Lafros MaCS: an experimental Scala monitoring and control API

Rob Dickens

```
// Latterfrosken
software.development(limited);
```

Introduction

The Lafros MaCS (Monitoring and Control System) API is designed to help facilitate the local or remote, interactive and programmatic, monitoring and control of a distributed target in soft realtime.

It does not provide all these facilities by itself, but rather defines a standard way to plug abstract devices into an abstract system that may be implemented by a third party.

It is a complete rewrite of an existing Java API, JMaCS[1], in Scala[2].

JMaCS

JMaCS is derived from experimental software[3][4] first developed for the ESR.

It sets out to exploit the full potential of an all-Java system of distributed objects (as opposed to a heterogeneous one), which is able to exchange Java objects themselves:

- should be more capable than if restricted to exchanging passive data;
- should be more efficient than converting data to/from an external format.

Thus, in JMaCS,

- an instance of a Java class representing a command or programmable device (PD) program can be created and configured in a user interface (UI)-client object, and sent to a device interface (DI)-client one, with full propagation of any exception thrown in the latter; DI clients may also send such objects to each other;
- instances of a Java class representing status samples are created in the DI-client object, and sent to UI client and other DI-client ones;
- a Java class representing a monitor or control-panel GUI is named in the DI-client object, for subsequent instantiation and use in UI-client ones.

JMaCS API weaknesses

PDs are reusable definitions of devices, complete with monitor and control-panel GUIs, command interpreter, and API (whose implementation may be separate and in hardware):

- it is usually easier to implement one of these rather than a DI-client plug-in directly;
- as their name suggests, they are also programmable.

JMaCS PD definitions have the following weaknesses:

- the name of each PD is that of its package; however, it is necessary to duplicate the rightmost portion of this when naming the container class that is required;
- the container class's `IConstants`, `IDriver` and `IStatus` interfaces must extend respective tag interfaces, which is somewhat verbose;
- a proxy class has to be defined, but supplies no additional information about the PD's API;
- the `Interpreter` and `MonitorGui` classes, and all program ones are not fully type-safe, being passed arguments having tag interface types (or type `Object`), which must be cast to the appropriate PD ones.

It was with the above in mind that consideration was given to rewriting JMaCS in a new language.

Scala

It is no longer the case that programs targeting the Java platform must be written in the Java language, and among the alternatives is Scala. Apart from its being statically-typed and fully interoperable with Java, the following attributes were what first made this one appealing:

- better support for writing components (such as PDs), thanks to a form of multiple implementation inheritance involving traits;
- tighter and more concise syntax, thanks to its implicitly final method-parameters, and type inference;
- support for an alternative style of concurrency (to that based on Java's synchronized blocks), in the form of actors.

However, Scala soon turned out to have other attractions:

- a novel combination of features in support of writing fully type-safe code: inner-classes, type members, singleton objects, and the ability to override the type of the self reference;
- extensibility through libraries (rather than adding to the language itself), made possible in large part by the fact that all values are objects, and all operators are methods;
- full support for functional programming and closures: besides methods, functions may be defined as values of function types;
- utility also as a scripting language.

Developing in Scala

Compilation of Scala source files produces regular Java .class files, that may be packaged as regular .jar files. However,

- the Scala library .jar file is required in the classpath, for execution;
- applications require their entry point to be a `def main(args: Array[String]) {…}`, residing in a singleton object.

No significant changes were therefore required in order to switch from developing JMaCS to developing MaCS. Note, however, that special measures will be required when developing downloadable apps, if the entire Scala library is not to be downloaded as well.

Use of Scala idioms

As previously mentioned, it was the part of the JMaCS API to do with defining PDs, namely the org.jmacs.pd package, that was most in need of benefiting from being rewritten in Scala, and this indeed turned out to be where most opportunities to employ Scala idioms arose. The design pattern found to be of greatest value here is one (based on the subject/observer case study of [5]) which will be called the *type-safe singleton*, where inner classes referring to abstract type-members are exposed via a singleton object. This was employed as follows:

- all elements of the PD are represented by inner classes/traits of an abstract generic container class, `_AnyPd`, also having various abstract type-members to which the inner classes/traits may refer;
- `_AnyPd` has various abstract subclasses representing six PD variants (according on whether or not they produce status or define constants), some of which supply or else 'fine tune' the required type members;
- PD developers then define a singleton object (conventionally called `pd`, in a package whose name corresponds to that of the PD) which extends the desired `_AnyPd` variant; being concrete, it is required to 'fill-in' the remaining abstract type-members, which are the traits (representing such things as the driver) which define that particular PD's API;
- developers may then define any further classes constituting the PD's definition (such as its monitor GUI) in a fully type-safe way, by extending the corresponding inner class of `_AnyPd`, which is now accessible via the singleton object, `pd`.

JMaCS vs MaCS: PD API definition

We now present a PD definition for a steerable antenna (such as the one at the ESR), having only a very minimal API. Shown below is the Java code necessary to define the PD's API in the case of JMaCS (left),

```java
package com.lafros.jmacs.pd.cat.antenna.steerable;
import org.jmacs.IDi;
import org.jmacs.pd.Device;
public class Steerable implements java.io.Serializable {
  public static final long serialVersionUID = 1;
  public interface Cmds {
    String az = "az";
    String el = "el";
    String dir = "dir";
  }
  public interface IConstants extends Device.IConstants {
    double minAz();
    double maxAz();
    double azVel();
    double minEl();
    double maxEl();
    double elVel();
  }
  public interface IDriver extends Device.IDriver {
    void az(double val);
    void el(double val);
    void dir(double az, double el);
  }
  public interface IStatus extends Device.IStatus {
    double az();
    double el();
  }
  public static class Proxy extends Device.Proxy {
    final IDriver proxyDriver = (IDriver)createProxyDriver(IDriver.class);
    public Proxy(final IDi.IDriver.IClient.IContext context) {
      super(context);
    }
    public Proxy(final IDi.IDriver.IClient.IContext context,
                 final String diName) {
      super(context, diName);
    }
    public IDriver getProxyDriver() {
      return this.proxyDriver;
    }
  }
}
```

```scala
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
}
```

next to the Scala code necessary in the case of MaCS. As can be seen, the weaknesses pointed out earlier have been eliminated in the MaCS case.

Since the API includes status, and there are commands (defined elsewhere in the MaCS case), the PD definition should also include classes representing a monitor GUI, control-panel GUI, and command interpreter...

JMaCS vs MaCS: PD command-interpreter

Shown below, side by side as before, are the two versions of the PD's command interpreter. The main difference to note here is that, by extending `pd.CmdInterpreter`, only the MaCS version (right) is fully type-safe, whereas the JMaCS one is required to cast the driver (from the tag type to the type defined by the PD). Note also that the commands are defined in the same file in the MaCS case, which is not possible in the JMaCS one. This example also illustrates how Scala's pattern-matching can provide a cleaner alternative to conditionals.

```java
package com.lafros.jmacs.pd.cat.antenna.steerable;
import java.io.Serializable;
public class Interpreter implements org.jmacs.pd.IInterpreter {
  public Serializable interpretCmd(final Serializable cmd,
                                   final Context context,
                                   final boolean control)
    throws Exception {
    if (control && cmd instanceof String) handler:{
        final Steerable.IDriver driver =
          (Steerable.IDriver)context.getDriver();
      final String cmdString = (String)cmd;
      final String[] tokens = cmdString.split("\\s");
      if (tokens[0].equals(Steerable.Cmds.az)) {
        final double val = Double.parseDouble(tokens[1]);
        driver.az(val);
      }
      else if (tokens[0].equals(Steerable.Cmds.el)) {
        final double val = Double.parseDouble(tokens[1]);
        driver.el(val);
      }
      else if (tokens[0].equals(Steerable.Cmds.dir)) {
        final double az = Double.parseDouble(tokens[1]);
        final double el = Double.parseDouble(tokens[2]);
        driver.dir(az, el);
      }
      else
        break handler;
      return null;
    }
    throw new org.jmacs.CmdNotRecognisedException(cmd);
  }
}
```

```scala
package com.lafros.macspd.cat.antenna.steerable
object cmds {
  val az = "az"
  val el = "el"
  val dir = "dir"
}
class CmdInterpreter extends pd.CmdInterpreter {
  def apply(cmd: java.io.Serializable,
            control: Boolean,
            diName: Option[String],
            context: Context) = {
    val recognised =
      if (control) cmd match {
        case s: String =>
          val tokens = s.split("\\s")
        tokens(0) match {
          case cmds.az =>
            context.driver.az = tokens(1).toDouble
          true
          case cmds.el =>
            context.driver.el = tokens(1).toDouble
          true
          case cmds.dir =>
            context.driver.dir = (tokens(1).toDouble, tokens(2).toDouble)
              true
          case _ => false
        }
        case _ => false
      }
      else false
    if (recognised) None
    else throw new com.lafros.macs.CmdNotRecognisedException(cmd)
  }
}
```

JMaCS vs MaCS: PD implementation, deployment

To complete the PD definition, it may be similarly shown how, by extending `pd.MonitorGui`, only the MaCS version of the monitor GUI is fully type-safe. Note that full type-safety with respect to the PD's API is not an issue in the case of the control-panel GUI, since this may not reference the driver directly, but only the commands.

In the present case, it is appropriate not to include an implementation of the PD's API as part of the PD definition, so as not to limit its reusability. To provide one in this case requires a status factory, which once again, by extending `pd.StatusFactory`, may only be written in a fully type-safe way in the MaCS case.

The listings below show JMaCS and MaCS versions of the code required to create a DI-client object, given a concrete implementation of our example PD. Here again, only the MaCS version (bottom) is fully type-safe.

```java
final String devicePkgname = "com.lafros.jmacs.pd.cat.antenna.steerable";
final Class driverClass =
  com.lafros.jmacs.pd.sims.antenna.steerable.SteerableStatusFactory.class;
final Device.IConstants constants =
  new com.lafros.jmacs.pd.sims.antenna.steerable.SteerableConstants();
final Pdi pdi = new Pdi("target.antenna", devicePkgname, driverClass, constants);
pdi.register();
```

```scala
import com.lafros.macspd.cat
import com.lafros.macspd.sims
val di = cat.antenna.steerable.pd.createDi("target.antenna",
                                            classOf[sims.antenna.steerable.StatusFactory],
                                            sims.antenna.steerable.constants)
di.register()
```

MaCS vs JMaCS: PD programs - 1

Shown here is the JMaCS version of an example program for our example PD, having properties that may be configured so as to point the antenna in a sequence of directions.

```
package com.lafros.jmacs.pd.progs.antenna.steerable;
import com.lafros.jmacs.pd.cat.antenna.steerable.Steerable;
import org.jmacs.ISamplingDependent;
import org.jmacs.pd.Alert;
import org.jmacs.pd.IProgram;
import org.jmacs.util.Timer;
public class CommonDwell
 implements IProgram, ISamplingDependent {
  static final long serialVersionUID = 1;
  //
  // ensure compatibility
  private final Class deviceClass = Steerable.class;

  public CommonDwell() {
    // empty--supplied for sake of doc-comment
  }

  private double[] azs = {
    90, 0, -90, 0
  };
  private double[] els = {
    90, 180, 90, 0
  };
  private long dwell_ms = 30000;
  private boolean repeat = true;
  //
  private transient Steerable.IDriver driver;
  private transient Timer timer;
  private transient int i;

  public String toString() {
    return "uses supplied az & el arrays, dwell-time";
  }
```

```
  //
  // IProgram impl'n...
  public void init(final Context context) {
    //
    // check configuration
    if (this.azs == null ||
        this.els == null ||
        this.azs.length == 0 ||
        this.els.length == 0 ||
        this.els.length != this.azs.length ||
        this.dwell_ms <= 0)
      throw new RuntimeException("badly configured!");
    //
    this.driver = (Steerable.IDriver)context.getDriver();
    this.timer = context.getTimer();
    //
    // apply dwell_ms
    timer.setPeriodMillis(this.dwell_ms);
  }

  public boolean wake(final Alert alert) {
    //
    // point antenna
    this.driver.dir(this.azs[this.i], this.els[this.i]);
    //
    this.i++;
    //
    // reset i if repeat requested
    if (this.repeat &&
        this.i == this.azs.length)
      this.i = 0;
    return this.i < this.azs.length;
  }

  public void terminate() {
    // empty
  }
  // ...IProgram impl'n

  // ISamplingDependent impl'n
  public void setSamplingParams(final Timer.Params params) {
    // empty
  }
```

```
  public double[] getAzs() {
    return this.azs;
  }

  public void setAzs(final double[] azs) {
    this.azs = azs;
  }

  public double[] getEls() {
    return this.els;
  }

  public void setEls(final double[] els) {
    this.els = els;
  }

  public long getDwell_ms() {
    return this.dwell_ms;
  }

  public void setDwell_ms(final long dwell_ms) {
    this.dwell_ms = dwell_ms;
  }

  public boolean isRepeat() {
    return this.repeat;
  }

  public void setRepeat(final boolean b) {
    this.repeat = b;
  }
}
```

JMaCS vs MaCS: PD programs - 2

```
package com.lafros.macspd.progs.antenna.steerable
import com.lafros.macs.SamplingDependent
import com.lafros.macspd.cat.antenna.steerable.pd.Program
@SerialVersionUID(1L)
class CommonDwell extends Program with SamplingDependent {
  @BeanProperty var azs = Array(90D,   0D, -90D, 0D)
  @BeanProperty var els = Array(90D, 180D,  90D, 0D)
  @BeanProperty var dwell_ms = 30000L
  @BeanProperty var repeat = true

  @transient private var i: Int = _

  override def toString = "uses supplied az & el arrays, dwell-time"
  //
  // Program impl'n...
  override def init(context: Context) {
    //
    // check configuration
    if (azs == null ||
        els == null ||
        azs.length == 0 ||
        els.length == 0 ||
        els.length != azs.length ||
        dwell_ms <= 0)
      throw new RuntimeException("badly configured!")
    //
    // configure timer
    context.timer.periodMillis = dwell_ms
  }

  def complete(context: Context) = {
    context.addAlert("test", true)
    //
    // point antenna
    context.driver.dir = (azs(i), els(i))
    //
    i += 1
    if (i == azs.length && repeat) i = 0
    i == azs.length
  }
  // ...Program impl'n

  // SamplingDependent impl'n
  def notify(params: Timer.Params) {
    // empty
  }
}
```

Shown here is the MaCS version of the same program. Once again, only this one is fully type-safe, while also being somewhat more concise.

Note that the `@BeanProperty` annotation tells the compiler to add corresponding Java-style accessor methods, to allow property configuration using existing Java tools.

Note also that there is no `terminate()` in the MaCS case, since an empty implementation is already supplied by the `Program` trait—the corresponding `IProgram` Java interface in the JMaCS case is not allowed to do this.

Conclusions

The Lafros MaCS software, written in Scala, has been presented and compared with its predecessor, JMaCS, written in Java.

It has been shown that PD API definitions are cleaner and more concise when written to the MaCS API, in Scala, than when written to the JMaCS one, in Java.

Furthermore, it has also been shown that the remainder of each PD definition, together with PD implementations, PD deployment code, and PD programs, may be written in a way that is fully type-safe with respect to the PD's API, in the MaCS/Scala case, that was not possible in the JMaCS/Java one.

References

[1] Rob Dickens, JMaCS: a Java monitoring and control system, Proc. of SPIE Vol. 7019, 7019W (2008).

[2] Martin Odersky, Lex Spoon, Bill Venners, Programming in Scala (2008).

[3] Rob Dickens, Secure remote monitoring-and-control for the EISCAT Svalbard Radar: a case study in Java object-oriented design, 9th International EISCAT Workshop talk (Aug 1999).

[4] Rob Dickens, Monitoring and control of the 'radar.eiscat.esr' device, 10th International EISCAT Workshop poster (Jul 2001).

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