goto; conference

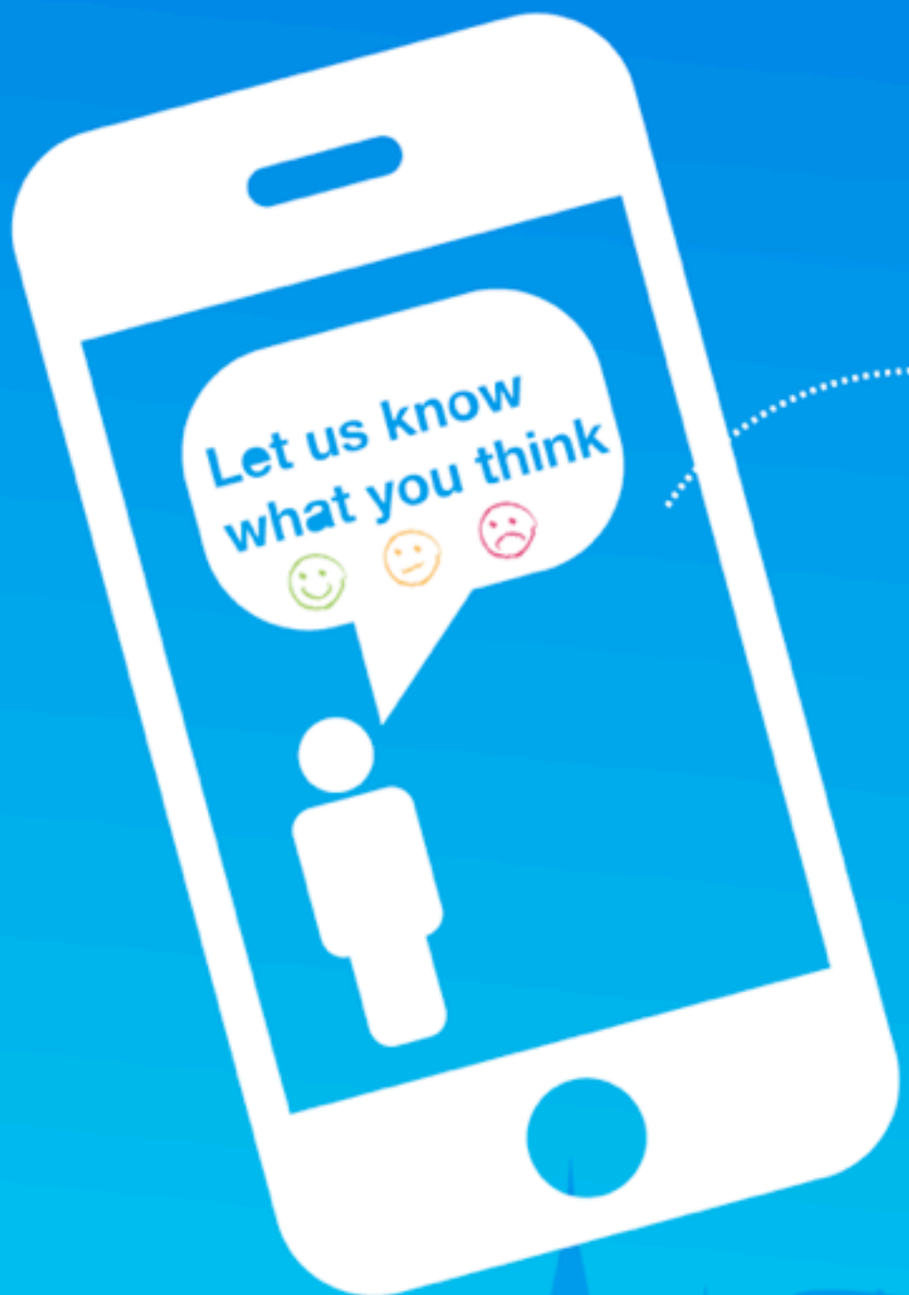
From Functional to Reactive

patterns in domain modeling

Debasish Ghosh
@debasishg

rsation #gotocph

goto; conference



Click ‘engage’
to rate sessions
and ask questions

ation #gotocph

Domain Modeling

(Functional) Domain Modeling

(Functional) Domain Modeling

(Responsive)

(Functional) Domain Modeling

(Responsive) (Elastic)

(Functional) Domain Modeling

(Responsive) (Elastic) (Resilient)

(Functional) Domain Modeling

(Responsive) (Elastic) (Resilient)

(Reactive)

What is a domain model ?

A domain model in problem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain. It does not describe the solutions to the problem.

Wikipedia (http://en.wikipedia.org/wiki/Domain_model)

The Functional Lens ..

"domain API evolution through algebraic composition"

The Functional Lens ..

(Reactive)

"domain API evolution through algebraic composition"

Agenda

- Formalizing a domain model
- Domain model *algebra*
- From functional to *algebraically* reactive
- Beyond algebra
- Actors and domain models
- Reactive streams - *typesafe & compositional*

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a time-

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

to thread through tracing metadata (à la Google Dapper [20]).

Your domain model is a function

Your domain model is a function

Your domain model is a

collection of functions

Your domain model is a

collection of functions

some simpler models are ..

Conference Management System

Bounded Context A

Conference reservations

Domain model A
- Ubiquitous language
- Entities
- Value objects
- Services

Code
Schemas
Other artifacts

Bounded Context B

Program management

Domain model B
- Ubiquitous language
- Entities
- Value objects
- Services

Code
Schemas
Other artifacts

Bounded Context C

Badge printing

Domain model C
- Ubiquitous language
- Entities
- Value objects
- Services

Code
Schemas
Other artifacts

https://msdn.microsoft.com/en-us/library/jj591560.aspx

A Bounded Context

- has a consistent vocabulary
- a set of domain behaviors modeled as functions on domain objects implemented as types
- related behaviors grouped as modules

$$\text{Domain Model} = \bigcup_{(i)} \text{Bounded Context}(i)$$

$$\text{Domain Model} = \bigcup_{(i)} \text{Bounded Context}(i)$$

$$\text{Bounded Context} = \{\ f(x) \mid p(x) \in \text{Domain Rules}\ \}$$

Domain Model = $\bigcup_{(i)}$ Bounded Context(i)

Bounded Context = { f(x) | p(x) $\in$ Domain Rules }

- domain *function*
- on an object of *type* x
- *composes* with other functions
- *closed* under composition

- business *rules*

- Functions / Morphisms
- Types / Sets
- Composition
- Rules / Laws

- Functions / Morphisms
- Types / Sets
- Composition
- Rules / Laws

algebra

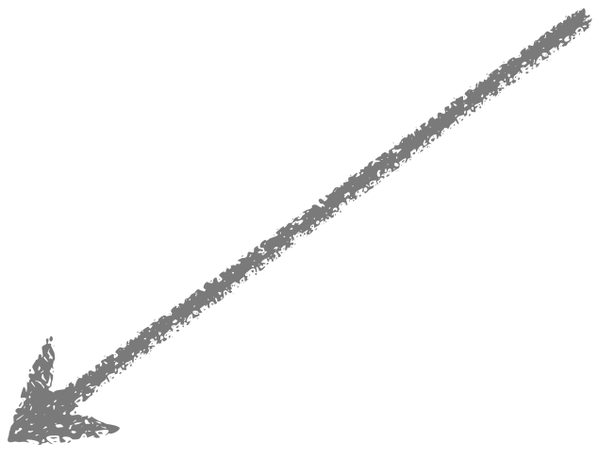
Domain Model Algebra

Domain Model Algebra

(algebra of types, functions & laws)

Domain Model Algebra

(algebra of types, functions & laws)

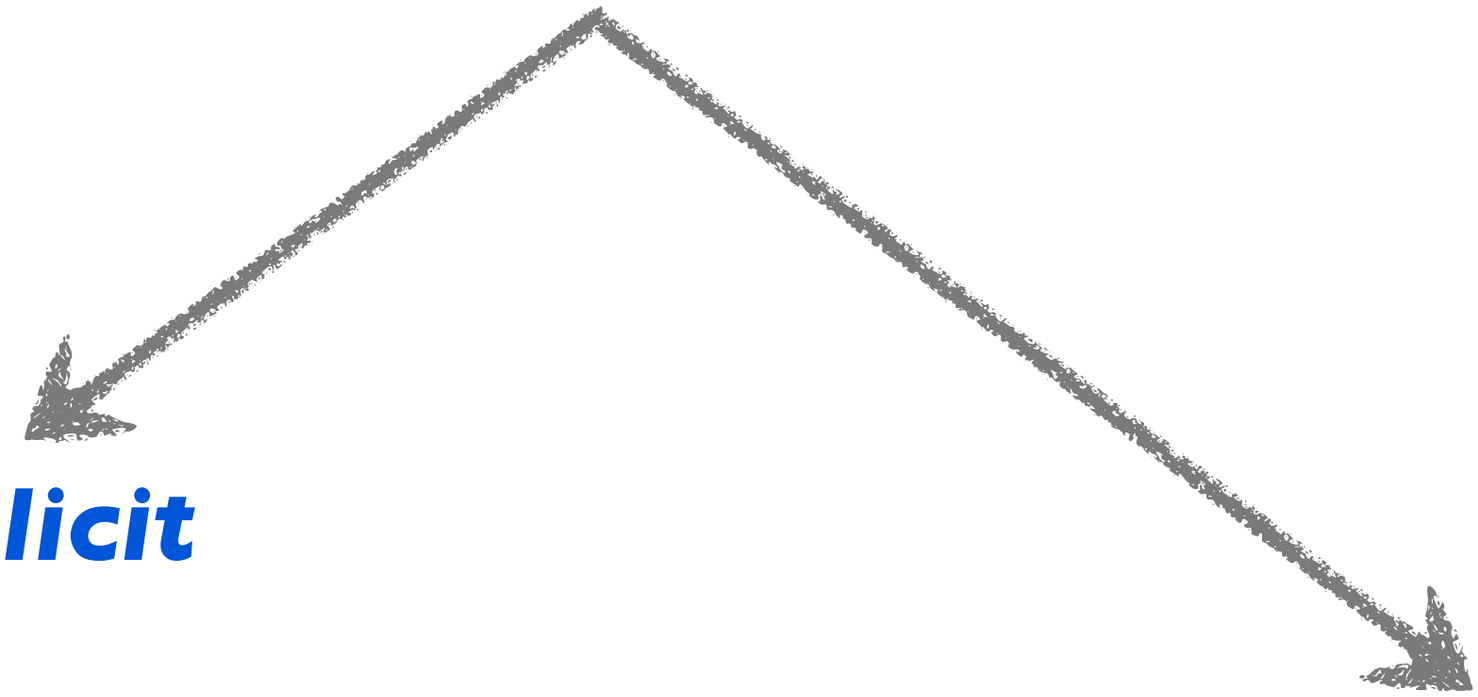


explicit

- types
- type constraints
- expression in terms of other generic algebra

Domain Model Algebra

(algebra of types, functions & laws)

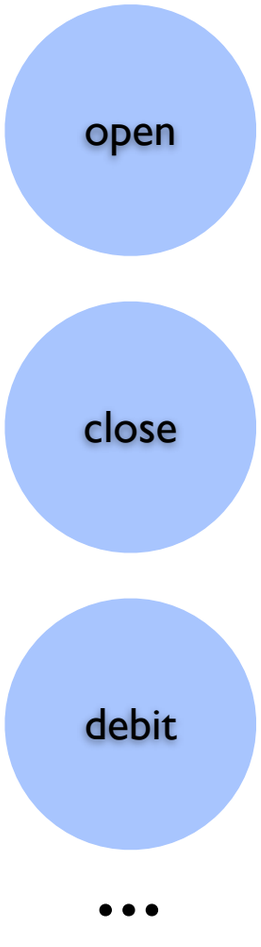


explicit

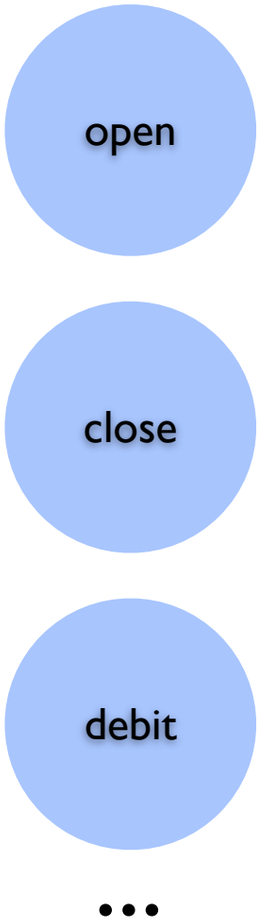
- types
- type constraints
- expression in terms of other generic algebra

verifiable

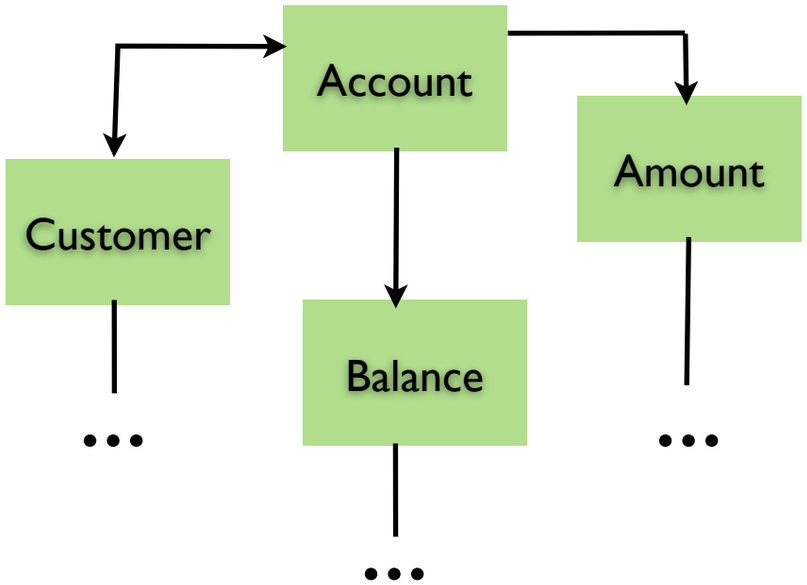
- type constraints
- more constraints if you have DT
- algebraic property based testing



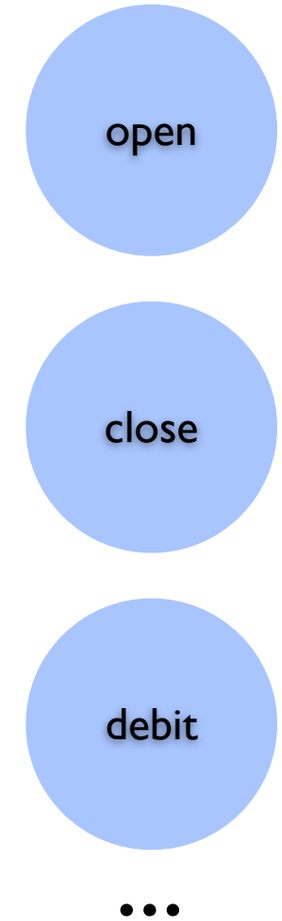
Domain
Behaviors



Domain
Behaviors

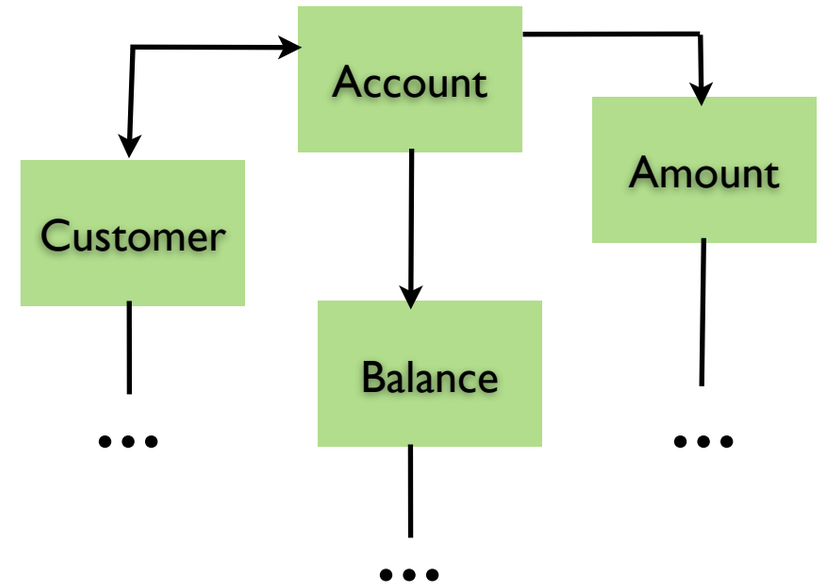


Domain Types

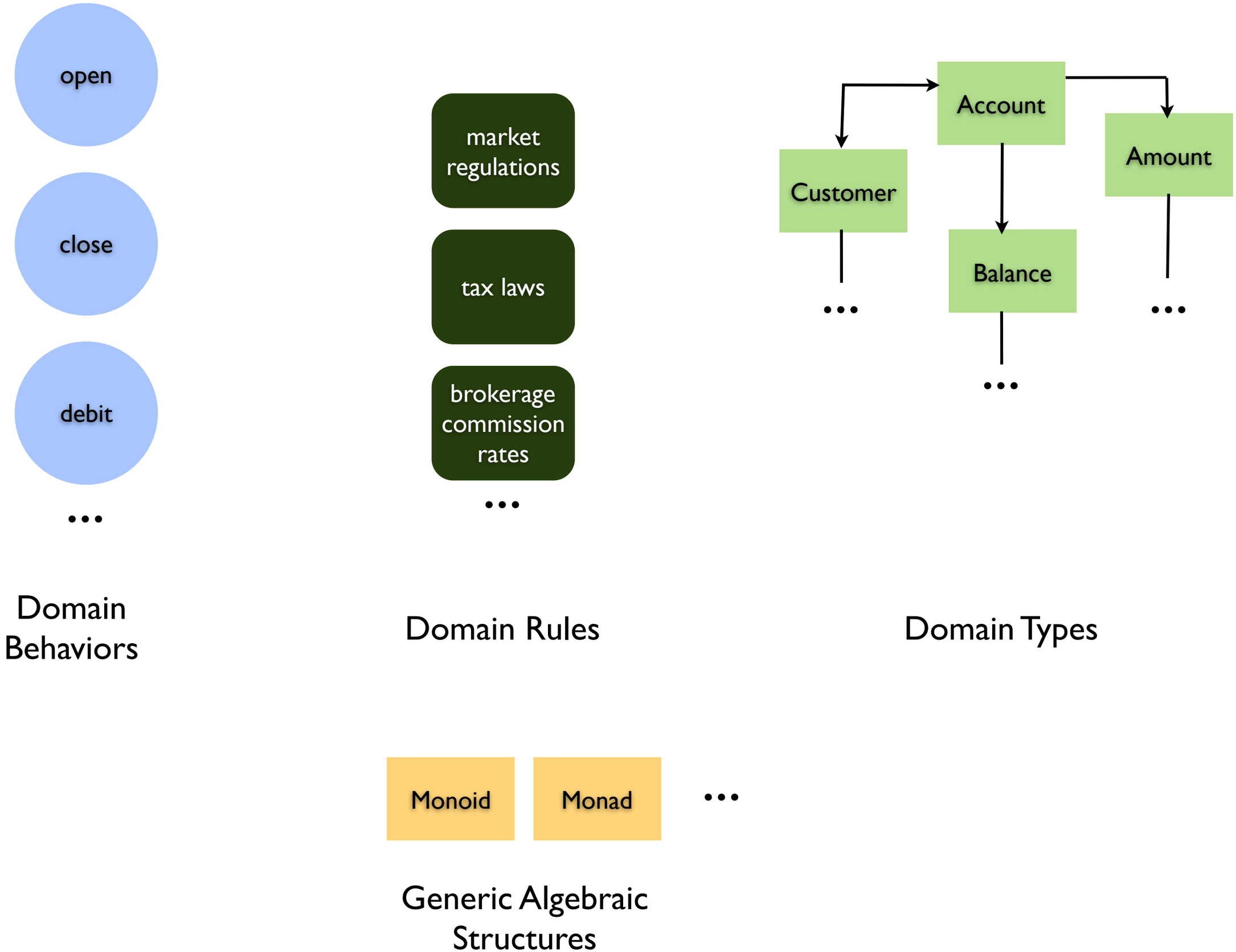


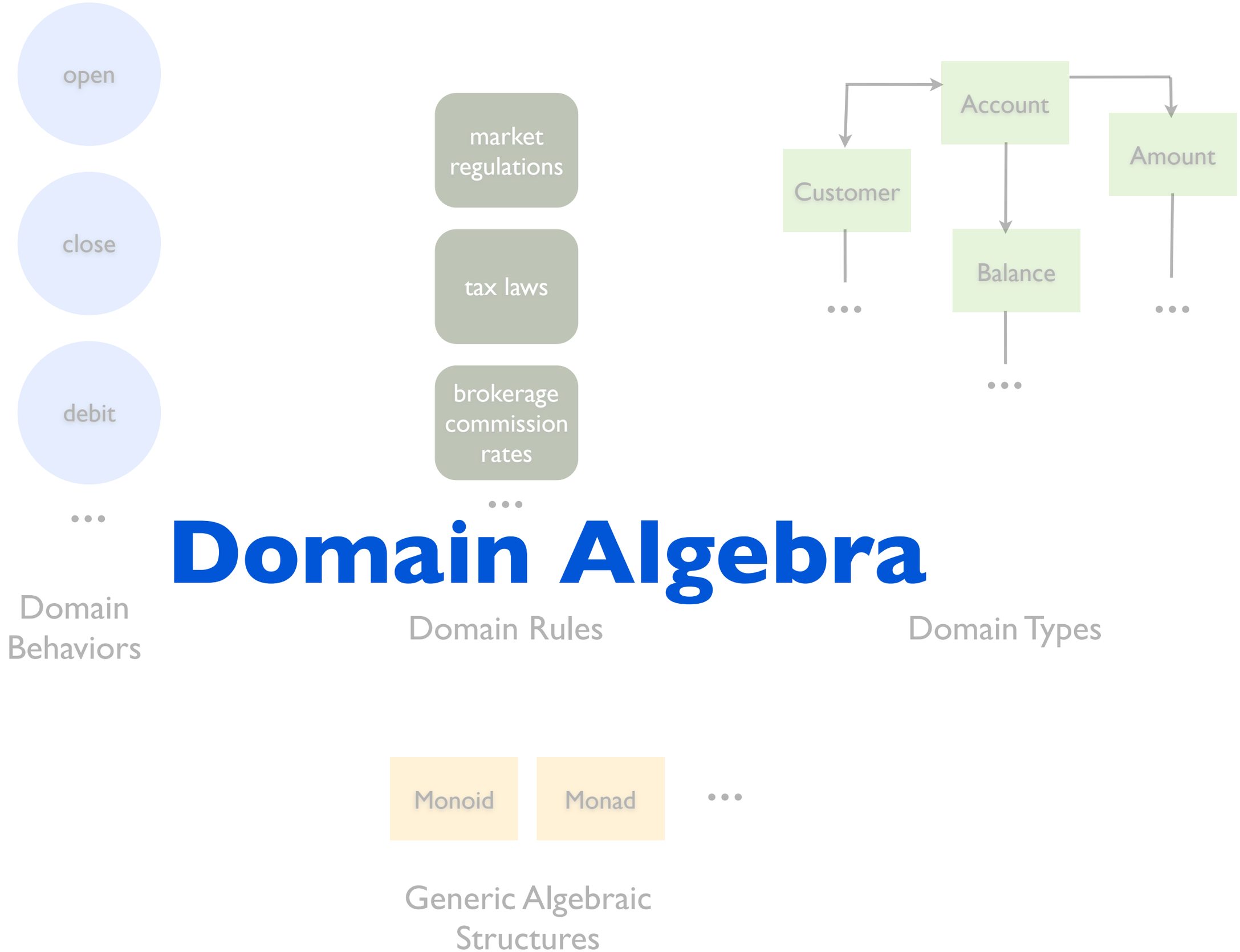
Domain Behaviors

Domain Rules



Domain Types


open
close
debit
...
Domain Behaviors
market regulations
tax laws
brokerage commission rates
...
Domain Rules
Account
Customer
Amount
Balance
...
...
...
Domain Types
Monoid
Monad
...
Generic Algebraic Structures


open
close
debit
...
market regulations
tax laws
brokerage commission rates
...
Account
Customer
Amount
Balance
...
...
...
Domain Algebra
Domain Behaviors
Domain Rules
Domain Types
Monoid
Monad
...
Generic Algebraic Structures

Domain Algebra

Domain Model = $\bigcup_{(i)}$ ~~Bounded Context~~(i)

Domain Algebra

~~Bounded Context~~ = { f(x) | p(x) ∈ Domain Rules }

- domain *function*
- on an object of *type* x
- *composes* with other functions
- *closed* under composition

- business *rules*

Bounded Context

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Algebra is the binding contract

Bounded Context

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Reactive

Algebra is the binding contract

Conference Reservations

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra A

Badge Printing

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra C

Program Management

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra B

- Algebras don’t unify across bounded contexts
- Decoupled in space and time
- Separate vocabulary
- Types break down

Conference Reservations

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra A

Protocols

Badge Printing

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra C

Program Management

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra B

Conference Reservations

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra A

Reactive

Protocols

Badge Printing

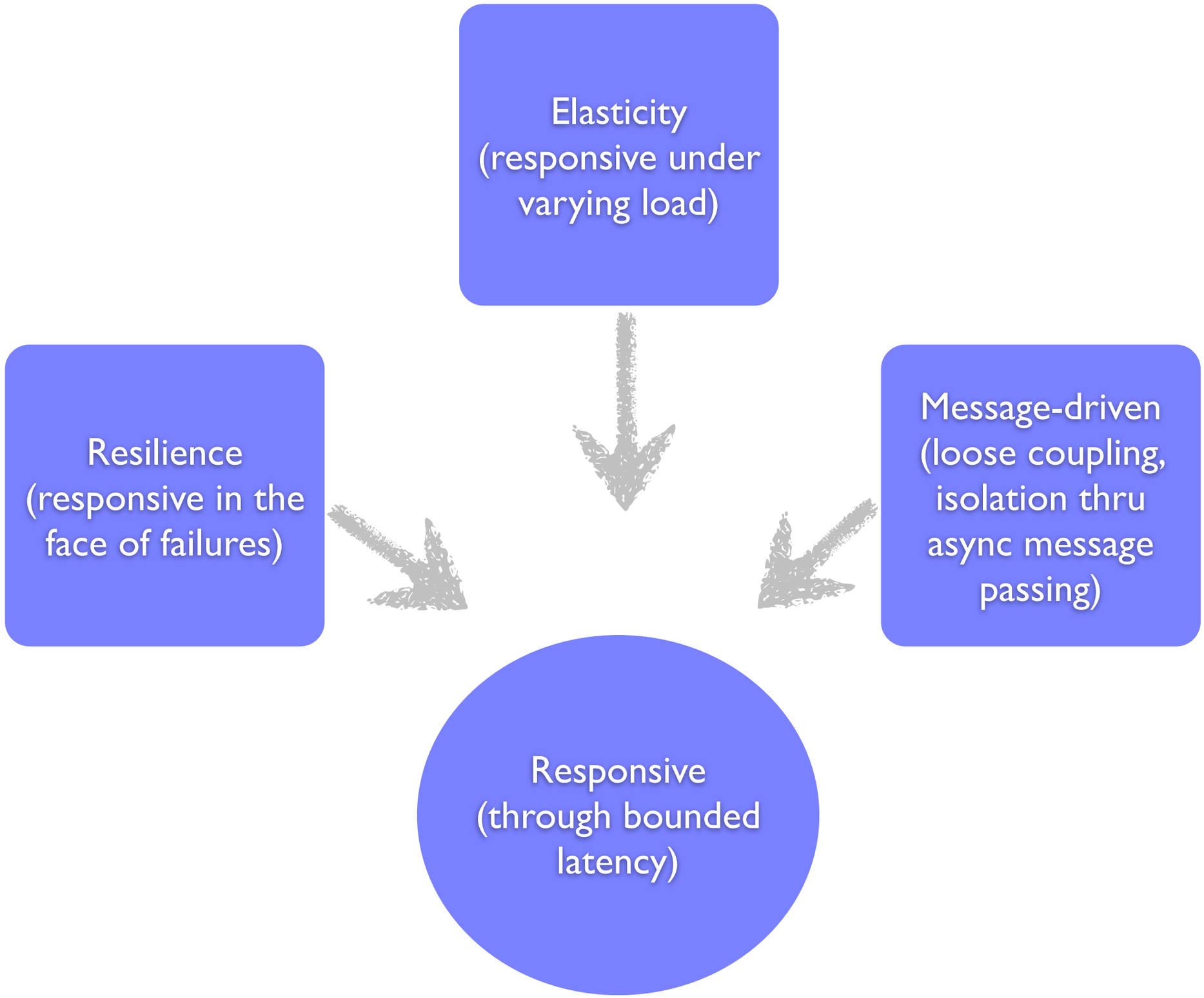
- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra C

Program Management

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra B



Being Reactive

```scala
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Module Name

```
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Module Name

Parameterized on types

```
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Module Name

Parameterized on types

Operation return type - either a successfully constructed type or a list of errors

```
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Module Name

Parameterized on types

Operation return type - either a successfully constructed type or a list of errors

```
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Operations - domain behaviors

Module Name

Parameterized on types

Operation return type - either a successfully constructed type or a list of errors

```
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(no: String, name: String, rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOp[Account]

  def close(no: String, closeDate: Option[Date]): AccountOp[Account]

  def debit(no: String, amount: Amount): AccountOp[Account]

  def credit(no: String, amount: Amount): AccountOp[Account]

  //..
}
```

Operations - domain behaviors

explicit & verifiable algebra

- Parametric - parameterized on types
- Statically Typed
- Modular and hence unit testable
- Composable

Composable

```
def transfer(from: String, to: String, amount: Amount)
  : AccountOp[(Account, Account)] = for {

    a <- debit(from, amount)
    b <- credit(to, amount)

  } yield ((a, b))
```

Composable

```
trait BankingService[Account, Amount, Balance]
  extends AccountService[Account, Amount, Balance]
    with InterestPostingService[Account, Amount]
    with InterestCalculation[Account, Amount]
    with TaxCalculation[Amount]
```

```scala
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountRepository => AccountOp[Account]

  //..
}
```

```scala
trait AccountService[Account, Amount, Balance] {

  type AccountOp[A] = NonEmptyList[String] \/ A

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountRepository => AccountOp[Account]

  //..
}
```

change the algebra to add functionality

```scala
trait AccountService[Account, Amount, Balance] {

  type Valid[A] = NonEmptyList[String] \/ A

  type AccountOp[A] = Kleisli[Valid, AccountRepository, A]

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountOp[Account]

  //..
}
```

more algebra, more functionality, more succinct

Being Reactive

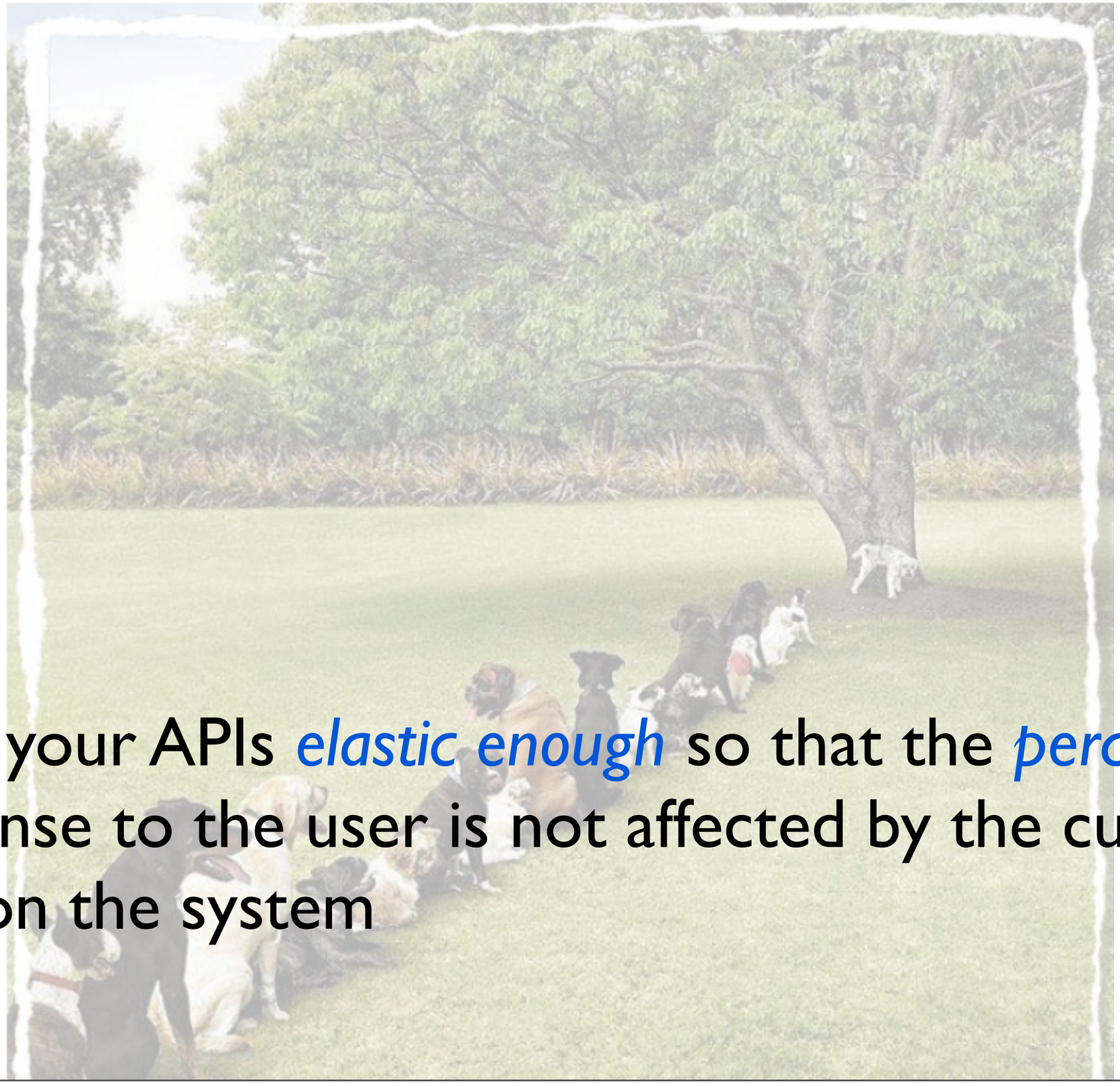
- Design should not have any *contention* or central bottlenecks that tend to hamper the *progress* of the system

Being Reactive

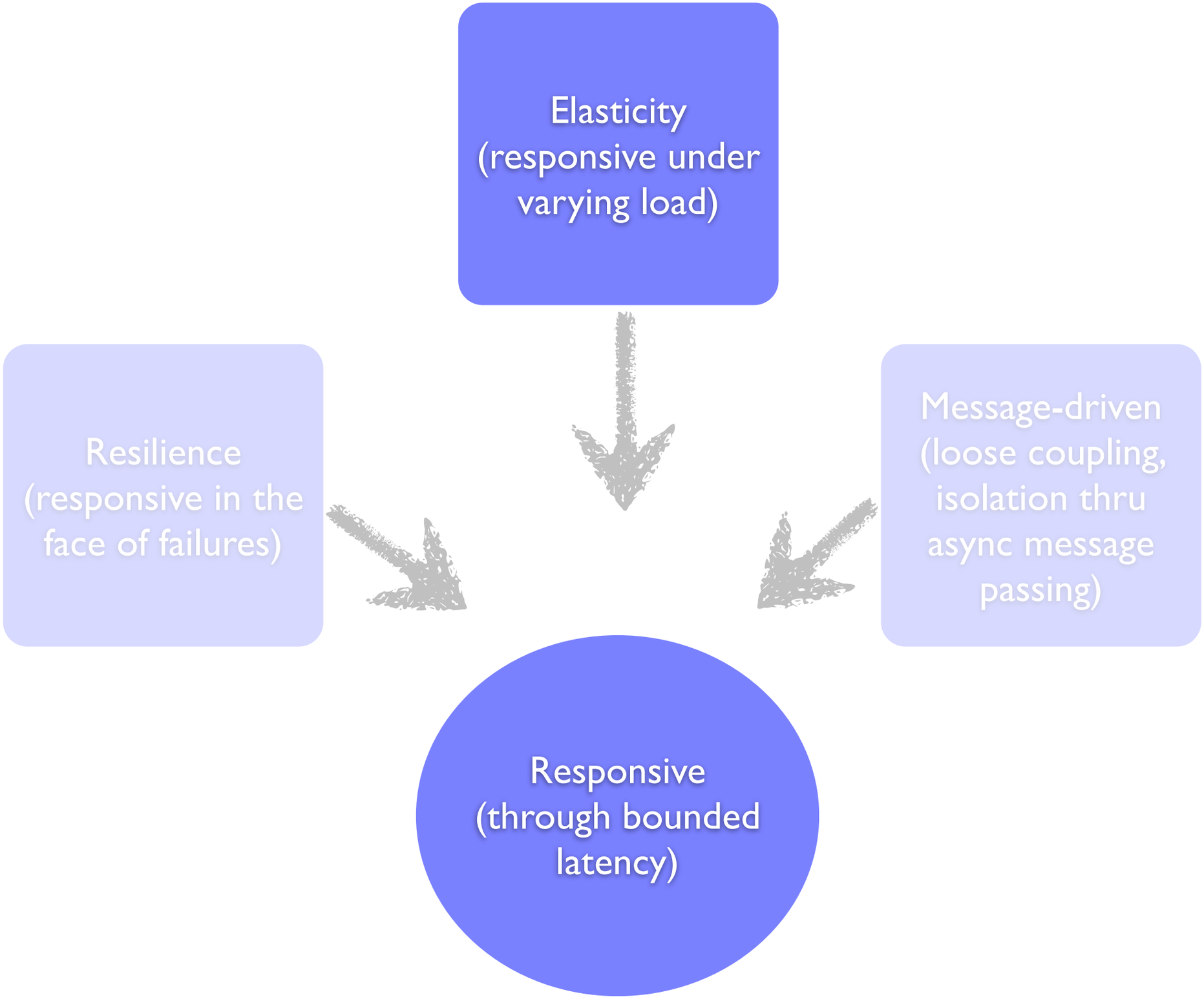
- If your domain service publishes APIs that does *blocking* calls to underlying databases and blocks the central thread of user interaction, you face the specter of *unbounded latency*

Blocking Kills

Blocking Kills



Make your APIs *elastic enough* so that the *perceived* response to the user is not affected by the current load on the system


Elasticity
(responsive under
varying load)
Resilience
(responsive in the
face of failures)
Message-driven
(loose coupling,
isolation thru
async message
passing)
Responsive
(through bounded
latency)

Being Reactive

- Without foregoing the benefits of algebraic reasoning with types

Enter Futures ..

Enter Futures ..

- A future is the essence of *asynchronous non blocking computation*

Enter Futures ..

- A future is the essence of *asynchronous non blocking computation*
- Futures *compose*

Enter Futures ..

- A future is the essence of *asynchronous non blocking computation*
- Futures *compose*
- Futures have an *algebra*

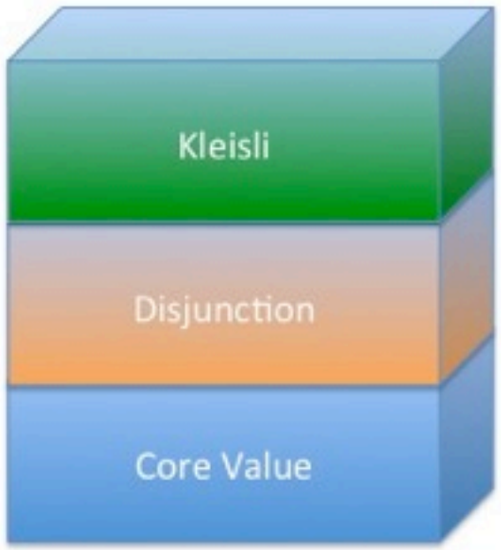
Enter Futures ..

- A future is the essence of *asynchronous non blocking computation*
- Futures *compose*
- Futures have an *algebra*
- Organize concurrent code around futures safely and in a compositional way

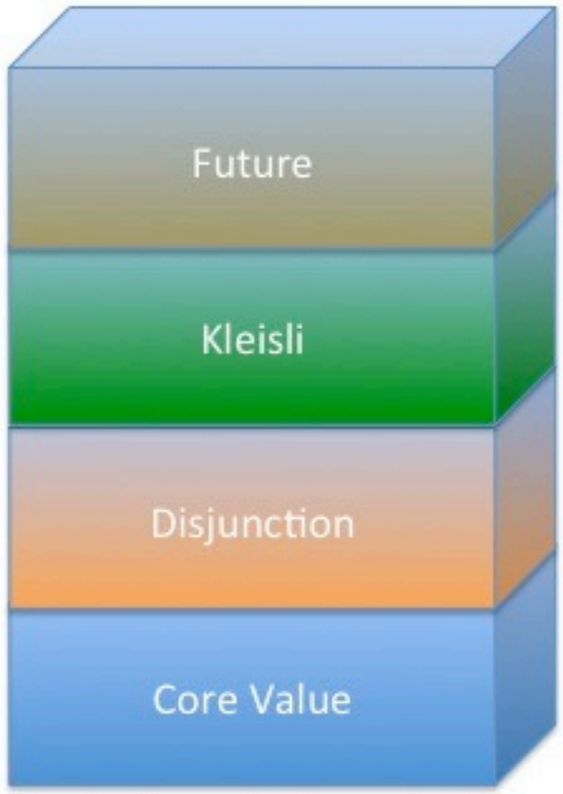
Goals towards Reactive API

- In our use case we would like to *augment our domain algebra* with future based APIs

- Just like an `Either` or a `Kleisli`, we would like to have *asynchrony as yet another stackable effect* within our computation



Stacking of Effects



Stacking of Effects

Monad Transformers

```
type Response[A] = String \/ Option[A]

val count: Response[Int] = some(10).right
for {
  maybeCount <- count
} yield {
  for {
    c <- maybeCount
    // use c
  } yield c
}
```

Monad Transformers

```
type Response[A] = String \/ Option[A]

val count: Response[Int] = some(10).right
for {
  maybeCount <- count
} yield {
  for {
    c <- maybeCount
    // use c
  } yield c
}
```

```
type Error[A] = String \/ A
type Response[A] = OptionT[Error, A]

val count: Response[Int] = 10.point[Response]
for {
  c <- count
  // use c : c is an Int here
} yield (())
```

Monad Transformers

```
type Response[A] = String \/ Option[A]

val count: Response[Int] = some(10).right
for {
  maybeCount <- count
} yield {
  for {
    c <- maybeCount
    // use c
  } yield c
}
```

richer algebra

```
type Error[A] = String \/ A
type Response[A] = OptionT[Error, A]

val count: Response[Int] = 10.point[Response]
for{
  c <- count
  // use c : c is an Int here
} yield (())
```

Monad Transformers

- collapses the stack and gives us a single monad to deal with
- order of stacking is important though

```scala
trait AccountService[Account, Amount, Balance] {

  type Valid[A] = EitherT[Future, NonEmptyList[String], A]

  type AccountOp[A] = Kleisli[Valid, AccountRepository, A]

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountOp[Account]

  //..
}
```

```scala
trait AccountService[Account, Amount, Balance] {

  type Valid[A] = EitherT[Future, NonEmptyList[String], A]

  type AccountOp[A] = Kleisli[Valid, AccountRepository, A]

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountOp[Account]

  //..
}
```

```
type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]
```

```
type Valid[A] = NonEmptyList[String] \/ A
```

The only change we need in the algebra is the plugging in of the new effect – the Future monad on top of the disjunction

Single monad – the disjunction

```
type Valid[A] = EitherT[Future, NonEmptyList[String], A]
```

Stack it up! Future monad on top of disjunction

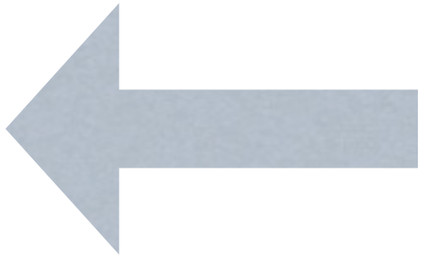
```scala
trait AccountService[Account, Amount, Balance] {

  type Valid[A] = EitherT[Future, NonEmptyList[String], A]

  type AccountOp[A] = Kleisli[Valid, AccountRepository, A]

  def open(

    no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType

  ): AccountOp[Account]

  //..
}
```

Reactive ..

Algebraically

```
class AccountServiceInterpreter
  extends AccountService[Account, Amount, Balance] {

  def open(no: String,
           name: String,
           rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType) =

    kleisli[Valid, AccountRepository, Account] { (repo:
AccountRepository) =>

        EitherT {
          Future {
            repo.query(no) match {
              //..
            }
          }
        }
    }
  //..
}
```

```
class AccountServiceInterpreter
  extends AccountService[Account, Amount, Balance] {

  def open(no: String,
           name: String,
           rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType) =

    kleisli[Valid, AccountRepository, Account] { (repo:
AccountRepository) =>

        EitherT {
          Future {
            repo.query(no) match {
              //..
            }
          }
        }
      }
    //..
  }
```



normal logic

We introduced a whole *new effect of asynchrony* to implement *reactive traits* in our domain model API *algebra & implementatio*n just by composing with another type without any change in the core domain logic. This is the essence of *typed functional programming*. We have types that model effects functionally and we can just *stack them up* in the proper order that we need.

Advantages

- We are still in the statically typed land even with asynchronous behaviors baked into our APIs
- We can reason about our program statically
- We can compose asynchronous components to form larger abstractions

Reactive & algebraic patterns in domain modeling

```
for {
  _ <- open(..)
  _ <- credit(..)
  d <- debit(..)

} yield d
```

Reactive & algebraic patterns in domain modeling

- **Compositional by types**

```
for {
  _ <- open(..)
  _ <- credit(..)
  d <- debit(..)

} yield d
```

Reactive & algebraic patterns in domain modeling

```
for {
  _ <- open(..)
  _ <- credit(..)
  d <- debit(..)

} yield d
```

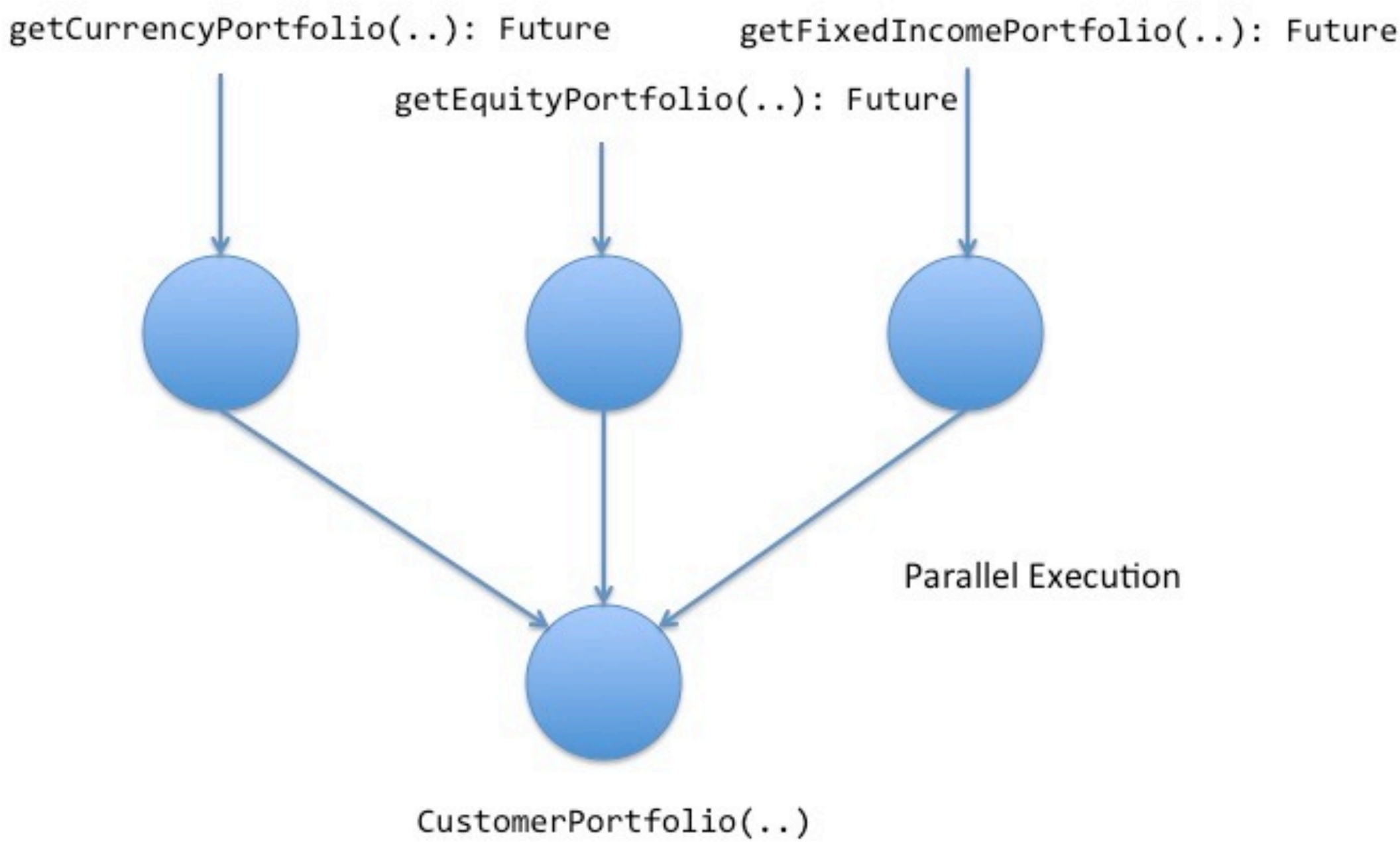
- **Compositional by types**
- **Individual operations sequential as they thread through the comprehension**

Reactive & algebraic patterns in domain modeling

```
for {
  _ <- open(..)
  _ <- credit(..)
  d <- debit(..)

} yield d
```

- **Compositional by types**
- **Individual operations sequential as they thread through the comprehension**
- **Composed operation doesn't block the main thread of execution**

Reactive & algebraic patterns in domain modeling



Reactive & algebraic patterns in domain modeling

```
trait PortfolioService {
  type PFOperation[A] = Kleisli[Future, AccountRepository, Seq[A]]

  def getCurrencyPortfolio(no: String, asOf: Date)
    : PFOperation[Balance]

  def getEquityPortfolio(no: String, asOf: Date)
    : PFOperation[Balance]

  def getFixedIncomePortfolio(no: String, asOf: Date)
    : PFOperation[Balance]
}
```

Reactive & algebraic patterns in domain modeling

```
val ccyPF: Future[Seq[Balance]] =
  getCurrencyPortfolio(accountNo, asOf)(AccountRepository)

val eqtPF: Future[Seq[Balance]] =
  getEquityPortfolio(accountNo, asOf)(AccountRepository)

val fixPF: Future[Seq[Balance]] =
  getFixedIncomePortfolio(accountNo, asOf)(AccountRepository)

val portfolio: Future[Portfolio] = for {
  c <- ccyPF
  e <- eqtPF
  f <- fixPF
} yield CustomerPortfolio(accountNo, asOf, c ++ e ++ f)
```

Be Algebraic, as long as you can ..

Beyond Algebra - Reactive Protocols

Conference Reservations

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra A

Reactive

Protocols

Badge Printing

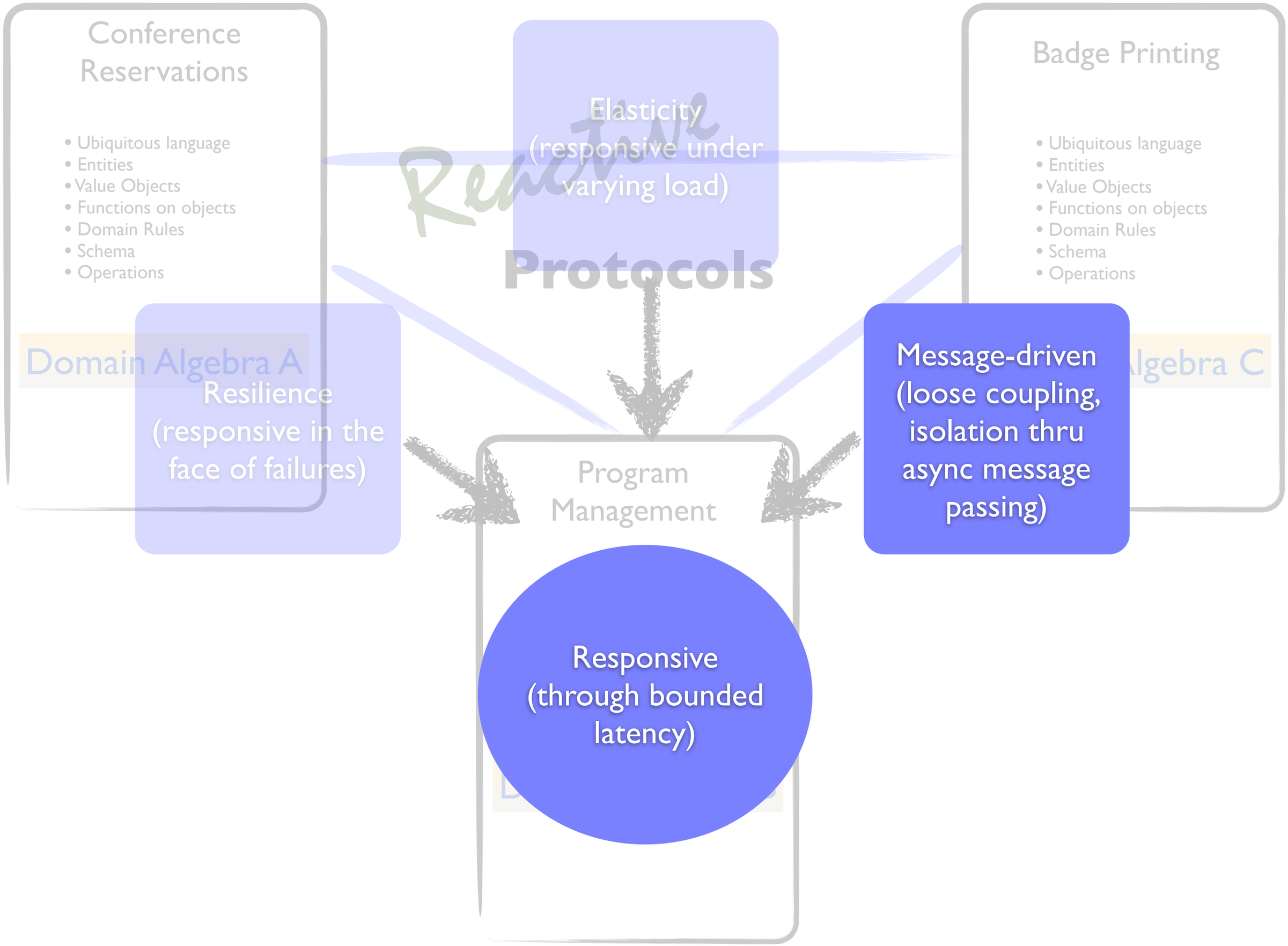
- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

Domain Algebra C

Program Management

- Ubiquitous language
- Entities
- Value Objects
- Functions on objects
- Domain Rules
- Schema
- Operations

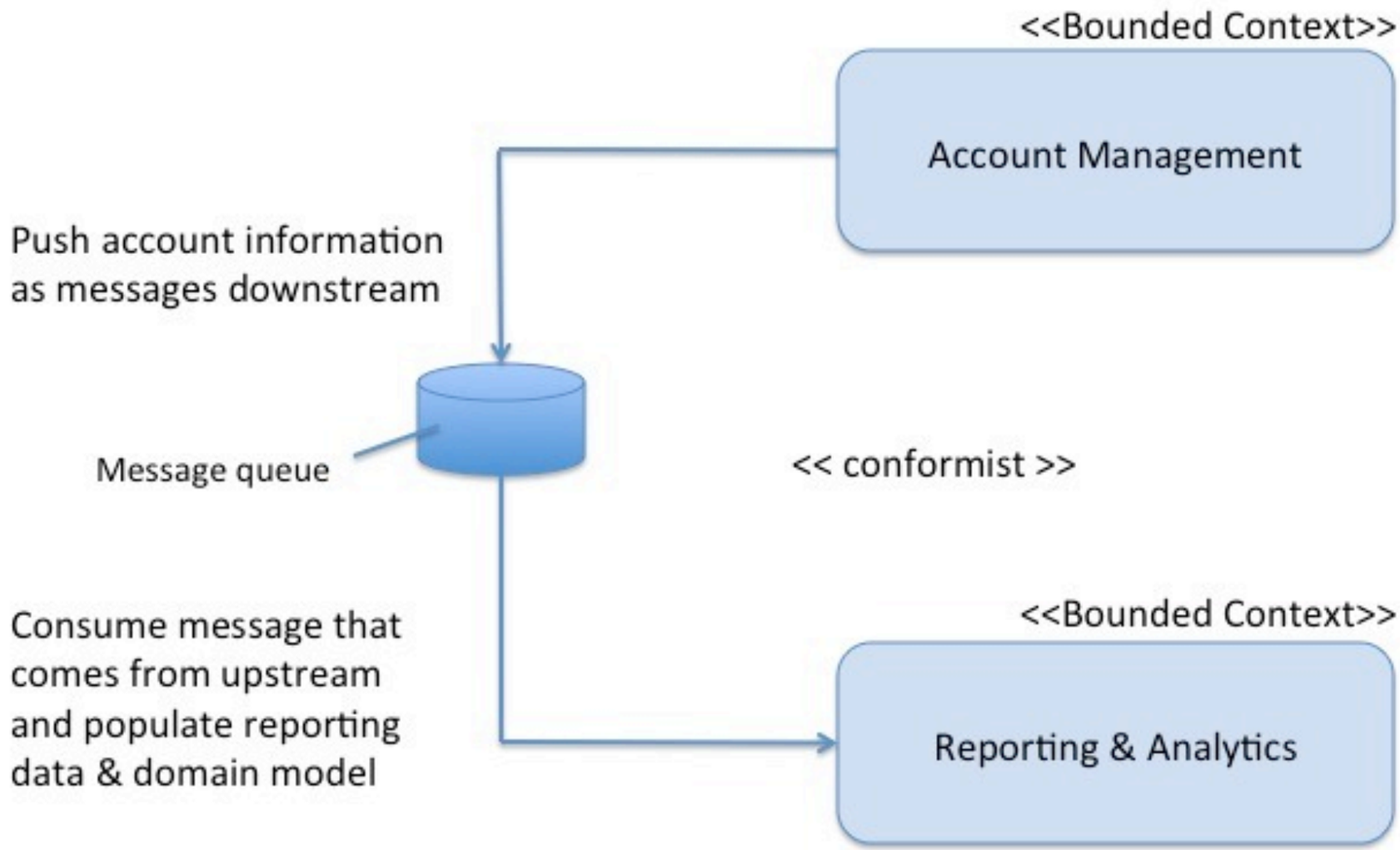
Domain Algebra B


Conference Reservations
• Ubiquitous language
• Entities
• Value Objects
• Functions on objects
• Domain Rules
• Schema
• Operations
Elasticity (responsive under varying load)
Reactive
Protocols
Badge Printing
• Ubiquitous language
• Entities
• Value Objects
• Functions on objects
• Domain Rules
• Schema
• Operations
Domain Algebra A
Resilience (responsive in the face of failures)
Program Management
Message-driven (loose coupling, isolation thru async message passing)
Responsive (through bounded latency)

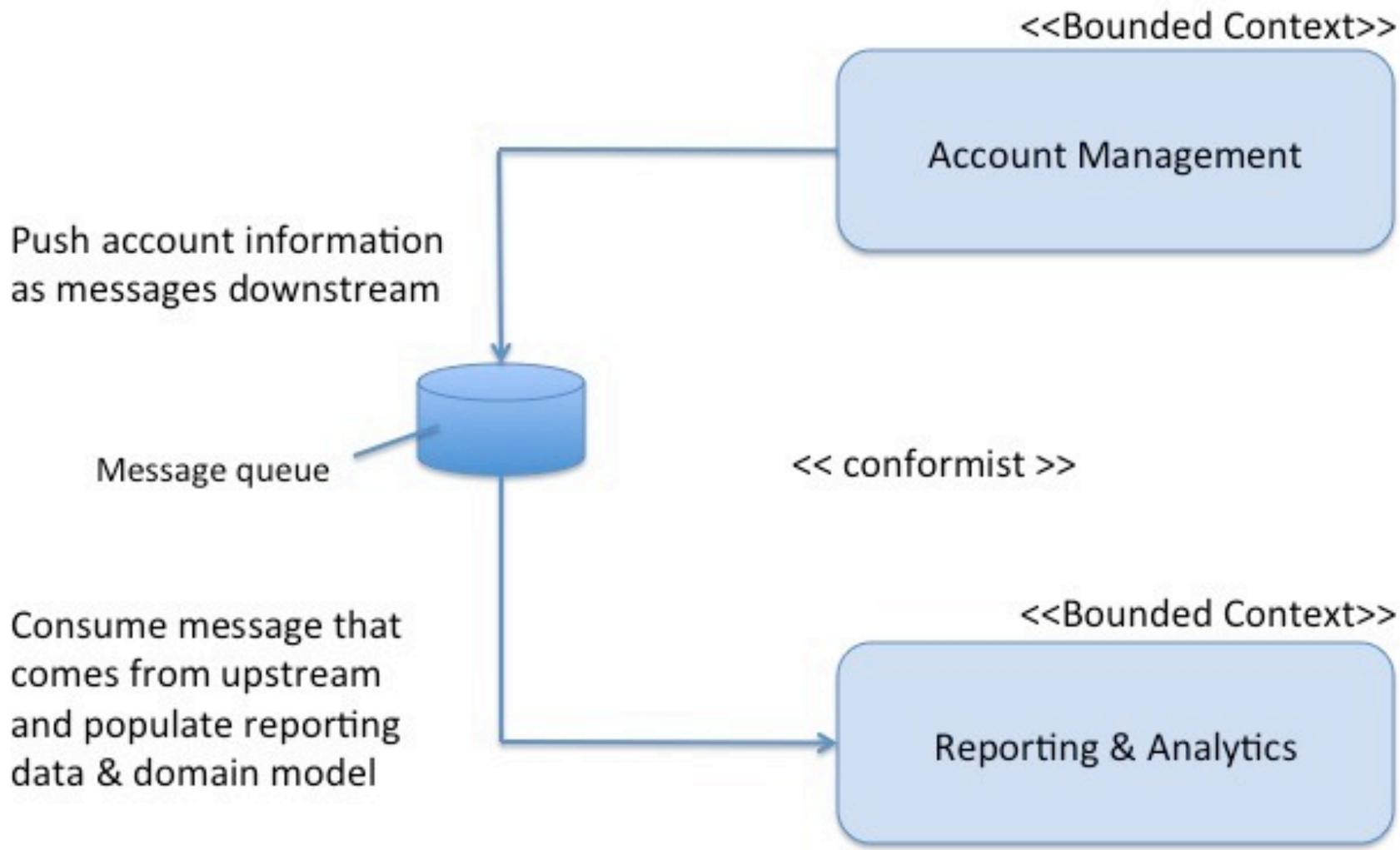
Asynchronous Messaging

Asynchronous Messaging

Asynchronous Messaging



Asynchronous Messaging



Actors and Domain Models

Actors and Domain Models

Powerful

Actors and Domain Models

Powerful

Un-algebraically **Powerful**

Actors and Domain Models

Powerful

Un-algebraically **Powerful**

Gain power at one semantic level but
lose the power of reasoning

Using actors *indiscriminately* throughout your domain model makes algebraic reasoning hard

```
fork:
  A => Future[A]

map:
  (A => B) => (Future[A] => Future[B])

join:
  Future[Future[A]] => Future[A]
```

```
receive: Any => Unit
```

Use the *least* powerful abstraction that does the job

For domain model resilience, choose *futures over actors* when you can ..

Actors and Domain Models

- As an implementation artifact to protect shared mutable state
- Centralized failure management

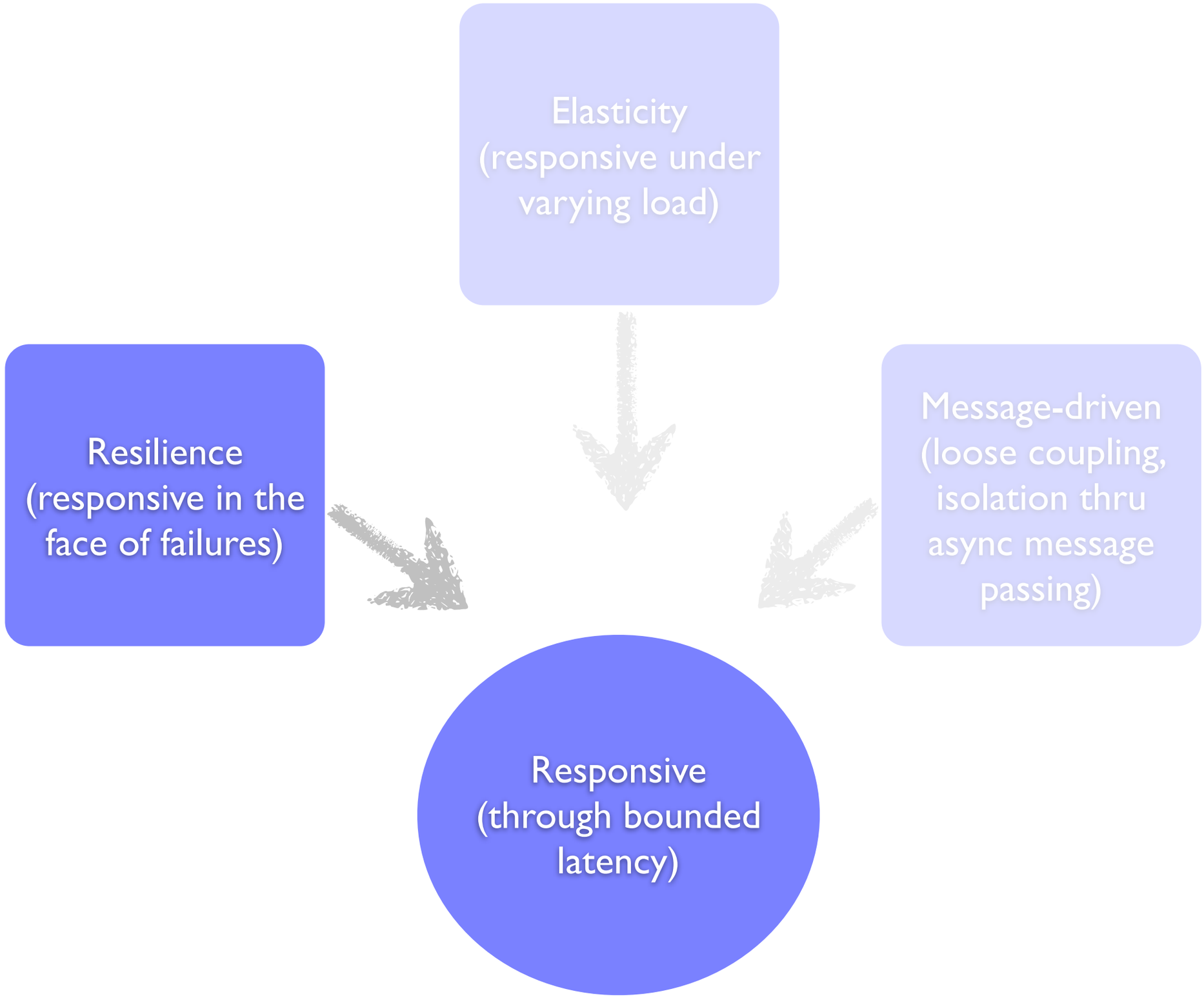
```scala
import scala.collection.mutable.{ Map => MMap }

class Summarizer extends Actor with ActorSubscriber with Logging {

  private val balance = MMap.empty[String, Balance]

  def receive = {
    case OnNext(data: Transaction) => updateBalance(data)

    case LogSummaryBalance => logger.info("Balance: " + balance)
  }

  def updateBalance(data: Transaction) =
    balance.get(data.accountNo).fold {
      balance += ..
    } { b =>
      balance += ..
    }
}
```

shared mutable state here

updated

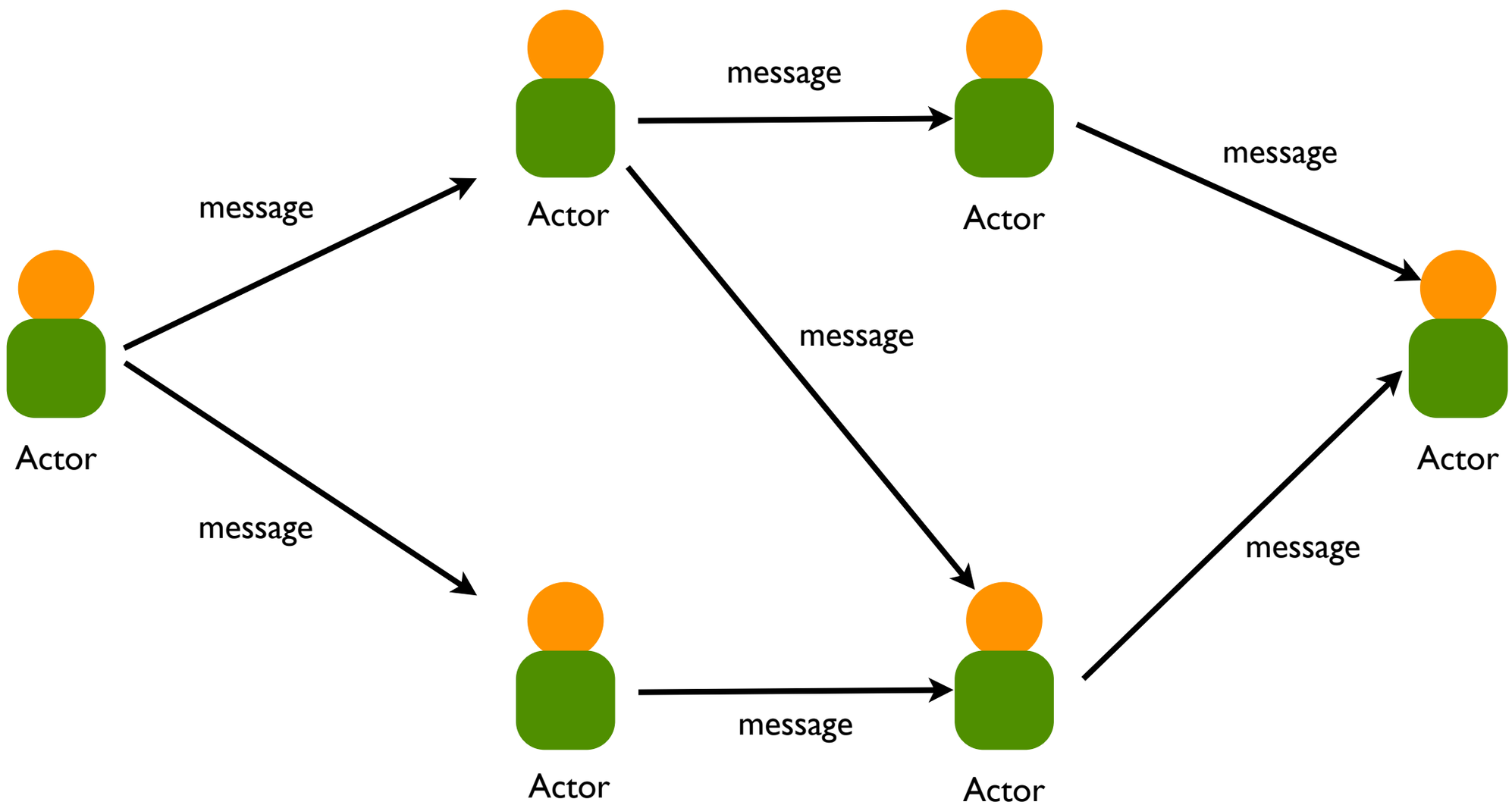
Centralized Failure Management

- Supervisor hierarchies that manages failures
- Kill, restart, suspend / resume
- No more messy failure handling code scattered throughout
- Requires careful upfront design though



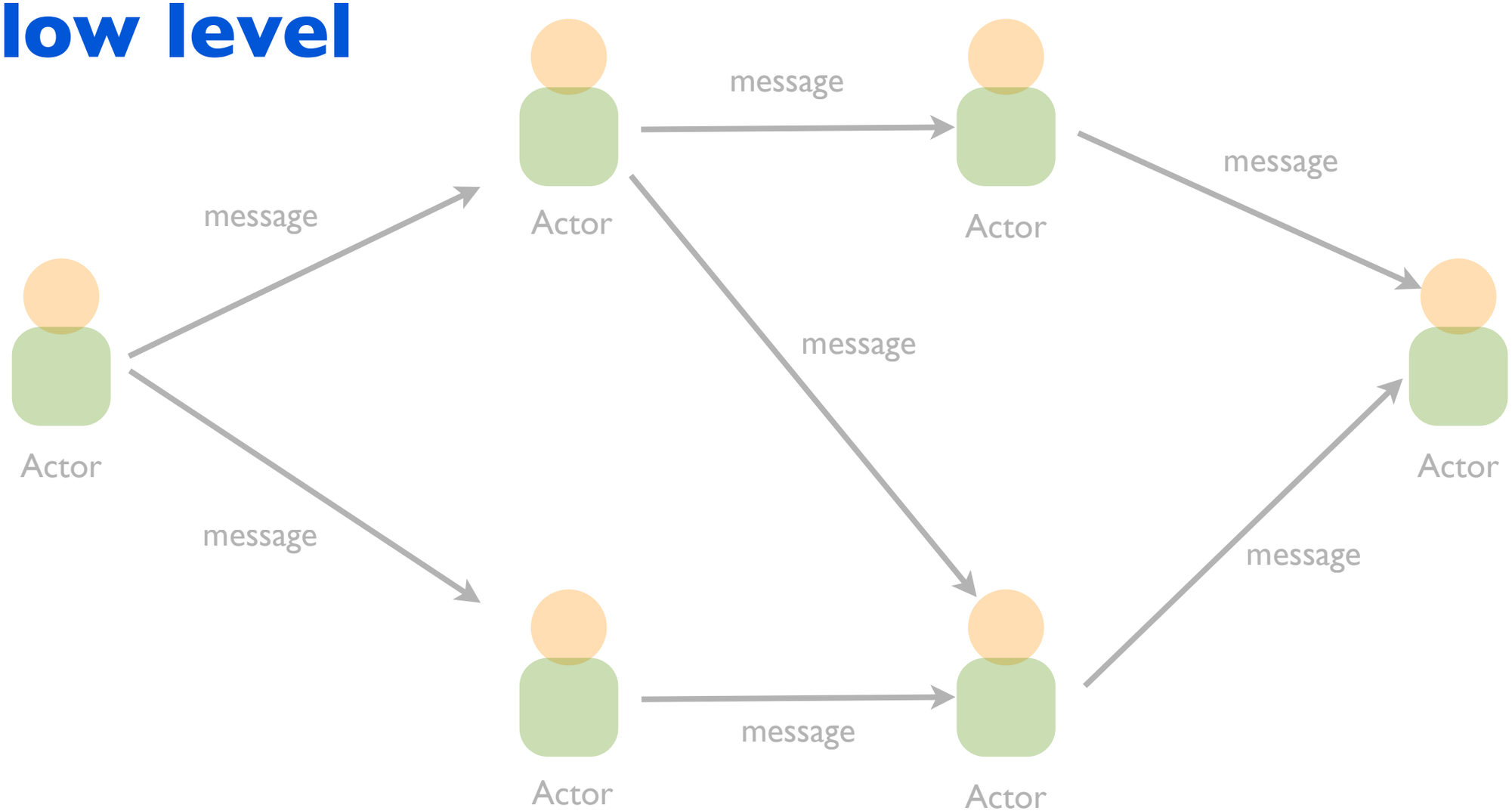
Being Reactive

Modeling Domain Workflows

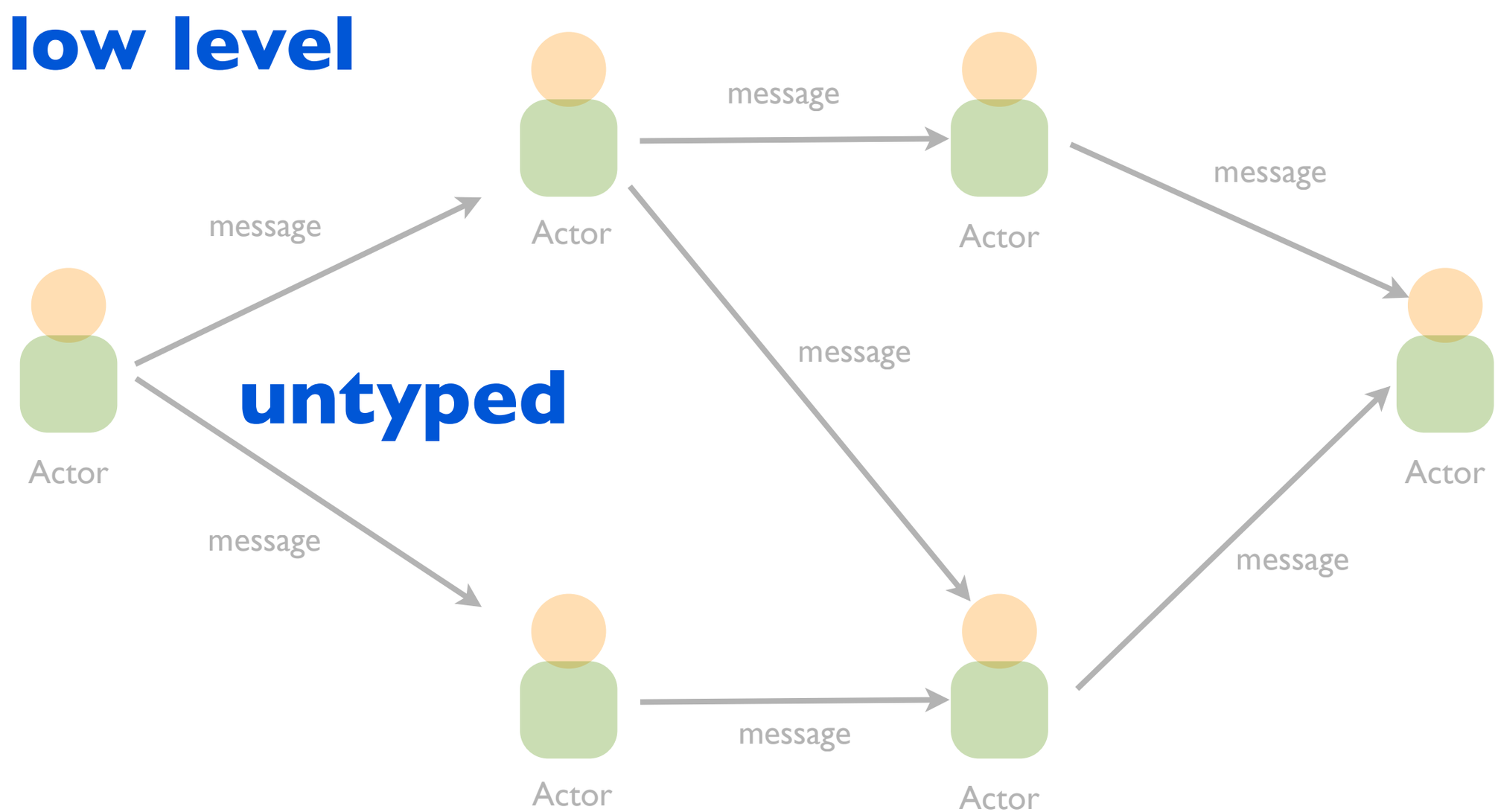


Modeling Domain Workflows

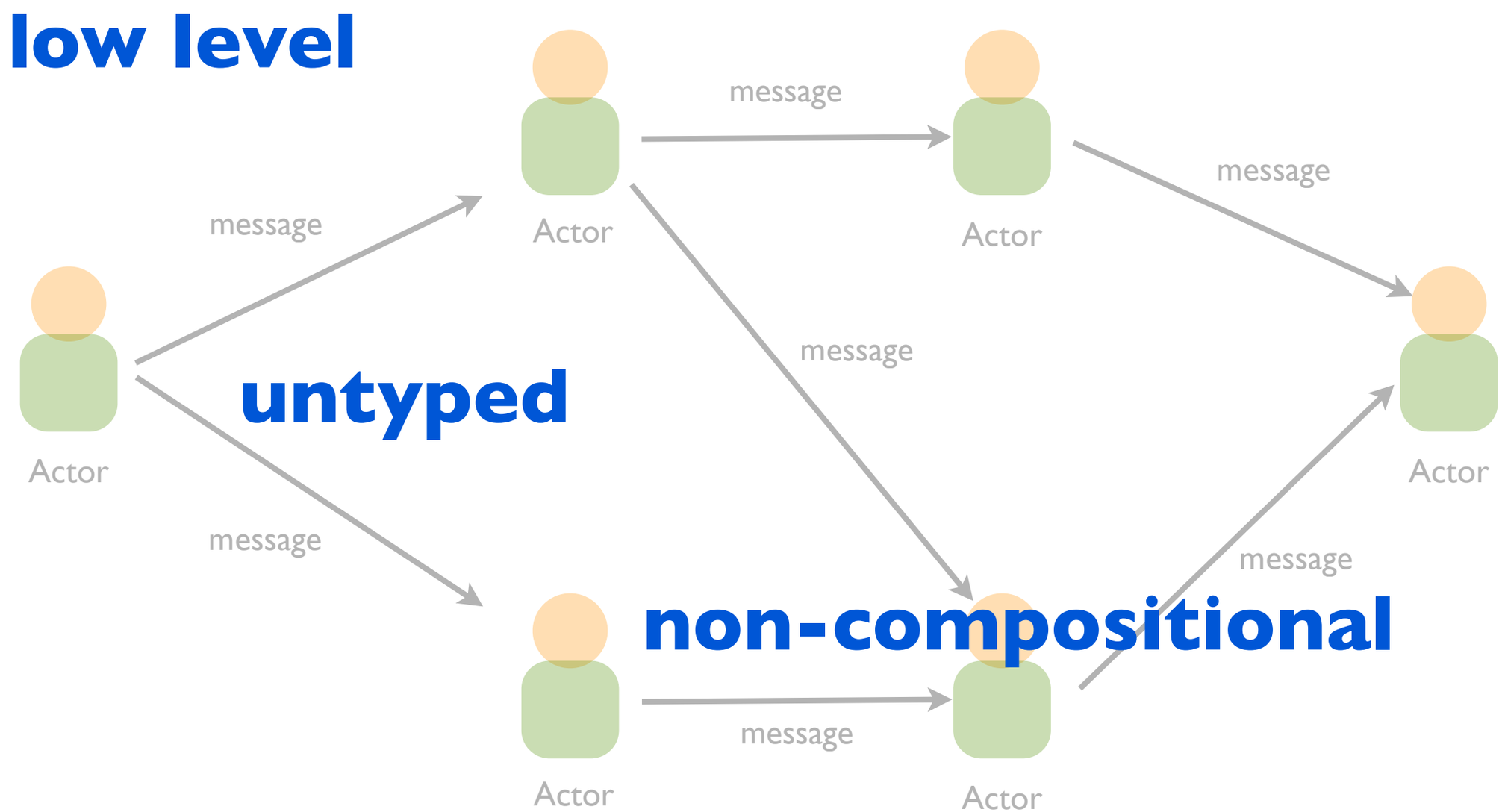
low level



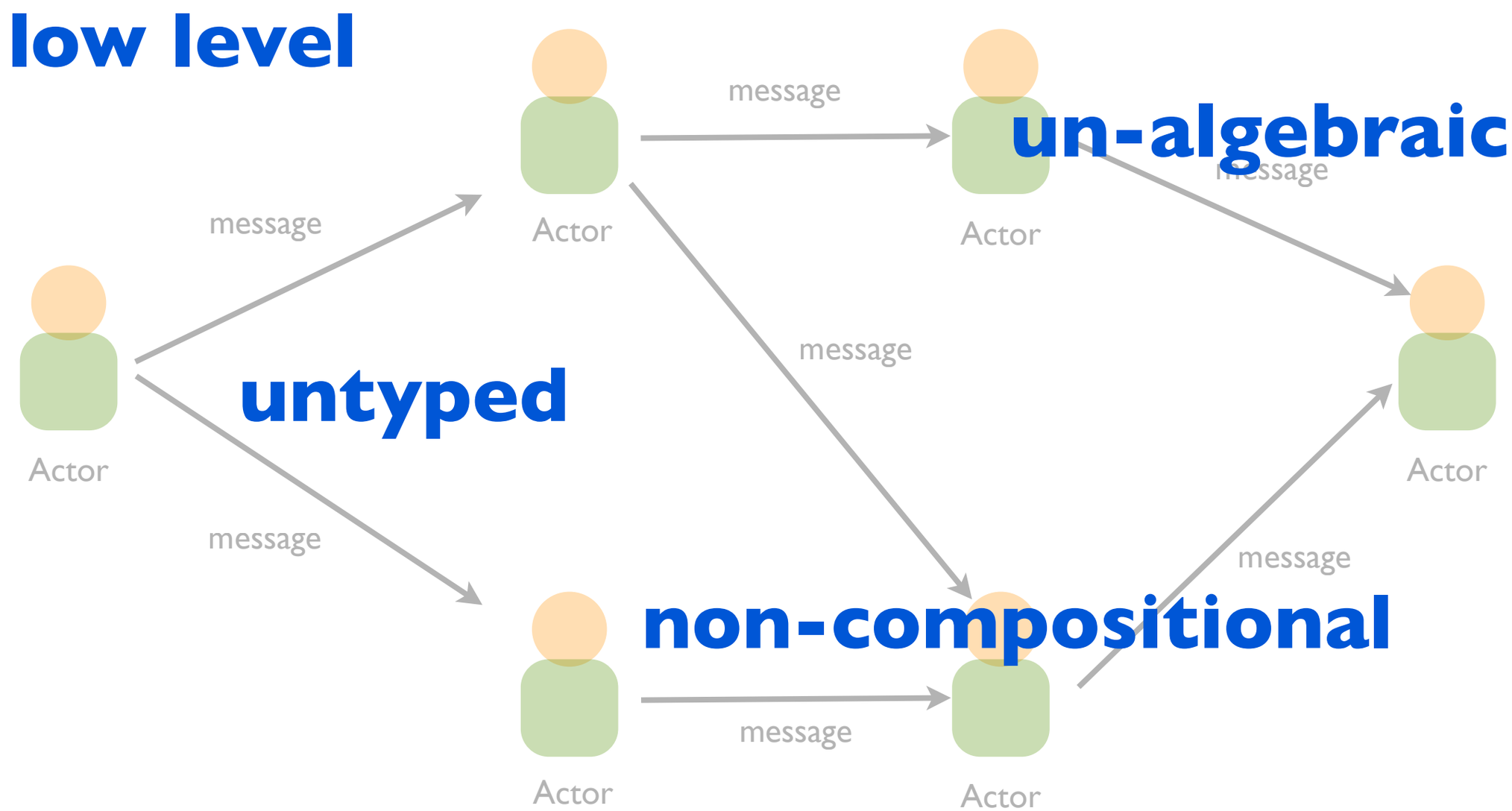
Modeling Domain Workflows



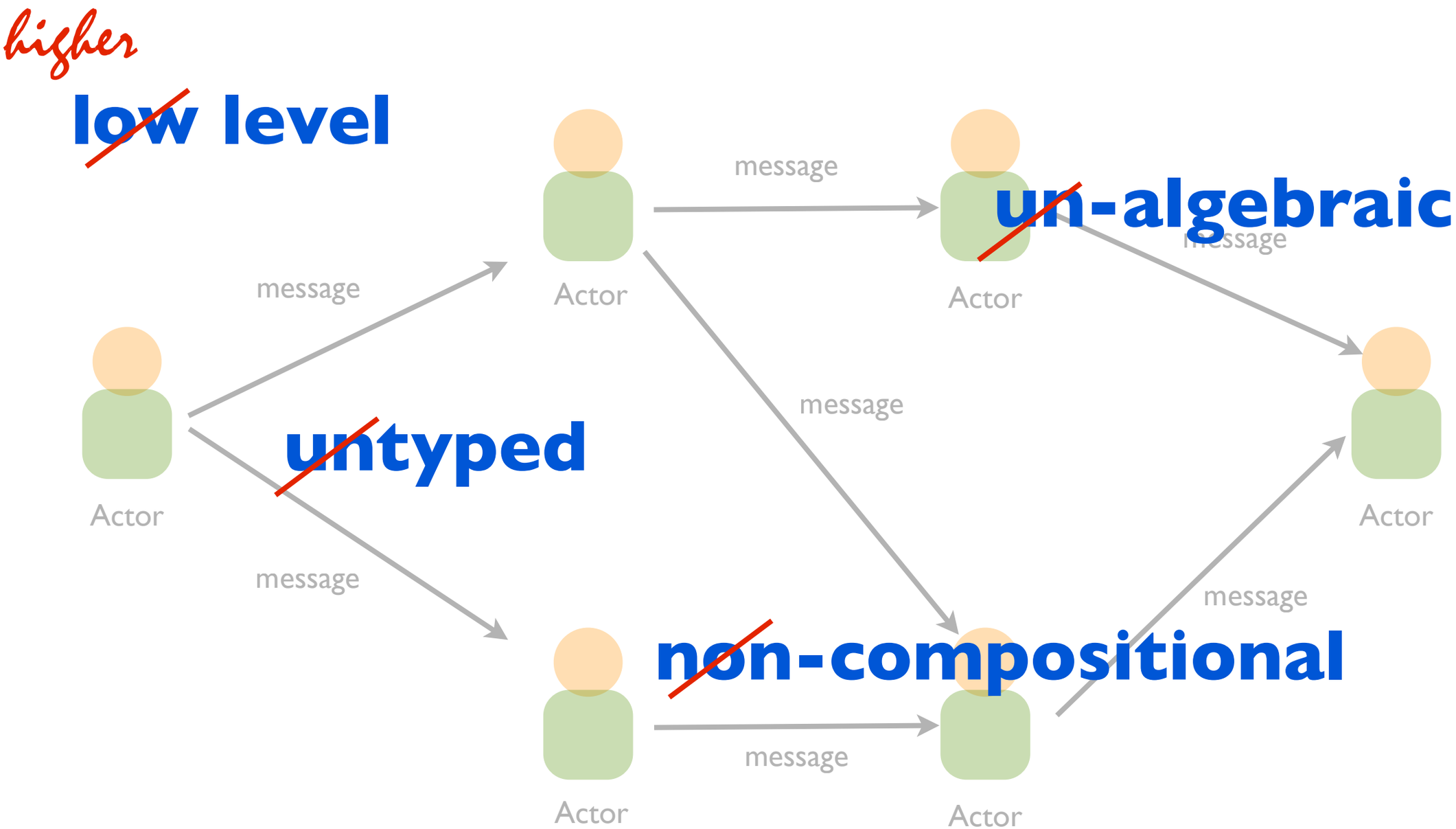
Modeling Domain Workflows



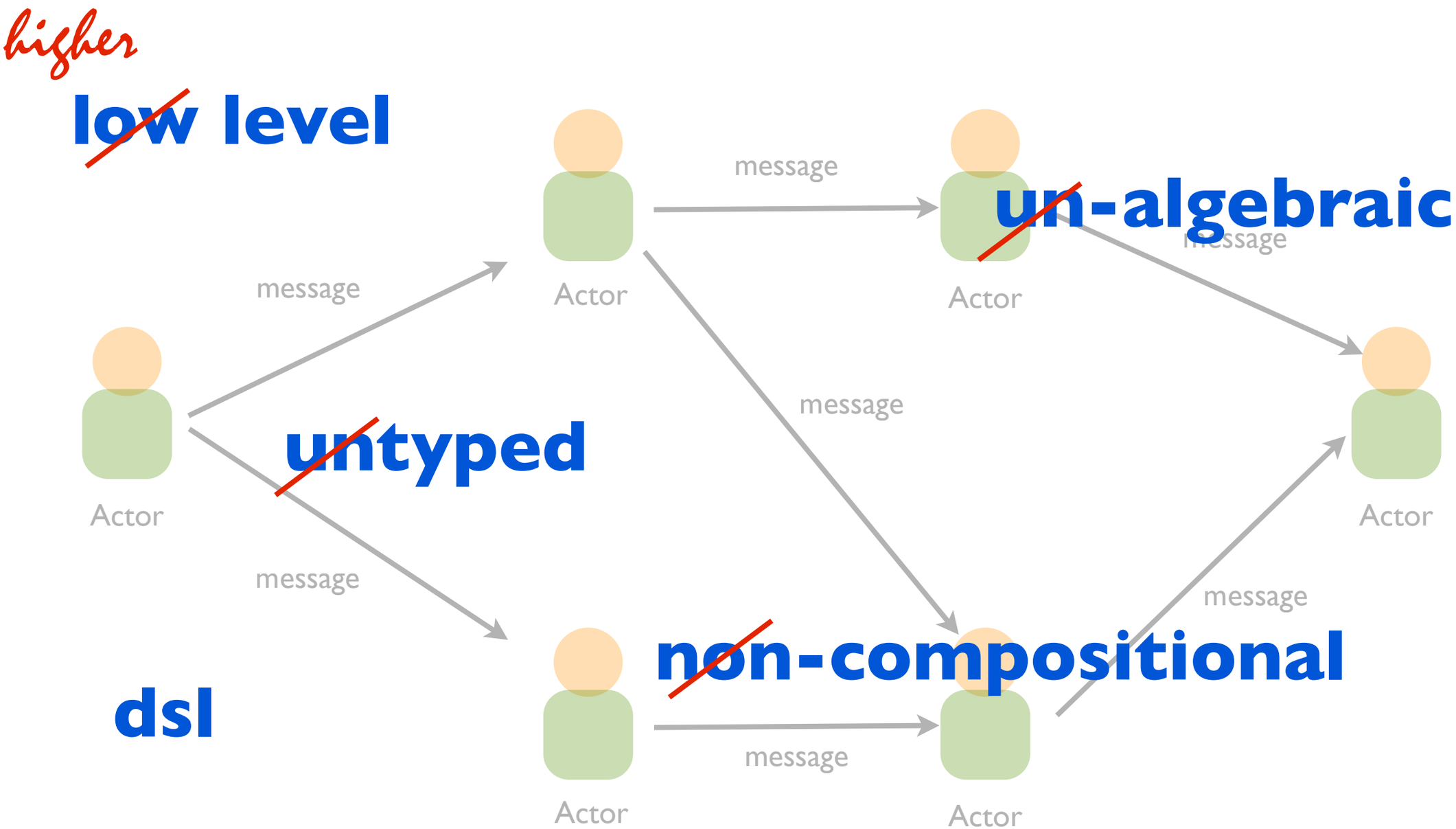
Modeling Domain Workflows



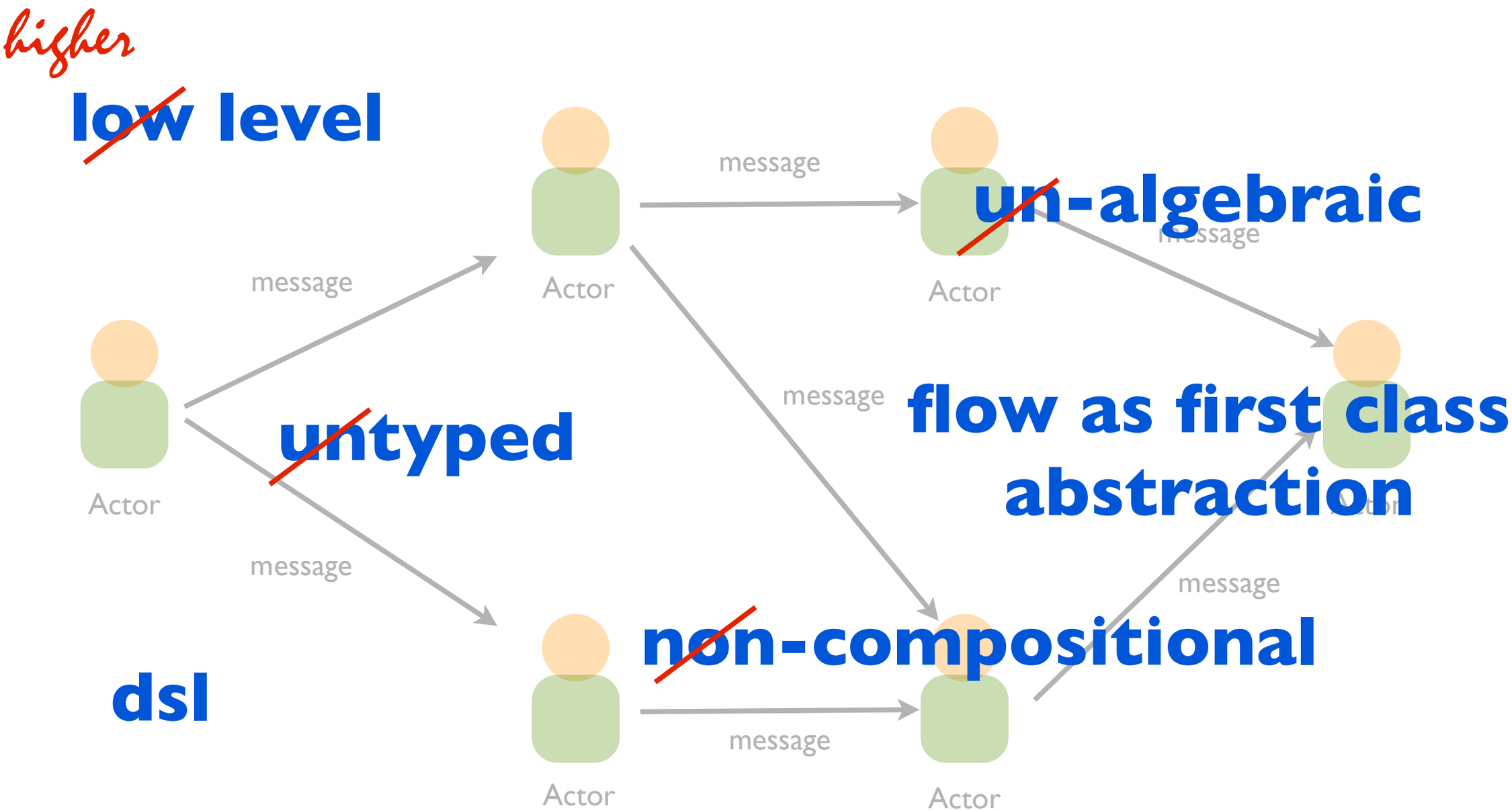
Modeling Domain Workflows



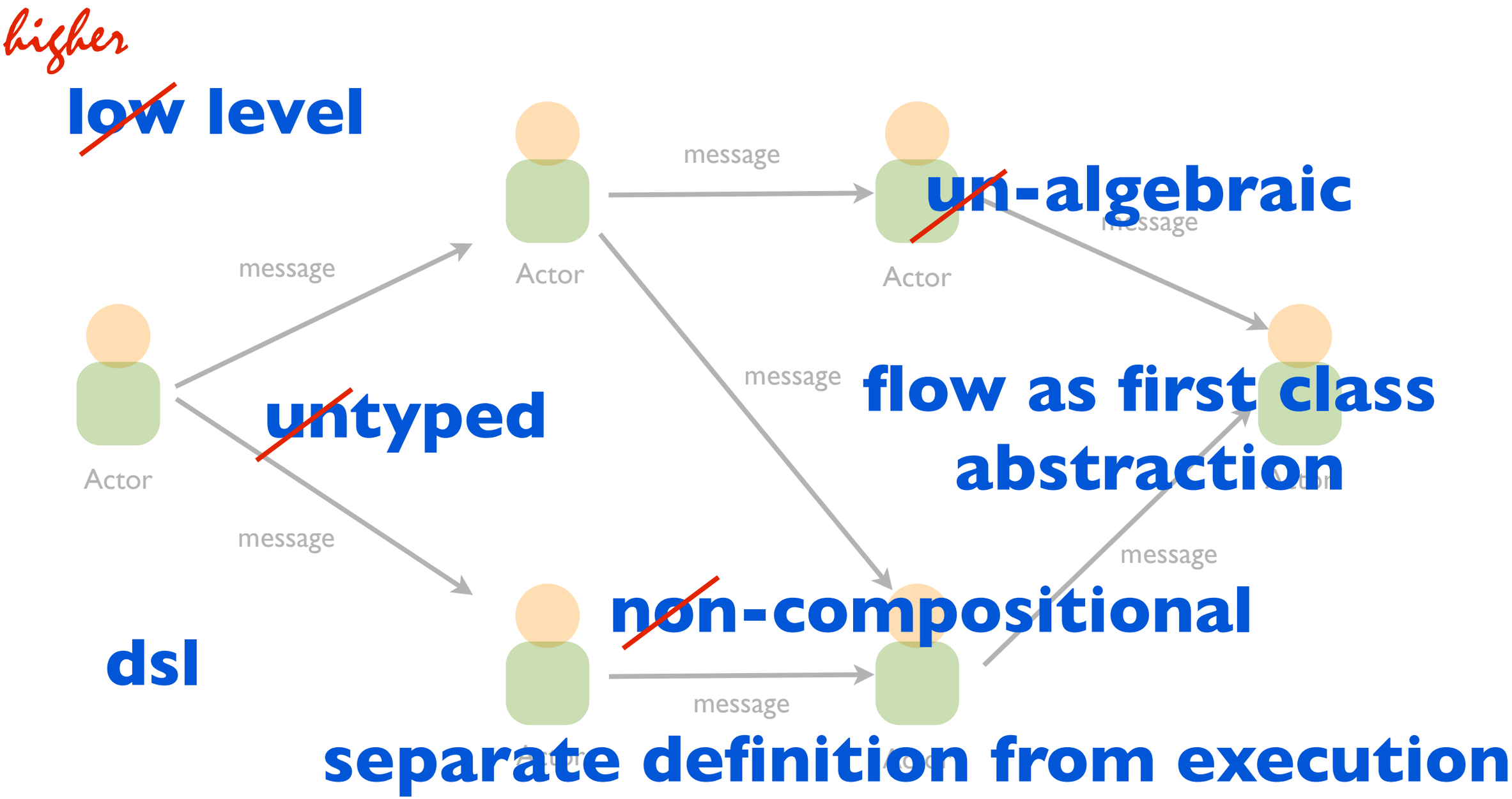
Modeling Domain Workflows



Modeling Domain Workflows



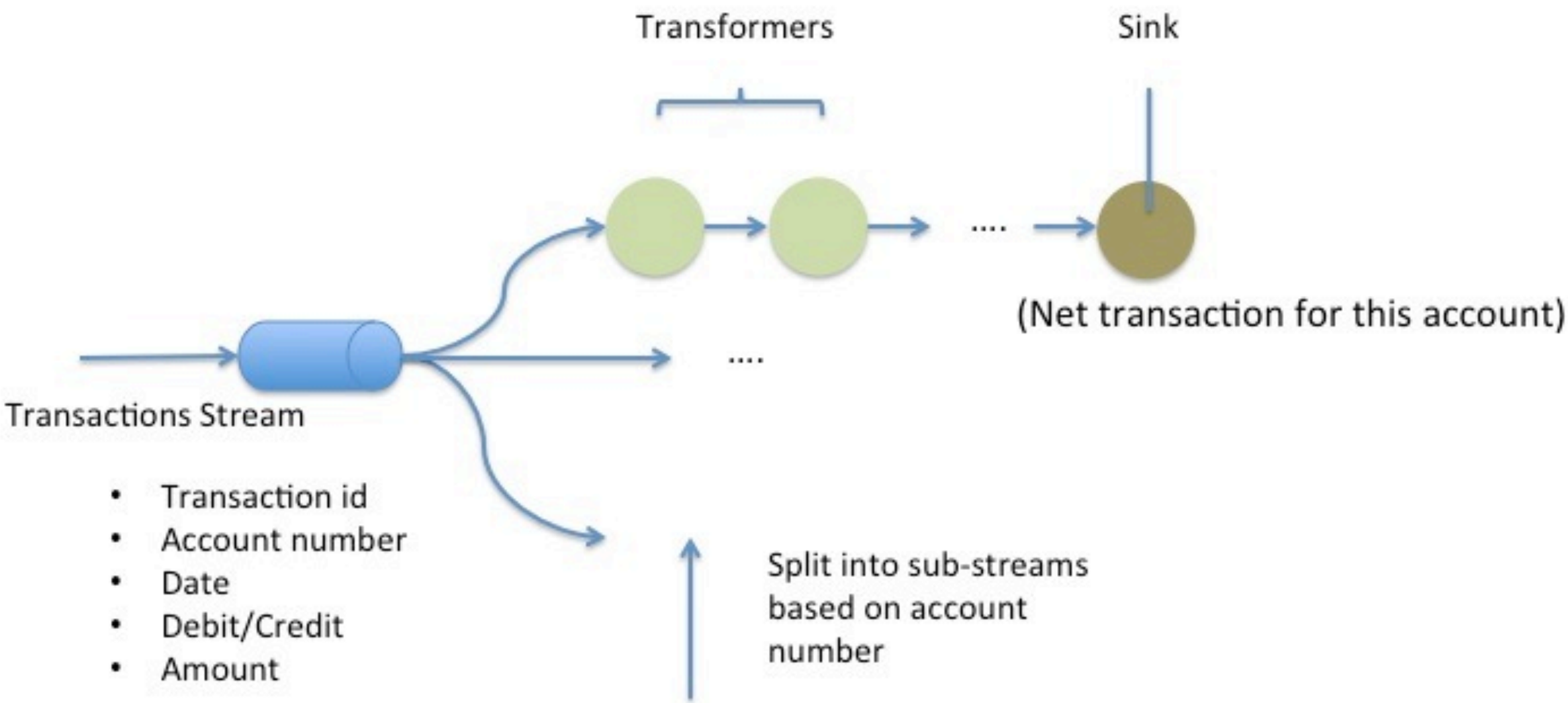
Modeling Domain Workflows



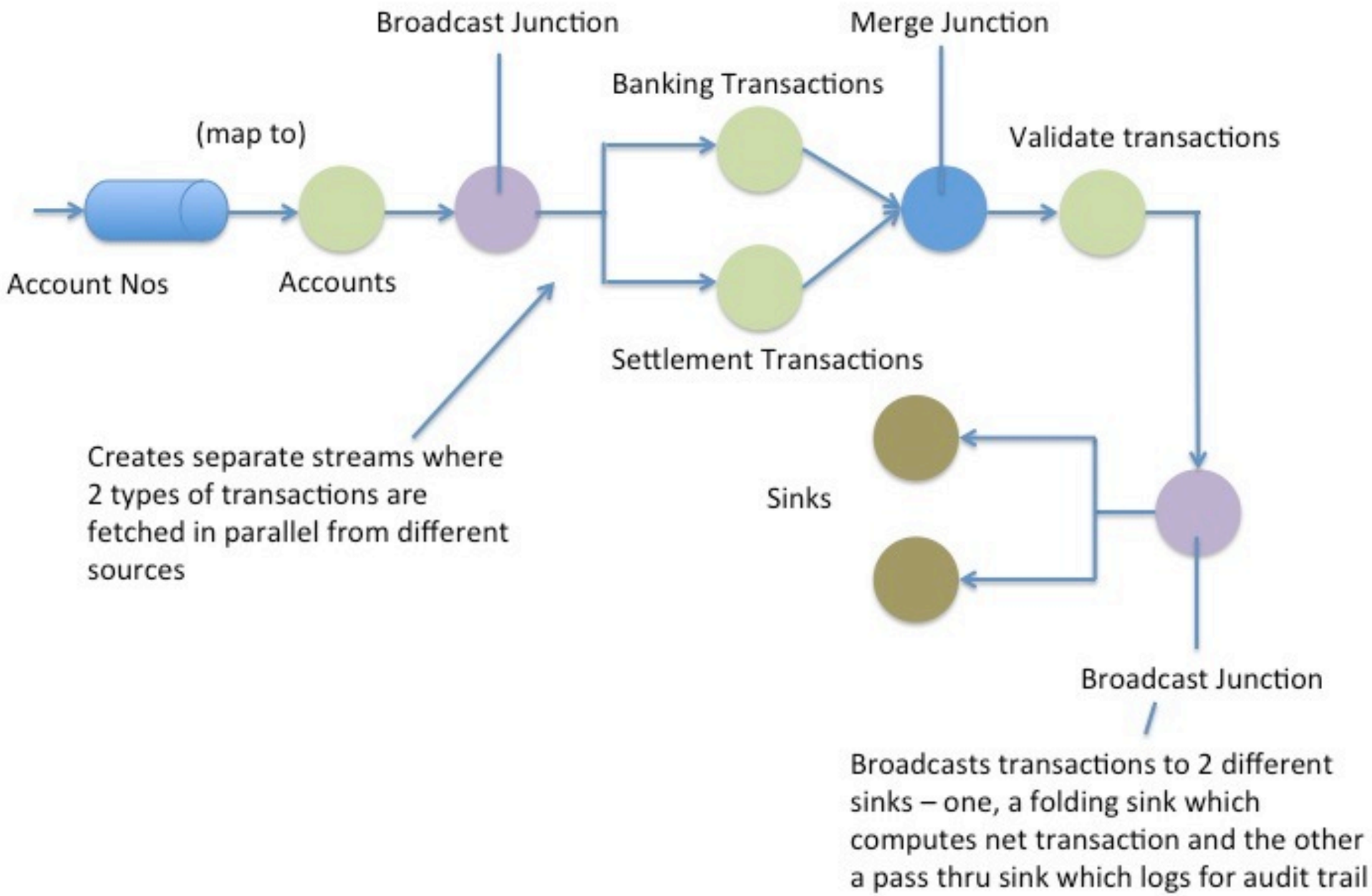
Modeling Domain Workflows

Akka Streams

Source	Pipeline starts here. Source[+Out, +Mat] takes data from input & has a single output
Sink	Pipeline ends here. Sink[+In, +Mat] has a single input to be written into
Flow	Basic transformation abstraction. Flow[-In, +Out, +Mat] has 1 input & 1 output. Mat is the actor materializer
Runnable Graph	The entire topology ready to run



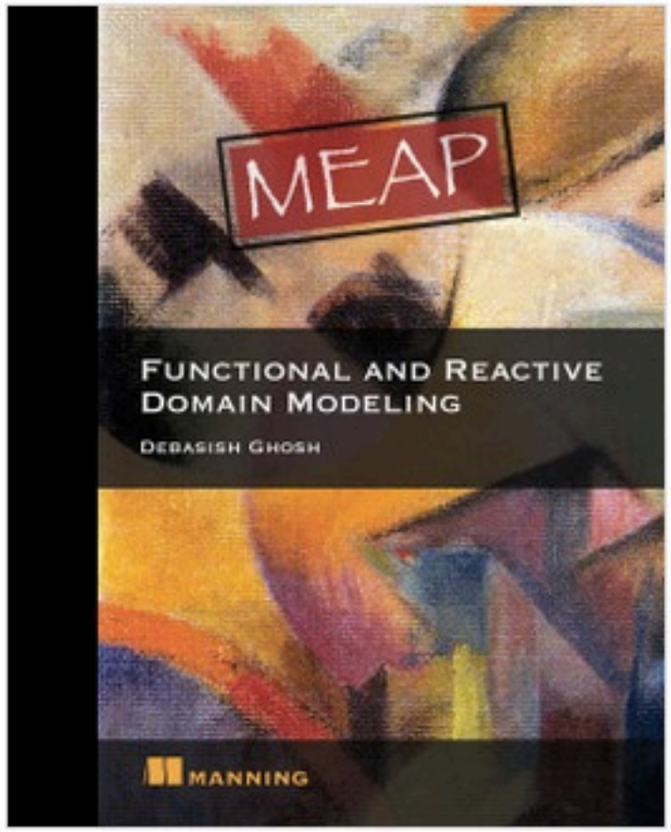
Business Use Case - The Domain Model



Implementation topology with Akka Streams

```scala
val graph = FlowGraph.closed(netTxnSink) { implicit b => ns =>
  import FlowGraph.Implicits._

  val accountBroadcast = b.add(Broadcast[Account](2))
  val txnBroadcast = b.add(Broadcast[Transaction](2))
  val merge = b.add(Merge[Transaction](2))

  val accounts = Flow[String].map(queryAccount(_, AccountRepository))

  val bankingTxns = Flow[Account].mapConcat(getBankingTransactions)

  val settlementTxns =
    Flow[Account].mapConcat(getSettlementTransactions)

  val validation = Flow[Transaction].map(validate)

  accountNos ~> accounts ~> accountBroadcast ~> bankingTxns
             ~> merge ~> validation ~> txnBroadcast ~> ns
  accountBroadcast ~> settlementTxns ~> merge
  txnBroadcast ~> audit
}
```

```
graph.run()
```



FREE DOWNLOADS

Chapter 1

Source Code

Functional and Reactive Domain Modeling

Debasish Ghosh
MEAP began October 2014 · Publication in Spring 2016 *(estimated)*
ISBN 9781617292248 · 325 pages *(estimated)* · printed in black & white

Functional and Reactive Domain Modeling teaches you how to think of the domain model in terms of pure functions and how to compose them to build larger abstractions. You will start with the basics of functional programming and gradually progress to the advanced concepts and patterns that you need to know to implemen complex domain models. The book demonstrates how advanced FP patterns like algebraic data types, typeclass based design, and isolation of side-effects can make your model compose for readability and verifiability.

On the subject of reactive modeling, the book focuses on higher order concurrency patterns like actors and futures. It uses the Akka framework as the reference implementation and demonstrates how advanced architectural patterns like event sourcing and CQRS can be put to great use in implementing scalable models. You wil learn techniques that are radically different from the standard RDBMS based applications that are based on mutation of records. You'll also pick up important patterns like using asynchronous messaging for interaction based on non blocking concurrency and model persistence, which delivers the speed of in-memory processing along with suitable guarantees of reliability.

https://www.manning.com/books/functional-and-reactive-domain-modeling

Thank You!

goto; conference

Please

Remember to rate this session

Thank you!