

# Functional & Event Driven

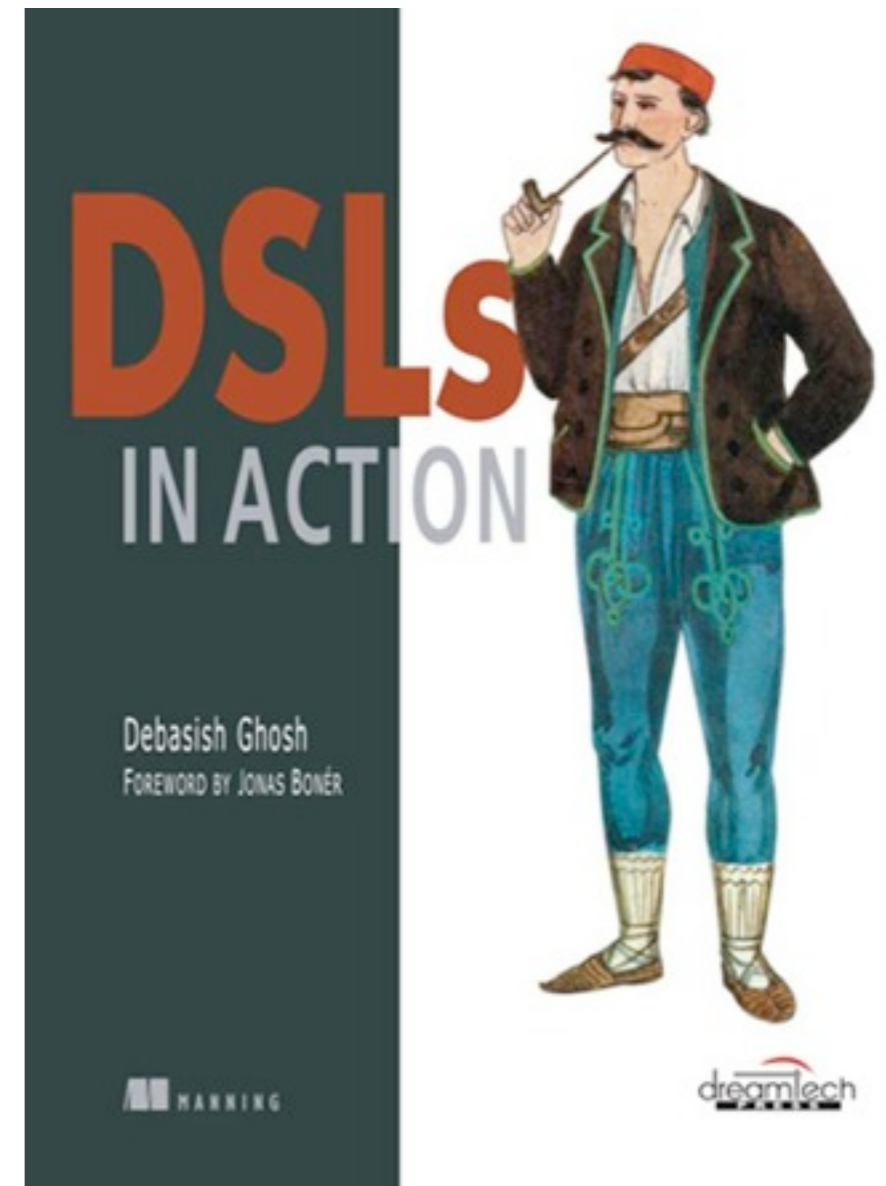
another approach to domain modeling

# Debasish Ghosh

@debasishg on Twitter

code @  
<http://github.com/debasishg>

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



# Agenda

- Domain Models
- Why Functional Domain Models
- Event based architectures - Event Sourcing
- Duality between functional models & even based models
- A sample event sourced domain model implementation

# 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

# Why Functional ?

- Pure with localization of side-effects
- Ability to reason about your model
- Expressive & Declarative
- Immutable

# A Functional Domain Model

- Built on (mostly) pure abstractions
- Pure functions to model domain behaviors
- Immutable artifacts
- .. and you get some parallelism for free

# Immutability

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

# Representation as ADTs

```
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  
  
}
```



# Representation as ADTs

```
// 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
```

# Pure functions as domain behaviors

```
// get the list of tax/fees applicable for this trade
// depends on the market
val forTrade: Trade => Option[List[TaxFeeId]] = {trade =>
  taxFeeForMarket.get(trade.market) <+> taxFeeForMarket.get(Other)
}
```

# Pure functions as domain behaviors

```
// enrich a trade with tax/fees and compute net value
val enrichTrade: Trade => Trade = {trade =>
  val taxes = for {
    taxFeeIds      <- forTrade // get the tax/fee ids for a trade
    taxFeeValues   <- taxFeeCalculate // calculate tax fee values
  }
  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)))
}
```

# Pure functions as domain behaviors

```
// combinator to 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))  
}
```

# Composability

Secret sauce ? Functions compose. We can pass functions as first class citizens in the paradigm of functional programming. We can use them as return values and if we can make them pure we can treat them just as mathematically as you would like to.

# 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:
  - $\text{get } S \rightarrow T$
  - $\text{putBack } (S \times T) \rightarrow S$

# 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 its 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 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))

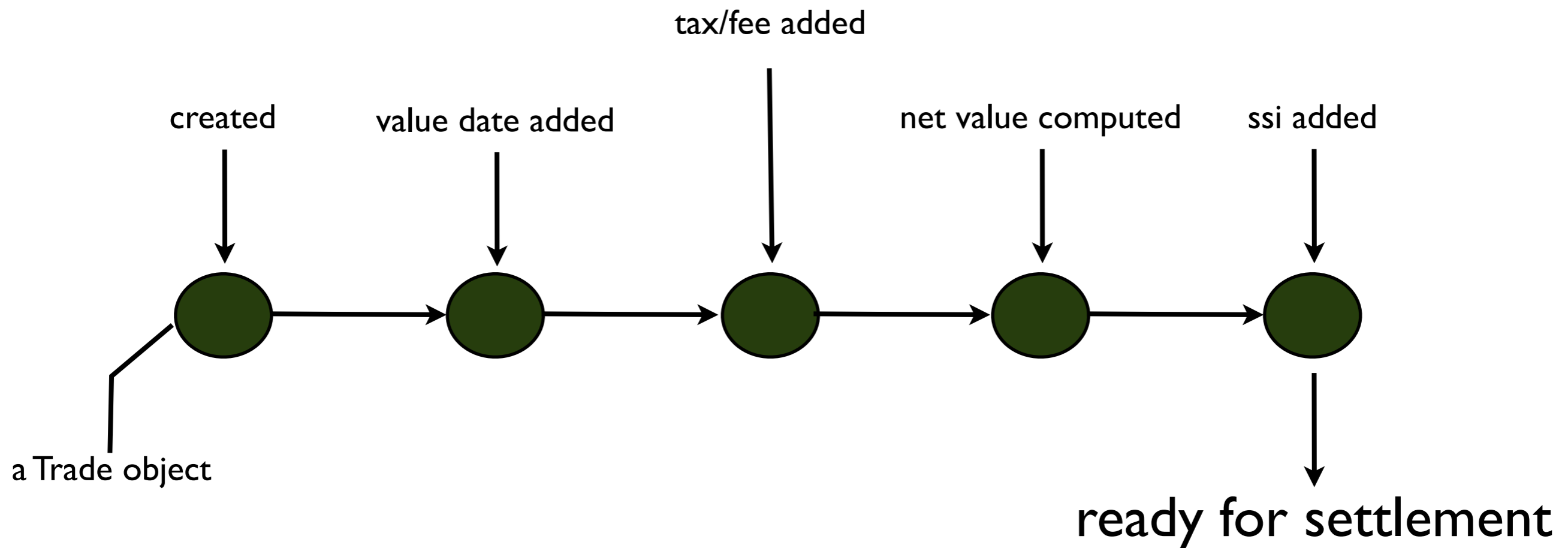
// add net amount
val netAmountLens: Lens[Trade, Option[BigDecimal]] =
  Lens((t: Trade) => t.netAmount,
       (t: Trade, n: Option[BigDecimal]) => t.copy(netAmount = n))

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

# 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**



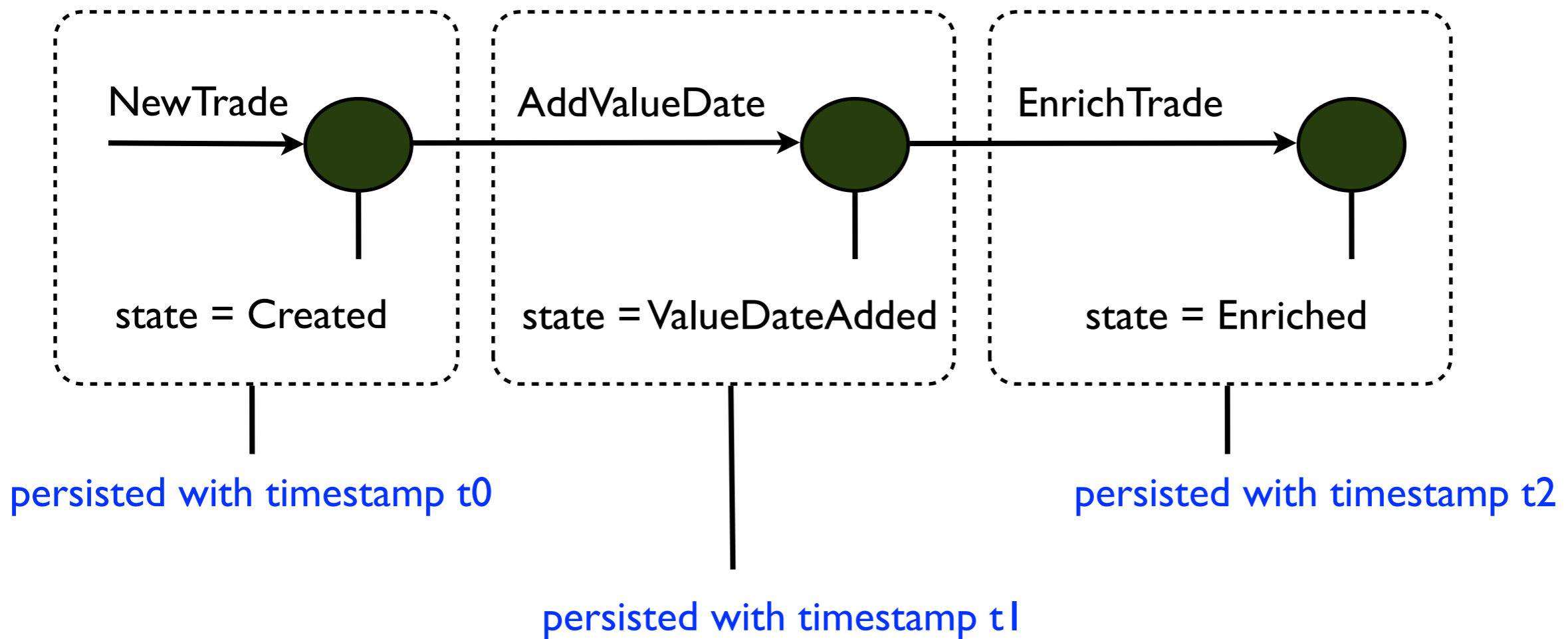


- The solution is to keep information in such a way that we have **EVERY BIT OF DETAILS** stored as the object changes from one state to another
- Enter Event Sourcing

# 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



$$(t_2 > t_1 > t_0)$$

.. 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

# 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

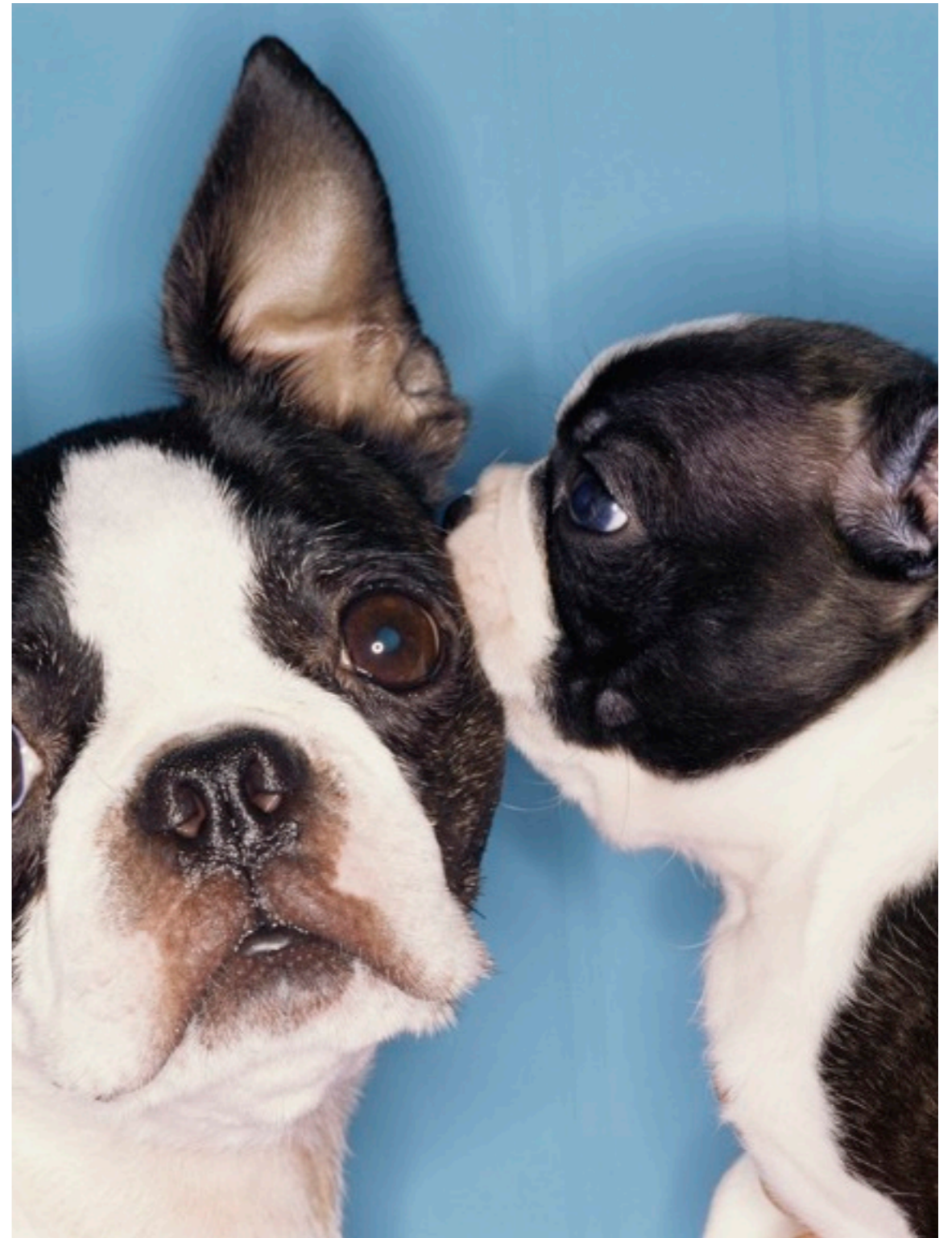
```
// closure for adding a value date  
val addValueDate: Trade => Trade = {trade =>  
  valueDateLens.set(trade, ..)  
}
```



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

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

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

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

# Event Logging

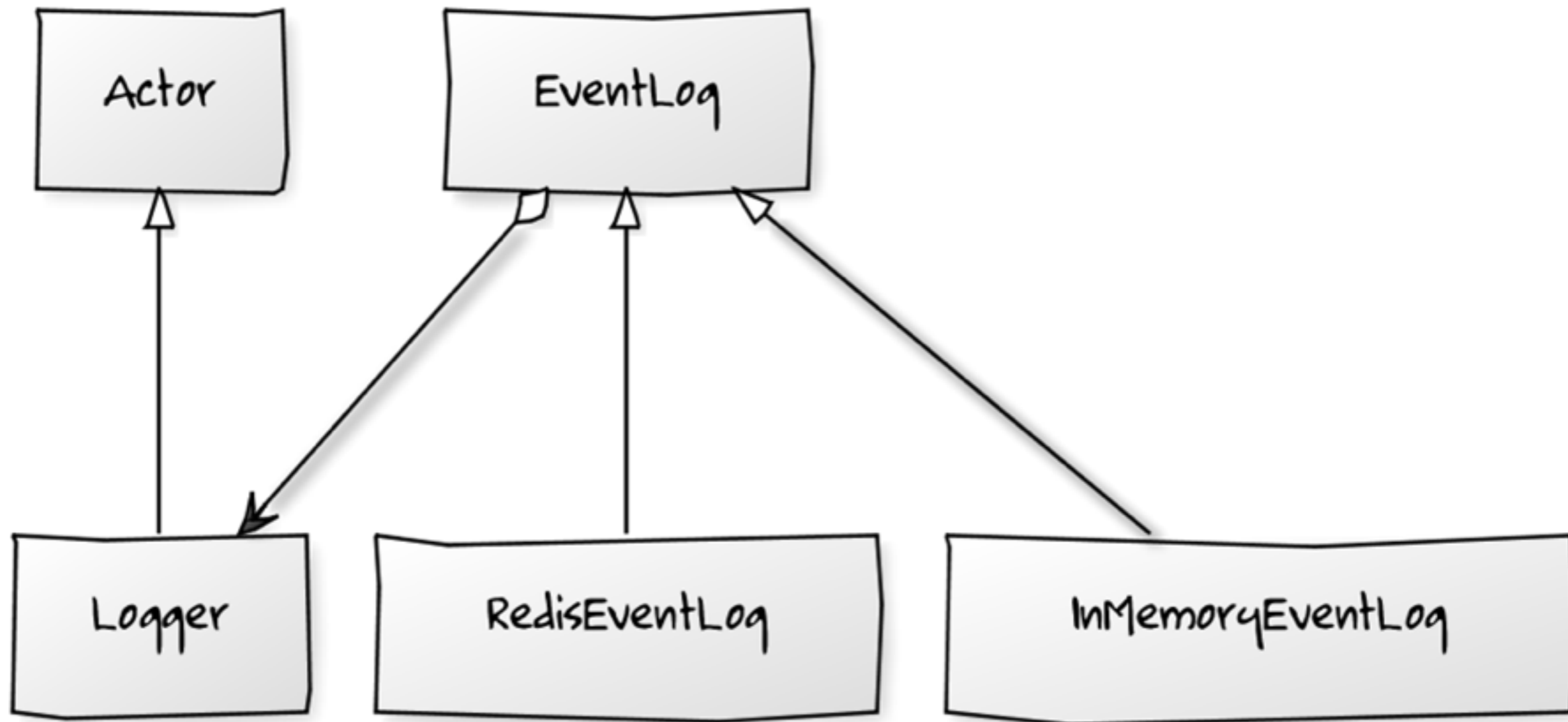
- Log asynchronously
- Persist the current state along with the data at this state
- Log the event `e`

|  
`log.map(_._appendAsync(data.refNo,  
Created,  
Some(data), e))`

# Event Logging

```
case class EventLogEntry(entryId: Long,  
  objectId: String,  
  inState: State,  
  withData: Option[Any], event: Event)
```

```
trait EventLog extends Iterable[EventLogEntry] {  
  def iterator: Iterator[EventLogEntry]  
  def iterator(fromEntryId: Long): Iterator[EventLogEntry]  
  def appendAsync(id: String, state: State,  
    data: Option[Any], event: Event): Future[EventLogEntry]  
}
```



# Log Anywhere



# Order preserving logging

```
class RedisEventLog(clients: RedisClientPool, as: ActorSystem)
  extends EventLog {
  val loggerActorName = "redis-event-logger"

  // need a pinned dispatcher to maintain order of log entries
  lazy val logger =
    as.actorOf(Props(new Logger(clients)).withDispatcher("my-pinned-dispatcher"),
              name = loggerActorName)

  //..
}

akka {
  actor {
    timeout = 20
    my-pinned-dispatcher {
      executor = "thread-pool-executor"
      type = PinnedDispatcher
    }
  }
}
```

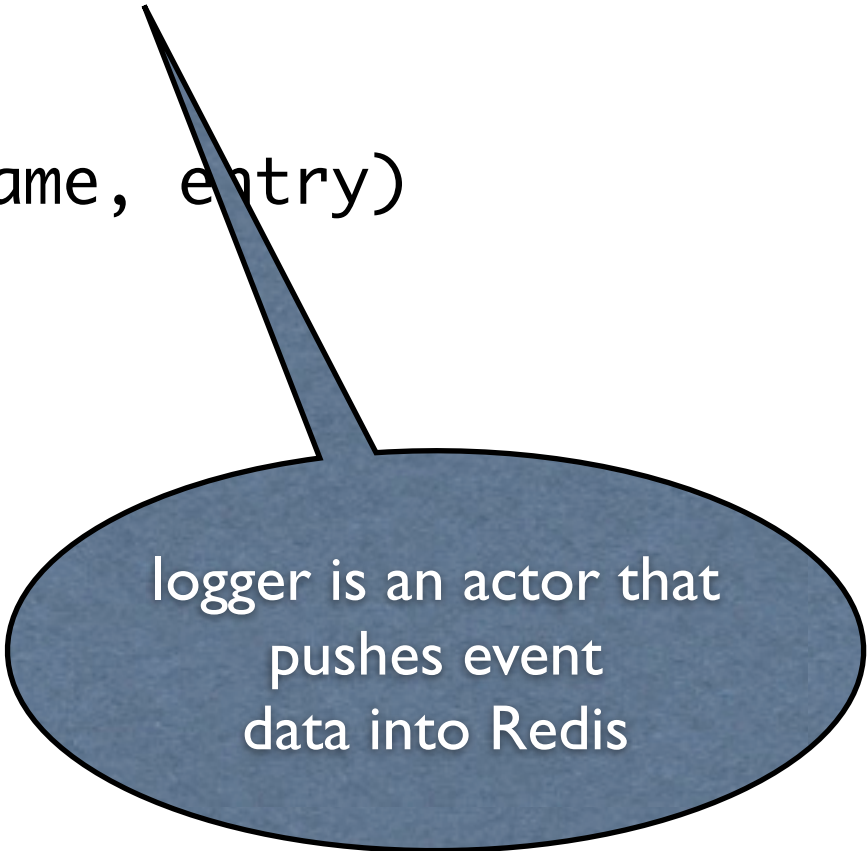
```

case class LogEvent(objectId: String, state: State,
  data: Option[Any], event: Event)

class Logger(clients: RedisClientPool) extends Actor {
  implicit val format =
    Format {case l: EventLogEntry => serializeEventLogEntry(l)}
  implicit val parseList =
    Parse[EventLogEntry](deSerializeEventLogEntry(_))

  def receive = {
    case LogEvent(id, state, data, event) =>
      val entry = EventLogEntry(RedisEventLog.nextId(), id, state,
data, event)
      clients.withClient {client =>
        client.lpush(RedisEventLog.logName, entry)
      }
      sender ! entry
    //..
  }
  //..
}

```



logger is an actor that pushes event data into Redis

```
def appendAsync(id: String, state: State,  
  data: Option[Any], event: Event): Future[EventLogEntry] =  
  
  (logger ? LogEvent(id, state, data, event))  
    .asInstanceOf[Future[EventLogEntry]]
```



Non-blocking : returns  
a Future

```

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

```
// 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]
```

```
// 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

- Event sourcing is an emerging trend in architecting large systems
- Focuses on immutability and hence plays along well with functional programming principles
- Event stores are append only - hence n locks, no synchronization - very fast at the event storage layer

# Summary

- Complicated in-memory snapshot may have some performance overhead, but can be managed with a good STM implementation
- In designing your domain model based on event sourcing, make your core model abstractions meaty enough while leaving the service layer with the responsibilities of managing event handling and state transitions



**Thank You!**