Domain Specific Languages (DSLs)

**Introduction**

A DSL is a programming language that mimics the terms, idioms, and expressions used among the experts in the target domain.

- Ideally, a domain expert, with no experience in programming, can read, understand and validate such code.

**Benefits**

- Encapsulation — a DSL hides the implementation details;
- Productivity — to be coupled to the domain, ease the coding phase;
- Communication — also non-programmer can be involved in the development;
- Quality — minor "impedance mismatch" between domain experts' requirements and implementing code.

**Drawbacks**

- Difficulties of creating good DSLs and
- Long-term maintenance

**Internal vs External DSLs**

DSLs are classified as:

- Internal or embedded
- External

**Internal DSL** is an idiomatic way of writing code in a general purpose programming language.

- No special-purpose parser is necessary
- Internal DSLs are parsed as any other code written in the language.

**External DSL** is a custom language with its own custom grammar and parser.

**Comparison**

- Internal DSLs are easier to create than external ones since they don't require a special-purpose parser.
- The constraints of the underlying language limit the options for expressing domain concepts.

**Case Study**

**Problem:** to create a payroll application that

- Computes an employee's paycheck every pay period (2 weeks long),
- The paycheck includes gross salary, net salary and deductions.

```scala
import payroll.api._
import payroll.api.DeductionsCalculator._
import payroll._
import Type2Money._

val buck = Employee(Name("Buck", "Trends"), Money(80000))
val jane = Employee(Name("Jane", "Doe"), Money(90000))

List(buck, jane).foreach { employee =>

  val deductions =
    federalIncomeTax(employee, biweeklyGross) +
    stateIncomeTax(employee, biweeklyGross) +
    insurancePremiums(employee, biweeklyGross) +
    retirementFundContributions(employee, biweeklyGross)

  val paycheck = Paycheck(biweeklyGross, biweeklyGross - deductions, deductions)

  print(format("%s %s: %s
", employee.name.first, employee.name.last, paycheck))
}
```

**Notes on the traditional solution**

- It is noisy, e.g., it mentions `employee` and `biweeklyGross` incessantly.
- The code is imperative, with the DSL it will be more declarative.
### Domain Specific Languages (DSLs)

**Case Study: Under the Hood**

#### Apart the payroll package that
- represents the 'under the hood' of our paycheck application and should be hidden to the domain experts
- the payroll script using it is difficult to be interpreted from a domain expert to check its correctness.

#### What about something like?

**Rules to calculate an employee’s paycheck:**
- Employee’s gross salary for 2 weeks minus deductions
  - FederalIncomeTax, which is 25% of gross salary.
  - StateIncomeTax, which is 5% of gross salary.
  - InsurancePensions, which are 500. in gross’s currency.
  - RetirementFundContributions are 10%. of gross salary.

- this reads like normal English, not as code.
- it contains some ‘bubble’ words as is, which...
- it is less obscure since we minimized explicit references to contextual information.

```scala
package payroll

case class Paycheck(gross: Money, net: Money, deductions: Money) {
  def plusGross(m: Money) = Paycheck(gross + m, net, deductions + m)
  def plusDeductions(m: Money) = Paycheck(gross, net + m, deductions + m)
}

package payroll

case class Employee(name: Name, annualGrossSalary: Money)

case class Name(first: String, last: String)

object Type2Money {
  implicit def int2Money(i: Int) = BigDecimal(i.toString)
  implicit def long2Money(l: Long) = BigDecimal(l.toString)
  implicit def double2Money(d: Double) = BigDecimal(d.toString)
  implicit def bigDecimal2Money(b: BigDecimal) = b
}

package payroll

object DeductionsCalculator {
  def federalIncomeTax(empl: Employee, gross: Money) = gross * .25
  def stateIncomeTax(empl: Employee, gross: Money) = gross * .05
  def insurancePremiums(empl: Employee, gross: Money) = Money(500)
  def retirementFundContributions(empl: Employee, gross: Money) = Money(gross * .10)
}
```

### Domain Specific Languages (DSLs)

**Case Study**

#### Embedded DSL

```scala
import payroll.PayrollCalculator

val buck = Employee(Name("Buck", "Trends"), Money(80000))
val jake = Employee(Name("Jake", "Doe"), Money(90000))
```

#### Some notes
- infix operator notation
- implicit conversions and user-defined types
- apply methods
package payroll.dsl

case class Duration(val amount: Int) {
  def weeks = amount * 5
  def years = amount * 260
}

package payroll.dsl
import payroll._

object rules {
  def apply(rules: Employee => Paycheck) = new PayrollBuilderRules(rules)
  implicit def int2Duration(i: Int) = Duration(i)
  implicit def employee2GrossPayBuilder(e: Employee) = new GrossPayBuilder(e)
  implicit def grossPayBuilder2DeductionsBuilder(b: GrossPayBuilder) = new DeductionsBuilder(b)
  implicit def double2DeductionsBuilderDeductionHelper(d: Double) = new DeductionsBuilderDeductionHelper(d)
}

protected[dsl] class PayrollBuilderRules(rules: Employee => Paycheck) {
  def apply(employee: Employee) = {
    try {
      rules(employee)
    } catch {
      case th: Throwable => new PayrollException("Failed to process payroll for employee: " + employee, th)
    }
  }
}

import payroll.Type2Money._

protected[dsl] class GrossPayBuilder(val employee: Employee) {
  var gross: Money = 0
  def salary_for(days: Int) = gross + dailyGrossSalary(employee.annualGrossSalary) * days
  def weeklyGrossSalary(annual: Money) = annual / 52.0
  def dailyGrossSalary(annual: Money) = annual / 260.0
}

protected[dsl] class DeductionsBuilder(gpb: GrossPayBuilder) {
  val employee = gpb.employee
  var paycheck: Paycheck = new Paycheck(gpb.gross, gpb.gross, 0)
  def currency = this
  def minus_deductions_for(deductionRules: DeductionsBuilder => Unit) = {
    deductionRules(this)
    paycheck
  }
  def addDeductions(amount: Money) = paycheck = paycheck plusDeductions amount
  def addDeductionsPercentageOfGross(percentage: Double) = {
    val amount = paycheck.gross * (percentage/100.)
    addDeductions(amount)
  }
}

class DeductionCalculator {
  def is(builder: DeductionsBuilder) = apply(builder)
  def are(builder: DeductionsBuilder) = apply(builder)
  def apply(builder: DeductionsBuilder) = {}
}

object federalIncomeTax extends DeductionCalculator
object stateIncomeTax extends DeductionCalculator
object insurancePremiums extends DeductionCalculator
object retirementFundContributions extends DeductionCalculator

protected[dsl] class DeductionsBuilderDeductionHelper(val factor: Double) {
  def in (builder: DeductionsBuilder) = {
    builder addDeductions Money(factor)
    builder
  }
  def percent_of (builder: DeductionsBuilder) = {
    builder addDeductionsPercentageOfGross percentage: Double = {
      val amount = paycheck.gross * (percentage/100.)
      addDeductions(amount)
    }
  }
}

References


