Domain Modeling in a Functional World

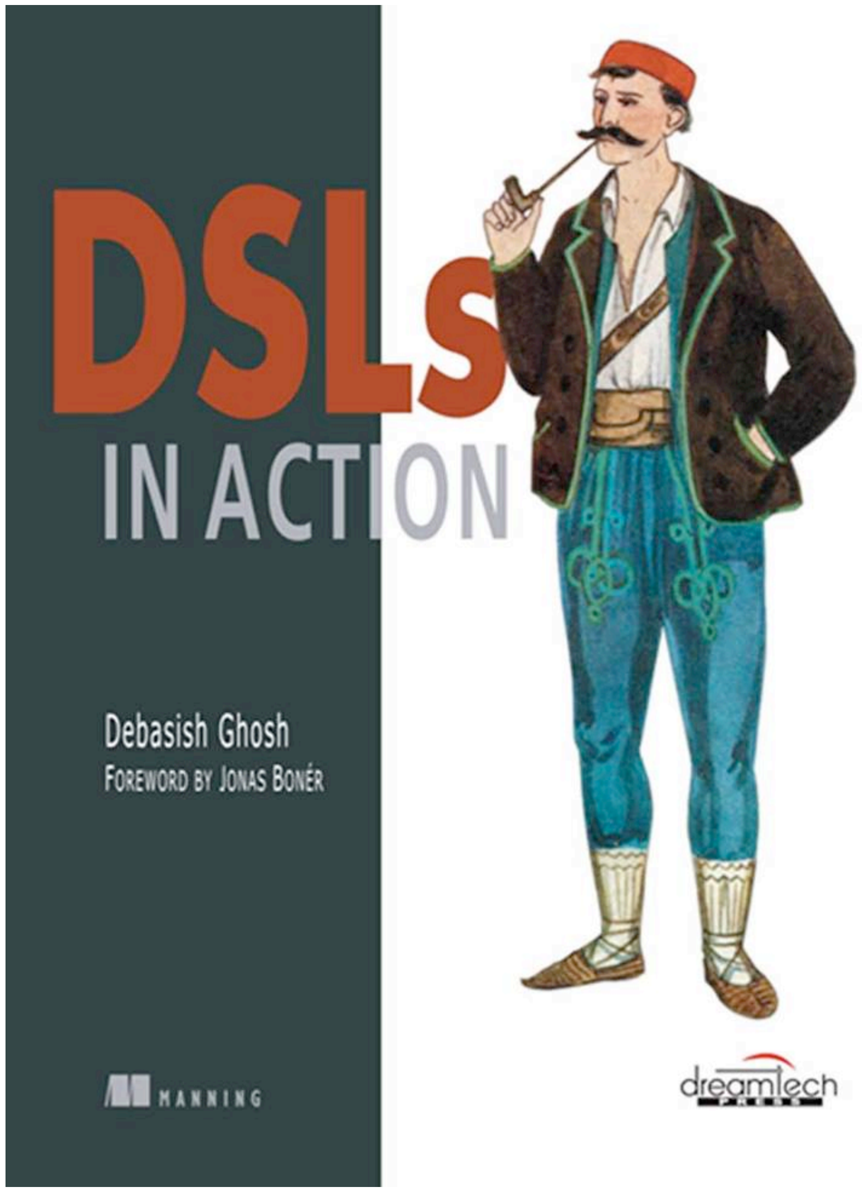
some real life experiences

Debasish Ghosh

@debasishg on Twitter

code @
http://github.com/debasishg

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



What is Domain Modeling

- We limit ourselves strictly to how the domain *behaves* internally and how it responds to events that it receives from the *external context*
- We think of a domain model as being comprised of the core granular abstractions that handle the business logic and a set of coarser level services that interacts with the external world

Agenda

- ☑ Immutability and algebraic data types
- ☑ Updating domain models, functionally
- ☑ Type classes & domain modeling
- ☑ Models of computation
- ☑ Managing states - the functional way
- ☑ A declarative design for domain service layer

Functional domain models

- ☑ Immutability and algebraic data types
- ☑ Updating domain models, functionally
- ☑ Type classes & domain modeling
- ☑ Models of computation
- ☑ Event Sourcing
- ☑ A declarative design for domain service layer

Immutability

- Immutable data
  - ✦ can be shared freely
  - ✦ no shared mutable state and hence no locks, semaphores etc.
  - ✦ and you get some parallelism free

Immutability

- Algebraic Data Types for representation of domain entities
- Immutable structures like type-lenses for functional updates
- No in-place mutation
- Heavy use of persistent data structures

Algebraic Data Types

Ok .. I get the Data Type part, but give me the ALGEBRA ...

Algebraic Data Types

- For simplicity let's assume that an algebraic data type induces an algebra which gives us some structures and some functions (or morphisms or arrows) to manipulate. *So we have some types and some functions that morph one type into another*
- Well .. almost .. it's actually an *initial algebra*, but let's stop at that without sounding more scary

Rather we look at some examples ..

- Unit represented by 1
- Optional data type represented as `Option` in Scala (`Maybe` in Haskell) - a *Sum* type $1 + X$
- Tuples represented as pairs (a, b) - the simplest *Product* type
- Recursive data types - Lists of X represented by $L = 1 + X * L$
- Binary Trees, represented as $B = 1 + X * B^2$

A Taste of Algebra

unit type

- A `List[Int]` can be either an *empty list* or consisting of one integer, *or two integers*, or three etc.

sum type

product type

- So a list of integers can be represented algebraically as

  1 + int + int * int + int * int * int + ... OR

  1 + int + int ** 2 + int ** 3 + ..

and that's not all ..

- we can have Taylor series expansion, Composition, and even Differentiation on types ..

and that's not all ..

- we can have Taylor series expansion, Composition, and even Differentiation on types ..

we can reserve them for a Halloween discussion

Product Type

Formal Definition from Bob Harper PFPL Chapter 14 :

*"The binary **product** of two types consists of ordered pairs of values, one from each type in the order specified. The associated eliminatory forms are projections, which select the first and second component of a pair. The nullary product, or unit type consists solely of the unique null tuple of no values, and has no associated eliminatory form"*

Product Type

- Implemented through *case classes* in Scala
- In a domain model we represent entities using **product types**
- A tuple is the simplest product type :

```
val point = (x_cord, y_cord)
```

Representation as ADTs (product type)

```
case class Trade(account: Account, instrument: Instrument,
  refNo: 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) {

  override def equals(that: Any) =
    refNo == that.asInstanceOf[Trade].refNo
  override def hashCode = refNo.hashCode

}
```

Sum Type

Formally from Bob Harper in PFPL, Chapter 15 :

*"Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by **sum types**, specifically the binary sum, which offers a choice of two things, and the nullary sum, which offers a choice of no things"*

Sum Type

- Implemented through subtyping in Scala
- `Option` is one of the most commonly used sum type

```scala
sealed abstract class Option[+A] extends Product with
Serializable //..

final case class Some[+A](x: A) extends
Option[A] //..

case object None extends Option[Nothing] //..
```

Representation as ADTs (sum type)

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

ADTs & Domain Modeling

- Encouraging immutability
  - In Scala you can use vars to have mutable case classes, but that's not encouraged
  - With Haskell algebraic data types are immutable values and you can use things like the State monad for implicit state update
  - There are some specialized data structures that allow functional updates e.g. Lens, Zipper etc.

Agenda

- Immutability and algebraic data types
- Updating domain models, functionally
- Type classes & domain modeling
- Models of computation
- Managing states - the functional way
- A declarative design for domain service layer

Updating a Domain Structure functionally

- A Type-Lens is a data structure that sets up a bidirectional transformation between a set of source structures S and target structures T
- A Type-Lens is set up as a pair of functions:
  - `get S -> T`
  - `putBack (S X T) -> S`

Type Lens in Scala

```scala
case class Lens[A, B] (
  get: A => B,
  set: (A, B) => A
) //..
```

A Type Lens in Scala

```
// change ref no
val refNoLens: Lens[Trade, String] =
  Lens((t: Trade) => t.refNo,
        (t: Trade, r: String) => t.copy(refNo = r))
```

a function that takes a
trade and returns it's reference no

a function that updates a
trade with a supplied reference no

Lens under Composition

- What's the big deal with a Type Lens ?
    - Lens compose and hence gives you a cool syntax to update nested structures within an ADT

```
def addressL: Lens[Person, Address] = ...
def streetL: Lens[Address, String] = ...
val personStreetL: Lens[Person, String] =
  streetL compose addressL
```

Lens under composition

Using the `personStreetL` lens we may access or set the (indirect) street property of a `Person` instance

```
val str: String =
  personStreetL get person

val newP: Person =
  personStreetL set (person, "Bob_St")
```

Functional updates in our domain model using Type Lens

```
// change ref no
val refNoLens: Lens[Trade, String] =
  Lens((t: Trade) => t.refNo,
       (t: Trade, r: String) => t.copy(refNo = r))

// add tax/fees
val taxFeeLens: Lens[Trade, Option[List[(TaxFeeId, BigDecimal)]]] =
  Lens((t: Trade) => t.taxFees,
       (t: Trade, tfs: Option[List[(TaxFeeId, BigDecimal)]]) =>
t.copy(taxFees = tfs))

// add net amount
val netAmountLens: Lens[Trade, Option[BigDecimal]] =
  Lens((t: Trade) => t.netAmount,
       (t: Trade, n: Option[BigDecimal]) => t.copy(netAmount = n))

// add value date
val valueDateLens: Lens[Trade, Option[Date]] =
  Lens((t: Trade) => t.valueDate,
       (t: Trade, d: Option[Date]) => t.copy(valueDate = d))
```

Agenda

- ☑ Immutability and algebraic data types
- ☑ Updating domain models, functionally
- ☑ Type classes & domain modeling
- ☑ Models of computation
- ☑ Managing states - the functional way
- ☑ A declarative design for domain service layer

Type class

- Ad hoc polymorphism
- Open, unlike subtyping - makes more sense in domain modeling because domain rules also change. Useful for designing *open* APIs which can be adapted later to newer types
- Leads to generic, modular and reusable code

Useful type classes for your domain model

- Functor - offers you the capability to map over a structure. And not surprisingly, it has a single method: *map*

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
  //..
}
```

More type classes

- Applicative Functors - Beefed up functors. Here the function is wrapped within a functor. So we lift a function wrapped in a functor to apply on another functor

```
trait Applicative[F[_]] extends Functor {
  def apply[A, B](f: F[A => B]): F[A] => F[B]
  def pure[A](a: A): F[A]
  //..
}
```

More type classes

- Monads - beefed up Applicative Functor, where in addition to `apply` and `map`, you get a `bind` (>>=) function which helps you bind a monadic value to a function's input that produces a monadic output

```
trait Monad[F[_]] extends Applicative {
  def >>=[A, B] (fa: F[A])(f: A => F[B]): F[B]
}
```

And some more ..

- Semigroup - a type class that offers an associative binary operation

```
trait Semigroup[F] {
  def append(f1: F, f2: => F): F
  //..
}
```

- Monoid - a Semigroup with an *identity* element

```
trait Monoid[F] extends Semigroup[F] {
  def zero: F
  //..
}
```

Type Classes & Domain Modeling

- Define protocols which all implementations need to honor. Useful for domain rules (e.g. Validations) that need to be implemented across a range of domain types (not necessarily related)
- Offer composability (`apply` function of Applicative functors) & sequencing (`bind` function of monads) of operations

Accumulating validation errors using Type Classes

- We can use Semigroup to accumulate errors in our domain validation logic
- Semigroup is an abstraction that offers an associative binary append operation - looks intuitive for our use case

Accumulating validation errors using Type Classes

```scala
sealed trait Validation[+e, +A] {
  def append[EE >: E, AA >: A](that: Validation[EE, AA])
    (implicit es: Semigroup[EE], as: Semigroup[AA])
    : Validation[EE, AA] = (this, that) match {

    // both Success: combine results using Semigroup
    case (Success(a1), Success(a2))     =>
      Success(as.append(a1, a2))

    // one side succeeds : return success value
    case (v1@Success(a1), Failure(_))  => v1
    case (Failure(_), v2@Success(a2))  => v2

    // both Failures: combine errors using Semigroup
    case (Failure(e1), Failure(e2))     =>
      Failure(es.append(e1, e2))
}
```

Type Classes - effective for the domain model

- A small data structure like Semigroup has a big effect in our functional domain model
- How does something with just an associative binary operation be so effective ?
- The secret sauce is in the power of function composition and the ability to implement type classes on any abstraction

```scala
   // using Validation as an applicative
// can be combined to accumulate exceptions
def makeTrade(account: Account, instrument: Instrument,
  refNo: String, market: Market, unitPrice: BigDecimal,
  quantity: BigDecimal) =
  (validUnitPrice(unitPrice).liftFailNel |@|
     validQuantity(quantity).liftFailNel) { (u, q) =>
       Trade(account, instrument, refNo, market, u, q)
     }
```

an applicative builder, admit, it looks scary!

```scala
   // validate quantity
def validQuantity(qty: BigDecimal): Validation[String,
BigDecimal] =
  if (qty <= 0) "qty must be > 0".fail
  else if (qty > 500) "qty must be <= 500".fail
  else qty.success


// validate unit price
def validUnitPrice(price: BigDecimal):
Validation[String, BigDecimal] =
  if (price <= 0) "price must be > 0".fail
  else if (price > 100) "price must be <= 100".fail
  else price.success
```

So far in the type class world ..

- Type classes lead to design of abstractions that can be extended post hoc
- Type classes and higher order functions is a potent combo of ad hoc polymorphism and function composition
- Don't ignore small abstractions like Semigroup and Monoid - when placed in the proper context, they can contribute to writing succinct code for your domain model

Composability FTW

- Functions compose mathematically. Same for functional programming (a big assumption - no side-effects)
- All functional programming languages encourage isolating side-effects from *pure* domain logic .. and some do enforce through the type system

For your domain model ..

Decouple pure logic from side-effects ..

domain rules isolated from impure stuff

easier to unit test

easier to reason about your logic

flexible reordering of operations - easier parallelism

What is composition ?

- compose - v. *to make or create by putting together parts or elements*
- a bottom up way of creating abstractions
- combinators FTW

Combinators

"a function which builds program fragments from program fragments; in a sense the programmer using combinators constructs much of the desired program automatically, rather than writing every detail by hand"

- John Hughes

Generalising Monads to Arrows

(http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf)

Combinators act as the glue ..

- `map`
- `filter`
- `bind`
- `apply`
- `...`

Pure functions as domain behaviors

```
// value a tax/fee for a specific trade
val valueAs:
  Trade => TaxFeeId => BigDecimal = {trade => tid =>
  ((rates get tid) map (_ * principal(trade)))
     getOrElse (BigDecimal(0))
}

// all tax/fees for a specific trade
val taxFeeCalculate:
  Trade => List[TaxFeeId] => List[(TaxFeeId, BigDecimal)] = {t =>
tids =>
  tids zip (tids map valueAs(t))
}
```

Pure functions as domain behaviors

```
// value a tax/fee for a specific trade
val valueAs:
  Trade => TaxFeeId => BigDecimal = {trade => tid =>
  ((rates get tid) map (_ * principal(trade)))
     getOrElse (BigDecimal(0))
}

// all tax/fees for a specific trade
val taxFeeCalculate:
  Trade => List[TaxFeeId] => List[(TaxFeeId, BigDecimal)] = {t =>
tids =>
  tids zip (tids map valueAs(t))
}
```

map

zip map

combinators

Pure functions as domain behaviors

```scala
// value a tax/fee for a specific trade
val valueAs:
  Trade => TaxFeeId => BigDecimal = {trade => tid =>
  ((rates get tid) map (_ * principal(trade)))
     getOrElse (BigDecimal(0))
}

// all tax/fees for a specific trade
val taxFeeCalculate:
  Trade => List[TaxFeeId] => List[(TaxFeeId, BigDecimal)] = {t =>
tids =>
  tids zip (tids map valueAs(t))
}
```

map

zip map

combinators

```scala
// enrich a trade with tax/fees and compute net value
val enrichTrade: Trade => Trade = {trade =>
  val taxes = for {

    // get the tax/fee ids for a trade
    taxFeeIds      <- forTrade

    // calculate tax fee values
    taxFeeValues   <- taxFeeCalculate
  }
  yield(taxFeeIds map taxFeeValues)

  val t = taxFeeLens.set(trade, taxes(trade))
  netAmountLens.set(t,
    t.taxFees.map(_.foldl(principal(t))
                            ((a, b) => a + b._2)))
}
```

monadic chaining for
an expressive DSL

```
val trds =
  for {
    trade       <- trades
    trValidated <- validate(trade)
    trEnriched  <- enrich(trValidated)
    trFinal     <- journalize(trEnriched)
  }
  yield trFinal
```

Agenda

- ☑ Immutability and algebraic data types
- ☑ Updating domain models, functionally
- ☑ Type classes & domain modeling
- ☑ **Models of computation**
- ☑ Managing states - the functional way
- ☑ A declarative design for domain service layer

The palette of computation models

the *applicative* model
to construct trades
with accumulating validation
errors

the *monadic* model
that chains computation
and gives us a nice DSL
for trade lifecycle

the *arrow* model
of computation
that generalizes monads

The Arrow model of computation

`A            => M[A]`

a *monad* lifts a value into a computation

`(B => C) => A[B,C]`

an *arrow* lifts a function from input to output into a computation

If the function is of the form `A => M[B]` then we call the arrow a *Kleisli*

Back to our domain model - from Orders to Trades

1. process client orders
2. execute in the market
3. allocate street side trades to client accounts

```scala
def tradeGeneration(market: Market, broker: Account,
  clientAccounts: List[Account]) =
  // client orders
  kleisli(clientOrders)
        // executed at market by broker
    >=> kleisli(execute(market)(broker))
          // and allocated to client accounts
      >=> kleisli(allocate(clientAccounts))
```

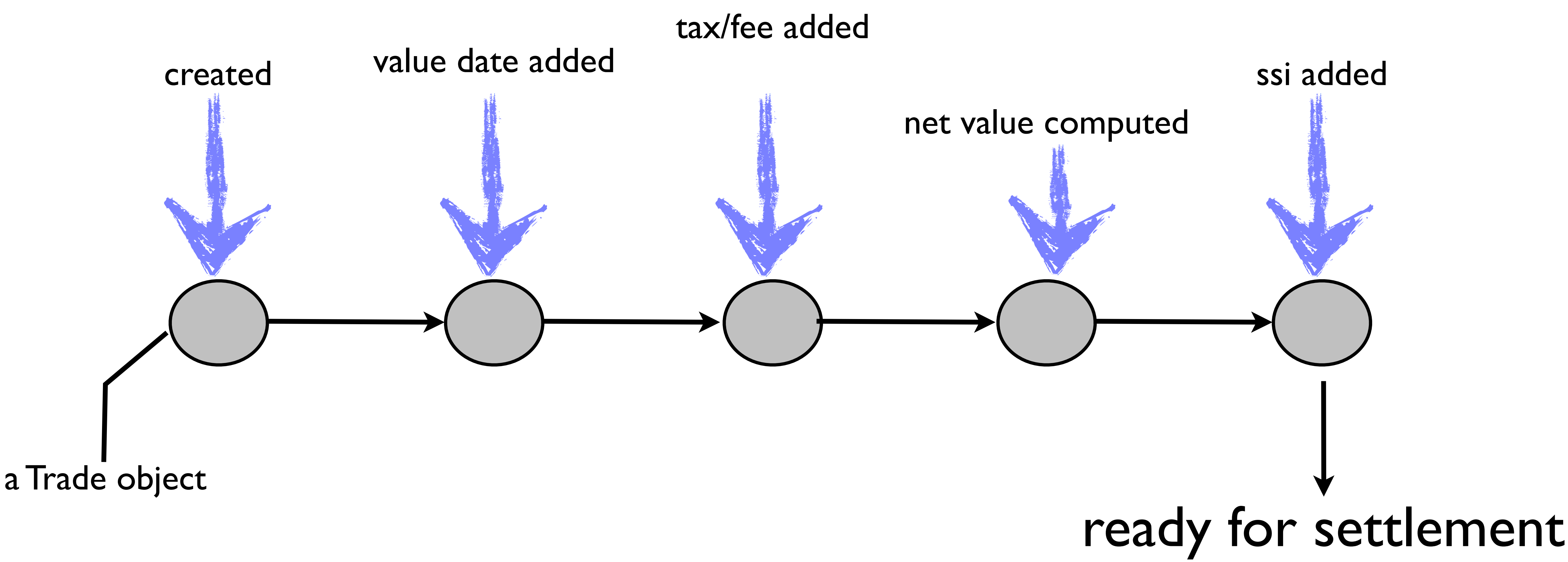
Agenda

- Immutability and algebraic data types
- Updating domain models, functionally
- Type classes & domain modeling
- Models of computation
- **Managing states - the functional way**
- A declarative design for domain service layer

Managing States

- But domain objects don’t exist in isolation
- Need to interact with other objects
- .. and respond to events from the external world
- .. changing from one state to another

A day in the life of a Trade object



What's the big deal ?

All these sound like changing states of a newly created Trade object !!

- but ..
- Changing state through in-place mutation is destructive
- We lose temporality of the data structure
- The fact that a Trade is enriched with tax/ fee NOW does not mean it was not valued at 0 tax SOME TIME BACK

What if we would like to have our system rolled back to THAT POINT IN TIME ?

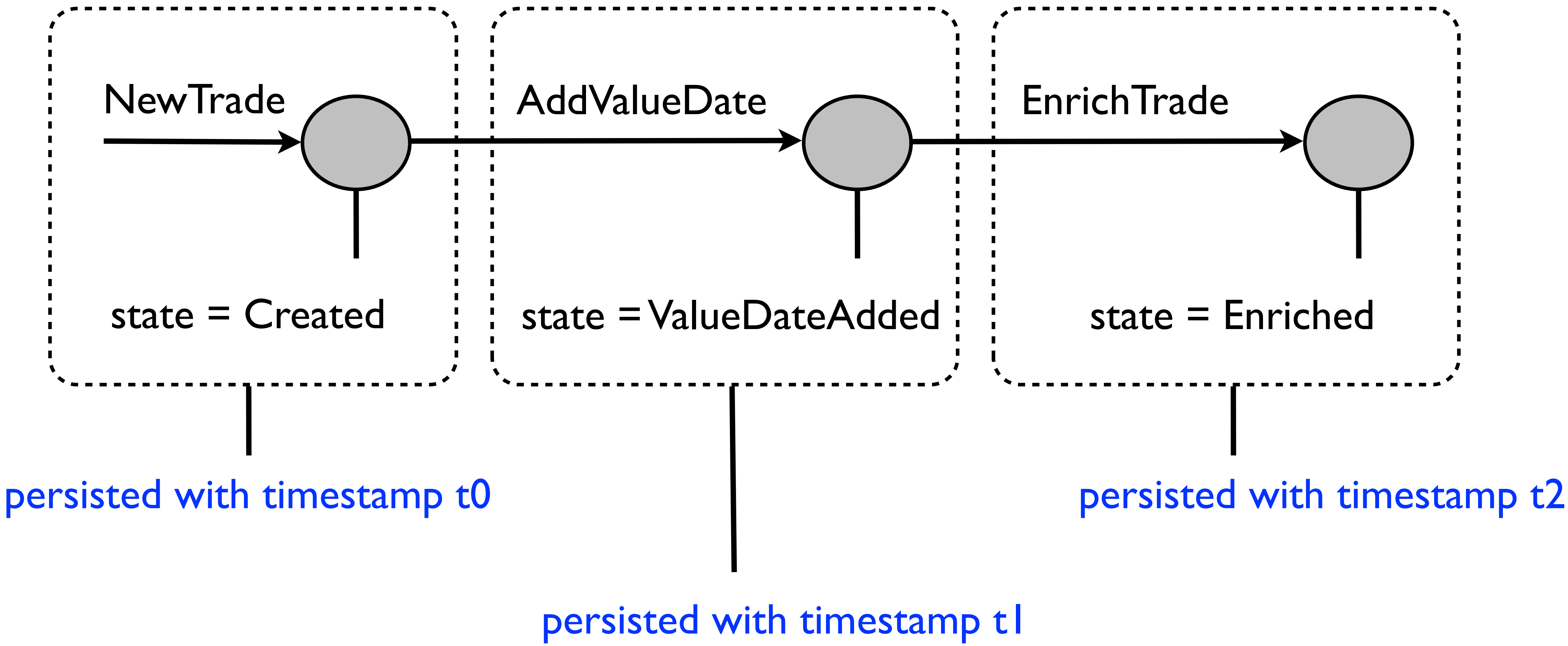
We are just being lossy

- The solution is to keep information in such a way that we have EVERY BIT OF DETAILS stored as the object changes from one state to another
- Enter Event Sourcing

Event Sourcing

- Preserves the temporality of the data structure
- Represent state NOT as a mutable object, rather as a sequence of *domain events* that took place right from the creation till the current point in time
- Decouple the state of the object from its identity. The state changes over time, identity is IMMUTABLE

Domain Events as Behaviors



(t2 > t1 > t0)

.. and since we store every event that hits the system we have the ability to recreate ANY previous state of the system starting from ANY point in time in the past

Events and States

```
sealed trait TradingEvent extends Event

case object NewTrade extends TradingEvent
case object EnrichTrade extends TradingEvent
case object AddValueDate extends TradingEvent
case object SendOutContractNote extends TradingEvent

sealed trait TradeState extends State

case object Created extends TradeState
case object Enriched extends TradeState
case object ValueDateAdded extends TradeState
```

The Current State

- How do you reflect the current state of the domain object ?
  - ✦ start with the initial state
  - ✦ manage all state transitions
  - ✦ persist all state transitions
  - ✦ maintain a snapshot that reflects the current state
  - ✦ you now have the ability to roll back to any earlier state

The essence of Event Sourcing

Store the events in a durable repository. They are the lowest level granular structure that model the actual behavior of the domain. You can always recreate any state if you store events since the creation of the domain object.

The Duality

- Event Sourcing keeps a trail of all events that the abstraction has handled

- Event Sourcing does not ever mutate an existing record

- In functional programming, data structures that keep track of their history are called persistent data structures. Immutability taken to the next level

- An immutable data structure does not mutate data - returns a newer version every time you update it

Event Sourced Domain Models

- Now we have the domain objects receiving domain events which take them from state A to state B
- Will the Trade object have all the event handling logic ?
- What about state changes ? Use the Trade object for this as well ?

Separation of concerns

- The Trade object has the core business of the trading logic. It manifests all state changes in terms of what it contains as data
- But event handling and managing state transitions is something that belongs to the *service* layer of the domain model

Separation of Concerns

- The service layer
  - receives events from the context
  - manages state transitions
  - delegates to Trade object for core business
  - notifies other subscribers
- The core domain layer
  - implements core trading functions like calculation of value date, trade valuation, tax/fee handling etc
  - completely oblivious of the context

Agenda

- ☑ Immutability and algebraic data types
- ☑ Updating domain models, functionally
- ☑ Type classes & domain modeling
- ☑ Models of computation
- ☑ Managing states - the functional way
- ☑ A declarative design for domain service layer

The Domain Service Layer

- Handles domain events and delegates to someone who logs (sources) the events
- May need to maintain an in-memory snapshot of the domain object's current state
- Delegates persistence of the snapshot to other subscribers like Query Services

CQRS

- The service layer ensures a complete decoupling of how it handles updates (commands) on domain objects and reads (queries) on the recent snapshot
- Updates are persisted as events while queries are served from entirely different sources, typically from read slaves in an RDBMS

In other words, the domain service layer acts as a state machine

Making it Explicit

- Model the domain service layer as an FSM (Finite State Machine) - call it the `TradeLifecycle`, which is totally segregated from the `Trade` domain object
- .. and let the FSM run within an actor model based on asynchronous messaging
- We use Akka's FSM implementation

FSM in Akka

- Actor based
  - available as a mixin for the Akka actor
  - similar to Erlang `gen_fsm` implementation
  - as a client you need to implement the state transition rules using a declarative syntax based DSL, all heavy lifting done by Akka actors

initial state

match on event AddValueDate

start state of Trade object

```
startWith(Created, trade)

when(Created) {
  case Event(e@AddValueDate, data) =>
    log.map(_.appendAsync(data.refNo, Created, Some(data), e))
    val trd = addValueDate(data)
    gossip(trd)
    goto(ValueDateAdded) using trd forMax(timeout)
}
```

notify observers

move to next state of Trade lifecycle

update data structure

log the event

```
startWith(Created, trade)

when(Created) {
  case Event(e@AddValueDate, data) =>
    log.map(_.appendAsync(data.refNo, Created, Some(data), e))
    val trd = addValueDate(data)
    gossip(trd)
    goto(ValueDateAdded) using trd forMax(timeout)
}
```

Handle events & process data updates and state changes

Log events (event sourcing)

Notifying Listeners

State change - functionally

```
// closure for adding a value date
val addValueDate: Trade => Trade = {trade =>
  valueDateLens.set(trade, ..)
}
```

pure referentially transparent implementation

Notifying Listeners

- A typical use case is to send updates to the Query subscribers as in CQRS

- The query is rendered from a separate store which needs to be updated with all changes that go on in the domain model

Notifications in Akka FSM

```scala
trait FSM[S, D] extends Listeners {
  //..
}
```

Listeners is a generic trait to implement listening capability on an Actor

```scala
trait Listeners {self: Actor =>
  protected val listeners = ..
  //..
  protected def gossip(msg: Any) =
    listeners foreach (_ ! msg)
  //..
}
```

```scala
class TradeLifecycle(trade: Trade, timeout: Duration,
  log: Option[EventLog])
  extends Actor with FSM[TradeState, Trade] {
  import FSM._

  startWith(Created, trade)

  when(Created) {
    case Event(e@AddValueDate, data) =>
      log.map(_.appendAsync(data.refNo, Created, Some(data), e))
      val trd = addValueDate(data)
      gossip(trd)
      goto(ValueDateAdded) using trd forMax(timeout)
  }

  when(ValueDateAdded) {
    case Event(StateTimeout, _) =>
      stay

    case Event(e@EnrichTrade, data) =>
      log.map(_.appendAsync(data.refNo, ValueDateAdded, None,  e))
      val trd = enrichTrade(data)
      gossip(trd)
      goto(Enriched) using trd forMax(timeout)
  }
  //..
}
```

domain service as
a Finite State Machine

- declarative
- actor based
- asynchronous
- event sourced

```scala
// 1. create the state machine
val tlc =
  system.actorOf(Props(
    new TradeLifecycle(trd, timeout.duration, Some(log))))

// 2. register listeners
tlc ! SubscribeTransitionCallBack(qry)

// 3. fire events
tlc ! AddValueDate
tlc ! EnrichTrade
val future = tlc ? SendOutContractNote

// 4. wait for result
finalTrades += Await.result(future, timeout.duration)
               .asInstanceOf[Trade]
```

```scala
// check query store
val f = qry ? QueryAllTrades
val qtrades = Await.result(f,timeout.duration)
                   .asInstanceOf[List[Trade]]

// query store in sync with the in-memory snapshot
qtrades should equal(finalTrades)
```

Summary

- Functional programming is *programming with functions* - design your domain model with function composition as the primary form of building abstractions
- Have a clear delineation between side-effects and pure functions. Pure functions are easier to debug and reason about
- Immutability rocks - makes it way easier to share abstractions across threads
- With immutable domain abstractions you can make good use of transactional references through an STM

Thank You!