第10章
名前設計

目的が読み取れる名前設計をしよう

この章の目的

◆名前を適当につけると…
- ■責務が入り乱れて密結合になったり
- ■巨大化して神クラスになったり

◆命名を適当にするとどんな弊害が生まれるかを学ぶ
- ■どのように回避するかも学ぶ

内容

内容とキーワード

目的駆動名前設計

目的駆動名前設計

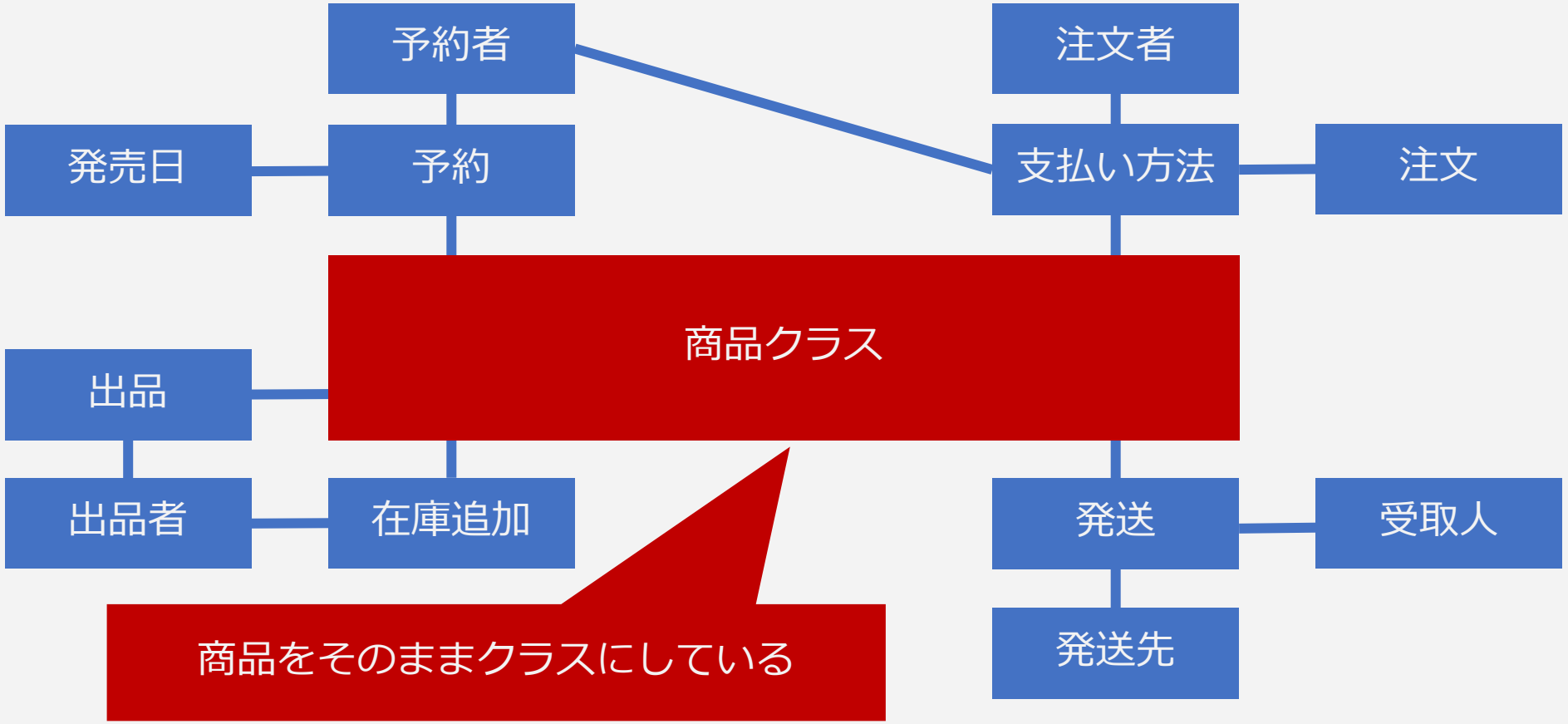
◆名前から目的や意図が読み取れるようにする

■名前を適当につけてしまうと…

- 責務が入り乱れて密結合になる
- クラスが巨大化して神クラスになる

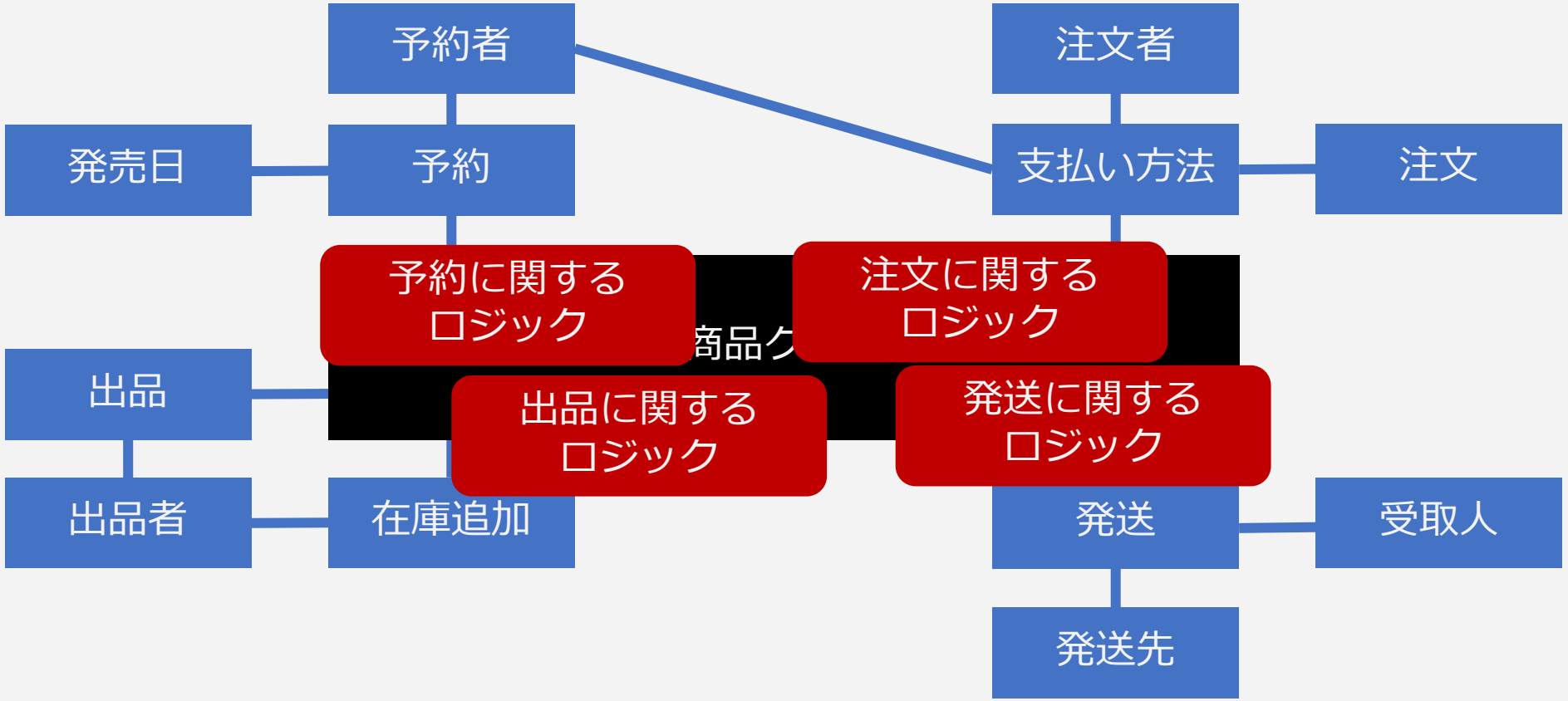
悪魔を呼び寄せる名前

◆例えばECサイトでの商品をクラスとして設計すると



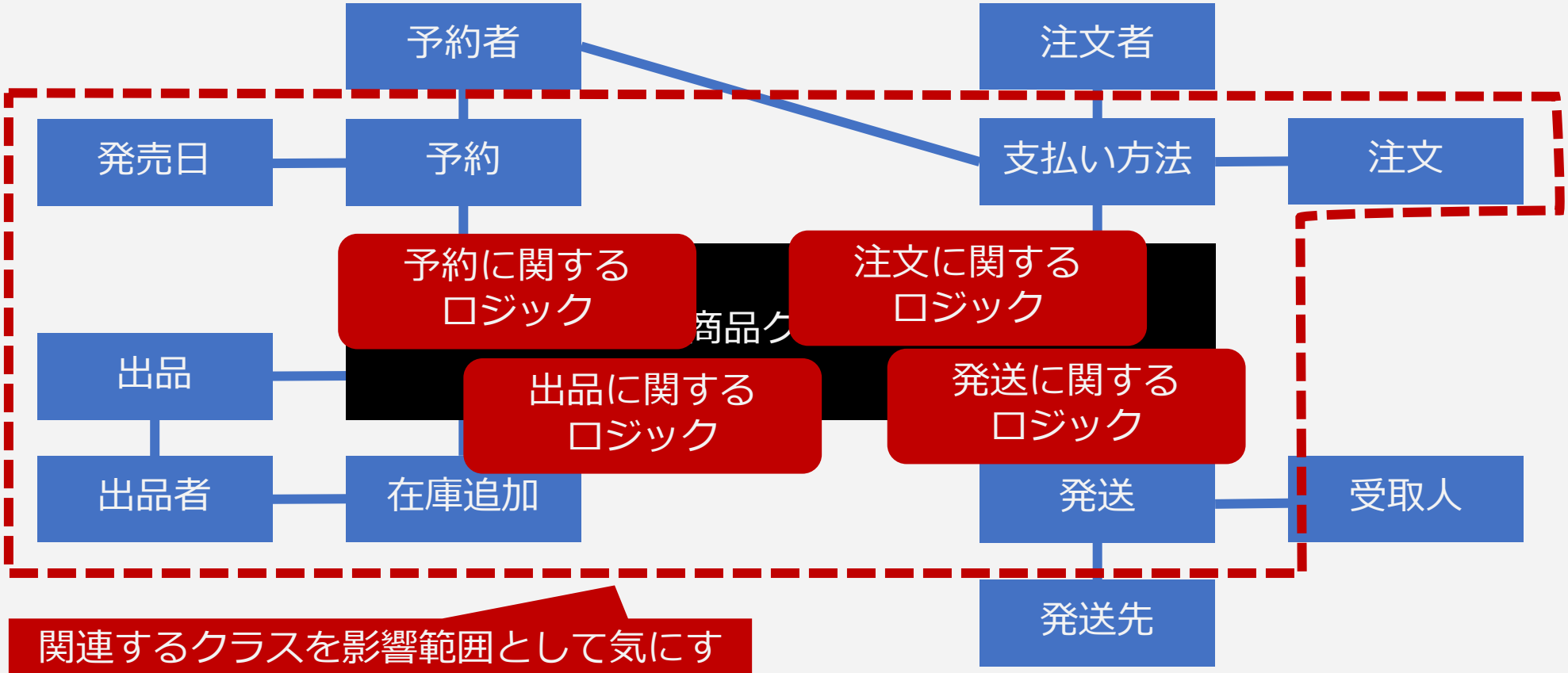
悪魔を呼び寄せる名前

◆よく見ると商品クラスは責務が多い



悪魔を呼び寄せる名前

◆また、影響範囲が広すぎる

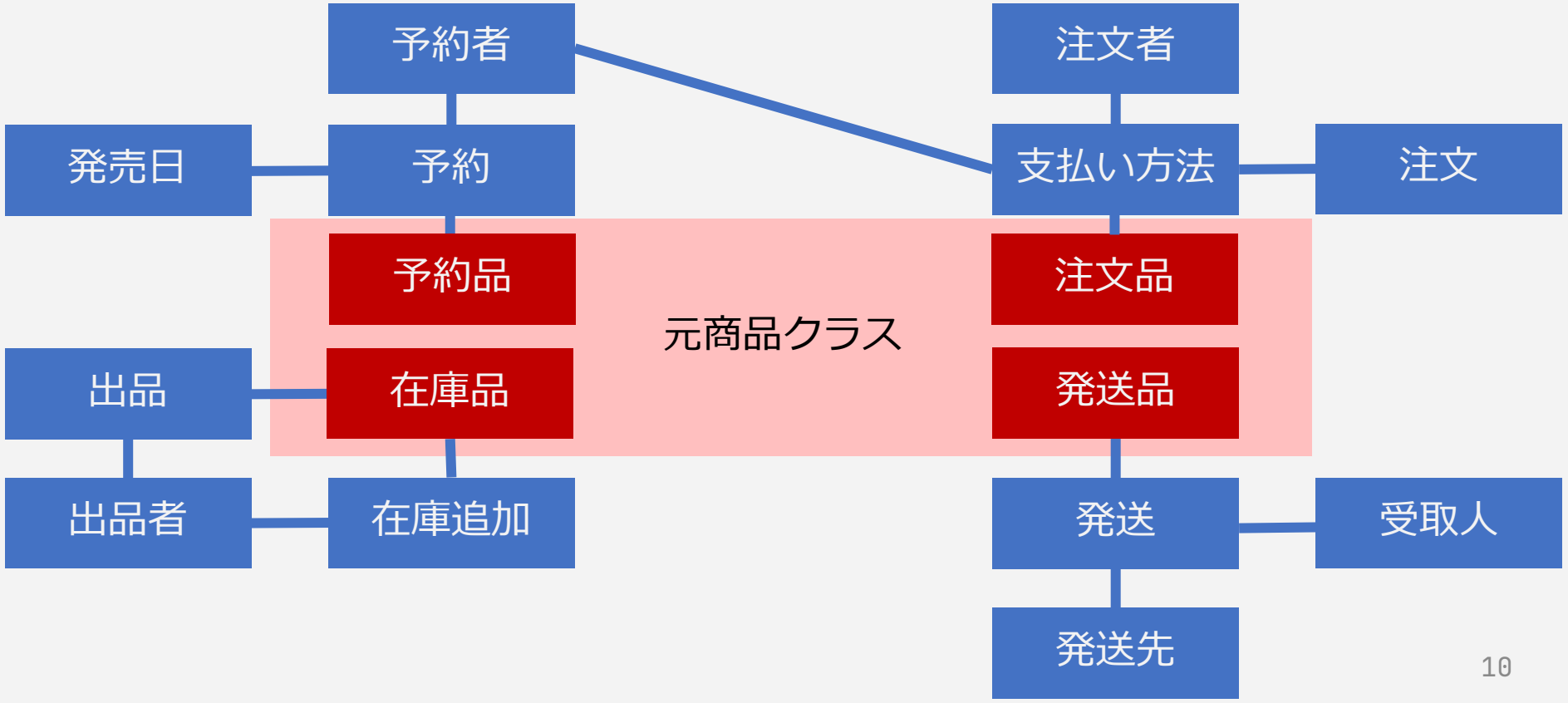


関心の分離

関心の分離

◆関心の分離とは

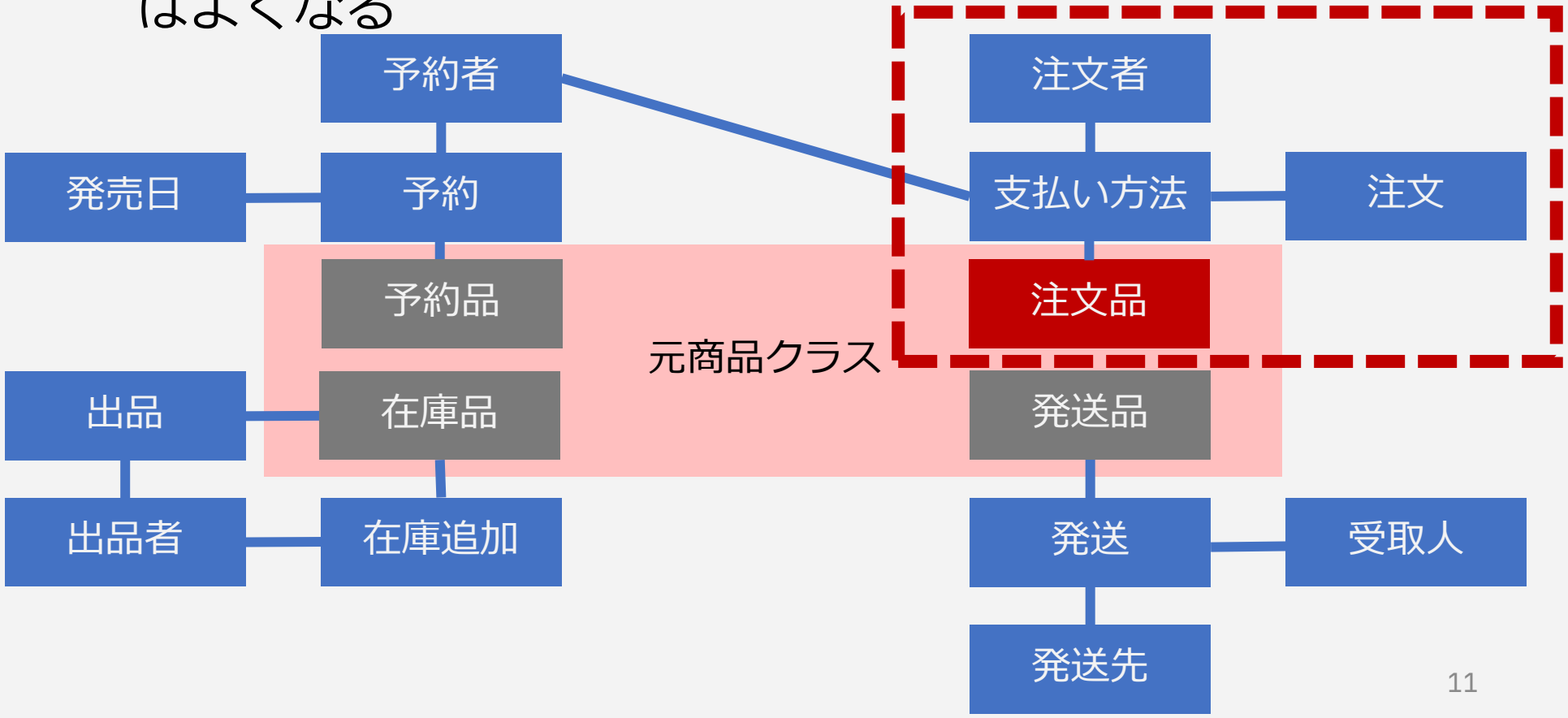
■関心事（ユースケースや目的・役割）ごとに分離（命名）



関心の分離

◆分離することのメリットは？

■例えば注文の仕様変更時は枠線内のクラスだけを気にすればよくなる



大雑把で意味が不明瞭な名前

大雑把で意味不明な名前

◆「商品」などといった命名は大雑把な名目です
- ■予約、注文、発送といった様々な目的で使用されてしまう
- ■こういったクラスを目的不明オブジェクトと呼ぶ

◆大雑把にならないように関心の分離を意識した名前設計を行う

名前設計

名前設計

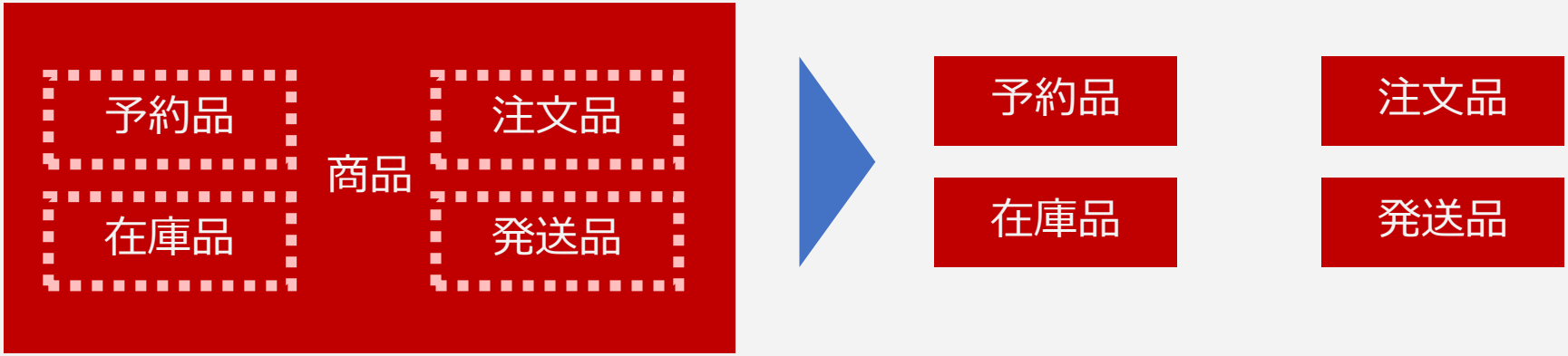
◆名前設計とは
- ■クラスやメソッドに名前をつけること
- ■関心の分離を意識し、ビジネスに沿った名前を付与する

◆事項から、名前設計で重要な事柄を説明します

可能な限り具体的で、意味範囲が狭い、目的に特化した名前を選ぶ

◆目的特化の、より意味の狭い名前を選択

■特定の目的に特化した極めて意味範囲の狭い名前



◆ビジネス目的に特化させることで…

■名前とは無関係なロジックを排除できる

■関係するクラスの個数が少なくなる

■どこを変更すればいいかわかりやすくなる

存在ベースではなく、目的ベースで名前を考える

◆ビジネスに特化した名前とは？

■存在ではなく目的に特化した名前を付ける

存在ベース	目的ベース
住所	配送先、配送元、勤務先、本籍地
金額	請求金額、消費税額、延滞保証料
ユーザー	アカウント、個人プロフィール、職務経歴
ユーザー名	アカウント名、表示名、本名、法人名
商品	入庫品、予約品、注文品、配送品

どんなビジネス目的があるか分析する

◆ビジネス目的に特化するには…
- ■どんなビジネス目的があるか網羅する必要がある

◆そのために…
- ■ソフトウェアが対象とする目的や事柄を分析する
- ■登場人物や事柄を列挙したり、関係性を整理する

声に出して話してみる

◆ラバーダッキング
- ■デバッグ手法の一種
- ■何か問題が発生したときに誰かに説明してみる
- ■すると自ら原因に気づき自己解決する手法

◆開発するメンバーと積極的に話し合う
- ■ユビキタス言語（チーム全体で意図を共有する為の言語）

利用規約を読んでみる

◆目的に特化した名前ってどう考えるの…?

◆利用規約を呼んでみよう

■利用規約はサービスの取り扱いやルールが極めて厳密な言い回しで書かれているため、参考になる

◆例

■購入者が商品購入手続きを完了した時点をもって、売買契約が締結されたものとします。売買契約が締結した場合、出品者は当社にサービス利用料を支払うものとします。
サービス利用料は、売買契約が締結した時点の商品の販売価格に、販売手数料率を乗じた金額となります。

違う名前に置き換えられないか検討する

◆十分に名前を検討しないと…
- ■意味範囲が十分に狭くなかったり
- ■複数の意味を持っていたりする

◆違う名前に書き換えてみてる
- ■もっと意味が狭くできないか検討する

◆例えばホテルの宿泊予約システムについて
- ■「ユーザー」は意味が広すぎる
- ■「顧客」でも「宿泊客」と「支払う人」の両方の意味が
- ■「宿泊客」と「支払者」に分けるのが得策

疎結合高凝集になっていないか点検する

◆目的に特化した命名にすれば…
- ■目的以外のロジックを寄せ付けにくくなる

◆もしそうなっていないのであれば…
- ■名前を見直しましょう

◆他のクラスといつく関連づけられているか確認する
- ■何個も関連づけられているならば密結合の可能性あり

設計時の注意すべきリスク

設計時の注意すべきリスク（1）

◆名前に無頓着になってはいけない
- ■目的駆動名前設計は名前に注意を払い、名前とロジックを対応付けることを前提としている
- ■チーム内で一人でも名前に無頓着な人がいると全体に影響するかも

◆仕様変更時の意味範囲の変化に注意
- ■開発中の仕様変更により名前の意味が変化することがある
- ■名前設計には見直しが必要です

設計時の注意すべきリスク（2）

◆会話には登場するのにコード上に登場しない名前

■会話には登場する重要な概念がコード上に埋没している

（コードに概念はないけど）
キャンセル回数が設定より多い会員は要注意会員です。
キャンセル回数は変数で定義してますよ。

こうなってしまうと
詳しい人に聞かないとコードの理解が
困難になってしまうので
適切にクラスを設計してください

設計時の注意すべきリスク（3）

◆形容詞で区別が必要なときはクラス化のチャンス
- ■口頭で説明する時に形容詞で違いを説明している場合
- ■それはクラスとして表現することができる

◆例
- ■このフラグが立っているときのUserは要注意会員
- ■Ticketクラスは、年齢が60歳以上ならシニア料金用になって、さらに平日なら平日のシニア料金用に変わる

◆このような区別が可能な概念はクラスとしてそれぞれ設計できないか検討しましょう

意図が分からない名前

技術駆動命名

◆技術ベースでの命名を技術駆動命名と呼ぶ
- ■プログラミング用語やコンピュータ用語に基づいた名前
- ■ビジネス目的を指し示す命名にはふさわしくない

◆例
- ■`MemoryStateManager`
- ■`changeIntValue01`

◆注意
- ■技術駆動命名を用いる分野もあります
- ■組み込み系など

ロジック構造をなぞった名前

◆ロジック構造をそもまま名前にしてしまうと…

```
class Magic {
boolean
isMemberHpMoreThanZeroAndIsMemberCanActAndIsMemberMpMoreThanMagic
CostMp(Member member) {
    if (0 < member.hitPoint) {
        if (member.canAct()) {
            if (costMagicPoint <= member.magicPoint) {
                return true;
            }
        }
    }
    return false;
}
```

ヒットポイントが0より大きくて、
行動可能で、
魔法力が残存していて…
うーん…結局何なの？

ロジック構造をなぞった名前

◆意図、目的が分かるように命名する

```
class Magic {
 Boolean canChant(final Member member) {
    if(member.hitPoint <= 0) return false
    if(member.canAct()) return false;
    if(member.magicPoint < costmagicPoint) return false;

    return true;
}
```

なるほど！魔法が使えるかどうか確かめる
ためのメソッドなんだね

驚き最小の原則

◆以下のコードを見てください

```
int count = order.itemCount();
```

このメソッドは多分商品数を返してくれるんだと思いますよ。
だって返り値もInt型だしね。

驚き最小の原則

◆しかしコードの中身を見てみると

```
class Order {
    private final OrderId id;
    private final Items items;
    private GiftPoint giftPoint;

    int itemCount() {
        int count = items.count();

        // 注文数が10個以上ならお買い物ギフトポイントを100追加する。
        if(10 <= count) {
            giftPoint = giftPoint.add(new GiftPoint(100));
        }
        return count;
    }
}
```

え！？ギフトポイントの追加ってなんだ！？

驚き最小の原則

◆驚き最小の原則
- ■使う側が想像したとおりに動作する
- ■予想外の驚きが最小になるように設計する

◆予想通りにストレス無く利用できるように
- ■ロジックと名前を対応づける設計が重要

構造を大きく歪ませてしまう名前

データクラスに陥る名前

◆例えば以下の様なデータクラスを作ってしまうと…

```
class ProductInfo {
    int id;
    String name;
    int price;
    String productCode;
}
```

Infoってことはデータクラスか。
じゃあこのクラスにロジックを実装するのはダメなのかな？

結果的に低凝集になる

データクラスに陥る名前

◆-Info,-Dataと名付けるとデータクラスを想起する
- ■このようなデータのみを想起させる命名は避ける
- ■先ほどのProductInfoはProductに変更すべき

◆例外
- ■DTO(Data Transfer Object)
  - 一部例外的にデータクラスを用いる場合があります
  - 更新責務と表示責務とでモジュールを分離するコマンド・クエリ責務分離(CQRS)と呼ばれるアーキテクチャパターンがある
  - このアーキテクチャパターンではDTOと呼ばれる設計パターンが使用され、データクラスを用います
  - 詳しい説明はここでは省きます

クラスが巨大化する名前

◆Managerとうい名前。意味が大きすぎませんか？
- ■Managerということで様々な責務を負わせがち
- ■Manager内の概念を一つ一つ取り出し、分けましょう

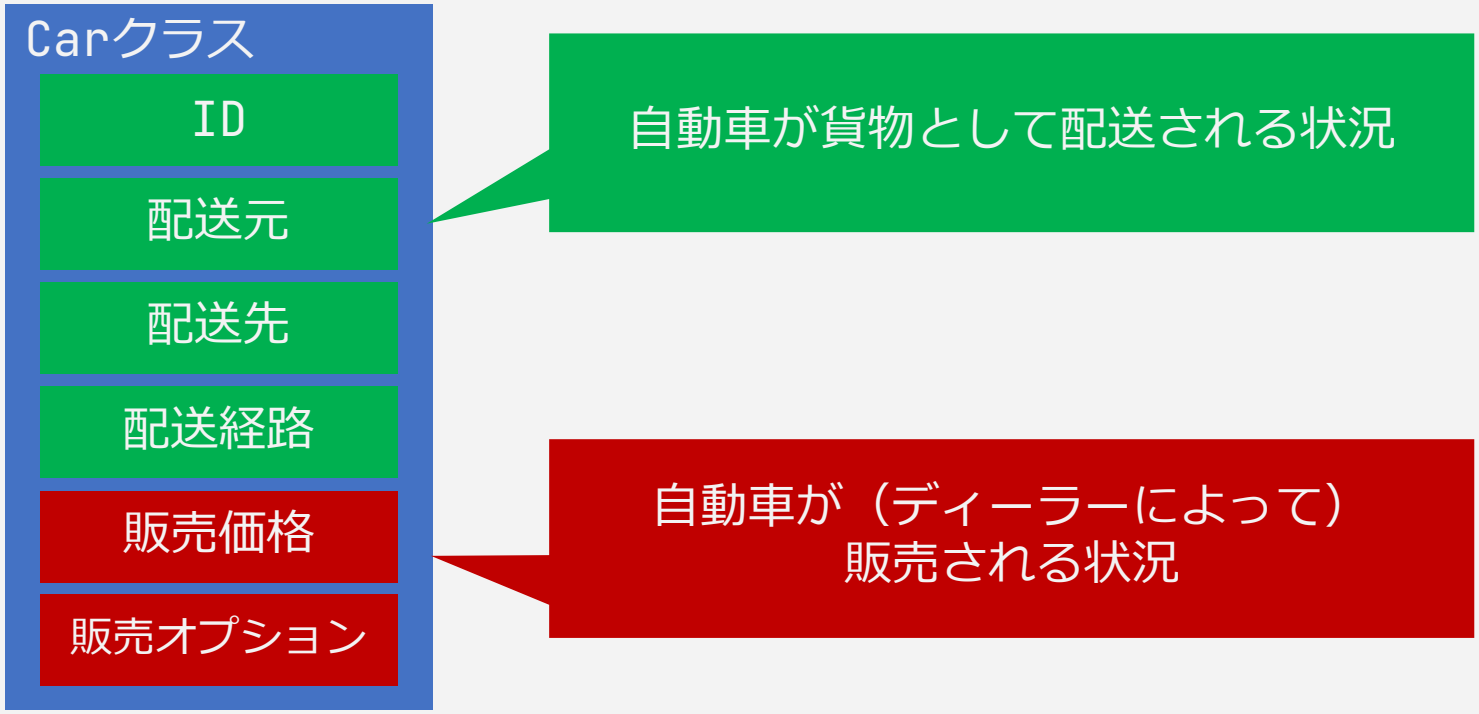
状況によって意味や扱いが異なる名前

◆例えば「アカウント」について考えてみる
- ■金融業界では「銀行口座」を意味する
- ■コンピュータセキュリティでは「ログイン権限」を意味する

◆以上のような状況により意味が異なる名前がある
- ■状況（コンテキスト）によって付いて回る概念が全く異なることがある

状況によって意味や扱いが異なる名前

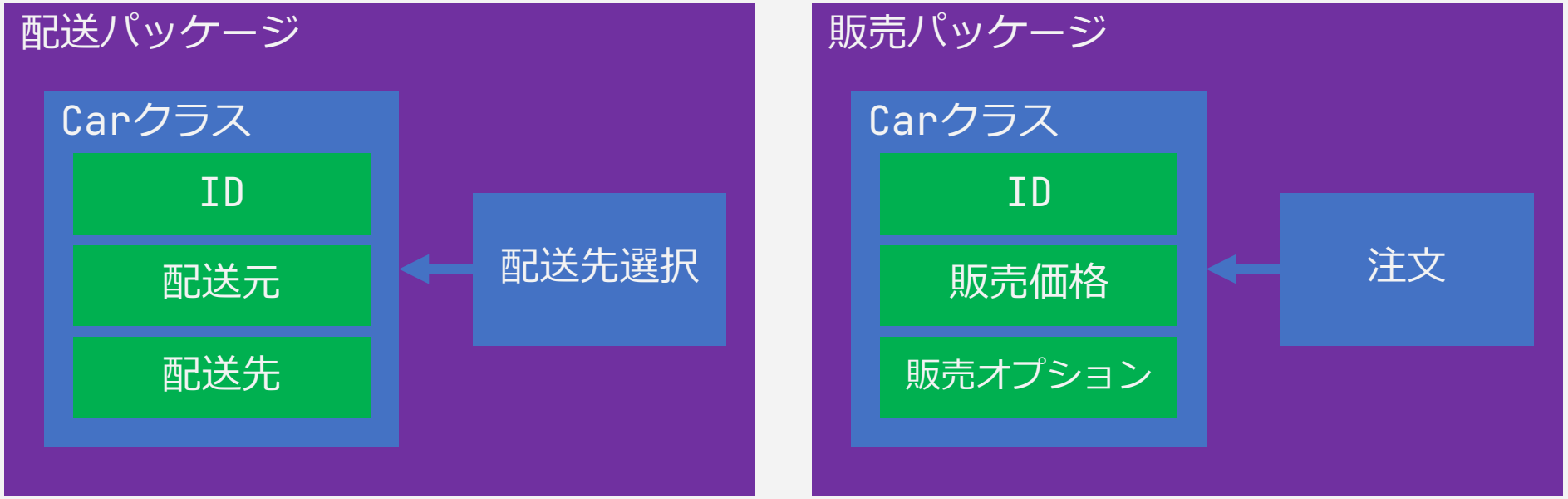
◆コンテキストが異なるものを一緒にしてしまうと…



◆クラスが巨大化する上に、開発者が混乱する

状況によって意味や扱いが異なる名前

◆各コンテキストはパッケージとして宣言する
■こうすることで変更影響が小さくなる



連番命名

◆連番命名は言わずもがな悪質

◆しかしながら組織的に管理されている場合もある
  ■こうなった場合は組織的な取り組みが必要

名前的に居場所が不自然なメソッド

「動詞+目的語」のメソッド名に注意

可能な限り動詞1語で済む名前にする

不適切な居場所のbooleanメソッド

名前の省略

意図が分からなくなる省略

基本的には名前は省略しないこと

そのほかの省略をどう判断するか

次の章に向けて