世界一わかりやすいClean Architecture

~ 技術レイヤー分割より大切なモノ ~

Atsushi Nakamura

Goals of this session

Goals of this session

全員が

「クリーンアーキテクチャチョットデキル」

ようになること

Easiest Clean Architecture

Overview

Overview

Clean Architectureの次の点についてお話させていただきます。

- 誤解されがちな二つのこと
- Clean Architectureの本質
- 技術レイヤー分割より大切なこと
- おさえるべき三つのこと

Slide 5

注意事項

注意事項

今日お話しすることは、あくまで私の解釈です

- 極端に要約しているので、これがすべてではありません
- またあえて拡大解釈している箇所もあります
- 書籍の記載とやや矛盾する点もあります

しかし、私は今日お話しする内容こそ、Clean Architecture の本質だと思っています。

異論・反論大歓迎

異論・反論大歓迎です。

ぜひご意見を聞かせてください。議論しましょう！

About Me

中村 充志 / Atsushi Nakamura

- リコージャパン株式会社 所属
- Enterprise（おもに金融）系SIerのITアーキテクト
- 最近はプロダクトオーナー風なことも
- 「持続可能なソフトウェア」の探求がライフワーク

- Blog https://zenn.dev/nuits_jp
- Twitter(X) @nuits_jp
- GitHub https://github.com/nuitsjp

Today’s Contents

コンテンツ

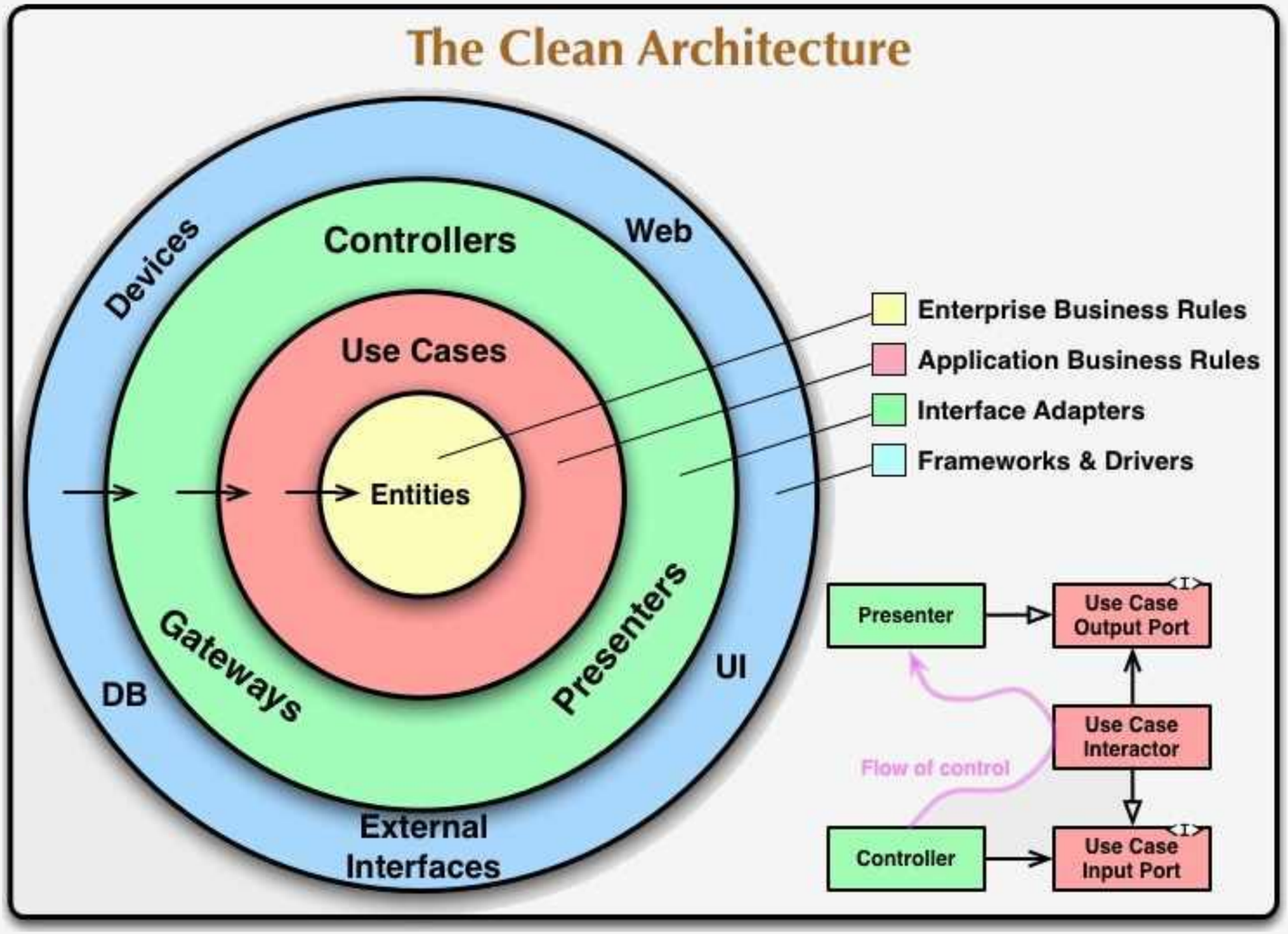
本日お話するプレゼンテーション・原稿・サンプルコードは、すでにconnpassのページにすべて公開しています。

- プレゼンテーション
  https://github.com/nuitsjp/Easiest-Clean-Architecture
- 原稿
  https://zenn.dev/nuits_jp/articles/2025-04-30-easiest-clean-architecture
- サンプルコード
  https://github.com/nuitsjp/Easiest-Clean-Architecture

Easiest Clean Architecture

誤解されがちな二つのこと

Clean Architectureといえば・・・



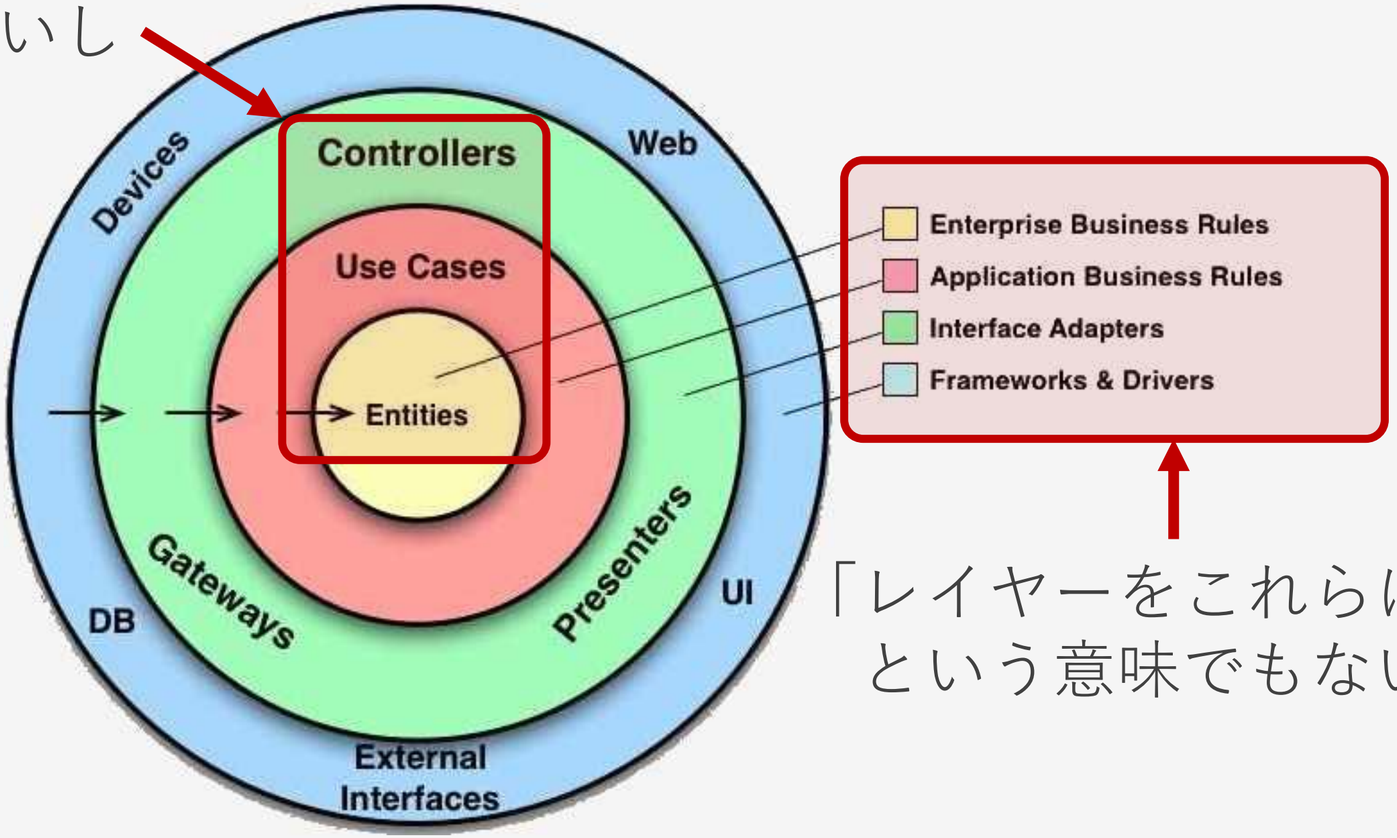
出典：The Clean Architecture
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

この図が良くできてい過ぎて誤解を招く

ひとつ目の誤解

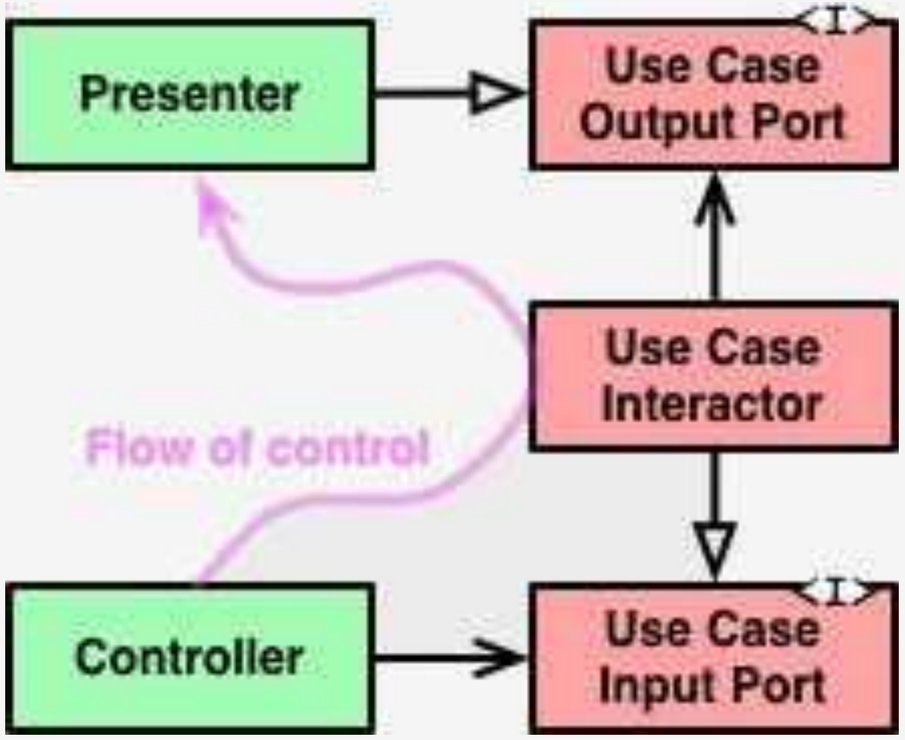
Clean Architectureとは

「関心をControllerやUse Case, Entityに分離しろ」
という意味ではないし

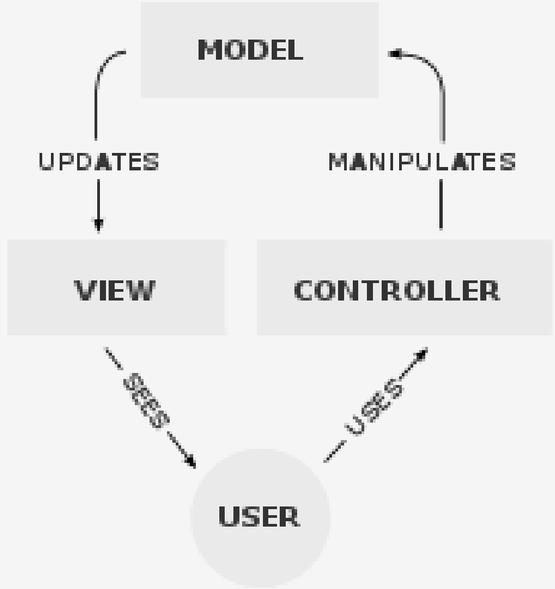


「レイヤーをこれらに分割しろ」
という意味でもない

ふたつ目の誤解



MVC Model（出展Wikipedia）

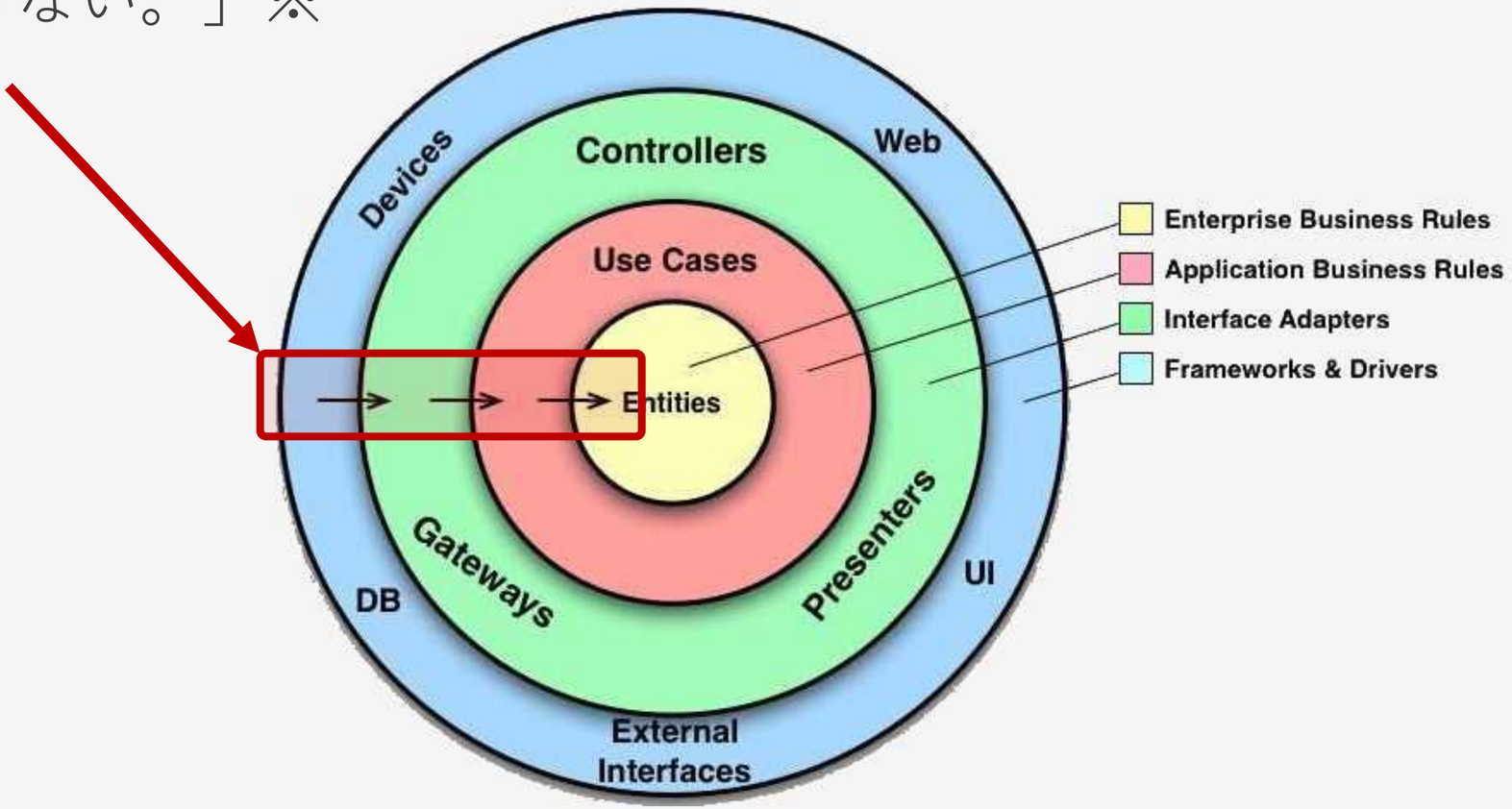


「データや処理の流れを一方通行にしろ」
　という意味ではないし
「PresentationをMVC的に分離しろ（MVVMなどの否定）」
　という意味でもない

では、どういう意味か？

本当に大切なことは一つだけ

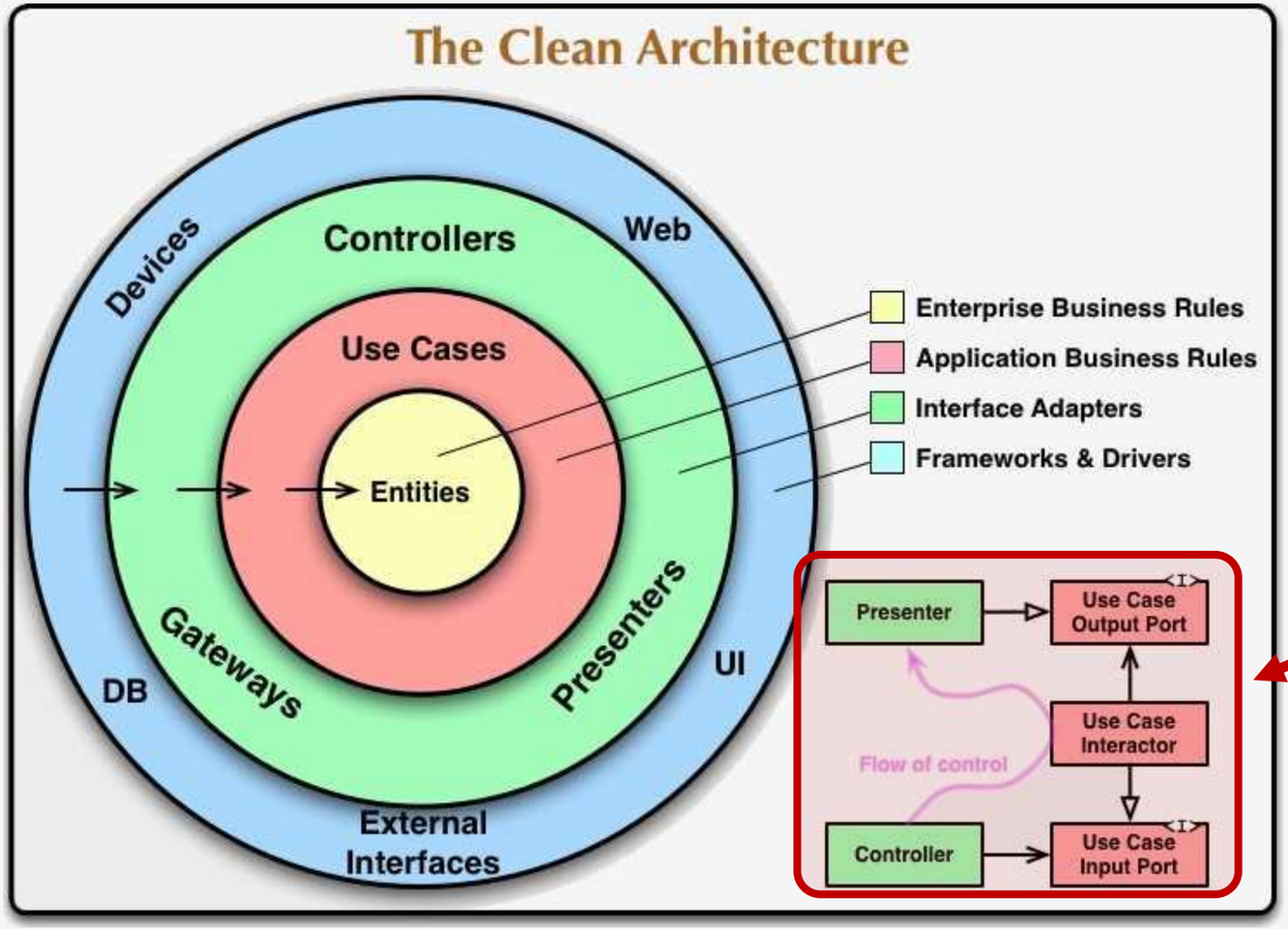
「依存性は、内側（上位レベルの方針）だけに
向かっていなければいけない。」※



※出典：Clean Architecture　達人に学ぶソフトウェアの構造と設計

実現手段の図示

内側の変化に応じて、外側を呼び出したい場合どうすればよいのか？



こうすればよい ※

※ より厳密な意図は後述

それあってますか？

はい（当者比）

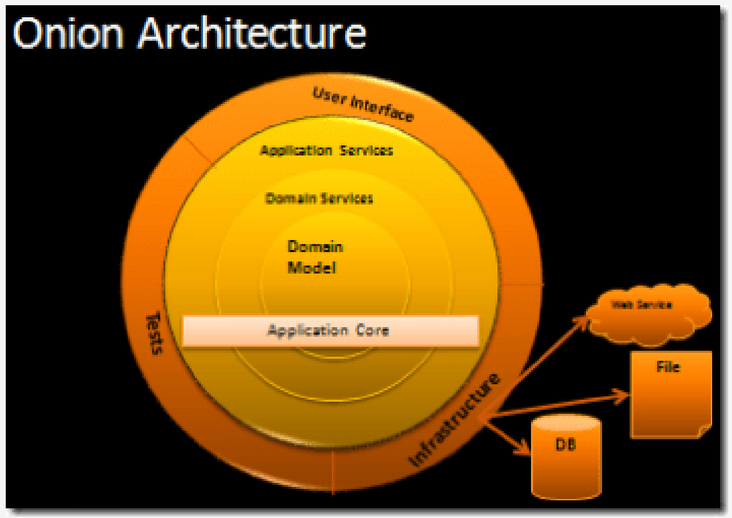
Clean Architecture 発表前夜

当時、以下のようなArchitectureが着目されていました。

- Hexagonal Architecture
- Onion Architecture
- Screaming Architecture
- DCI
- BCE

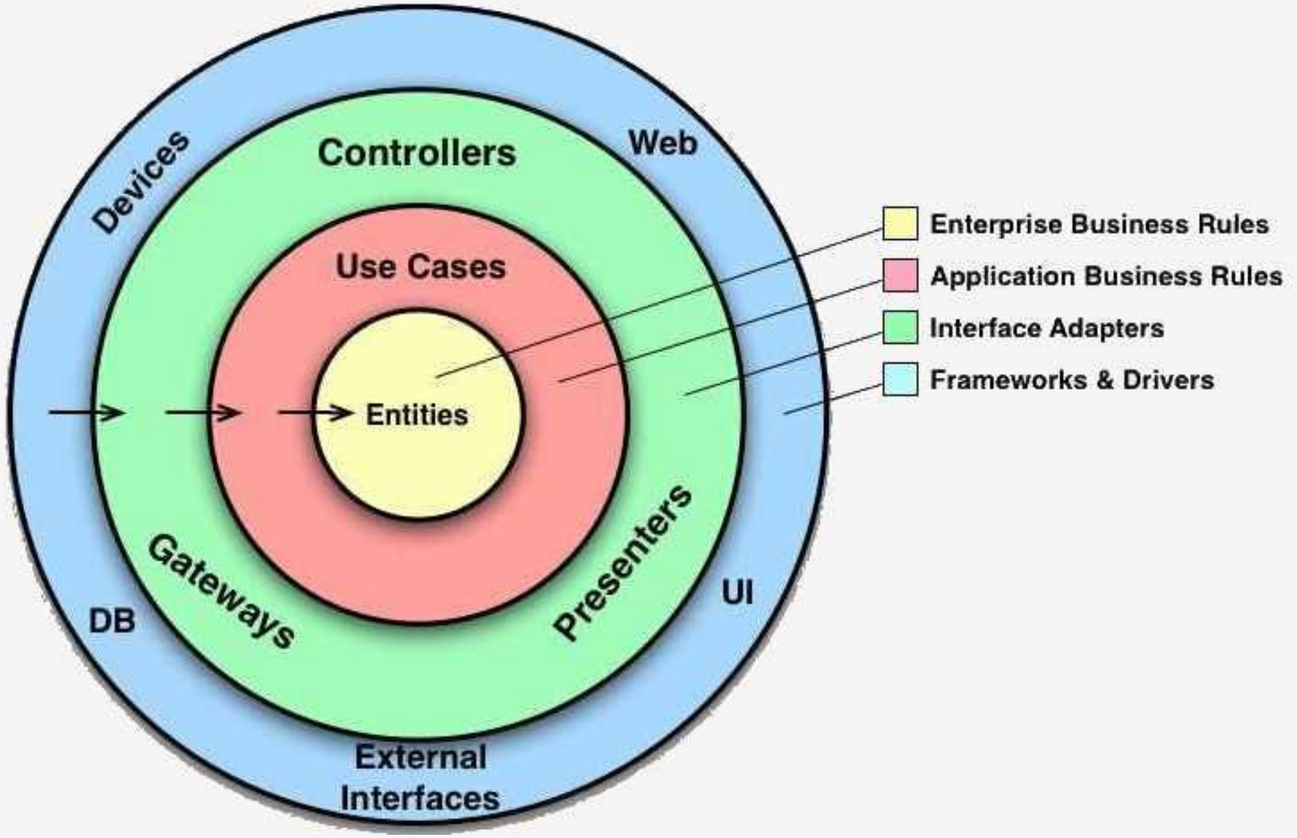
Clean Architectureの図は・・・

Hexagonal architecture



共通する特徴をまとめ、汎化・例示したものにすぎない

汎化



Onion Architecture：https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
Hexagonal architecture：https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)

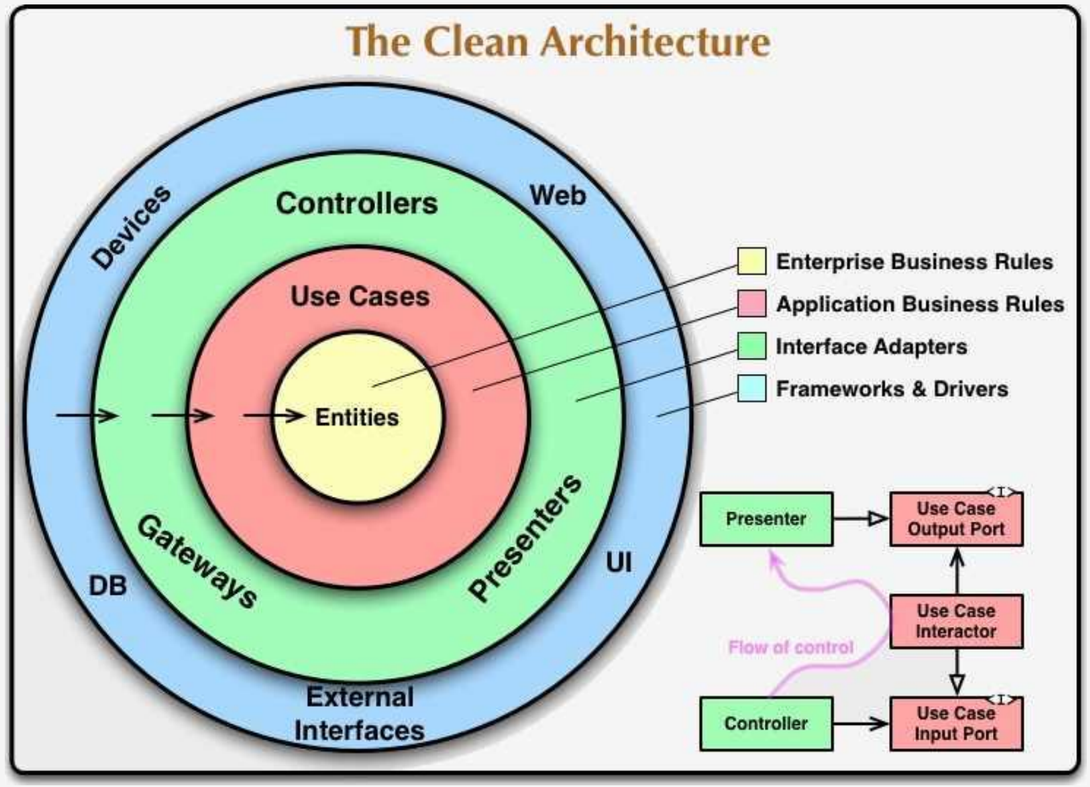
あくまで例示であると取れる記載もある

Clean Architecture説明の冒頭に、著名なアーキテクチャ（Hexagonal, Onion, etc...）には類似の共通点が見られるとある。

そして続けてこう記載されている。

> The diagram in Figure is an attempt at integrating all these architectures into a single actionable idea.
> （邦訳：図は、これらのすべてのアーキテクチャを単一の実行可能なアイデアに統合したものである。）

すべてが同一アーキテクチャに帰結するとすれば



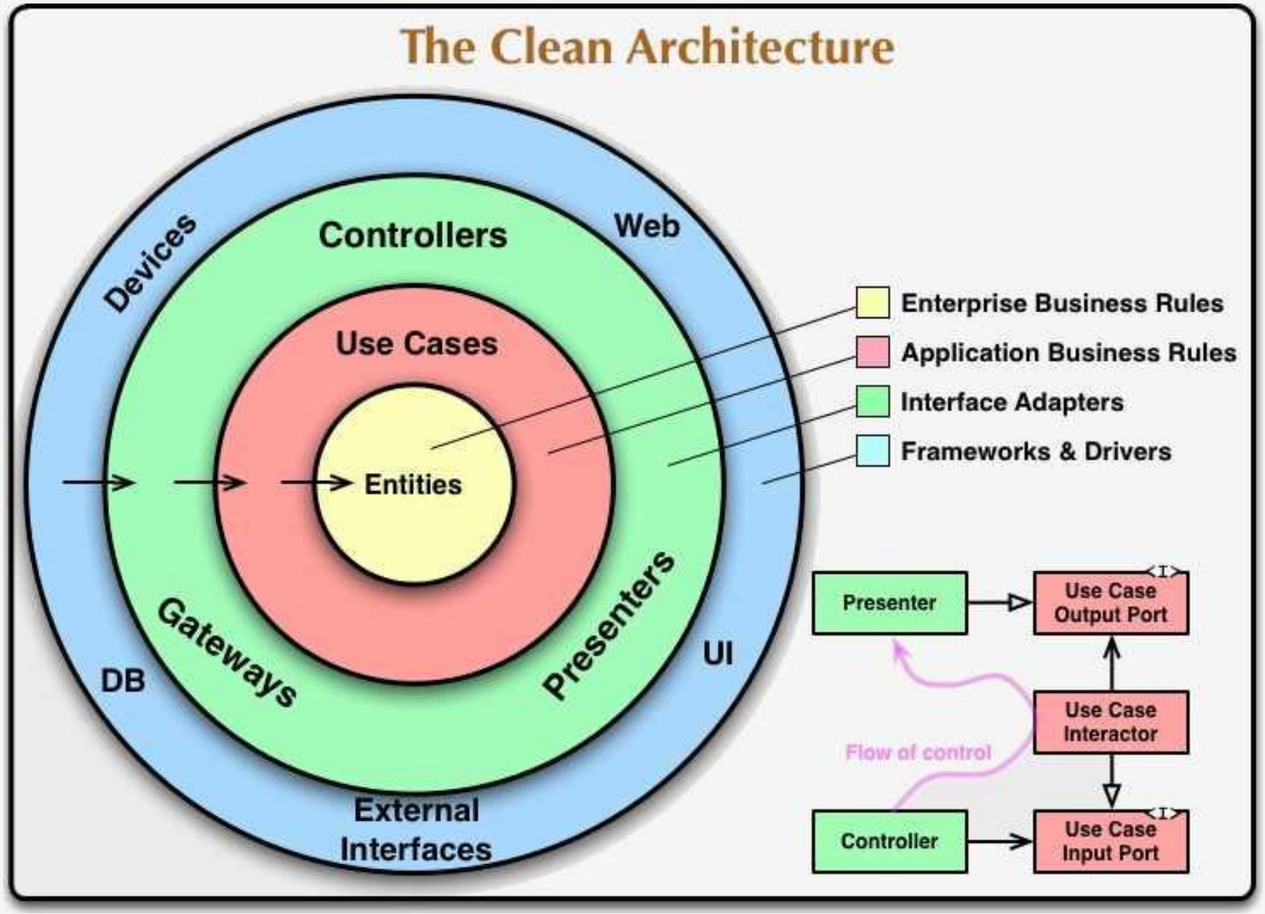
以下は、あくまで一例であることは自明

- レイヤーが4つ※1
- UIがMVC的である
- Web限定
- Output PortはUse Caseとの間「だけ」にある※2
- などなど

※1 とは限らないと書籍にも明記されている

※2 eventみたいなPub/SubがUse Case限定じゃ困っちゃう

あの図は・・・



全てのアーキテクチャはClean Architectureで説明可能であることを例示したモデル。
Clean Architectureを採用したソフトウェアの具体例の一つにすぎない

著者は明言しています

- The architecture rules are the same! -

（邦訳：アーキテクチャのルールはどれも同じである！）

誤解を恐れず言えば・・・

誤解を恐れず言えば・・・

- 「あの図」の構成要素に本質的な価値はない
- 本当に価値があるのは以下である
  - 「あの図」が実現しようとした目的
  - 「あの図」に至った背景
  - 「あの図」を実現する手段

誤解を恐れず言えば・・・

- 「あの図」の構成要素に本質的な価値はない
- 本当に価値があるのは以下である
  - 「あの図」が実現しようとした目的
  - 「あの図」に至った背景
  - 「あの図」を実現する手段

目的とは

以下のような状態を実現すること

1. フレームワーク非依存
2. テスト可能
3. UI非依存
4. データベース非依存
5. 外部エージェント非依存

一読の価値あり

目的・背景・手段
が凝縮されている

Slide 33

Easiest Clean Architecture

おさえるべき三つのこと

おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ

おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ

2. 制御の流れと依存方向は分離しコントロールせよ

おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ
2. 制御の流れと依存方向は分離しコントロールせよ
3. 上位レベルとは相対的・再帰的であることに留意せよ

さあ具体例を見てみよう！

アプリケーション「HatPepper」※

- 位置情報から周辺の店舗を検索する「コンソールアプリケーション」
- 「リクルートWebサービス」の「グルメサーチAPI」を利用させていただく
（いつもお世話になっております。）

No.	店名	ジャンル
1	天ぷら日本酒天喜代 東京駅店	居酒屋
2	手打ちそば みや川 GRANSTA八重北店	和食
3	味の牛たん喜助 東京駅八重洲北口店	焼肉・ホルモン
4	日式台湾食堂WUMEI	中華
5	うなぎ四代目菊川 東京駅黒塀横丁店	和食
6	はせがわ酒店 東京駅グランスタ店	その他グルメ
7	北出タコス グランスタ東京店	各国料理
8	Japanese Malt Whisky SAKURA グランスタ東京店	バー・カクテル
9	鉄板 お好み焼き 電光石火 東京駅店	お好み焼き・もんじゃ
10	ニユートーキヨー ビヤホール 東京駅八重洲口店	ダイニングバー・バル

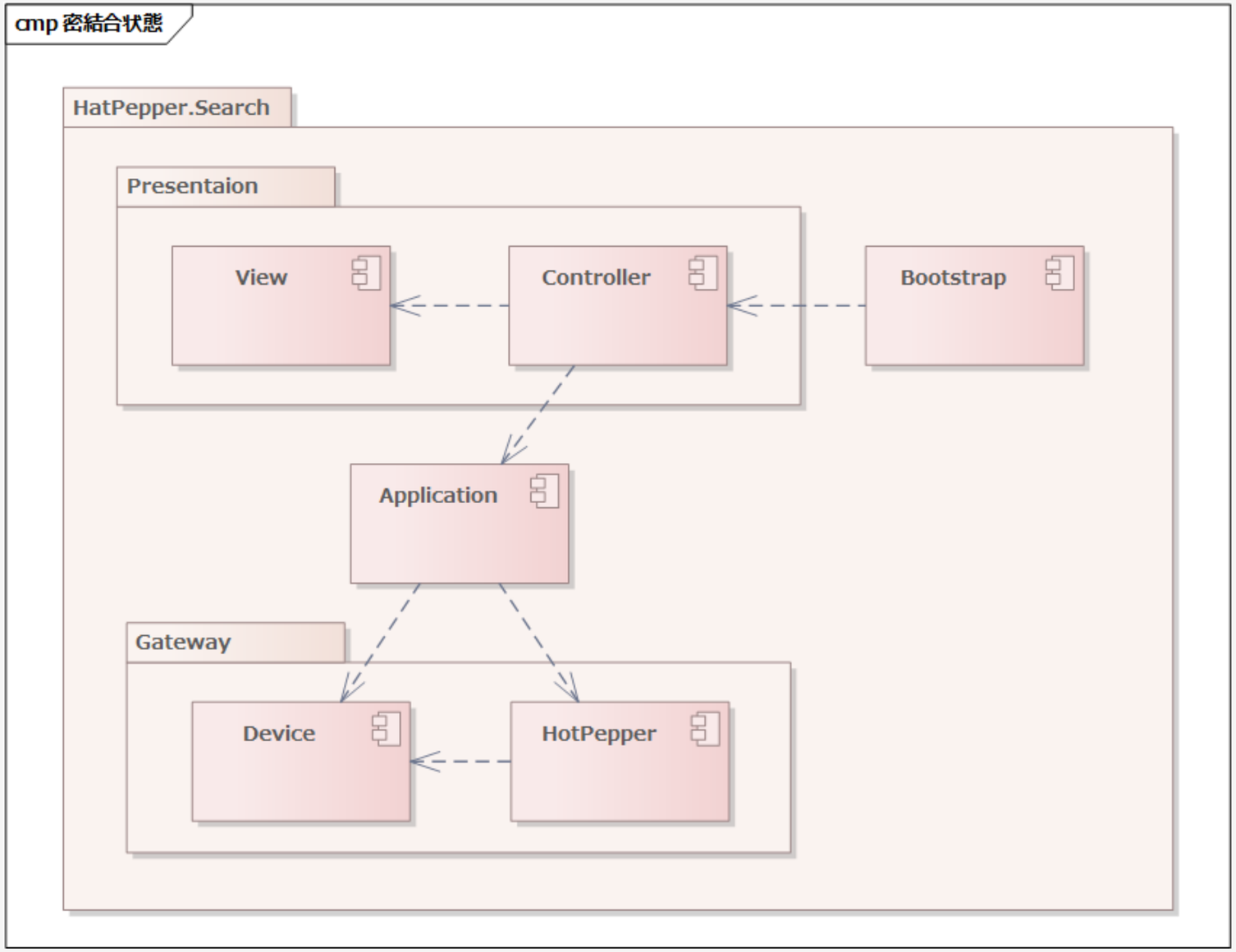
※「ホットペッパー」は株式会社リクルート様の登録商標です。

※「HatPepper」は登録商標ではありません。安心。

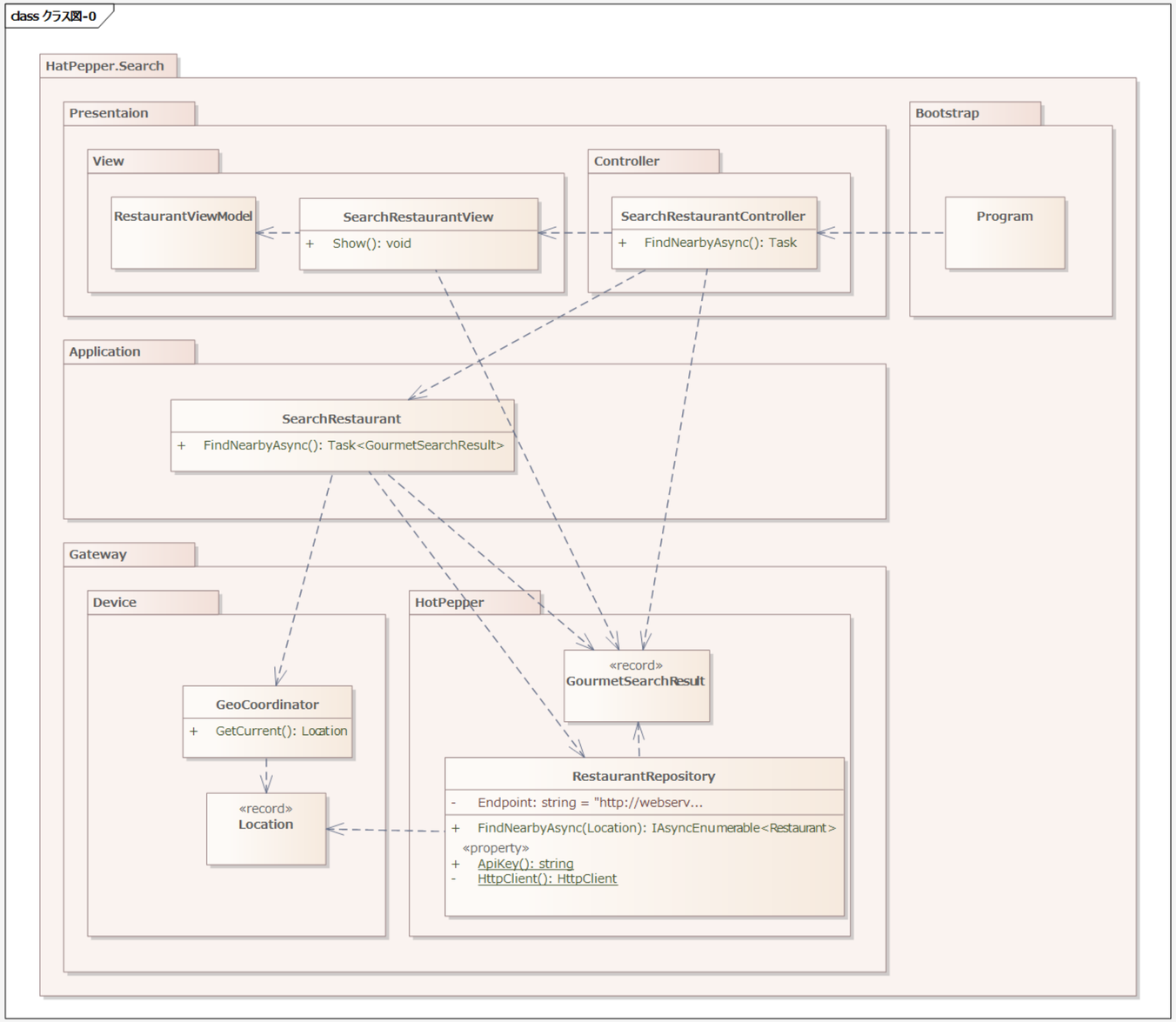
Easiest Clean Architecture

Layered Architecture

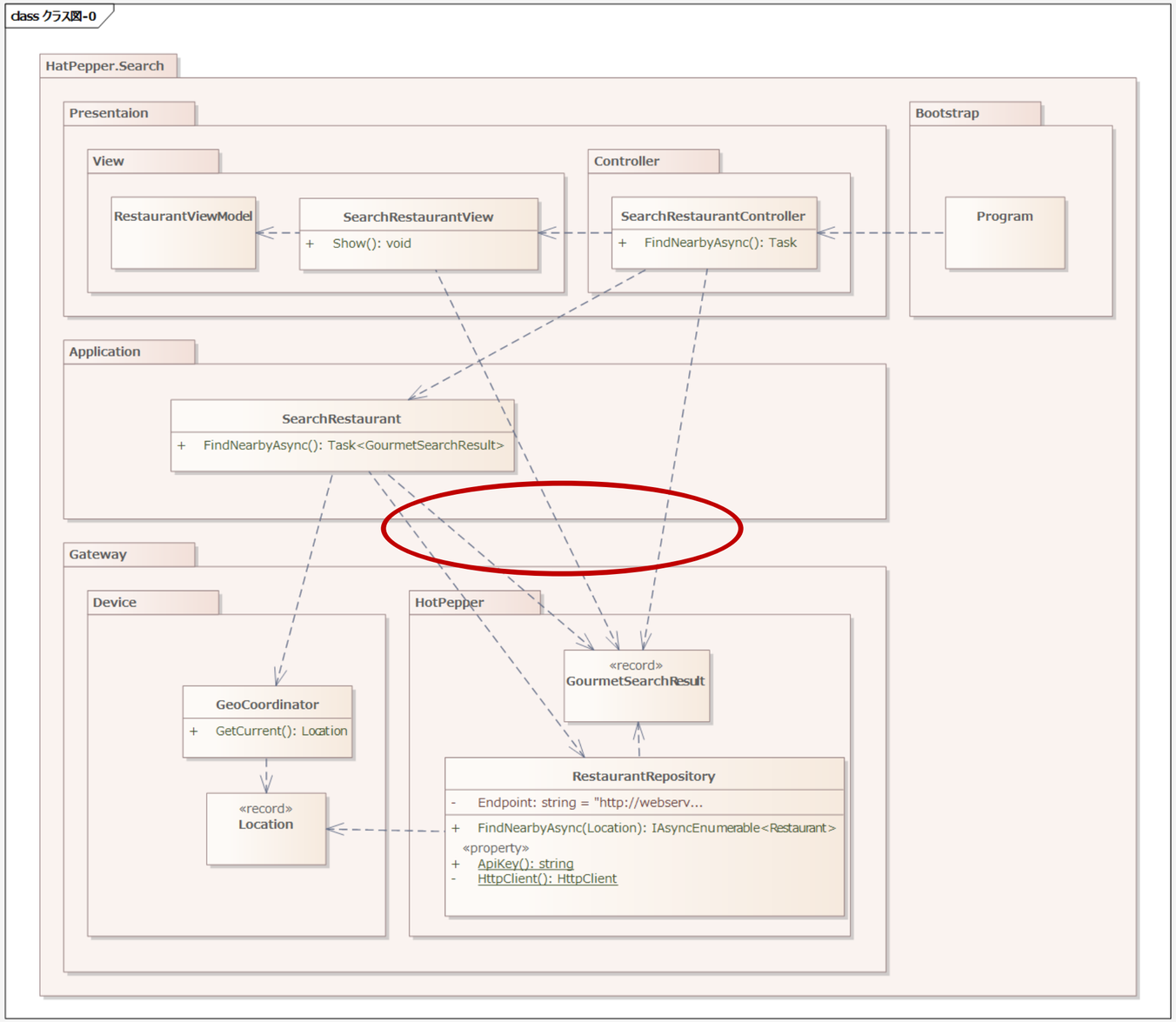
コンポーネント図



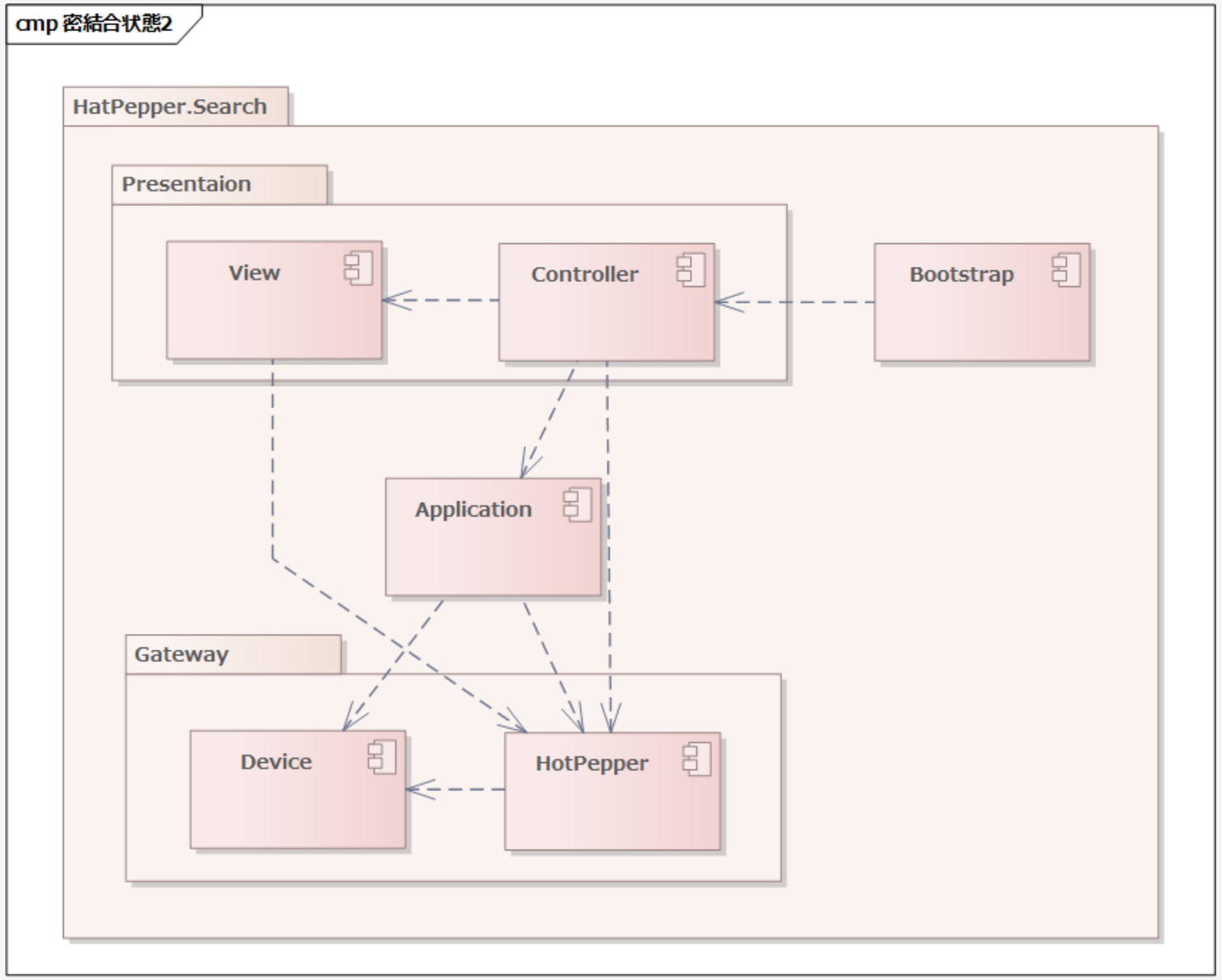
クラス図



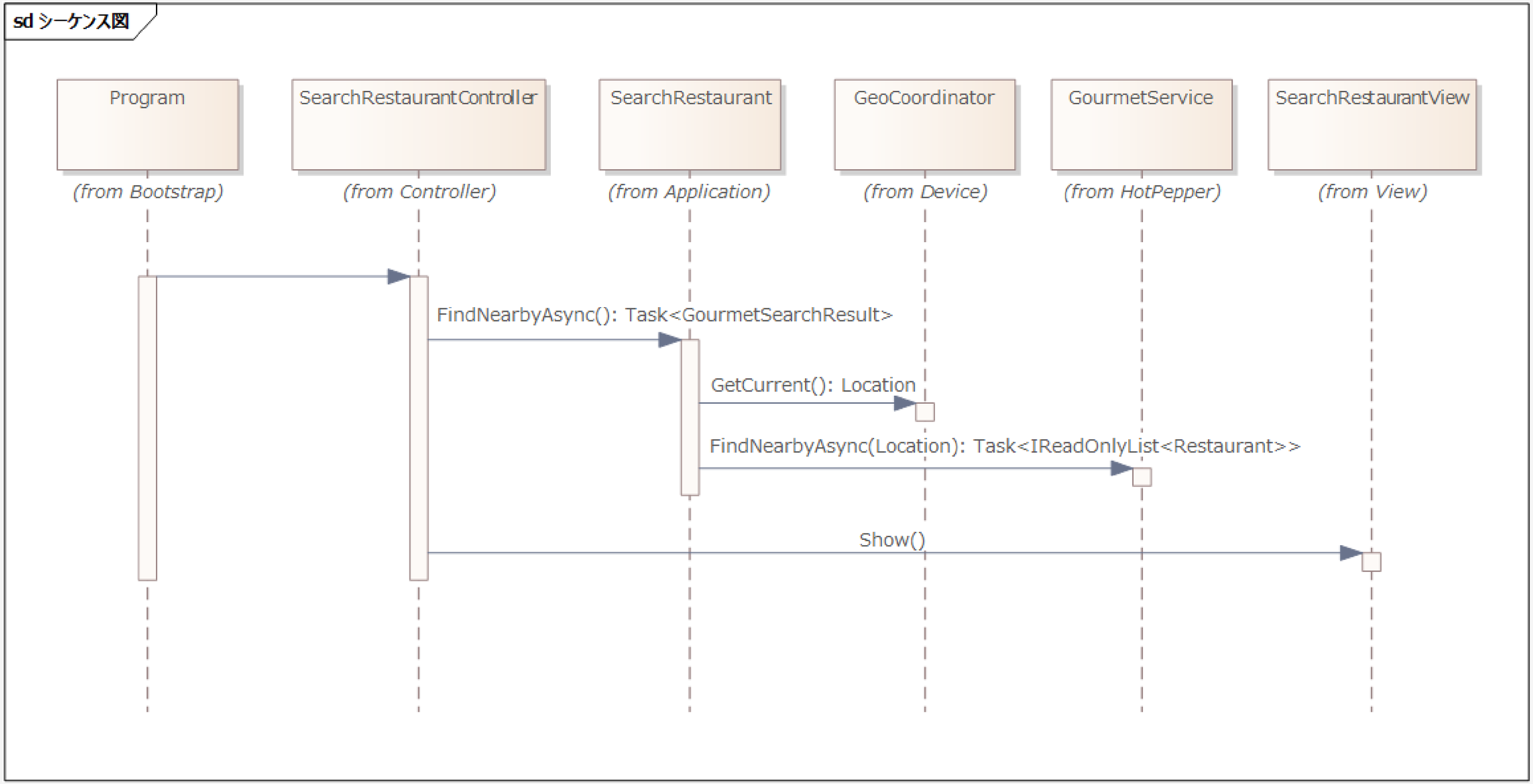
クラス図



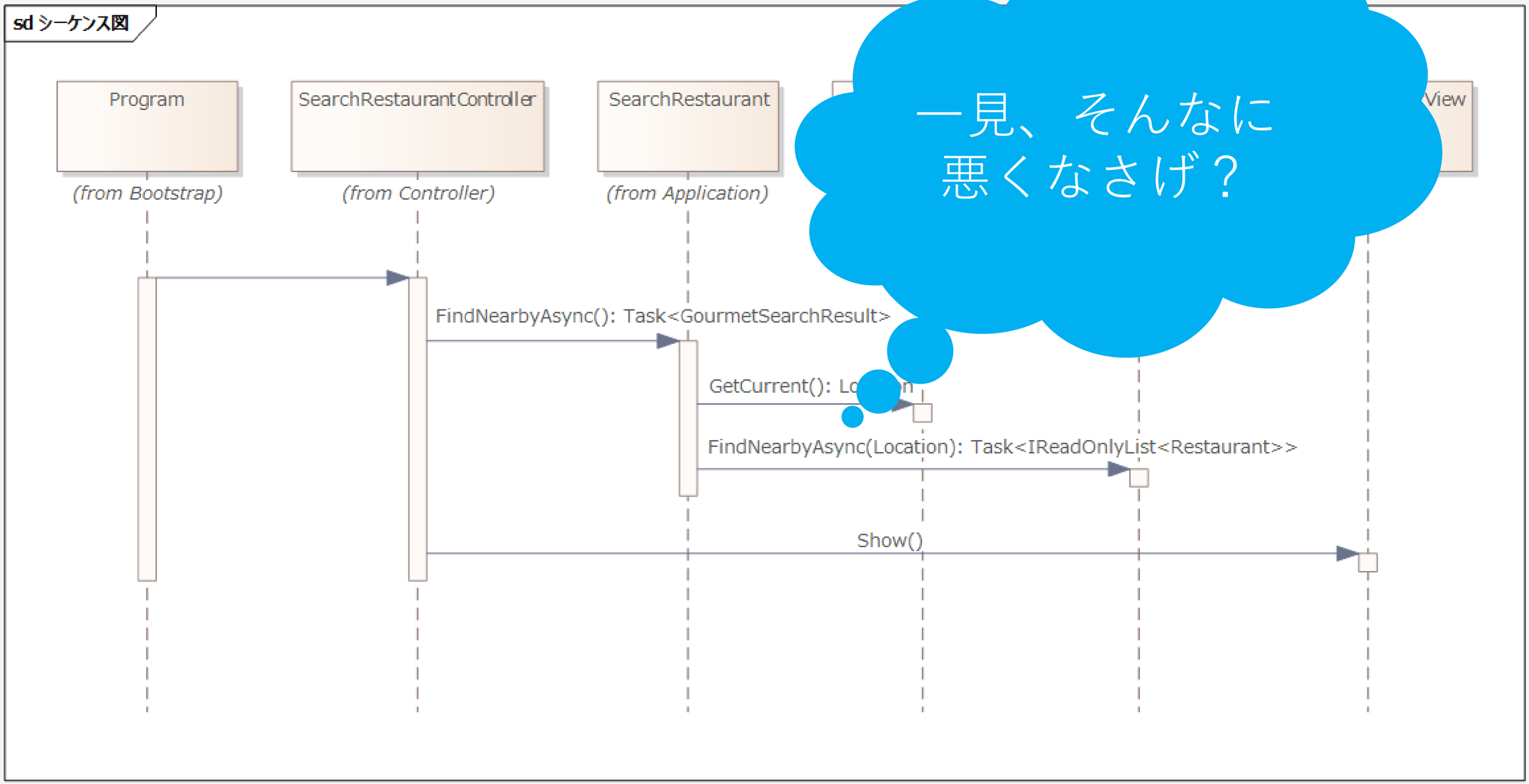
クラス図



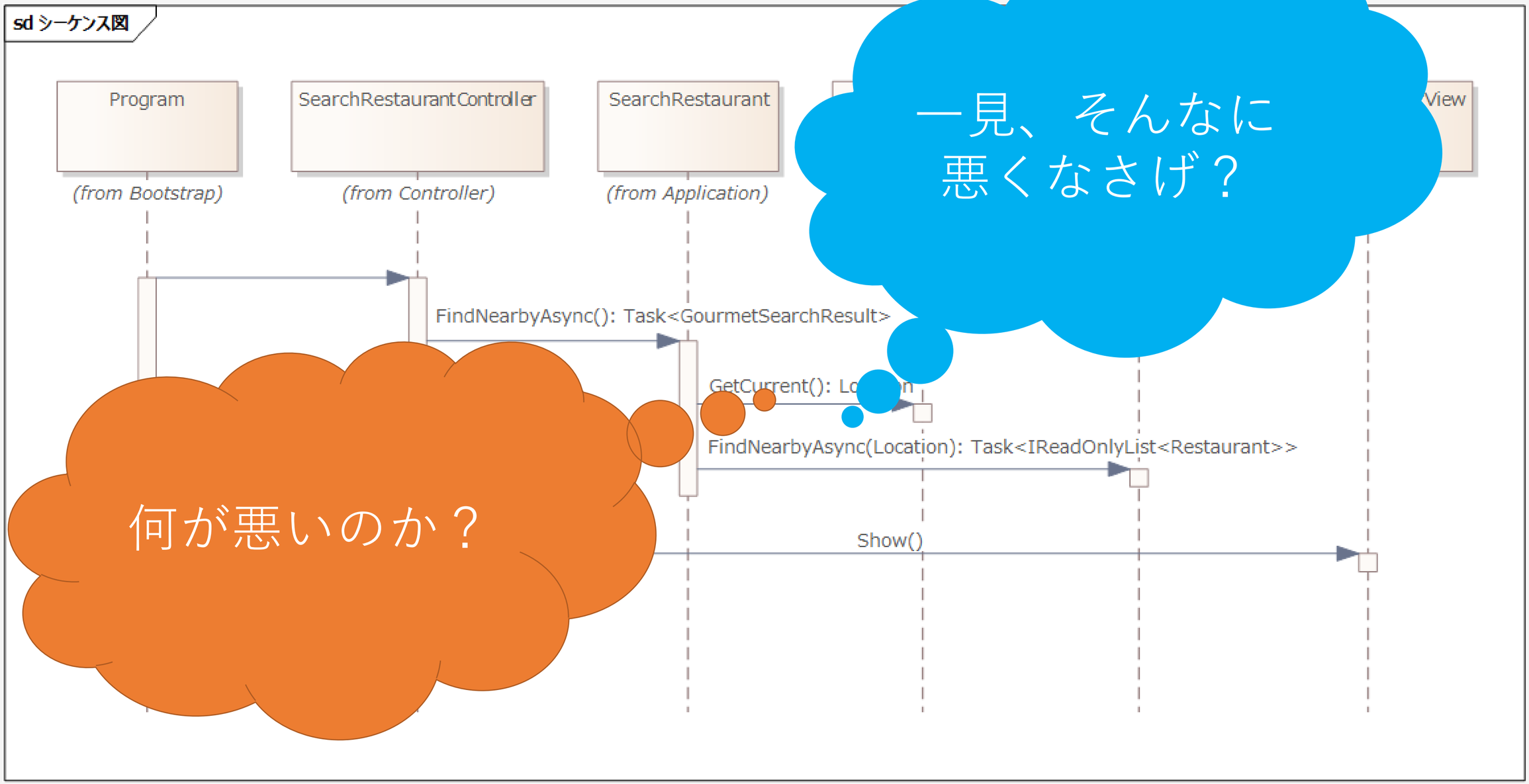
シーケンス図



シーケンス図

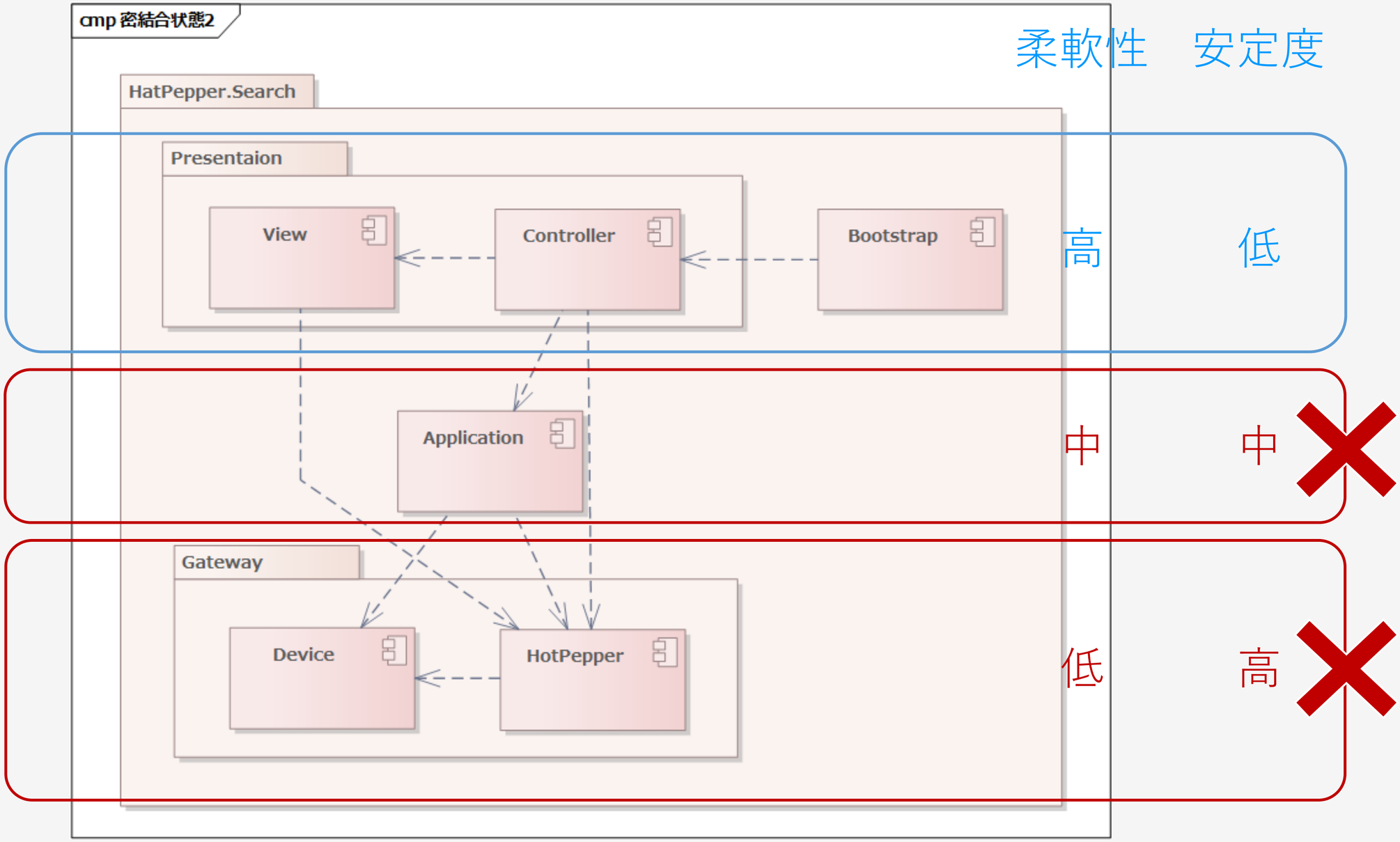


シーケンス図



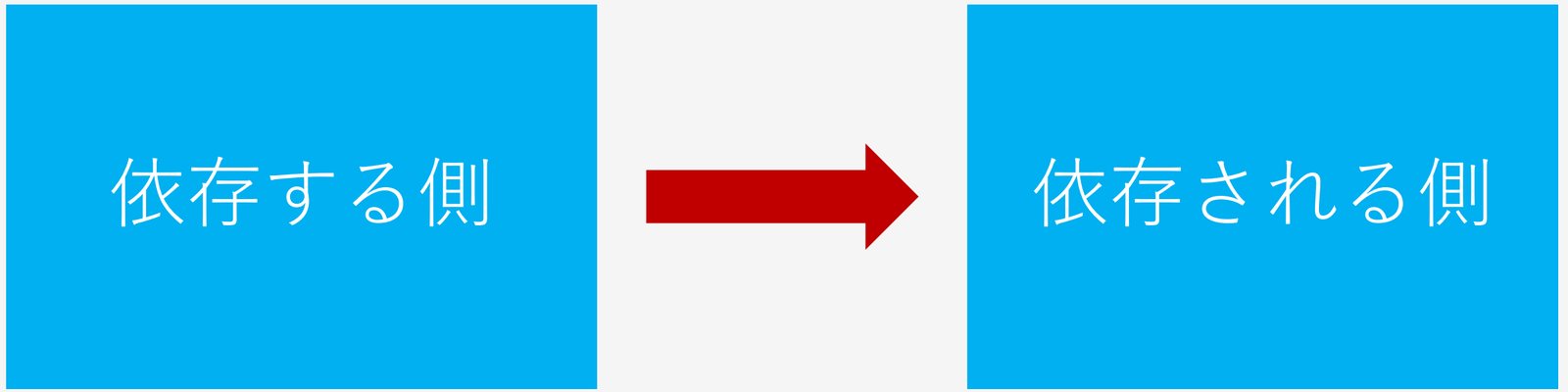
安定度と柔軟性のバランス配分に問題あり

柔軟性と安定度



依存関係による安定度と柔軟性のトレードオフ

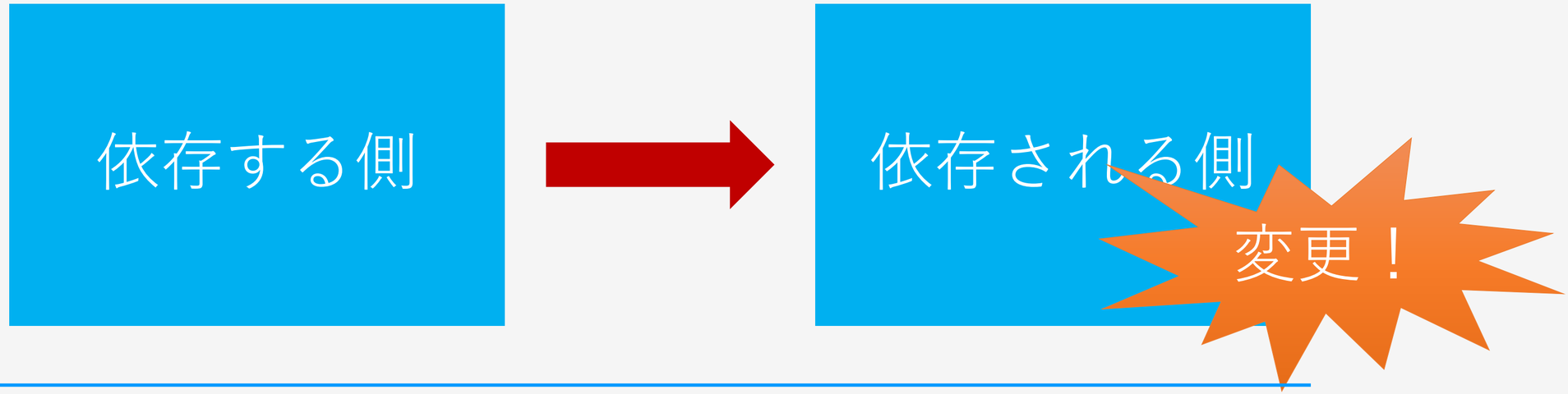
依存関係による安定度と柔軟性のトレードオフ



安定度

柔軟性

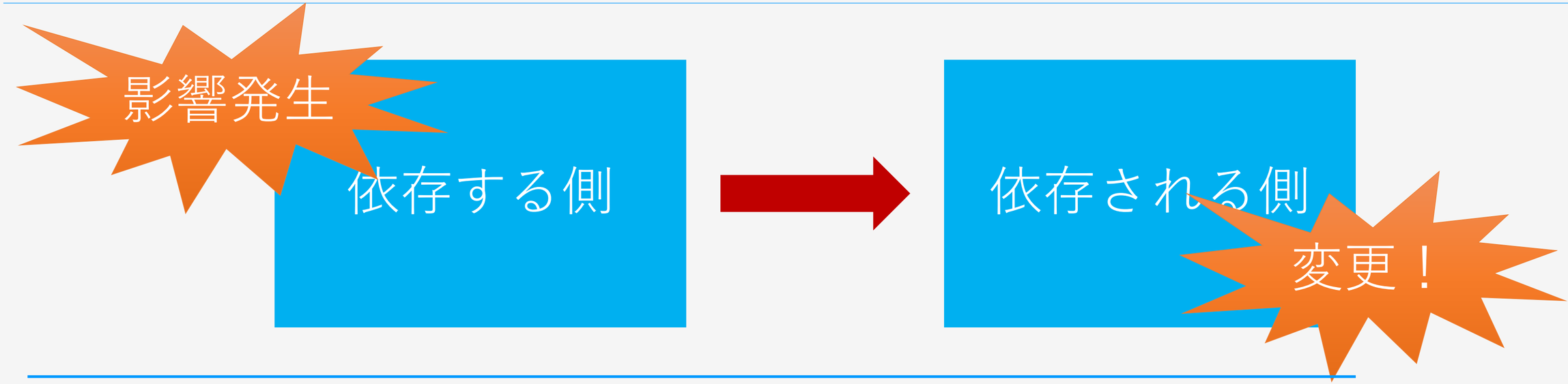
依存関係による安定度と柔軟性のトレードオフ



安定度

柔軟性

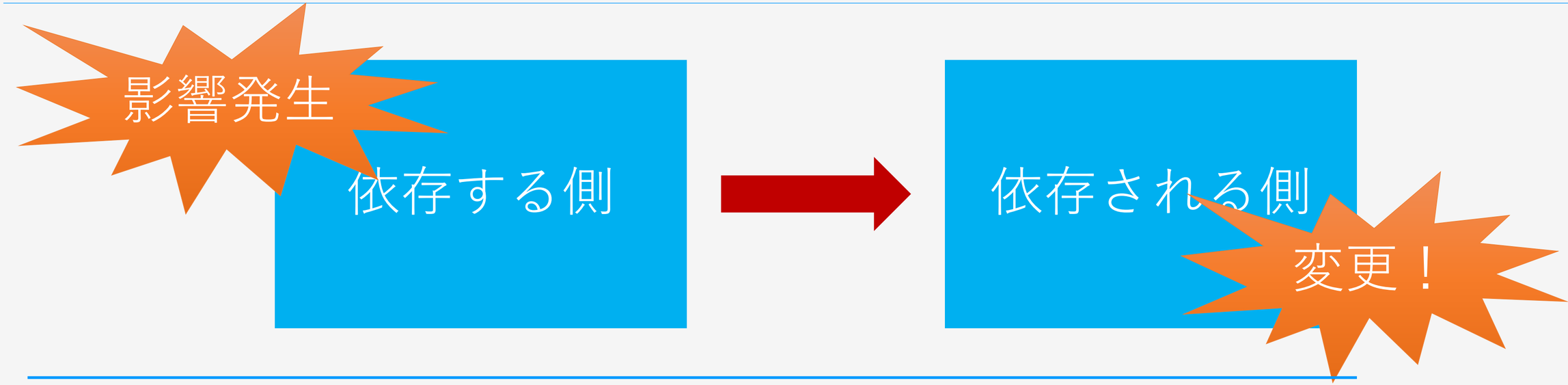
依存関係による安定度と柔軟性のトレードオフ



安定度

柔軟性

依存関係による安定度と柔軟性のトレードオフ

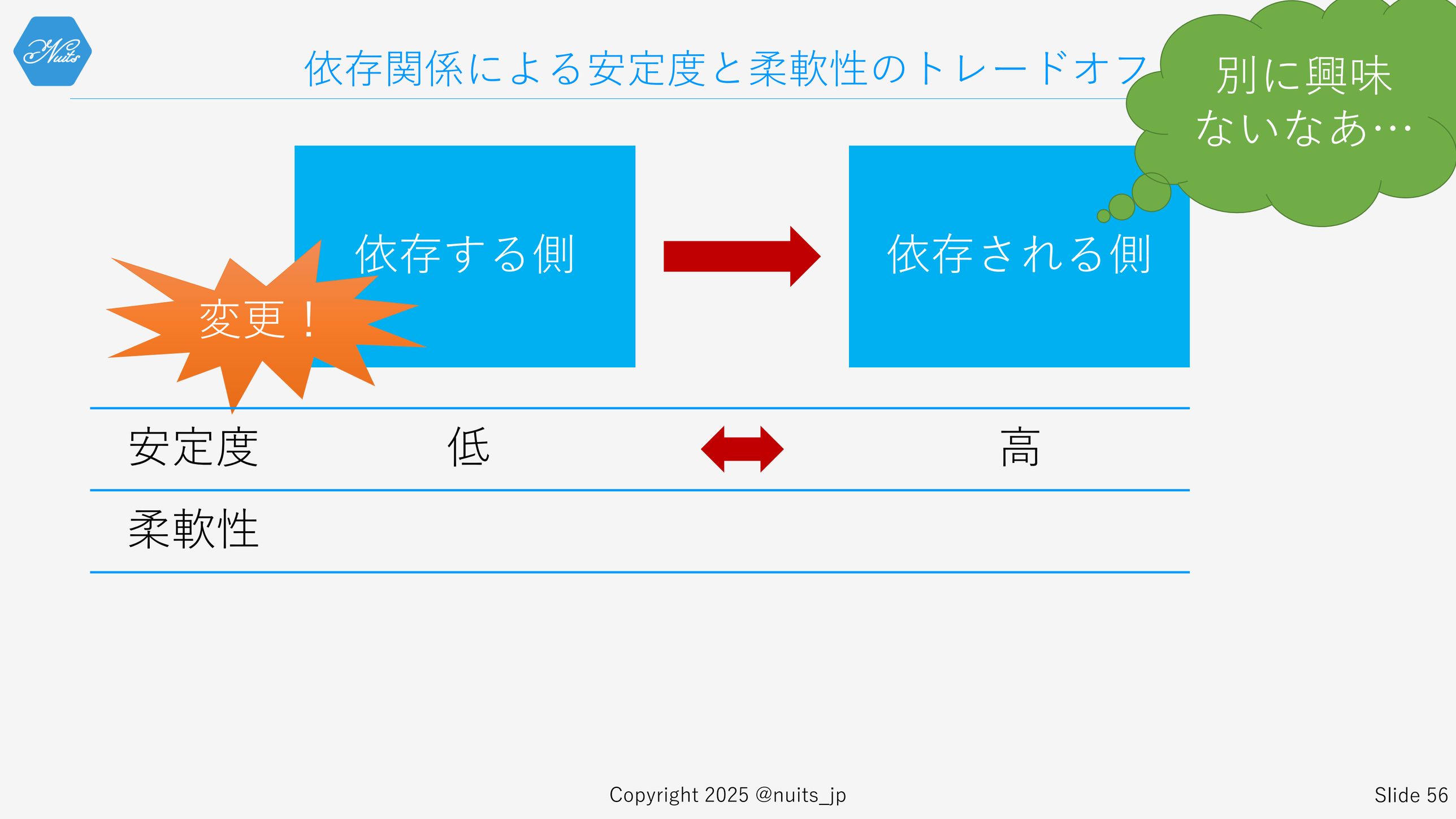


安定度	低	⟷	高
柔軟性			

依存関係による安定度と柔軟性のトレードオフ

安定度	低	⟷	高
柔軟性			

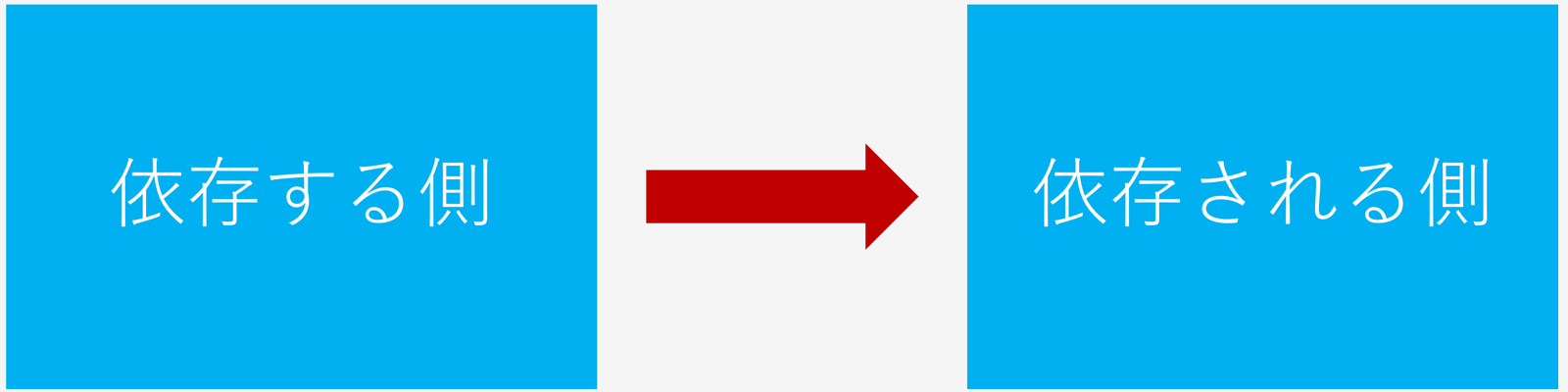
依存関係による安定度と柔軟性のトレードオフ



依存関係による安定度と柔軟性のトレードオフ

安定度	低	⟷	高
柔軟性	高	⟷	低

依存関係による安定度と柔軟性のトレードオフ

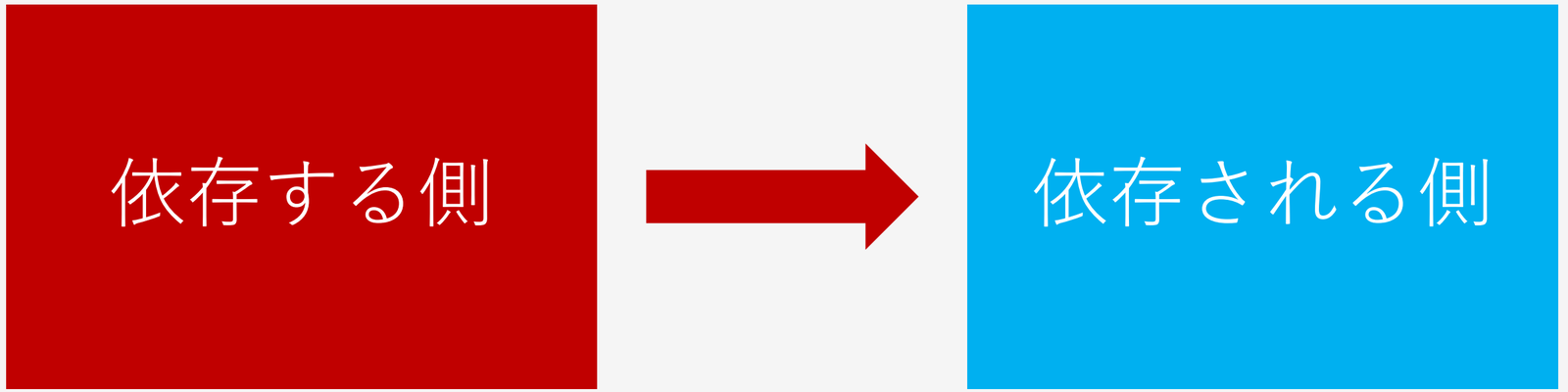


安定度	低	⟷	高
柔軟性	高	⟷	低

安定度と柔軟性は
トレードオフ！

逆説的に考えると・・・

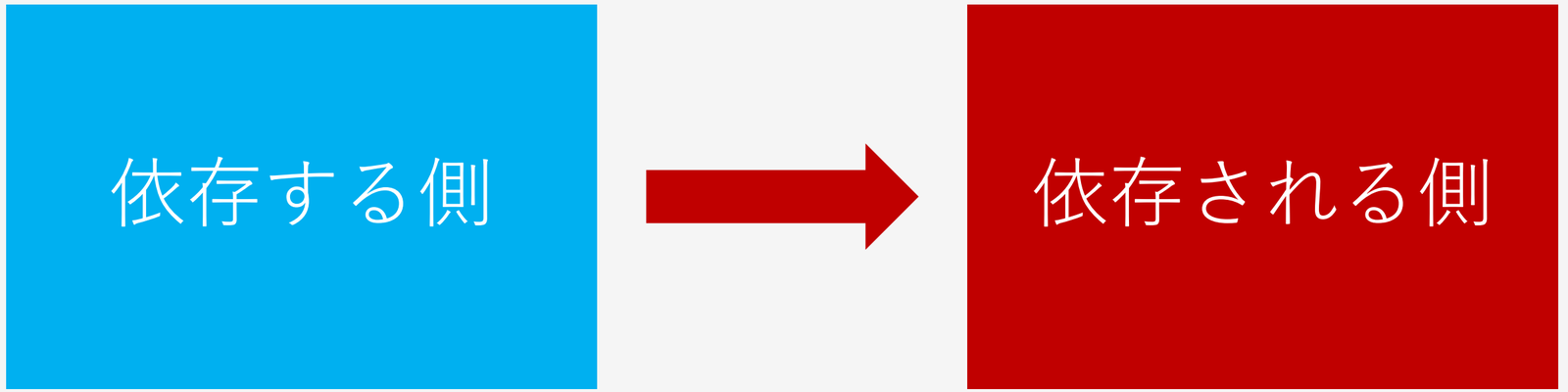
依存関係による安定度と柔軟性のトレードオフ



安定度	低	⟷	高
柔軟性	高	⟷	低

変更頻度の高いオブジェクトは、依存する側に設計すべし

依存関係による安定度と柔軟性のトレードオフ

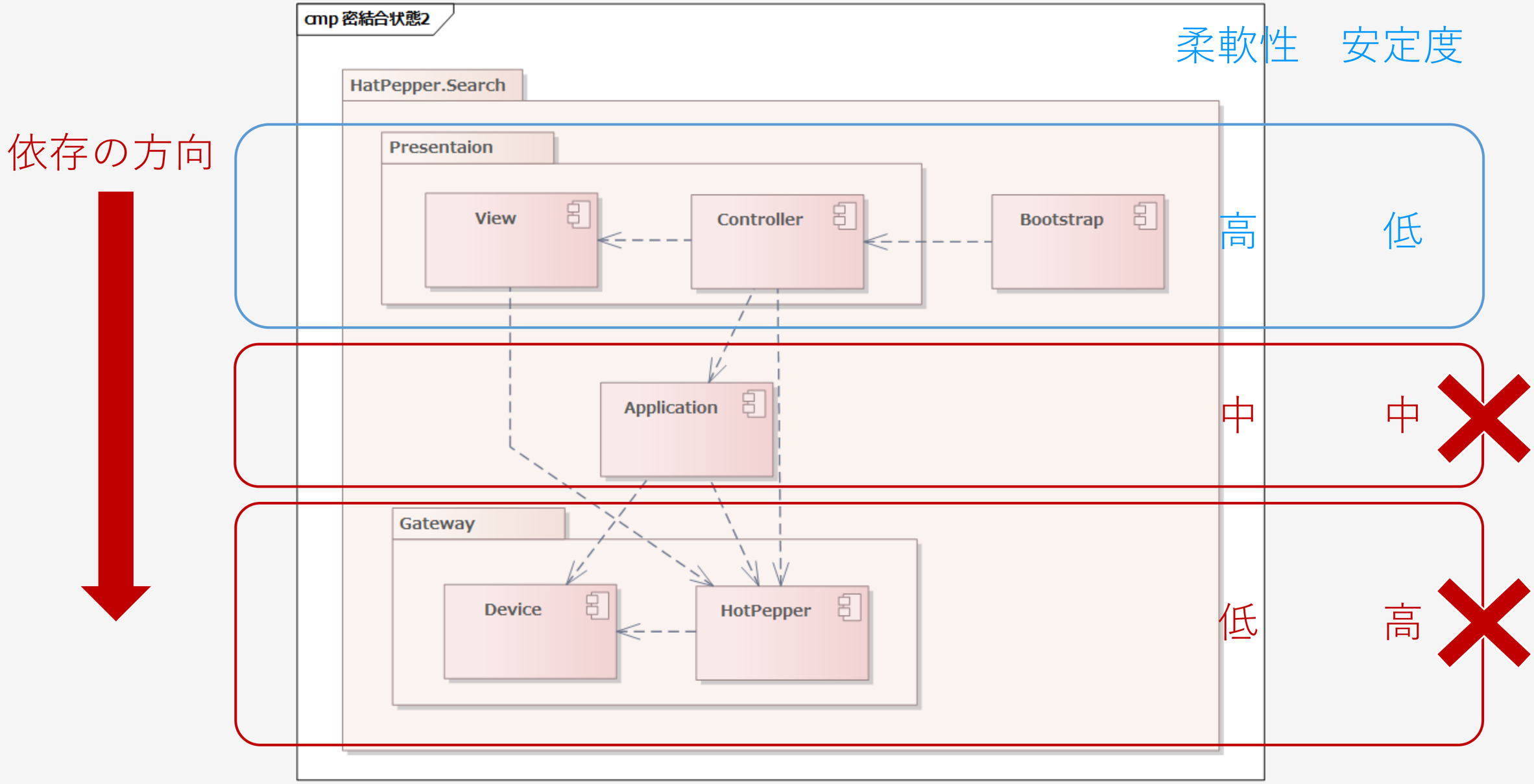


安定度	低	⟷	高
柔軟性	高	⟷	低

アプリケーションの本質的な価値を提供するオブジェクトは、軽微な変更の影響を受けてはいけない。
つまり依存される側に設計すべし。

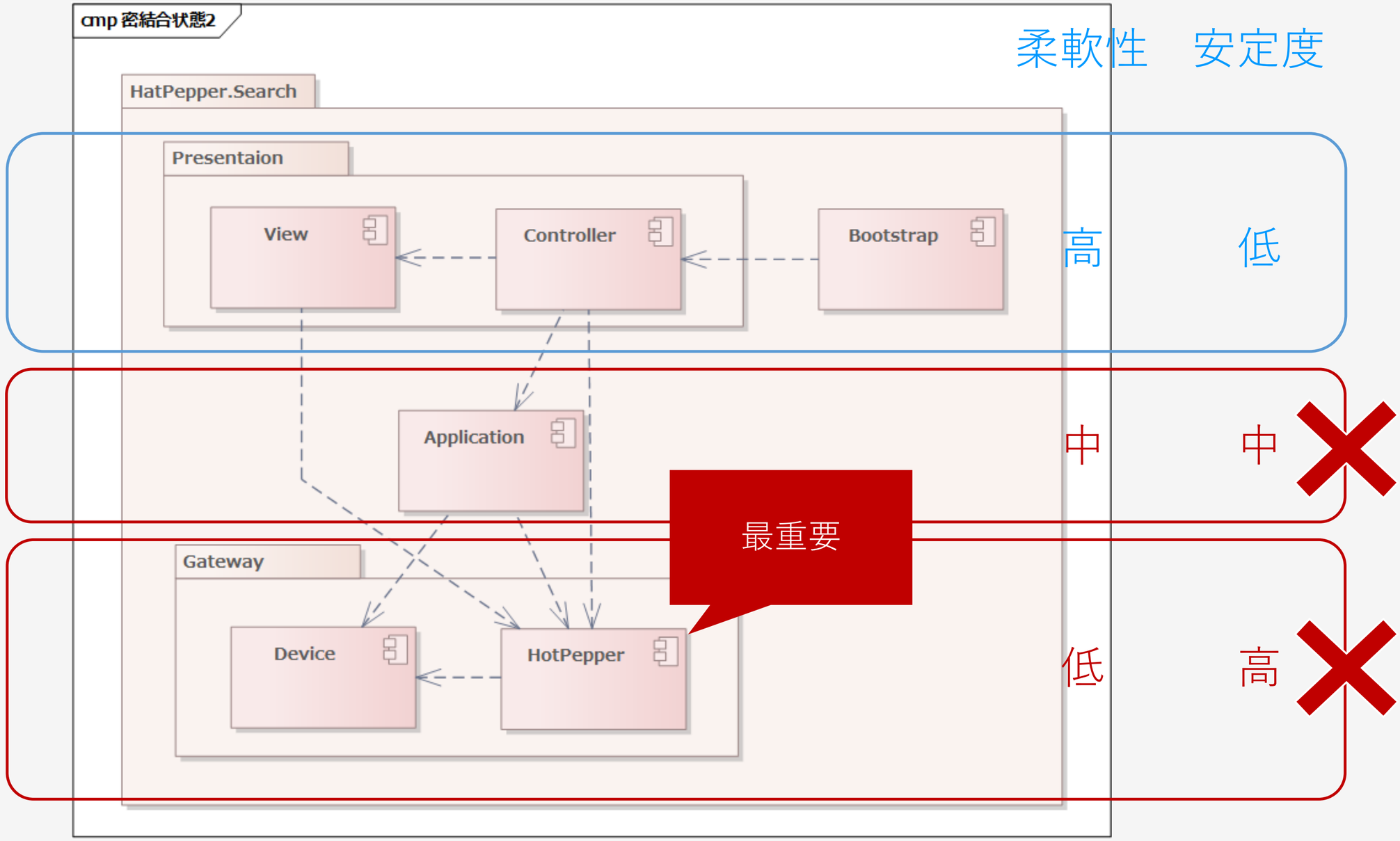
さて、振り返ってみよう

柔軟性と安定度



どこの柔軟性がもっとも重要ですか？

柔軟性と安定度



このアプリケーションの本質的な価値とは？

アプリケーション「HatPepper」※

- **位置情報から周辺の店舗を検索する**
  「コンソ　アプリケーション」
-

これこそが価値の本質

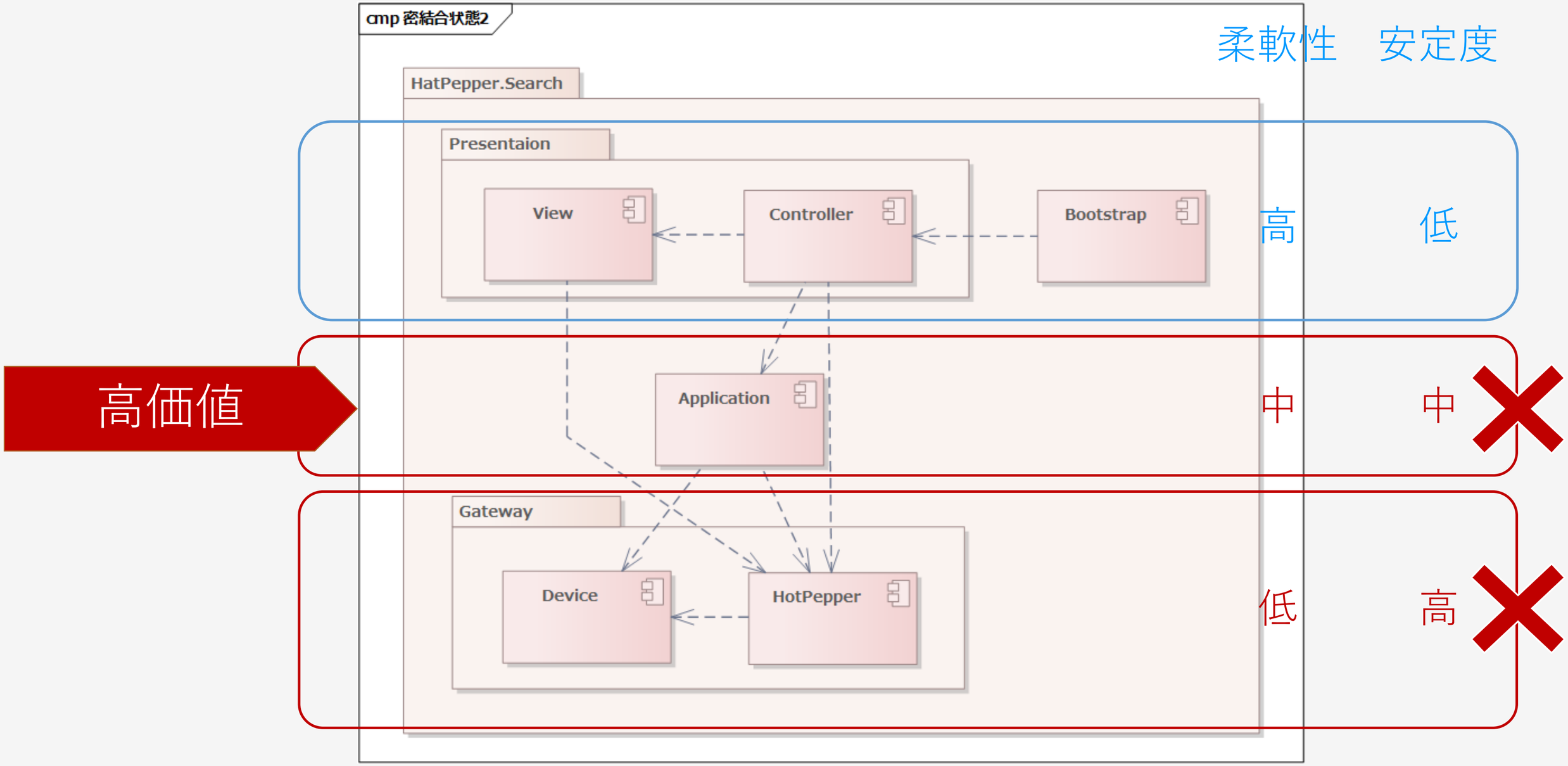
ってい

（いつもお世話になっております。）

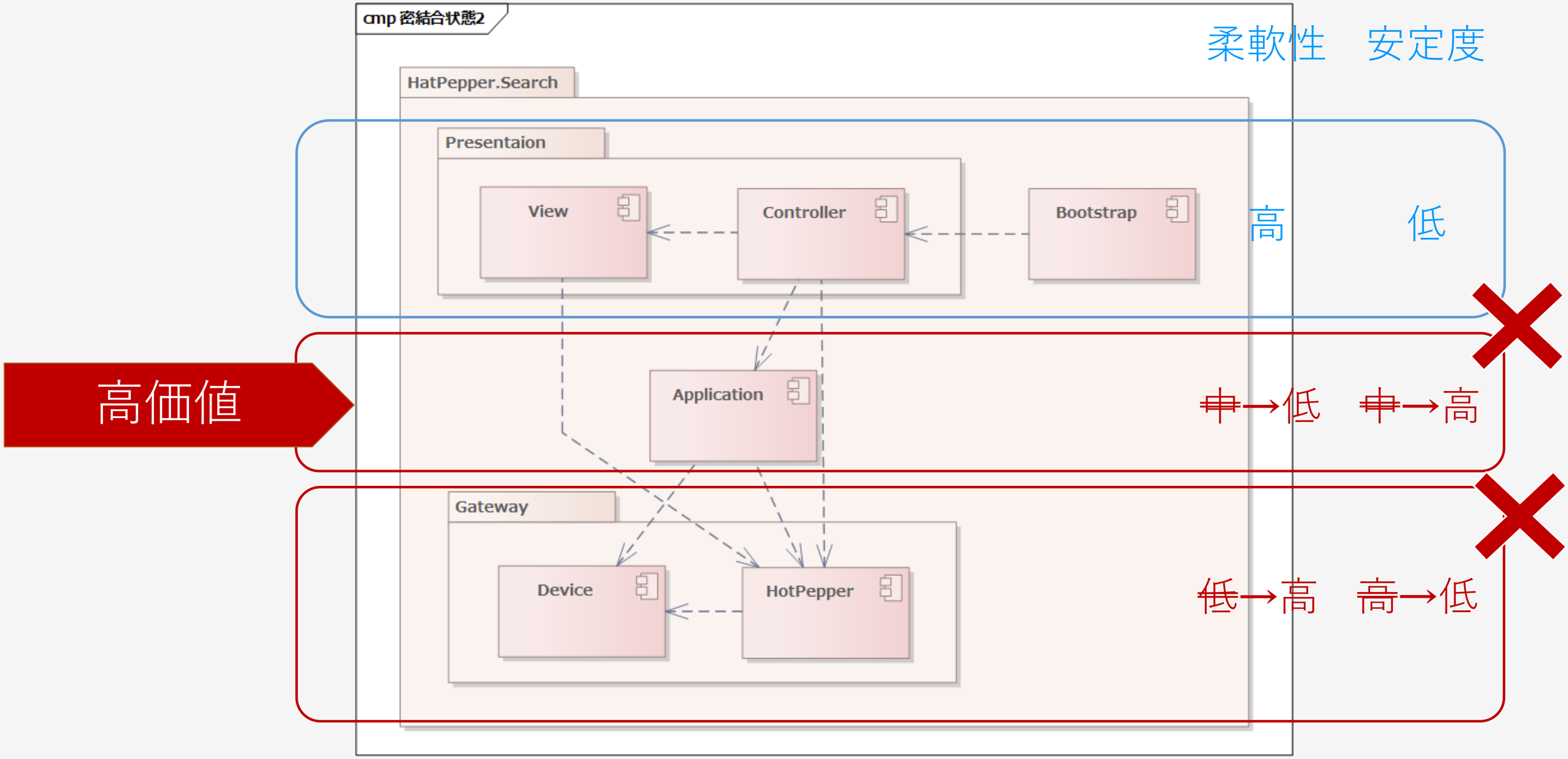
No.	店名	ジャンル
1	天ぷら日本酒天喜代 東京駅店	居酒屋
2	手打ちそば みや川 GRANSTA八重北店	和食
3	味の牛たん喜助 東京駅八重洲北口店	焼肉・ホルモン
4	日式台湾食堂WUMEI	中華
5	うなぎ四代目菊川 東京駅黒塀横丁店	和食
6	はせがわ酒店 東京駅グランスタ店	その他グルメ
7	北出タコス グランスタ東京店	各国料理
8	Japanese Malt Whisky SAKURA グランスタ東京店	バー・カクテル
9	鉄板 お好み焼き 電光石火 東京駅店	お好み焼き・もんじゃ
10	ニユートーキヨー ビヤホール 東京駅八重洲口店	ダイニングバー・バル

※「ホットペッパー」は株式会社リクルート様の登録商標です。
※「HatPepper」は登録商標ではありません。安心。

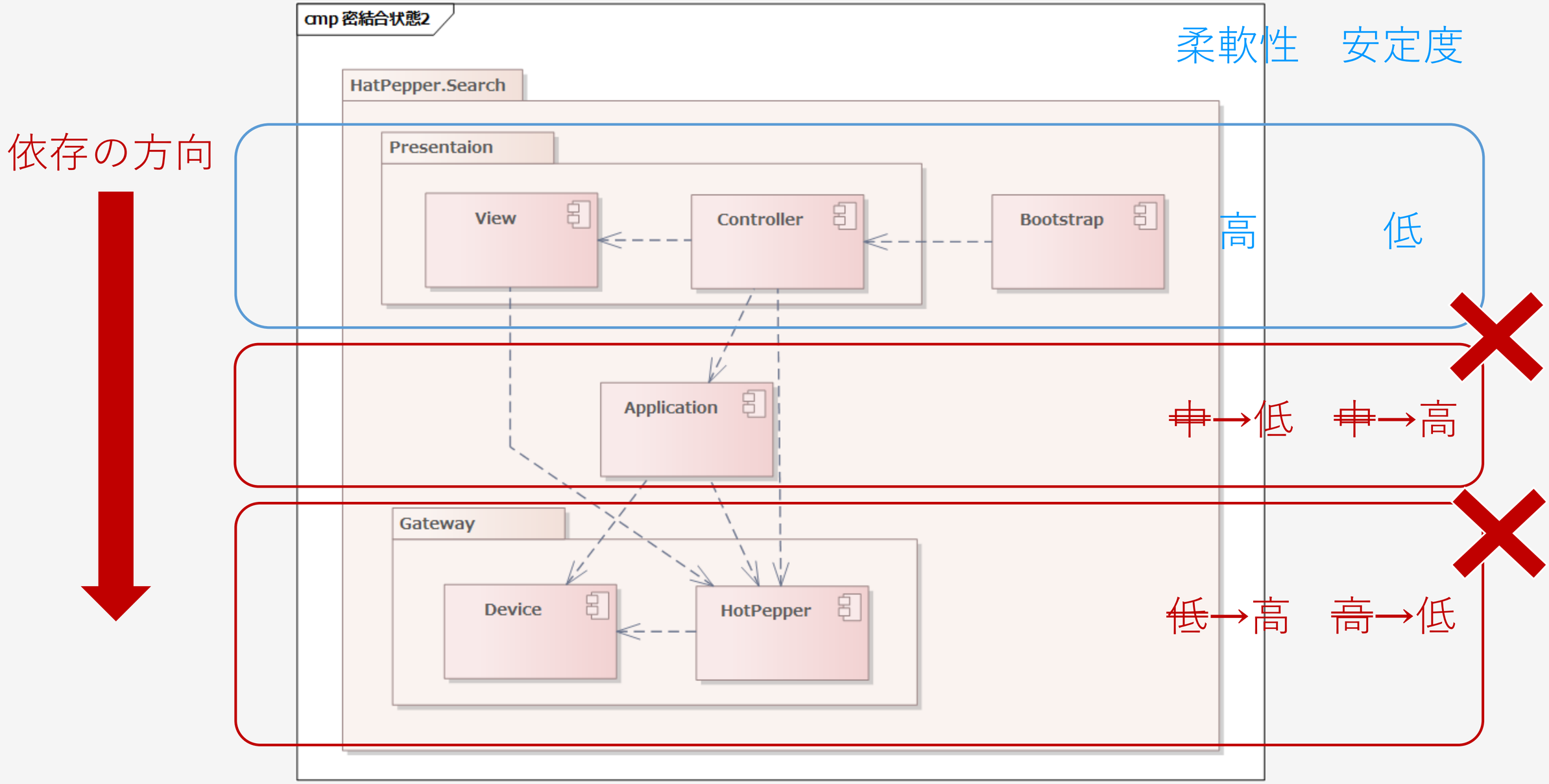
柔軟性と安定度



柔軟性と安定度



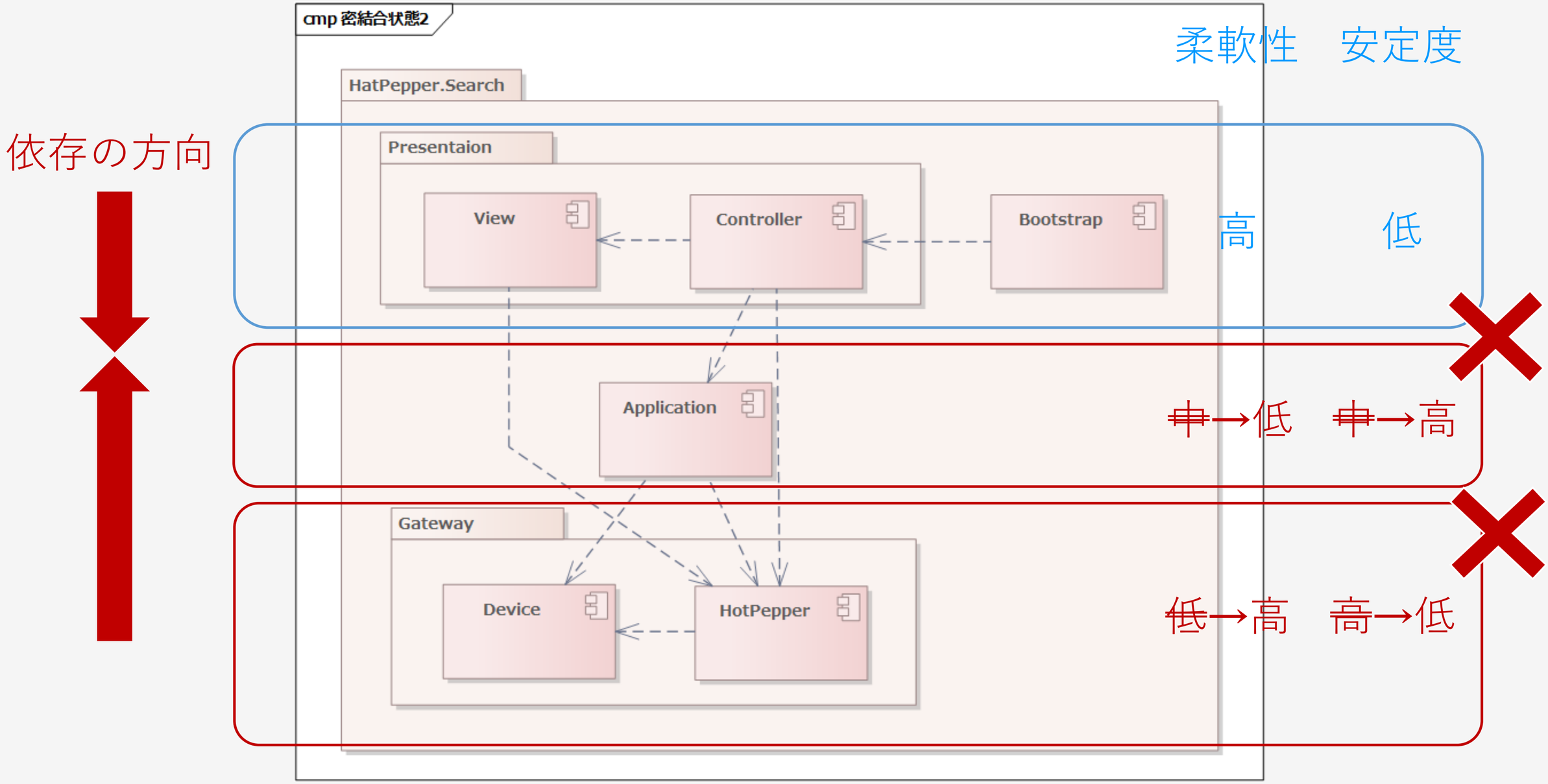
柔軟性と安定度



Easiest Clean Architecture

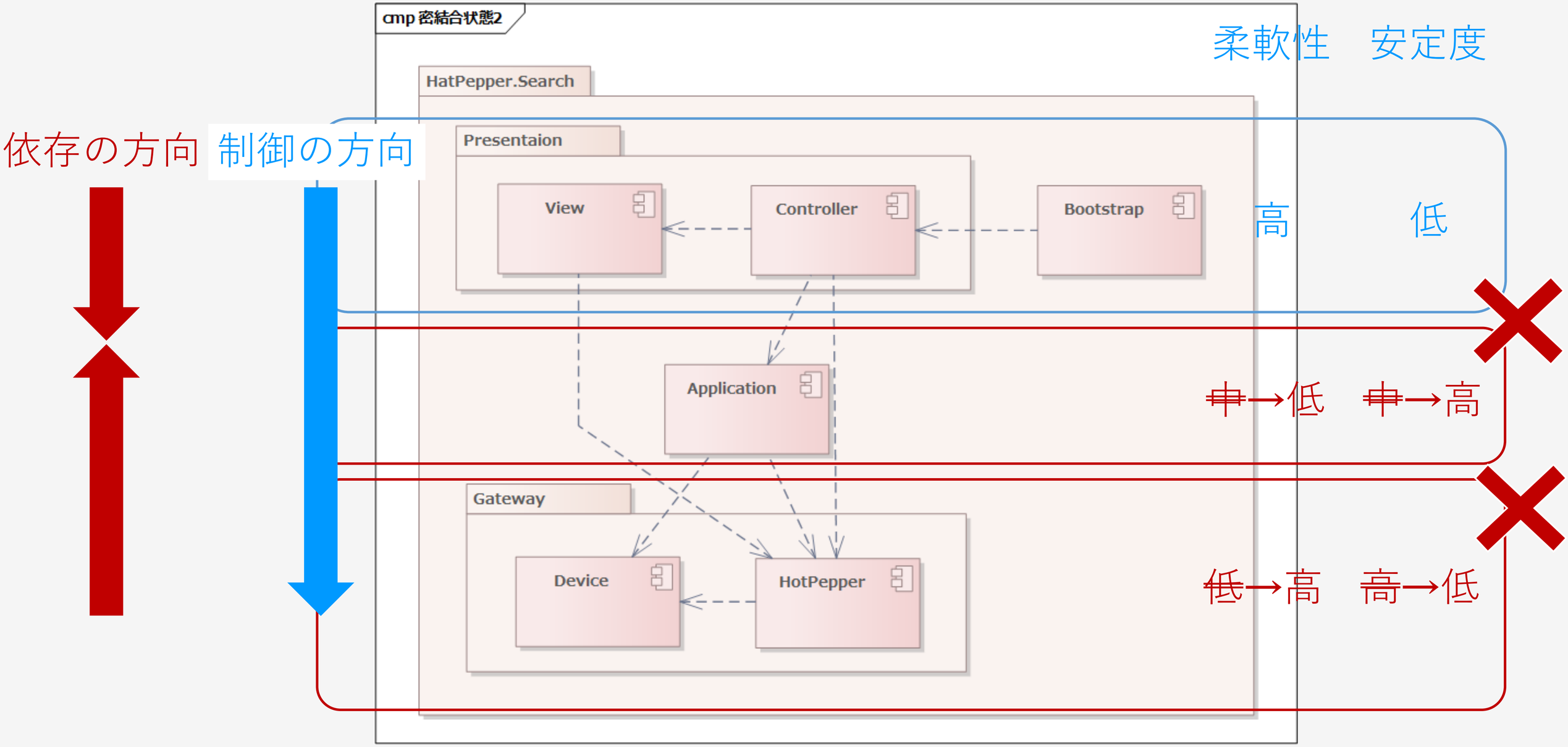
依存性は、より上位レベルの方針にのみ向けよ

柔軟性と安定度



実現可能なのか？

柔軟性と安定度



Easiest Clean Architecture

制御の流れと依存関係の分離

制御の流れと依存関係の分離

一般的に

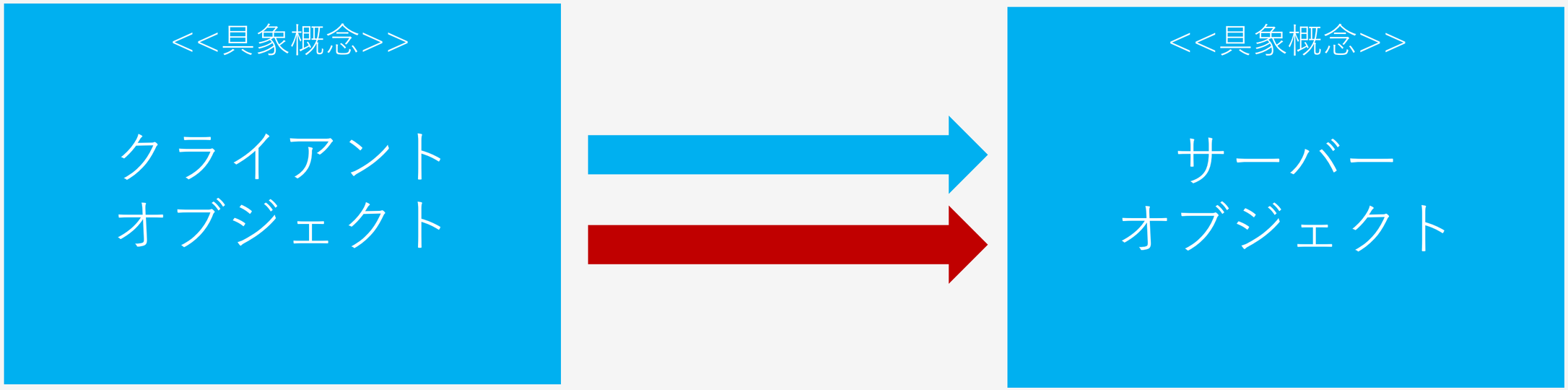
制御の流れ＝依存方向

になりがち

凡例

制御の流れ

依存方向



制御の流れと依存方向は分離しコントロールできる

制御の流れと依存関係の分離

そのためには関係のコントラクト（契約・仕様・お約束）のコンテキスト（文脈）を制御する

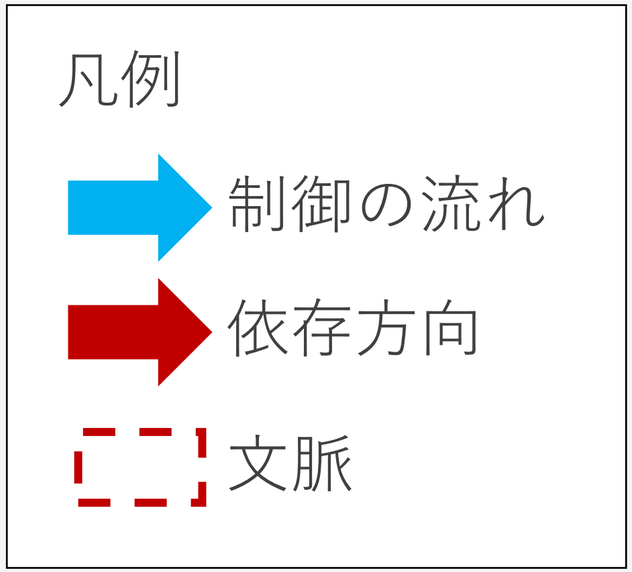
凡例

制御の流れ

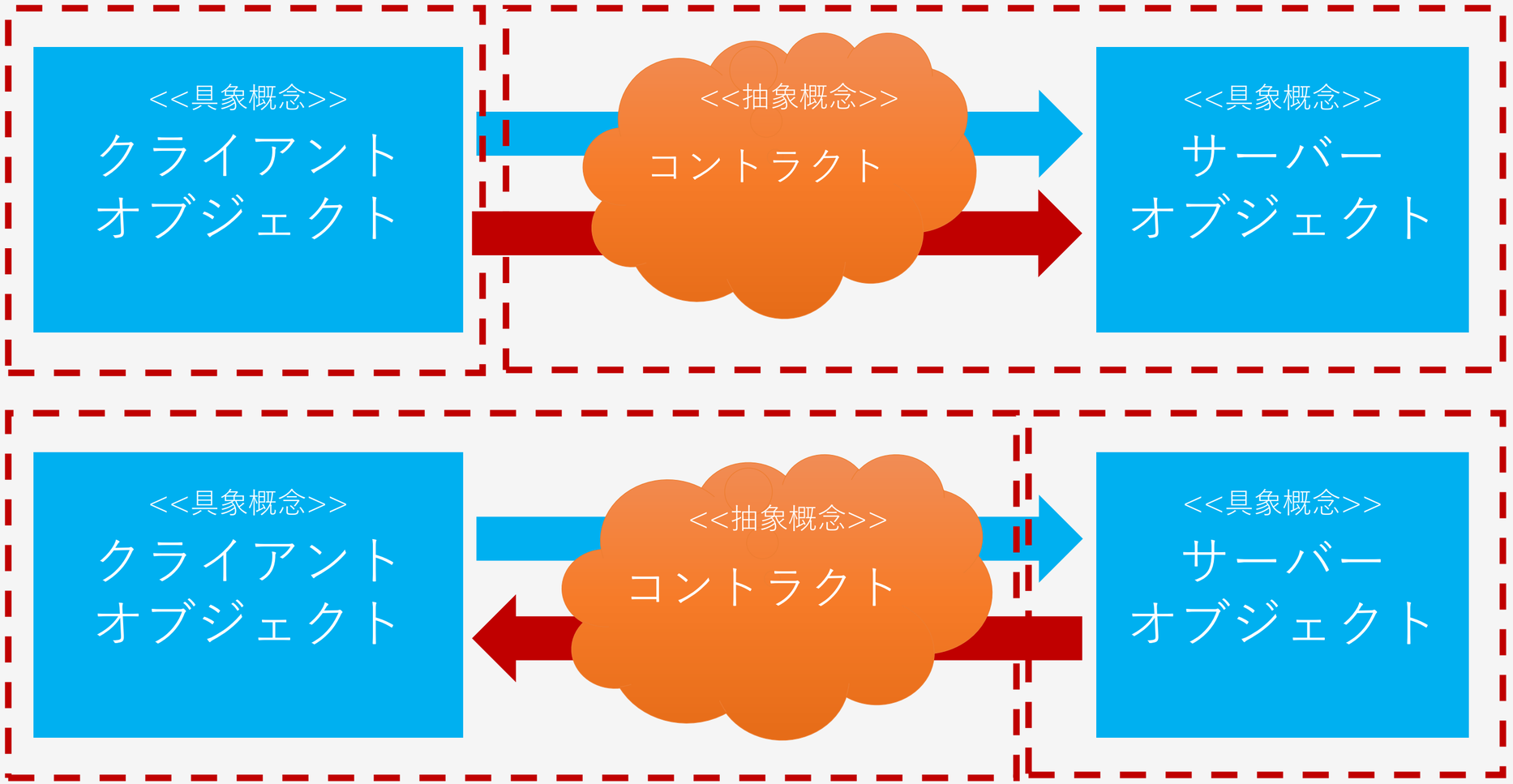
依存方向

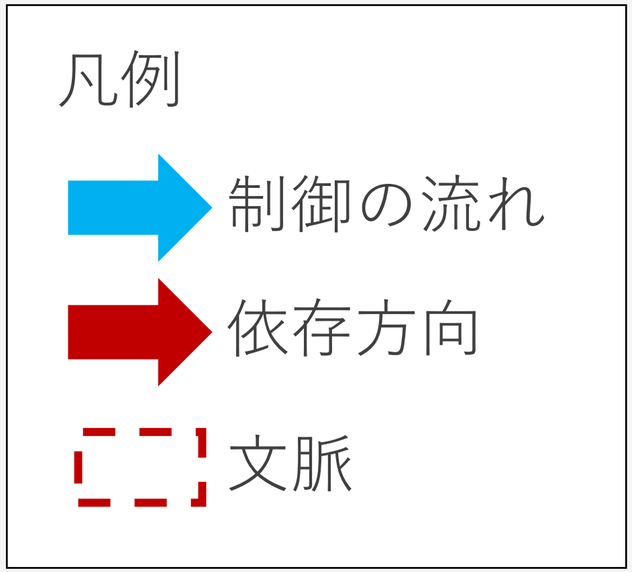
それぞれの具象概念は抽象的な契約（仕様）に依存する

契約の文脈をコントロールする

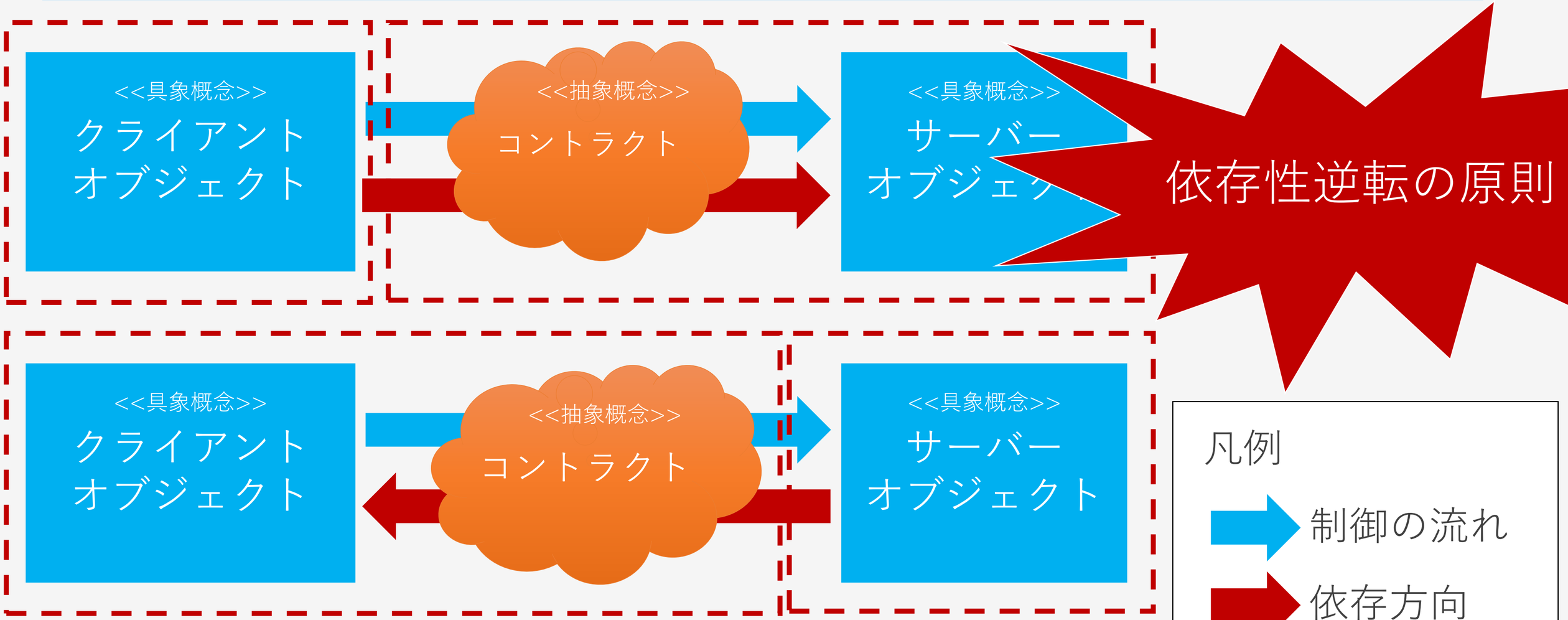


契約の文脈をコントロールする



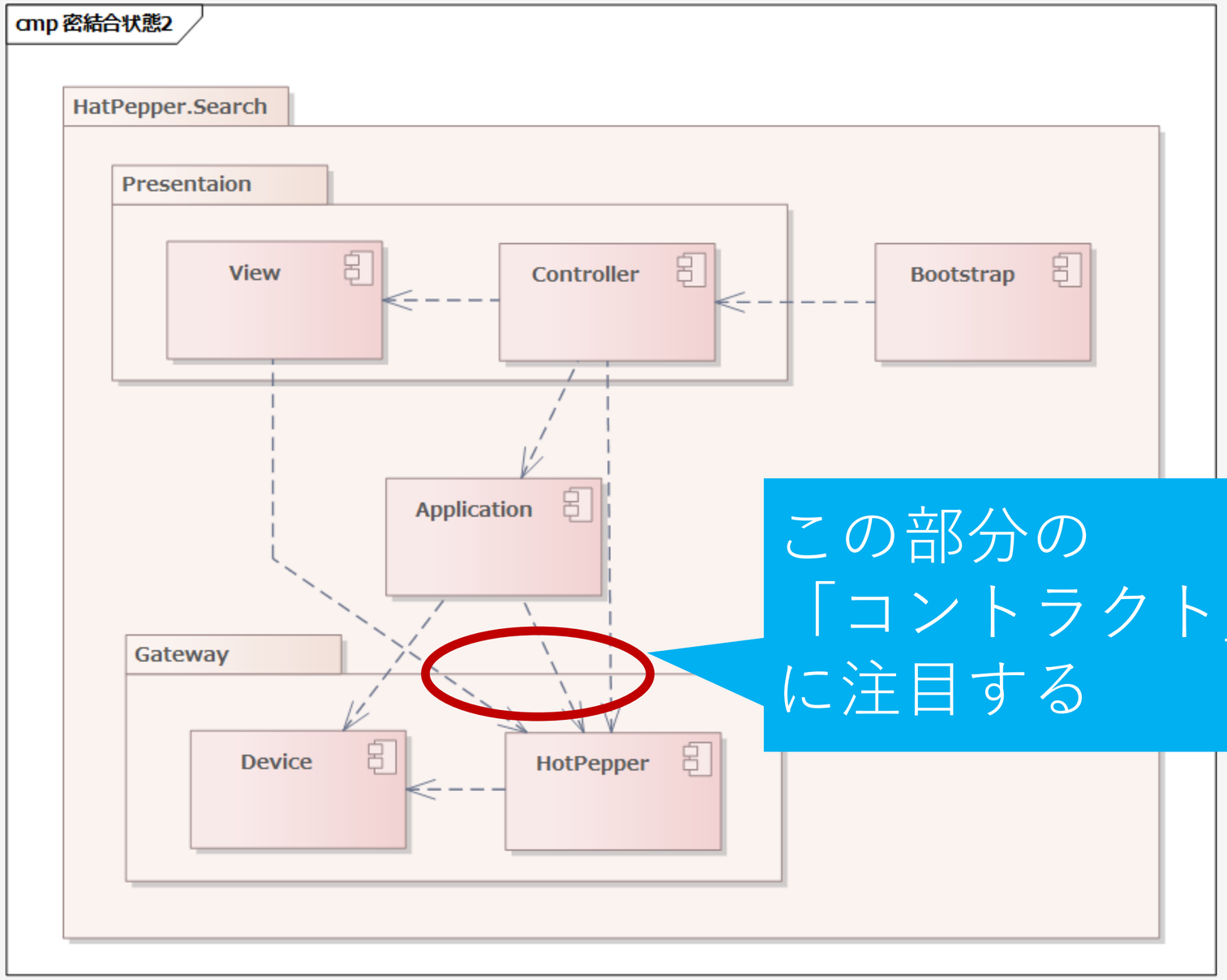


契約の文脈をコントロールする

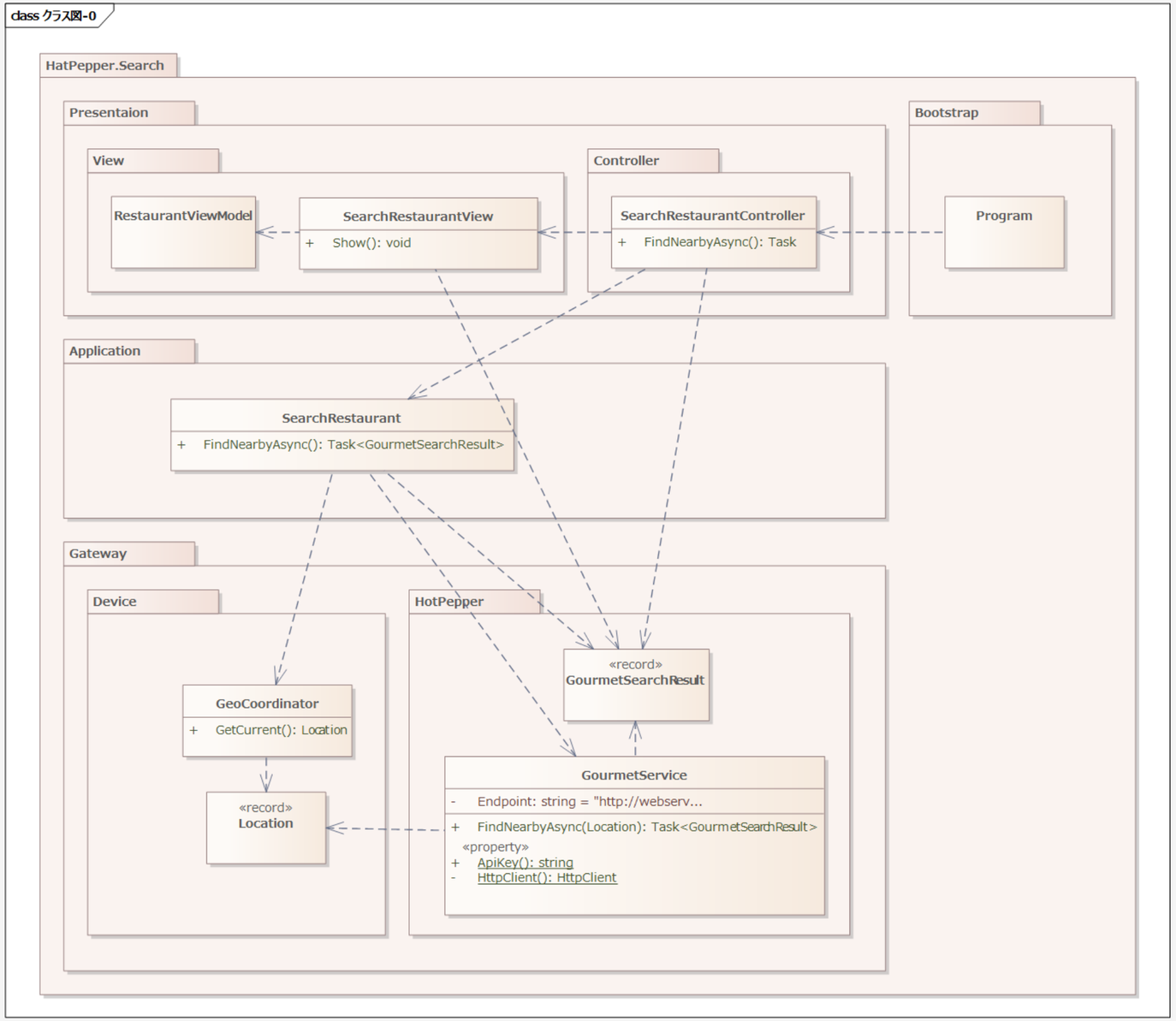


具体例を見てみよう

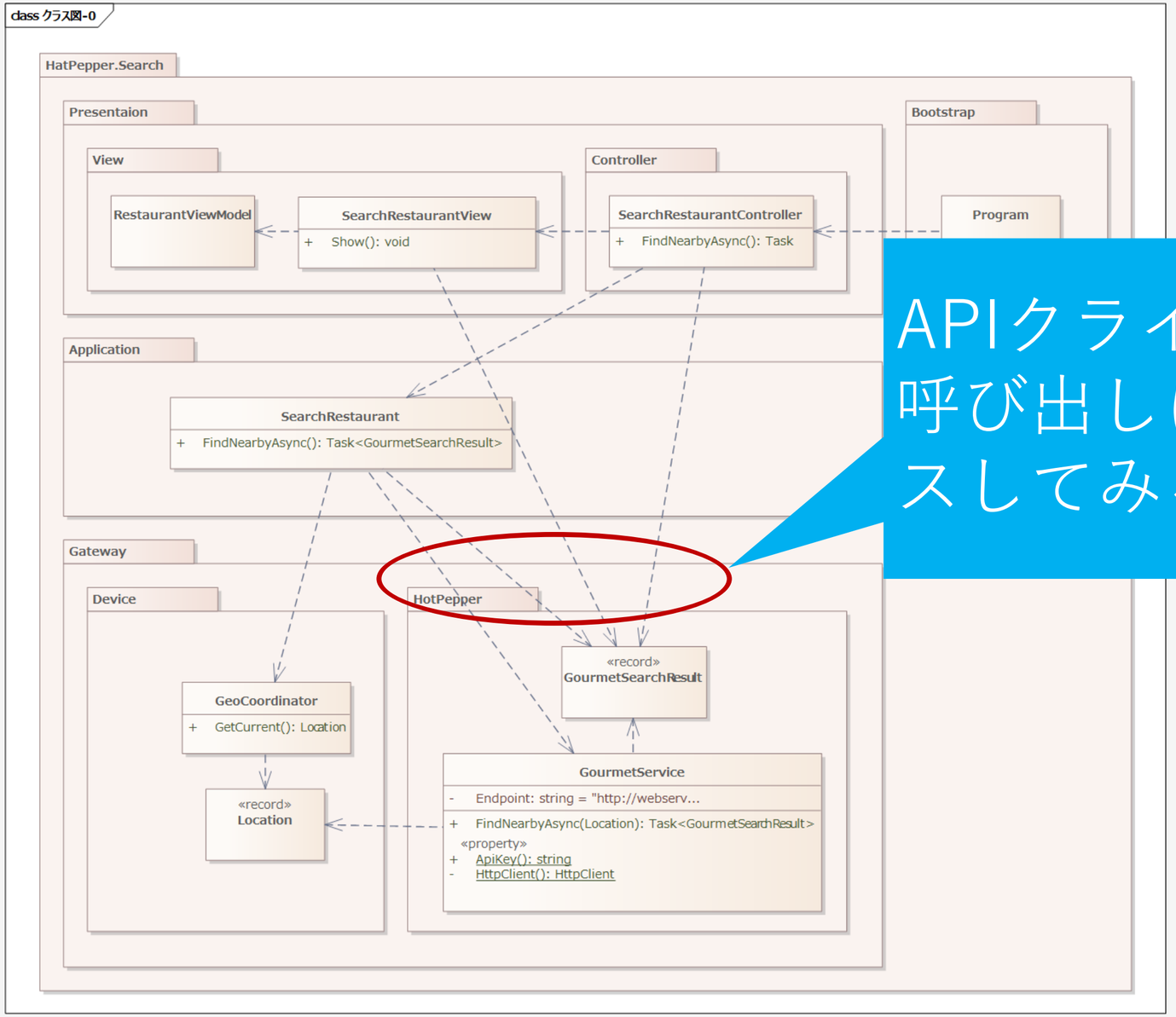
柔軟性と安定度



クラス図

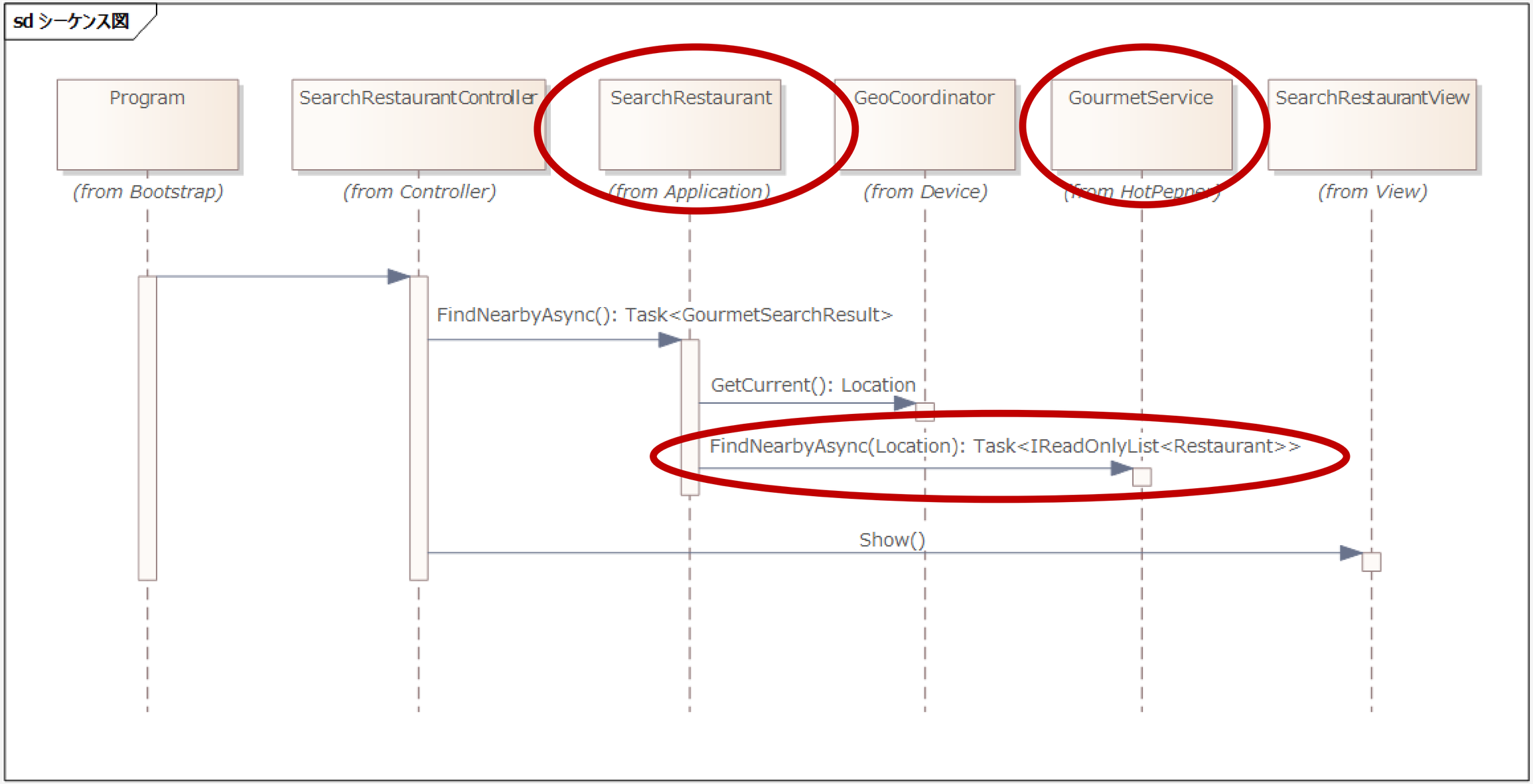


クラス図

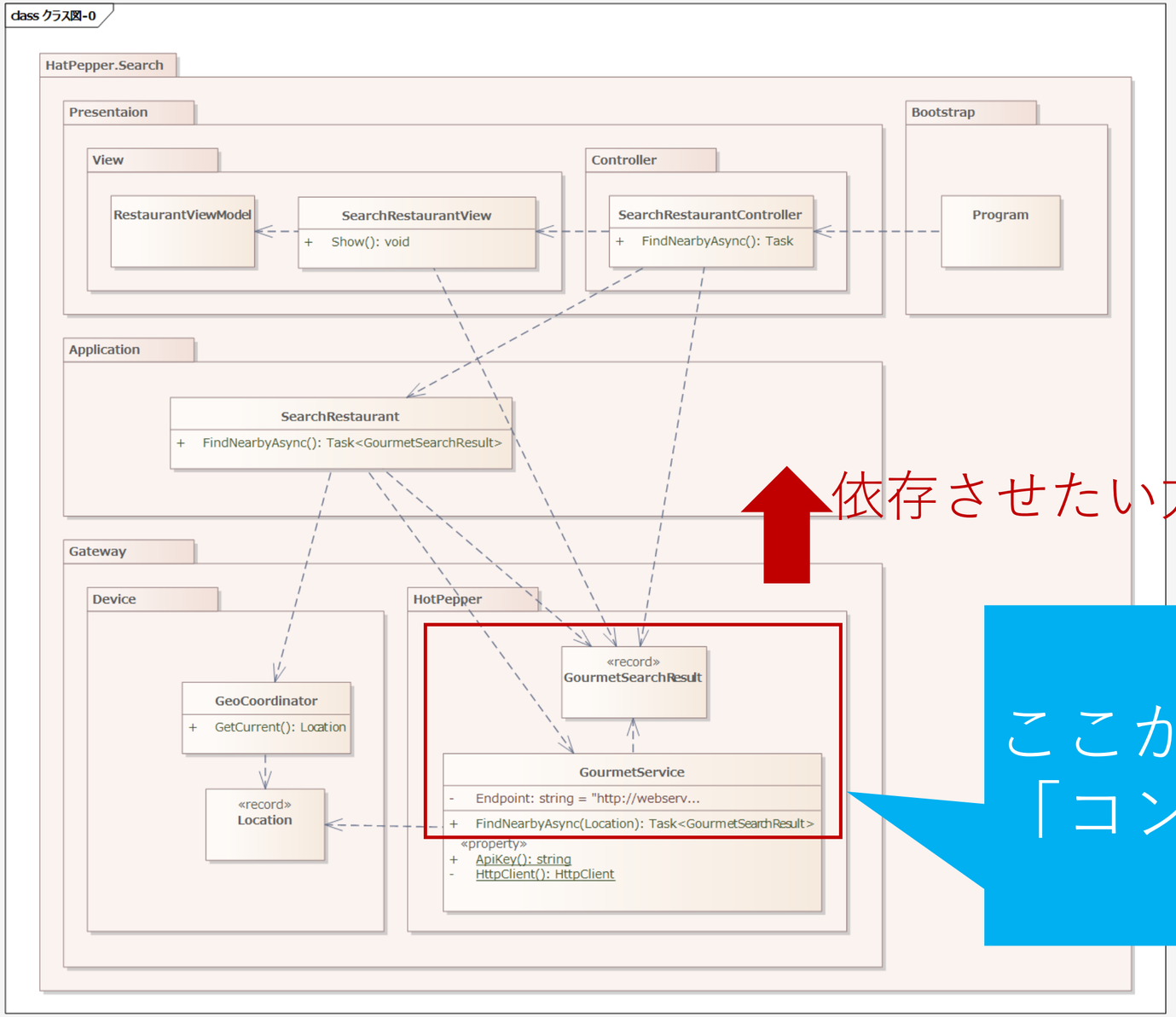


APIクライアントの呼び出しにフォーカスしてみる

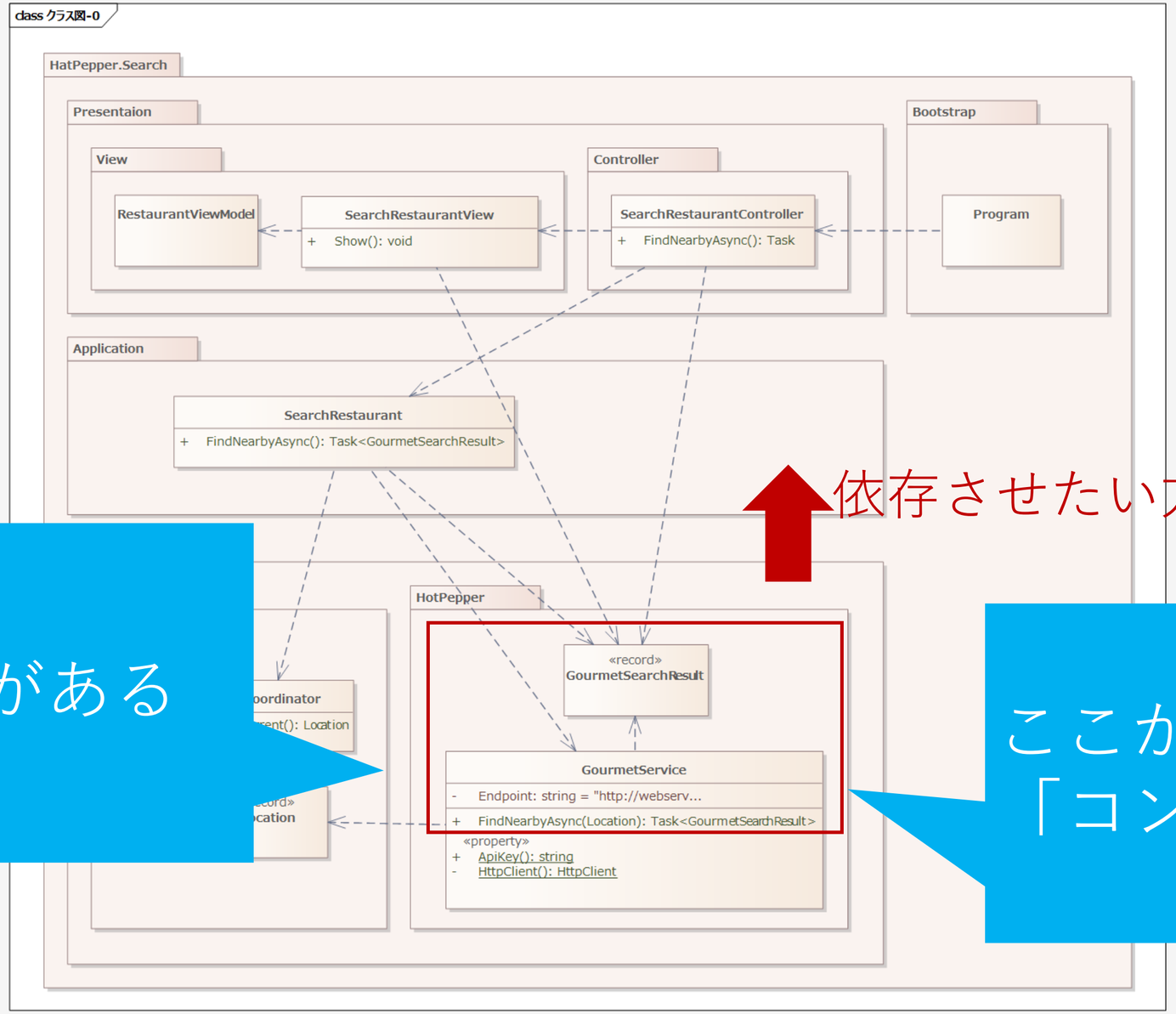
シーケンス図



クラス図



クラス図



依存させたい方向

二つの問題がある

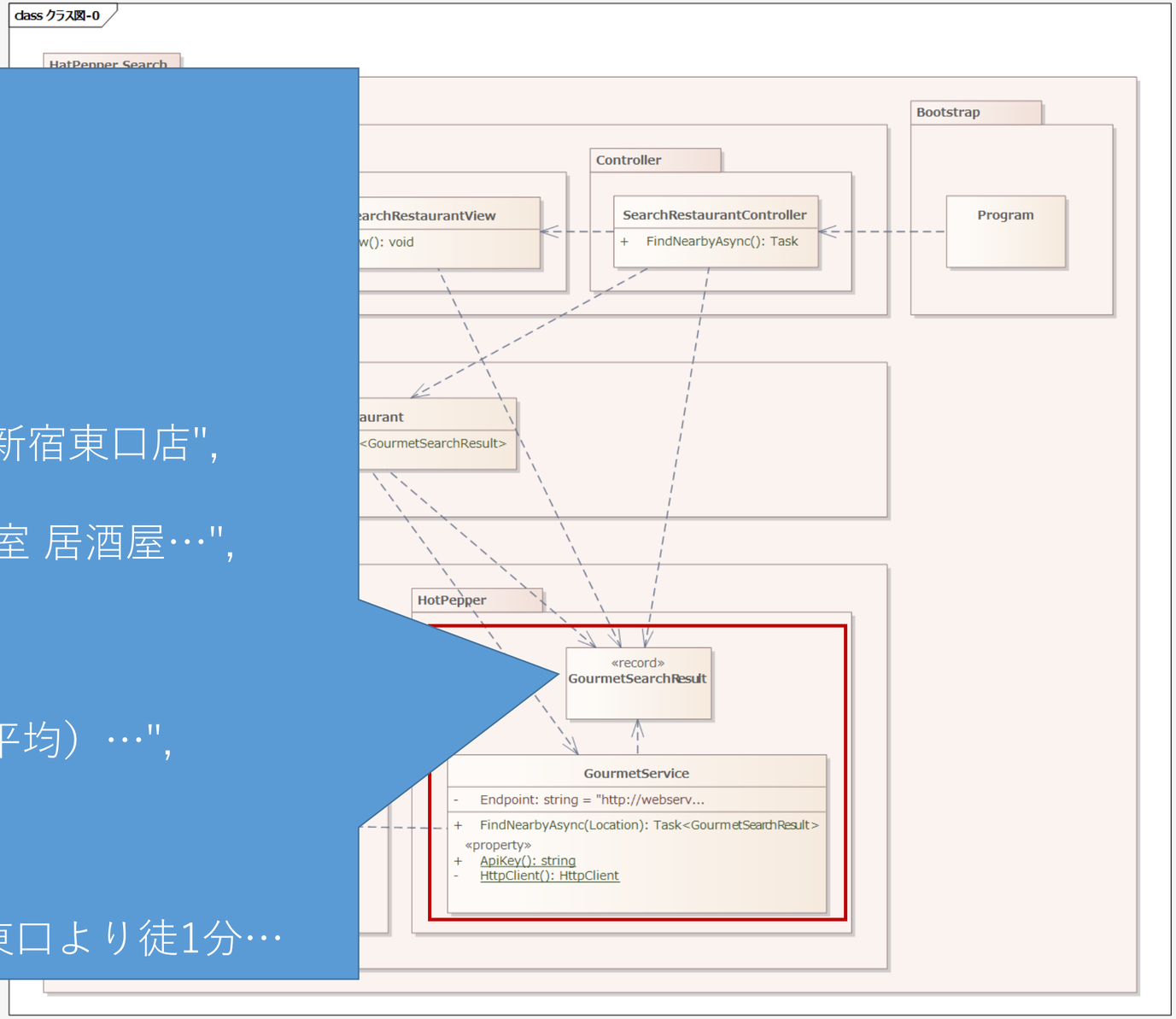
ここが
「コントラクト」

ひとつ　コントラクトがGateway側に定義されている

ふたつめ　コントラクトがWeb APIの文脈で記述されている

クラス図


```
{
  "results": {
    "results_start": 1,
    "results_returned": "2",
    "api_version": "1.26",
    "shop": [
      {
        "name": "彩羽鶏 いろはどり 新宿東口店",
        "genre": {
          "catch": "新宿 月あかり 個室 居酒屋…",
          …
        },
        "budget": {
          "average": "2500円（通常平均）…",
          "name": "2001～3000円",
          "code": "B002"
        },
        "mobile_access": "JR新宿駅東口より徒1分…
```

どうすれば？

契約の文脈をコントロールする

変更前

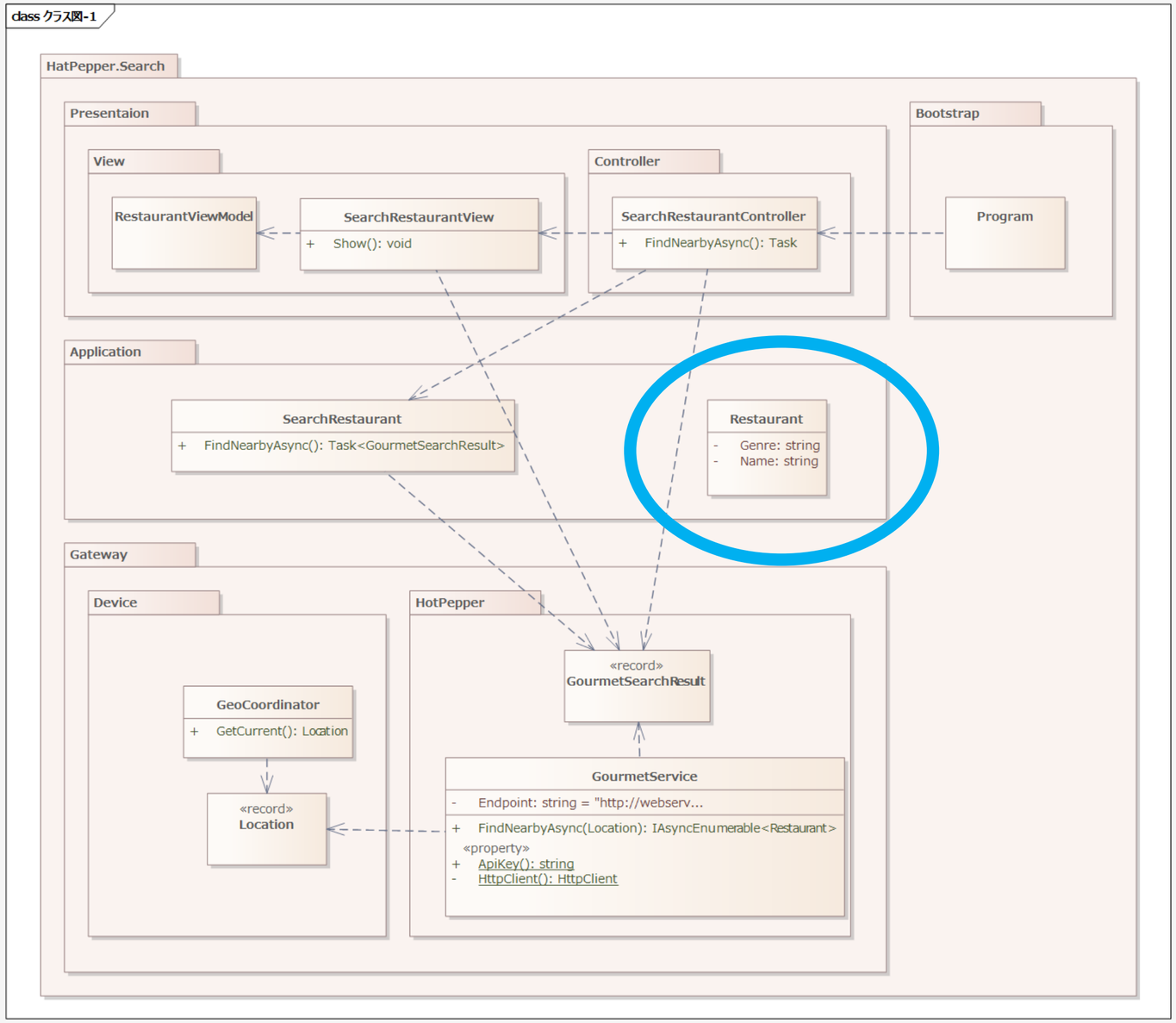
契約の文脈をコントロールする

変更前

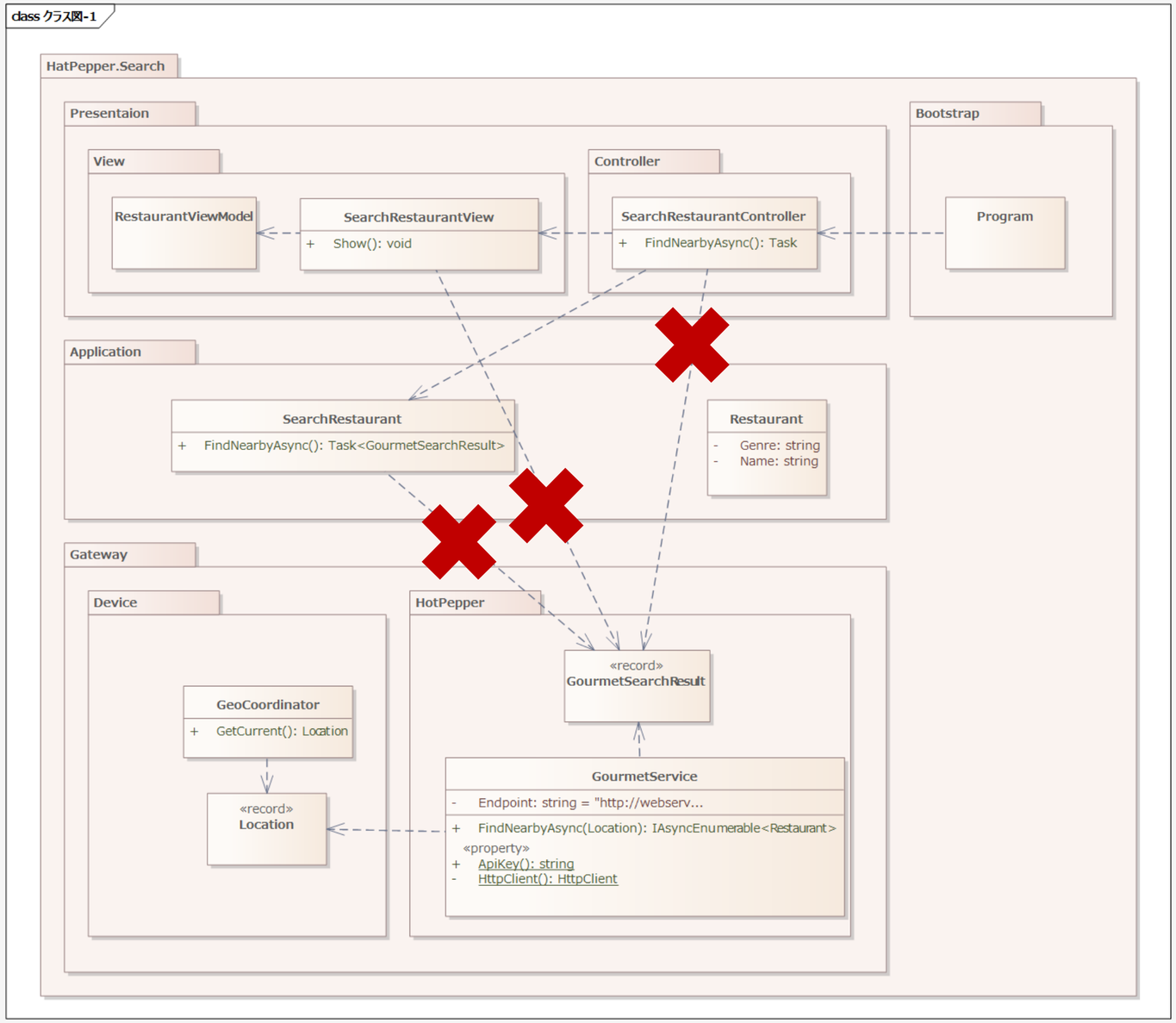
変更後

というわけで・・・

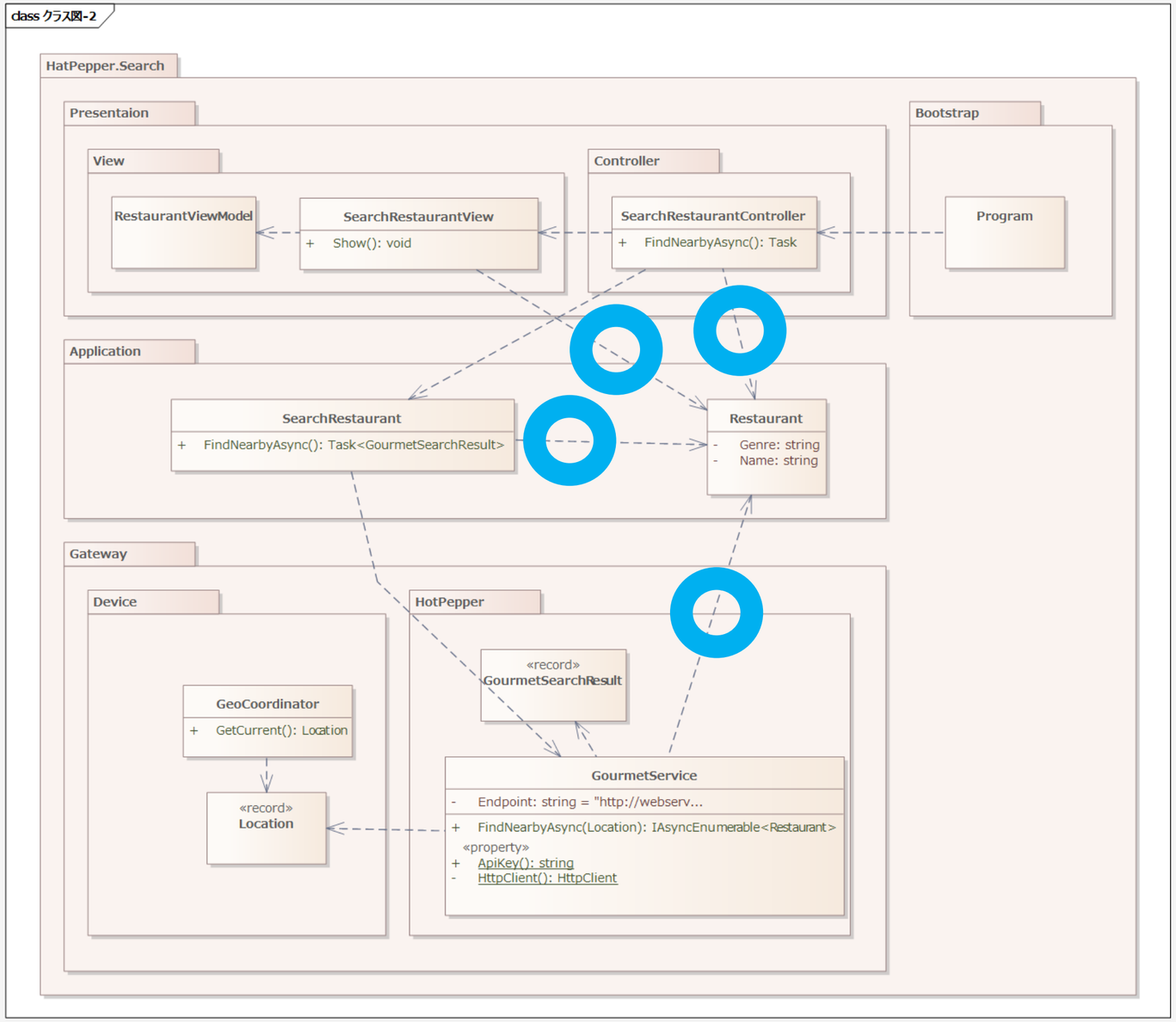
クラス図



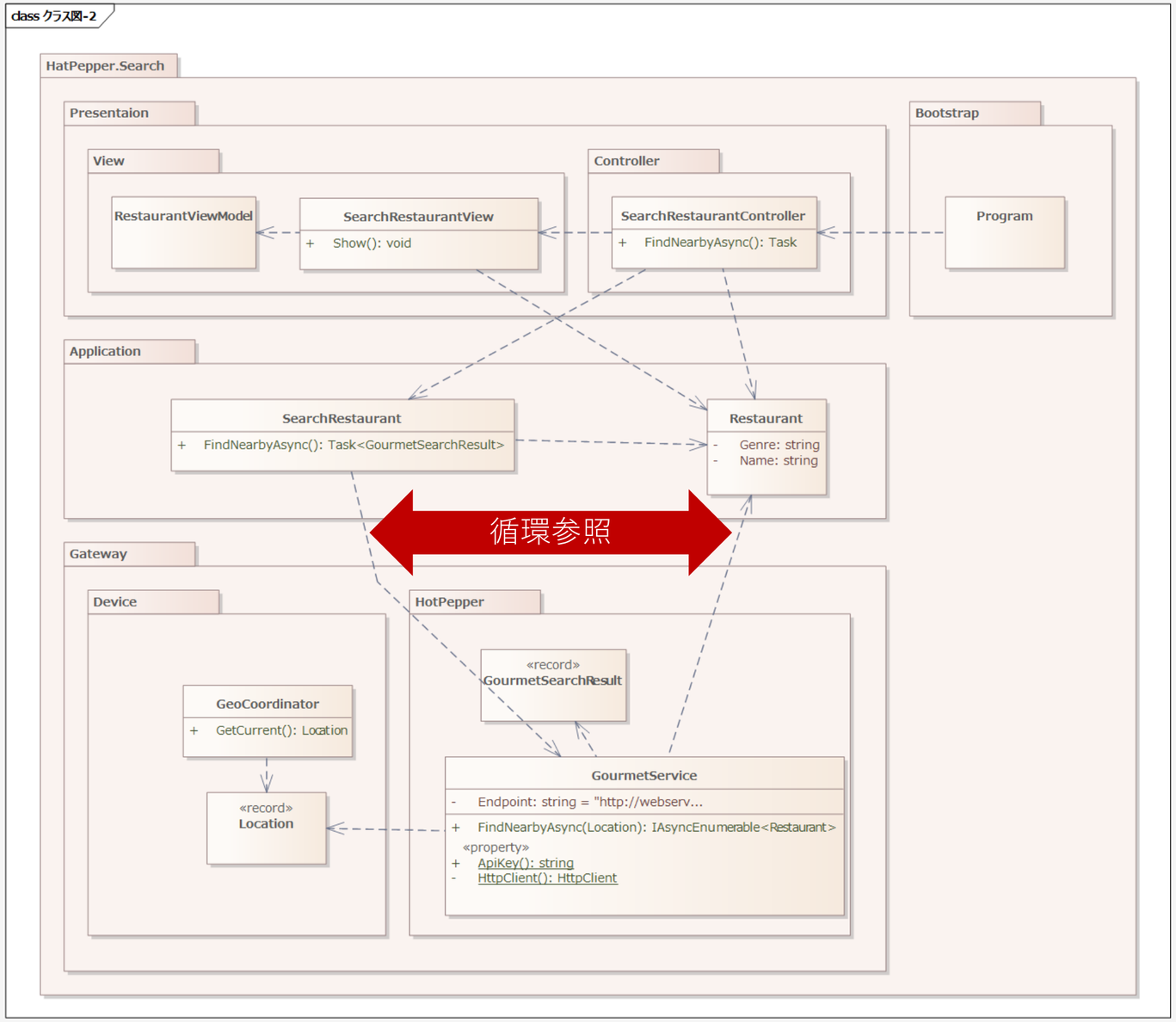
クラス図



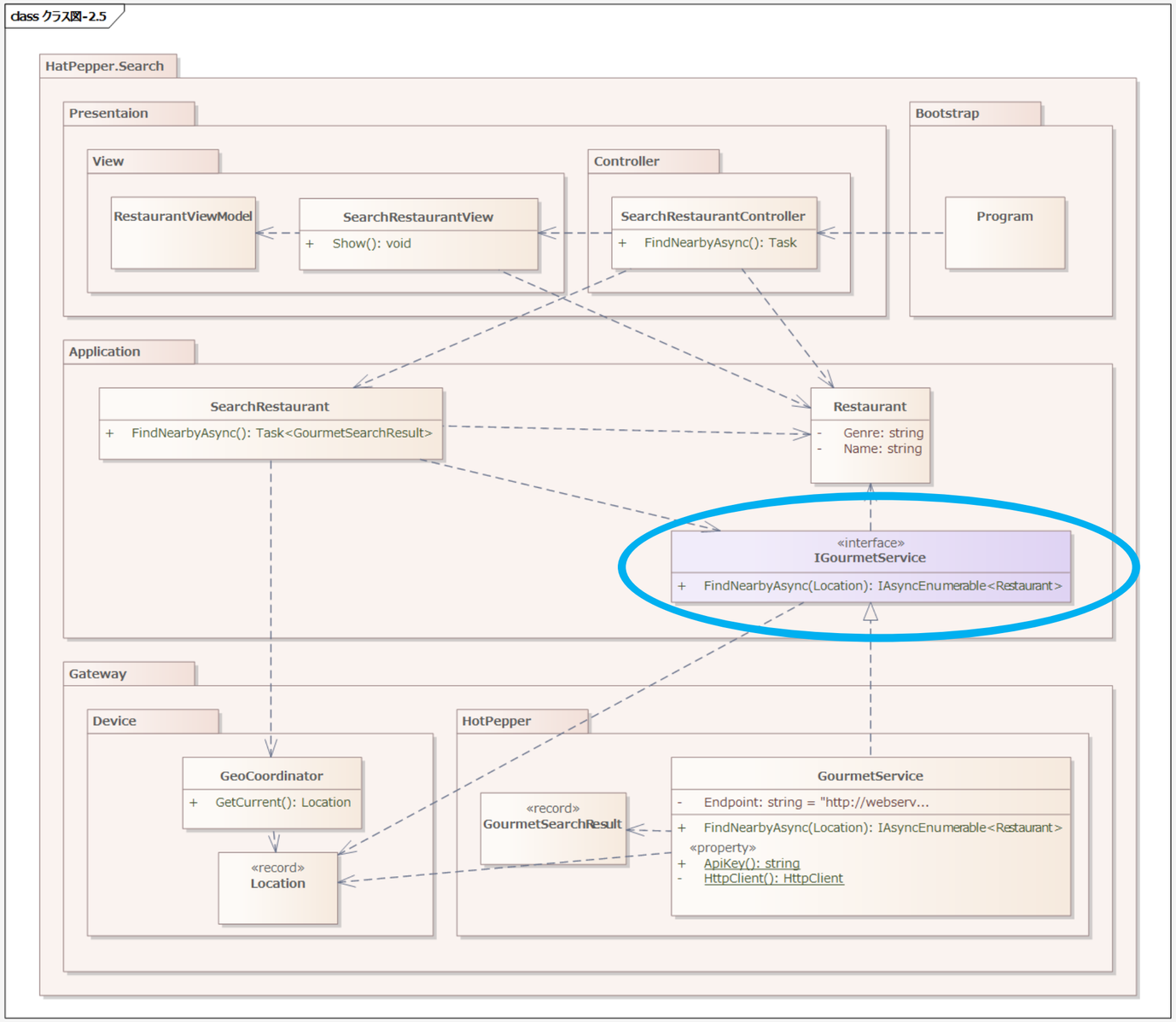
クラス図



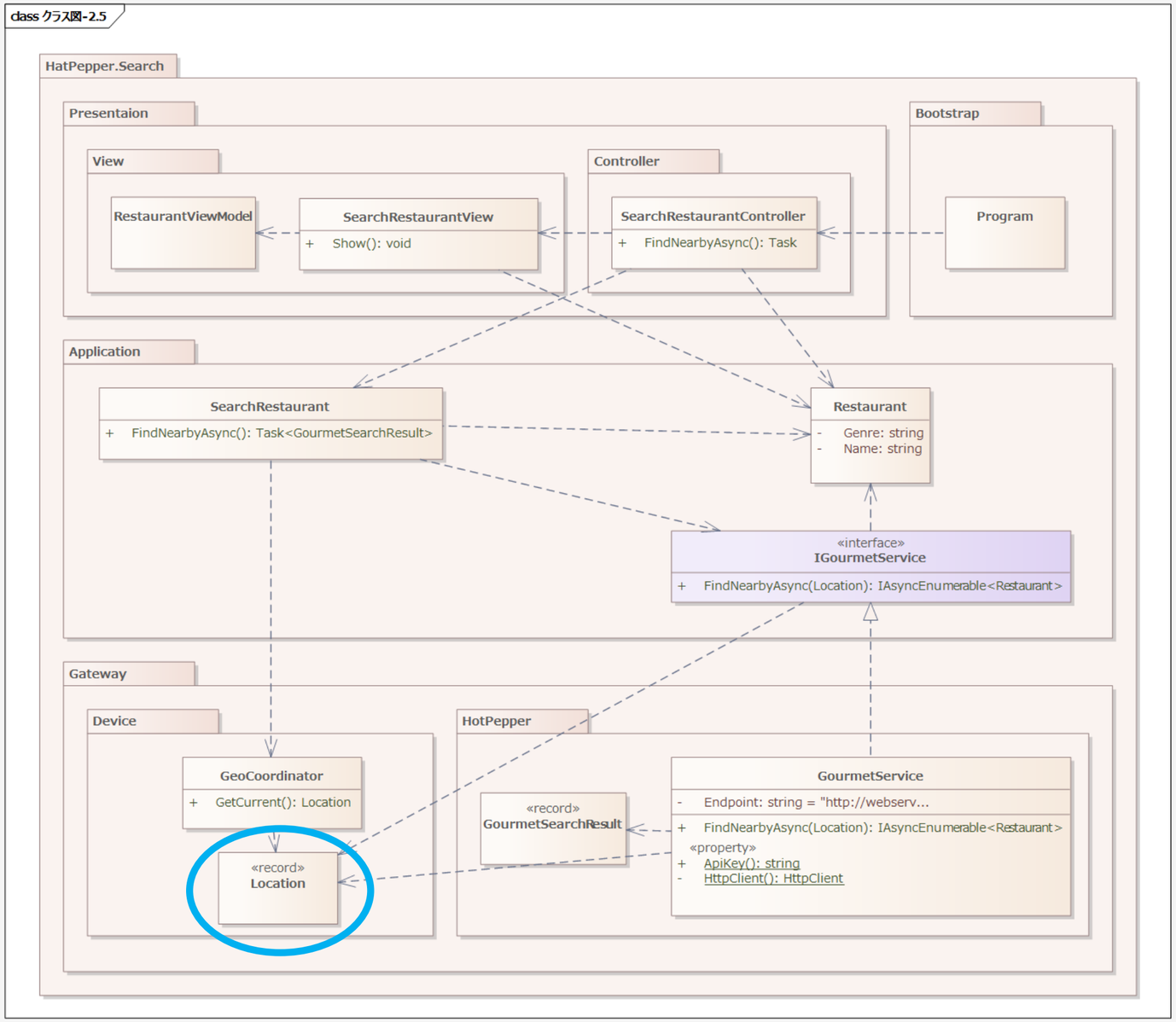
クラス図



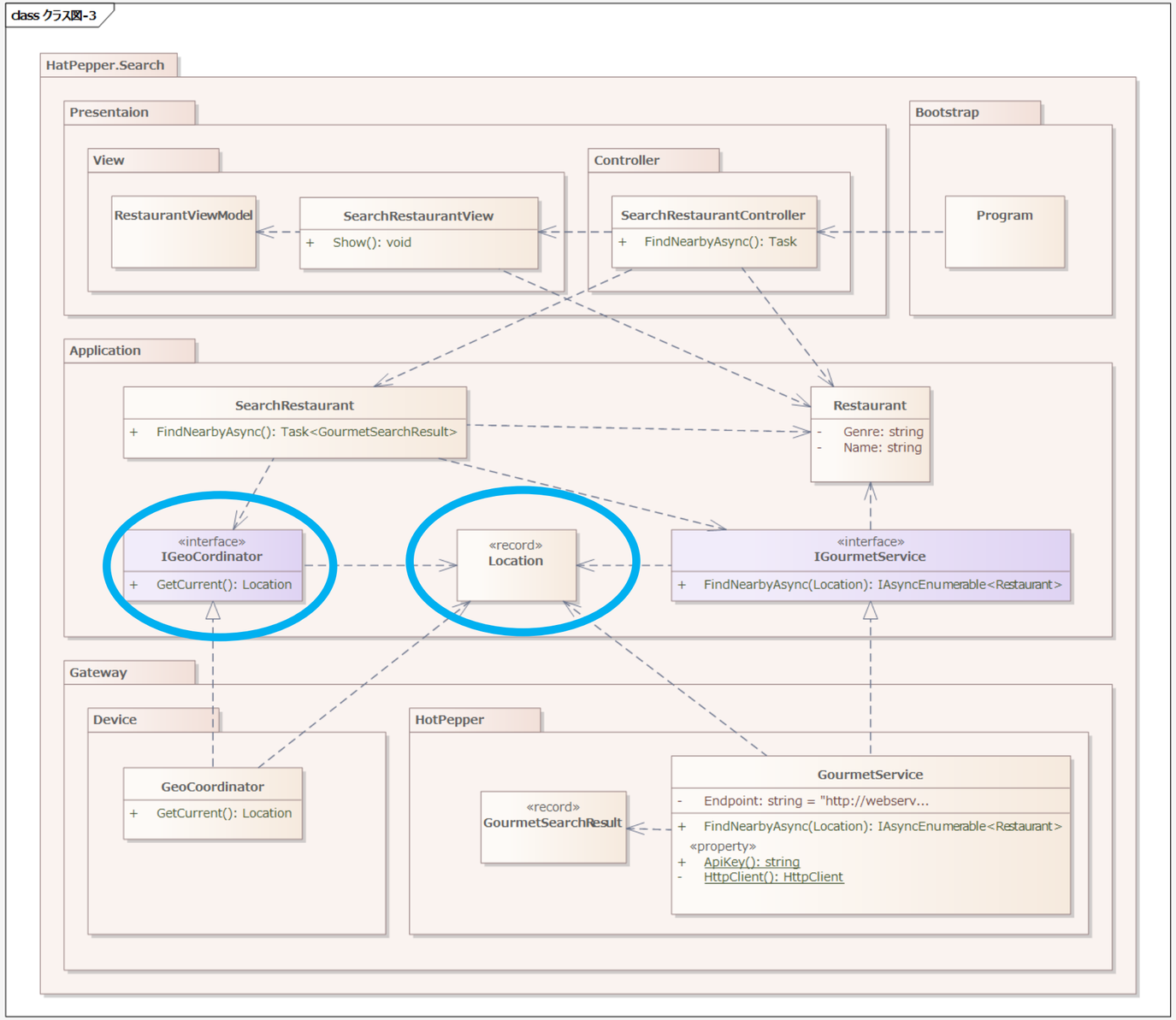
クラス図



クラス図



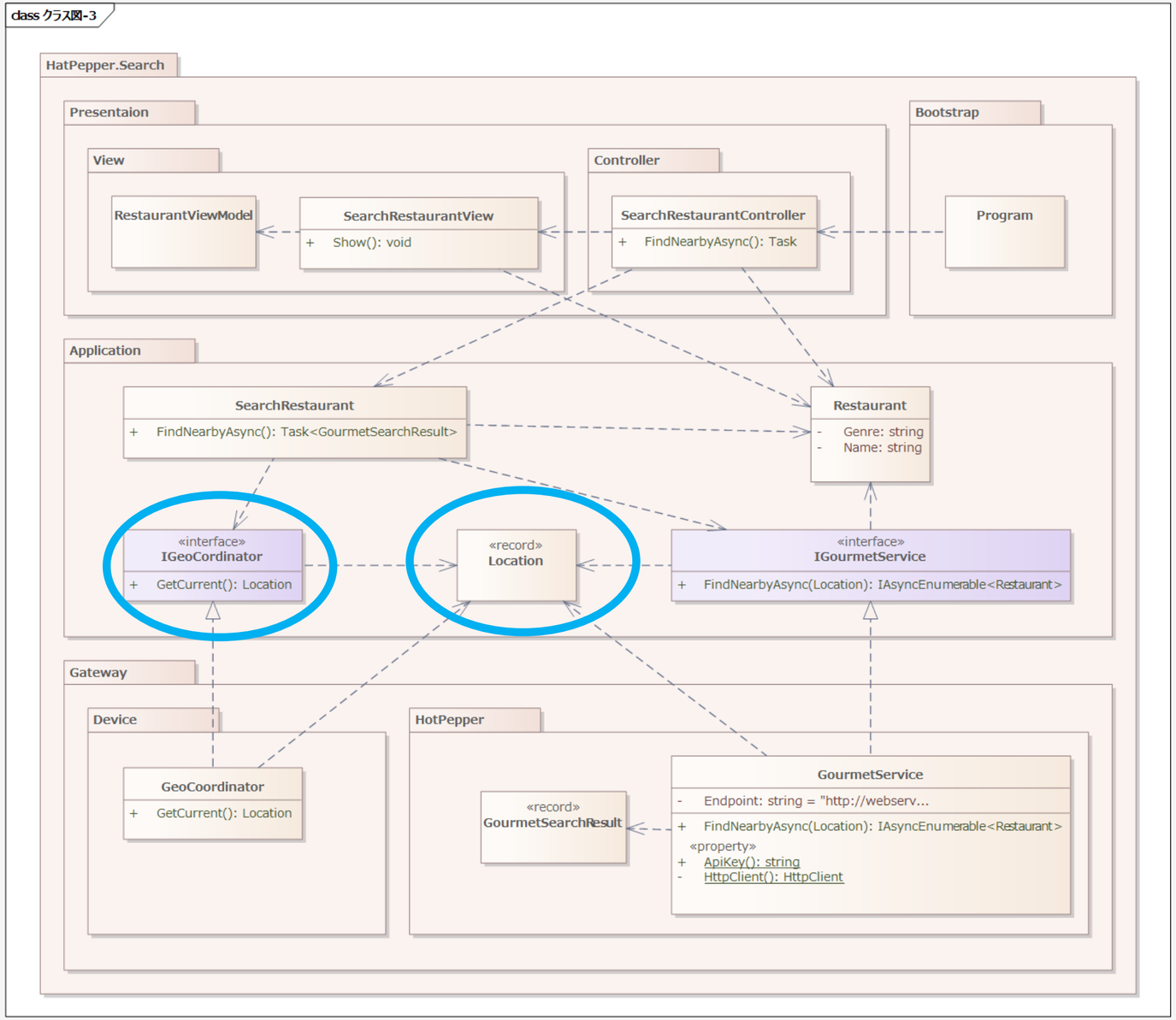
クラス図



クラス図

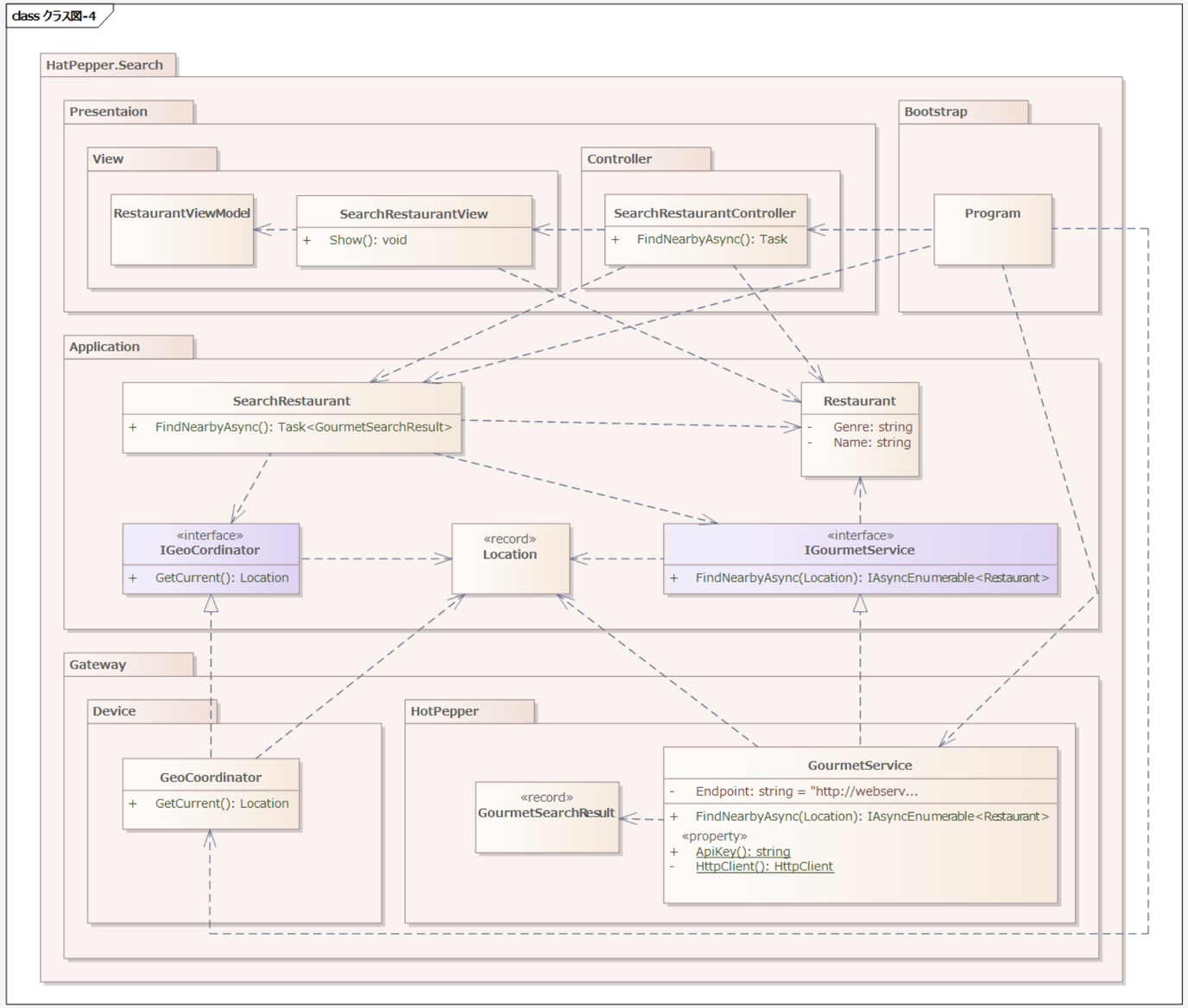
依存の方向

制御の方向

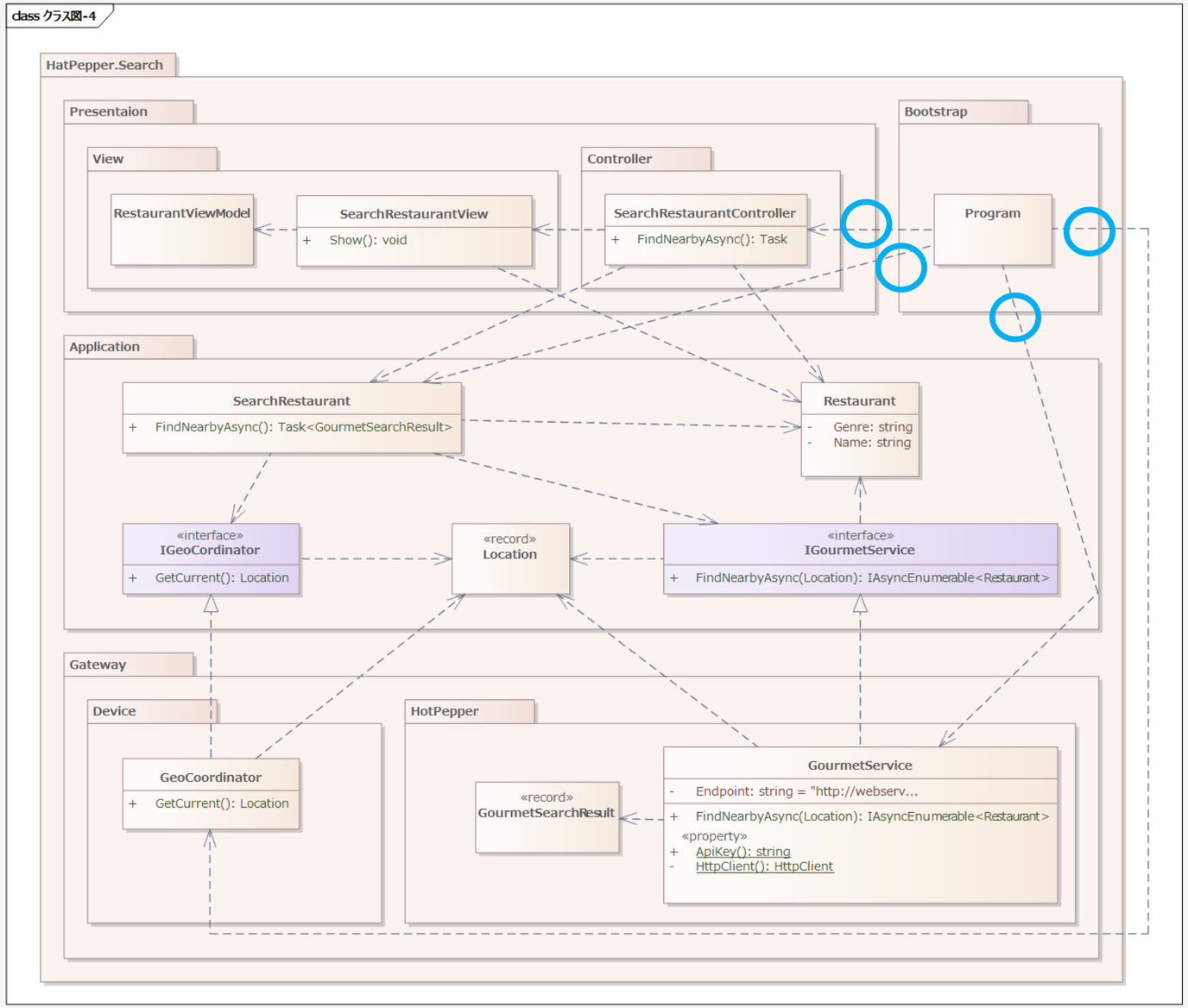


本当に実装できるの？

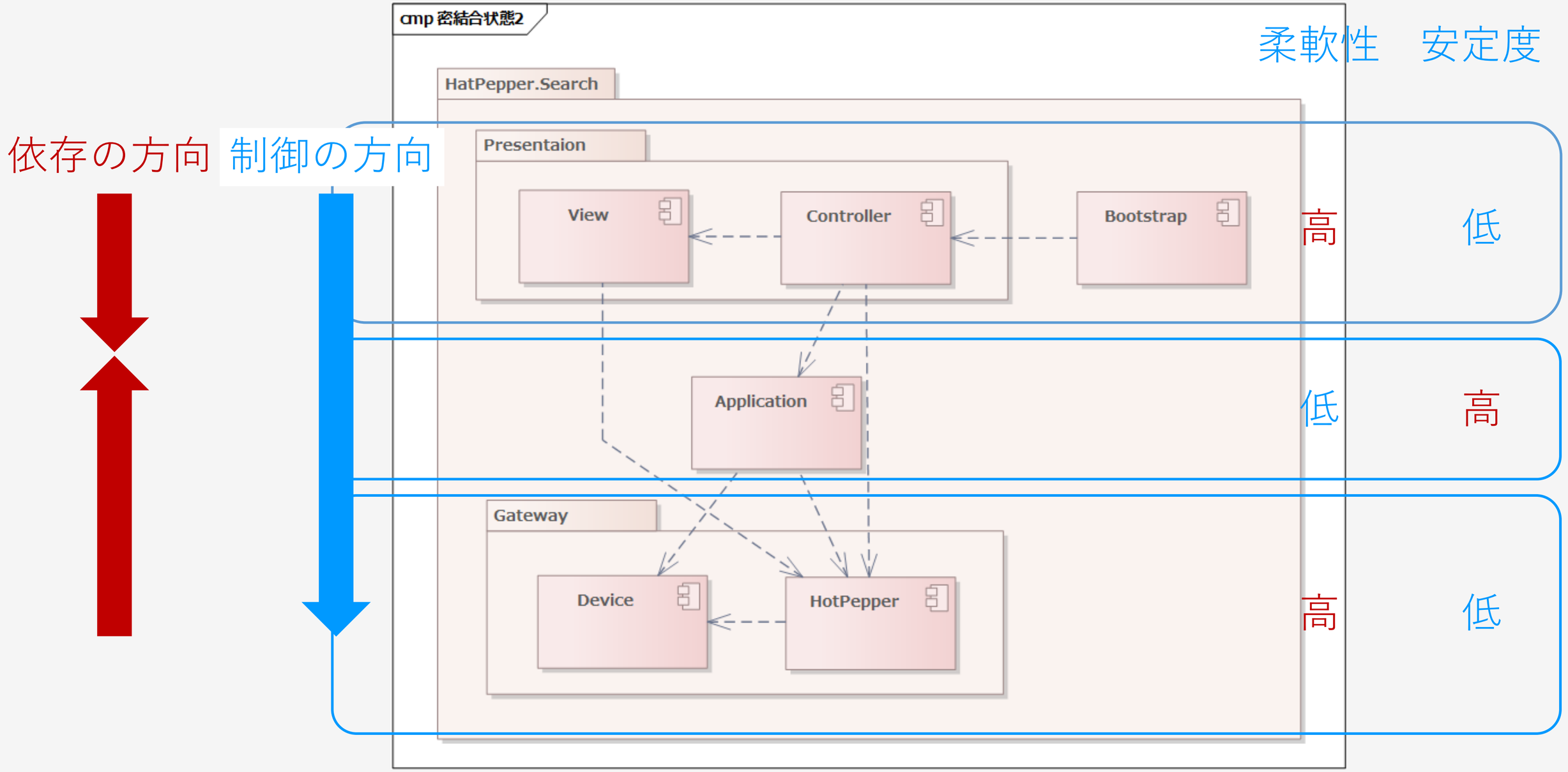
クラス図



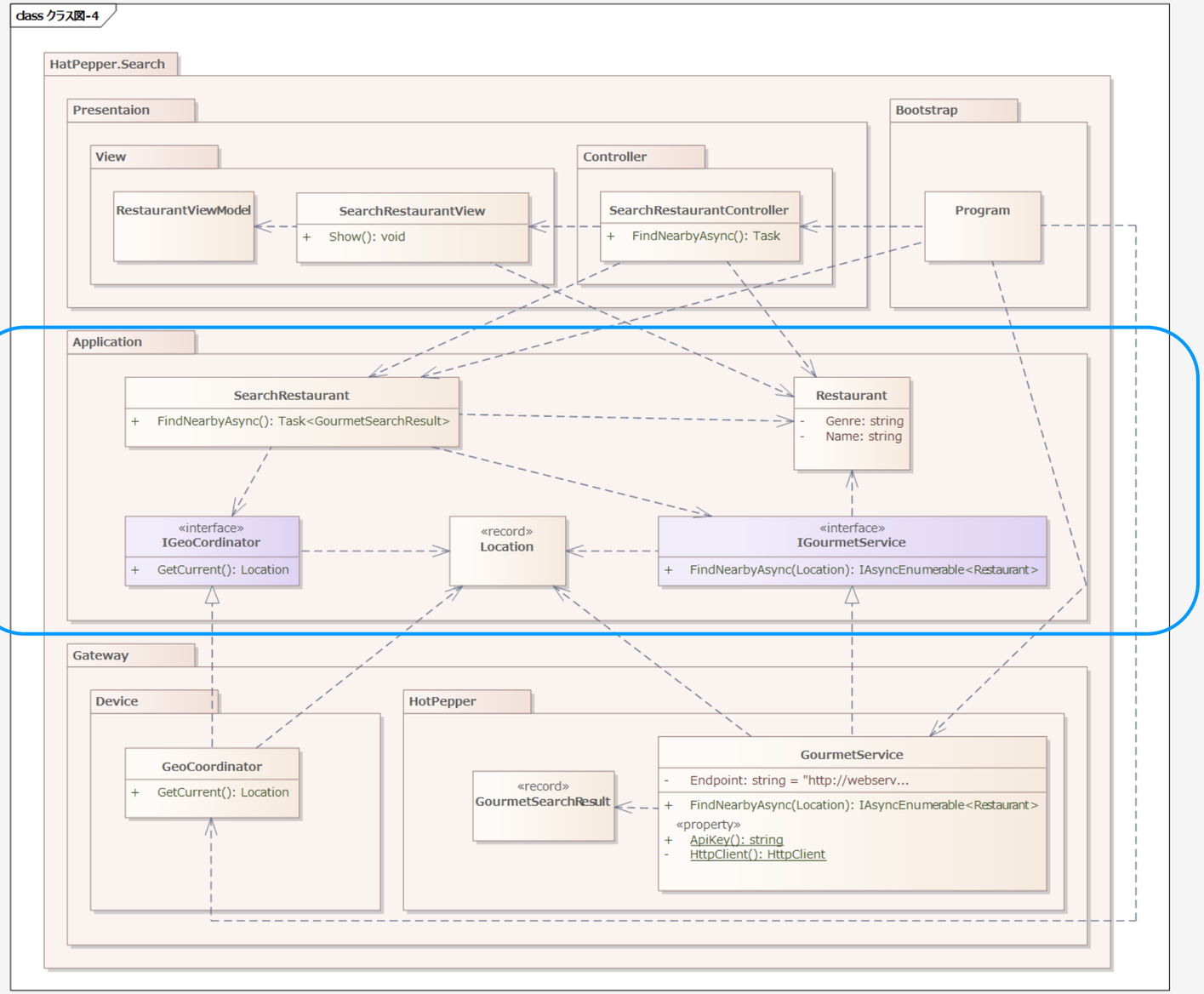
クラス図



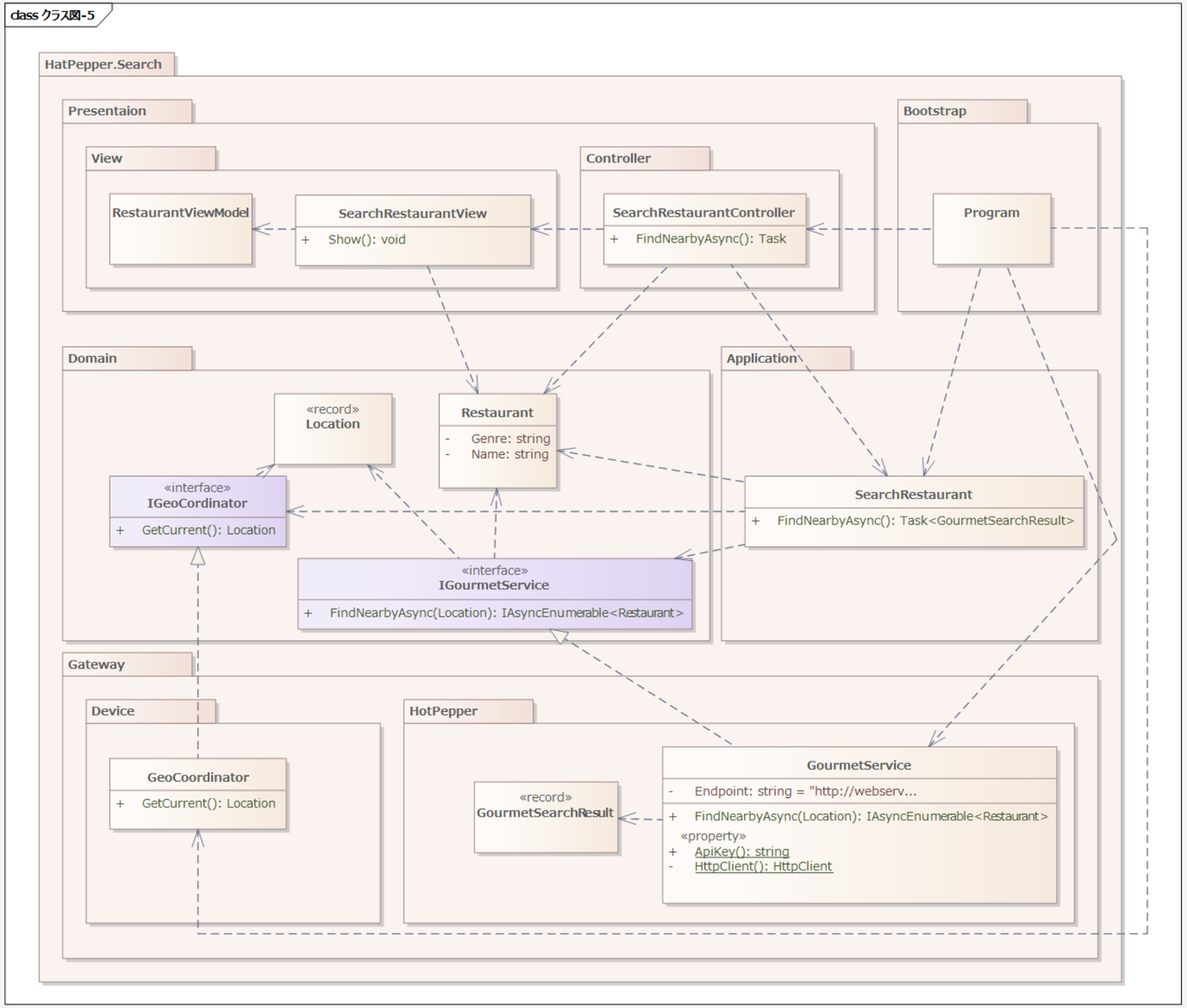
制御の流れと依存関係の分離



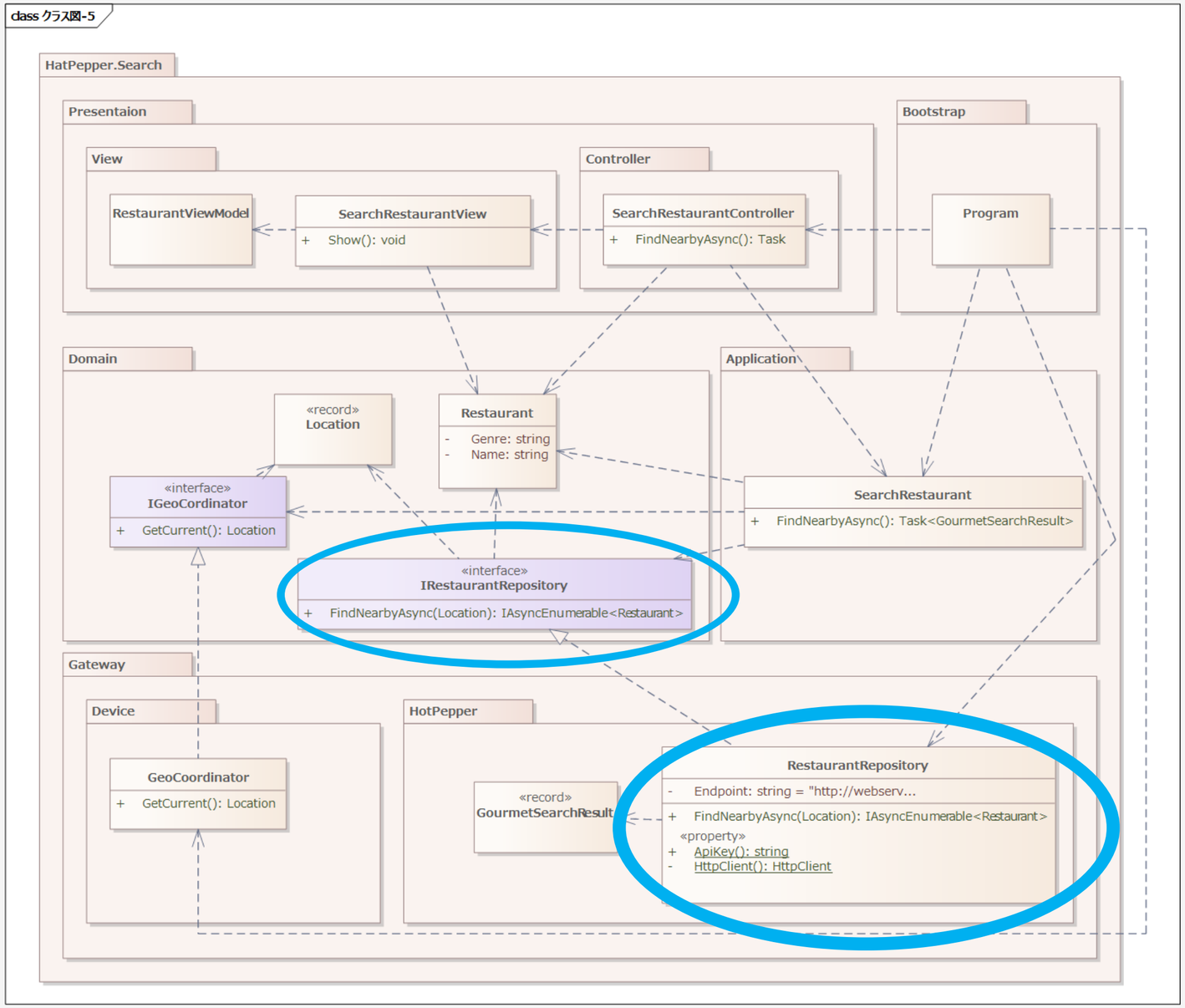
クラス図



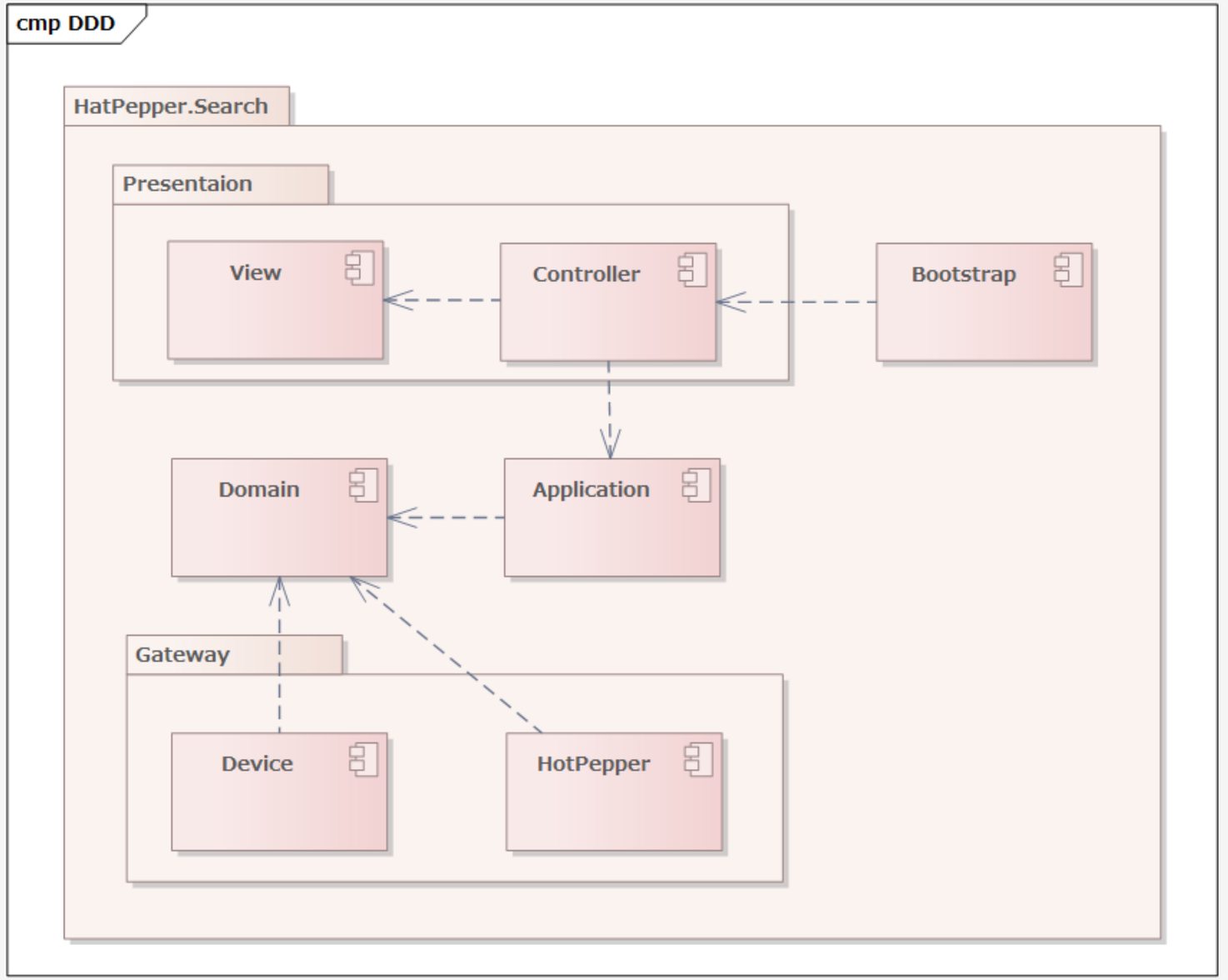
クラス図



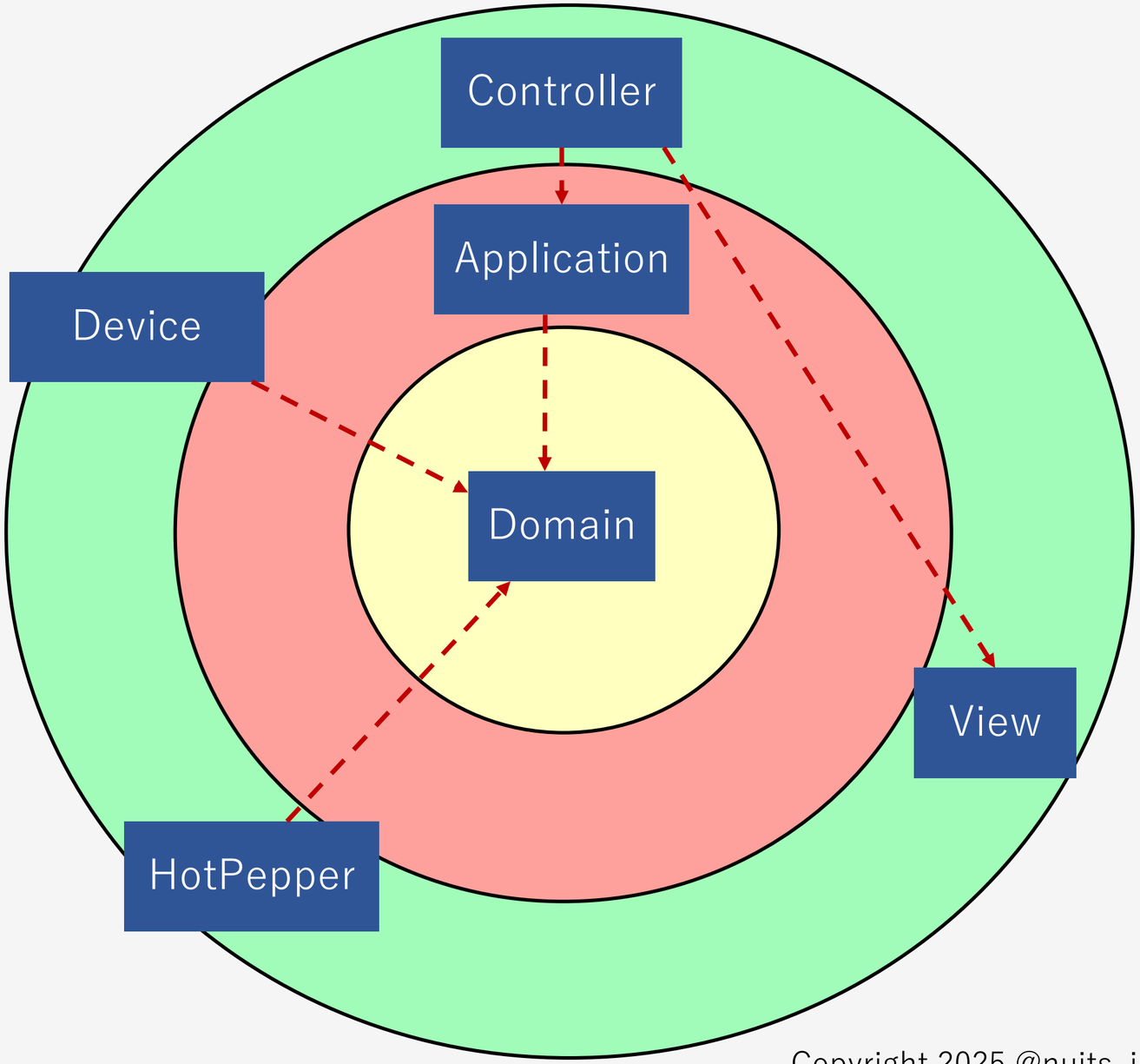
クラス図

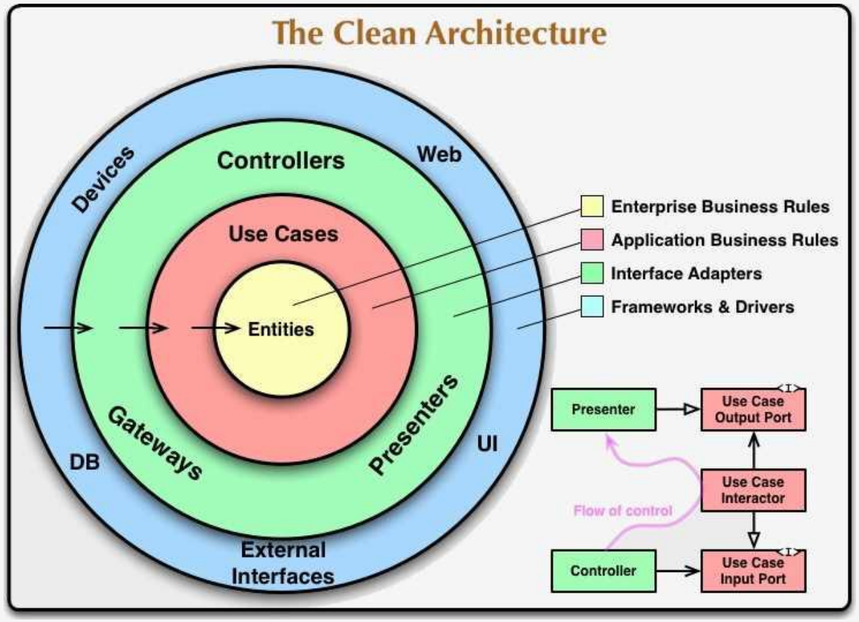


コンポーネント図



アプリケーション構造





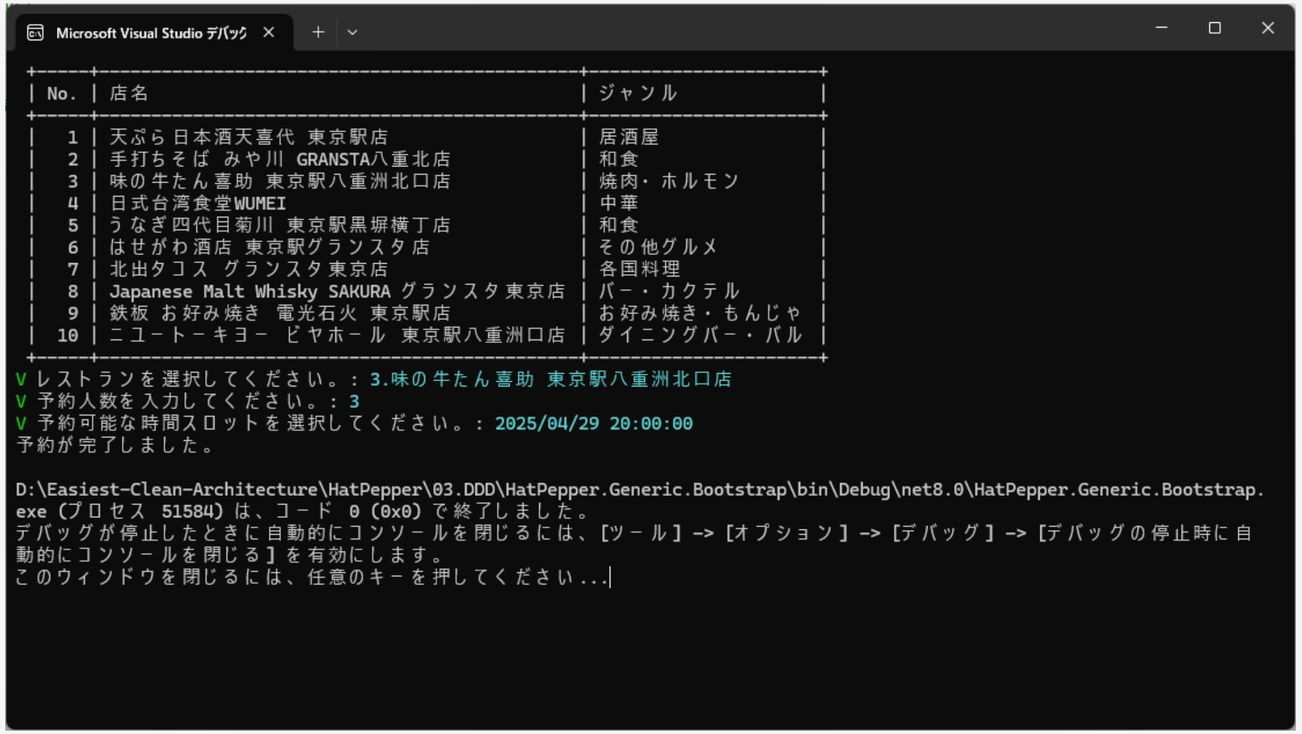
おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ
2. 制御の流れと依存方向は分離しコントロールせよ
3. 上位レベルとは相対的・再帰的であることに留意せよ

視点を上げてみよう

アプリケーション「HatPepper」※

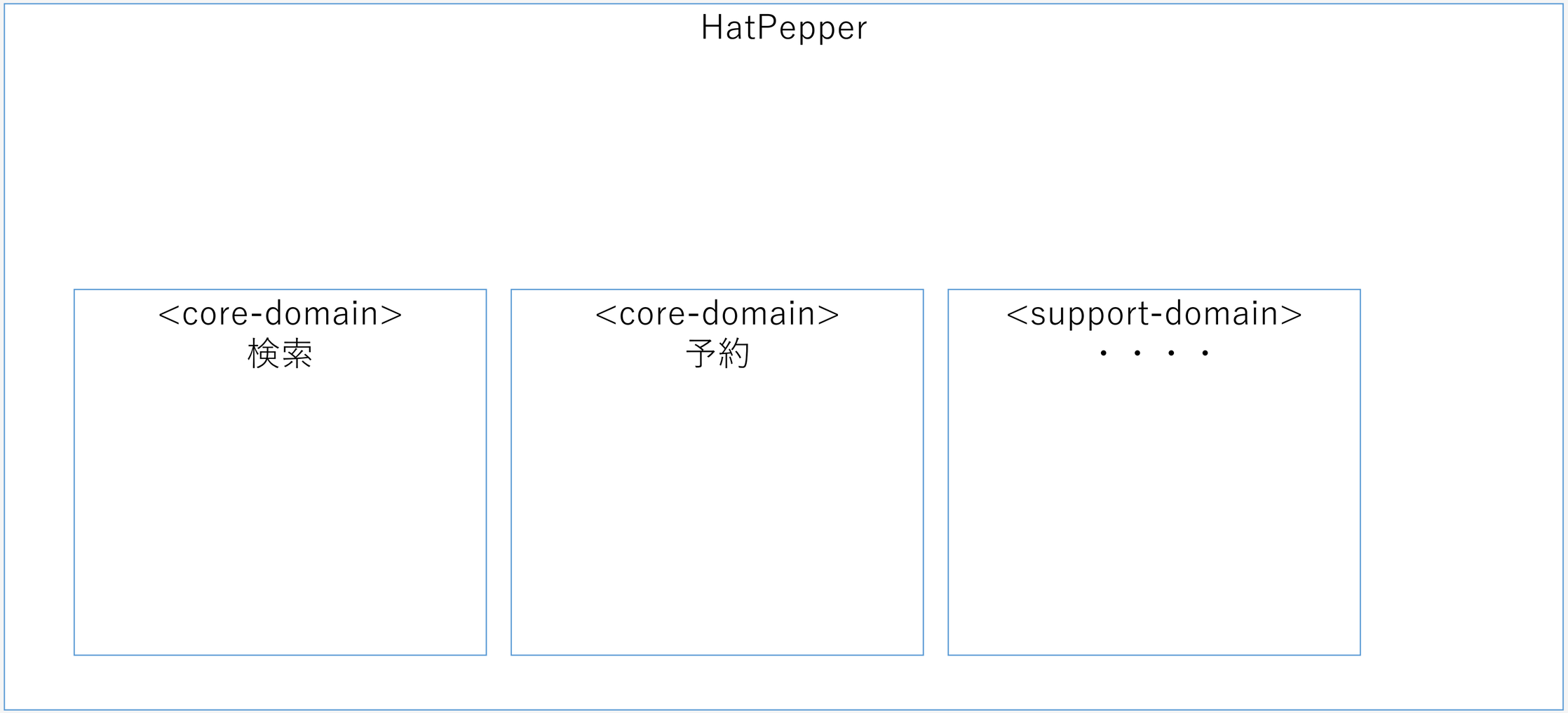
- 位置情報から周辺の店舗を検索**し**
  予約するグルメサイト
- 「リクルートWebサービス」の
  「グルメサーチAPI」を利用させていただく
  （いつもお世話になっております。）



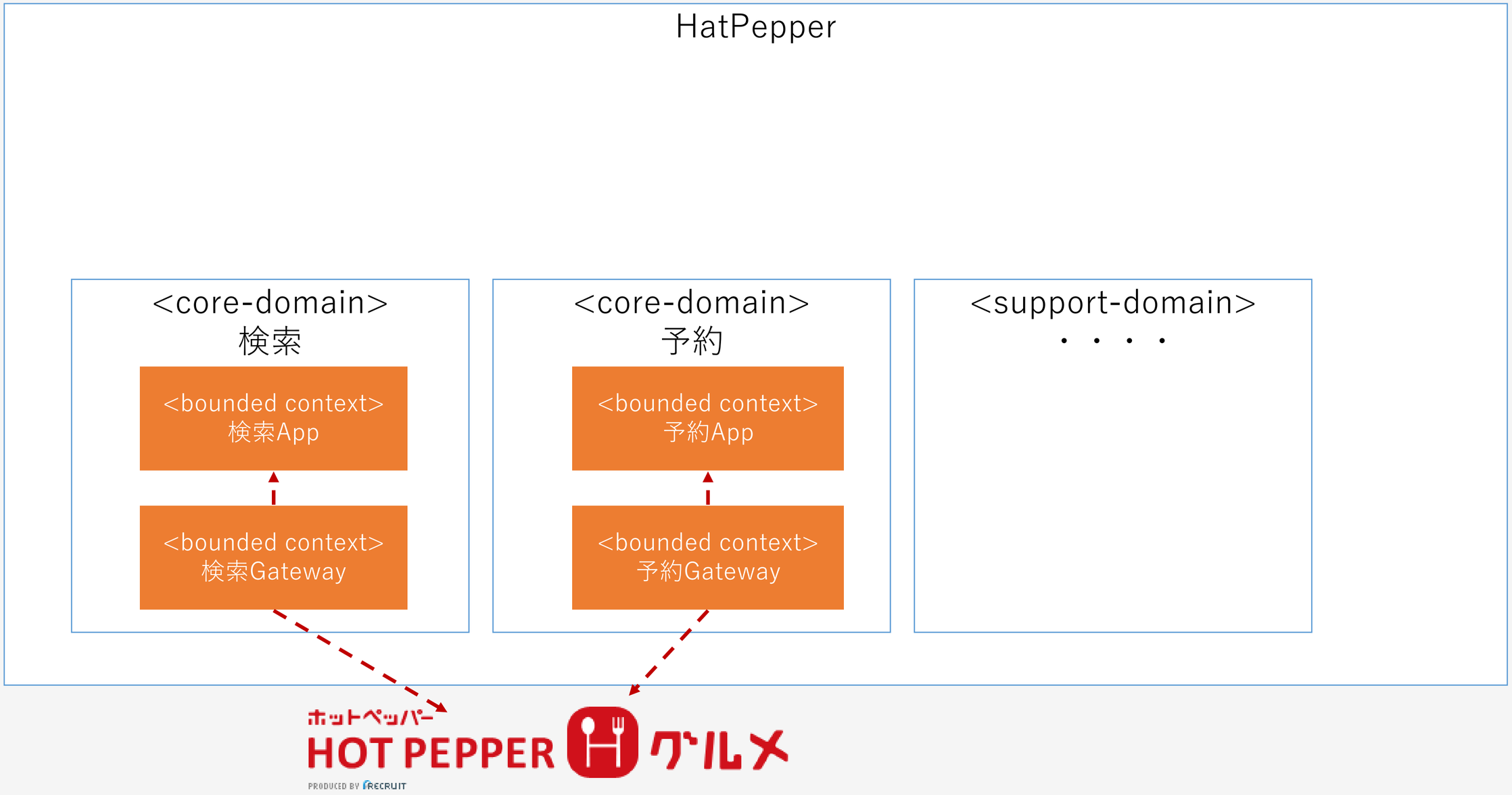
※「ホットペッパー」は株式会社リクルート様の登録商標です。

※「HatPepper」は登録商標ではありません。安心。

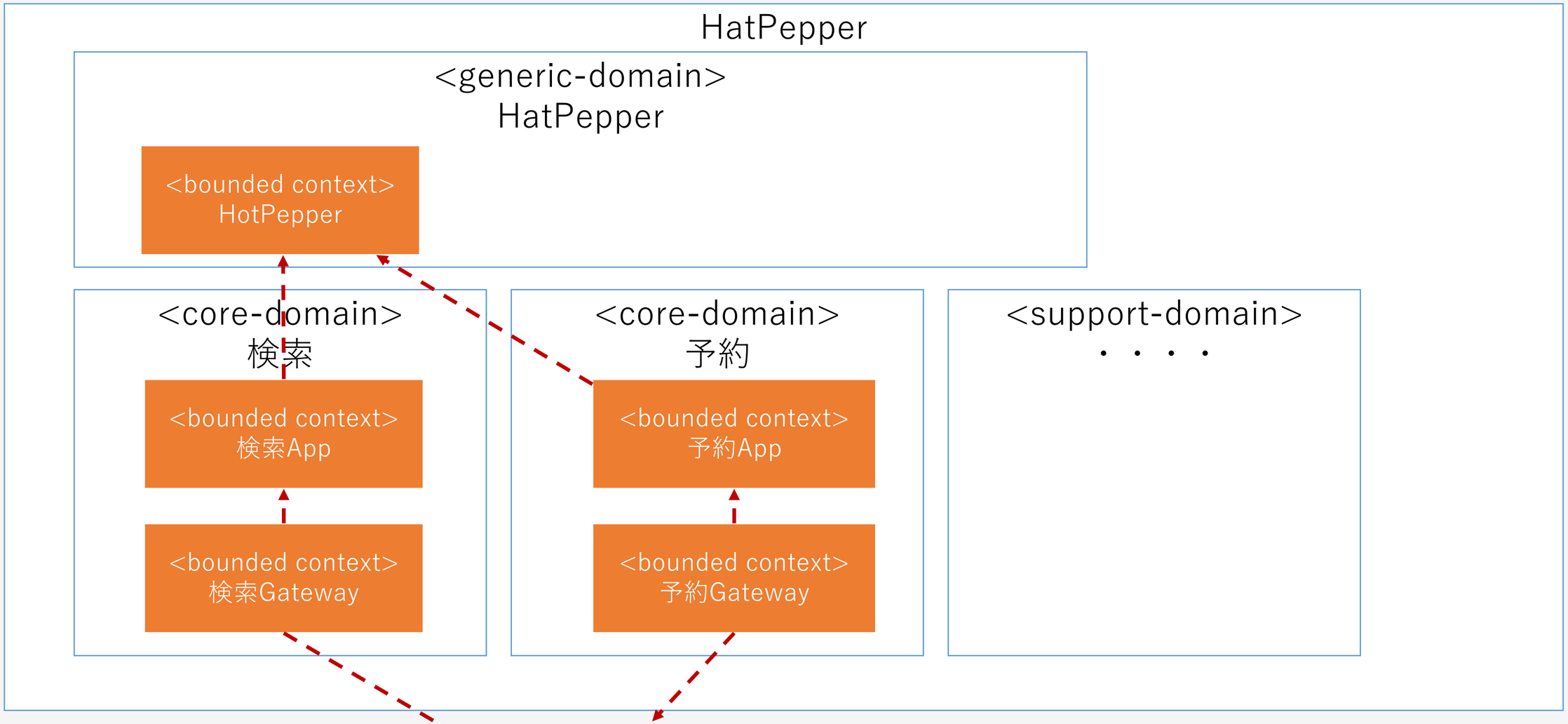
コンテキストマップ



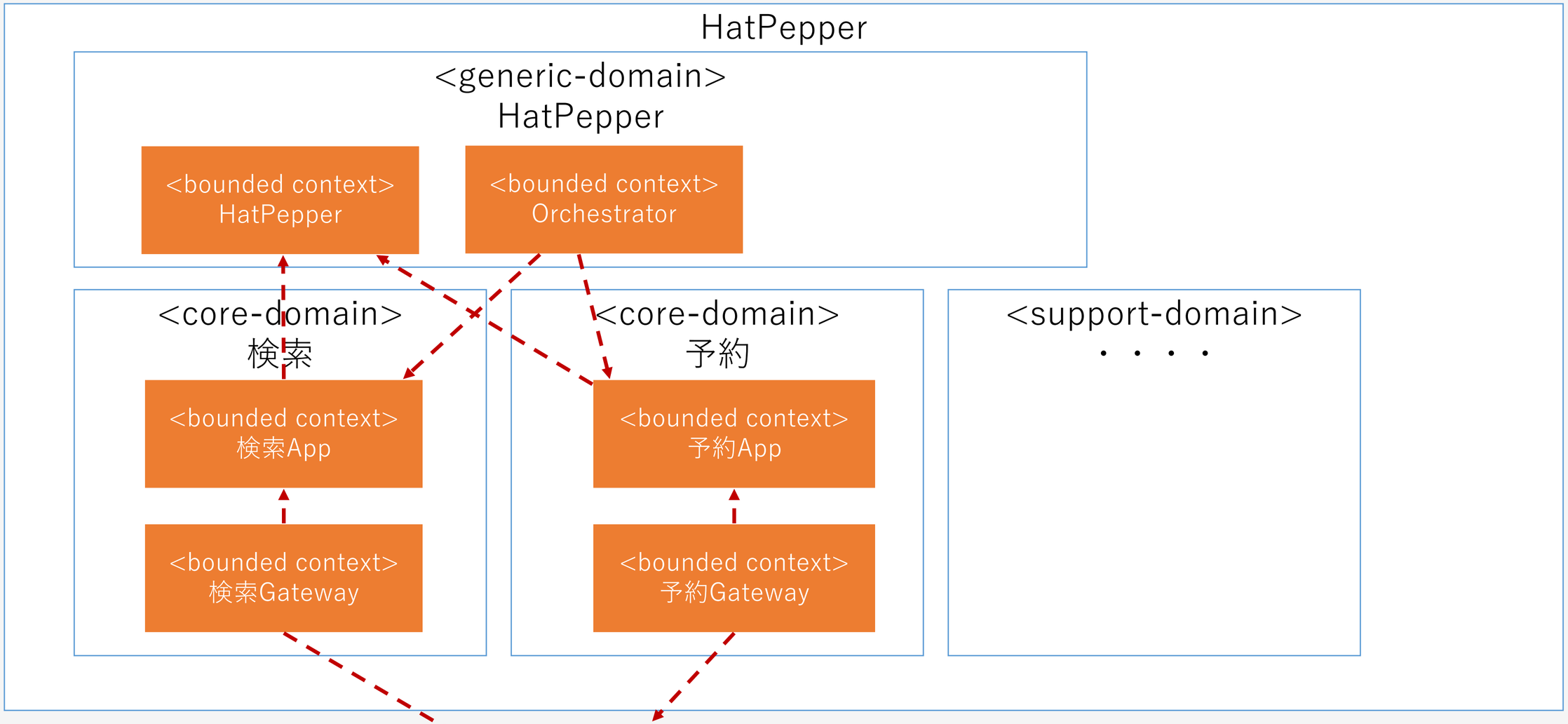
コンテキストマップ



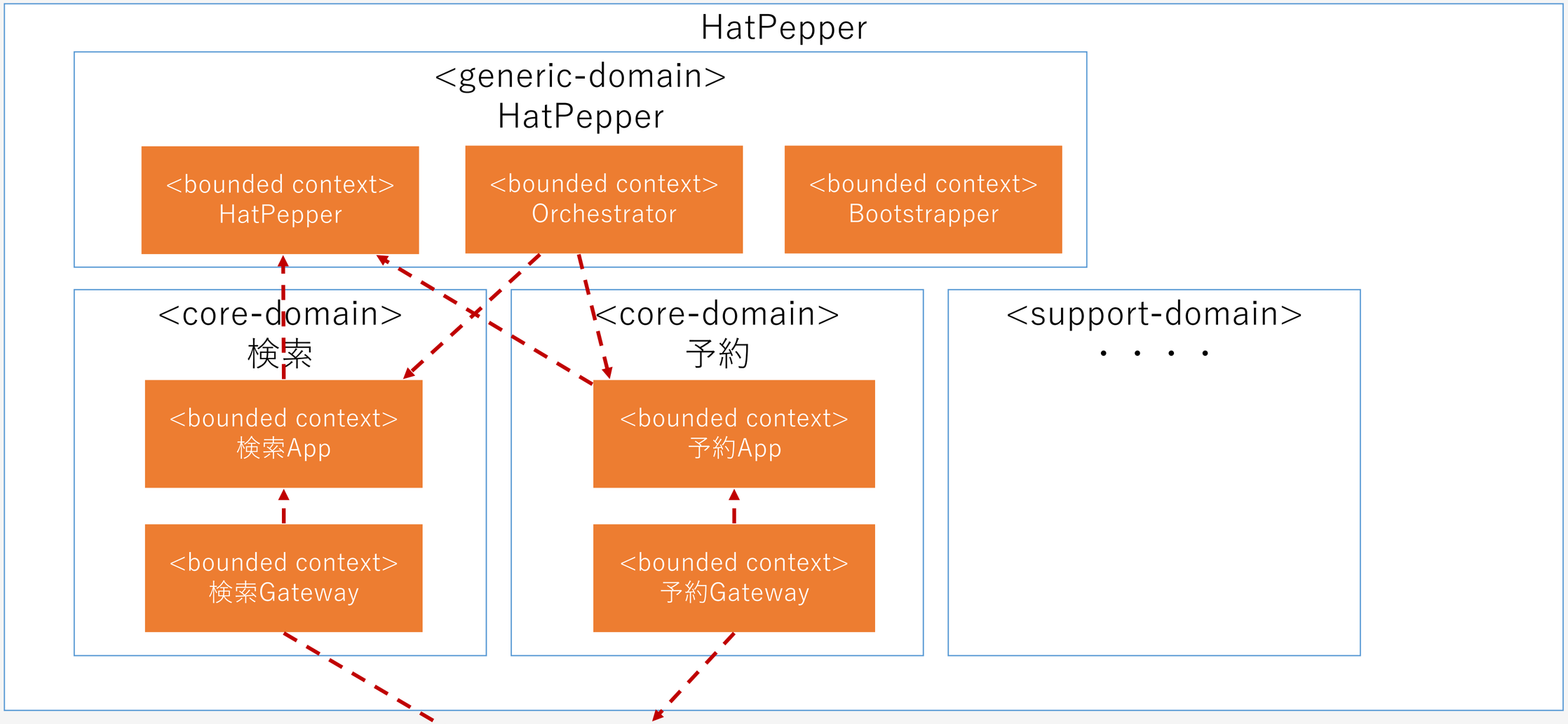
コンテキストマップ



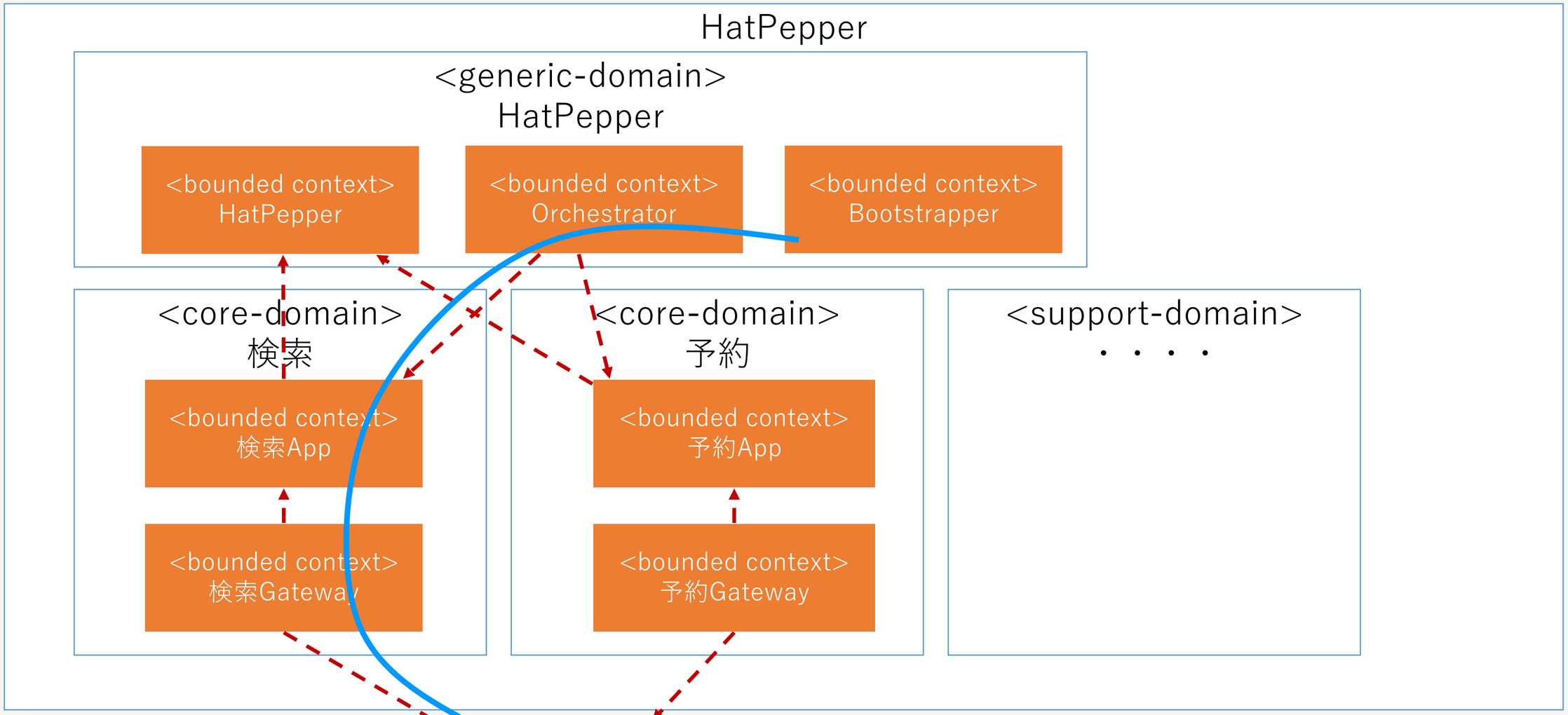
コンテキストマップ



コンテキストマップ



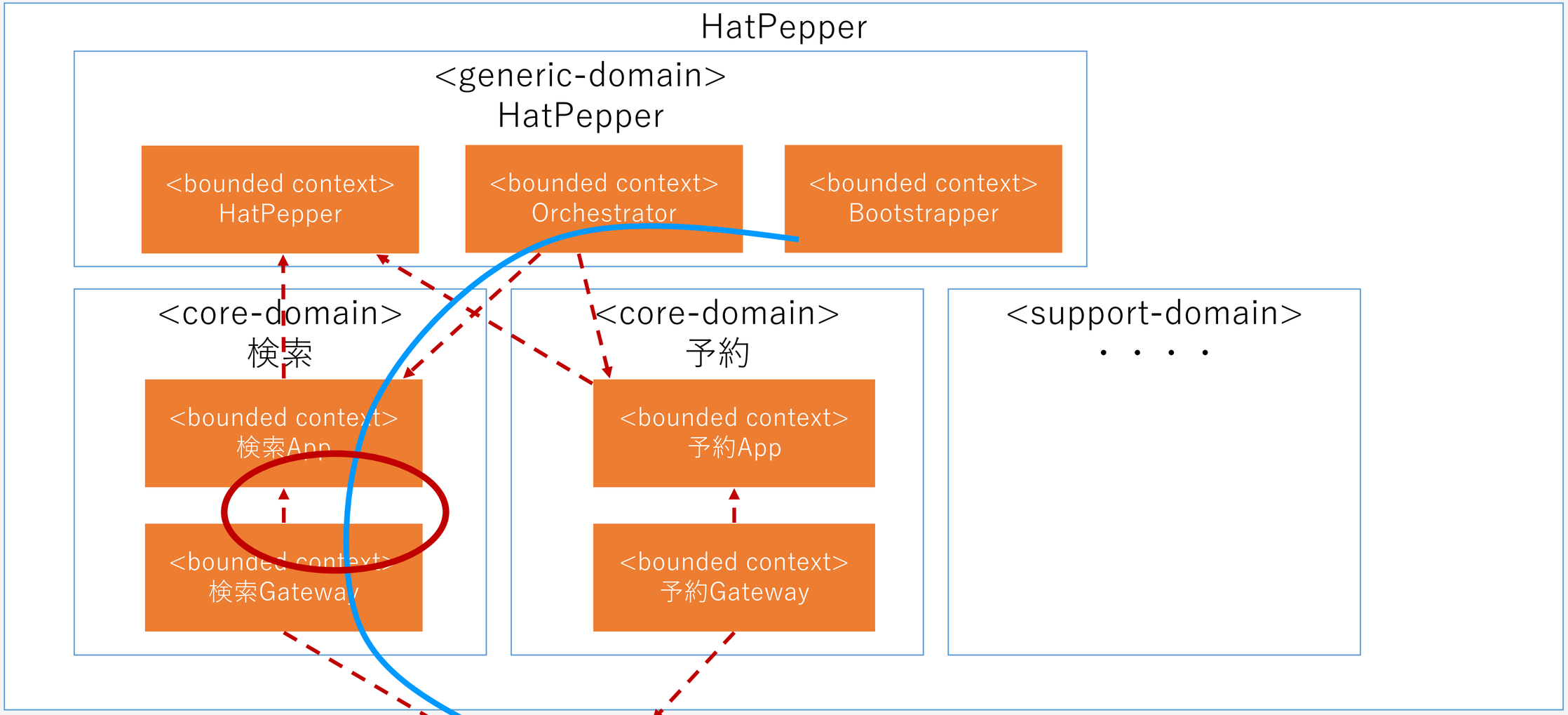
コンテキストマップ



ホットペッパー
HOT PEPPER グルメ
PRODUCED BY RECRUIT

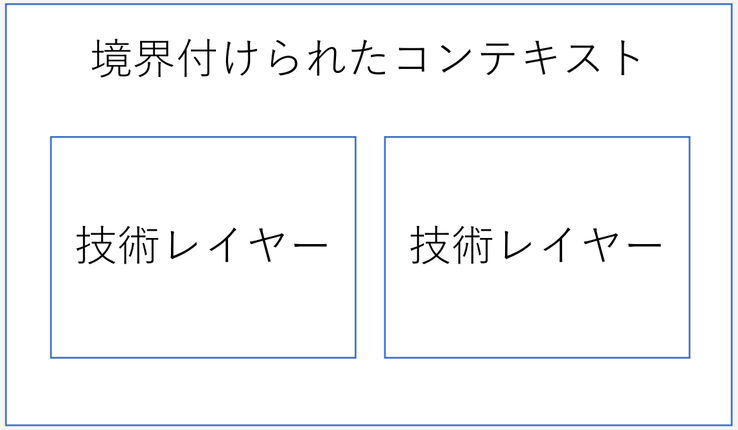
上位レベルとは相対的・再帰的であることに留意せよ

コンテキストマップ

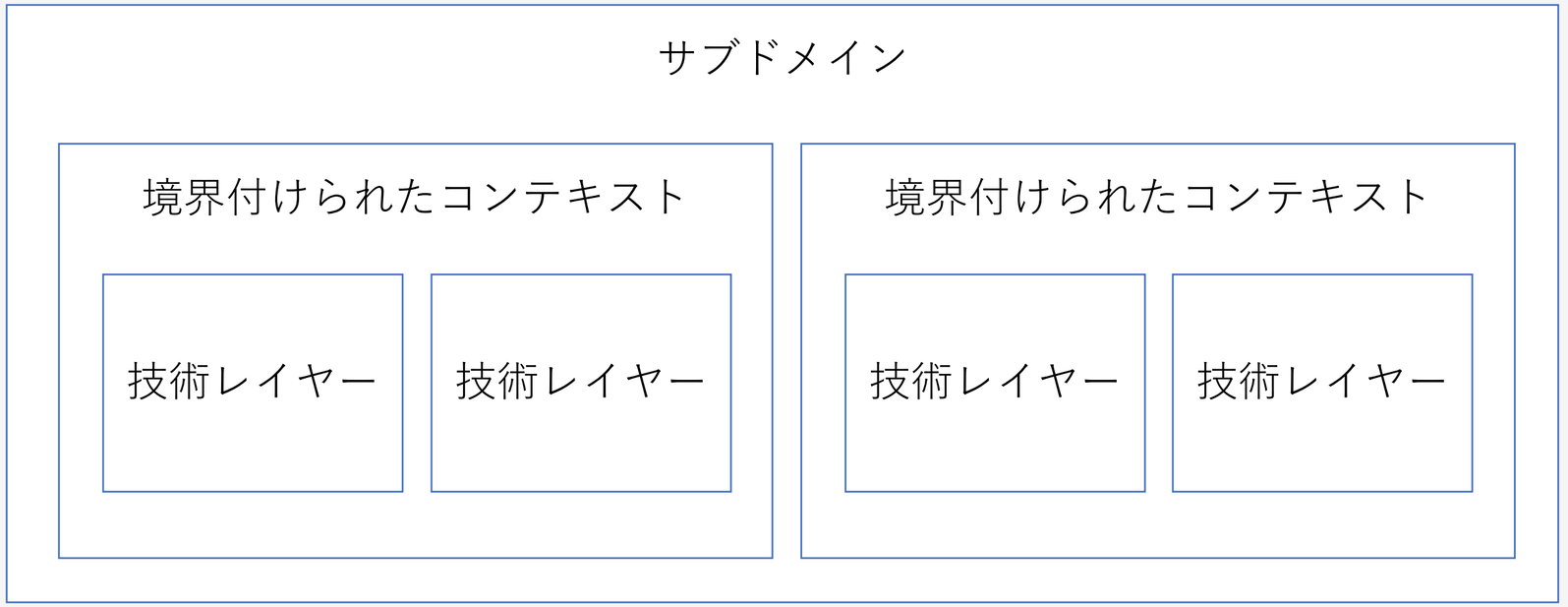


ホットペッパー
HOT PEPPER グルメ
PRODUCED BY RECRUIT

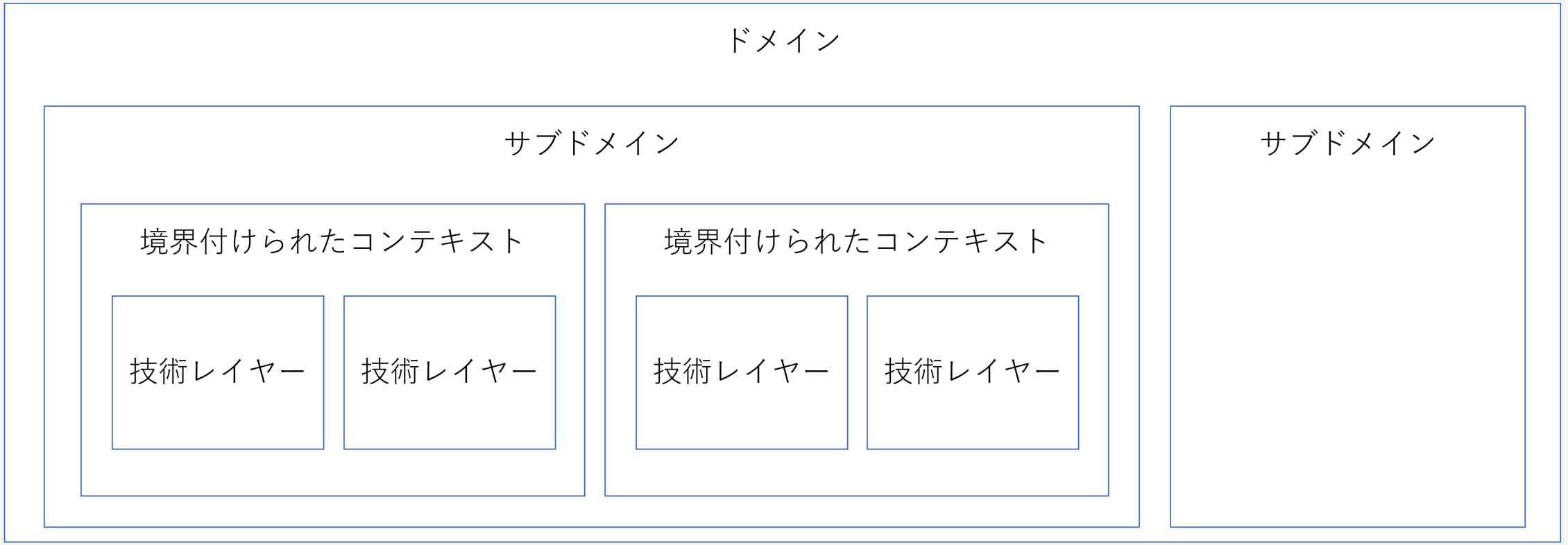
ソフトウェアのフラクタル



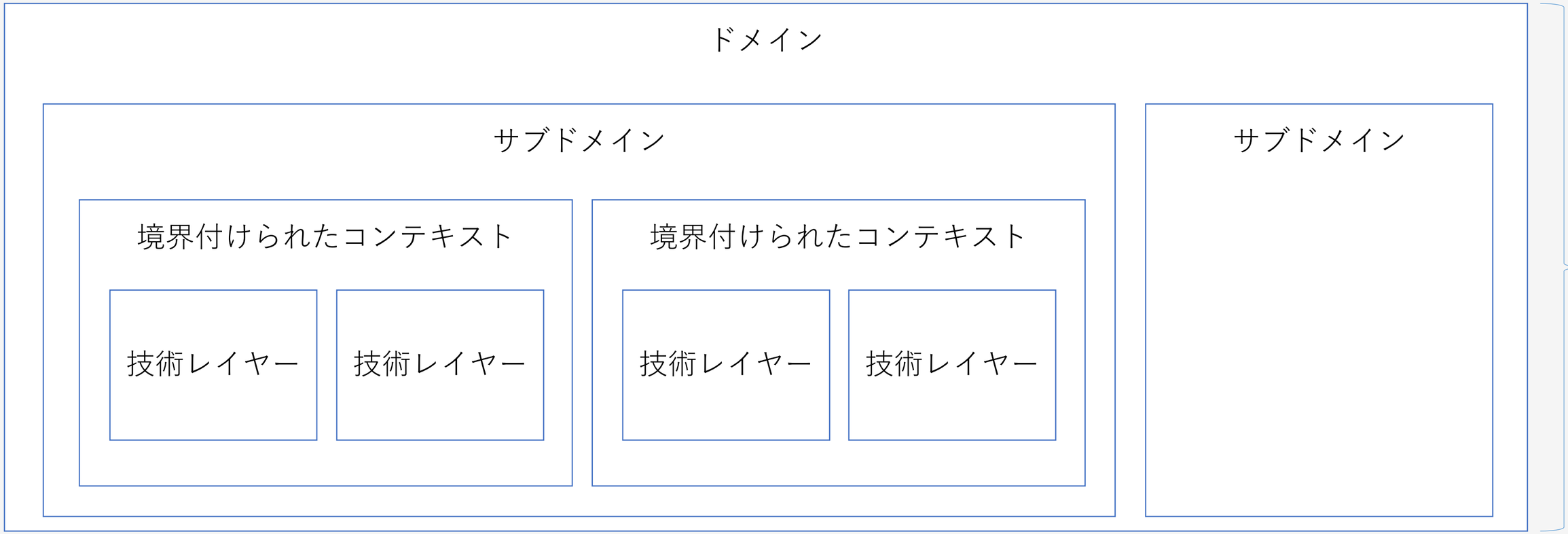
ソフトウェアのフラクタル



ソフトウェアのフラクタル



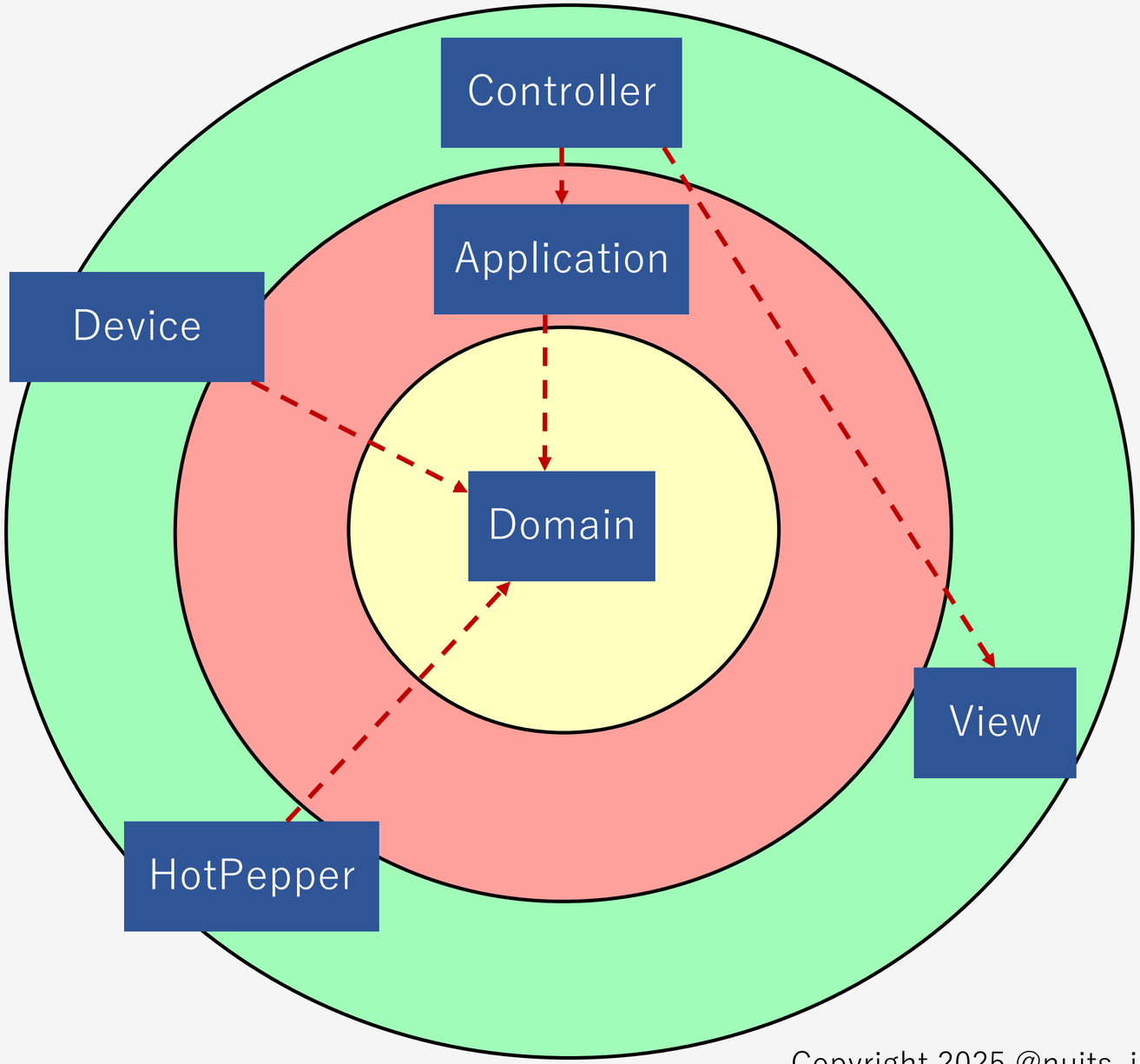
ソフトウェアのフラクタル

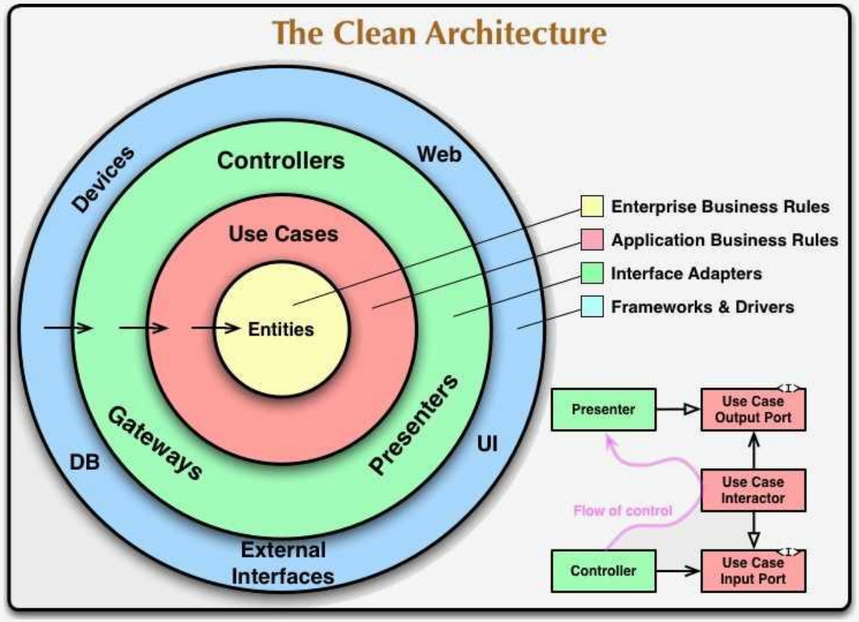


ソフトウェア オブジェクトすべてに、ここまでの話は適用可能

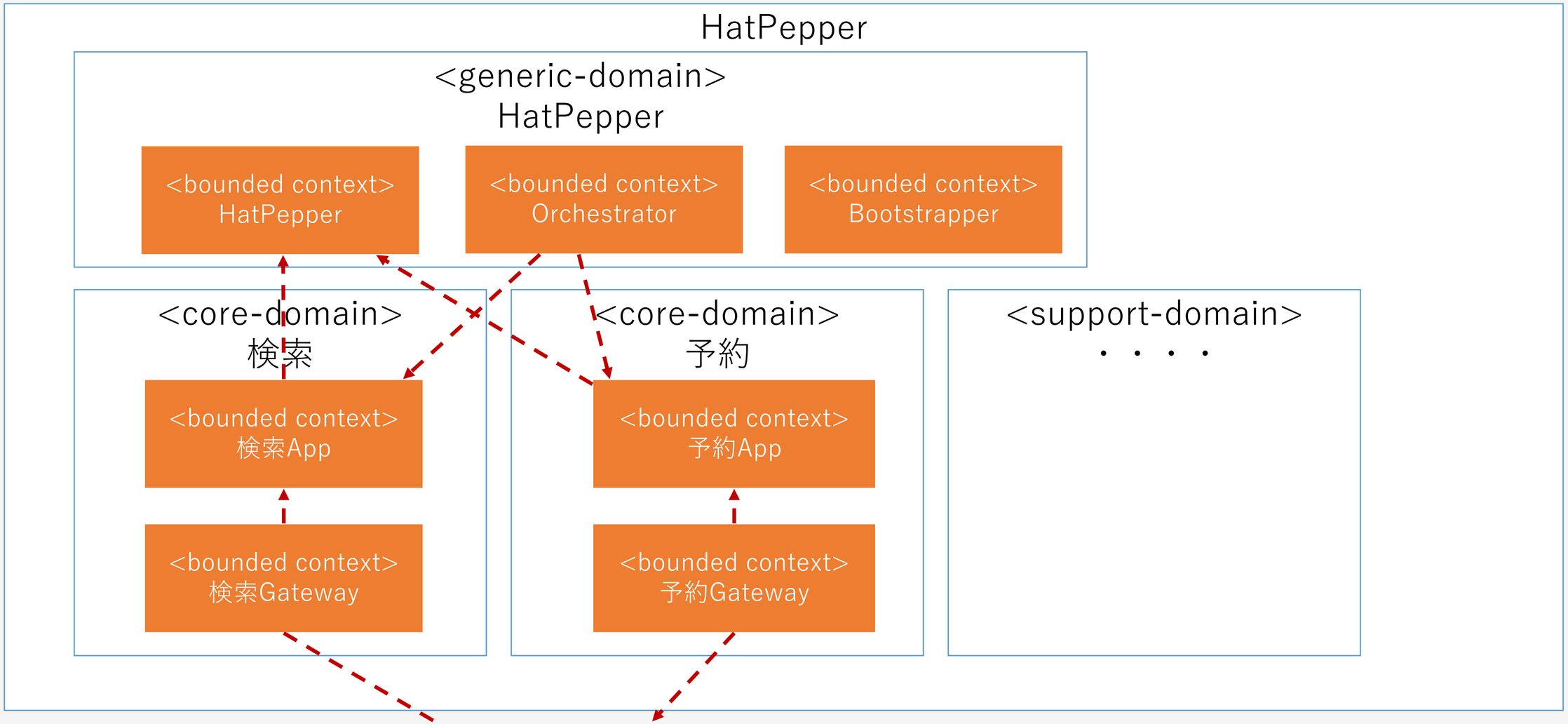
- 要素間のインターフェースの文脈により、制御の流れと依存関係は制御可能
- 依存関係によって、安定性と柔軟性をコントロール可能

アプリケーション構造

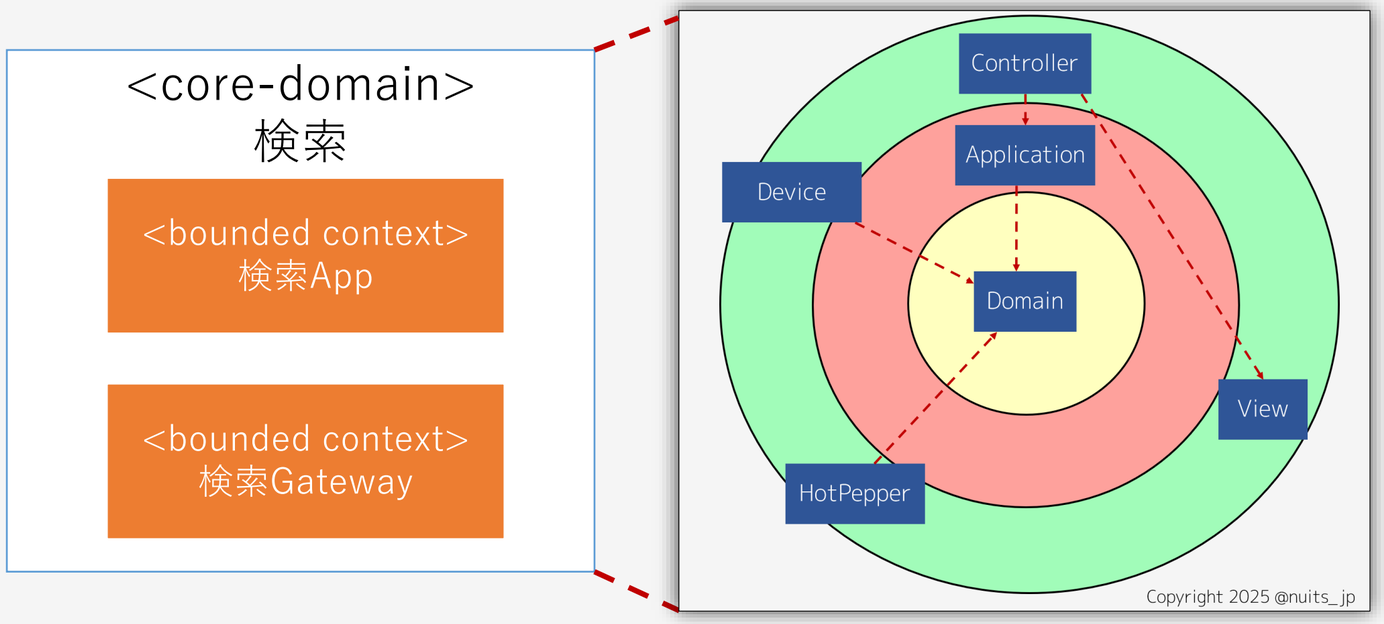




コンテキストマップ



コンテキストマップ



- The architecture rules are the same! -

Easiest Clean Architecture

What is Software Architecture ?

！ここからは私見成分高め！

ソフトウェア アーキテクチャといえば何を思い浮かべますか？

1. MVC
2. MVP
3. MVVM
4. MVPVM
5. クリーン アーキテクチャ
6. レイヤー アーキテクチャ
7. オニオン アーキテクチャ

などなど

これらはソフトウェア アーキテクチャの類型的なパターン群です

ソフトウェア アーキテクチャとは何か？

ソフトウェアアーキテクチャでは、ソフトウェア システムの構成に関する一連の重要な判断を網羅しています。これには、システムを構成する要素とインターフェイスの選択、要素間のコラボレーションとして指定される動作、このような構成と動作の要素のより大きなサブシステムに対する構成、この構成の指針となるアーキテクチャスタイルが含まれます。また、機能性、ユーザビリティ、復元性、パフォーマンス、再利用性、理解できること、経済的な制約、テクノロジの制約、トレードオフ、および外観への配慮も必要です。

by Philippe Kruchten, Grady Booch, Kurt Bittner, Rich Reitman

Microsoft 「アプリケーション アーキテクチャ ガイド2.0」より引用

ソフトウェア アーキテクチャとは何か？

アーキテクチャは、システムを大きなレベルで分解したもので、決定事項の変更は困難です。システムには複数のアーキテクチャが存在し、アーキテクチャにとって重要な事項はシステムの運用中に変化します。要するに、重要な要素は、すべてアーキテクチャということになります。

by Martin Fowler

Microsoft 「アプリケーション アーキテクチャ ガイド2.0」より引用

ソフトウェア アーキテクチャとは何か？

ソフトウェアアーキテクチャとは、抽象化と問題の分割によって複雑性を減らすことを主に念頭に置いたものである。ただし、今までのところ、「ソフトウェアアーキテクチャ」という用語に関して、万人が合意した厳密な定義は存在しない

Wikipedia「ソフトウェアアーキテクチャ」より引用

とは言え、おおよその共通認識はある

Slide 138

ソフトウェア アーキテクチャとは何か？かみ砕くと…

1. ソフトウェアアーキテクチャとは
   システムアーキテクチャのうち、ソフトウェア領域のアーキテクチャである
   この時「システム」とは「IT」だけではなく、それらを取り巻く社会やビジネスを含めた「仕組み」を指すことも多い

※ 元来「System」とは制度・組織・体系・系統などのこと

2. ソフトウェア アーキテクチャとは
    1. ソフトウェアにおける重要な決定事項全てである
    2. その中でも特につぎの2点が重要
        1. ソフトウェア全体を、どのように分割するか
        2. 分割した部分同士を、どう結合し相互作用させるか

ソフトウェア アーキテクチャとは何か？かみ砕くと…

3. ソフトウェアアーキテクチャを構築するのはなぜか？
   - 「システムの実現をサポートする」
   - 「持続可能なソフトウェア」を
   - 「バランスよく」構築するため

4. ソフトウェアアーキテクチャのバランスとは？
   1. Quality（機能・非機能）
   2. Cost
   3. Delivery（開発期間）

   アーキテクチャは非機能要求からも大きく影響をうける

5. Clean Architectureによると・・・
   アーキテクチャは、次のものを与えてくれます
   - フレームワーク非依存
   - テスト可能
   - UI非依存
   - データベース非依存
   - 外部エージェント非依存

Clean Architectureを考えるうえで重要なポイント

2. ソフトウェア アーキテクチャとは
    1. ソフトウェアにおける重要な決定事項全てである
    2. その中でも特につぎの2点が重要
        1. ソフトウェア全体を、どのように分割するか
        2. 分割した部分同士を、どう結合し、相互作用させるか

ソフトウェア アーキテクチャとは何か？かみ砕くと…

2. ソフトウェア アーキテクチャとは
    1. ソフトウェアにおける重要な決定事項全てである
    2. その中でも特につぎの2点が重要
        1. ソフトウェア全体を、どのように分割するか
        2. 分割した部分同士を、どう結合し、相互作用させるか

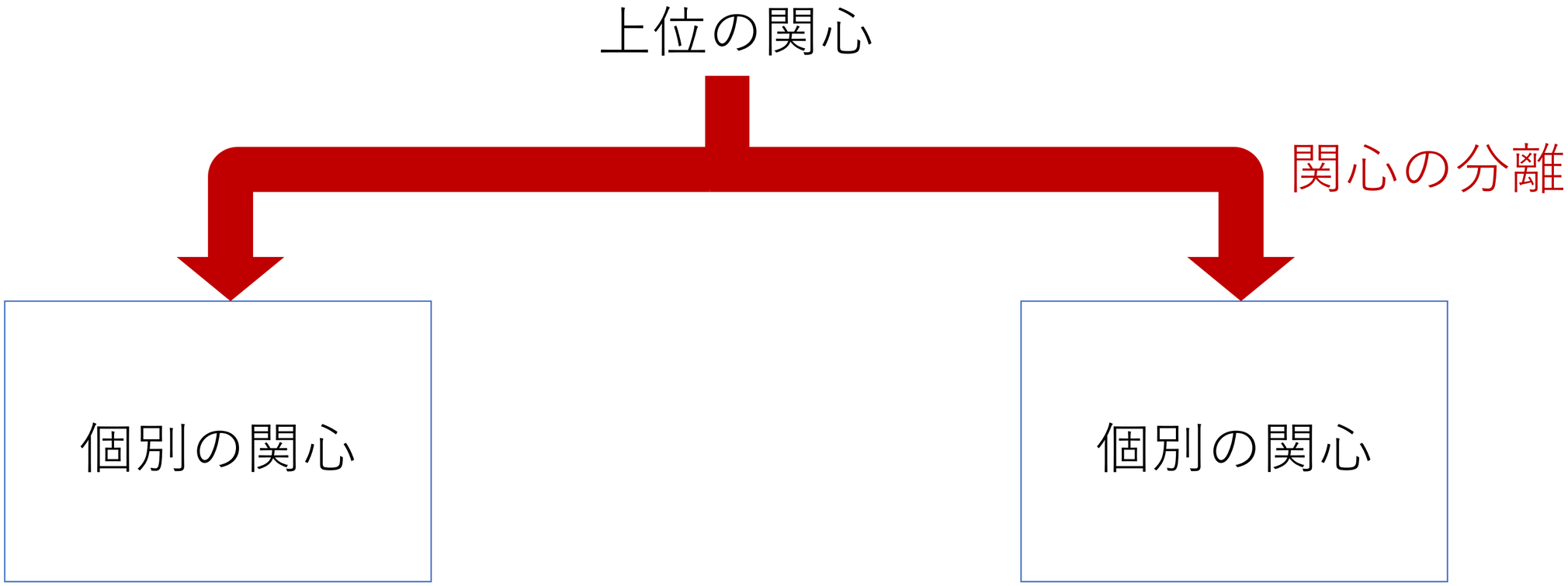
ソフトウェアアーキテクチャ3種の神器

ソフトウェアアーキテクチャ3種の神器

1. 関心の分離（SoC：Separation of concerns）
2. 疎結合
3. 依存性逆転の原則

上位の関心

ソフトウェア アーキテクチャの3種の神器



ソフトウェア アーキテクチャの3種の神器

上位の関心	個別の関心
ドメイン	サブドメイン
サブドメイン	境界付けられたコンテキスト
境界付けられたコンテキスト	技術レイヤー

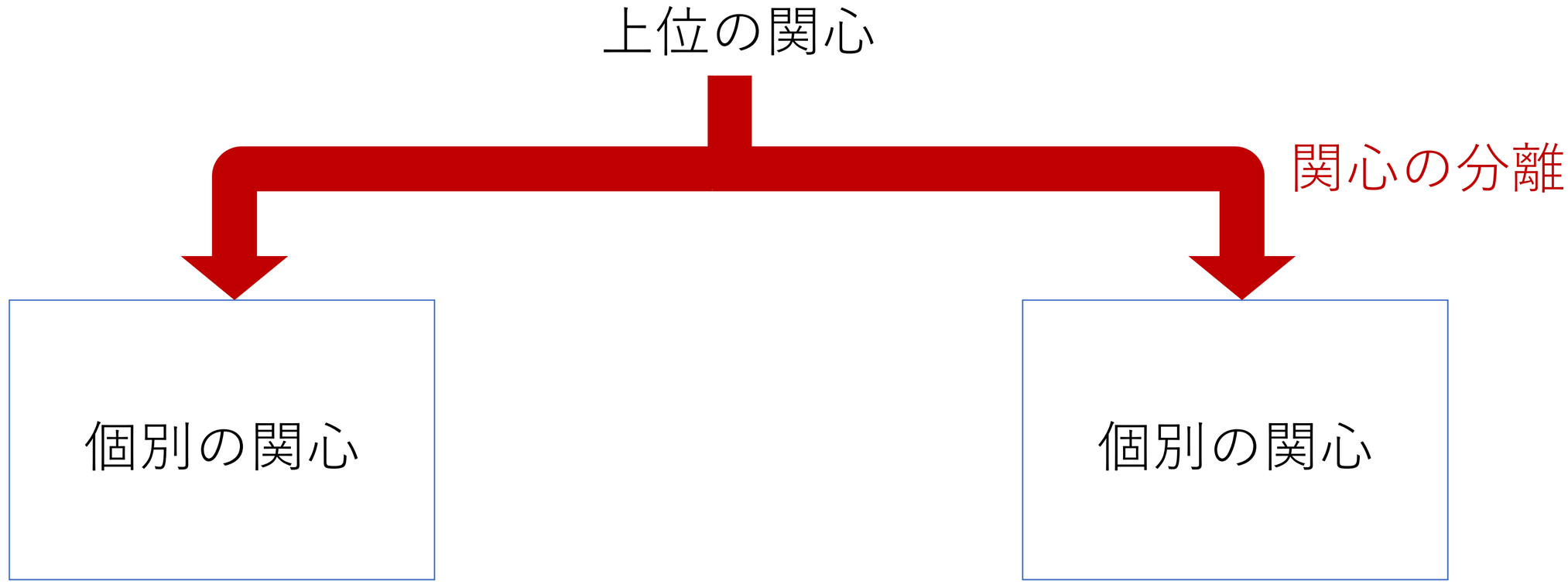
ソフトウェアアーキテクチャの3種の神器

直交する関心に注意

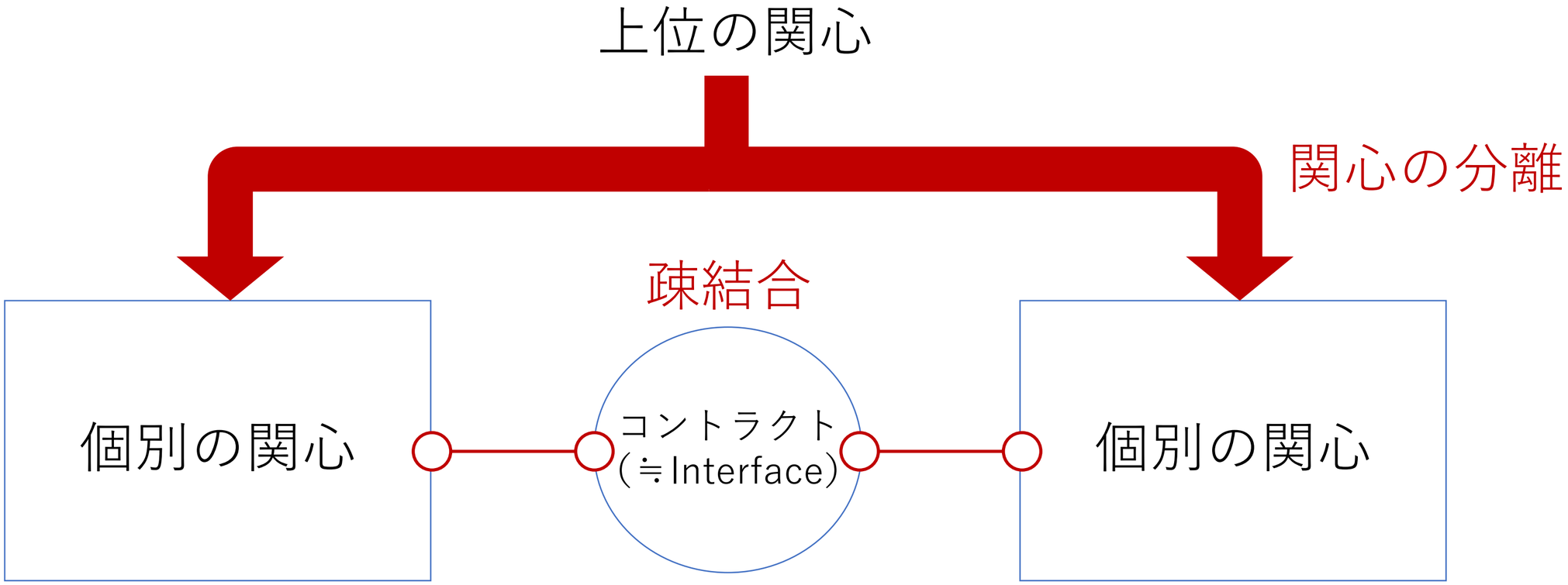
関心の分離

上位の関心	個別の関心
ドメイン	サブドメイン
サブドメイン	境界付けられたコンテキスト
境界付けられたコンテキスト	技術レイヤー

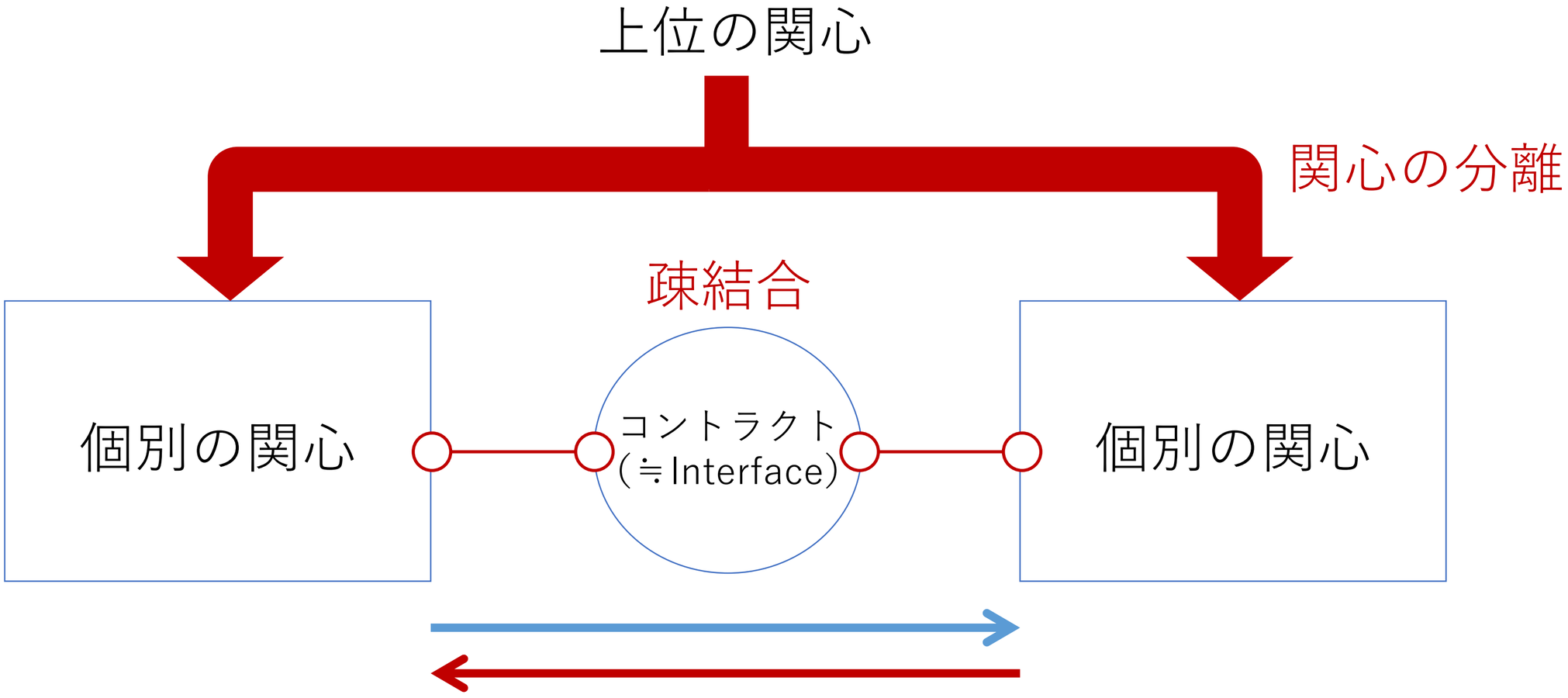
ソフトウェア アーキテクチャの3種の神器



ソフトウェア アーキテクチャの3種の神器



ソフトウェア アーキテクチャの3種の神器



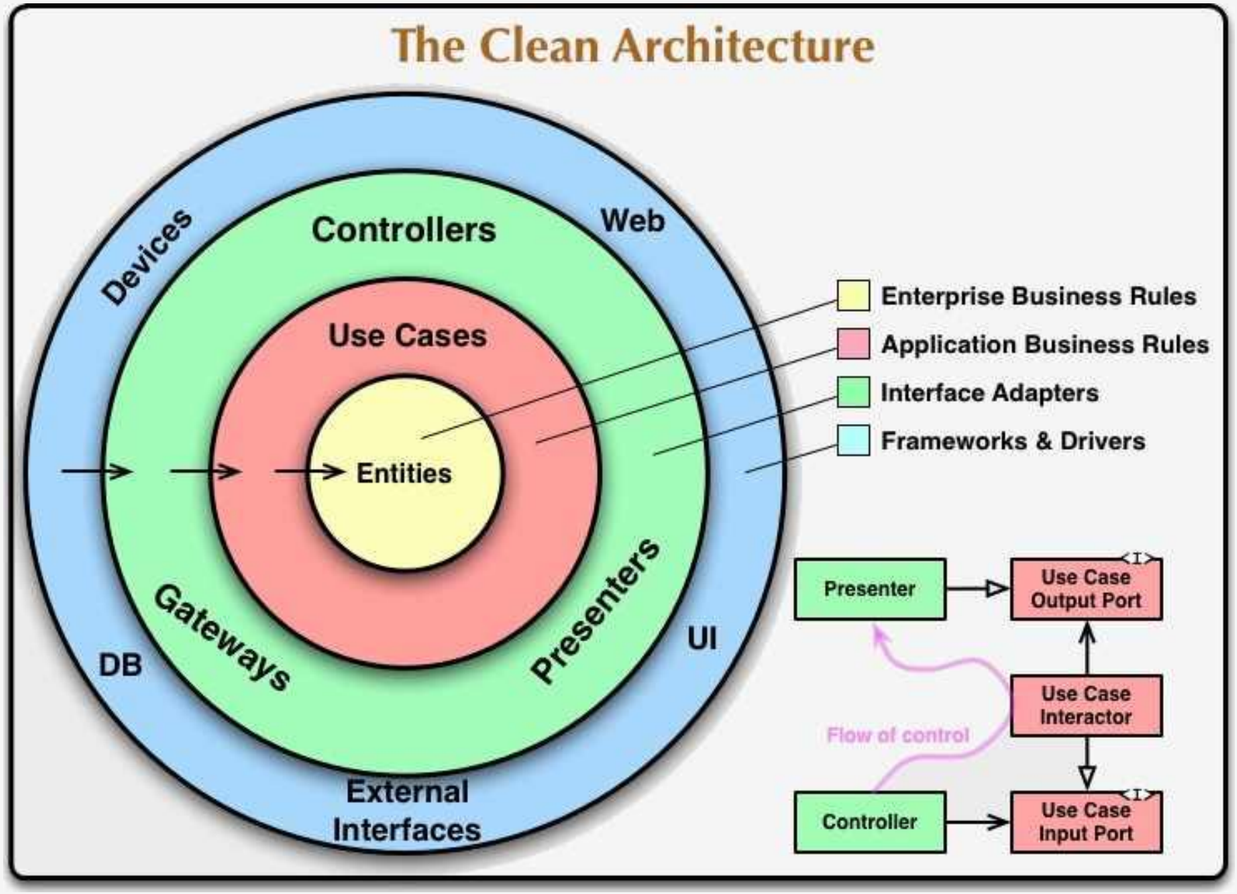
まとめ

まとめ

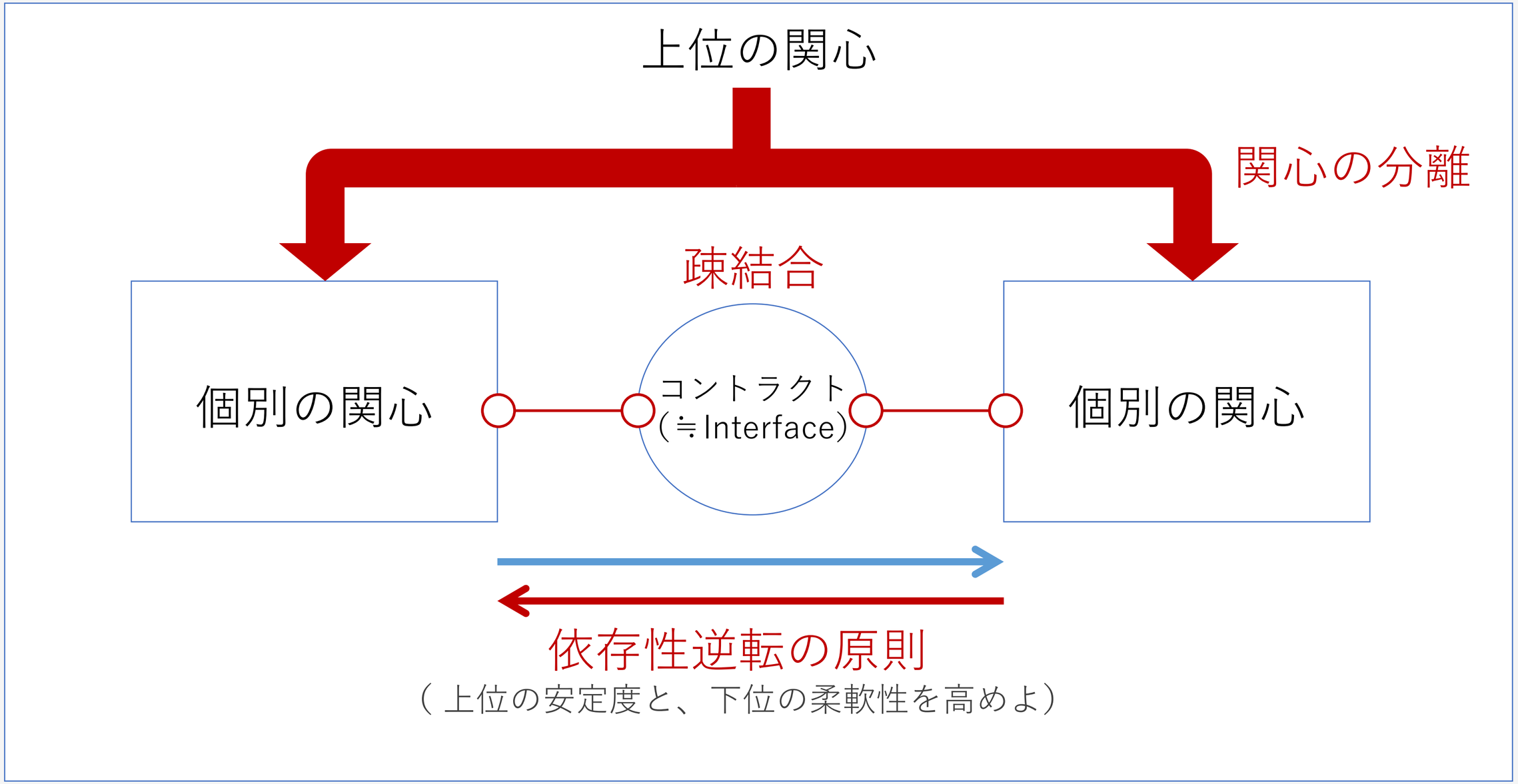
Clean Architectureとは、どんなソフトウェアにも適用可能な
普遍的なアーキテクチャの原則である

Clean Architecture is ...

クリーンアーキテクチャとは
「ソフトウェアアーキテクチャ3種の神器を適用した
リファレンスアーキテクチャのひとつ」



ソフトウェア アーキテクチャの3種の神器



1. 依存性は、より上位レベルの方針に**のみ**向けよ

おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ

2. 制御の流れと依存方向は分離しコントロールせよ

おさえるべき三つのこと

1. 依存性は、より上位レベルの方針に**のみ**向けよ
2. 制御の流れと依存方向は分離しコントロールせよ
3. 上位レベルとは相対的・再帰的であることに留意せよ

誤解されがちな二つのこと

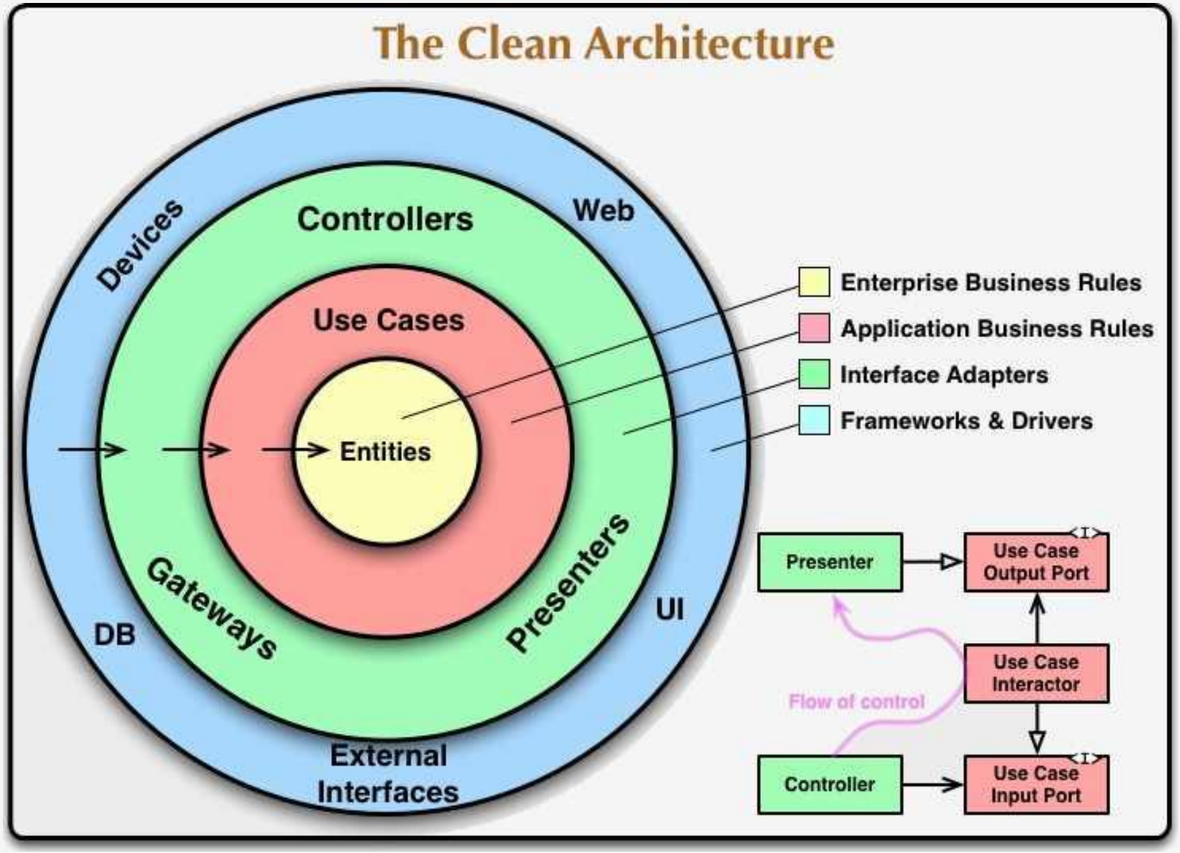
Clean Architectureは、次のふたつを誤解されがち

1. 参考図の通りに関心を分離するのがClean Architectureである
2. データや処理の流れを一方通行にしろ

上記は、陥りがちな誤りであり、Clean Architectureの本質とは異なります。

Clean Architectureの本当の価値

- 「この図」が実現しようとした目的
- 「この図」に至った背景
- 「この図」を実現する手段



一読の価値あり

目的・背景・手段
が凝縮されている

Slide 163

という訳で・・・

Today’s Goal

みなさん

「クリーンアーキテクチャチョットデ゛キル」

ようになりましたね？

Thank You!

Any Questions?