集約の設計と実装

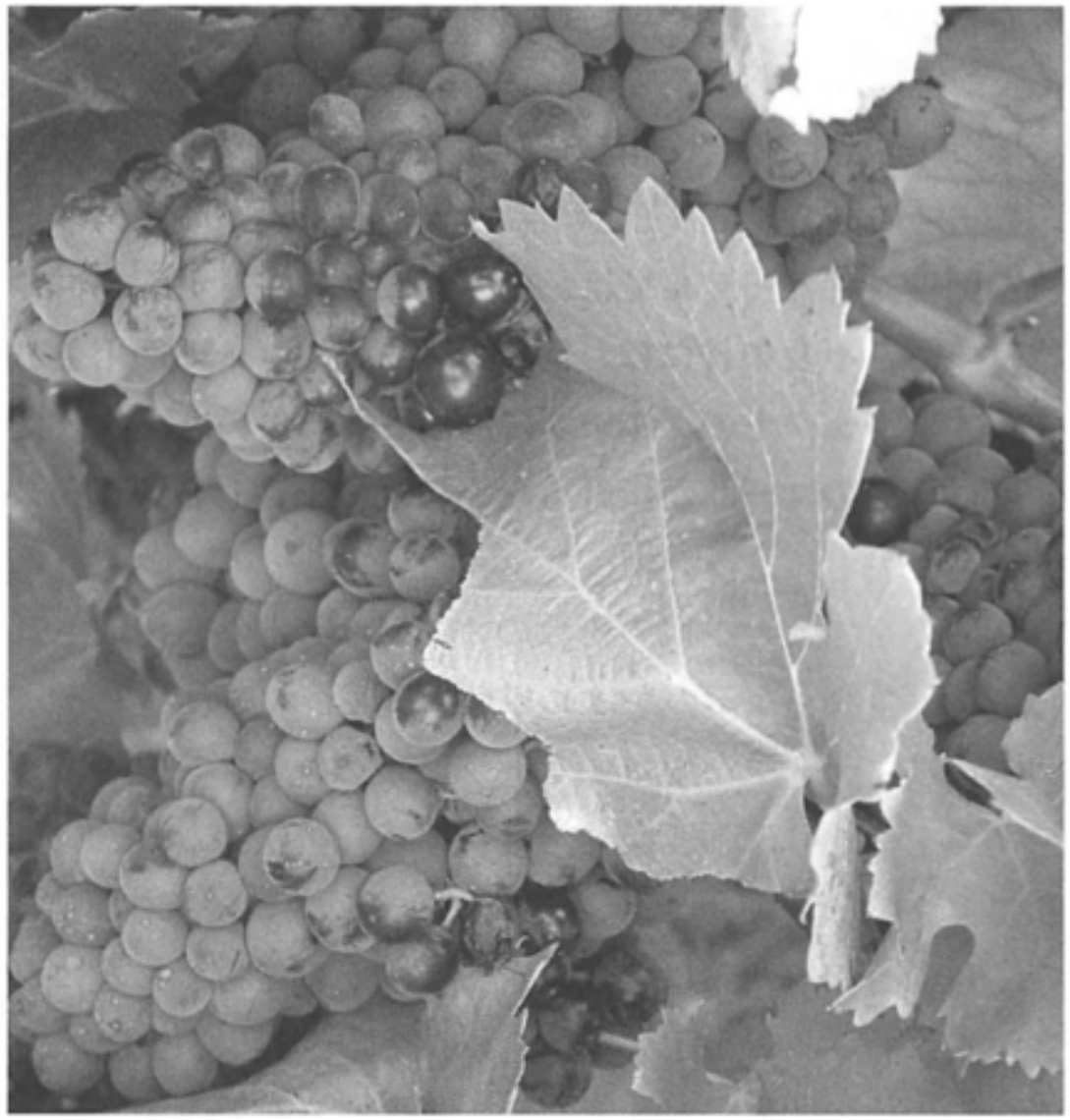
かとじゅん(@j5ik2o)

アジェンダ

- 目的
  - DDDの戦術的パターンの中で比較的に重要で難解である、**集約**をどのように設計・実装するとよいか、注意点なども踏まえながら解説
- アジェンダ
  - 集約とは
  - 集約の境界
  - 自動車修理店での事例
  - 契約による設計(DbC)
  - 集約を実装するためのルール(Evans and Vernon)
  - まとめ

集約とは

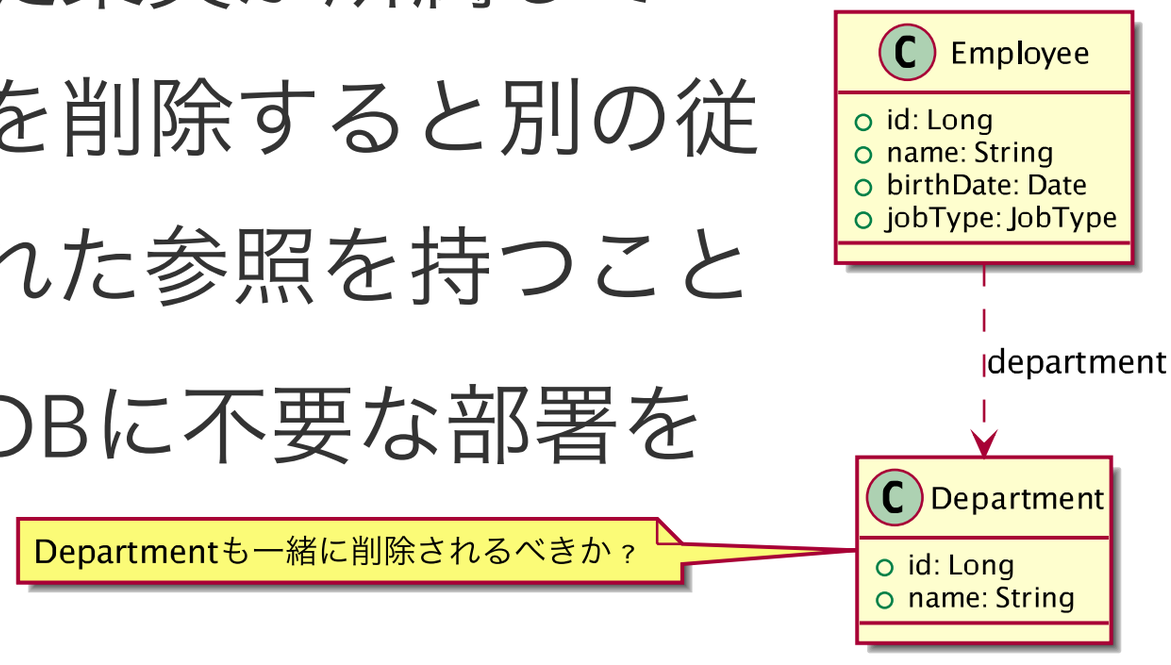
- ライフサイクルを共にする、関連が深い複数のオブジェクトを、**ひとかたまりの"集約"**として扱う
- 集約を**変更の一貫性を保証する一つの単位**として、他のオブジェクトの「境界」を明確に分ける
- オブジェクトは他のオブジェクトと関連を持つ可能性があるが、**不適切なロックはユーザビリティを低下させる。**同じ行を編集していないのに文書全体をロックなどはロック競合の温床になる



境界はどこ？

集約

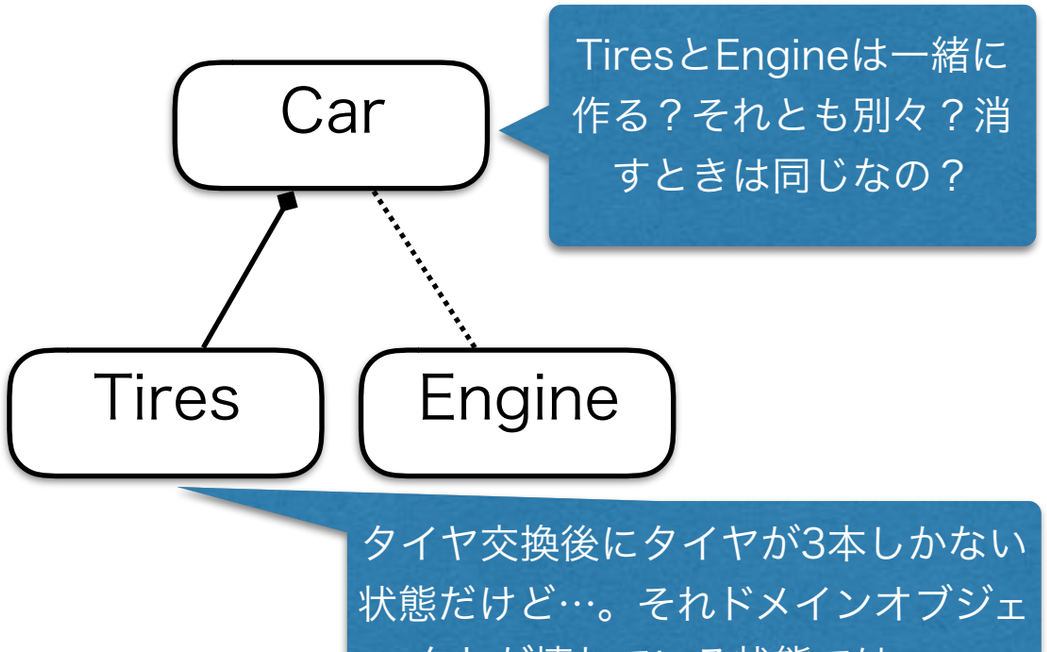
- DBから従業員オブジェクトを削除するケース
  - 従業員と共に名前・生年月日・職務区分はなくなるが、部署はどうだろうか？
  - 部署が従業員の境界に関係ないことは自明だが、その判断基準を論理的に説明可能か？
- たとえば、同じ部署に別の従業員が所属しているかもしれない。その部署を削除すると別の従業員オブジェクトは削除された参照を持つことになる。逆に放置すれば、DBに不要な部署をため込むことになる



境界未定義による問題

集約

- 存在する複数のオブジェクトが、いつ作られ、いつなくなるのか？
- データを変更する範囲(トランザクションスコープなど)やデータの一貫性を維持するルールがなければならない
- 境界未定義のまま、DBのロック機構を利用することができるが、その場しのぎの解決策と言わざるを得ない(扱いを間違えればロックが競合する可能性がある)
- 本質はモデルにある。これらの問題は境界を定義することで解決できる

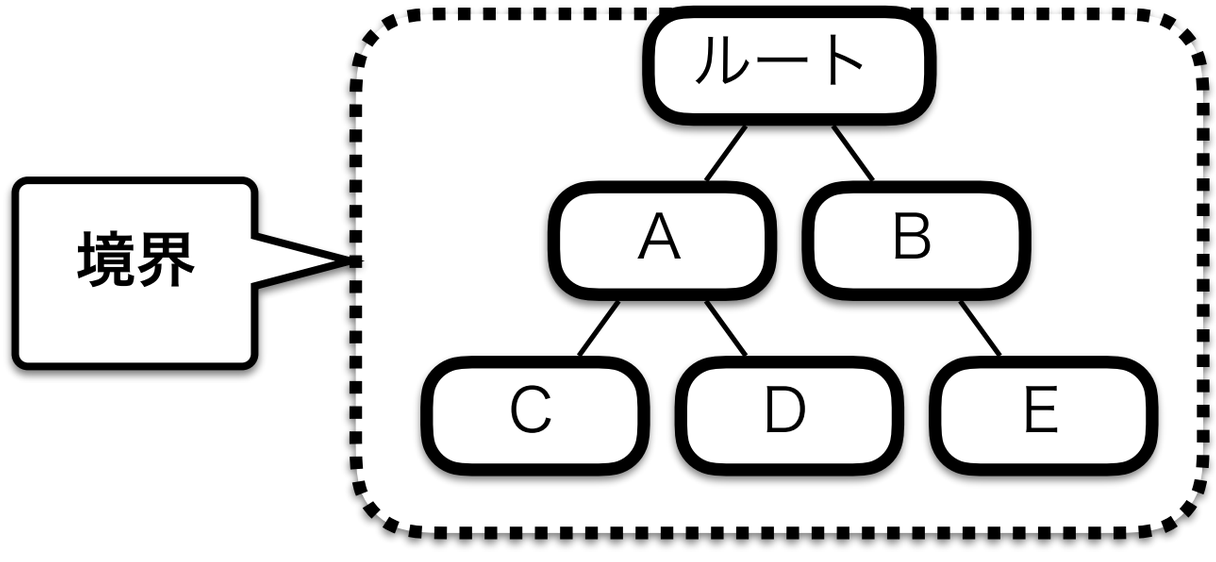



```
// 集約を跨がるようなトランザクション？それとも一つの集約として扱っているのか？
def updateCar(car: Car) = {
  withTranscation { implicit con =>
    updateRecord(car.id, car.tires)
    updateRecord(car.engine.id, car.engine.typeName)
  }
}
// あれ？こっちは単独で更新するようだ。境界が明示されていない…
def updateEngine(engine: Engine) = {
  withTranscation { implicit con =>
    updateRecord(engine.id, engine.typeName)
  }
}
```

境界をどのように定義するのか

集約

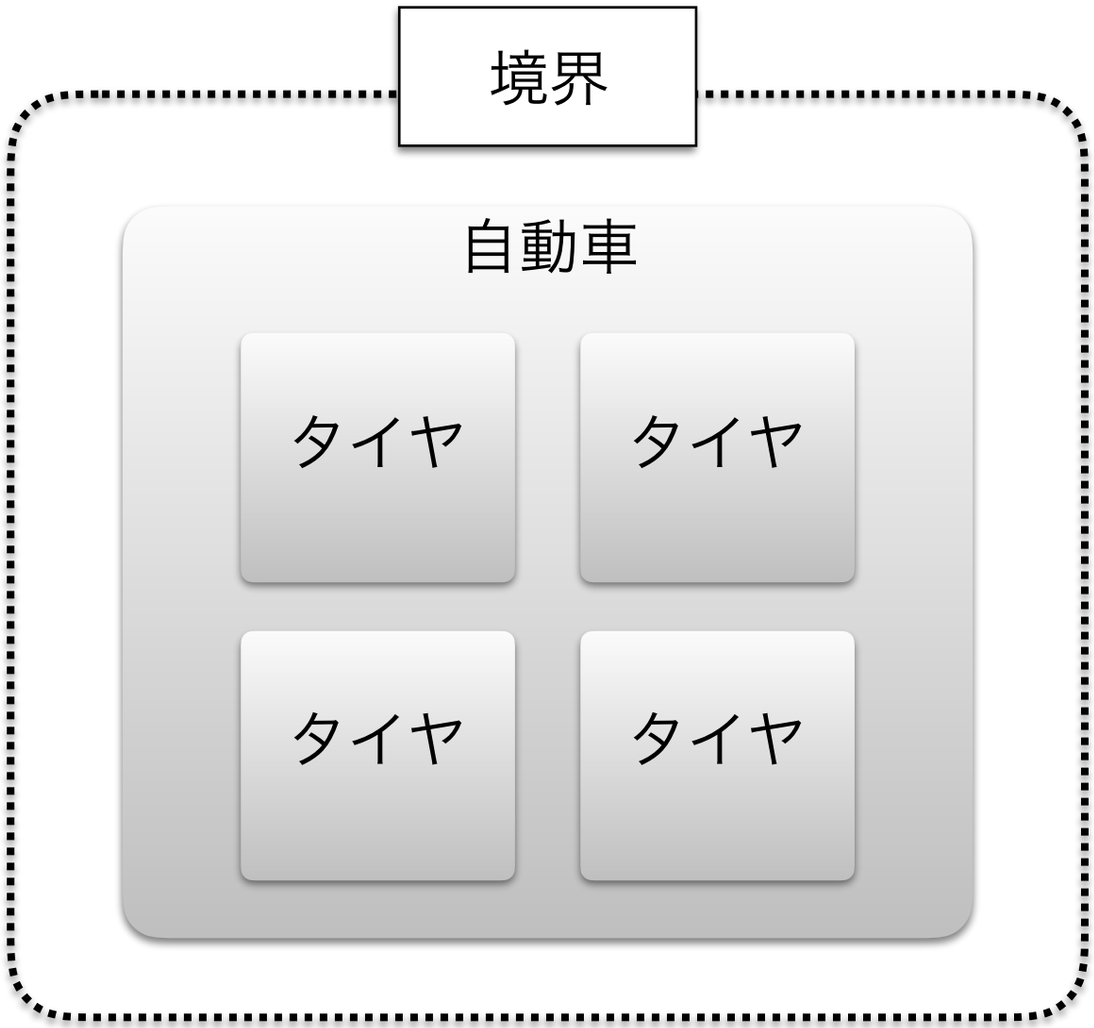
- モデル内にある参照をカプセル化するための抽象化が必要。**関連するオブジェクト群をひとかたまりとし、データを変更する単位として扱う。**これが集約の役割
- 各集約には**境界とルート**がある
- 境界は集約内部に何があるかを定義する
- ルートは集約に含まれている特定の1エンティティ(ルートエンティティ)



自動車修理店向けシステムの例

集約

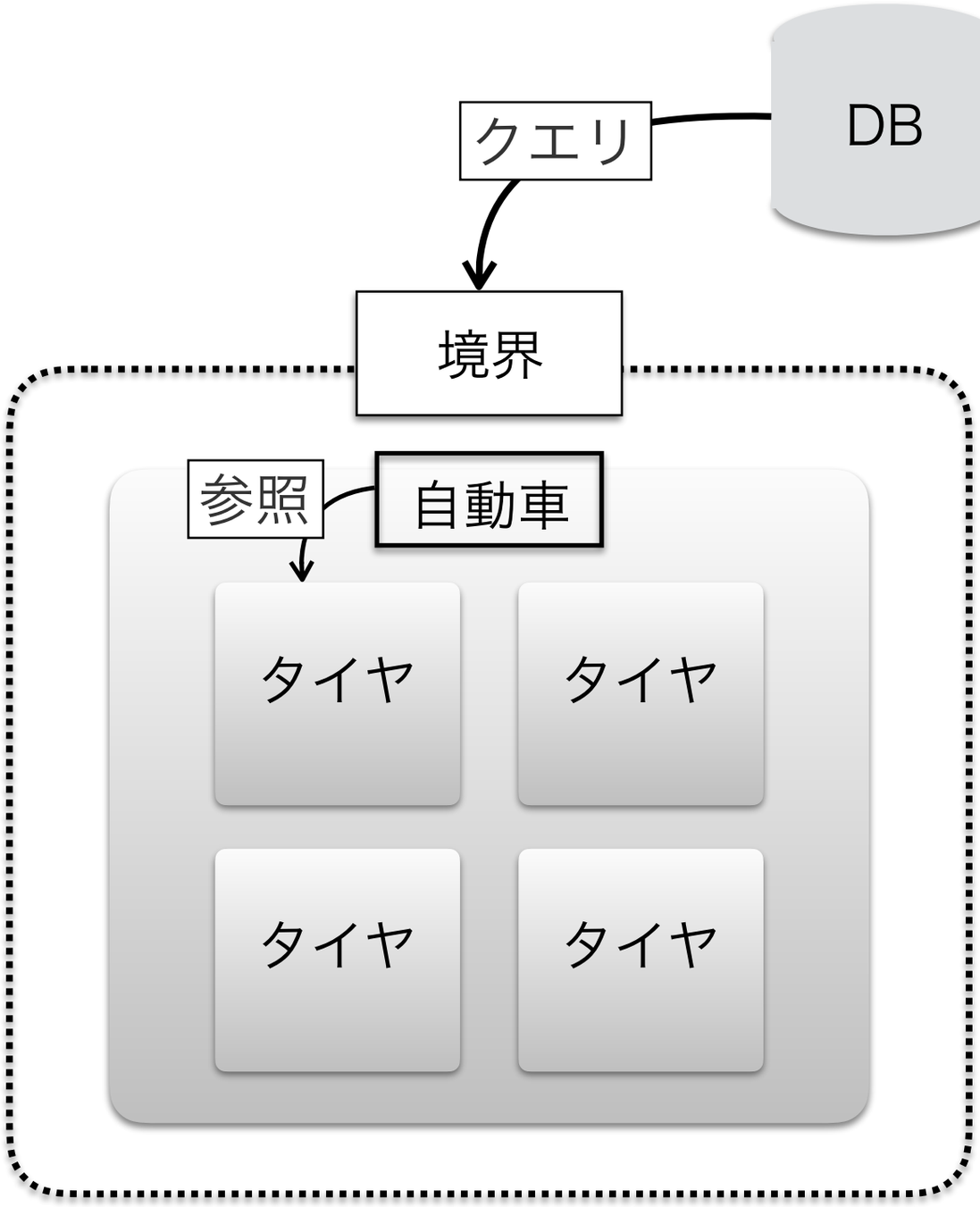
- タイヤのローテーションの記録と、各タイヤの走行距離と接地面の摩耗度合いを追跡する
- 自動車はグローバルな同一性を持つルートエンティティ。識別子には車両番号を利用する
- タイヤは自動車から離れると同一性を気にしなくてよい。交換された古いタイヤは追跡を止めるか、ただの値オブジェクトとして在庫を管理すればよい



FYI: オブジェクトを辿る二つの方法

集約

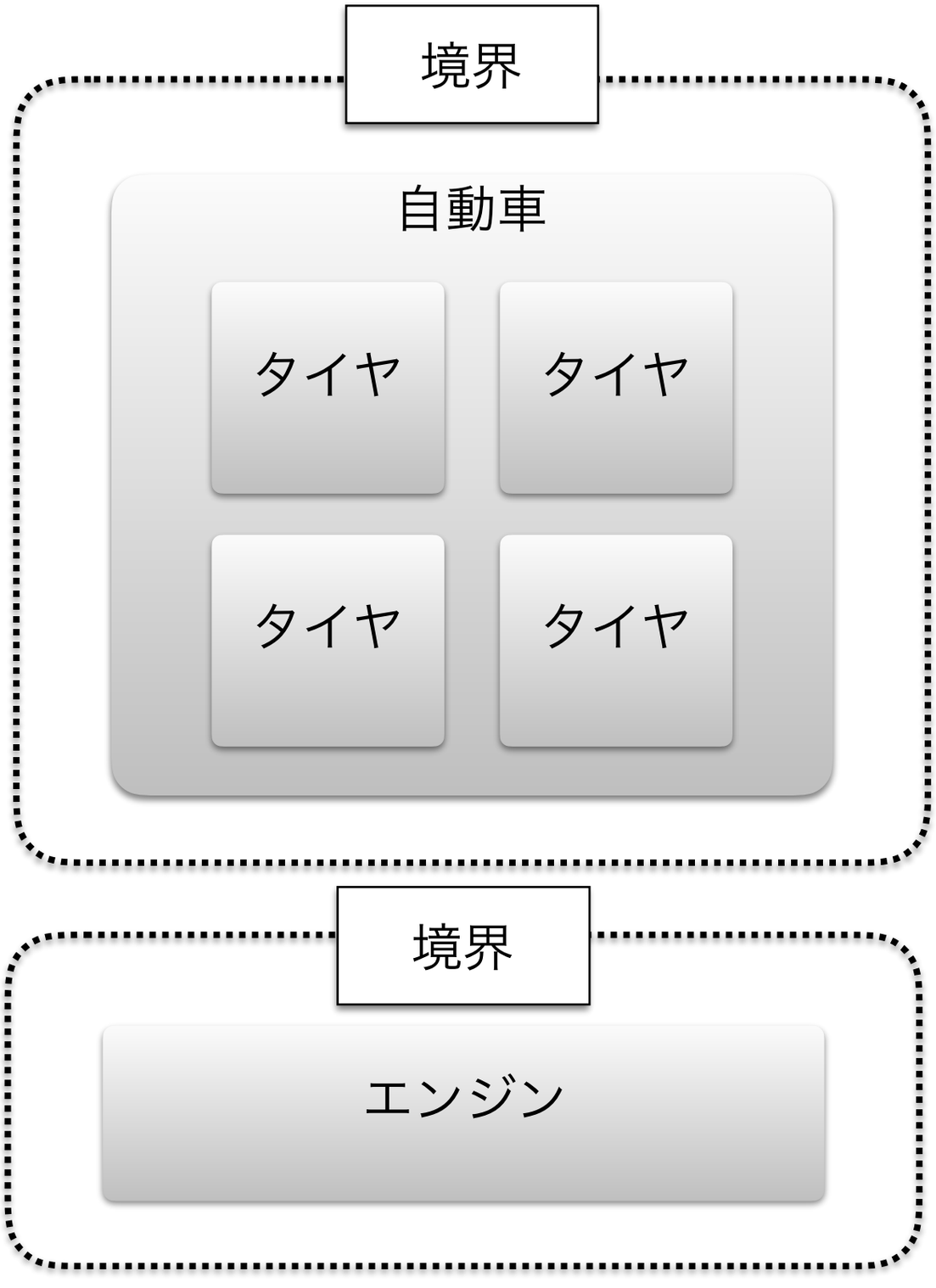
- 特定のタイヤを探す場合は、
  まず自動車を特定してから。
  車両番号を使ってDBに**クエリ**
  をかける
- そのため、タイヤが自動車に装着されている間もグローバルな同一性は不要。ルートエンティティから**参照**を辿る
- すべてのオブジェクトが集約だと、整合性の維持管理が煩雑化する可能性がある



ユースケースから境界を見つけ出す

集約

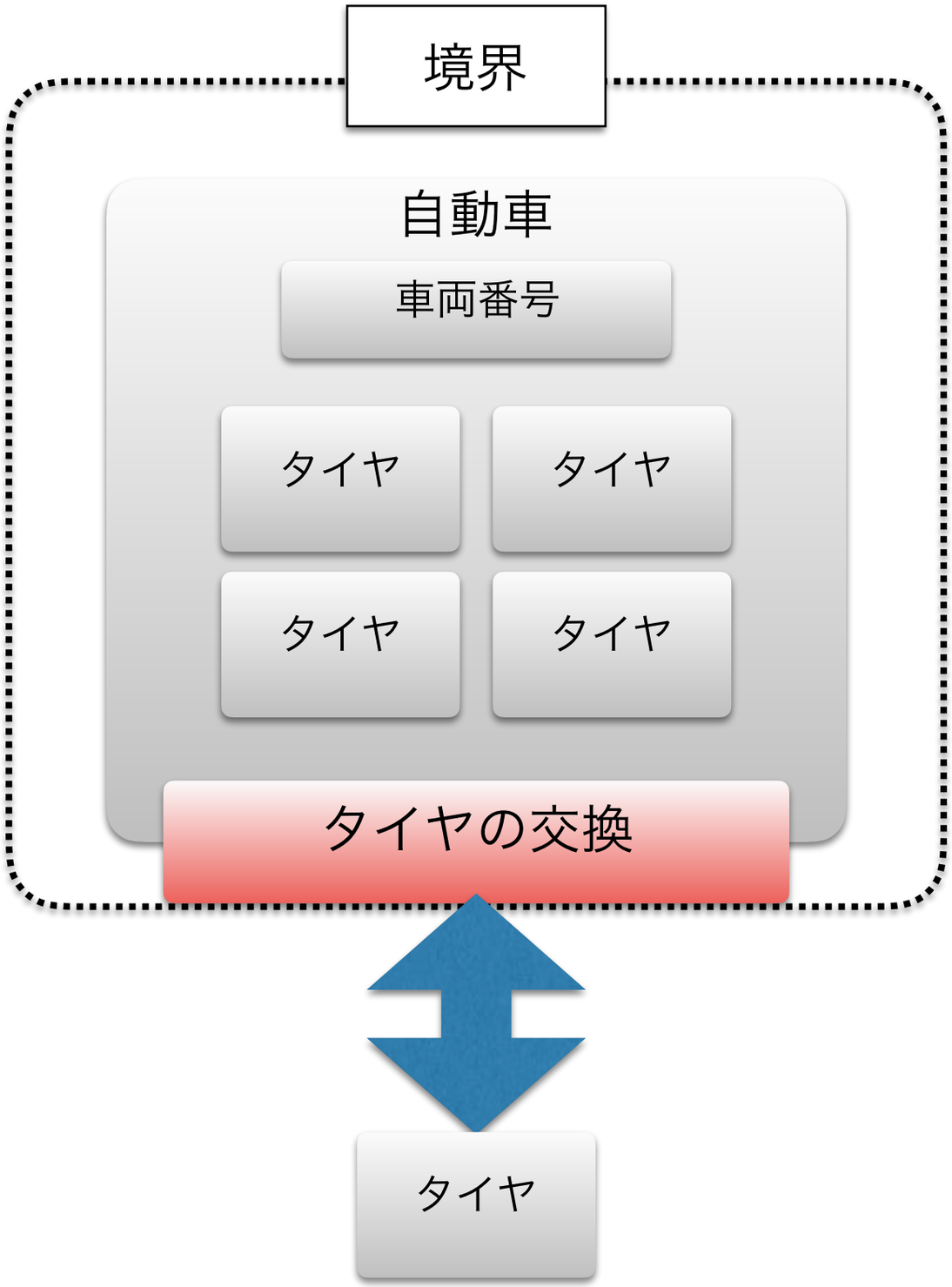
- 一方、エンジンは独立した集約 or 集約内部のオブジェクト？
- 自動車と独立して追跡するユースケースであれば、エンジンは独立した集約となる。
- **車がエンジンを内包するという思い込みとは相反するかもしれないが、ユースケースを無視してモデルを作ると役に立たないモデルになってしまう**
- ユースケースからモデルを創造することは必要だが、**モデルの整合性が成り立つか検証も必要。これは両輪**



Evansルール1

集約

- ルートエンティティは、グローバルな同一性を持ち、**不変条件**(そのモデルが持つ性質)をチェックする責務を負う
- タイヤの交換はメソッドなどを介して行われる。決して内部のオブジェクトを直接操作させてはならない(タイヤローテーションルールを強制させるなどの不変条件を維持するために)



不变条件 is 何?

横道に逸れて

契約による設計(DbC)

先頭の三文字を返すメソッド1

DbC

- 引数の文字列から先頭三文字を抜き出し返すメソッド
- 正しく動作しそうです...

```
package example

object Example1 extends App {

  def getThree(s: String): String = s.substring(0, 3)

  println(getThree("12345"))

}
```

```
123
```

先頭の三文字を返すメソッド2

DbC

- これはサービス側のバグでしょうか？
- それともクライアント側のバグでしょうか？

```
package example

object Example1 extends App {

  def getThree(s: String): String = s.substring(0, 3)

  println(getThree("12"))

}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 3
    at java.lang.String.substring(String.java:1963)
    at example.Example1$.getThree(Example1.scala:5)
…
    at scala.collection.immutable.List.foreach(List.scala:389)
    at scala.App.main(App.scala:76)
    at scala.App.main$(App.scala:74)
    at example.Example1$.main(Example1.scala:3)
    at example.Example1.main(Example1.scala)
```

答えは仕様未定義

- 答えは、いいえ
- メソッドの挙動が正しいか間違っているかはっきりしていません。このケースでは仕様が明確に定義されていない(仕様未定義)
- コードを読めば仕様はわかるのでは？

コードから仕様がわからない例

DbC

- 戻り値をチェックするだけでは、正しさはわからない

```
package example

case class User(id: Long, name: String, deptId: Long)

trait UserRepository {
  def findByDeptId(depId: Long): Option[Seq[User]]
}

object UserRepository1 extends UserRepository {
  override def findByDeptId(depId: Long): Option[Seq[User]] = None
}

object UserRepository2 extends UserRepository {
  override def findByDeptId(depId: Long): Option[Seq[User]] = Some(Seq.empty)
}

object Example2 extends App {

 def printUser(userRepository: UserRepository, deptId: Long) =
   userRepository.findByDeptId(deptId) match {
     case None => println("users are empty")
     case Some(users) => println(s"users = $users")
   }

  printUser(UserRepository1, 1) // users are empty
  printUser(UserRepository2, 1) // users = List()

}
```

どちらの戻り値を返すのが正しい？

一般的に、仕様を自然言語で示さない限り、わからない。
どう表すといいか → DbCが一つの答え

契約による設計(Design by Contract)

DbC

- バグ(＝仕様と実装の矛盾)と判断する正しさとはいったい何か？
  - **“正しさとは相対的な概念である”**
    - オブジェクト指向入門 第2版 原則・コンセプト 第11章 契約による設計:信頼性の高いソフトウェアを構築する
  - **ソフトウェア要素そのものが正しいかどうかではなく、対応する仕様と一致するかを論ずるべき(ソフトウェア要素と仕様のペアで考える)**
  - **正しさを評価するには、“表明”を使った仕様の記述方法を利用するのが一般的**
- 正しさの公式(ホアのトリプル) = **{P} A {Q}**
  - 「任意のAの実行は、Pの状態になったときに始まり、Qの状態になったときに終了する」
  - P = 事前条件(Aが呼ばれる前に満たすべき条件)。クライアントを束縛する条件
  - A = ソフトウェア要素(処理本体)
  - Q = 事後条件(Aが呼ばれた後に満たすべき条件)。サービスを束縛する条件

サービスとクライアントが交わす契約

DbC

- 顧客(クライアント)が提供者(サービス)を利用する前提の契約
- あるクラスに対応するルーチンrが、require preとensure postを宣言している場合、そのクラスは顧客(クライアント)に以下のことを言っているに等しい
- 「もしそちらが pre を満たした状態でr を呼ぶと約束して下さるならば、お返しに、post を満たす状態を最終的に実現することをお約束します」

例: 事前・事後条件を表明するメソッド

DbC

- requireを使って事前条件を表明する。ensuringは事後条件の表明
- クライアント側がStringIndexOutOfBoundsExceptionに依存しているならば、サービス側の実装に依存していることになる！

```
def getThree(s: String): String = {
  require(s.length >= 3, "length of s is too long!")
  s.substring(0, 3)
}.ensuring(_.length == 3, "length of s doesn't equals 3")
```

```
Exception in thread "main" java.lang.IllegalArgumentException: requirement failed:
length of s is too long!
   at scala.Predef$.require(Predef.scala:277)
   at example.Example1$.getThree(Example1.scala:6)
…
   at scala.App.$anonfun$main$1$adapted(App.scala:76)
   at scala.collection.immutable.List.foreach(List.scala:389)
   at scala.App.main(App.scala:76)
   at scala.App.main$(App.scala:74)
   at example.Example1$.main(Example1.scala:3)
   at example.Example1.main(Example1.scala)
```

例) PHPのmt_rand()関数

DbC

PHP の mt_rand() は一貫して壊れている(consistently broken ...
sucrose.hatenablog.com/entry/2016/02/19/235506
2016/02/19 - PHPでMerserne Twister法で擬似乱数を生成する関数のmt_rand()にバグがあり出力がおかしい、という話が流れてきておもしろかったので簡単にまとめておく kusanoさんがmt_rand()の実装に9年以上前から1文字違いでバグがあったことを ...

PHPのmt_rand()にプルリクを送った - kusano_k's blog 84 users
kusano-k.hatenablog.com/entry/2016/02/20/223229
2016/02/20 - PHP の mt_rand() は一貫して壊れている(consistently broken)らしい - 唯物是真 @Scaled_Wurm PHPのmt_rand()が実装 ... 乱数を推測するツールがあると教えてもらってそのツールを見ていたら、ソースコードの中に mt_twist という関数と ...

なぜmt_rand()の誤った実装をサクッと修正できないのか - Qii...
qiita.com › Items › PHP
2016/02/20 - scaled_wurm さんのPHP の mt_rand() は一貫して壊れている(consistently broken)らしいに便乗してみる。 擬似乱数の特性は「周期性があること」です。周期性とはつまり再現性で、ほとんどの擬似乱数は長い周期性をもち次に出力される数 ...

はてなブックマーク - PHP の mt_rand() は一貫して壊れてい...
b.hatena.ne.jp/entry/sucrose.hatenablog.com/entry/2016/02/.../235506
2016/02/20 - PHP の mt_rand() は一貫して壊れている(consistently broken)らしい - 唯物是真 @Scaled_Wurm ... psne 一貫して壊れている物を直したら結果が変わってしまう (例えば再現性のある乱数の結果が欲しい条件でmt_srand(1)としたコードに ...

PHP の mt_rand() は一貫して壊れている(consistently broken ...
buconews.com/it/8809
2016 – 02 – 19 PHP の mt_rand() は一貫して壊れている(consistently broken)らしい list Tweet PHP で Mersenne Twister法 で 擬似乱数 を生成する関数の mt_rand() にバグがあり出力がおかしい、という話が流れてきておもしろかったので簡単にまとめて ...

- mt_rand()は一貫して壊れているが、困っているのは、契約ではなくサービス側の実装に依存してしまった人たち

表明は制御構造にあらず

DbC

- 実行時の表明違反は、そのソフトウェアにバグがある証拠
  - 事前条件違反は、クライアントにバグがある証拠
  - 事後条件違反は、サービスにバグがある証拠

不変条件とは

DbC

- ホアのトリプルによって、処理の正しさを判定することができるが、**クラスの属性(データ)にも満たすべき条件**がある。それを**不変条件**と呼ぶ
- 例えば、自然数の不変条件は「0以上」です。

- **{ INV & P } A { INV & Q}** が成り立つとき、すべてが成功する

それぞれの契約を表明する

DbC

- 注文オブジェクトの注文アイテムは空であってはならない。
- addOrderItem時に事前条件と事後条件を確認する

```
case class Order(items: Seq[OrderItem]) {
  require(items.nonEmpty)

  def addOrderItems(otherItems: Seq[OrderItem]): Order = {
    require(otherItems.nonEmpty)
    copy(items = this.items ++ otherItems)
  }.ensuring(items.nonEmpty)

}
```

- もっと簡潔に記述できないか

コンストラクタでの表明

DbC

- Immutableであればコンストラクタで、不変条件を表明すれば、それを使うメソッドでの表明を省略できる

```
case class OrderItems(values: Seq[OrderItem]) {
  require(values.nonEmpty)
}

object OrderItems {
  implicit object SeqmigroupInstance extends Semigroup[OrderItems] {
    override def combine(x: OrderItems, y: OrderItems): OrderItems =
      OrderItems(x.values ++ y.values)
  }
}

case class Order(items: OrderItems) {

  def addOrderItems(otherItems: OrderItems): Order =
    copy(items = this.items.combine(otherItems))

}
```

FYI: 型レベルでの表明

DbC

- cats.data.NonEmptyListでの例
- 型レベルで不変条件を表明できる

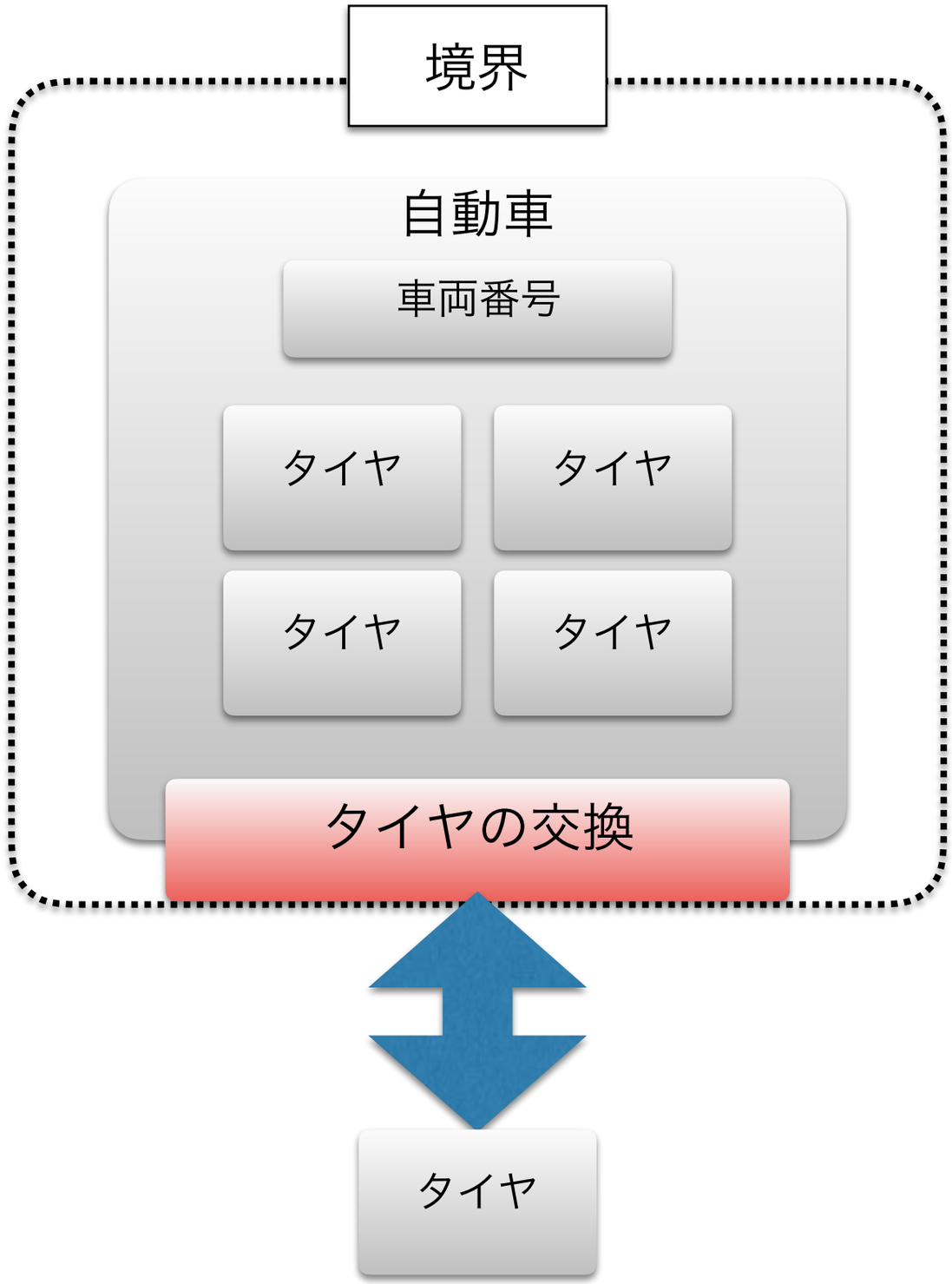
```
object Order {

  type OrderItems = NonEmptyList[OrderItem]

  // NonEmptyListは、先頭要素のheadを必ず必要とするため、型として空にはできない
  // final case class NonEmptyList[+A](head: A, tail: List[A]) { …

}

case class Order(items: OrderItems) {

  def addOrderItem(otherItems: OrderItems): Order =
    copy(items = this.items.combine(otherItems))

}
```

- すべてのケースでこの方法が使えるわけではない

再掲: Evansルール1

集約

- ルートエンティティは、グローバルな同一性を持ち、**不変条件**(そのモデルが持つ性質)をチェックする責務を負う
- タイヤの交換はメソッドなどを介して行われる。決して内部のオブジェクトを直接操作させてはならない(タイヤローテーションルールを強制させるなどの不変条件を維持するために)



FYI: 可変フィールドを暴露してはならない

集約

- 集約の不変条件を破壊できてしまう例

```scala
case class Order(private val _items: Seq[OrderItem]) {
  require(_items.nonEmpty, "items are empty!")
  val items: ListBuffer[OrderItem] = ListBuffer.empty[OrderItem]
  addOrderItems(_items)

  def addOrderItems(otherOrderItems: Seq[OrderItem]): Unit = {
    require(otherOrderItems.nonEmpty, "otherOrderItems are empty!")
    items.appendAll(otherOrderItems)
    items.ensuring(_.nonEmpty, "items are empty!")
  }

}

object Main extends App {

  val order = Order(Seq(OrderItem(1L, ProductName("red-pen"), 100, 1)))
  order.addOrderItems(
    mutable.Seq(OrderItem(2L, ProductName("blue-pen"), 100, 2)))
  println(order.items)
  // ListBuffer(OrderItem(1,ProductName(red-pen),100,1), OrderItem(2,ProductName(blue-
pen),100,2))

  order.items.clear()
  println(order.items)
  // ListBuffer()

}
```

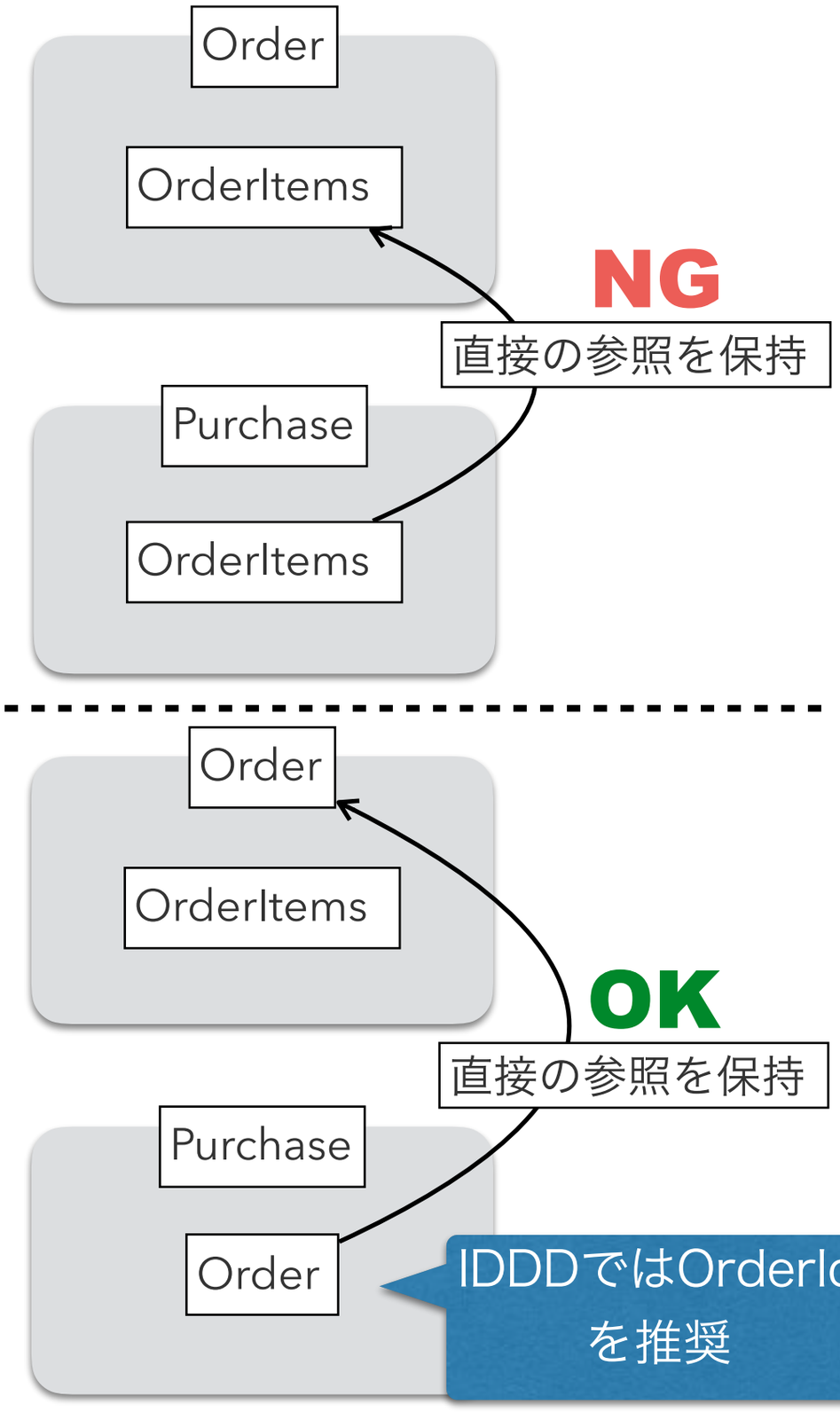
状態を可変できるフィールドは集約の境界外に暴露してはならない
(varでも同様だが、valでも可変クラスの場合も注意が必要)

集約の預かり知らない処理で不変条件を破壊できる

Evansルール2

集約

- 集約外のオブジェクトは、**ルートエンティティを除き**、境界内部への参照を保持することができない。**内部オブジェクトへの参照を他のオブジェクトに渡し一時的に利用することができるが、フィールドとして利用できない**(所有権の明確化)
- DBからクエリを使って取得できるのは、ルートエンティティのみ
- **削除の際は、集約に内在するあらゆるオブジェクトを一度に削除する**(ルート以外は外部から参照されていないのでルートを削除すればGCも効率がよい)
- 集約内部のオブジェクトに対する**変更がコミットされるときは、集約全体の不変条件がすべて満たされていること**

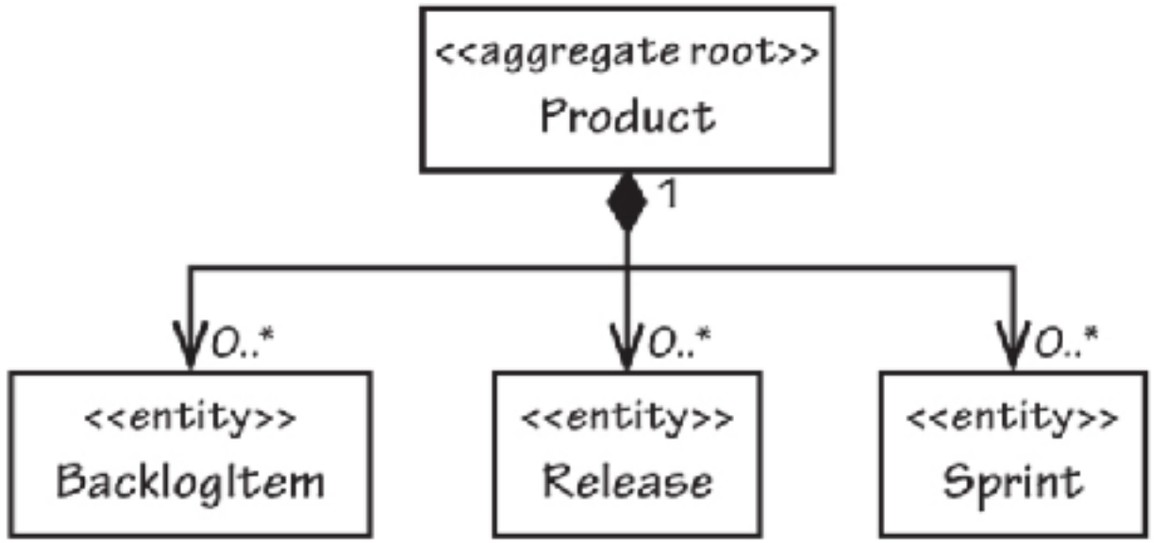


境界定義の善し悪し

例:巨大な集約

集約(Vernon)

- バックログ、リリース、スプリントのすべてを保持するプロダクト集約。プロダクトにバージョンがあり楽観ロックされる
- AとBのユーザが同じバージョンで使い始める→Aがバックログを追加しコミット
  Bがリリースを変更してコミットするが、楽観ロックエラー



- 頻繁に複数のユーザが更新するケースでは、同時に変更しようとしたとき、成功するのは1人だけは受け入れられない

例:複数の集約

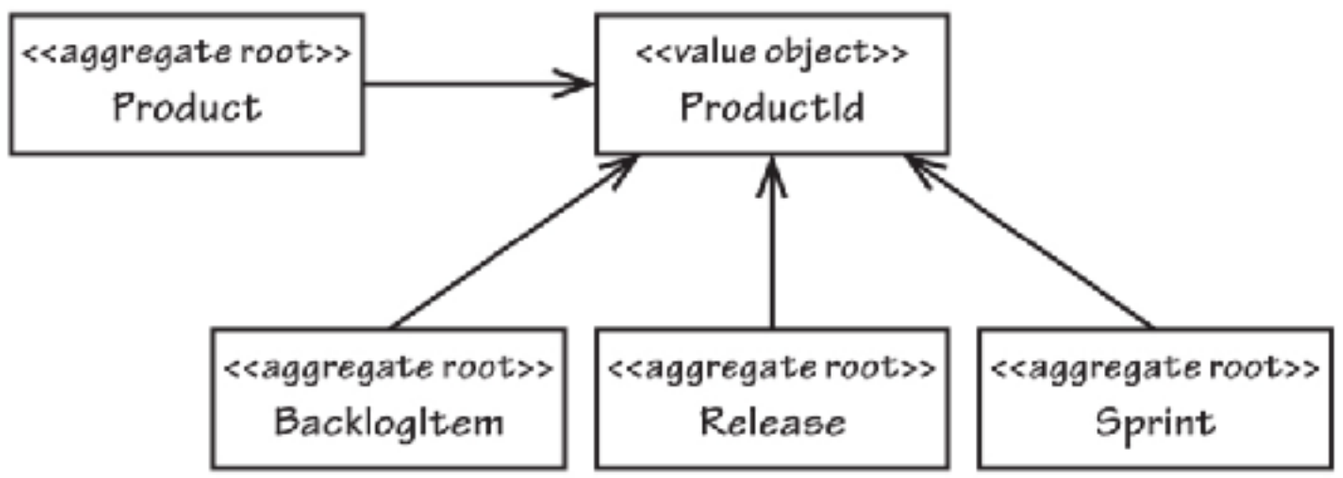
集約(Vernon)

- 巨大な集約を四分割。ProductIdによって関連付けられる。

  バックログの追加時は、トランザクション境界内で行うため、競合が起きずに安全に更新できる
- トランザクション面では有利でも、クライアント側からすると不便かもしれない(Productがあることを確認してBacklogItemを追加など)
- バックログなどのN数

  を制限する不変条件

  が守れなくなる可能性がある



Vernonの集約設計ルール

ルール1: 真の不変条件を、整合性の境界内にモデリングする

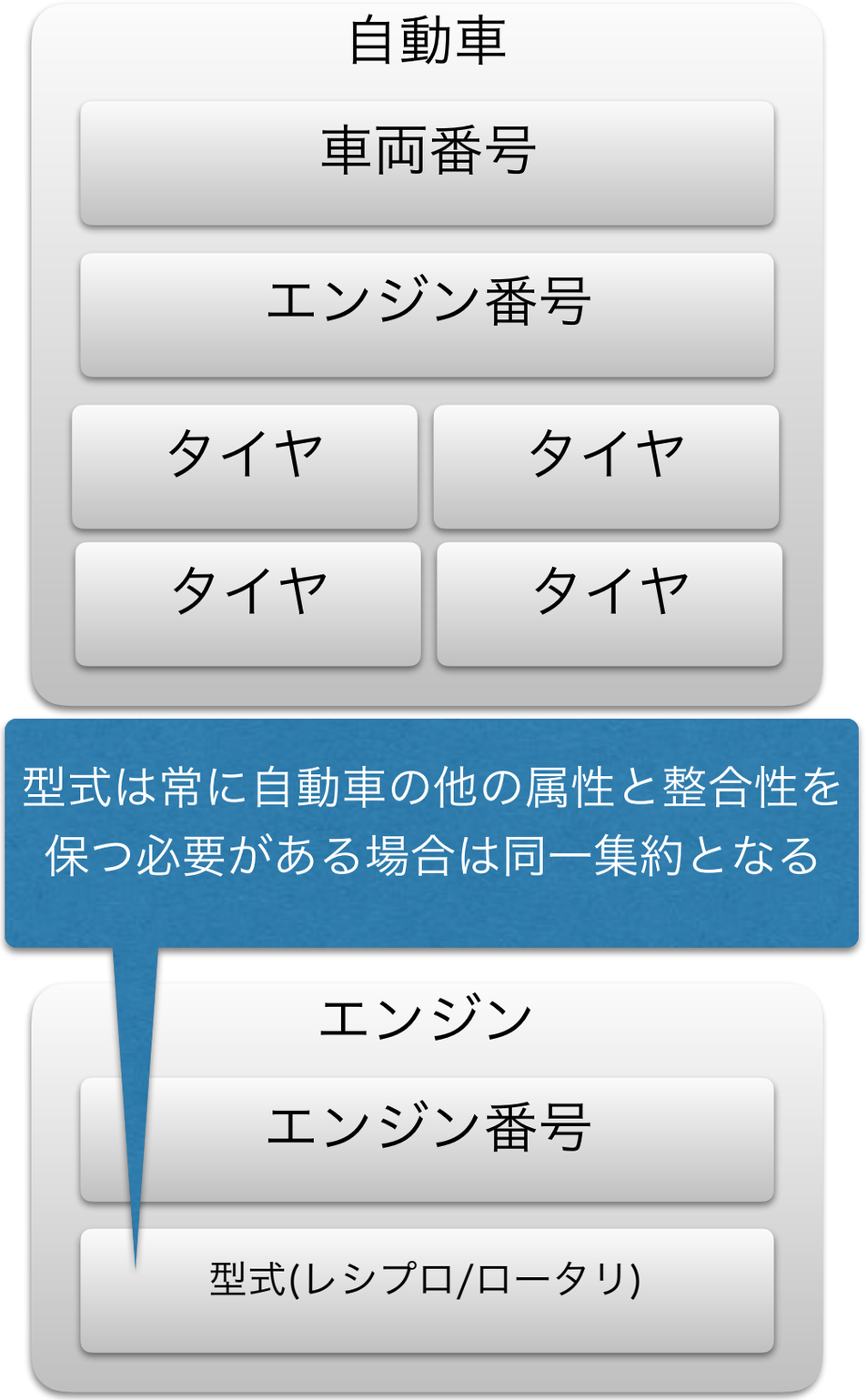
集約(Vernon)

- 整合性には2種類ある: **トランザクション整合性**(強い整合性)と**結果整合性**(弱い整合性)。不変条件に必要な整合性は、トランザクション整合性
- **集約はトランザクション整合性の境界と同義**。ドメインで必要とするあらゆる変更に対して、**トランザクション内での不変条件の整合性を安全に維持する**
- **トランザクションの分析後**に、集約の設計の善し悪しを正しく判断することができる

ルール2: 小さな集約を設計する

集約(Vernon)

- 巨大な集約では、基本操作の多くで、**複数の大規模なコレクションの読み込みが発生する。**パフォーマンスに悪影響がある
- "他の要素との整合性を保つ必要があるもの"が判断基準。**ただしユースケースを一方的に無視できない(Evans)**
- モデルの一部をエンティティ化する場合
  - それが今後も変わり続けるなら→集約化
  - 変更時にまるごと置き換えればよい→値オブジェクト
    - **ルートエンティティと一緒にシリアライズでき、クエリも単純化できる。エンティティの方がオーバーヘッドがある**



FYI: ユースケースを鵜呑みにしない

集約(Vernon)

- ユースケース分析には、ビジネスアナリストが重要な役割を果たすが、**開発者の視点を反映したものとは限らない**
- **個々のユースケースとモデルの設計の整合性を合わせる**必要がある
  - 複数のトランザクション(ex. Product, BacklogItem)を単一のトランザクションとして扱うようなユースケースで問題が生じる
- **このようなユースケースは鵜呑みにしないこと**
  - 場合によっては、モデルもしくはユースケースを見直す必要がある
  - できれば、最初から全員同席でモデリングした方がよい

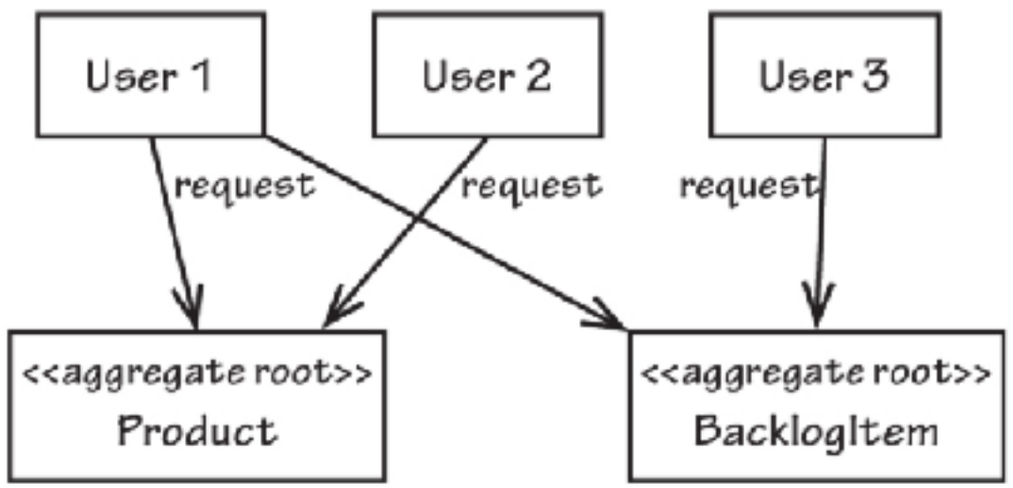
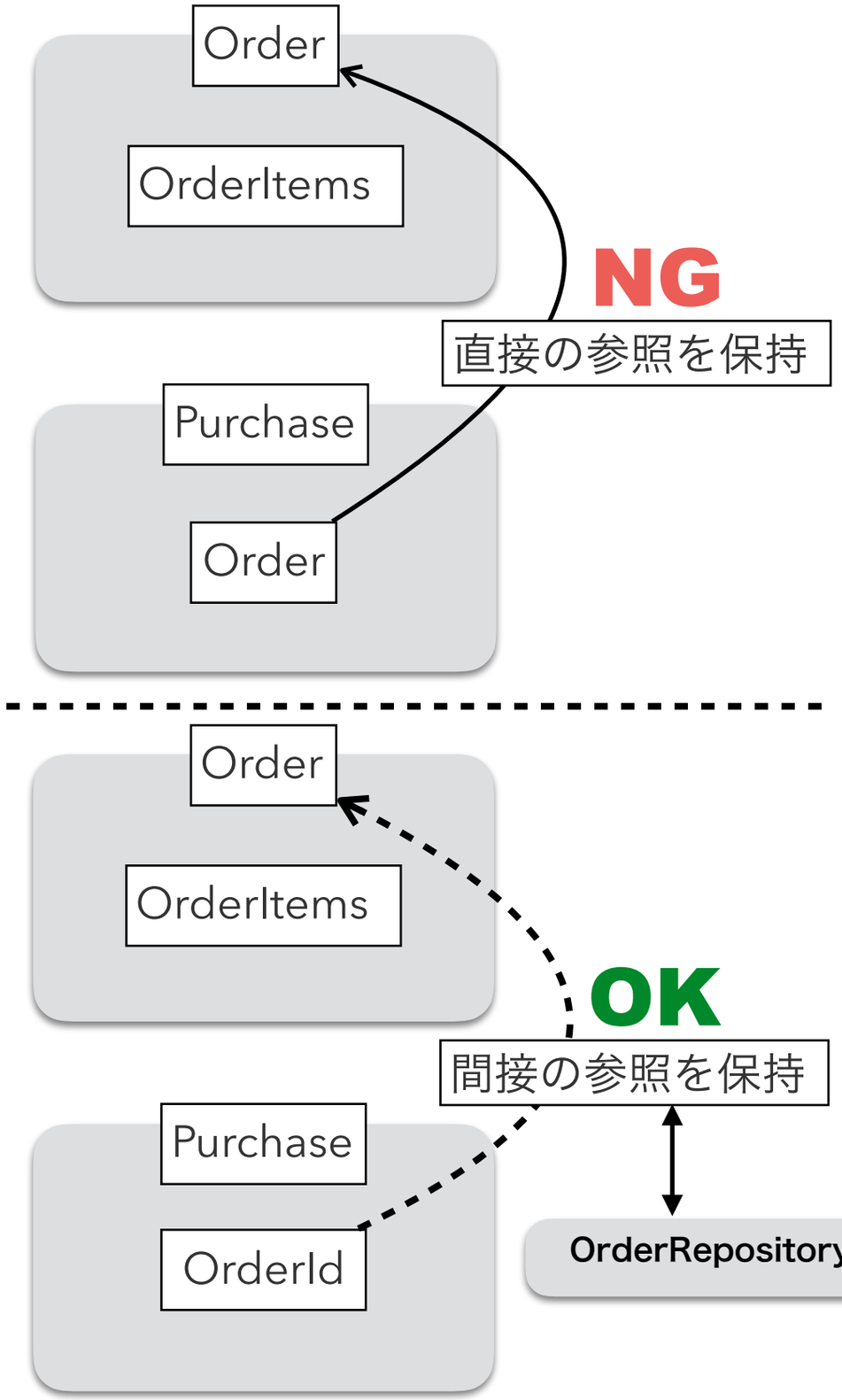


図10-4：三人のユーザーが、二つの集約のインスタンスに同時にアクセスしようとして、並行性制御の競合が起こる。トランザクションの失敗が多発することになるだろう

ルール3: 他の集約への参照には、その識別子を利用する

集約(Vernon)

- ある集約が別の集約のルートエンティティへの参照を保持してもかまわないが、**参照先の集約と参照元の集約が同じ境界に属するわけではない**
- **同一トランザクション内で両方を変更してはいけない。**変更できるのはどちらか一方だけ
  - 同一トランザクションでは、結局 巨大な集約になる
- **他の集約への参照を保持しなければ、その集約を変更することができない**
  - 外部の集約への参照はIDだけを利用しポインタは保持しないようにする
  - メモリ消費量も抑えられる。割り当てだけでなく、GC面でも好影響が及ぶ



ルール4: 境界の外部では結果整合性を用いる

集約(Vernon)

- クライアントからの単一のリクエストで、複数の集約を変更する場合は結果整合性を使うこと
  - ある集約が参照する外部の集約は削除されていかもしれないし、その操作で想定しない状態に遷移しているかもしれない。(プロダクトが持つタグに関連付く、バックログとスプリントを消そうしたら、すでに消えていることがある)
- ある集約が状態を変更する際に、サブスクライバにドメインイベントを通知し、整合性を維持する処理を実行することも可能

まとめ

- ビジネスルールを集約の不変条件として作り込むことで、堅牢なシステムの実現に貢献できる
- 難しいことだが、ユースケースとモデルの整合性の両方を軽視しないこと
- 集約のより詳細な情報を得る場合は、Evans本 第2部第6章 集約、Vernon本 第10章 集約を読むとよい