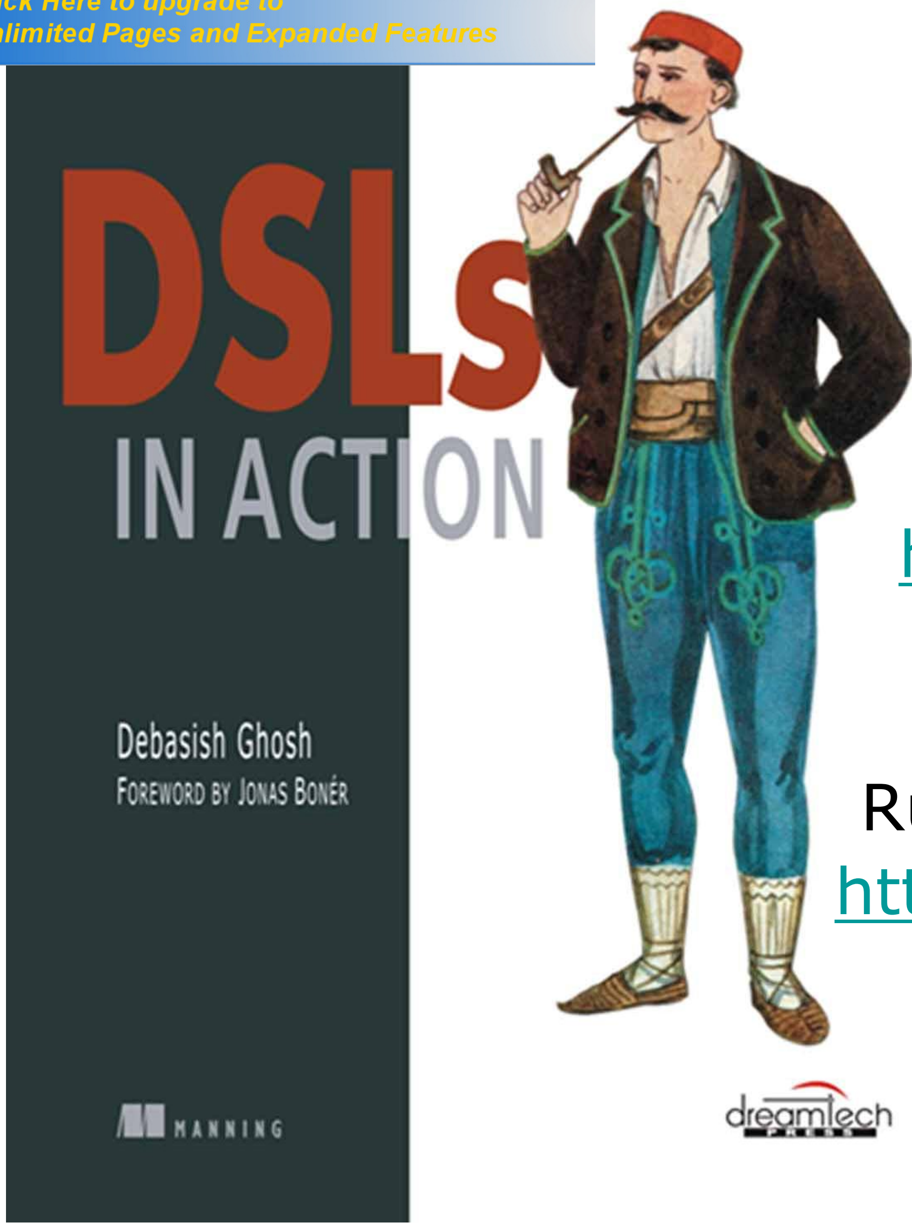
Domain Specific Languages

-

a *user* view and an *implementation* view

does expressiveness imply a compromise on the underlying domain model ?



Debasish Ghosh

@debasishg on twitter

Code @
http://github.com/debasishg

Blog @
Ruminations of a Programmer
http://debasishg.blogspot.com

Open Source Footprints

- Committer in Akka (http://akka.io)

- Owner/Founder of :
  - □ sjson (JSON serialization library for Scala objects □ http://github.com/debasishg/sjson)
  - □ scala-redis (Redis client for Scala □ http://github.com/debasishg/scala -redis)
  - □ scouchdb (Scala driver for Couchdb □ http://github.com/debasishg/scouchdb)

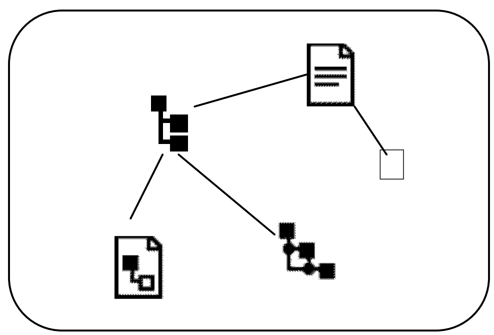
- model-view architecture of a DSL
- how abstractions shape a DSL structure
- choosing proper abstractions for compositionality
- using proper language idioms
- composing DSLs

- **model-view** architecture of a DSL
- how abstractions shape a DSL structure
- choosing proper abstractions for compositionality
- using proper language idioms
- composing DSLs

View Or Syntax

```
new_trade 'T-12435'
   for account 'acc-123'
   to buy 100 shares of 'IBM',
   at UnitPrice = 100,
      Principal = 12000, Tax = 500
```

Model



Semantic model of the domain

view / syntax

is derived from

model

view / syntax

is a veneer of abstraction over

model

view / syntax

is decoupled thru an interpreter from

model

view / syntax

can be multiple over the same

model

y API

```
# create instrument
instrument =
  Instrument.new('Google')
instrument.quantity = 100

#create trade
Trade.new('T-12435',
  'acc-123', :buy, instrument,
  'unitprice' => 200,
  'principal' => 120000,
  'tax' => 5000
)
```

syntax

```
new_trade 'T-12435'
   for account 'acc-123'
   to buy 100 shares of
   'IBM',
   at UnitPrice = 100,
      Principal = 12000,
      Tax = 500
```

syntax

```
new_trade 'T-12435'
   for account 'acc-123'
   to buy 100 shares of
   'IBM',
   at UnitPrice = 100,
      Principal = 12000,
      Tax = 500
```

domain syntax

↓

interpreter

↓

idiomatic Ruby model

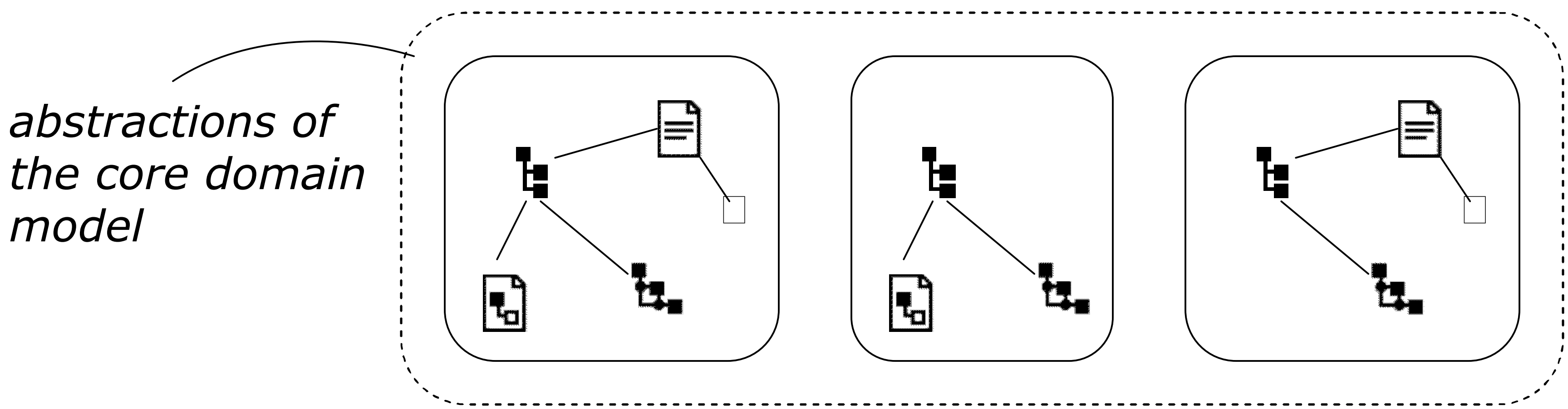
isolation layer between the model and the view of the DSL

```
def parse(dsl_string)
  dsl = dsl_string.clone
  dsl.gsub!(/=/, '=>')
  dsl.sub!(/and /, '')
  dsl.sub!(/at /, '')
  dsl.sub!(/for account /, ',')
  dsl.sub!(/to buy /, ', :buy, ')
  dsl.sub!(/(\d+) shares of ('.*?')/,
    '\1.shares.of(\2)')
  puts dsl
  dsl
end
```

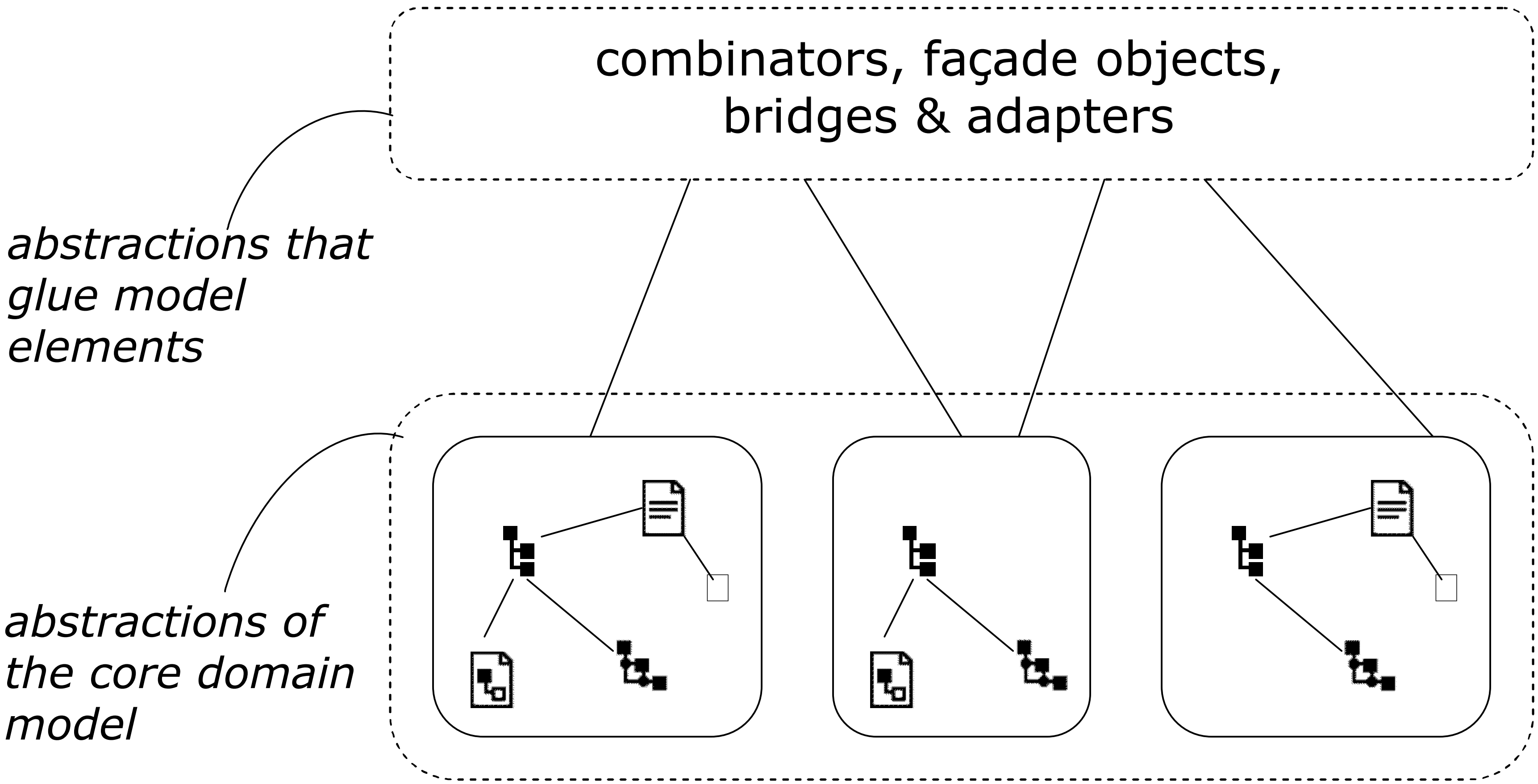
Not all transformations can be done using simple regular expression based string processing

- model-view architecture of a DSL
- how **abstractions** shape a DSL structure
- choosing proper abstractions for compositionality
- using proper language idioms
- composing DSLs

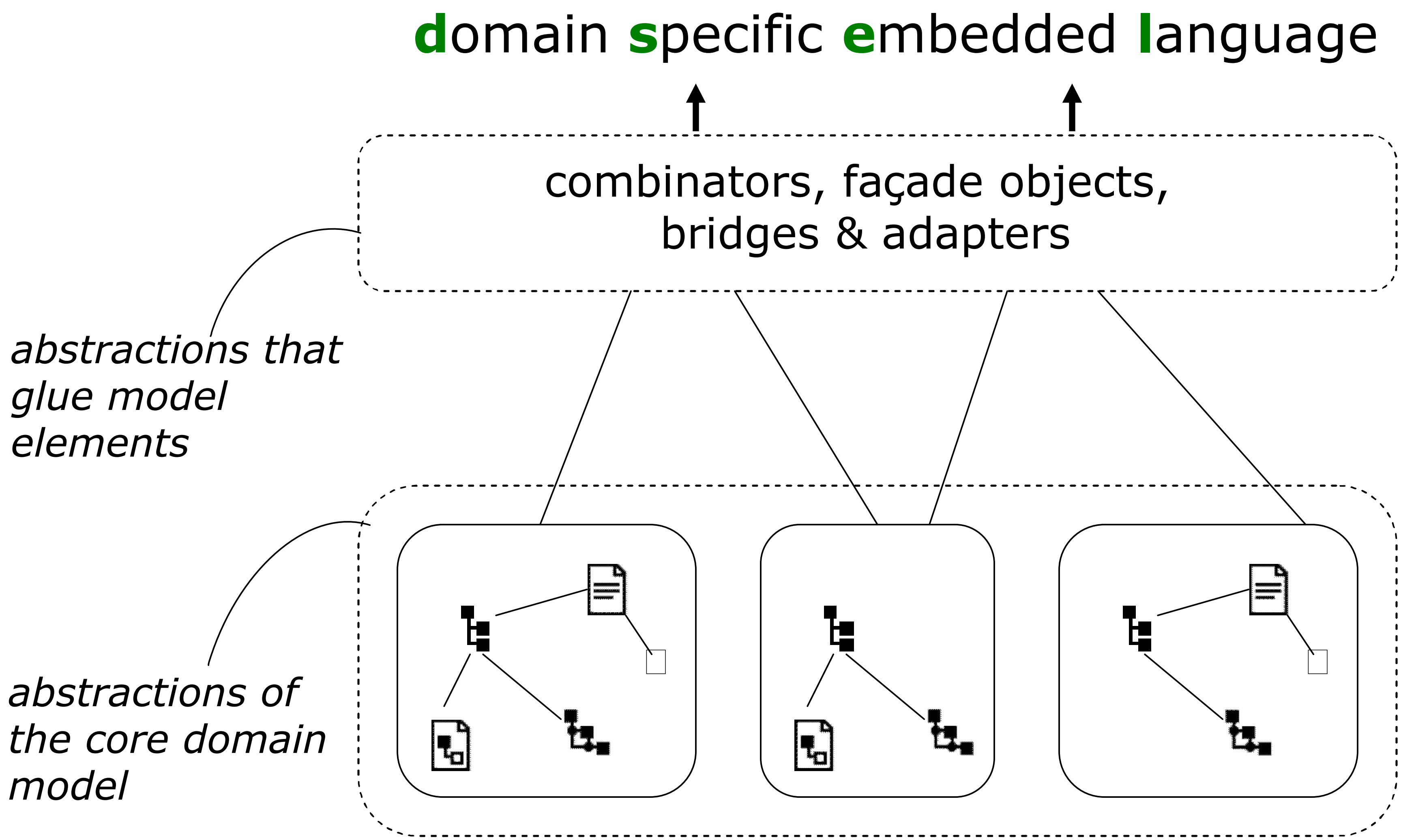
tractions



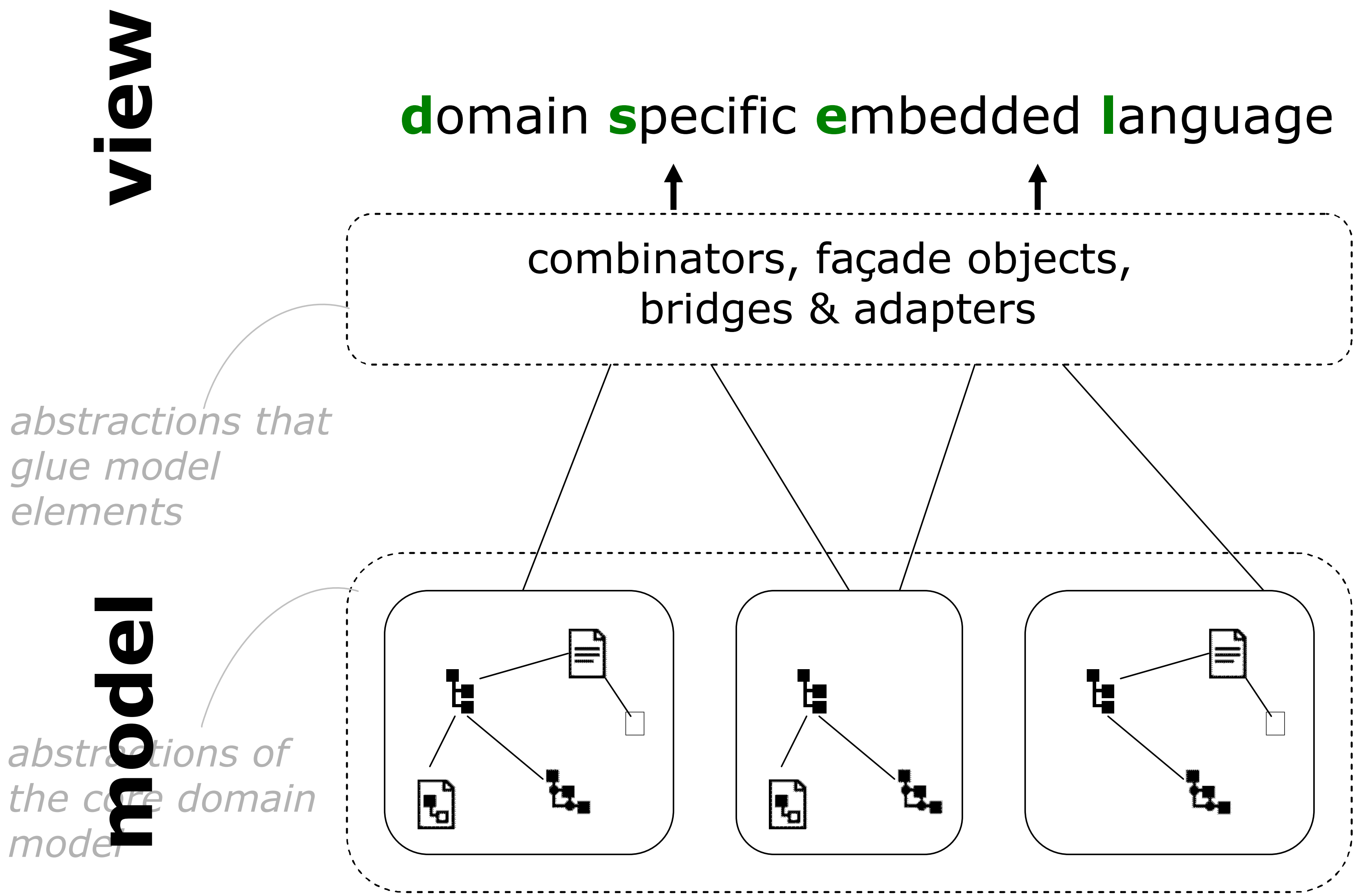
tractions


combinators, façade objects, bridges & adapters
abstractions that glue model elements
abstractions of the core domain model

tractions


domain specific embedded language
combinators, façade objects, bridges & adapters
abstractions that glue model elements
abstractions of the core domain model

ractions


view
domain specific embedded language
combinators, façade objects,
bridges & adapters
abstractions that glue model elements
model
abstractions of the core domain model

layers of **abstractions ..**

- keep the underlying domain model clean
- ensure separation of concerns between the model and the view of the DSL
- you can also implement the glue layer using a different programming language (polyglot programming)

[illegible] over a **Java** domain model

```
withAccount(trade) {
  account => {
    settle(trade using
      account.getClient
             .getStandingRules
             .filter(_.account == account)
             .first)
    andThen journalize
  }
}
```

DSL in Scala over a **Java** domain model

Java abstractions

```
withAccount(trade) {
  account => {
    settle(trade using
      account.getClient
             .getStandingRules
             .filter(_.account == account)
             .first)
    andThen journalize
  }
}
```

DSL in Scala over a **Java** domain model

Java abstractions

```
withAccount(trade) {
  account => {
    settle(trade using
      account.getClient
             .getStandingRules
             .filter(_.account == account)
             .first)
    andThen journalize
  }
}
```

Wired through Scala control structures

We need **abstractions** that generalize over computations ..

main – stocks, trades, bulls, bears ..

- Security trade
  - Exchange of securities and currencies (also known as Instruments)
  - In the stock exchange (market)
  - On a specified "trade date"
  - Between counter parties
  - In defined quantities
  - At market rates
  - Resulting in a net cash value for the seller
  - On the "value date"

main – stocks, trades, bulls, bears ..

- **Security trade**
  - Exchange of securities and currencies (also known as **Instruments**)
  - In the stock exchange (**market**)
  - On a specified "**trade date**"
  - Between counter parties' **accounts**
  - In defined **quantities**
  - At market rates
  - Resulting in a **net cash value** for the seller
  - On the "**value date**"

```scala
// The Trade model
case class Trade(
  account: Account,
  instrument: Instrument,
  referenceNo: String,
  market: Market,
  unitPrice: BigDecimal,
  quantity: BigDecimal,
  tradeDate: Date = Calendar.getInstance.getTime,
  valueDate: Option[Date] = None,
  taxFees: Option[List[(TaxFeeId, BigDecimal)]] =
None,
  netAmount: Option[BigDecimal] = None)
```

```
// various tax/fees to be paid when u do a trad
sealed trait TaxFeeId
case object TradeTax extends TaxFeeId
case object Commission extends TaxFeeId
case object VAT extends TaxFeeId
case object Surcharge extends TaxFeeId
```

```
// computes value date of a trade
val valueDateProcessor: Trade => Trade = //..
impl

// computes tax/fee of a trade
val taxFeeProcessor: Trade => Trade = //..
Impl

// computes the net cash value of a trade
val netAmountProcessor: Trade => Trade = //..
impl
```

```
val trades = //.. list of security trades

// get a list of value dates for all trades
val valueDates = trades map (_.valueDate)

// enrich a trade passing it thru multiple
processors
val richTrade = (valueDateProcessor map
                   taxFeeProcessor map
                     netAmountProcessor) apply
trade

// pass a trade thru one specific processor
val t = some(trade) map valueDateProcessor
```

```
.. list of security trades

// get a list of value dates for all trades
val valueDates = trades map (_.valueDate)

// enrich a trade passing it thru multiple
processors
val richTrade = (valueDateProcessor map
                    taxFeeProcessor map
                      netAmountProcessor)
apply trade

// pass a trade thru one specific processor
val t = some(trade) map
valueDateProcessor
```

```
val trades = //.. list of security trades

// get a list of value dates for all trades
val valueDates = trades map (_.valueDate)

// enrich a trade passing it thru multiple processors
val richTrade = (valueDateProcessor map
                    taxFeeProcessor map
                      netAmountProcessor) apply trade

// pass a trade thru one specific processor
val t = some(trade) map valueDateProcessor
```

```
val trades = //.. list of security trades

// map across the list functor
val valueDates = trades map (_.valueDate)

// map across the Function1 functor
val richTrade = (valueDateProcessor map
                   taxFeeProcessor map
                     netAmountProcessor) apply
trade

// map across an Option functor
val t = some(trade) map valueDateProcessor
```

a **map** is ..

a combinator

a **map** ..

pimps abstractions to give us an ad hoc polymorphism implementation on top of OO a **map** is ..

fundamentally powered by type classes

a pimp

```
sealed trait MA[M[_], A] extends PimpedType[M[A]] {
  def map[B](f: A => B)(implicit t: Functor[M]): M[B]
    t.fmap(value, f)
```

a type class

source: scalaz : https://github.com/scalaz/scalaz

```
trait Functor[F[_]] extends InvariantFunctor[F]
  def fmap[A, B](r: F[A], f: A => B): F[B]
}
```

source: scalaz : https://github.com/scalaz/scalaz

a **functor** is ..
a type class for all things that can be mapped over

```
// map across the Function1 functor
val richTrade = (valueDateProcessor map
                   taxFeeProcessor map
                     netAmountProcessor) apply
trade
```

```
Function1[]
```

```
implicit def Function1Functor[R] =
  new Functor //..
```

the idea is to **generalize** abstractions ..

.. and by generalizing **map** over functors we get a combinator that can glue a large set of our domain abstractions

.. and this gives a feeling of uniformity in the DSL syntax

semantic model

+

combinators

=

DSL

- model-view architecture of a DSL
- how abstractions shape a DSL structure
- choosing proper abstractions for **compositionality**
- using proper language idioms
- composing DSLs

a **DSL** ..

- evolves from the model non-invasively
- using combinators
- that compose model elements
- interprets them into computations of the domain

so, a domain model also has to be ..

designed for compositionality

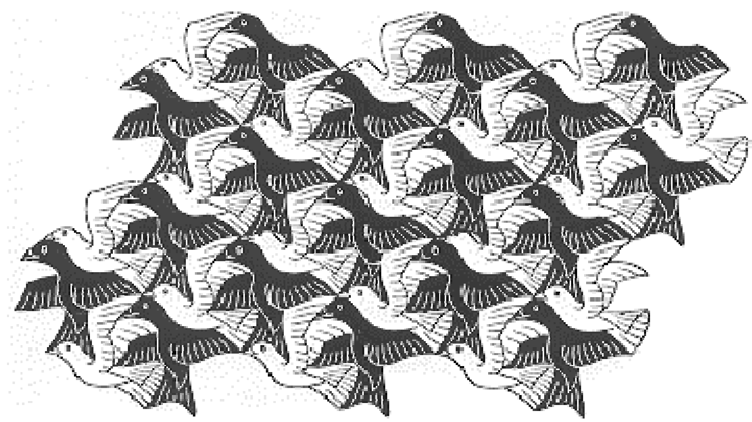
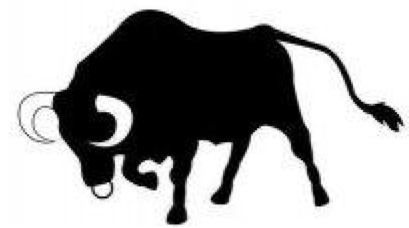


Image source: Mathematical art of M. C. Escher
(http://im-possible.info/english/articles/escher_math/escher_math.html)



domain rule ..

enrichment of trade is done in steps:

1. get the tax/fee ids for a trade
2. calculate tax/fee values
3. enrich trade with net cash amount

the **DSL** ..

```
val enrich = for {
  // get the tax/fee ids for a trade
  taxFeeIds <- forTrade

  // calculate tax fee values
  taxFeeValues <- taxFeeCalculate

  // enrich trade with net cash amount
  netAmount <- enrichTradeWith
}
yield((taxFeeIds map taxFeeValues) map netAmount)

// enriching a trade
trade map enrich should equal(…)
```

the **DSL** ..

Model elements that can be composed using the for-comprehension

```
val enrich = for {
  // get the tax/fee ids for a trade
  taxFeeIds <- forTrade

  // calculate tax fee values
  taxFeeValues <- taxFeeCalculate

  // enrich trade with net cash amount
  netAmount <- enrichTradeWith
}
yield((taxFeeIds map taxFeeValues) map netAmount)

// enriching a trade
trade map enrich should equal(…)
```

reiterating ..

a **combinator** acts ..
as a glue combining model elements

But ..

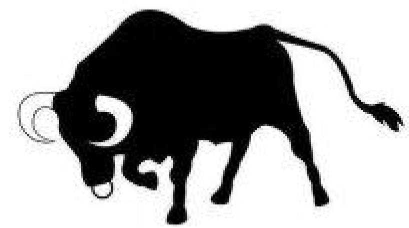
there is a **prerequisite** ..

model elements should be composable

we call this the **compositionality** of the model

Q: How to design your model
so that it's **compositional** ?

A: choose the right **abstractions** ..
depending on the programming **paradigm**
(FP, OO ..) and the implementation
language used



domain rule ..

net cash value calculation of trade:

1. use strategy for specific markets, if available
2. otherwise use default strategy
3. we can supply specific strategy inline as well

the **DSL** ..

```
// type alias for domain vocabulary
type NetAmount = BigDecimal
type CashValueCalculationStrategy =
  PartialFunction[Market, Trade => NetAmount]


// declarative business logic
val cashValueComputation:
  Trade => NetAmount = { trade =>
   (forHongKong orElse
     forSingapore orElse
       forDefault)(trade.market)(trade)
}
```

the **DSL** ..

```
val cashValue:
  Trade => CashValueCalculationStrategy => NetAmount
  { trade => pf =>
    if (pf.isDefinedAt(trade.market))
      pf(trade.market)(trade)
    else cashValueComputation(trade)
}
```

`PartialFunction` is ..

the secret sauce

- all functions for specific and default computation return `PartialFunction`

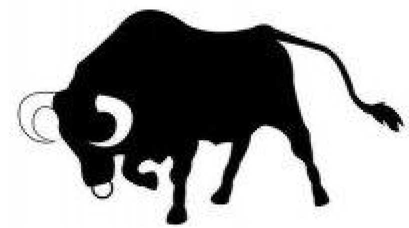
- that offer combinators for chaining of abstractions ..

```
val forHongKong: CashValueCalculationStrategy =
  case HongKong => { trade => //.. }
}

val forSingapore: CashValueCalculationStrategy =
  case Singapore => { trade => //.. }
}

val forDefault: CashValueCalculationStrategy = {
  case _ => { trade => //.. }
}
```

- model-view architecture of a DSL
- how abstractions shape a DSL structure
- choosing proper abstractions for compositionality
- using proper **language idioms**
- composing DSLs

choice of abstractions depends on the **programming language** chosen for implementation of your DSL

always keep an eye on the **paradigms** that your implementation language supports and play to the **idioms** of the language



domain rule ..

A trade needs to be enriched with a set of tax and fees before we calculate its net cash value

the **Java** way ..

```
// use the decorator pattern
new CommissionDecorator(
  new TaxFeeDecorator(new Trade(..)));

public class Trade { //..

public class TaxFeeDecorator extends Trade {
  private Trade trade;
  //..
}
public class CommissionDecorator extends Trade {
  private Trade trade;
  //..
}
```

the **Java** way ..

- decorators and the decoratee share a common interface
- statically coupled through inheritance
- the decorator usage syntax is outside-in

the **Ruby** way ..

```
## decorator pattern
Trade.new(..).with TaxFee, Commission

class Trade
  attr_accessor ..
  def initialize ..
  def with(*args)
    args.inject(self) {|memo, val| memo.extend v
  end
  def value
    @principal
  end
end
```

ay ..

```
## mixins
module TaxFee
  ## calculate taxfee
  def value
    super + ..
  end
end
```

```
## mixins
module Commission
  ## calculate commission
  def value
    super + ..
  end
end
```

the **Ruby** way ..

- more dynamic compared to Java
- uses Ruby idioms of runtime meta-programming
- decorators don't statically extend the core abstraction

if we were to do the same in **Clojure** ..

```
## decorator pattern
(with-tax-fee trade
  (with-values :tax 12)
  (with-values :commission 23))

(defmacro with-tax-fee
  "wrap a function in one or more decorators"
  [func & decorators]
  `(redef ~func (-> ~func ~@decorators)))
```

the **Clojure** way ..

- more functional – uses HOF
- function threading makes the decorator implementation almost trivial
- control structure uses macros, which are compile time meta-programming artifacts (unlike Ruby) – no run time overhead

- model-view architecture of a DSL
- how abstractions shape a DSL structure
- choosing proper abstractions for compositionality
- using proper language idioms
- **composing** DSLs

if you thought DSLs grow through c**omposition of abstractions** ..

.. then the final DSL is the **whole** formed from the composition of its **parts** ..

... and if the final DSL is also an abstraction ..

.. there's no reason we can compose it with **another abstraction** (aka another DSL) to form a **bigger whole** ..

constrained abstract type

```
trait TradeDsl {
  type T <: Trade
  def enrich: PartialFunction[T,
T]

  //.. other methods
}
```

abstract method

d income trade

```
trait FixedIncomeTradeDsl extends TradeDsl {
  type T = FixedIncomeTrade

  import FixedIncomeTradingService._
  override def enrich: PartialFunction[T, T] = {
    case t =>
      t.cashValue = cashValue(t)
      t.taxes = taxes(t)
      t.accruedInterest = accruedInterest(t)
      t
  }
}

object FixedIncomeTradeDsl
    extends FixedIncomeTradeDsl
```

concrete definition for enrichment of fixed income trade *ity trade*

```
trait EquityTradeDsl extends TradeDsl {
  type T = EquityTrade

  import EquityTradingService._
  override def enrich: PartialFunction[T, T] = {
    case t =>
      t.cashValue = cashValue(t)
      t.taxes = taxes(t)
      t
  }
}

object EquityTradeDsl
    extends EquityTradeDsl
```

concrete definition for enrichment of equity trade

.. and now a new market rule comes up that needs to be applied to **all** types of trades

.. you **don't want to affect the core logic**, since the rule is a temporary and promotional one and is likely to be discontinued after a few days ..

.. design the DSL for the new rule in such a way that you can **plug in** the semantics of **all types of trade** within it ..

and now the rule itself ..

"Any trade on the New York Stock Exchange of principal value > 10,000 USD must have a discount of 10% of the principal on the net cash value."

```
trait MarketRuleDsl extends TradeDsl {
  val semantics: TradeDsl
  type T = semantics.T
  override def enrich: PartialFunction[T, T] = {
    case t =>
      val tr = semantics.enrich(t)
      tr match {
        case x if x.market == NYSE &&
                  x.principal > 1000 =>
          tr.cashValue = tr.cashValue - tr.principal *
0.1
          tr
        case x => x
      }
  }
}
```

```
trait MarketRuleDsl extends TradeDsl {
  val semantics: TradeDsl
  type T = semantics.T
  override def enrich: PartialFunction[T, T] = {
    case t =>
      val tr = semantics.enrich(t)
      tr match {
        case x if x.market == NYSE &&
                  x.principal > 1000 =>
          tr.cashValue = tr.cashValue - tr.principal *
0.1
          tr
        case x => x
      }
  }
}
```

polymorphic embedding

new business rule that acts on top of the enriched trade

pluggable composition of DSLs

```
object EquityTradeMarketRuleDsl extends MarketRuleDsl {
  val semantics = EquityTradeDsl
}

object FixedIncomeTradeMarketRuleDsl extends
MarketRuleDsl {
  val semantics = FixedIncomeTradeDsl
}
```

plug-in the concrete semantics

a sneak peek at the **fully composed** DSL ..

```
import FixedIncomeTradeMarketRuleDsl._

withTrade(
  200 discount_bonds IBM
    for_client NOMURA
    on NYSE
    at 72.ccy(USD)) {trade =>
      Mailer(user) mail trade
      Logger log trade
    } cashValue
```

Summary

- A DSL has a syntax (for the user) and a model (for the implementer)
- The syntax is the view designed on top of the model
- The model needs to be clean and extensible to support seamless development of domain syntax
- The model needs to be compositional so that various model elements can be glued together using combinators to make an expressive syntax

QA