

...a PD's 'built-in' API is *type-safe*

```
package macs

class Driver
abstract class Context {
  val driver: Driver
  ...
}
abstract class Program {
 def invoke(context: Context)
...
```

```
package mydevice
package myprogs
import macs._

class MyProgram extends Program {
  def invoke(context: Context) {
    val driver = context.driver.asInstanceOf[MyDriver]
    driver.doSomething()
    //
    // type-safe means being able to do this
    //context.driver.doSomething()
  }
```

unsafe!

3/5 Type-safe API design

- can engineer type-safety using inner classes
- desired types appear as either type parameters or abstract members of the container class

Simplest example

- type-safe APIs ‘instantiated’ for desired types, by instantiating container class
- even though same inner class involved, `inner1` and `inner2` have different (*path-dependent*) types

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

```
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
```

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

So ‘type-safe API design’ is really container-class design

- need to be able to compose container classes such that their singleton instances provide the desired type-safe APIs
- e.g. the Lafros MaCS PD framework includes several abstract container classes whose singleton instances represent PDs

4/5 How key Scala language features assist

- singleton objects
- abstract types
- self-types and traits

Singleton objects

- convenient way (cf. Java) to expose the inner classes to global access

```
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
```

Abstract types 1/3

- alternative to type parameters, for use where,
  - have multiple types with no intuitive order
  - need to use an inner class as a type bound

```
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
}
```

```
class Container[T <: Inner] {
                   // ^ not in scope!
  class Inner
  ...
```

```
class Container {
  type T <: Inner // okay
  class Inner
  ...
```

Abstract types 2/3

- abstract types also address certain limitations that would otherwise be encountered if instances of one inner class need to be passed to methods of another
- `Inner1` now refers to a different type to the one referred to in the method intended to be overridden!

```
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()
    }
  }
}
```

Abstract types 3/3

- using `Inner1` as an upper bound to a corresponding abstract type provides a solution

- allowing 'family members' to refer to each other in this way known as *family polymorphism*

```
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()
    }
  }
}
```

Self-types and traits I

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

- 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 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)
}
```

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

Self-types and traits 2

- where 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're mixed into the container class's self type
- the self types of those traits should also be set so as to accommodate any interdependencies, with circular ones being allowed

5/5 Design of the Lafros MaCS PD framework (case study)

- begin with design of container class
- requires certain type parameters and/or abstract types, such that an instance would expose a type-safe API to support...

Case study: API required to support...

- writing controls-GUI class - instantiated and displayed by UI clients
- writing command-interpreter class - instantiated by DI client
- writing status-factory class - instantiated by DI client to create status samples
- writing monitor-GUI class - instantiated by UI clients, to display status samples
- creating DI client
- writing program classes - instantiated by UI or DI clients for submission as commands

Case study: type-param/ abstract-type requirements

- want to sample PD properties efficiently
  - consider constant ones separately
- require driver for control
- no intuitive order!

```
abstract class Pd {
  type StatusType <: Serializable
  type ConstantsType <: Serializable
  type DriverType <: AnyRef
  ...
}
```

Case study: but not all PDs have status and constants! 1/3

- so put `StatusType` and `ConstantsType` in their own respective traits
  - can then mix them 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 {
  ...
}
...
```

Case study: but not all PDs have status and constants! 2/3

- a PD module having no constant properties would then 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 {...}
```

Case study: but not all PDs have status and constants! 3/3

- Six such container classes are included

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

Example PD: minimalistic steerable antenna

```
package com.lafros.macspd.cat.antenna.steerable
import java.io.Serializable

object pd extends com.lafros.macs.pd.Pd {
  type ConstantsType = Constants
  type DriverType = Driver
  type StatusType = Status
}

trait Driver {
  var az: Double
  var el: Double
  var dir: (Double, Double)
}

trait Status extends Serializable {
  def az: Double
  def el: Double
}

trait Constants extends Serializable {
  val azVel: Double
  val elVel: Double
  val minAz: Double
  val maxAz: Double
  val minEl: Double
  val maxEl: Double
}
```

Example PD: minimalistic steerable antenna

```
package com.lafros.macspd.cat.antenna.steerable
import java.io.Serializable

object pd extends com.lafros.macs.pd.Pd {
  type ConstantsType = Constants
  type DriverType = Driver
  type StatusType = Status
  implicit val cm: ClassManifest[ConstantsType] = implicitly // Scala 2.8
}

trait Driver {
  var az: Double
  var el: Double
  var dir: (Double, Double)
}

trait Status extends Serializable {
  def az: Double
  def el: Double
}

trait Constants extends Serializable {
  val azVel: Double
  val elVel: Double
  val minAz: Double
  val maxAz: Double
  val minEl: Double
  val maxEl: Double
}
```

Update: no longer required
(Thank you, Martin)

Case study: controls-GUI component 1/2

- after all that... no type-safety benefits in this case!
  - not practical to call `DriverType` methods interactively, if implemented it as a proxy
- hence continue to submit commands explicitly, as though implementing for the DI-Driver plug-in directly

Case study: controls-GUI component 2/2

- even so, still add an inner class which just extends the core API class
- provides convenient and conventional access when implementing

```
package com.lafros
package 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 {...}
```

Case study: command-interpreter component 1/3

- modelled as a function
- taking 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
  }
}
```

Case study: command-interpreter component 2/3

- an appropriate `CmdInterpreter-ContextType` is then supplied by each PD container class

```
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
  ...
}
```

Case study: command-interpreter component 3/3

- a PD module might then define its command interpreter, type-safely, 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)
  }
}
```

Case study: status-factory component 1/2

- here, have to extend a non-nested trait used by the PD to DI-Driver adapter, so as to make it PD-specific
- note how the self type of `StatusEtc` is set to `AnyPd`, so as to satisfy the dependency on `DriverContainer`

```
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
  }
}
```

Case study: status-factory component 2/2

- a PD might then define its status factory, type-safely, 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, ...)
}
```

Case study: monitor-GUI component 1/2

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

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

Case study: monitor-GUI component 2/2

- a PD could then define its monitor GUI, type-safely, 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"
    ...
  }
}
```

Case study: device-interface creation 1/2

- 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 = {...}
}
```

- other components (GUIs, command interpreter), instantiated using `java.lang.Class.forName`

Case study: device-interface creation 2/2

- 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()
```

Case study: programs 1/4

- 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)
      ...
```

Case study: programs 2/4

- a `Serializable` inner class requires a `Serializable` container one!

```
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
    ...
  }
}
```

- and ProgramContextType is still an abstract type rather than a type parameter...

Case study: programs 3/4

- ...since the values we assign to it in each container class must be nested

```
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
}
```

Case study: programs 4/4

- a program for our ConstantlessPd may then be written, type-safely, as follows
- cf. slide 13!

```
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
}
```

Conclusion

- Lafros MaCS PDs,
  - introduced
  - used to demonstrate type-safe API design using Scala
- Scala language features key to finally making the MaCS PD approach practical!

Thanks for your attention!