Squants

The Scala API for Quantities, Units of Measure and Dimensional Analysis

Gary Keorkunian
Lead Software Architect, Viridity Energy

What is Squants?

- Dimensional Type Library representing Quantities and their Units of Measure
- Not the first attempt at “Units” in Scala, but ...
- Currently ~50 Quantity Types and over ~160 UOM’s in 10 packages (energy, electro, market, mass, motion, photo, radio, space, thermal and time)
- Extensible - create new Quantity Types and Units of Measure in app code
- Smart unit conversions within Quantity types
- Smart type conversions between Quantity types
- Support for Money, Price, and FX
- Quantity Ranges (supports foreach, map and folds over user defined divisions)
- Natural Language DSL via optional Implicits
- Use Cases include Dimensional Analysis, Domain Models, Anticorruption Layers
- Easy to use in the Scala REPL / SBT Console (non-programmer friendly)
- Immutable and thread safe
- No Runtime Dependencies
- ~12,500 LOC / > 550 Tests / < 1MB jar file

Why Squants?

Provide developers with a type safe way to define domain specific expressions, operations, domain models and API’s.

Consider the following code:

```
val loadKw: Double = 1.2
val energyMwh: Double = 24.2
val sumKw = loadKw + energyMwh
```

What’s wrong with it?

Troubles with Doubles

Mismatches types (Power vs Energy)

Mismatches scale (Kilo vs Mega)

IT COMPILES!

Scala provides type safety …. but it’s not enough. We need **dimensional** type safety!

Dimensional Types

Squants includes a library of data types that represent about 50 different types of quantities and the units in which they're measured.

Consider the following code:

```
val load1: Power = Kilowatts(12)
val load2: Power = Megawatts(0.023)
val sum = load1 + load2
assert(sum == Kilowatts(35))
assert(sum == Megawatts(0.035))
```

What's right with it?

Clearly see what the val's represent

Quantity type works with multiples units (scales)

Math operations automatically convert scales

Squants removes the burden of type management and unit conversion from the developer.

Dimensional Type Safety

Squants prevents the “Apples and Oranges” problems of mixed type addition.

Consider the following code:

```
val load: Power = Kilowatts(1.2)
val energy: Energy = KilowattHours(23.0)
val sum = load + energy
```

This code will not compile because the operation Power + Energy is not defined.

Squants turns the semantic error made when using Double into a compile error!!

Smart Type Conversions

Squants provides smart conversions between types for valid operations.

```
val load: Power = Kilowatts(1.2)
val time: Time = Hours(2)
val energy: Energy = load * time
assert( energy == KilowattHours(2.4))
```

Often, multiplication and division operations between types will result in a value of a another type
eg. Power * Time => Energy

Squants' cross-type operations can deepen developers' understanding of these domains.

Unit Conversions

Multiple units per quantity type with simple methods for extracting values in the desired unit ...

```
val load: Power = Kilowatts(1200)
val mw: Double = load to Kilowatts // returns 1200.0 (default method)
val gw: Double = load to Gigawatts // returns 0.0012
val my: Double = load to MyPowerUnit // returns ??? Whatever you want
val kw: Double = load toKilowatts // returns 1200.0 (convenience method)
```

… and formatted strings

```
val kw: String = load toString Kilowatts // returns “1200.0 kW”
val mw: String = load toString Megawatts // returns “1.2 MW”
val gw: String = load toString Gigawatts // returns “0.0012 GW”
```

Money and Price

Money - a Quantity of Purchasing Power measured in Currencies

```
val tenBucks: Money = USD(10)
val someYen: Money = JPY(1200)
val goldStash: Money = XAU(50)
val digitalStash: Money = BTC(50)
```

Price - a Ratio between Money and another Quantity

```
val threeForADollar : Price[Dimensionless] = USD(1) / Each(3)
val energyPrice: Price[Energy] = USD(102.20) / MegawattHours(1)
val milkPrice: Price[Volume] = USD(4) / UsGallons(1)

val costForABunch: Money = threeForADollar * Dozen(10) // returns USD(40)
val energyCost: Money = energyPrice * MegawattHours(4) // returns USD(408.80)
val milkQuota: Volume = milkPrice * USD(20) // returns UsGallons(5)
```

FX Support

Currency Exchange Rates

```
val rate = CurrencyExchangeRate(USD(1), JPY(100))
val someYen: Money = JPY(350)
val dollarAmount: Money = rate.convert(someYen) // returns USD(3.5)
val someBucks: Money = USD(23.50)
val yenAmount: Money = rate * someBucks          // returns JPY(2350)
```

Money Context

```
implicit val moneyContext = MoneyContext(defCur, curList, exchangeRates)
val someMoney = Money(350) // 350 in the default Cur
val usdMoney: Money = someMoney in USD
val usdDouble: Double = someMoney to USD
val yenCost: Money = (energyPrice * MegawattHours(5)) in JPY
val northAmericanSales: Money = (CAD(275) + USD(350) + MXN(290)) in USD
```

Quantity Ranges

Represent a range of Quantity values defined by a lower and upper bound

```
val load1: Power = Kilowatts(1000)
val load2: Power = Kilowatts(5000)
val range: QuantityRange[Power] = QuantityRange(load1, load2)
```

Convert to lists or apply maps across divisions and multiples of Quantity Ranges

```
val rs1 = range * 10 // Create 10 sequential ranges starting with range2
val rs2 = range / 10 // Divide the range into 10 sequential ranges

range.foreach(10)(println) // Divide into 10 ranges and print each
val xs = range2.map(250.kw)(range => {

}) // Divide into ranges of 250.kW each and map
```

Natural Language DSL

Easy to write, easy to read expressions for creating Quantities

```scala
// Create Quantities using Unit Of Measure Factory objects (no implicits required)
val load = Kilowatts(100)
val time = Hours(3.75)
val money = USD(112.50)
val price = Price(money, MegawattHours(1))

// Create Quantities using formatted Strings
val load = Power("40 MW")

// Create Quantities using Unit of Measure names and/or symbols (uses implicits)
val load1 = 100 kW                   // Simple expressions don't need dots
val load2 = 100 megawatts
val time = 3.hours + 45.minutes  // Compound expressions need dots

// Create Quantities using operations between other Quantities
val energyUsed = 100.kilowatts * (3.hours + 45.minutes)
val price = 112.50.USD / 1.megawattHours
val speed = 55.miles / 1.hours
```

Natural Language DSL (cont)

Easy to write, easy to read expressions for creating Quantities

```
// Use single unit values to simplify expressions

// Hours(1) == 1.hours == hour
val ramp = 100.kilowatts / hour
val speed = 100.kilometers / hour

// MegawattHours(1) == 1.megawattHours == megawattHour == MWh
val price = 100.dollars / MWh

// Implicit conversion support for using Numbers to lead some expressions
val price = 10 / dollar     // 1 USD / 10 ea
val freq = 60 / second      // 60 Hz
val load = 10 * 4.MW        // 40 MW

// Create Quantity Ranges using `to` or `plusOrMinus` (+-) operators
val range1 = 1000.kW to 5000.kW // 1000.kW to 5000.kW
val range2 = 5000.kW +- 1000.kW // 4000.kW to 6000.kW
```

Use Case - Dimensional Analysis

Develop Type Safe Dimensional Analysis Code

```
val energyPrice: Price[Energy] = 45.25.money / megawattHour
val energyUsage: Energy = 345.kilowatts * 5.4.hours
val energyCost: Money = energyPrice * energyUsage

val dodgeViper: Acceleration = 60.miles / hour / 3.9.seconds
val speedAfter5Seconds: Velocity = dodgeViper * 5.seconds
val TimeTo100MPH: Time = 100.miles / hour / dodgeViper

val density: Density = 1200.kilograms / cubicMeter
val volFlowRate: VolumeFlowRate = 10.gallons / minute
val flowTime: Time = 30.minutes
val totalMassFlow: Mass = volFlowRate * flowTime * density
```

Use Case - Domain Models

Improve clarity and type safety of Domain Models and API’s

```
// This is OK …

case class Generator(id: String, maxLoadKW: Double, rampRateKWph: Double,
operatingCostPerMWh: Double, currency: String, maintenanceTimeHours: Double)
...
val gen1 = Generator("Gen1", 5000, 7500, 75.4, "USD", 1.5)
val gen2 = Generator("Gen2", 100, 250, 2944.5, "JPY", 0.5)

assetManagementActor ! ManageGenerator(gen1)
```

```
// … but this is much better

case class Generator(id: String, maxLoad: Power, rampRate: PowerRamp,
operatingCost: Price[Energy], maintenanceTime: Time)
...
val gen1 = Generator("Gen1", 5 MW, 7.5.MW/hour, 75.4.USD/MWh, 1.5 hours)
val gen2 = Generator("Gen2", 100 kW, 250 kWph, 2944.5.JPY/MWh, 30 minutes)

assetManagementActor ! ManageGenerator(gen1)
```

Use Case - Anticorruption

Build Anticorruption layers to work with services that use basic types

```
class ScadaServiceAnticorruption(val service: ScadaService) {
  // ScadaService returns load as Double representing Megawatts
  def getLoad: Power = Megawatts(service.getLoad(meterId))
  }
  // ScadaService.sendTempBias requires a Double representing Fahrenheit
  def sendTempBias(temp: Temperature) =
    service.sendTempBias(temp.to(Fahrenheit))
}
```

```
trait WeatherServiceAntiCorruption {
  val service: WeatherService
  def getTemperature: Temperature = Celsius(service.getTemperature)
  def getIrradiance: Irradiance = WattsPerSquareMeter(service.getIrradiance)
}
```

Use Case - Anticorruption

Build Anticorruption layers to provide multi-currency support

```
class MarketServiceAnticorruption(val service: MarketService)
     (implicit val moneyContext: = MoneyContext) {

  // MarketService.getPrice returns a Double representing $/MegawattHour
  def getPrice: Price[Energy] =
    (USD(service.getPrice) in moneyContext.defaultCurrency ) / megawattHour

  // MarketService.sendBid requires a Double representing $/MegawattHour
  // and another Double representing the max amount of energy in MegawattHours
  def sendBid(bid: Price[Energy], limit: Energy) =
    service.sendBid((bid * megawattHour) to USD, limit to MegawattHours)

}
```

Use Case - Anticorruption

Build Anticorruption into Akka Routers

```
// LoadReading message used within our Squant's enabled application context
case class LoadReading(meterId: String, time: Long, load: Power)

class ScadaLoadListener(router: Router) extends Actor {

  def receive = {

   // ScadaLoadReading, from an external service, types load as a string
   // eg, "10.3 MW", "345 kW"
   case msg @ ScadaLoadReading(meterId, time, loadString) ⇒
    // This handler converts the string to a Power value and emits the events
    // to the routees, presumably actors within a Squants enabled context
    router.route(LoadReading(meterId, time, Power(loadString)), sender())

  }

}
```

Creating New Units

Add new Units of Measure to existing Quantities with 3 lines of code

```
object HorsePower extends PowerUnit {
  val symbol = "hp"
  val multiplier = Watts.multiplier * 746 // 1 HP == 746 W)
}

...

val myCar = HorsePower(300)          // returns Power value eq to 223.8 kW
val s = myCar toString HorsePower    // returns “300 hp”
val hp = Watts(1492) to HorsePower   // returns 2
```

```
case class Ark(length: Length, width: Length, height: Length)
...
object Cubits extends LengthUnit {
  val symbol = "cubit"
  val multiplier = Meters.multiplier * .445 // 1 cubit ==.445m
}

val noahsArk = Ark(Cubits(300), Cubits(50), Cubits(30))
```

Creating New Quantities

Create new Quantity Types with ~ 10 lines of code

```
final class Thingee private(val value: Double) extends Quantity[Thingee] {
  def valueUnit = Thangs
  def toThangs = to(Thangs)           // optional convenience method
}

object Thingee {
  private[mypackage] def apply(value: Double) = new Thingee(value)
}

trait ThingeeUnit extends UnitOfMeasure[Thingee]

object Thangs extends ThingeeUnit with ValueUnit {
  def apply(value: Double) = Thingee(value)
  val symbol = "th"
}
...
val myThingee = Thangs(12.3)          // returns Thingee quantity eq to 12.3 th
val s = myThingee toString Thangs     // returns “12.3 th”
val sum = Thangs(3) + Thangs(2)       // returns Thingee quantity eq to Thangs(5)
val prod = Thangs(3) * 9              // returns Thingee quantity eq to Thangs(27)
val quot = Thangs(63) / Thangs(9)     // returns Double value eq to 7
val quot2 = Thangs(63) / 9            // returns Thingee quantity eq to Thangs(7)
```

Installing Squants

To use Squants in your SBT based project include the following dependency:

```
    "com.squants"  %% "squants"  % "0.2.1-SNAPSHOT"
```

To use Squants interactively in the Scala REPL, clone and run sbt console:

```
> git clone https://github.com/garyKeorkunian/squants
> cd squants
> sbt console
```

All Quantity types, Units of Measure and implicit conversions are imported to the REPL automatically and requires very little Scala knowledge for productive use.

Quantity Packages

Currently includes 10 packages with ~50 Quantity Types and ~160 UOMs

- **electro**: Capacitance, Conductivity, ElectricalConductance, ElectricalResistance, ElectricCurrent, ElectricCharge, ElectricPotential, Inductance, MagneticFlux, MagneticFluxDensity, Resistivity
- **energy**: Energy, EnergyDensity, Power, PowerRamp, SpecificEnergy
- **market**: Money (w/ support for Prices and FX)
- **mass**: ChemicalAmount, Density, Mass,
- **motion**: Acceleration, AngularVelocity, Force, Jerk, MassFlowRate, Momentum, Pressure, Velocity, VolumeFlowRate, Yank
- **photo**: Luminance, LuminousEnergy, LuminousFlux, LuminousExposure, LuminousIntensity, Illuminance
- **radio**: Irradiance, Radiance, RadiantIntensity, SpectralIntensity, SpectralPower
- **space**: Area, Angle, Length, SolidAngle, Volume
- **thermal**: Temperature, ThermalCapacity
- **time**: Frequency, Time
- …. more to come

Roadmap to 1.0

- JSON Marshalling Support (tests only)
- Validate and Improve Money and FX Support
- Implement Vector Quantity support
- More Quantity Types
- More Units of Measure
- More Dimensional Relationships
- Optimize Performance and Conversion Precision (BigDecimal?)
- Enhance Documentation (ScalaDoc, README, Cheat Sheet)
- Sample Project
- A Mascot

Interested in contributing?
Contact Gary at gkeorkunian@viridityenergy.com

Squants

Q & A and Code Review

Gary Keorkunian
Lead Software Architect, Viridity Energy