マイクロサービスとDDD

マイクロサービス：

モノリシックアプリとの対比で語られますが、これまでの一般的なアプリケーションは一つのプログラムに全機能を盛り込んだもので、いわばそのアプリケーションだけですべてをこなすものでした。

それに対しマイクロサービスは、機能を分割し、各機能ごとに小さなアプリケーションを作成し、それらを連携させて集合体としてのアプリケーションを作成するものです。

マイクロサービスのメリットは、各機能が疎結合となることで、それぞれにパラメータ引き渡すだけになり、機能ごとのメンテナンス・拡張・テストが行いやすくなることが挙げられます。

デメリットは、基本的に各機能ごとにDBやデータ構造を持つため、機能を横断するようなトランザクションスコープを作ることができないことです。そのため結果整合性が基本となります。

どちらもメリット・デメリットがあるので、案件によってどちらがいいかを考える必要があります。

DDD：

エリック・エヴァンスによって提唱されたアーキテクチャデザイン

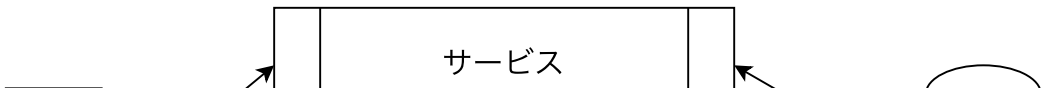
ドメイン領域(境界付けられたコンテキスト:bounded context)、ドメインモデル(Entity)、値オブジェクト(Value Object)、ユビキタス言語という用語がその後に大きな影響を与えた。
マイクロサービスとの親和性がとても高い。

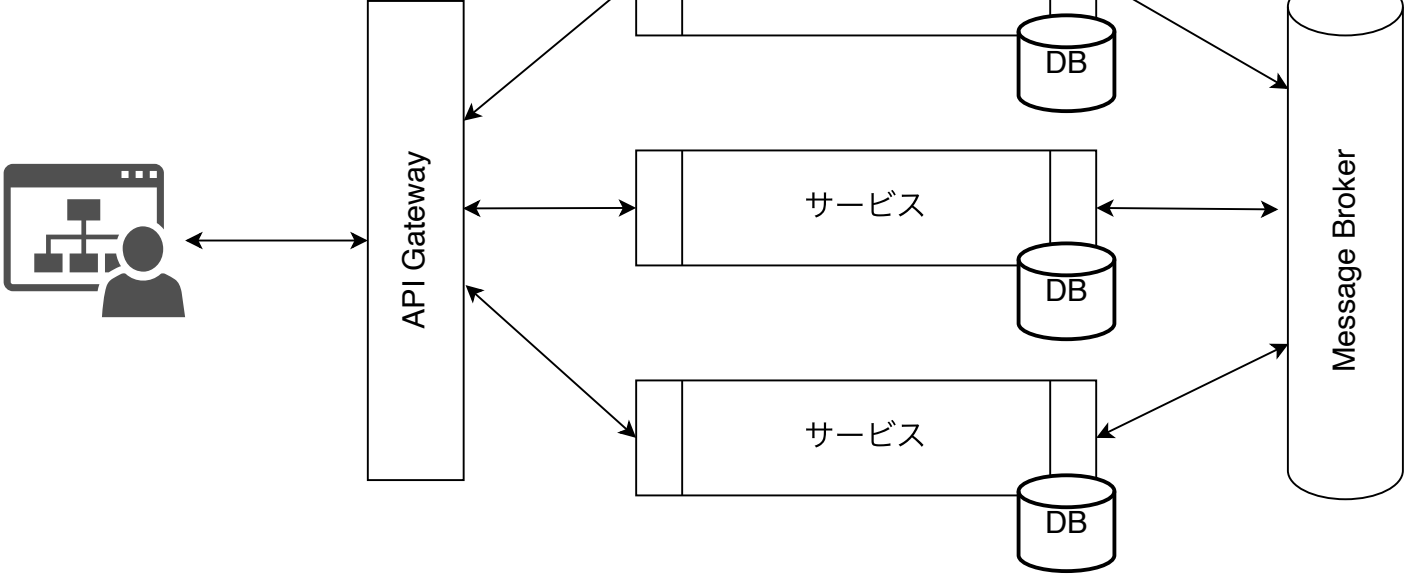
これまでの現実世界と乖離したシステム開発ではなく、現実世界の業務領域(ドメイン)とシステムが同じ言葉・同じルールで開発されることを目的とし、その中枢をドメインモデルが担うというものです。

難しく考えると分からなくなるので、これから実装をやってみながら、その一例に触れてもらえればと思います。

モノリシックアプリによる基本的な設計例

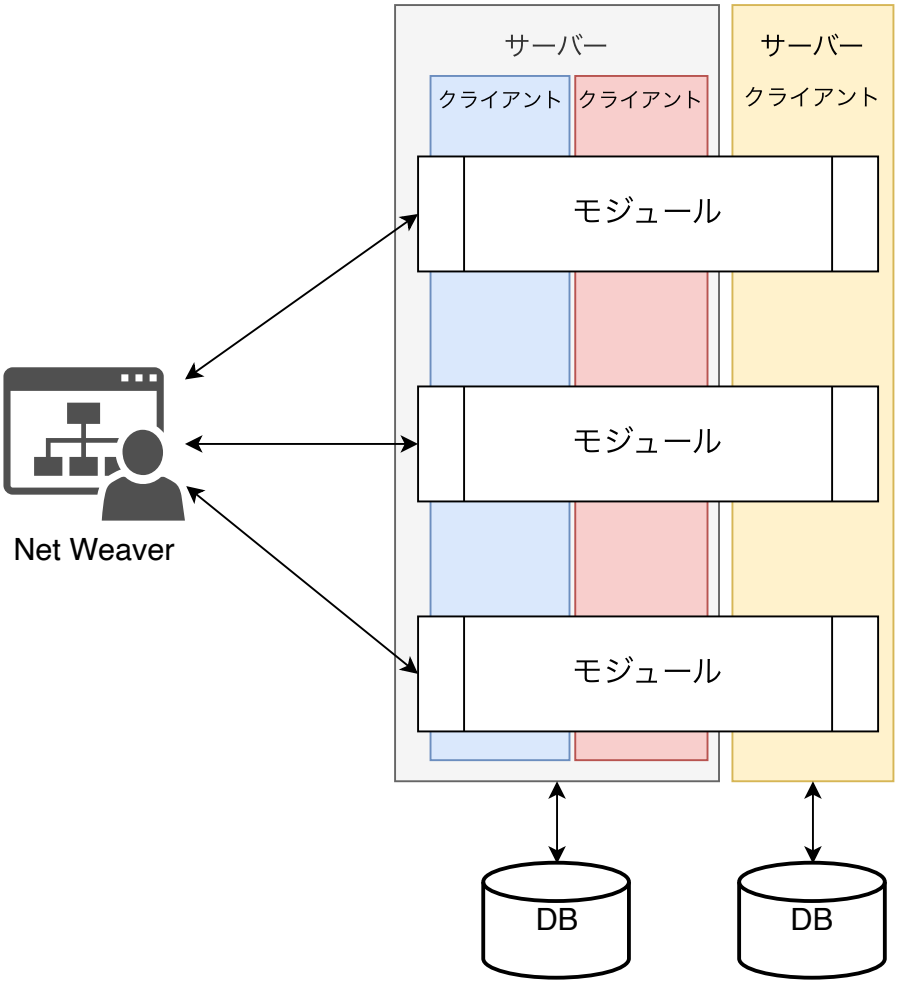
マイクロサービスによる基本的な設計例



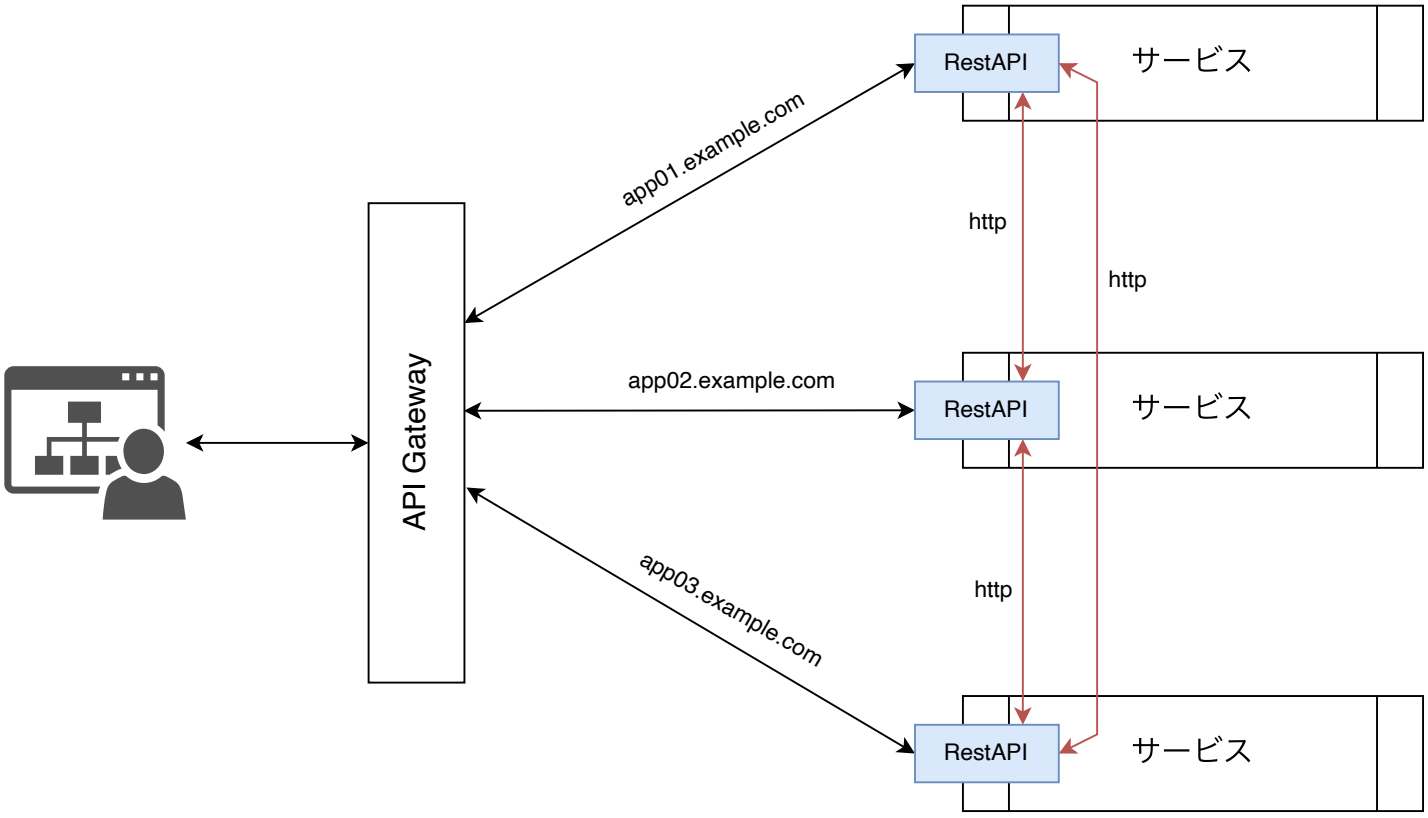


実装のしやすさから、各サービスをDockerに代表されるようなコンテナーを利用して開発されることが多い

参考) SAPの構造



画面(Net Weaver)から各分割されたモジュール(FI, CO, MM, SD, LE, PP, QM)を呼び出しサーバーごとのDBに格納しているイメージです
各サーバー・クライアントはすべてのモジュールを実装していて独立・並列処理されています
モジュール構成のモノリシックアプリを複数サーバーで連携しているイメージ



RestAPIを使用したパターン

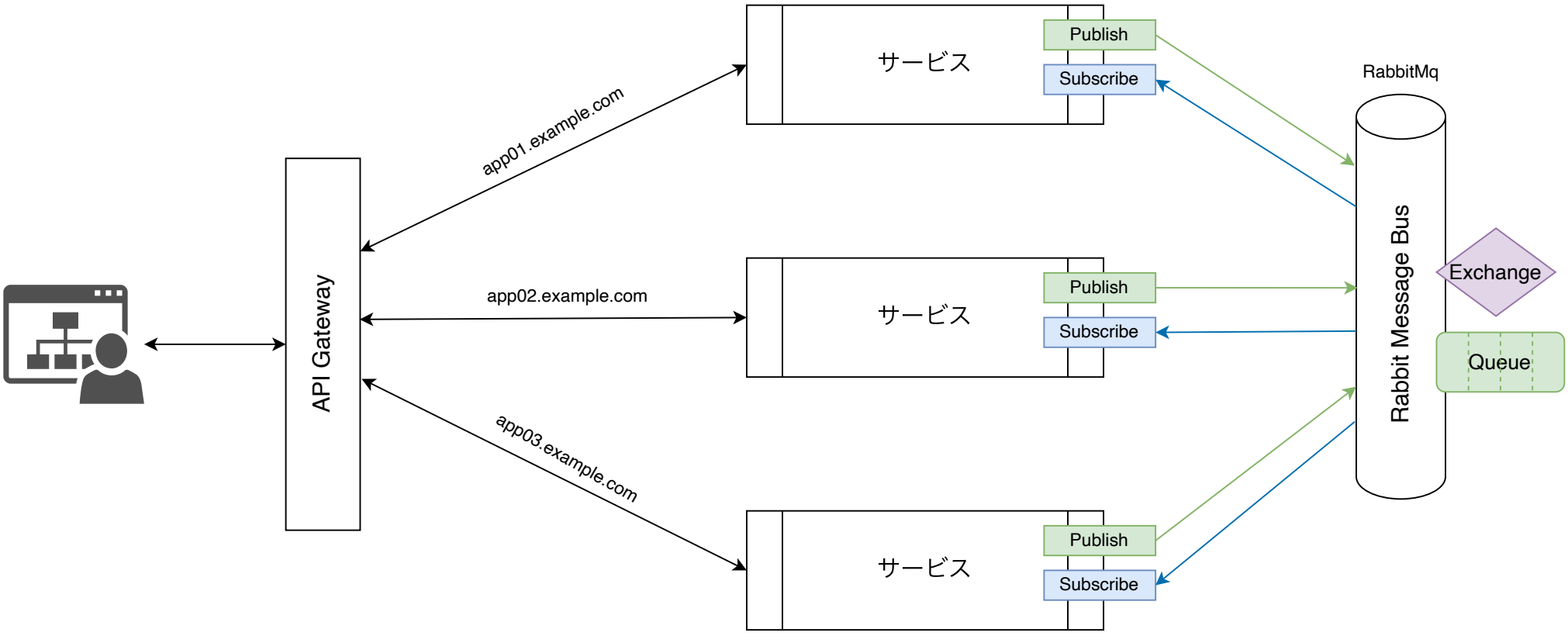
サービス同士のデータ連携、メッセージの送受信にRestAPIを使う

互いのHttpアドレスを知っていることが必要

Restのため、Request, Responseという双方向通信であり、Responseに対する処理を行う共依存性がある

Request先のアドレスを知っている必要があり、Request先に改修があると、送信元も改修する必要がある

基本的に同期通信(Response待機)



RabbitMqを使用したパターン

メッセージの送信はProducer (Publisher) が行う
メッセージの受信はConsumer (Subscriber) が行う

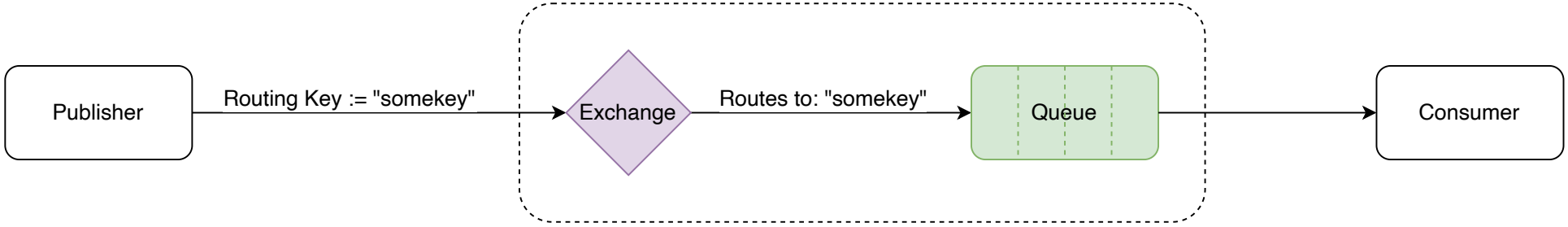
メッセージルールとしてRoutingKeyを指定しExchangeがルールに応じて
Queueに送られます。

送信側も受信側も相手を知らなくていい : 疎結合

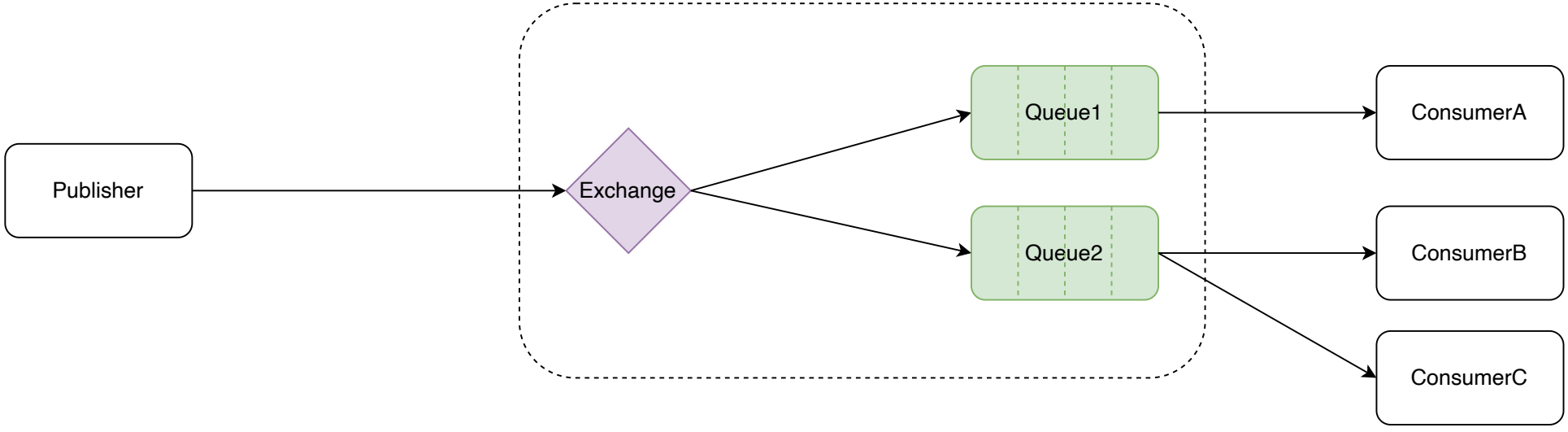
非同期通信 (AMQP: Advanced Message Queuing Protocol)

RabbitMQの送受信パターン

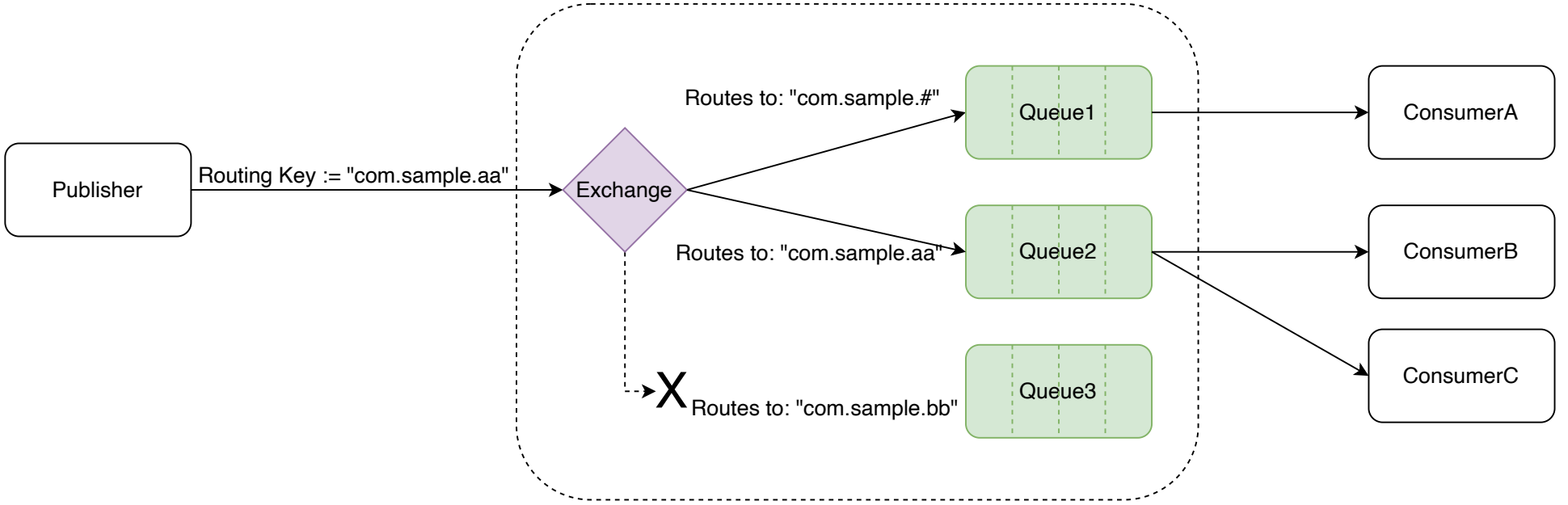
Direct Exchange : Routing Key に基づいてキューにメッセージを送信します



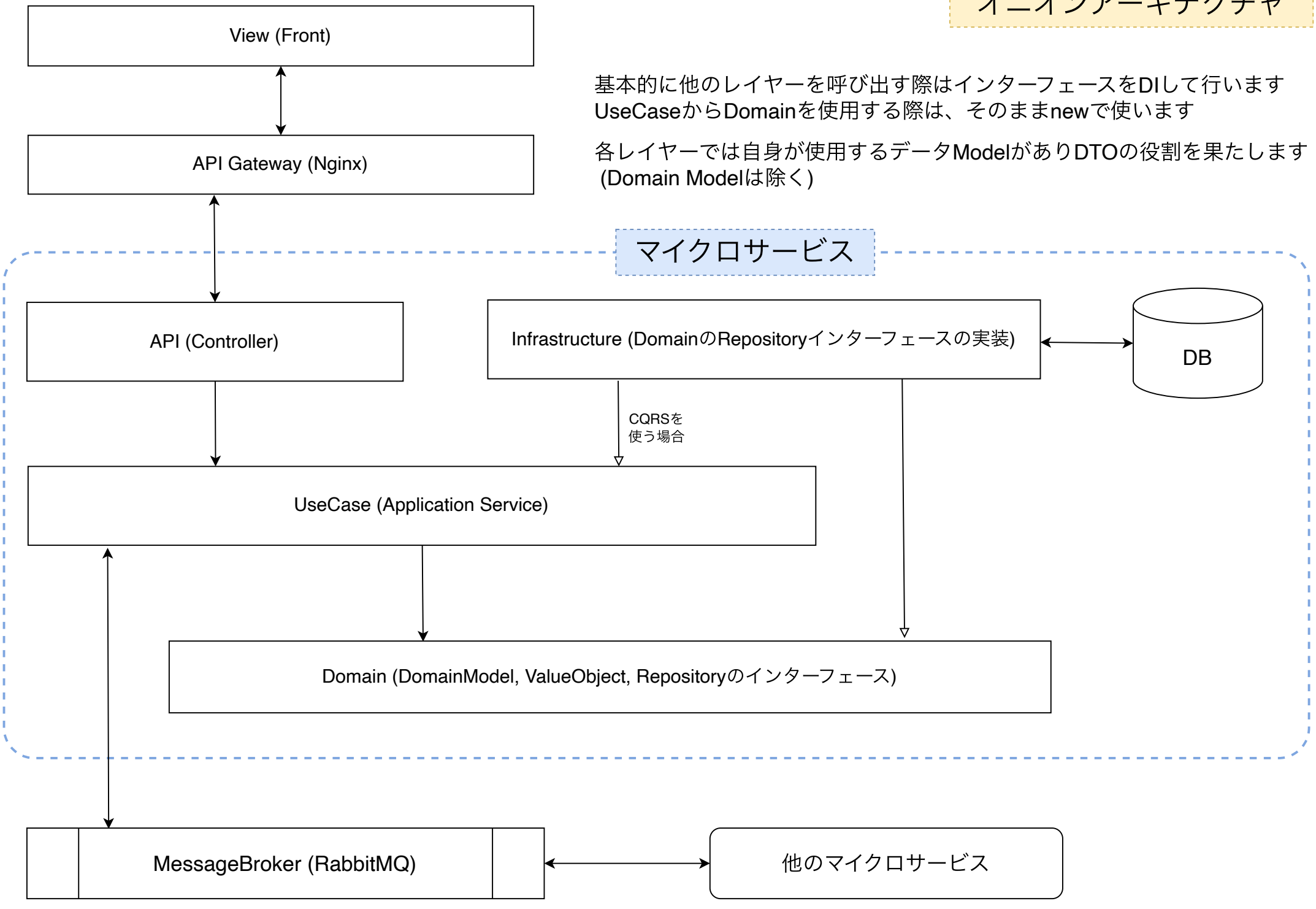
Fanout Exchange : Routing Keyの値は関係なくExchangeに結びついているすべてのキューにメッセージを送信します



Topic Exchange : Routing Keyのパターンに応じ、それに一致するキューにメッセージを送信します



オニオンアーキテクチャ



要件

A社は自社の製品を販売するためにECサイトを立ち上げることにしました。

この製品はすでに地元で人気を博していて、ECサイトによってさらに販路の展開を図りたいとA社は考えています。

販売のためのECサイトは自社製品を表示して、お客さんに購入してもらうという一般的なサイトです。

必要な機能としては、

- 商品情報(名前・価格など)と在庫の表示
- 商品カートと購入機能
- 購入を受注して発送すること
- 在庫状況から自社工場に発注し納品する
- 日次と月次で売上を集計したい

が主なものです。

また実装にあたり以下の条件を付けてほしい

- 配送料は一律500円とするが、商品の購入価格が合計5,000円以上の場合は無料とする(A社負担)
- 商品ごとに1人(1カート)で購入できる数を限定する
- カートに入れられるのは10品目まで
- 1回あたり5万円以上の購入はできない
- 在庫が残り3以下となったら発注処理を行う
    - これは日次処理で行うものとする
- 商品には、販売価格と仕入れ価格がある
- A社側の処理(発注・仕入れ)は自動的に行われる
- 日次・月次はジョブとして定期的に自動的に行われるが、手動でも行える
    - ただしジョブスケジューラがあるという前提のみ
    - プログラミングとしては手動の部分を作成するのみでOK

※今回は複雑にならないように以下の機能は実装しないか、仮のデータで行います

- アカウント作成やログイン機能
    - アカウントやユーザーの住所等はデモデータで行います
- ユーザーの支払い方法選択や入金確認
    - 購入ボタンだけでOKとする
- 価格等は消費税込みとして計算はしない

境界付けられたコンテキストに分けるとは

現実社会の業務＝ドメイン領域

このドメインを分けることから始めます

つまり、A社の業務を分割するということです

たとえば会社では通常、営業部門・経理部門・製造部門といったように部署が分かれていると思います。そのように分けることで業務ごとに責任を明確にして、その業務をスムーズにできるようにするためです。

システム開発でも同じように分割して、各責務を分離すればいいのではないか
これがDDDの考え方です。

手を付け始めるのはシステムからではなく業務から

A社の要件を業務として分けてみます

この分ける作業は本当にプロジェクトとして行う場合は、まず大きく分けて、そこから少しずつ小さくしていきます。
最初から細かくしてしまうと、実装時に実は一緒だったとかいうことが起きたりします。
一度作ったものを二つ合わせるよりは、作ったものを分割するほうが責務分離とコードの保守性からしてもやりやすいです。

さて、機能と条件についてもう一度見てましょう

機能

- 商品情報(名前・価格など)と在庫の表示
- 商品カートと購入機能
- 購入を受注して発送すること
- 在庫状況から自社工場に発注し納品する
- 日次と月次で売上を集計したい

条件

- 配送料は一律500円とするが、商品の購入価格が合計5000円以上の場合は無料とする(A社負担)
- 商品ごとに1人(1カート)で購入できる数を限定する
- カートに入れられるのは10品目まで
- 1回あたり5万円以上の購入はできない
- 在庫が残り3以下となったら発注処理を行う
    - これは日次処理で行うものとする
- 商品には、販売価格と仕入れ価格がある
- A社側の処理(発注・仕入れ)は自動的に行われる
- 日次・月次はジョブとして定期的に自動的に行われるが、手動でも行える

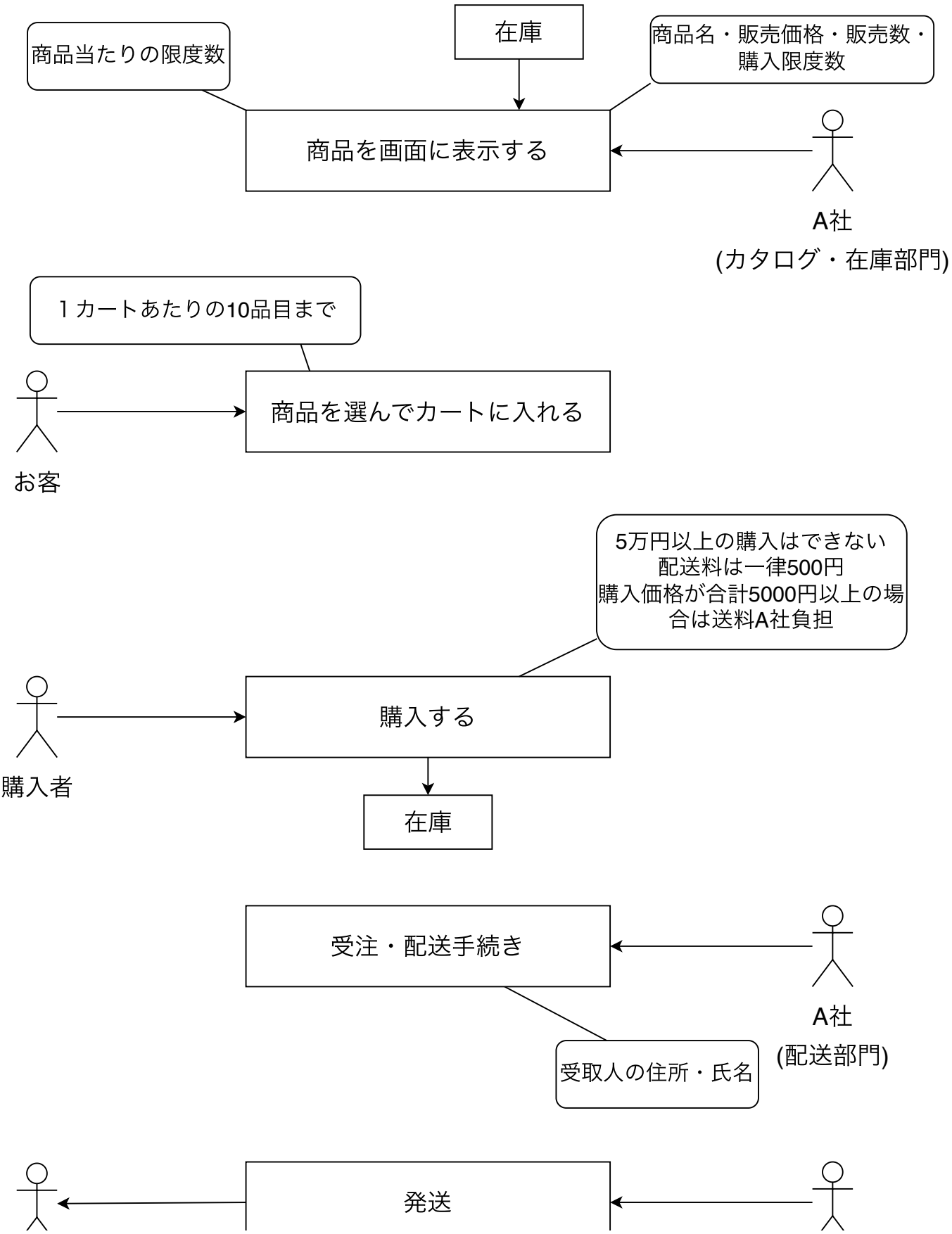
業務フロー(ユースケース図)をまず作成します
このとき重要なのが、各業務ごとで同じ人物が別の呼び方になるということです。

どういうことかと言えば、ユーザーといっても、購入するときは「購入者」と呼び、商品を受け取るときは「受取人」となるということです。

この業務ごとで役割が変わるのであれば、システム開発でも同じように分けることをDDDでは重要視しています

なぜなら、責務分離された各ドメイン領域は、それぞれの責務で開発を行い、他の領域の責務は知らなくていいという原則があるからです。(疎結合)

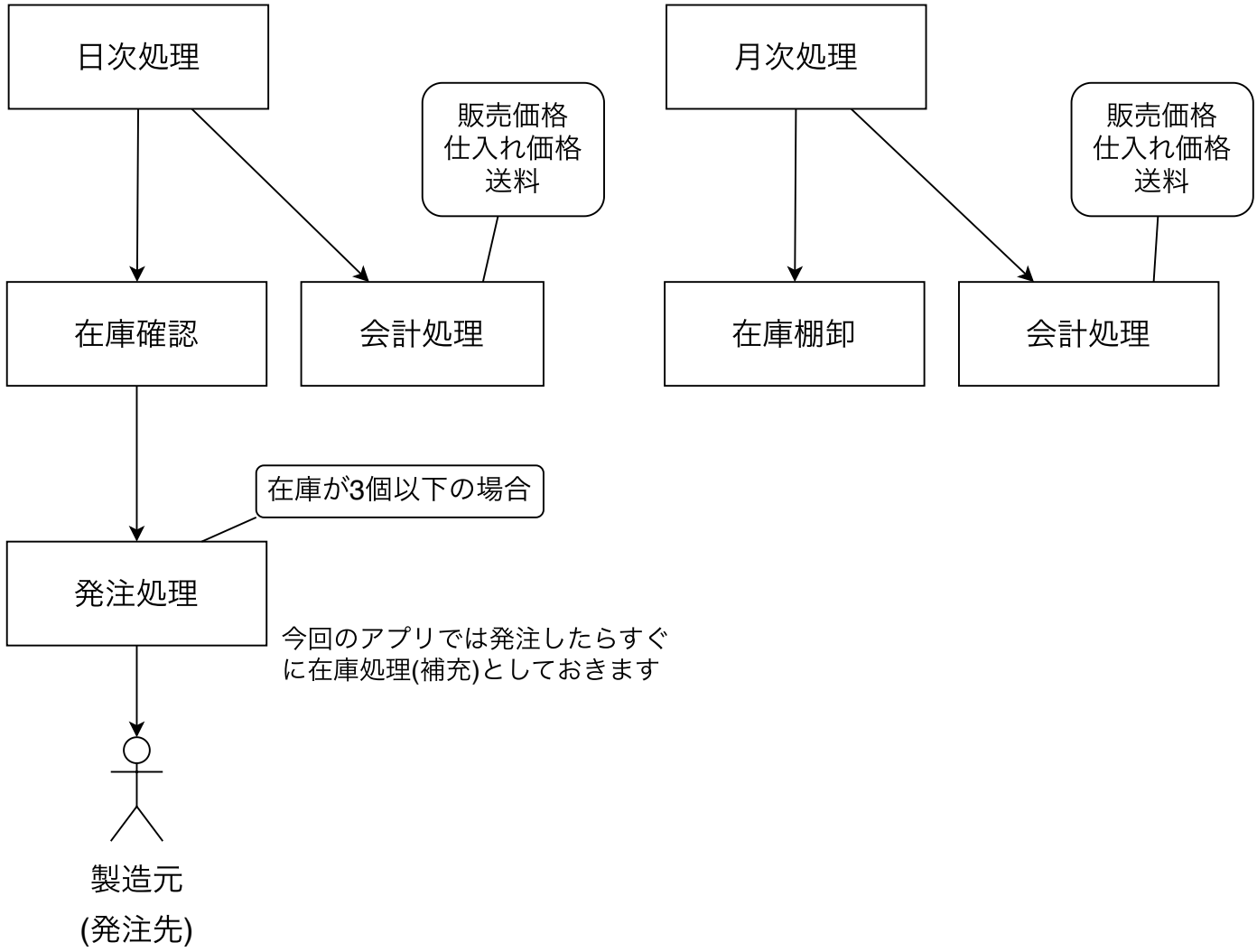
A社から説明された業務の流れ (ユースケース図)



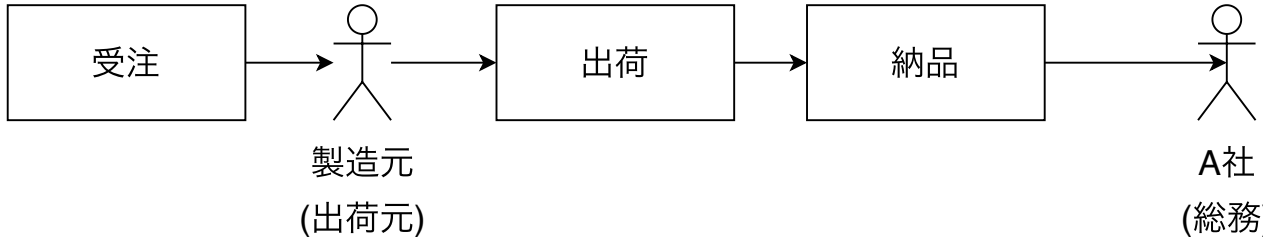
受取人

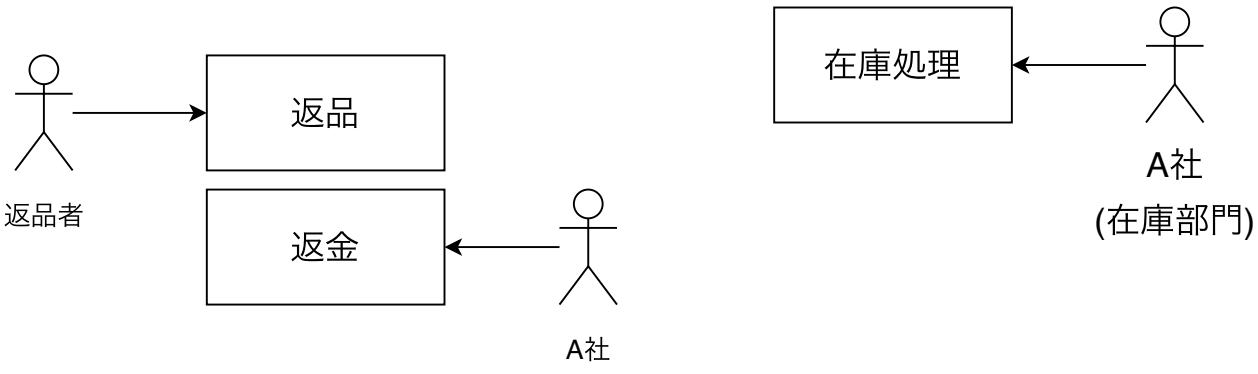
A社
(配送部門)
(もしくは配送業者)

---ジョブ---



今回のアプリでは実装対象外ですが、現実社会では以下のフローもあります

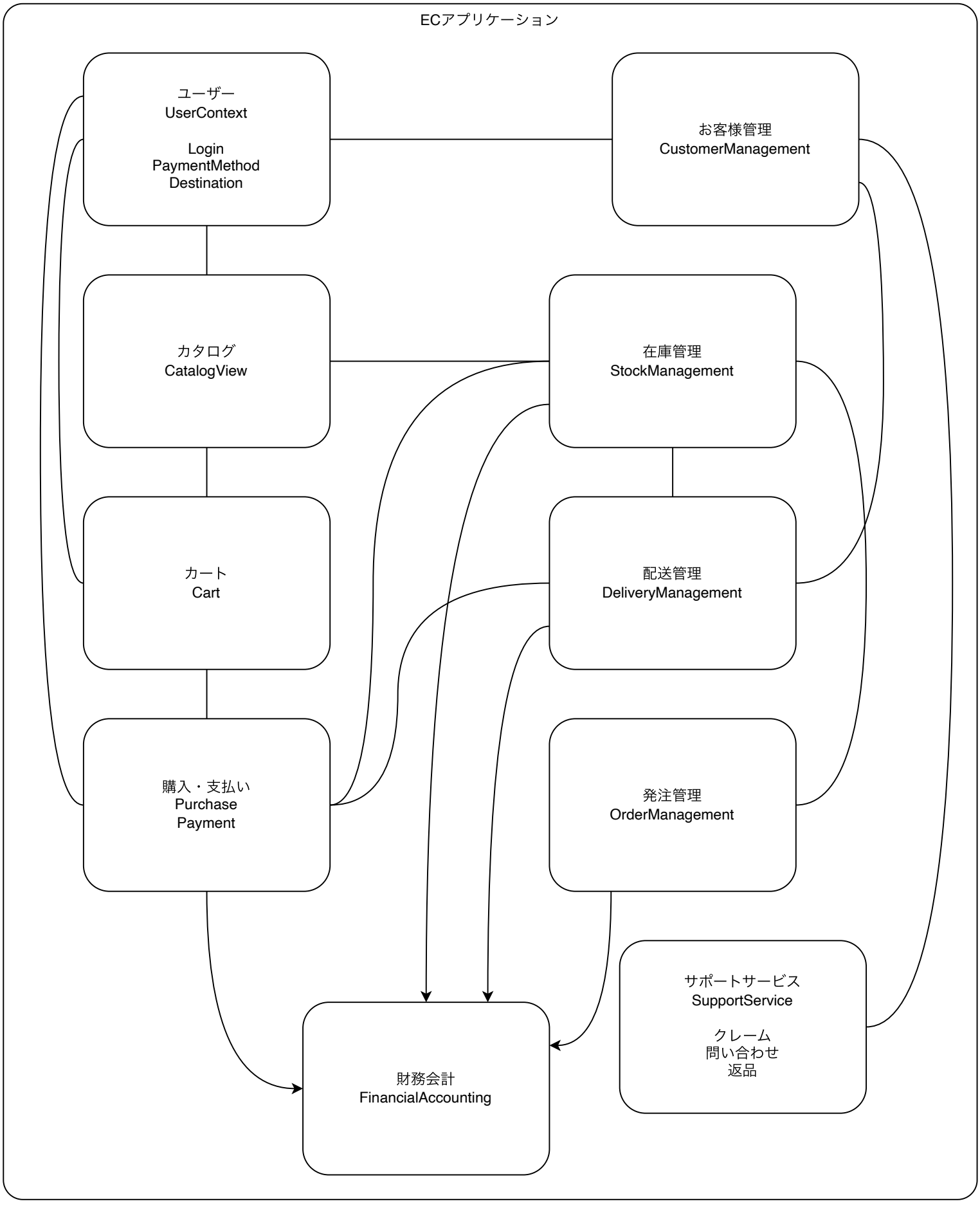




A社からの要件を簡単なフローにしましたが、不足しているものがたくさんあります
そのため次はドメインエキスパートとの会話で業務・ユースケースを十分なものにしていきます

コンテキストマッピング

境界づけられたコンテキストの間の関連を示すものです
A社から説明された業務を境界づけられたコンテキストに分け、現時点での関連例を示します
これで完成というわけではなく、DDDでは常に業務の変更等に応じてダイナミックにコンテキストもドメインモデルも変更されていきます。



ドメインエキスパートと確認した事項

Q：在庫・カタログについて

ここで使用する在庫は、在庫発注で補充されますか？
　はい、在庫発注→納品→在庫補充となります

画面に表示される販売数は在庫数ですか？それとも購入限度数ですか？
　購入限度数が表示されます
　在庫のほうが少ない場合は在庫数です

在庫・カタログの新規追加や変更・調整はどのように行いますか？
またその際のルールはありますか？
　管理用画面があり、以下の処理を行います
　在庫商品管理
　　新規処理 : 商品の新規追加
　　変更処理 : 商品の名前の変更・在庫数・仕入れ価格
　　削除処理 : 商品の削除
　ルールとしては以下のものがあります
　　新規・変更 : 販売用商品名は20文字以内(40byte)、価格は平均仕入れ価格の1.5倍以上3倍以内
　　　　購入限度数は1以上5以下
　カタログでは
　　新規 : 表示用商品名・販売価格・購入限度数の登録
　　変更 : 表示用商品名・販売価格・購入限度数
　　削除 : 過去に販売実績のあるものは削除できません

商品名は変わることがありますか？また仕入れ価格が変動することはありますか？
　はい、商品名も仕入れ価格も変わることがあります
　商品名が変わってもカタログとして表示する名前は販売用商品名を使用します
　販売価格は平均仕入れ価格(小数点以下は四捨五入)の1.5倍から3倍以内ということになります
　平均仕入れ価格の計算は在庫のある商品から行います

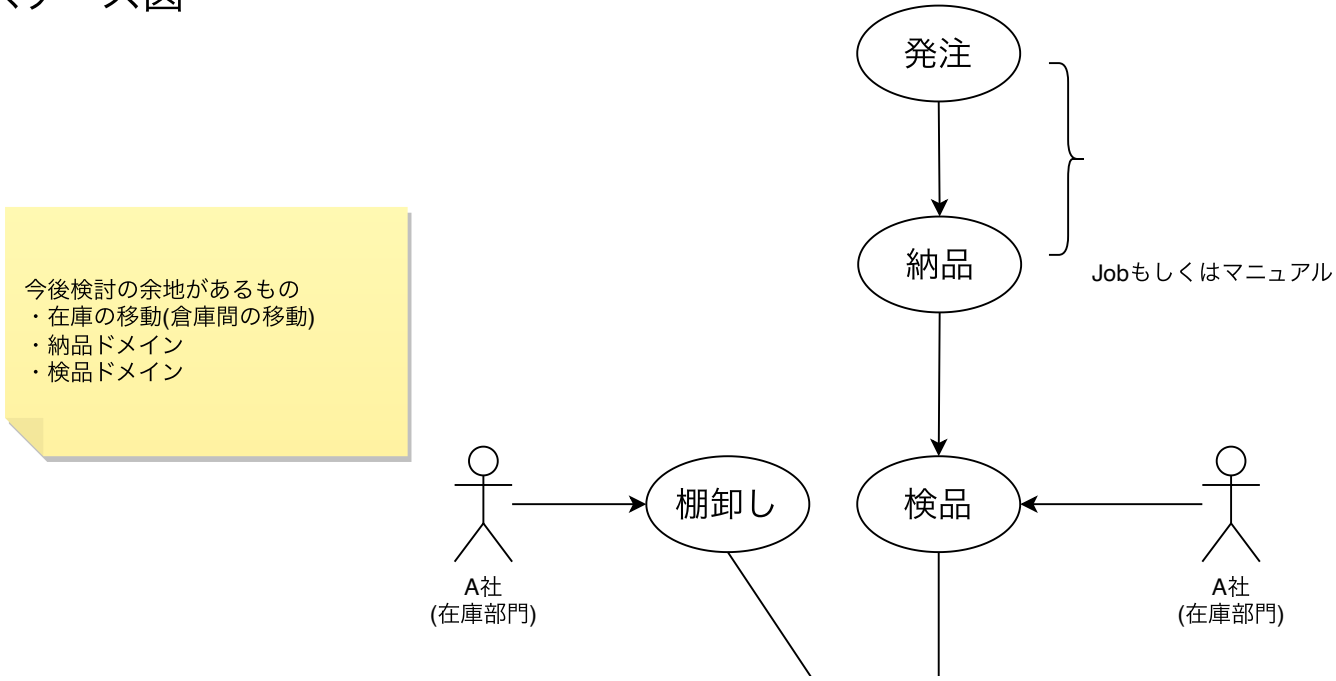
在庫調整等はありますか？
　はい、仕入れ処理ミス、検品による不良品による在庫追加・削除があります

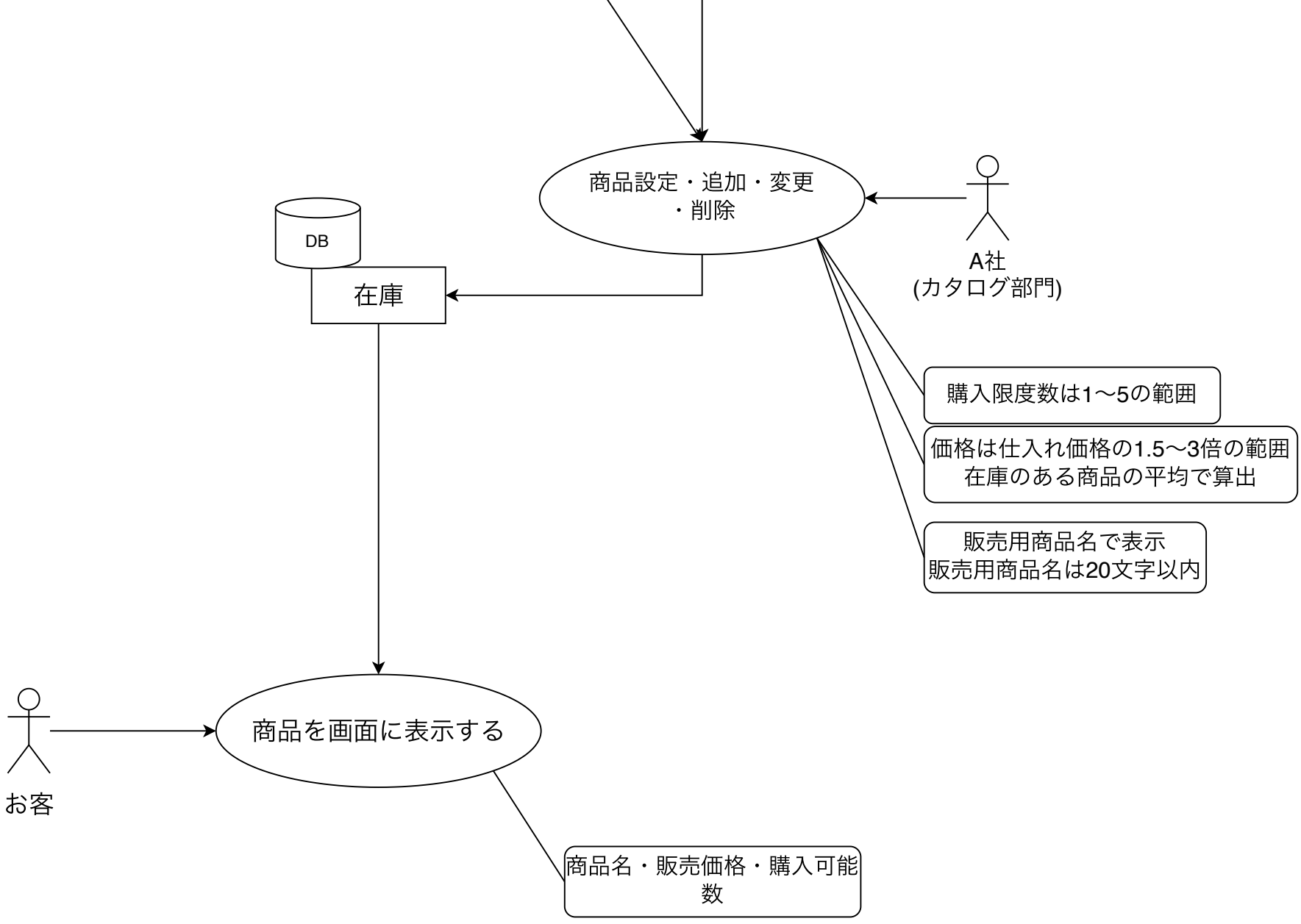
在庫調整したで商品数を変更(追加・変更・削除)した場合、仕入れとの差異はどのように処理しますか
　商品数の調整は棚卸しで実在庫と差異があった場合の処理となります。
　棚卸しは仕入れからの在庫量と実在庫との比較で行われます。
　この在庫処理での調整は、棚卸し処理後の差異を調整するためのものですので仕入れとの調整は不要です。
　　→つまり在庫ドメインよりも仕入れ・納品・検品ドメイン領域の責務と本来はなるということ
　仕入れ後の商品管理に起因する場合(破損・紛失・発見等)も仕入れ後のものですので仕入れとの調整は不要です。
　この場合は削除分の仕入れ価格を損金として計上し、追加の場合は復活在庫として仕入れ価格は平均仕入れ価格を四捨五入して計上します
　在庫数は仕入れ処理での仕入れ数が元になりますので発注との差異は処理不要です。
　帳簿上在庫＝期首在庫 + 仕入れ数 - 売上数

カタログ表示で在庫のない商品は表示しますか？
　発注・納品予定のない商品は表示しないでください

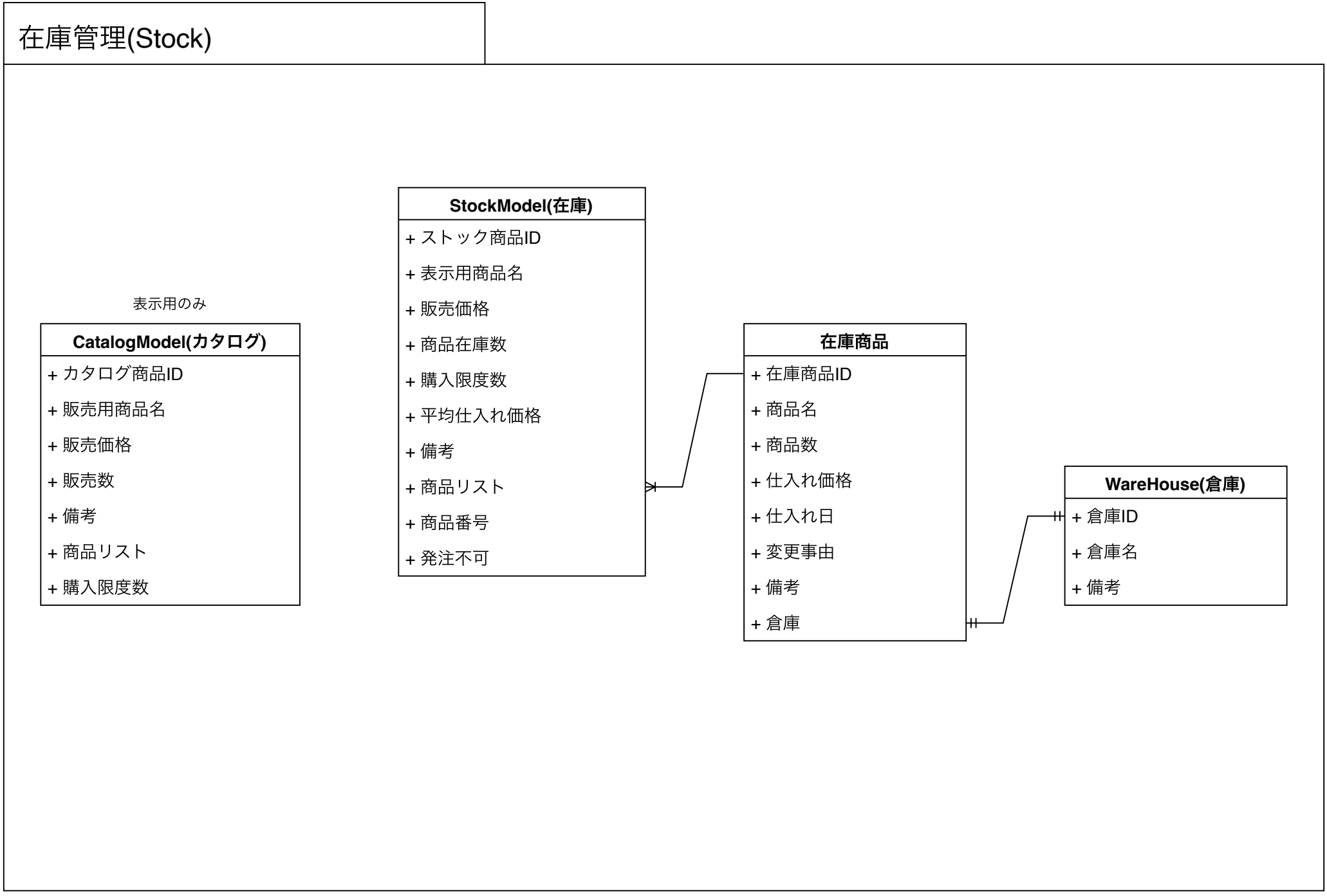
新規の商品情報を登録することはできますか？
　商品情報の登録をして、商品は発注・納品されることで登録されます

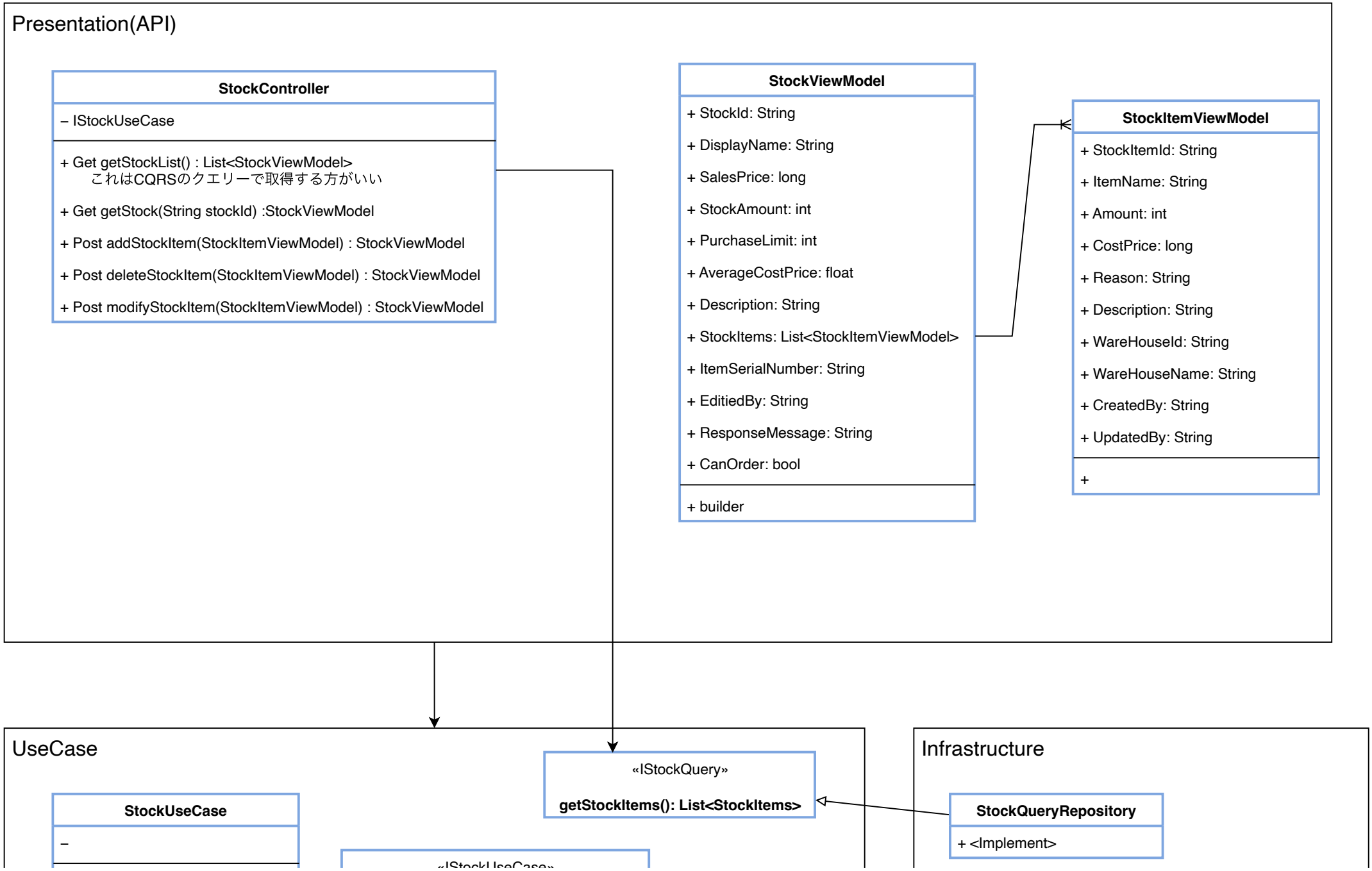
ユースケース図

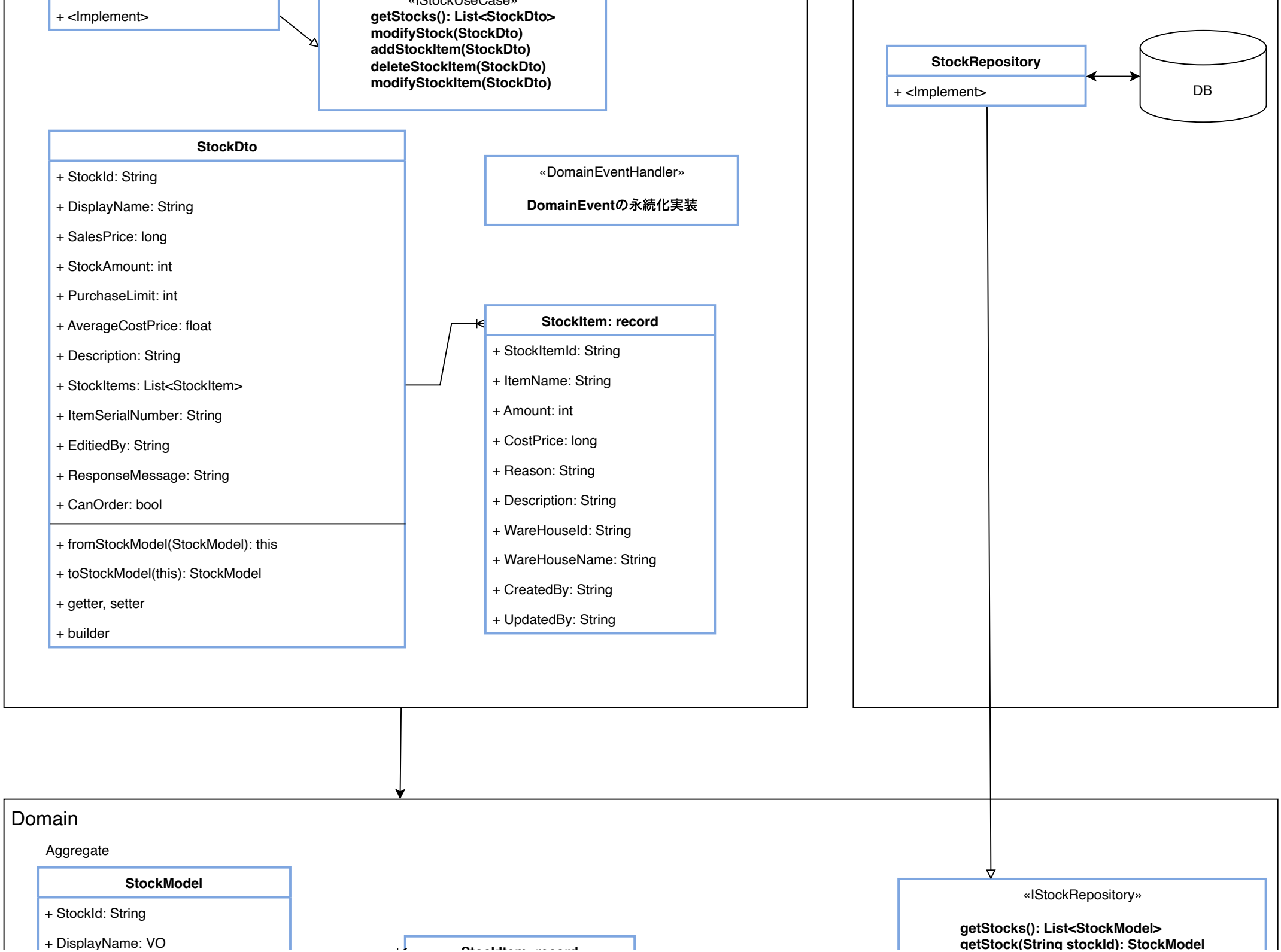


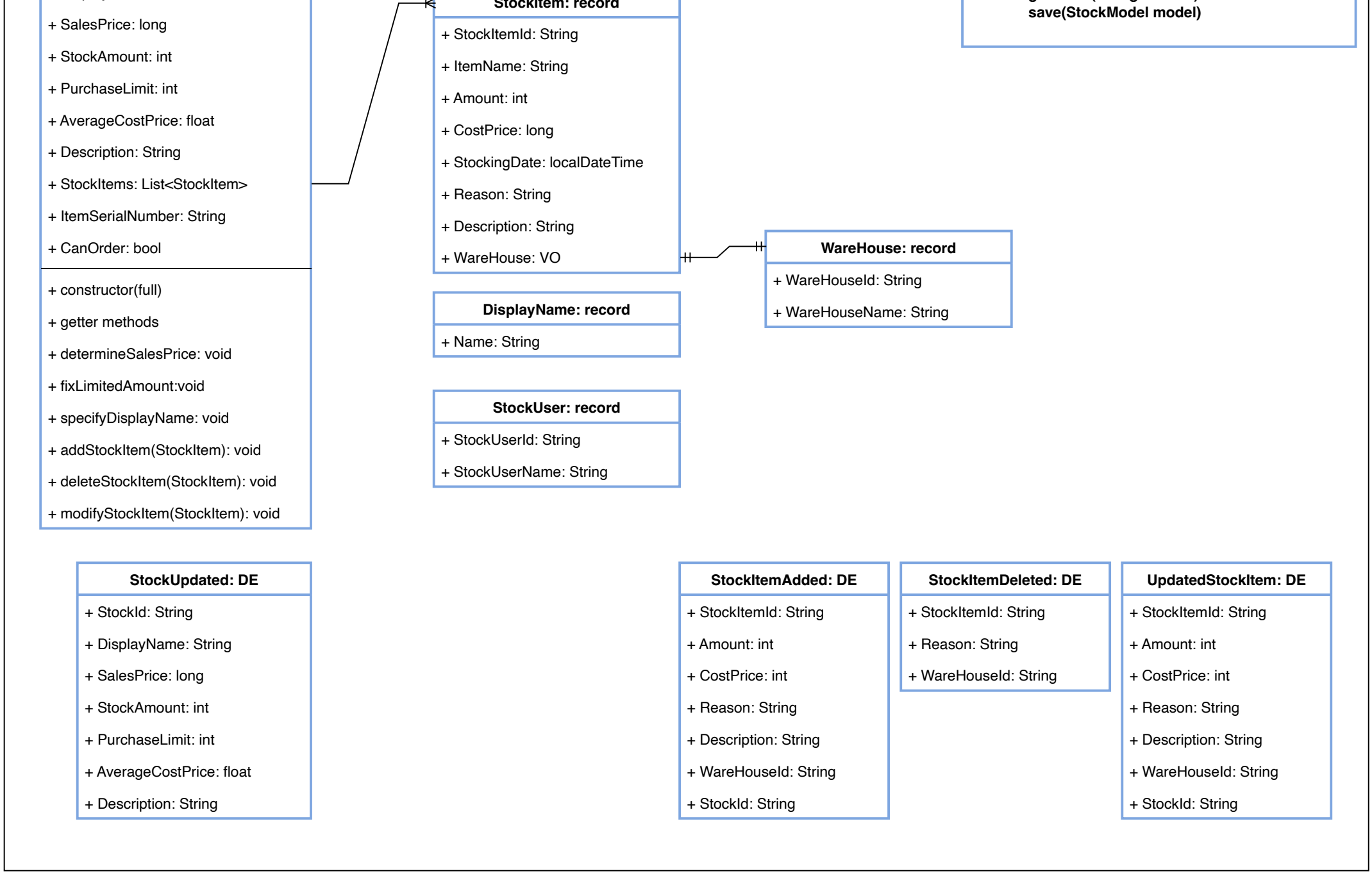

商品設定・追加・変更
・削除
A社
(カタログ部門)
DB
在庫
購入限度数は1〜5の範囲
価格は仕入れ価格の1.5〜3倍の範囲
在庫のある商品の平均で算出
販売用商品名で表示
販売用商品名は20文字以内
お客
商品を画面に表示する
商品名・販売価格・購入可能
数

ドメインモデル


在庫管理(Stock)
表示用のみ
CatalogModel(カタログ)
+ カタログ商品ID
+ 販売用商品名
+ 販売価格
+ 販売数
+ 備考
+ 商品リスト
+ 購入限度数
StockModel(在庫)
+ ストック商品ID
+ 表示用商品名
+ 販売価格
+ 商品在庫数
+ 購入限度数
+ 平均仕入れ価格
+ 備考
+ 商品リスト
+ 商品番号
+ 発注不可
在庫商品
+ 在庫商品ID
+ 商品名
+ 商品数
+ 仕入れ価格
+ 仕入れ日
+ 変更事由
+ 備考
+ 倉庫
WareHouse(倉庫)
+ 倉庫ID
+ 倉庫名
+ 備考

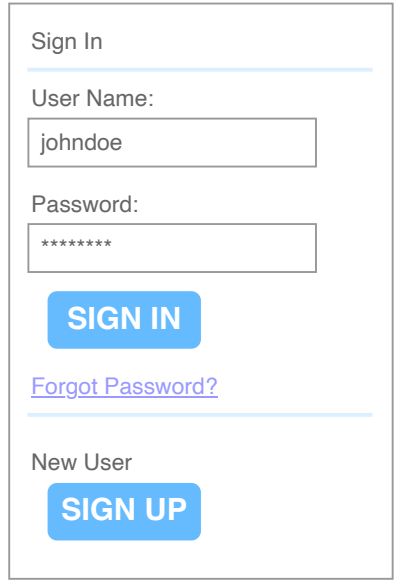

Presentation(API)
StockController
– IStockUseCase
+ Get getStockList() : List<StockViewModel>
これはCQRSのクエリーで取得する方がいい
+ Get getStock(String stockId) :StockViewModel
+ Post addStockItem(StockItemViewModel) : StockViewModel
+ Post deleteStockItem(StockItemViewModel) : StockViewModel
+ Post modifyStockItem(StockItemViewModel) : StockViewModel
StockViewModel
+ StockId: String
+ DisplayName: String
+ SalesPrice: long
+ StockAmount: int
+ PurchaseLimit: int
+ AverageCostPrice: float
+ Description: String
+ StockItems: List<StockItemViewModel>
+ ItemSerialNumber: String
+ EditiedBy: String
+ ResponseMessage: String
+ CanOrder: bool
+ builder
StockItemViewModel
+ StockItemId: String
+ ItemName: String
+ Amount: int
+ CostPrice: long
+ Reason: String
+ Description: String
+ WareHouseId: String
+ WareHouseName: String
+ CreatedBy: String
+ UpdatedBy: String
+
UseCase
«IStockQuery»
getStockItems(): List<StockItems>
StockUseCase
–
«IStockUseCase»
Infrastructure
StockQueryRepository
+ <Implement>


+ <Implement>
«IStockUseCase»
getStocks(): List<StockDto>
modifyStock(StockDto)
addStockItem(StockDto)
deleteStockItem(StockDto)
modifyStockItem(StockDto)
StockRepository
+ <Implement>
DB
StockDto
+ StockId: String
+ DisplayName: String
+ SalesPrice: long
+ StockAmount: int
+ PurchaseLimit: int
+ AverageCostPrice: float
+ Description: String
+ StockItems: List<StockItem>
+ ItemSerialNumber: String
+ EditiedBy: String
+ ResponseMessage: String
+ CanOrder: bool
+ fromStockModel(StockModel): this
+ toStockModel(this): StockModel
+ getter, setter
+ builder
«DomainEventHandler»
DomainEventの永続化実装
StockItem: record
+ StockItemId: String
+ ItemName: String
+ Amount: int
+ CostPrice: long
+ Reason: String
+ Description: String
+ WareHouseId: String
+ WareHouseName: String
+ CreatedBy: String
+ UpdatedBy: String
Domain
Aggregate
StockModel
+ StockId: String
+ DisplayName: VO
«IStockRepository»
getStocks(): List<StockModel>
getStock(String stockId): StockModel

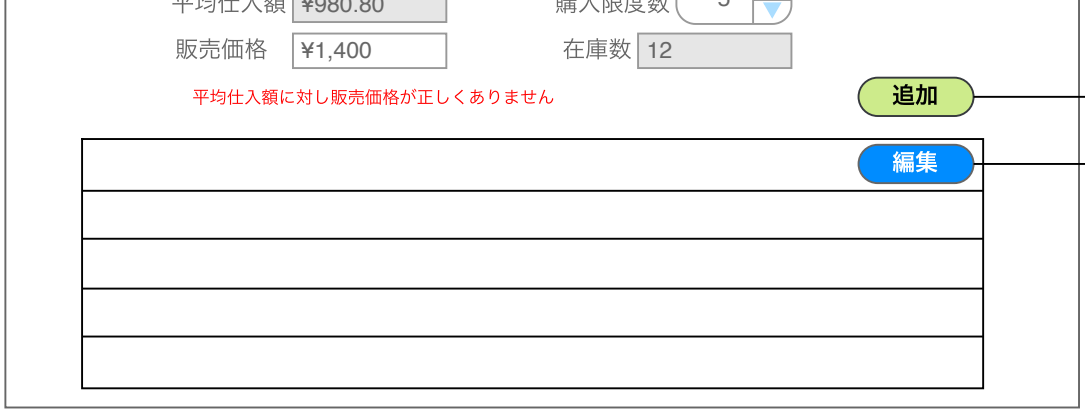

save(StockModel model)
StockItem: record
+ StockItemId: String
+ ItemName: String
+ Amount: int
+ CostPrice: long
+ StockingDate: localDateTime
+ Reason: String
+ Description: String
+ WareHouse: VO
+ SalesPrice: long
+ StockAmount: int
+ PurchaseLimit: int
+ AverageCostPrice: float
+ Description: String
+ StockItems: List<StockItem>
+ ItemSerialNumber: String
+ CanOrder: bool
+ constructor(full)
+ getter methods
+ determineSalesPrice: void
+ fixLimitedAmount:void
+ specifyDisplayName: void
+ addStockItem(StockItem): void
+ deleteStockItem(StockItem): void
+ modifyStockItem(StockItem): void
WareHouse: record
+ WareHouseId: String
+ WareHouseName: String
DisplayName: record
+ Name: String
StockUser: record
+ StockUserId: String
+ StockUserName: String
StockUpdated: DE
+ StockId: String
+ DisplayName: String
+ SalesPrice: long
+ StockAmount: int
+ PurchaseLimit: int
+ AverageCostPrice: float
+ Description: String
StockItemAdded: DE
+ StockItemId: String
+ Amount: int
+ CostPrice: int
+ Reason: String
+ Description: String
+ WareHouseId: String
+ StockId: String
StockItemDeleted: DE
+ StockItemId: String
+ Reason: String
+ WareHouseId: String
UpdatedStockItem: DE
+ StockItemId: String
+ Amount: int
+ CostPrice: int
+ Reason: String
+ Description: String
+ WareHouseId: String
+ StockId: String

商品一覧

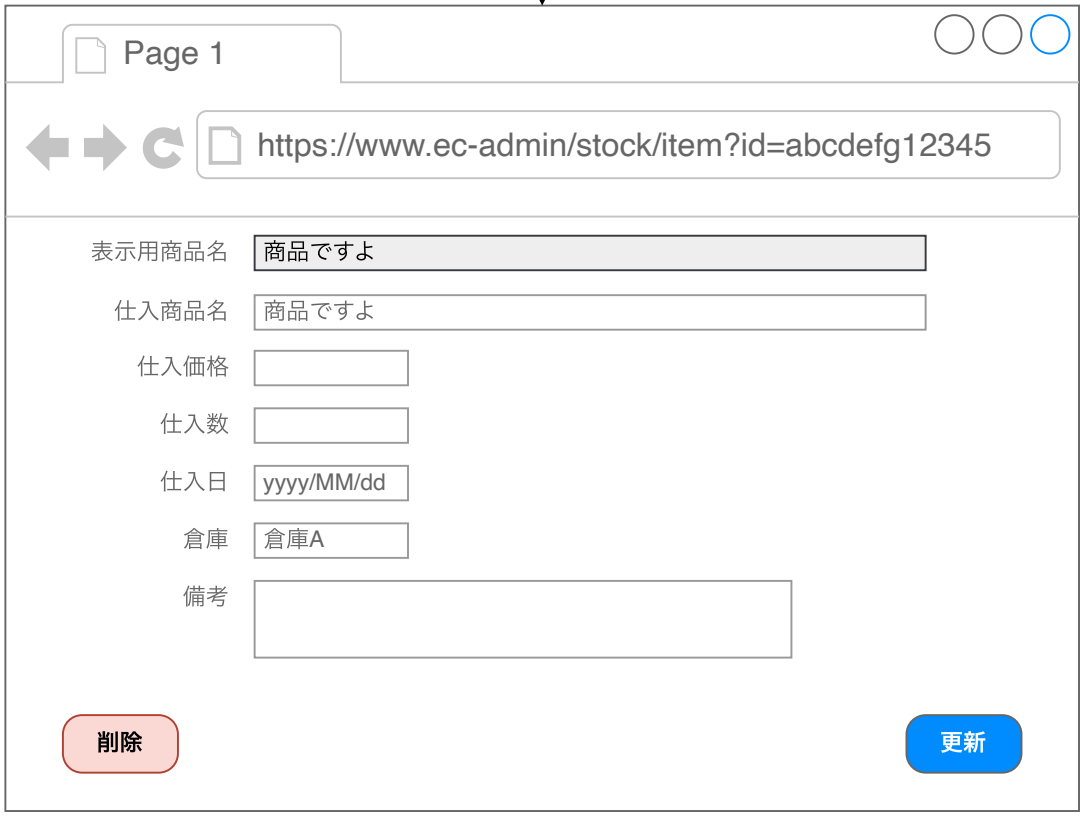
未実装・未設計ですが実際は必要



選択商品在庫等一覧


平均仕入額
¥980.80
購入限度数
販売価格
¥1,400
在庫数
12
平均仕入額に対し販売価格が正しくありません
追加
編集

選択商品詳細


Page 1
https://www.ec-admin/stock/item?id=abcdefg12345
表示用商品名
商品ですよ
仕入商品名
商品ですよ
仕入価格
仕入数
仕入日
yyyy/MM/dd
倉庫
倉庫A
備考
削除
更新


確認
削除しますか？
Cancel
OK


確認
更新しますか？
Cancel
OK

Q：カートについて

カートを使用できるお客は登録ユーザーですか？それとも未登録ユーザーも利用可能ですか？
　未登録ユーザーでも可能ですが、購入時にユーザー登録が必要となります
　登録していないと配送先が分からないので最終的に登録は必須です

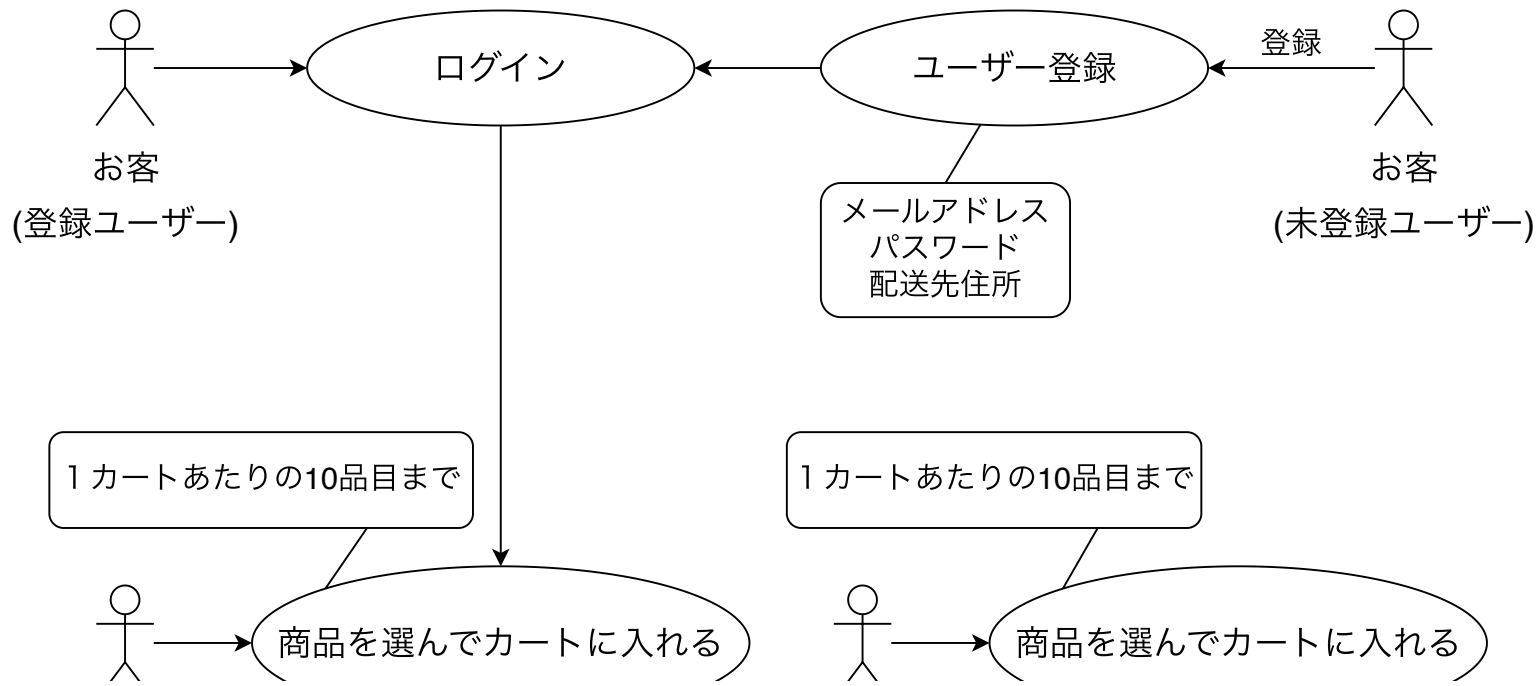
登録ユーザーはログインを行いますか？ログインに必要なものは何ですか？
　ログイン機能が必要です
　必要なものはメールアドレスとパスワードです
　ユーザーの識別にはメールアドレスを使用します

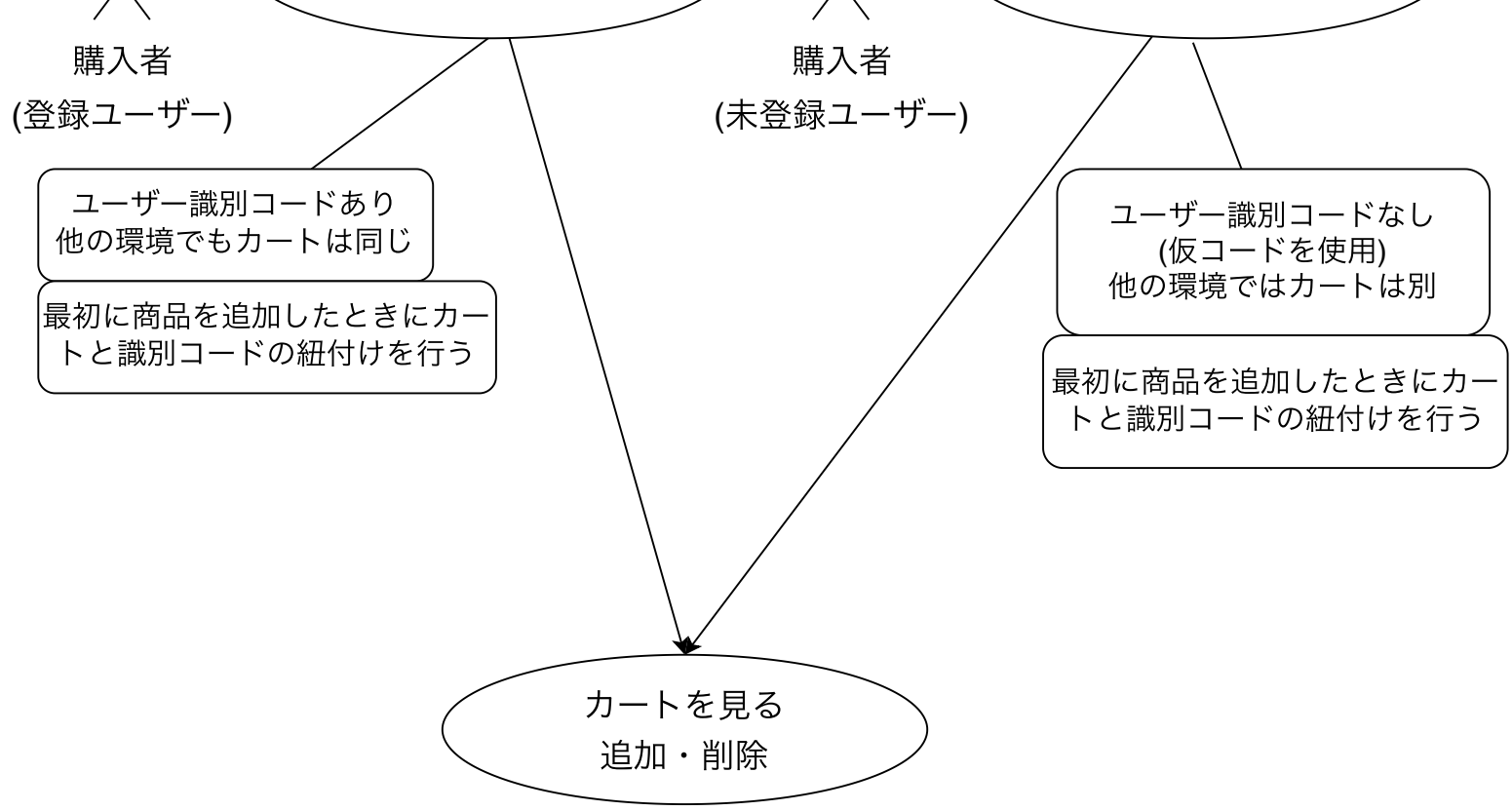
たとえばWebページでカートに入れた後、別のPCでログインした場合、カートの内容は引き継がれる必要がありますか？
　はい、PCやモバイルなど、どこから見てもカートの内容は引き継がれるようにしてください
　ただし未登録ユーザーの場合は引き継がれる必要はありません

その場合カートが購入者IDと紐づけられる必要があります。この紐付けはログイン時では未登録ユーザーは紐付けができなくなります。この場合の処理はどのようにしますか？
　ログインしても購入しない場合も考えられるので、カートに商品を入れた際に紐付けを行なってください。登録ユーザーの場合はユーザーID、未登録ユーザーの場合は仮のIDを割り振ってください
　購入時にログインやユーザー登録が行われた際に、仮IDを登録ユーザーIDに変更して購入処理をするようにしてください。

商品をカートに入れた場合、在庫を減らしますか？それとも購入時に減らしますか？
　購入時に減らしてください
　お客がカートに入れたままで購入・発送をしていないと日次処理で発注処理が行われるのは困ります
　さらに発注した後カートから戻されると在庫が無駄に増えてしまいます

カートに入れる時は商品の購入限度数はチェックしますか？また購入限度額はチェックしますか？
　購入限度数はチェックしてそれ以上カートに入らないようにしてください
　購入限度額は購入処理の際にチェックをするのでカートに入れる時はチェックしません
　　(限度額がオーバーしたからと出し入れを要求するのは購入者にとって面倒なので)





Q：購入について

購入時にカートの商品の数を増減したり、商品を削除するといった操作は行いますか？
　はい、5万円以上といった制限がありますので、購入時に調整ができるようにしていただきたいです

商品の数を増やす場合は最大購入数制限はありますか？
　はい、カートに入れる際と同じように処理してください
　ただし在庫を超えて購入はできません

商品を削除してカートの中身がなくなった場合はどうしますか？
　購入画面から商品カタログサイトに遷移してください

5万円以上という制限は配送料を含めますか？
　いいえ配送料は含めず、購入商品の合計価格に対してです

御社が送料を負担する場合と、そうではない場合に、送料の表示はどのようにしますか？
　配送料が必要な場合は、配送料 500円 と表示してください
　弊社が負担する場合は、配送料 500円 と表示後、その後に 配送料弊社負担 -500円 と表示して相殺しているのが分かるようにしてください

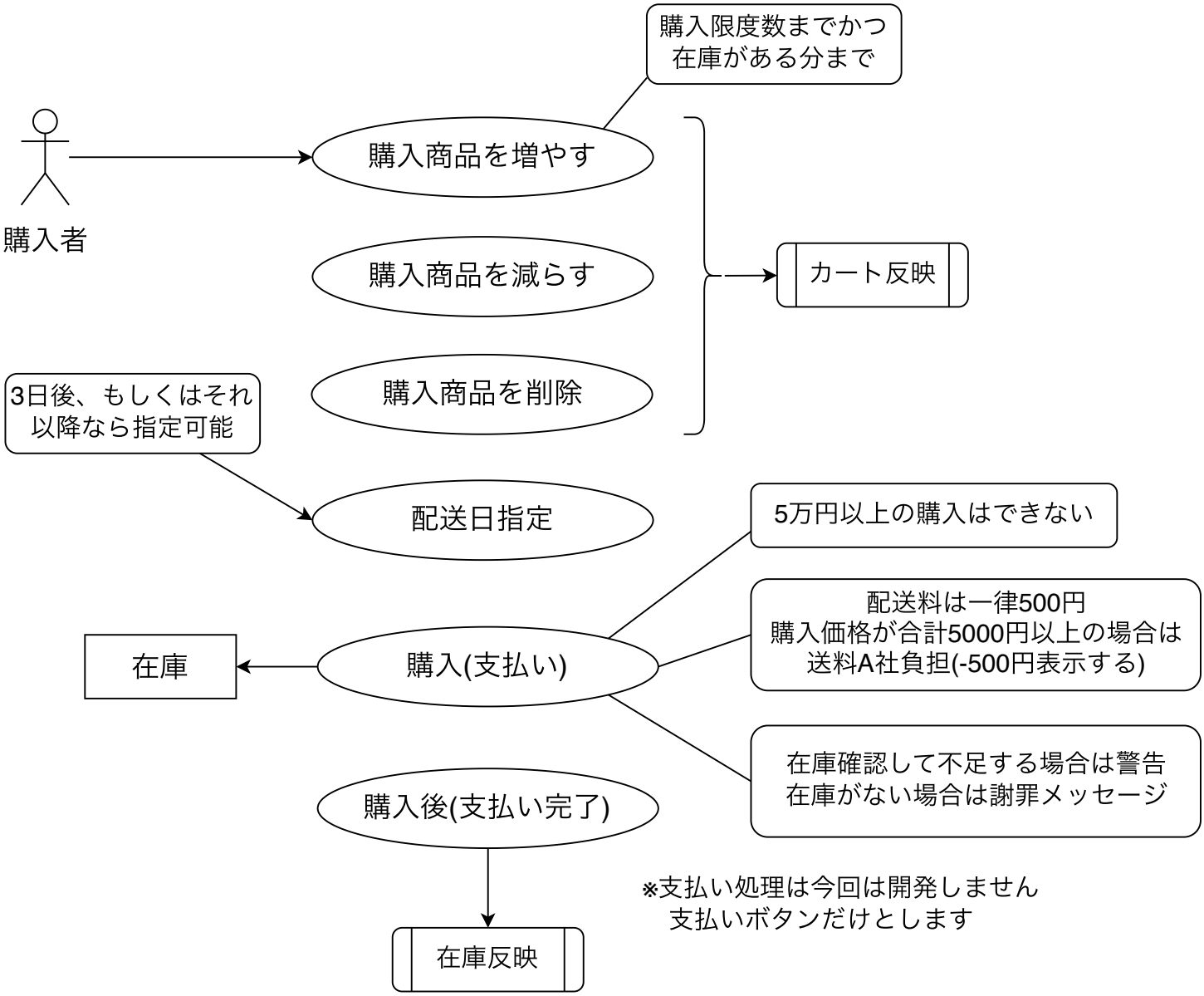
配送日は指定できますか?また基準的な配送予定日数はありますか?
　指定は3日後以降ならできますが時間指定はできません
　基準としては3日後と指定ください

購入が完了(支払い済み)となったら在庫の数を減らしますか？
　はい、支払い完了となりましたら在庫の数を減らしてください

購入処理前に他のお客が対象商品を購入して在庫数が不足する場合はどうしますか？

カートに入れた際には在庫があっても、購入時に在庫が不足となっている場合は警告メッセージを表示して購入可能数を表示して購入者に調整を依頼してください
また在庫が無くなった場合は、謝罪メッセージを表示して購入不可としてください

購入処理をした際に在庫を減らす順番はありますか？
1. 仕入れ価格の低いもの、2. 仕入れ日の古いもの、から減らしてください



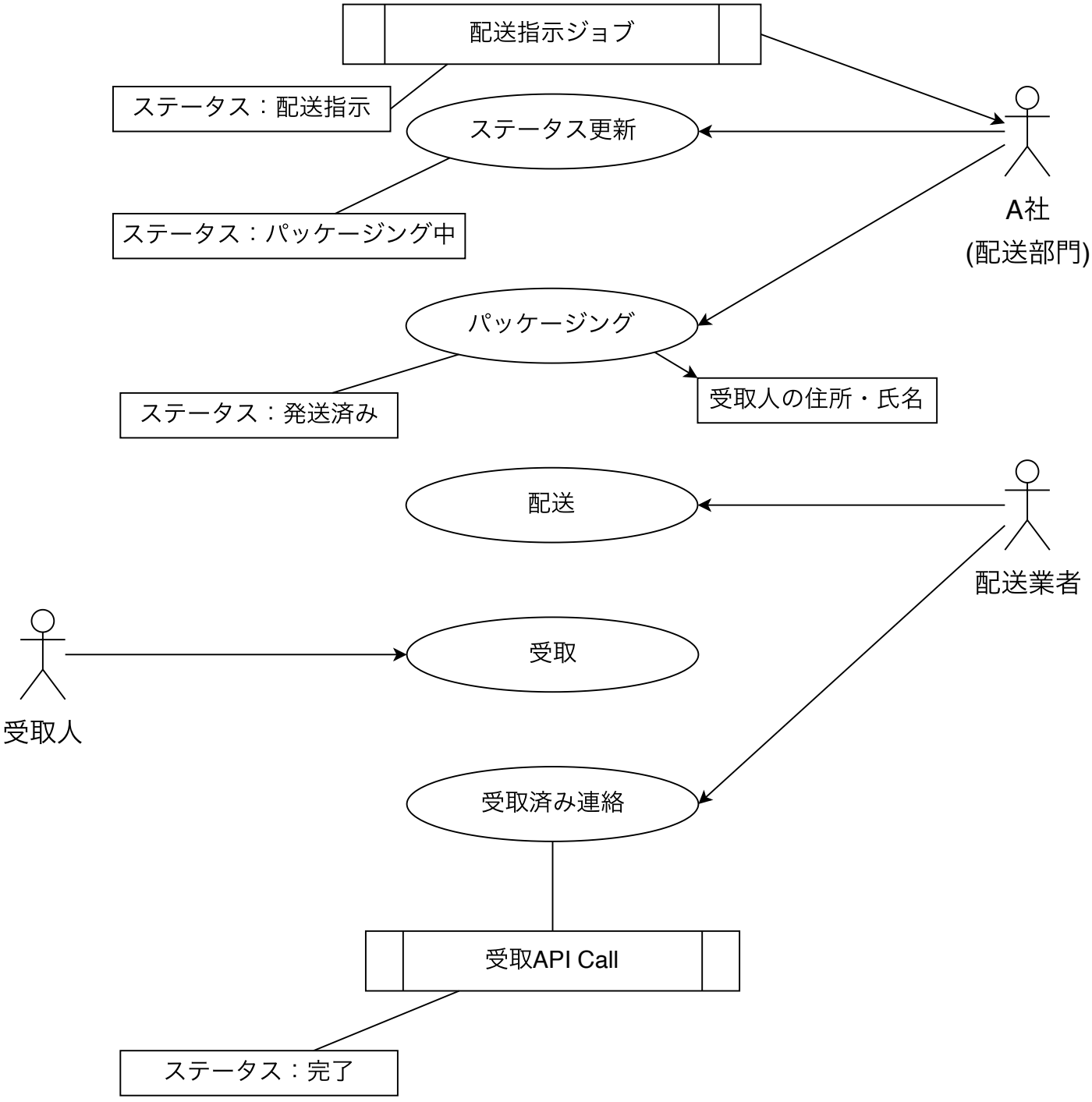
Q : 受注・配送手続き・発送について

受注から配送手続きの流れの詳細を教えてください
購入者の支払いが完了したら、配送手続きの指示が配送部門に出されます
このときのステータスは「配送指示」です
配送部門は指示を受けたらステータスを「パッケージング中」にしパッケージングを行います
パッケージングした荷物を配送業者に受け渡ししたらステータスを「発送済み」にします

受取人が荷物を受け取った際の流れの詳細を教えてください
受取人が荷物を受け取ると、配送業者が受取済みの連絡が届きます
受取済みの連絡を受けたらステータスが自動的に「完了」となります

配送業者の受取済み連絡のシステムは今回の開発対象ではありません

※配送業者の受取済み連絡のシステムは今回の開発対象ではありません
既存システムでAPIリクエストが来るようになっています



Q：日次処理・月次処理について

日次処理のタイミングはいつでしょうか？
　午前1時に起動するようにしてください

日次処理の在庫確認はどのように行うか詳細を教えてください
　前日に購入された商品を確認し、商品数が3以下のものについて自動発注します
　発注量は各商品ごとに適正在庫量が定められていて、それに基づいて決まります

購入と発送にタイムラグがあると思いますが日次処理ではあくまでも購入から算出された在庫量とい

うことになります？
　はい。実際の在庫計算は月次処理で棚卸を行うことで確認をします

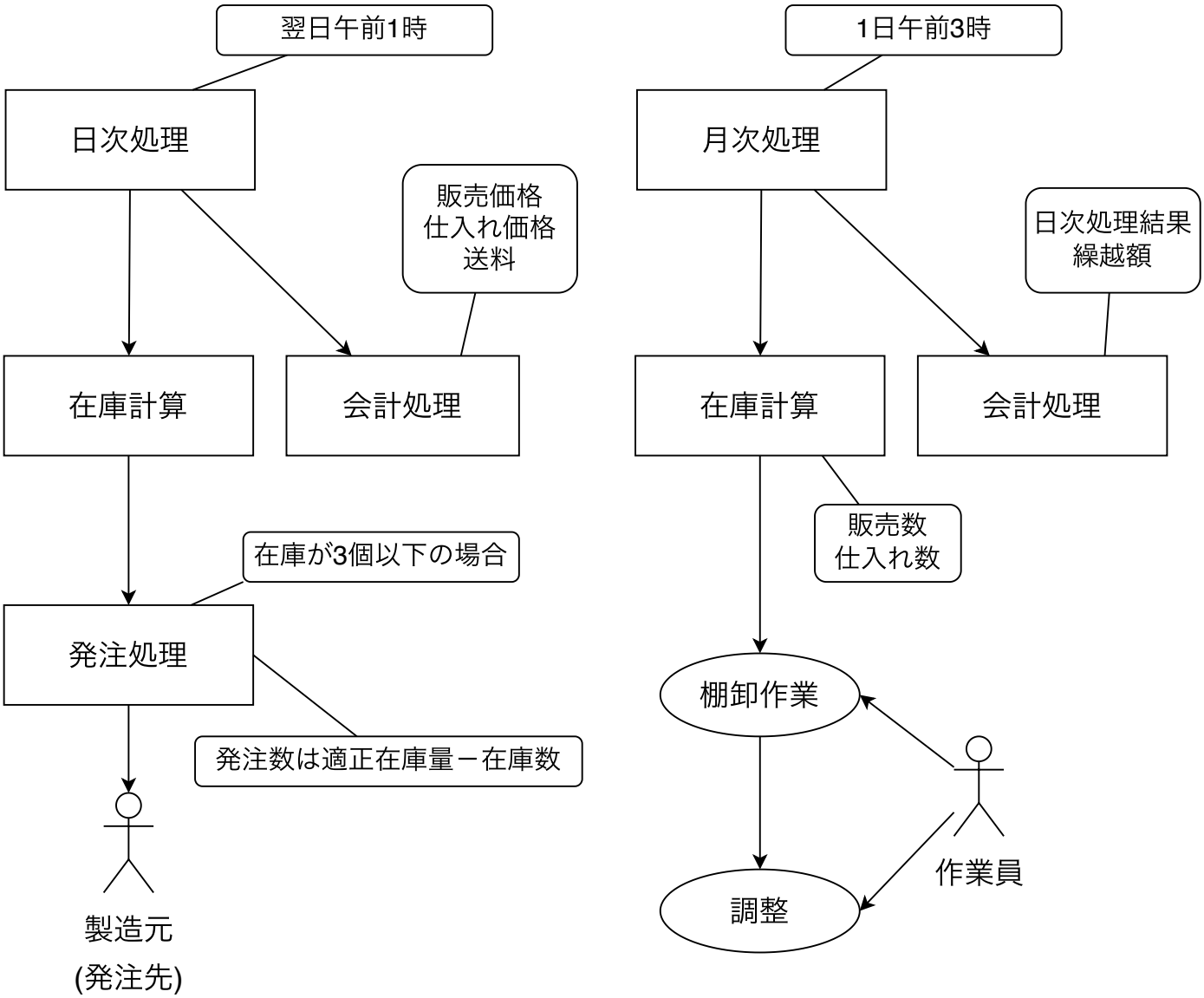
日次処理で行う会計処理を教えてください
　売上：商品販売価格x数量
　仕入れ：その商品の仕入れ価格(古いものから算出します)
　送料：お客様負担の場合は売上・運賃として計上し相殺されます
　　弊社負担の場合は費用として計上します

月次処理のタイミングはいつでしょうか？
　毎月1日午前3時に起動してください

月次処理の在庫確認はどのように行いますか？
　また前月の1日から月末までの売上数量、納品数量を算出し、計算上の各商品の在庫数を出します
　月初日にその在庫数が実際にあるかを作業員が確認を行います。
　確認は商品バーコードと数量チェックを行い、OK/NGを出します
　NGの場合の処理は差分数を記録し、数量を実数に変更します

月次処理の会計処理を教えてください
　該当月の売上・支出を日次処理結果から集約し、前月の繰越額と総計して、次月の繰越額を算出します

---ジョブ---



今回のアプリでは発注したらすぐ
に在庫処理(補充)としておきます

Q : カートについて
画面構成は以下のようなものでよろしいでしょうか？
　商品一覧から商品(商品情報画面)を選択
　商品情報で個数を選んでカートに追加
　カートに商品を入れたらカート確認画面に遷移する
　カート確認画面でカートの内容の修正(個数の変更)
　カート確認画面で購入ボタンを押すと、購入画面に遷移する

カートを使用できるお客は登録ユーザーですか？それとも未登録ユーザーも利用可能ですか？
　購入者の管理はしませんのでログイン等はありません
　購入時(注文)に送付先を入れていただきます

商品をカートに入れた場合、在庫を減らしますか？それとも購入時に減らしますか？
　購入時に減らしてください
　お客がカートに入れたままで購入・発送をしていないと日次処理で発注処理が行われるのは困ります
　さらに発注した後カートから戻されると在庫が無駄に増えてしまいます

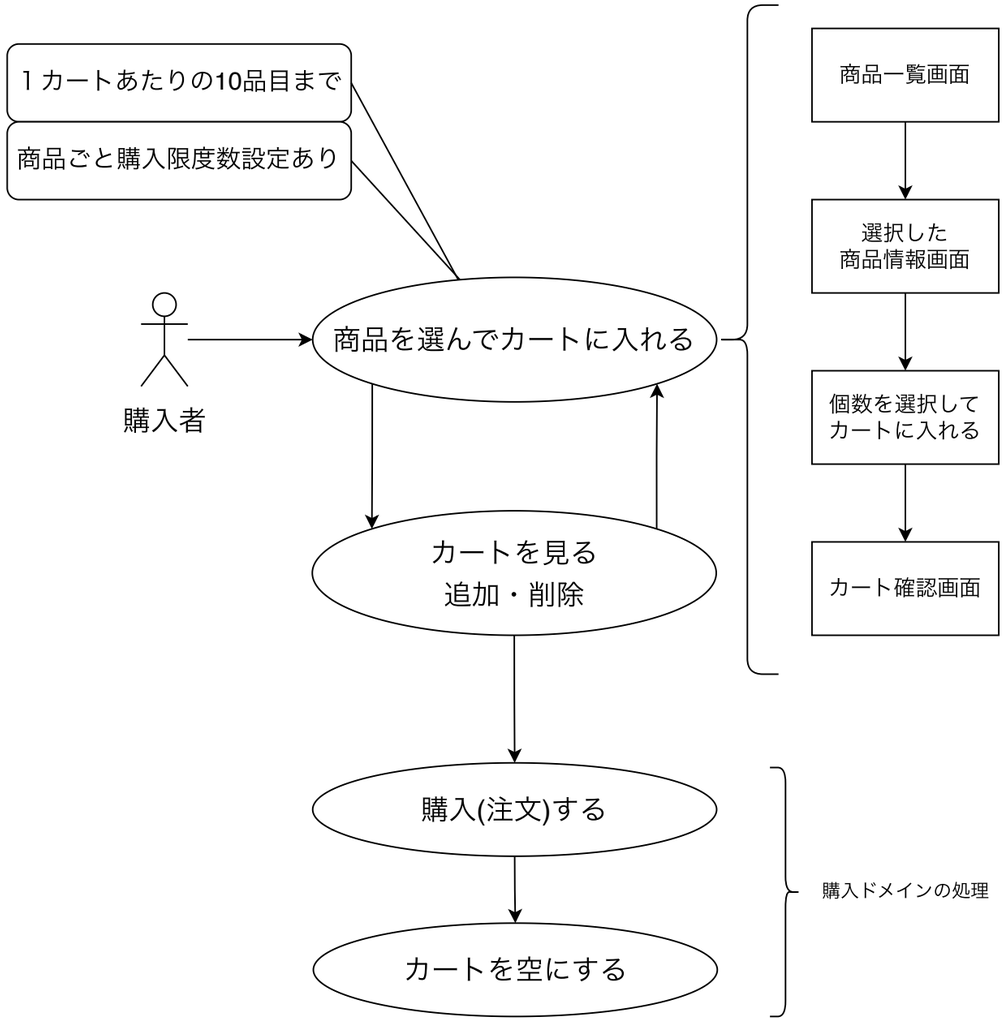
カートに入れる時は商品の購入限度数はチェックしますか？また購入限度額はチェックしますか？
　購入限度数はチェックしてそれ以上カートに入らないようにしてください
　購入限度額は購入処理の際にチェックをするのでカートに入れる時はチェックしません
　　(限度額がオーバーしたからと出し入れを要求するのは購入者にとって面倒なので)

カートについて以下の要件がありますが追加等ありますでしょうか
　・商品ごとに1人(1カート)で購入できる数を限定する
　・カートに入れられるのは10品目まで
　現在は追加はありません

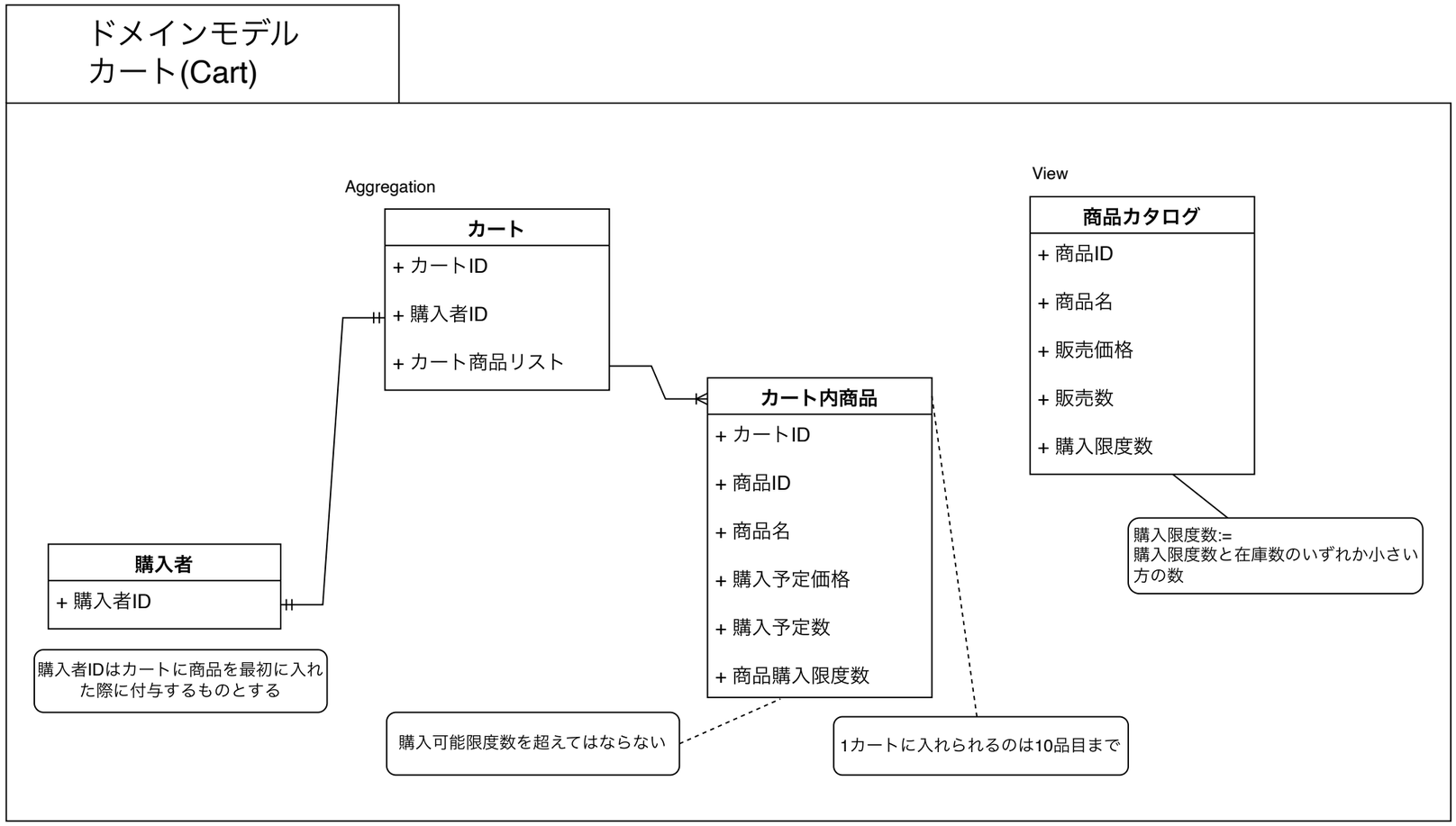
商品販売価格が変更となった際のカート内の商品の価格はどのように連動しますか？
例えばカートに入れたまま翌日になったら100円価格が上がっていた場合等です
　カート内の商品も価格変動に連動してください
　カートの中を見るという画面に遷移した際に最新の価格になるようにしてください
　購入時は最新価格で計算して購入処理を行うようにしてください

購入(注文)が完了したらカートはクリアされるということでよろしいですか？
　はいお客様の注文が完了したタイミングでカートは空になります

ユースケース図 (業務フロー)



ユースケースとは業務におけるフローを表したもの
各イベントがどのような流れで行われるかというものということです



ドメインモデルとは業務の中心的な事項をモデルとして記述したもの

業務の中心的な事項とは
　業務ルール
　　カートに入れられるのは10商品まで

　業務イベント
　　Aさんが商品1をカートに入れた

このような業務ロジックに関わるイベント(行動)・ルールをドメインモデルに記述します

各レイヤーごとに独自のData Modelを持ちます

プレゼンテーション・レイヤー
　コントローラー(API)を実装し、ResponseとしてView用のデータを返します

各レイヤーごとにDataクラスを持ち、処理メソッドを持つため、
DDDではクラスの数がこれまでのコーディングよりも非常に多くなります

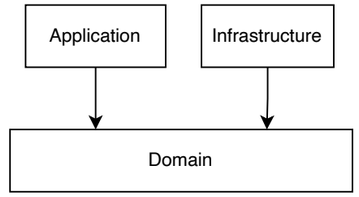
ユースケース・レイヤー
　業務フローを実装したもの
　DataとしてはRepositoryから取得したドメインモデルを使用し、
　結果をDTO(Data Transfer Object)を用いて表現する

ドメインレイヤー
　ドメインモデル : 業務ロジックと業務ルールを実装したもの
　イミュータブルであり、プロパティにValueObjectを使用する

インスラストラクチャレイヤー
　永続化処理 (DB保存) を行う

そのため、規模が小さい場合はコーディングの負担を減らす目的で
プレゼンテーションとユースケースを一体化させる場合もあります

MVCモデルに近いですが、
Domainのコーディングが
明確となっているため、
レイヤーのルールが守られ
コードの保全が行いやすい



Presentation(API)

CartController
- – IStockUseCase
- + Post putItem(PutCartItemRequest) : CartViewModel
- + Post deleteStockItem(PutCartItemRequest) : CartViewModel
- + Post modifyStockItem(PutCartItemRequest) : CartViewModel

CatalogueController
- – ICatalogueQueryService
- + getAll(pageNumber: int): List<CatalogueViewModel>
- + getById(id: String): CatalogueViewModel

CartViewModel
- + CartId: String
- + BuyerId: String
- + CartItemViewList: List<CartItemViewModel>
- + ResponseMessage: String
- + @Data

CartItemViewModel
- + CartId: String
- + ItemId: String
- + ItemName: String
- + ItemPrice: long
- + formattedItemPrice: String
- + itemAmount: int
- + itemAmountView: String
- + itemLimitedCount: int
- + itemLimitedCountView: String

CatalogueViewModel
- + CatalogItemId: String
- + CatalogItemName: String
- + SalesPrice: long
- + StockAmount: int
- + PurchaseLimit: int

UseCase

CartIdがnull or emptyの場合はカートを作成する

CartUseCase
- – ICartRepository
- + <Implement>

CatalogueQueryService
- – CatalogueItemJpaRepository
- + <Implement>

«ICartUseCase»

«ICatalogueQueryService»

Infrastructure

CartRepository
- + <Implement>

DB

CartEntity (name: CartTable)
- + CartId: String
- + BuyerId: String

CartDto getCartByCartId(String cartId)

CartDto putItemIntoCart(String cartId,
CartItemDto cartItemDto)

CartDto modifyCartItem(String cartId,
CartItemDto cartItemDto)

List<CatalogueItemEntity> getAll(int pageNumber)
CatalogueItemEntity get(String id)

CartDto

+ CartId: String
+ BuyerId
+ CartItems: List<CartItemDto>

+ fromCartModel(CartModel): this
+ toCartModel(this): CartModel
+ getter, setter
+ builder

CartItemDto

+ CartId: String
+ ItemId: String
+ ItemName: String
+ ItemPrice: long
+ ItemCount: int
+ ItemLimitedCount: int

+ CartItemEntities: List<CartItemEntity>

多 対 多
実際上は中間テーブルが自動作成されます

CartItemEntity (name: CartItemTable)

+ Id: String
+ CartId: String
+ ItemId: String
+ ItemName: String
+ ItemPrice: long
+ ItemAmount: int
+ ItemLimitedCount: int

CartTable_CartItemTable

+ CartTableId: String
+ CatItemTableId: String

CatalogueItemEntity

+ CatalogItemId: String
+ CatalogItemName: String
+ SalesPrice: long
+ StockAmount: int
+ PurchaseLimit: int

CartJpaRepository: JpaRepository<CartEntity, String>

+

CartItemJpaRepository: JpaRepository<CartItemEntity, String>

+

CatalogueJpaRepository: JpaRepository<CatalogueEntity, String>

+

Domain

CartModel

+ CartId: String
+ BuyerId
+ CartItems: List<CartItem>

+ constructor(full)
+ getter methods
+ putItemIntoCart(CatalogItem): void
+ modifyItemInCart(CatalogItem): void
+ clearCart(): void

«ICartRepository»

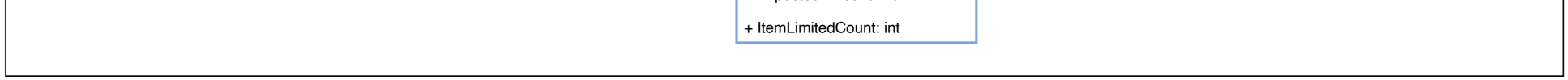
save(CartModel model)
getCartByCartId(String cartId): CartModel
getCartByBuyerId(String buyerId): CartModel
getItemByItemIdFromCatalog(String itemId): CartItem

Buyer

+ buyerId: String

CartItems: record

+ CartId: String
+ ItemId: String
+ ItemName: String
+ ExpectedPrice: long
+ ExpectedAmount: int



コンストラクタ
　カートを作成した時にカートId, 購入者Id, 商品を割り当てますが、
　カートIdがnullの場合は新しくIdを割り当てる
　購入者Idがnullの場合は新しくIdを割り当てる
　カート内商品がnullの場合は新しくArrayListを割り当てる　　ようにする。

putItemIntoCart(CatalogItem): void
　カタログ商品をカートに追加するメソッド

　カートに入れた商品がすでにカート内にある場合
　　商品の購入希望数に追加する

　カートに入れた商品がすでにカート内にあり、価格が異なる場合
　　すでにカートに入っている商品を新しく入れた商品の価格に統一する

　カートの中のカタログ商品の種類の数が10を超えたらエラーとする
　IllegalArgumentException
　メッセージ: カートには10商品を超えて入れることはできません

　カタログ商品で設定されている購入限度数を超えて同じ商品をカートに入れた場合はエラーとする
　IllegalArgumentException
　メッセージ: 購入限度数%sを超えてカートに入れることはできません
　　%s := カタログ商品の購入限度数 (int)

modifyItemInCart(CatalogItem): void
　カート画面でカタログ商品の商品数を変更する
　カタログデータから商品情報を取得して、最新データで変更登録する
　(カートに入れた後で限度数等が変更となっている可能性があるため)
　カタログ商品で設定されている購入限度数を超えて増やした場合はエラーとする
　IllegalArgumentException
　メッセージ: 購入限度数%sを超えてカートに入れることはできません
　　%s := カタログ商品の購入限度数 (int)
　購入希望数が0の場合はカートから削除する

clearCart(): void
　カートの中身を全て削除する

プロパティとフィールド
　プロパティとはクラスが持っている情報
　ex) PersonクラスにNameプロパティがある・・・その人の「名前」がプロパティ
　　外部に表すもの

　フィールドとはプロパティがクラス内部で持っている値そのもの
　ex) Nameプロパティは private String nameで値を保持している
　　nameフィールドが変われればNameも変わる

```
public Class Person{
　private String name;

　public String Name() { return this.name; }

　public void changeName(String value) { this.name = value; }
}
```

Javaではこれをgetter, setterというメソッドで表すのでプロパティという概念が掴みにくいです
　public String getName() { return this.name; }
　public void setName(String value) { this.name = value; }.　どこにもNameそのものはない・・・

C#では public string Name { get; set; } というプロパティの書き方をします。
そのためフィールドという概念が掴みにくいですwww (フィールドを作成しなくても裏で勝手に作られる)

lombok アノーテーションについて
アノーテーションを付加することで、コンパイル時に自動的にコードが生成されます
　@AllArgsConstructor
　　すべてのフィールドを含めたコンストラクタを作成します

　@NoArgsConstructor
　　パラメータのないコンストラクターを生成します

　@Getter
　　フィールドに対するGetterメソッドを作成します

　@Setter
　　フィールドに対するSetterメソッドを作成します

　@Data
　　@ToString, @EqualsAndHashCode, @Getter, @Setter, @RequiredArgsConstructorを合わせたものです

　@Builder
　　クラスを生成する際にビルダーパターンでの生成を提供します

データベース用に使用されるアノーテーション
　@Entity
　　テーブルとして生成されます

　@Id
　　テーブルの主キーとなります

　@OnToOne, OneToMany, ManyToOne, ManyToMany
　　テーブルのリレーションを表します
　　それぞれ　1 対 1、1 対 多、多 対 1、多 対 多　となります
　　多: nと表すことが一般的
　　　n - n の場合は中間テーブルを必要となります
　　　Jpaではn - nの場合自動的に中間テーブルを作成します

Q : カートについて
画面構成は以下のようなものでよろしいでしょうか？
　商品一覧から商品(商品情報画面)を選択
　商品情報で個数を選んでカートに追加
　カートに商品を入れたらカート確認画面に遷移する
　カート確認画面でカートの内容の修正(個数の増減・削除)
　カート確認画面で購入ボタンを押すと、購入画面に遷移する
　購入画面遷移の際に未登録ユーザーの場合はログイン画面・ユーザー登録画面に遷移してから購入画面に遷移する

カートを使用できるお客は登録ユーザーですか？それとも未登録ユーザーも利用可能ですか？
　カートに商品を入れている間は未登録ユーザーでも可能です
　購入時にユーザー登録もしくはログインが必要となります
　ログインのタイミングはカートに入れている途中でも可能とします
　登録していないと配送先が分からないので最終的に登録は必須です

登録ユーザーはログインを行いますか？ログインに必要なものは何ですか？
　ログイン機能が必要です
　必要なものはメールアドレスとパスワードです
　ユーザーの識別にはメールアドレスを使用します

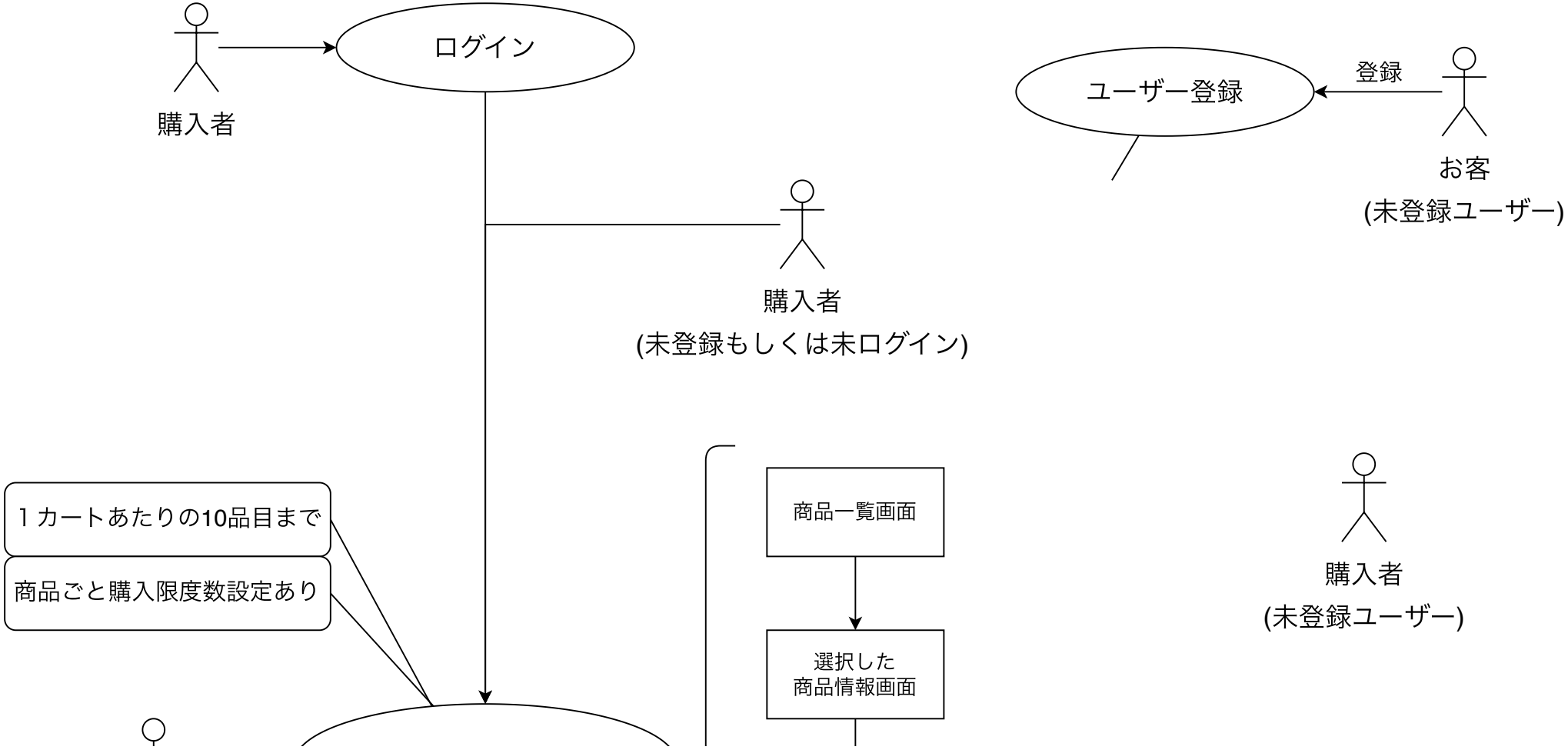
たとえばWebページでカートに入れた後、別のPCでログインした場合、カートの内容は引き継がれる必要がありますか？
　はい、PCやモバイルなど、どこから見てもカートの内容は引き継がれるようにしてください
　ただし未登録ユーザーの場合は引き継がれる必要はありません

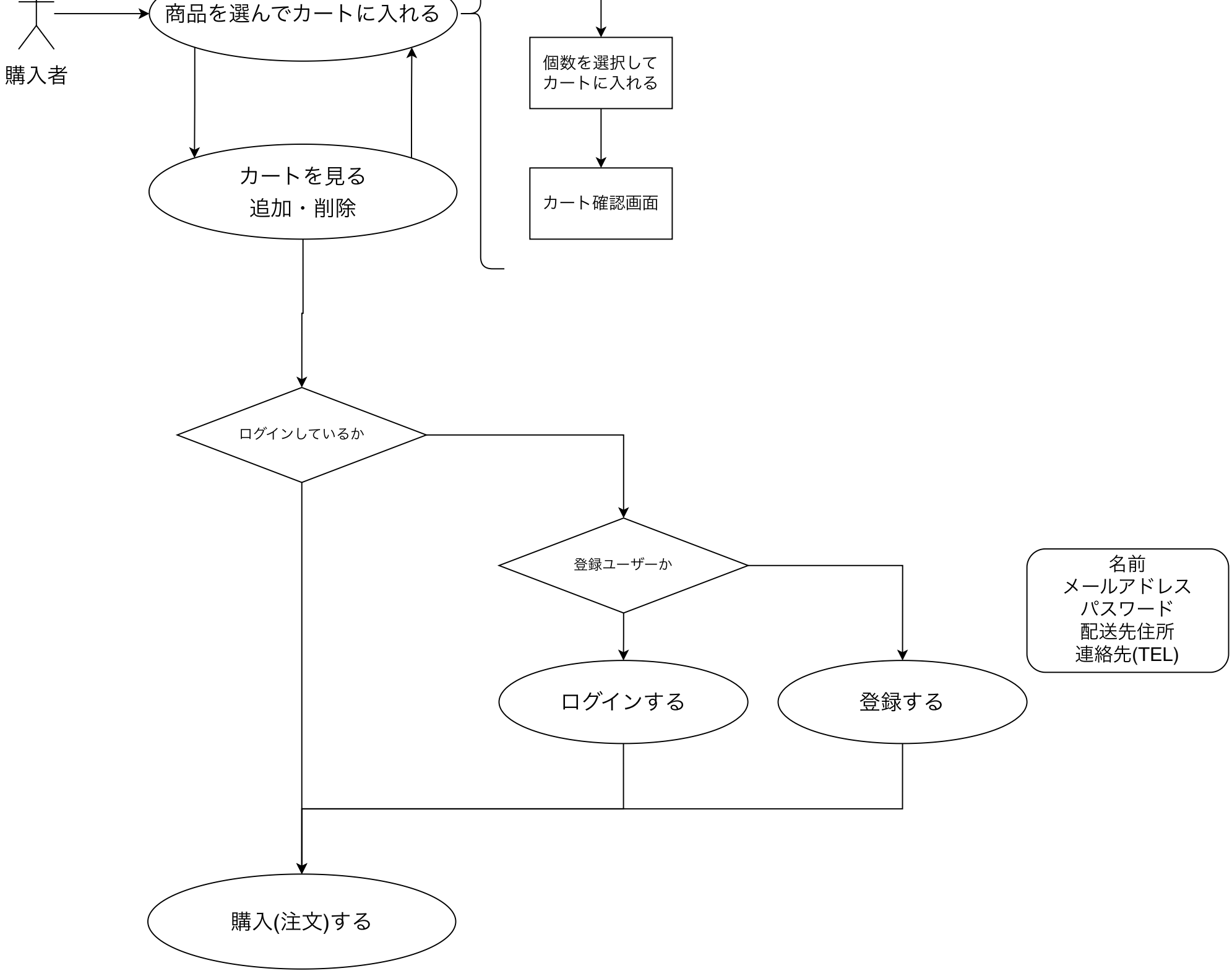
商品をカートに入れた場合、在庫を減らしますか？それとも購入時に減らしますか？
　購入時に減らしてください
　お客がカートに入れたままで購入・発送をしていないと日次処理で発注処理が行われるのは困ります
　さらに発注した後カートから戻されると在庫が無駄に増えてしまいます

カートに入れる時は商品の購入限度数はチェックしますか？また購入限度額はチェックしますか？
　購入限度数はチェックしてそれ以上カートに入らないようにしてください
　購入限度額は購入処理の際にチェックをするのでカートに入れる時はチェックしません
　　(限度額がオーバーしたからと出し入れを要求するのは購入者にとって面倒なので)

カートについて以下の要件がありますが追加等ありますでしょうか
・商品ごとに1人(1カート)で購入できる数を限定する
・カートに入れられるのは10品目まで
現在はありませんが、今後予定したい機能としてはクーポン、おすすめ商品等です

商品販売価格が変更となった際のカート内の商品の価格はどのように連動しますか？
例えばカートに入れたまま翌日になったら100円価格が上がっていた場合等です
カート内の商品も価格変動に連動してください
カートの中を見るという画面に遷移した際に最新の価格になるようにしてください
購入時は最新価格で計算して購入処理を行うようにしてください



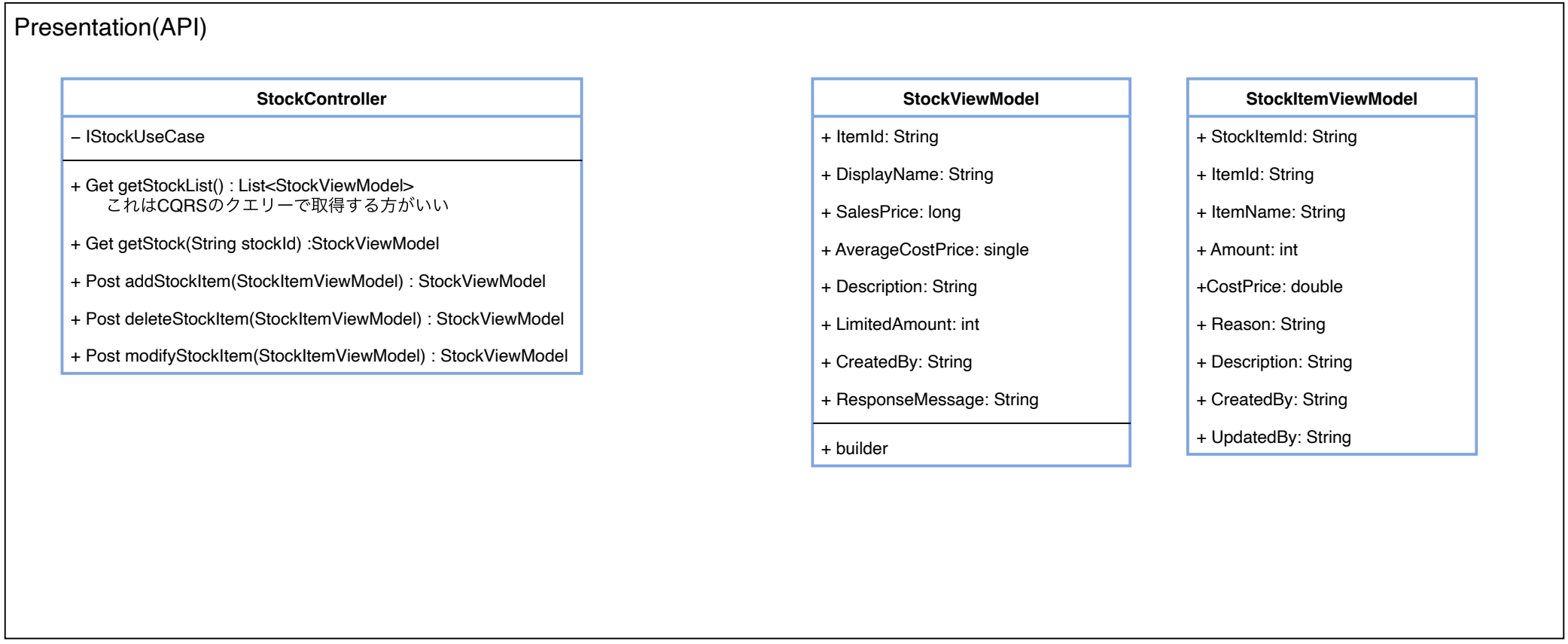

購入者
商品を選んでカートに入れる
個数を選択して
カートに入れる
カート確認画面
カートを見る
追加・削除
ログインしているか
登録ユーザーか
名前
メールアドレス
パスワード
配送先住所
連絡先(TEL)
ログインする
登録する
購入(注文)する

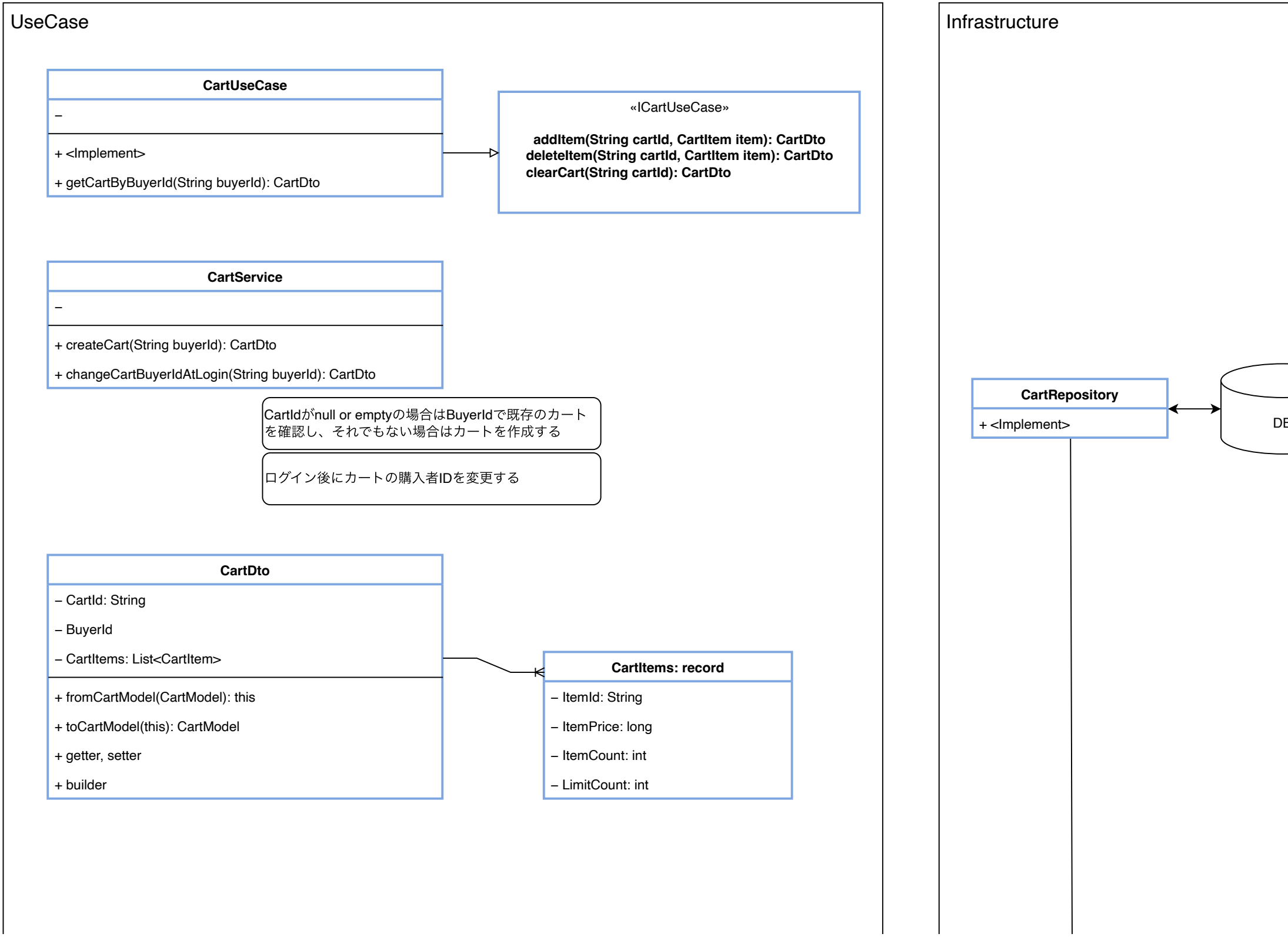
ユーザー識別コードなし
(仮コードを使用)
他の環境ではカートは別

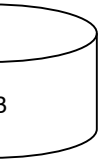
最初に商品を追加したときにカートと識別コードの紐付けを行う

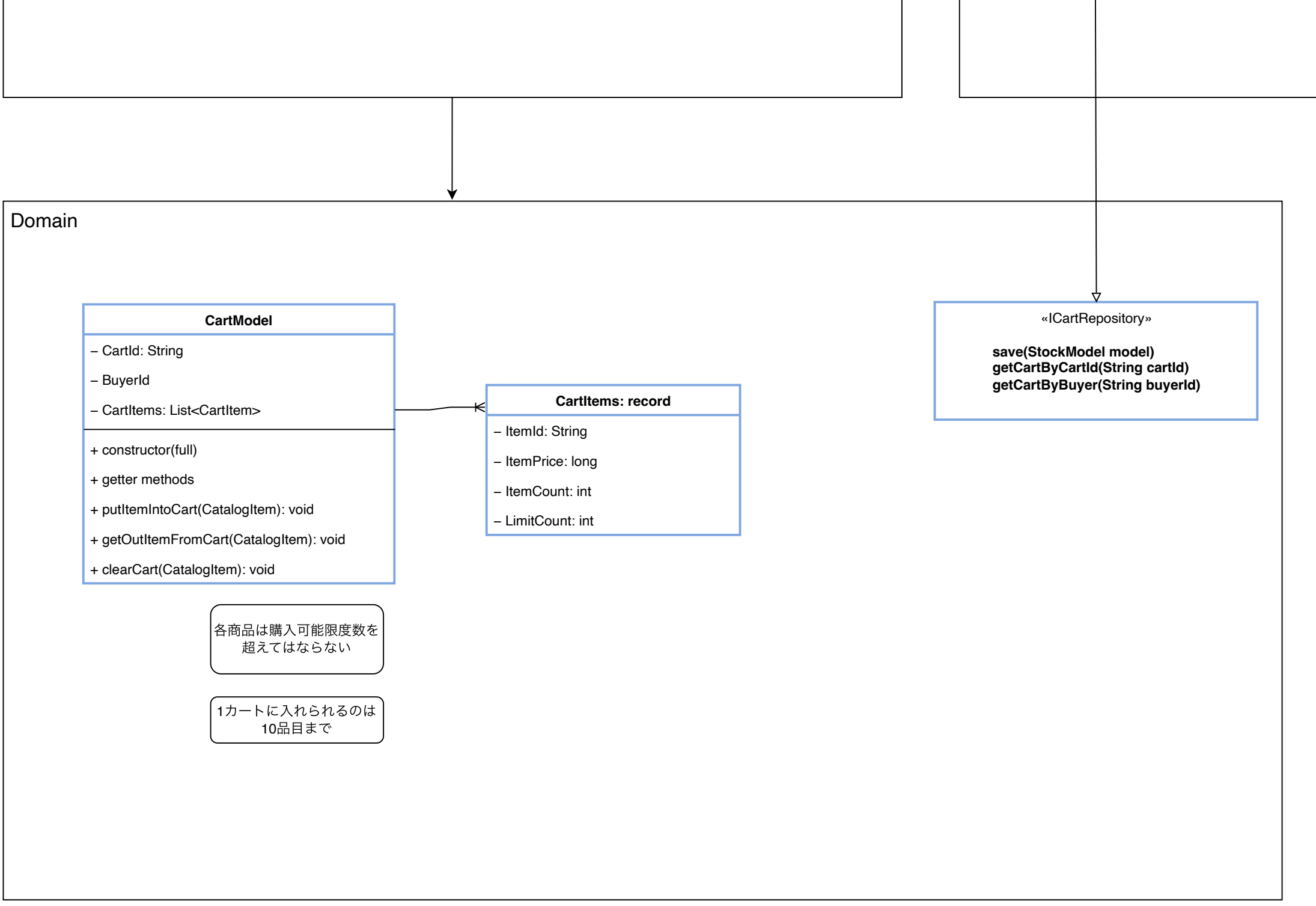
最初に商品を追加したときにカートと識別コードの紐付けを行う

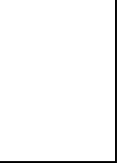
ユーザー識別コードあり
他の環境でもカートは同じ

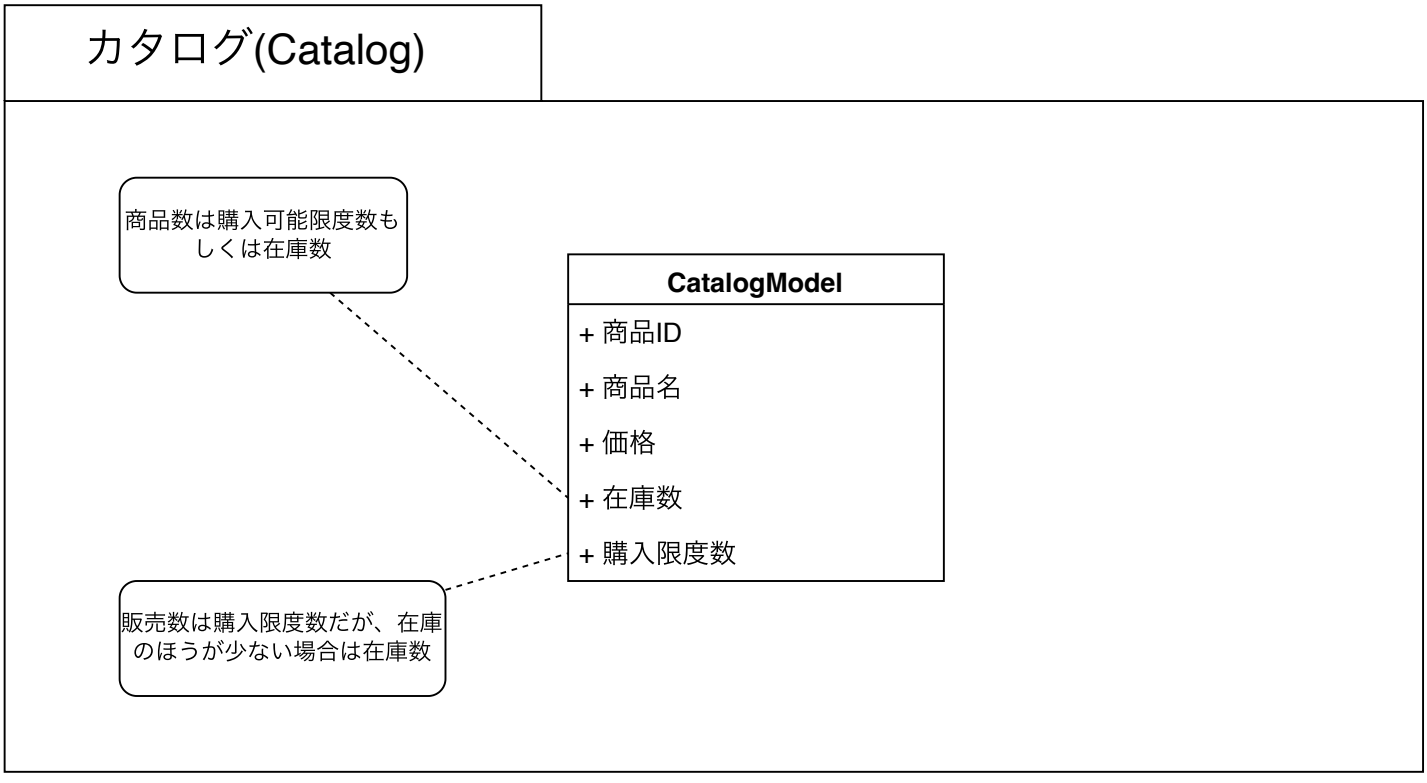

Presentation(API)
StockController
– IStockUseCase
+ Get getStockList() : List<StockViewModel>
これはCQRSのクエリーで取得する方がいい
+ Get getStock(String stockId) :StockViewModel
+ Post addStockItem(StockItemViewModel) : StockViewModel
+ Post deleteStockItem(StockItemViewModel) : StockViewModel
+ Post modifyStockItem(StockItemViewModel) : StockViewModel
StockViewModel
+ ItemId: String
+ DisplayName: String
+ SalesPrice: long
+ AverageCostPrice: single
+ Description: String
+ LimitedAmount: int
+ CreatedBy: String
+ ResponseMessage: String
+ builder
StockItemViewModel
+ StockItemId: String
+ ItemId: String
+ ItemName: String
+ Amount: int
+CostPrice: double
+ Reason: String
+ Description: String
+ CreatedBy: String
+ UpdatedBy: String


UseCase
CartUseCase
–
+ <Implement>
+ getCartByBuyerId(String buyerId): CartDto
«ICartUseCase»
addItem(String cartId, CartItem item): CartDto
deleteItem(String cartId, CartItem item): CartDto
clearCart(String cartId): CartDto
CartService
–
+ createCart(String buyerId): CartDto
+ changeCartBuyerIdAtLogin(String buyerId): CartDto
CartIdがnull or emptyの場合はBuyerIdで既存のカートを確認し、それでもない場合はカートを作成する
ログイン後にカートの購入者IDを変更する
CartDto
– CartId: String
– BuyerId
– CartItems: List<CartItem>
+ fromCartModel(CartModel): this
+ toCartModel(this): CartModel
+ getter, setter
+ builder
CartItems: record
– ItemId: String
– ItemPrice: long
– ItemCount: int
– LimitCount: int
Infrastructure
CartRepository
+ <Implement>
DB

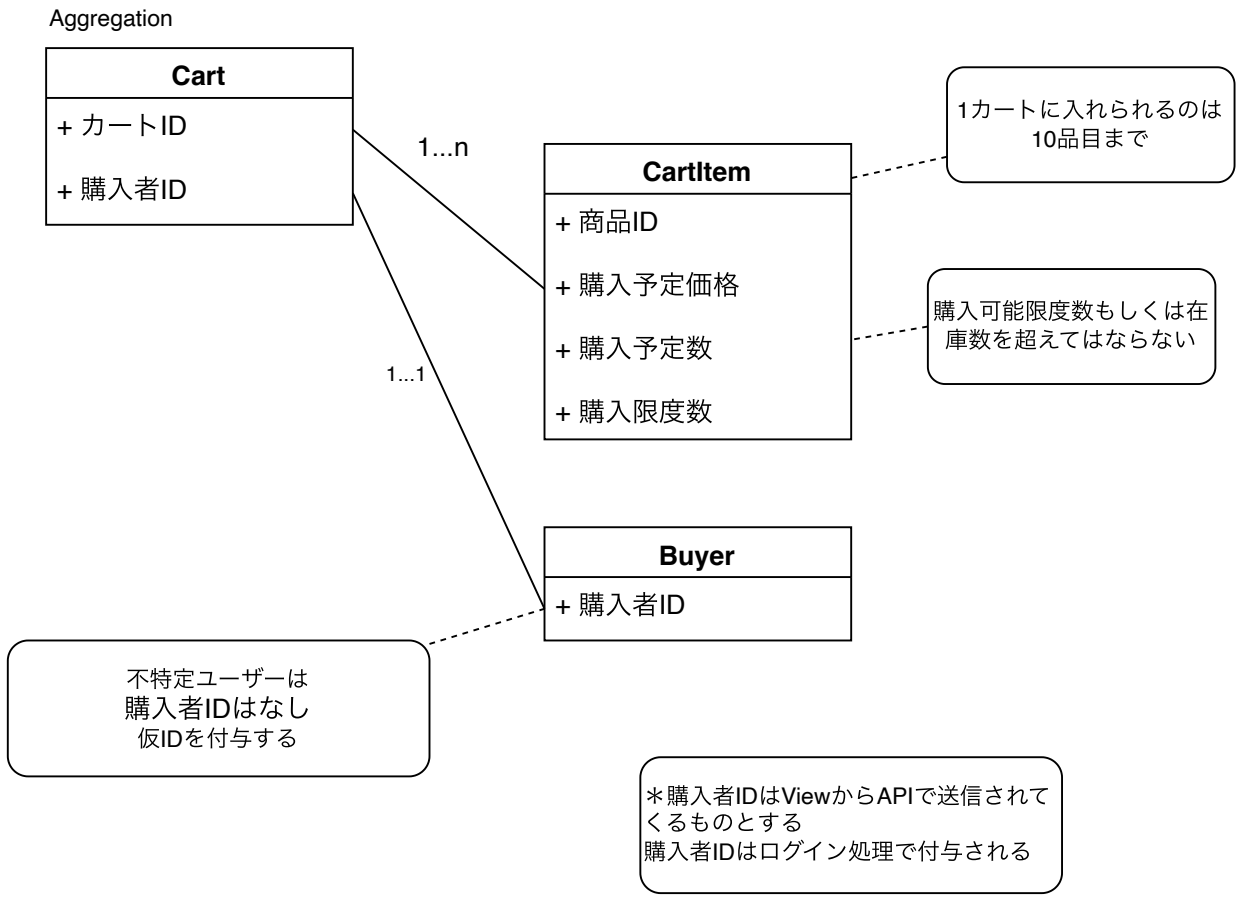



Domain
CartModel
– CartId: String
– BuyerId
– CartItems: List<CartItem>
+ constructor(full)
+ getter methods
+ putItemIntoCart(CatalogItem): void
+ getOutItemFromCart(CatalogItem): void
+ clearCart(CatalogItem): void
CartItems: record
– ItemId: String
– ItemPrice: long
– ItemCount: int
– LimitCount: int
«ICartRepository»
save(StockModel model)
getCartByCartId(String cartId)
getCartByBuyer(String buyerId)
各商品は購入可能限度数を
超えてはならない
1カートに入れられるのは
10品目まで

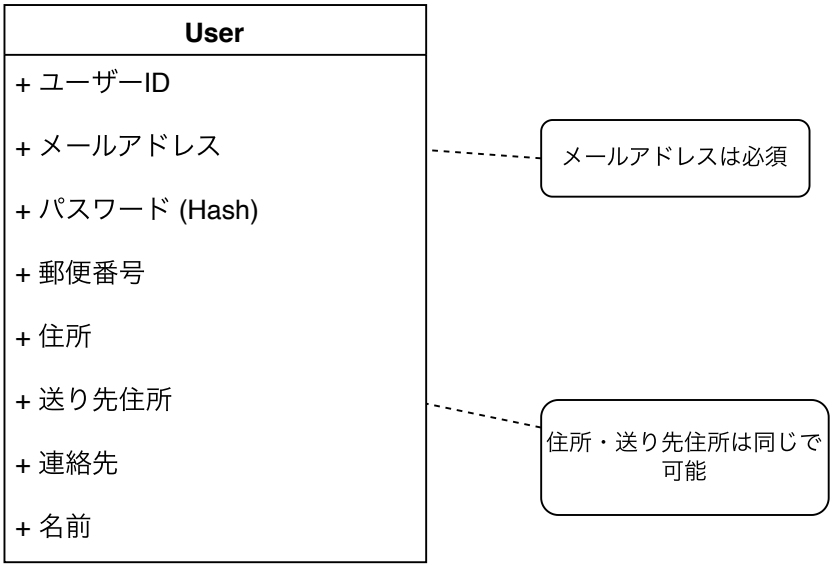



カタログ(Catalog)
商品数は購入可能限度数もしくは在庫数
CatalogModel
+ 商品ID
+ 商品名
+ 価格
+ 在庫数
+ 購入限度数
販売数は購入限度数だが、在庫のほうが少ない場合は在庫数

カート(Cart)



ユーザー登録



購入(Purchase)

Aggregation

購入者

+ 購入者ID
+ 郵便番号
+ 住所
+ 送り先住所
+ 連絡先
+ 名前

支払いCart

+ カートID
+ 購入者ID
+ 合計額
+ 送料
+ 配送予定日

5万円以上の購入はできない

配送料は一律500円
購入価格が合計5000円以上の場合は
送料A社負担(-500円表示する)

CartItem

+ 商品ID
+ 購入予定価格
+ 購入予定数
+ カートID

在庫確認して不足する場合は警告
在庫がない場合は謝罪メッセージ

増やす場合は購入限度数まで
かつ
在庫がある分まで

受注

受取人

+ 受取人ID
+ 郵便番号
+ 受取住所
+ 連絡先
+ 名前

配送パッケージ

+ パッケージID
+ ステータス
+ 配送予定日

<ステータス>
配送指示
パッケージング中
発送済み
完了

配送品

+ パッケージID
+ 商品ID

+ 商品数

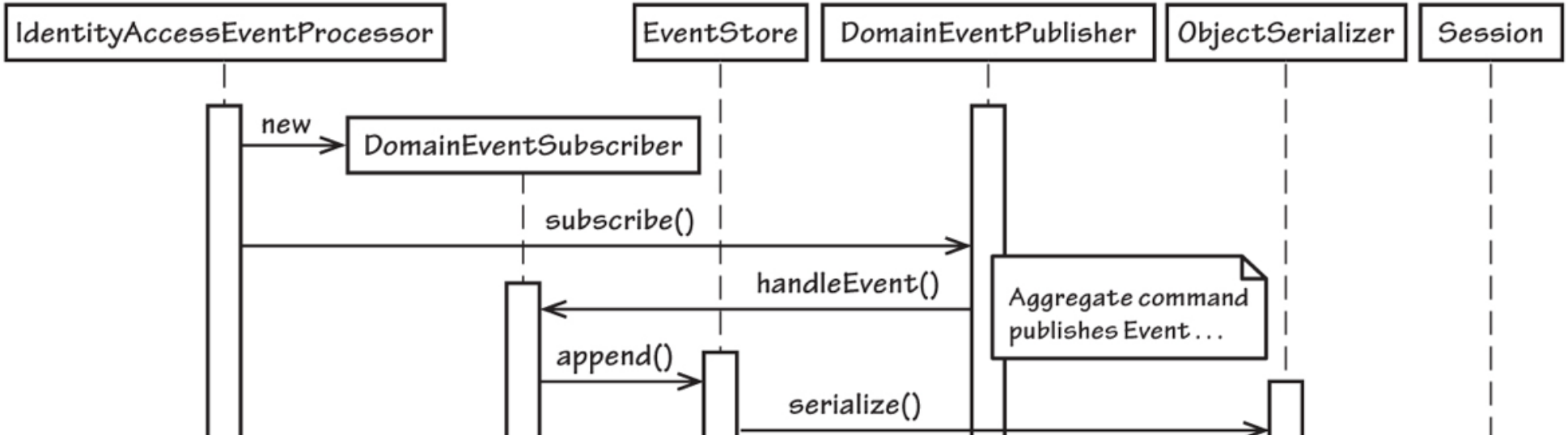
DomainModel: abstract

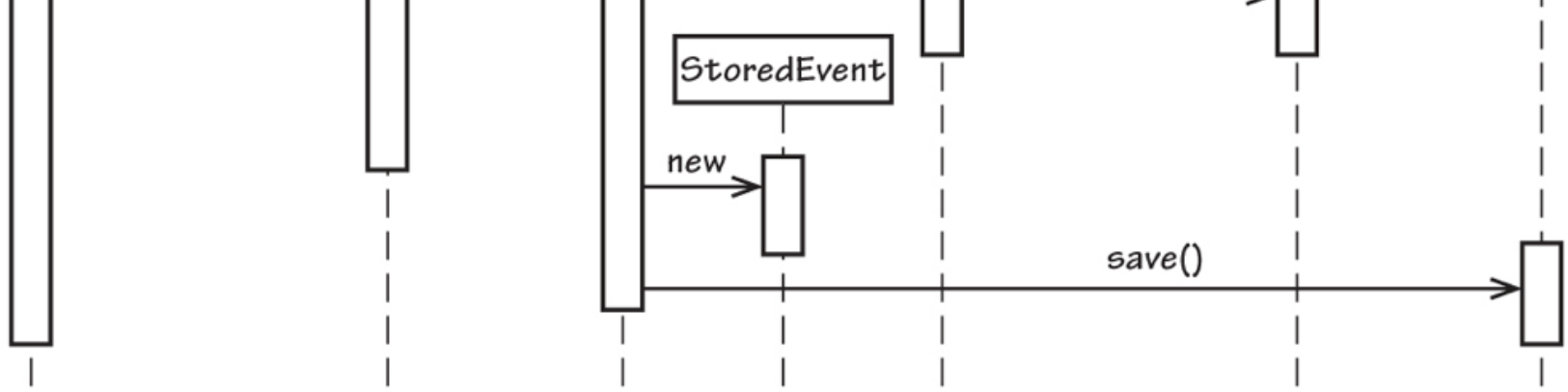
+ CreatedAt
+ CreatingUser
+ UpdatedAt
+ UpdatingUser

«interface»
AggregateRoot

(マーカーインターフェイス)

«interface»
DomainEvent




StoredEvent
new
save()