Read/Write Stack | JavaScriptアーキテクチャ

自己紹介

- Name : **azu**
- Twitter : @azu_re
- Website: Web scratch, JSer.info

This is Bikeshed.js 🚲

抽象的な話が多いので、実装はコード見て

(Pull Request投げて！)

これが正しいという話ではないです。

自転車置き場の議論なので！

中規模以上のJavaScript

- 設計が必要になる
- 正しい設計はない Bikeshed.js 🚲
  - 人、目的、何を作るかによってアーキテクチャは異なる
- 前回の続き? : How to work as a Team

用語

ドメインモデル	プレゼンテーション、アプリケーション、ドメイン、インフラストラクチャの4つのレイヤーに基づき、DDD スタイルで設計された階層化アーキテクチャ。とりわけ、モデルが特殊なオブジェクトモデルであることが想定される
コマンド／クエリ責務分離（CQRS）	コマンド部分とクエリ部分を処理するための並列セクションを持つ二重の階層化アーキテクチャ。これらのセクションは別々に設計することが可能で、DDD やクライアント／サーバーなど、異なるサポートアーキテクチャを使用することもできる
イベントソーシング	ほとんどの場合は、CQRS にヒントを得て、単純なデータではなくイベントのロジックに焦点を合わせた階層化アーキテクチャ。イベントはファーストクラスのデータとして扱われる。その他の問い合わせ可能な情報は格納されているイベントから推測される

ドメインモデル	プレゼンテーション、アプリケーション、ドメイン、インフラストラクチャの4つのレイヤーに基づき、DDD スタイルで設計された階層化アーキテクチャ。とりわけ、モデルが特殊なオブジェクトモデルであることが想定される
コマンド／クエリ責務分離（CQRS）	コマンド部分とクエリ部分を処理するための並列セクションを持つ二重の階層化アーキテクチャ。これらのセクションは別々に設計することが可能で、DDD やクライアント／サーバーなど、異なるサポートアーキテクチャを使用することもできる
イベントソーシング	ほとんどの場合は、CQRS にヒントを得て、単純なデータではなくイベントのロジックに焦点を合わせた階層化アーキテクチャ。イベントはファーストクラスのデータとして扱われる。その他の問い合わせ可能な情報は格納されているイベントから推測される

設計の目的

- 中規模以上のウェブアプリ
  - SPAというよりは、画面が複雑なElectronアプリのようなイメージ
- スケーラブル
  - 人、機能追加、柔軟性、独立性
- 見た目が複雑ではないアーキテクチャ
  - 書き方が特殊ではなく見て分かるもの

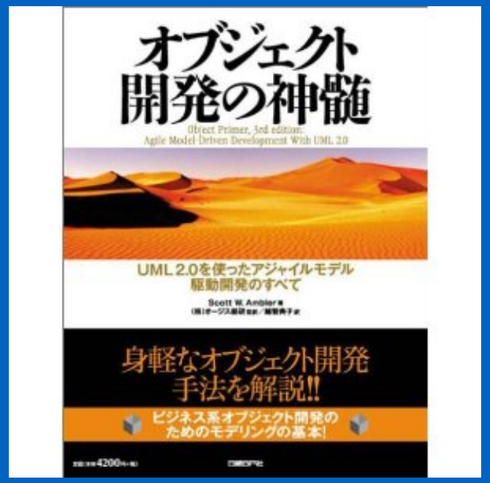
設計の目的

- テストが自然に書ける
  - パーツごとに無理なく依存を切り離せる
- 新しい機能を追加するときにどこに何があるかが分かる
- ドメインモデルを持てるようにする
  - わかりやすいモデルがありビジネスロジックを持つ
  - Fluxにおける「ドメインロジックをどこに実装するか」問題

要求にもとづいてアーキテクチャを作成する

要求にもとづいて作業をしていないアーキテクトは、
実際上「大仕掛なハッキング」をしているだけです。

– オブジェクト開発の神髄



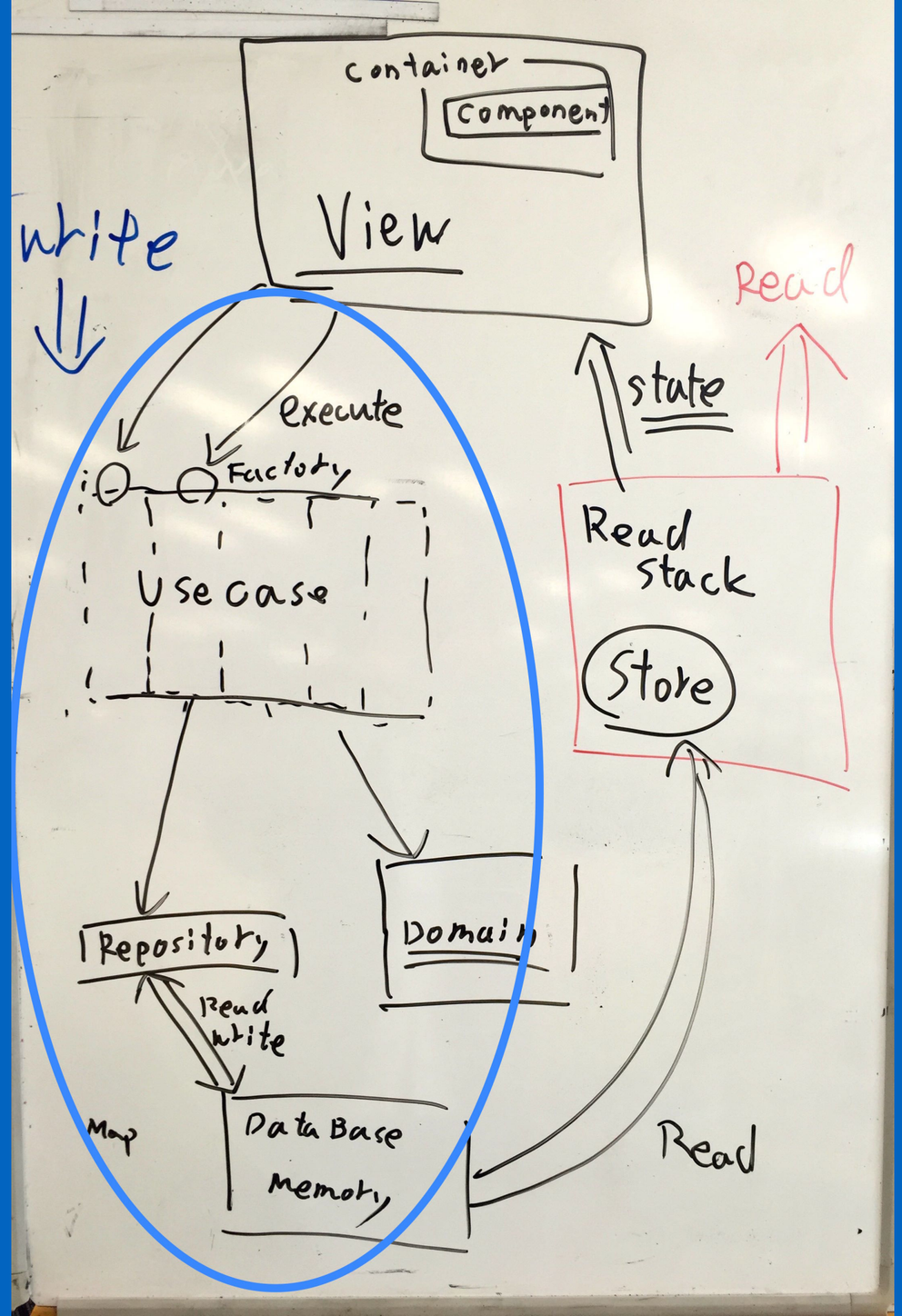
実装例

New framework by azu · Pull Request #7 · azu/presentation-annotator

考えるポイント

- クライアントサイドで問題点となるのはオブジェクトの永続化
  - シングルトンがでてくる問題
- Write StackとRead Stackを隔離する
  - データの流れがシンプルになる
- 結果統合性のみでは解決しない物がクライアントサイドにある
  - CQRS + Event Sourcing

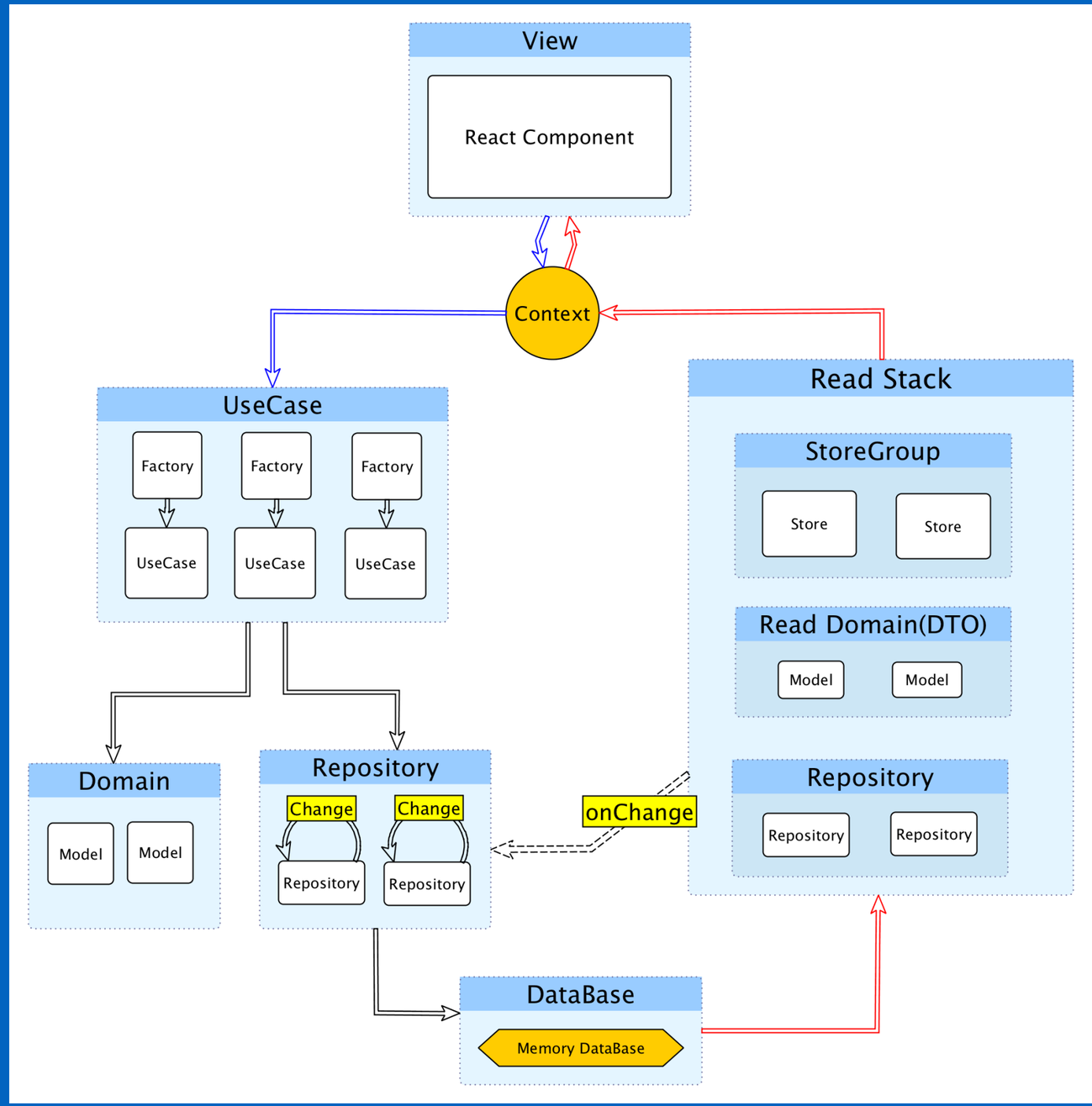
全体像(Simple版)


container
Component
View
write
Read
state
Execute
Factory
Usecase
Read
Stack
Store
Repository
Domain
Read
write
Map
Data Base
Memory
Read

全体像(Simple版)

画像は概念イメージで、データや処理の流れを表すものではありません

あえて表現するなら説明の流れにすぎません



登場人物

- View(React Component)
- Write Stack
  - UseCase
  - Domain
  - Repository ⬅
- Read Stack
  - Store
  - Repository ➡

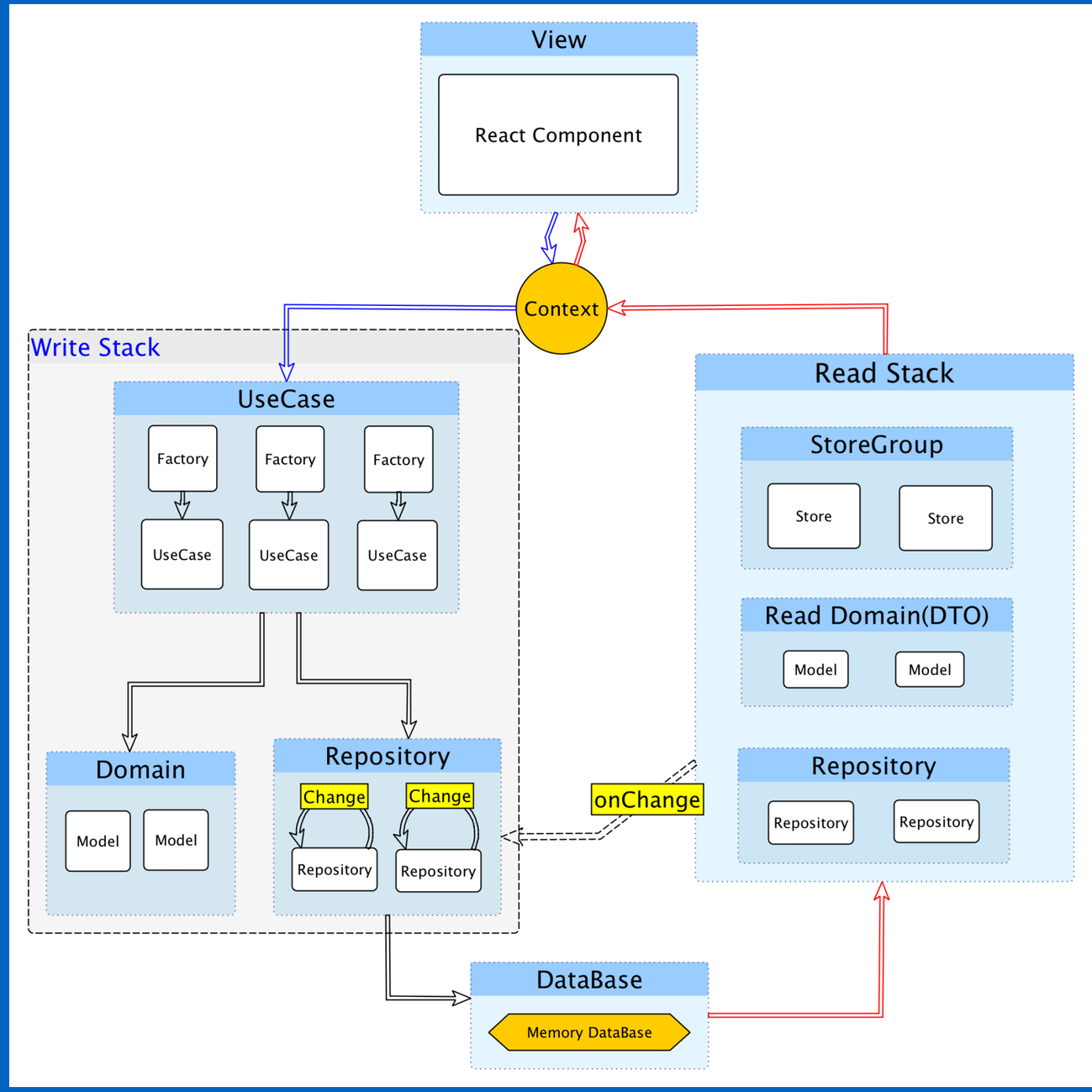
View

View

- Reactを使う 以上
- React ComponentとCSSの設計も色々ある
  - コンポーネントなCSS
  - SUIT CSS
  - コンポーネントの分類と命名規則
  - 長いので省略

Write Stack

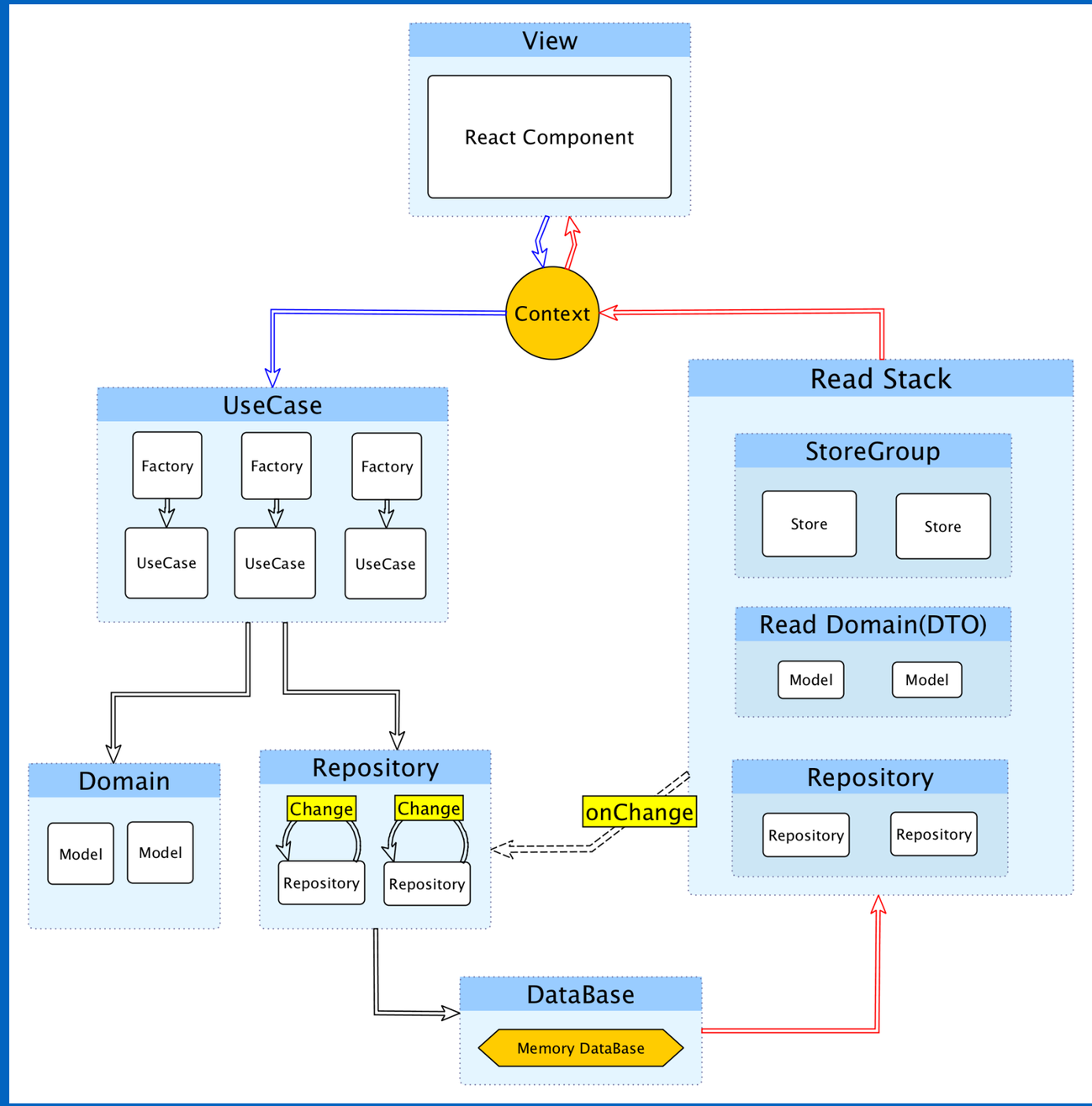
Write Stack


View
React Component
Context
Write Stack
UseCase
Factory
Factory
Factory
UseCase
UseCase
UseCase
Domain
Model
Model
Repository
Change
Change
Repository
Repository
onChange
Read Stack
StoreGroup
Store
Store
Read Domain(DTO)
Model
Model
Repository
Repository
Repository
DataBase
Memory DataBase

UseCase

UseCase

- ViewからUseCaseを発行(ActionCreatorと類似) [図]
- 振るまいの流れを記述する
- トランザクションスクリプトっぽくもある(アプリケーション層)
- UseCaseと対になるFactoryを持ってる
  - Factoryはテストのため
  - FactoryがRepositoryのインスタンスをUseCaseに渡す(依存関係逆転の原則)
- `execute()`にユースケース実装する

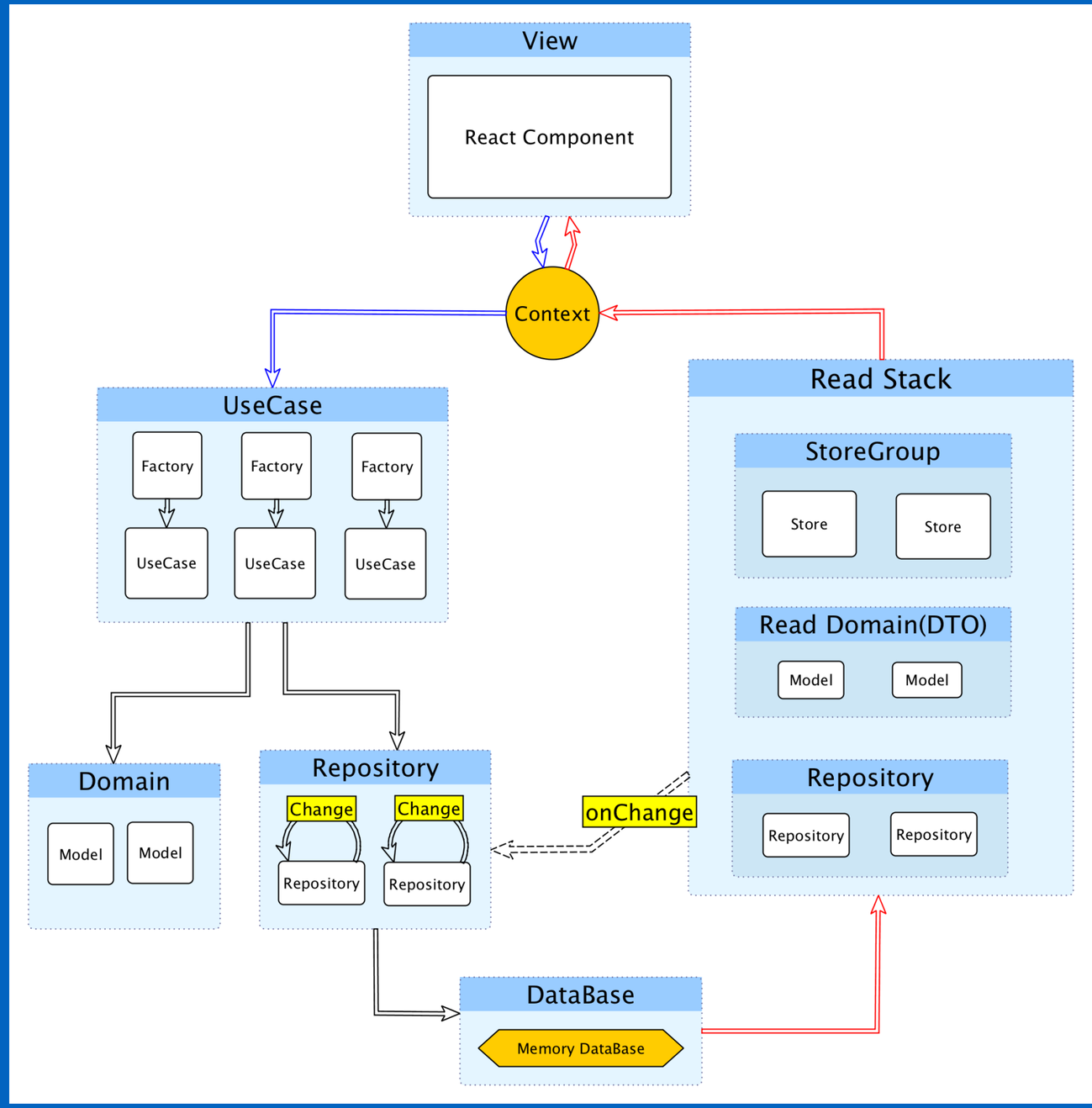


[図] 入れ物っぽい感じがするよね

Domain Model

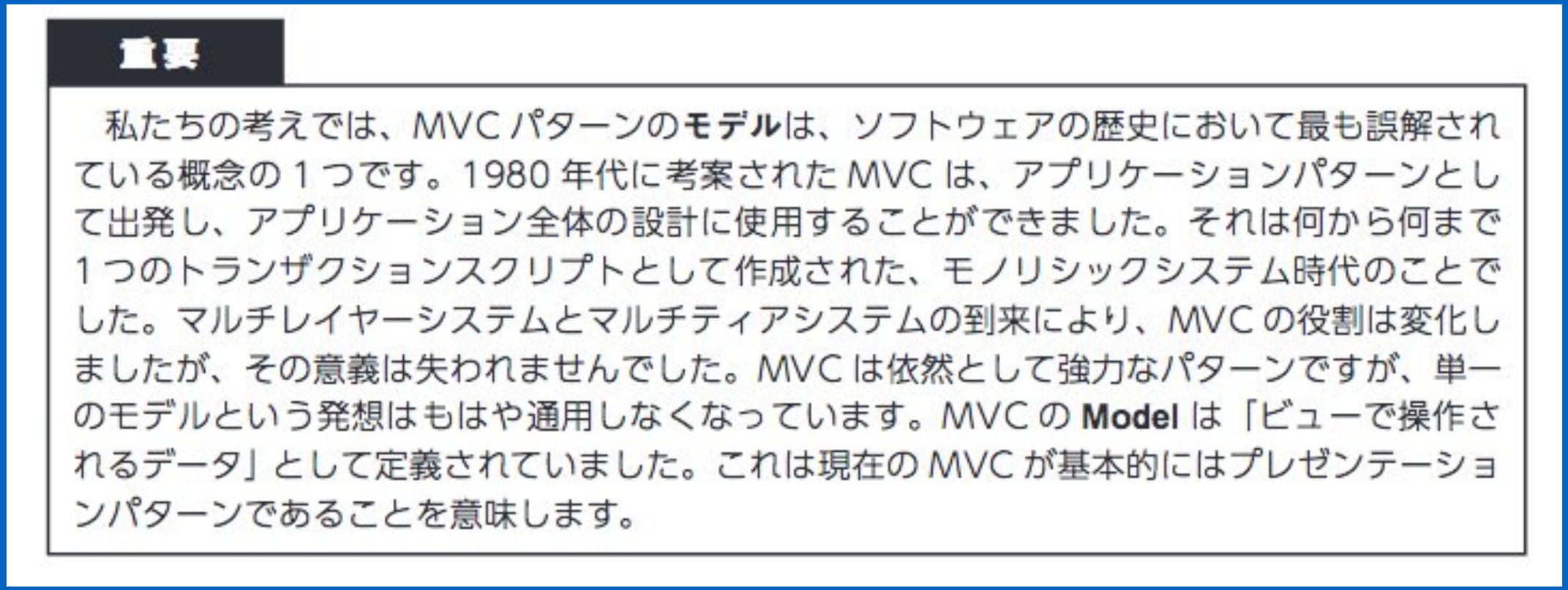
Domain Model

- 作ろうとしてるものを表現するオブジェクト[図]
- モデルクラス
- ここでは、データと**振る舞い**を持ったクラス
- できるだけPOJO(Plain Old JavaScript)である
- いまさらきけない「ドメインモデル」と「トランザクションスクリプト」



[図] 入れ物っぽい感じがするよね

モデルとは...



私たちの考えでは、MVC パターンの**モデル**は、ソフトウェアの歴史において最も誤解されている概念の 1 つです。1980 年代に考案された MVC は、アプリケーションパターンとして出発し、アプリケーション全体の設計に使用することができました。それは何から何まで 1 つのトランザクションスクリプトとして作成された、モノリシックシステム時代のことでした。マルチレイヤーシステムとマルチティアシステムの到来により、MVC の役割は変化しましたが、その意義は失われませんでした。MVC は依然として強力なパターンですが、単一のモデルという発想はもはや通用しなくなっています。MVC の **Model** は「ビューで操作されるデータ」として定義されていました。これは現在の MVC が基本的にはプレゼンテーションパターンであることを意味します。

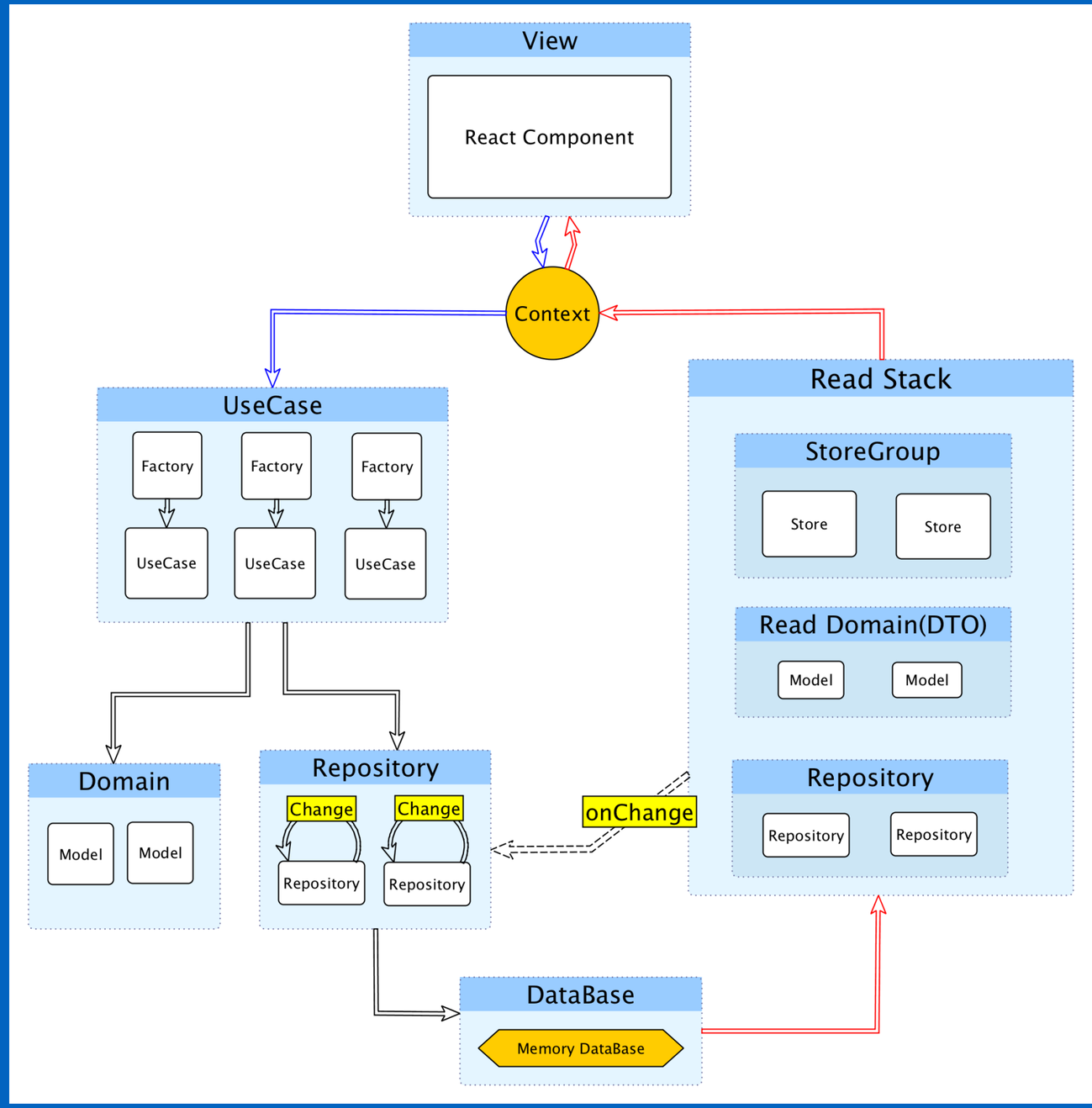
via .NETのエンタープライズアプリケーションアーキテクチャ

言葉を定義するのも設計

Repository

Repository

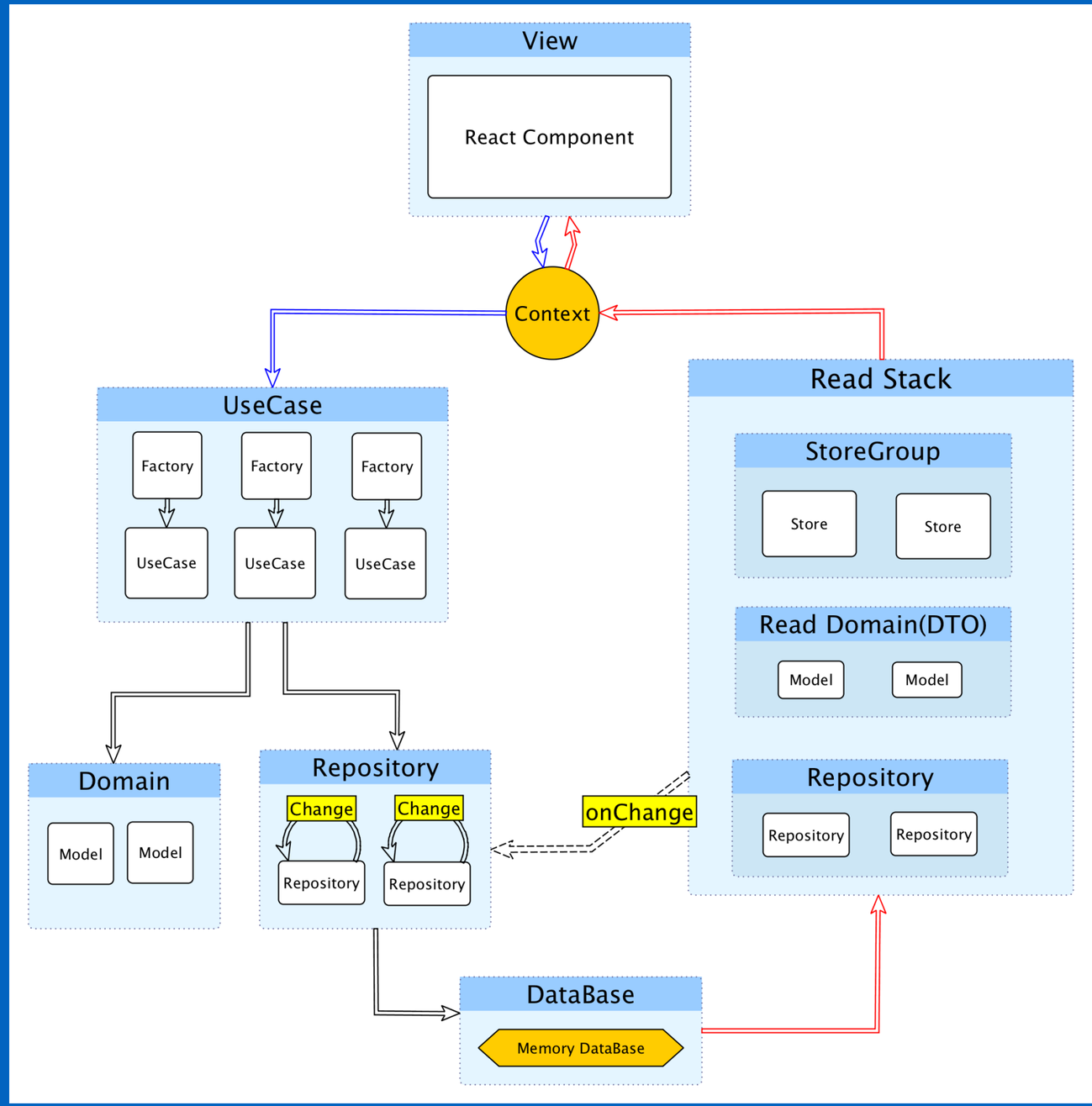
- ドメインモデルのインタンスを永続化するレイヤー [図]
  - Repositoryパターン
- シングルトン！！！
- `find(id)/save(model)/delete(model)` 表からはコレクションっぽい
  - JavaScriptの場合はメモリ(ただのMap)として保持する



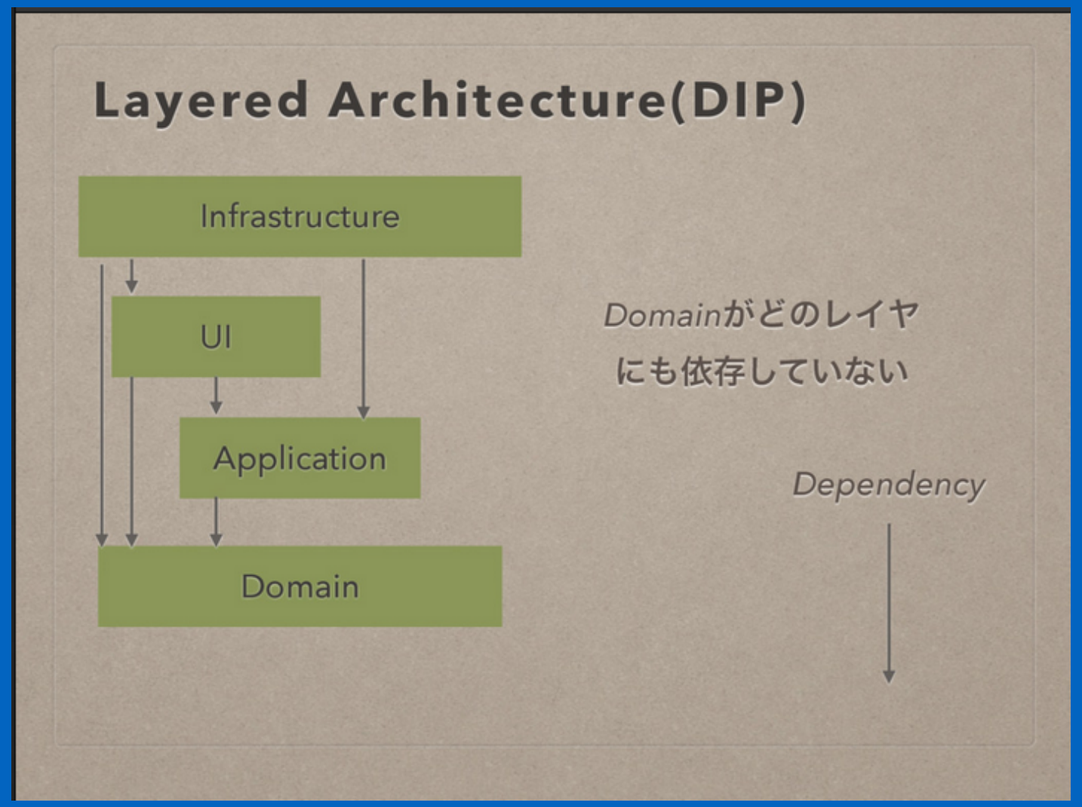
[図] 入れ物っぽい感じがするよね

Repositoryの永続化問題

- クライアントサイドJavaScriptでは永続化が難しい
- どこでインスタンス化するの?問題
- そのためシングルトンなどを使う
- 依存関係逆転の原則
- UseCaseのコンストラクタに引数(依存)としてrepositoryを渡す
  - Factoryはそのための存在



依存関係逆転の原則(DIP)



Scalaで学ぶヘキサゴナルアーキテクチャ実践入門 // Speaker Deck

設計の進め方

- 理想のAPIを擬似コードで書くのはあくまで参考
- クライアントサイドでは永続化の持ち方の問題が付きまとう
  - サーバならどっかにプロセス立てて、プロセス同士で通信みたいなことができる
- 実際にデータの流れと状態の持ち方を**コードとして**書いてみて、設計することが重要

処理の流れではなく、
データの流れを定義

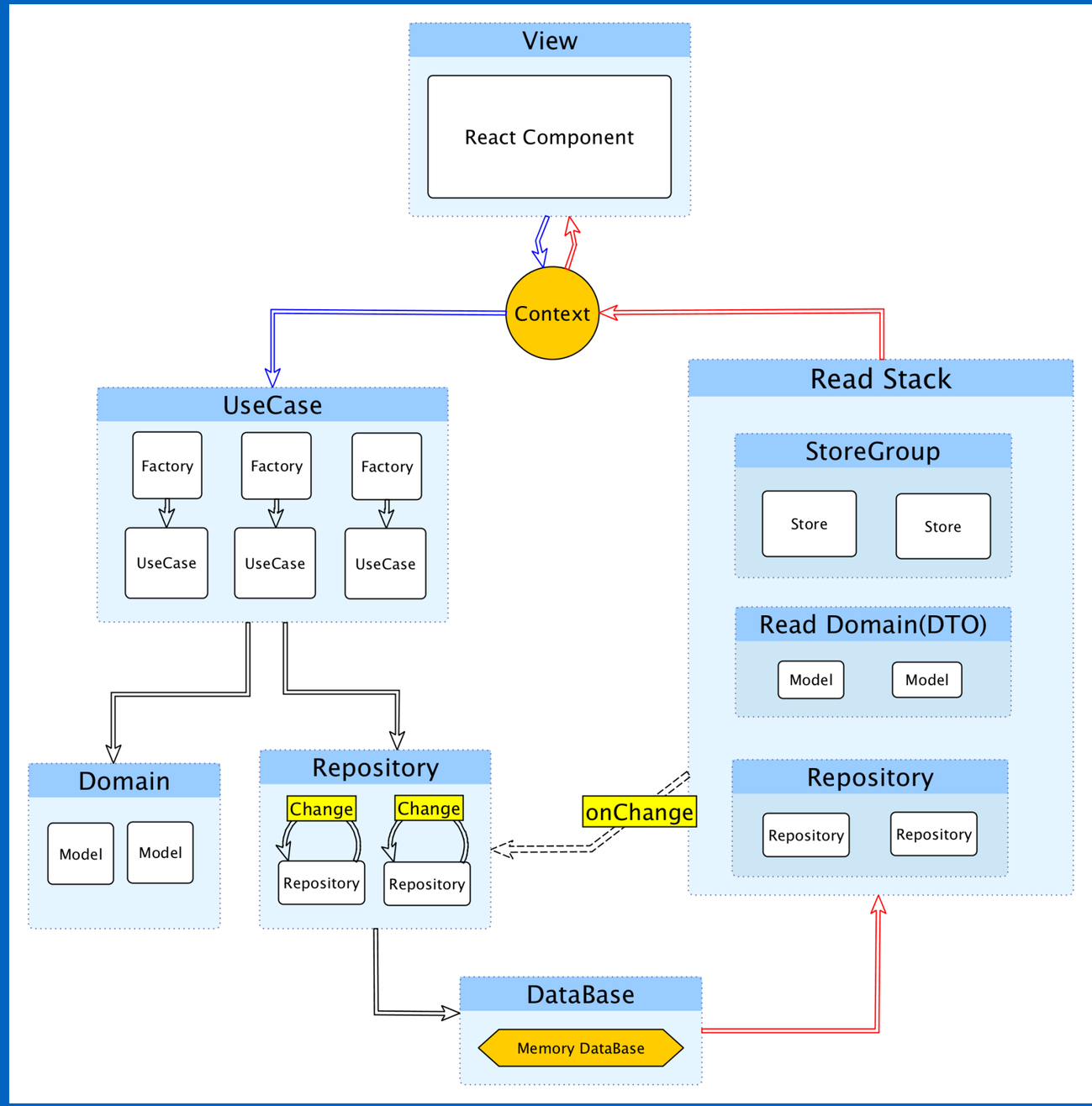
— *http://www.slideshare.net/MasashiSakurai/javascript-js-53219222*

DataBase

DataBase

- ただのMapオブジェクト [図]
- Repositoryをできるだけシンプルに保つため、データベースもシンプルに
  - `key : value` だと簡単で良い
- `localStorage`とかに入れても良い
- 変更されたら変更したことを通知する(実際はrepositoryが投げてる)
  - `emit("Changed")`

[図] 入れ物っぽい感じがするよね



UseCaseで処理の流れを記述

```
export class UpdatePageNoteUseCase {
    constructor({documentRepository}) {
        this.documentRepository = documentRepository;
    }
    execute({note, pageNumber}) {
        const document = this.documentRepository.lastUsed();
        document.updateNodeAtPage(note, pageNumber);
        this.documentRepository.save(document);// => onChange
    }
}
```

Factory

```
import documentRepository from "../infra/DocumentRepository";
export class UpdatePageNoteFactory {
    static create() {
        return new UpdatePageNoteUseCase({
            documentRepository
        });
    }
}
```

Async UseCase

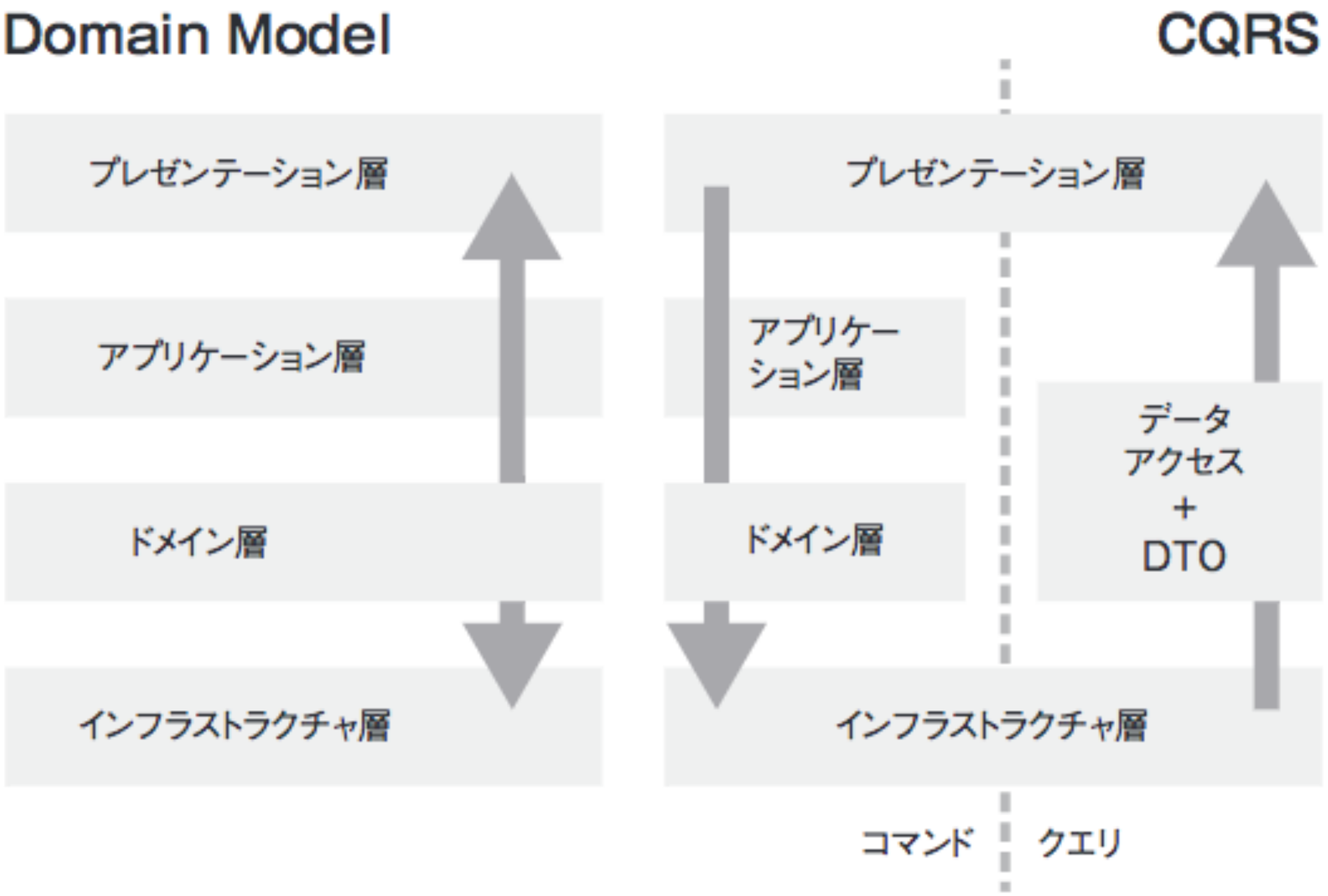
```
export class AddTodoItemUseCase{
    constructor({todoListRepository, todoBackendServer}) {
        this.todoListRepository = todoListRepository;
        this.todoBackendServer = todoBackendServer;
    }
    execute({title}) {
        const todoList = this.todoListRepository.lastUsed();
        const todoItem = todoList.addItem({title});
        return this.todoBackendServer.add(todoItem).then(() => {
            this.todoListRepository.save(todoList);
        });
    }
}
```

Transaction UseCase(再利用性)

```
export class TransactionTodoUseCase {
    //...
    async execute({title}) {
        const options = {
            todoListRepository: this.todoListRepository,
            todoBackendServer: this.todoBackendServer
        };
        const addTodo = new AddTodoItemUseCase(options);
        const updateTodoItem = new UpdateTodoItemTitleUseCase(options);
        const removeTodoItem = new RemoveTodoItemUseCase(options);
        // Add => Update => Remove
        await addTodo.execute({title});
        await updateTodoItem.execute({itemId: todoItem.id, title: "UPDATING TITLE"});
        await removeTodoItem.execute({itemId: todoItem.id});
    }
}
```

Read Stack

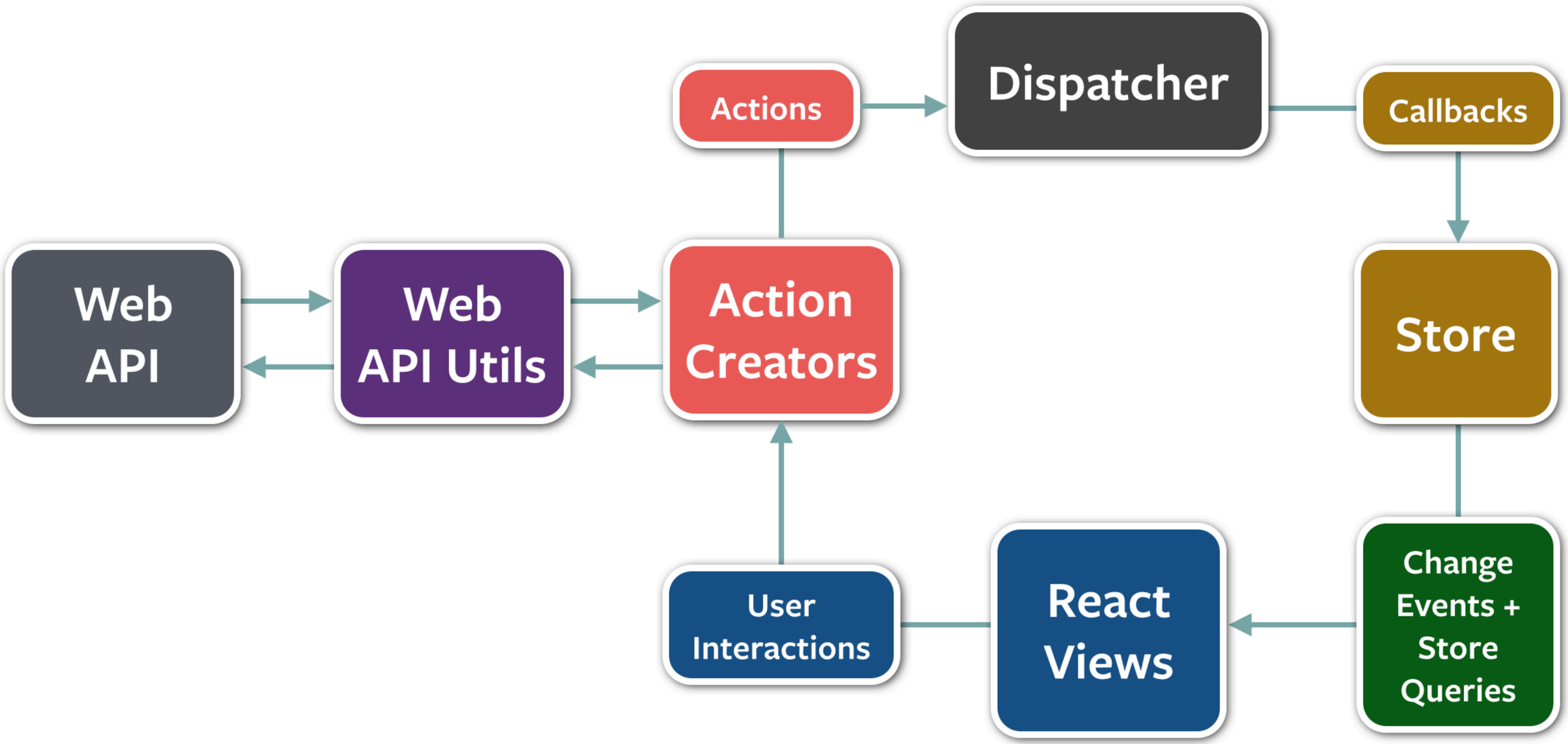
Read Stack?

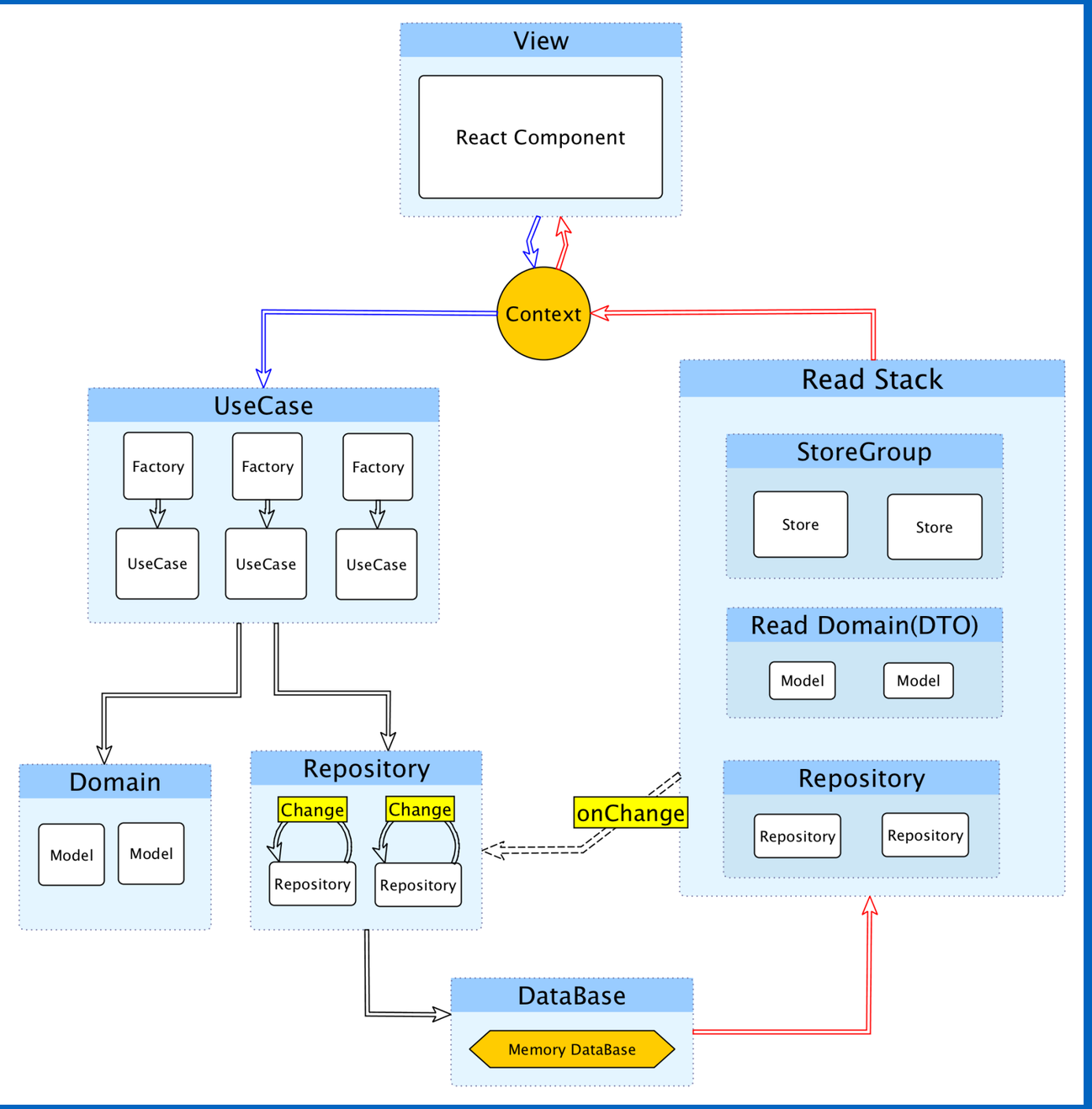


▲図 10-1：Domain Model とCQRS の視覚的な比較

Write(Command)とRead(Query)

- CQRS (Command Query Responsibility Segregation)
- ざっくり: WriteとReadを層として分けて責務を分離する
- 一方通行のデータフロー
- FluxとかReduxでやっていることと同じ

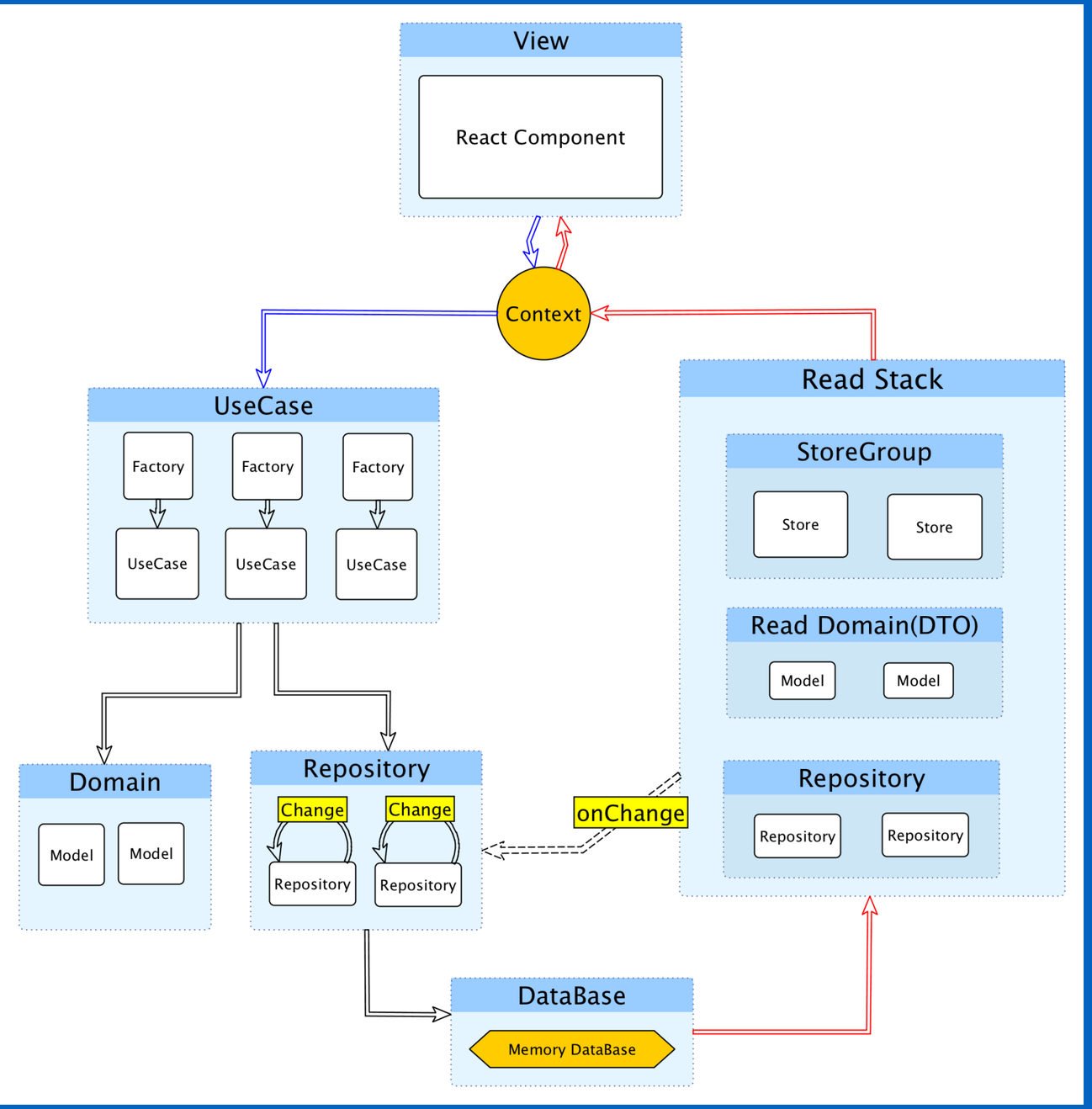

Actions
Dispatcher
Callbacks
Web API
Web API Utils
Action Creators
Store
User Interactions
React Views
Change Events + Store Queries



Read Stack

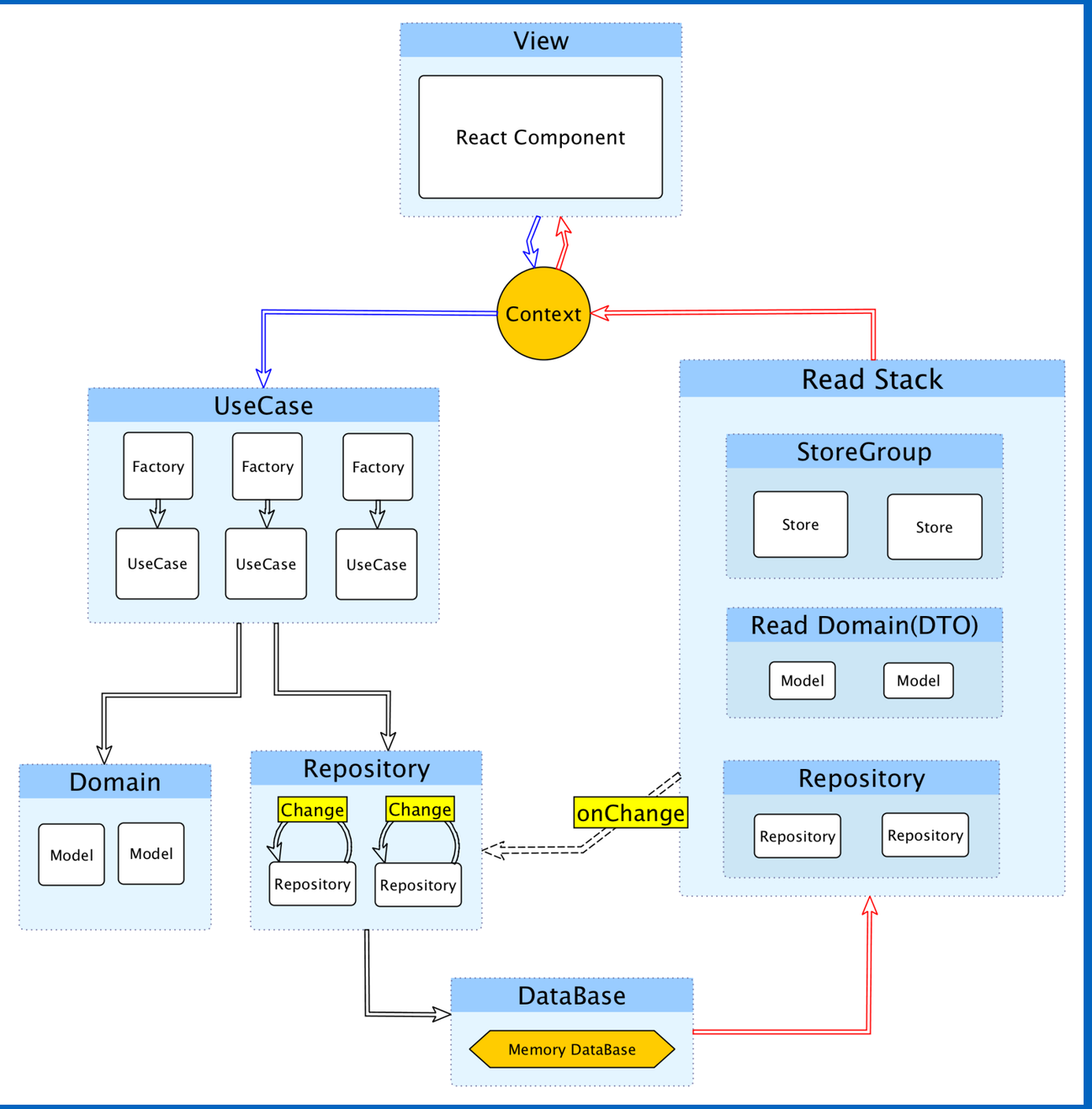
- Readはデータベースが読み込んでView用のデータを作って渡すだけ[図]
- 読み取り専用(変更はしない)ので色々簡略化できる
- 縦に別れたので、テスト依存関係が簡略化できる！

[図] 入れ物っぽい感じがするよね



Read Stack

- Repository
  - Write Stackと同じものを参照してもいいかも?
- Read Model
  - Writeのドメインから振るまいを消したモデルを作ってもよい
  - ドメインモデル貧血症にわざとする = Viewのためのモデル
- Store
  - FluxのStoreと同じだけど、Read Stackでは一番重要



Store

- Stateを持つオブジェクト[図]
- StateはUIに渡してUIが更新される
- Stateが更新された事をUIに伝える(Context経由)

[図] 入れ物っぽい感じがするよね

クライアントサイドで多発する問題

クライアントサイドで多発する問題

- 現在のアーキテクチャでは、永続化したデータしか使えない
- クライアントサイドではStateを更新したら、UIにすぐ反映されて欲しいことがある
  - 「ほんのいっとき」が許されないケースはクライアントサイドにはある
  - コンポーネントに閉じ込めるというのあり
- そのため縦(Read/Writeの層)じゃなくて、横のルールも必要

クエリデータベースとコマンドデータベースが同期していない状態では、プレゼンテーション層が古いデータを表示するかもしれませんし、システム全体の整合性が完全に確保されなくなります。データベースの整合性は何らかのタイミングで確保されますが、常に保証されるわけではありません —— これを**結果整合性**と呼びます。

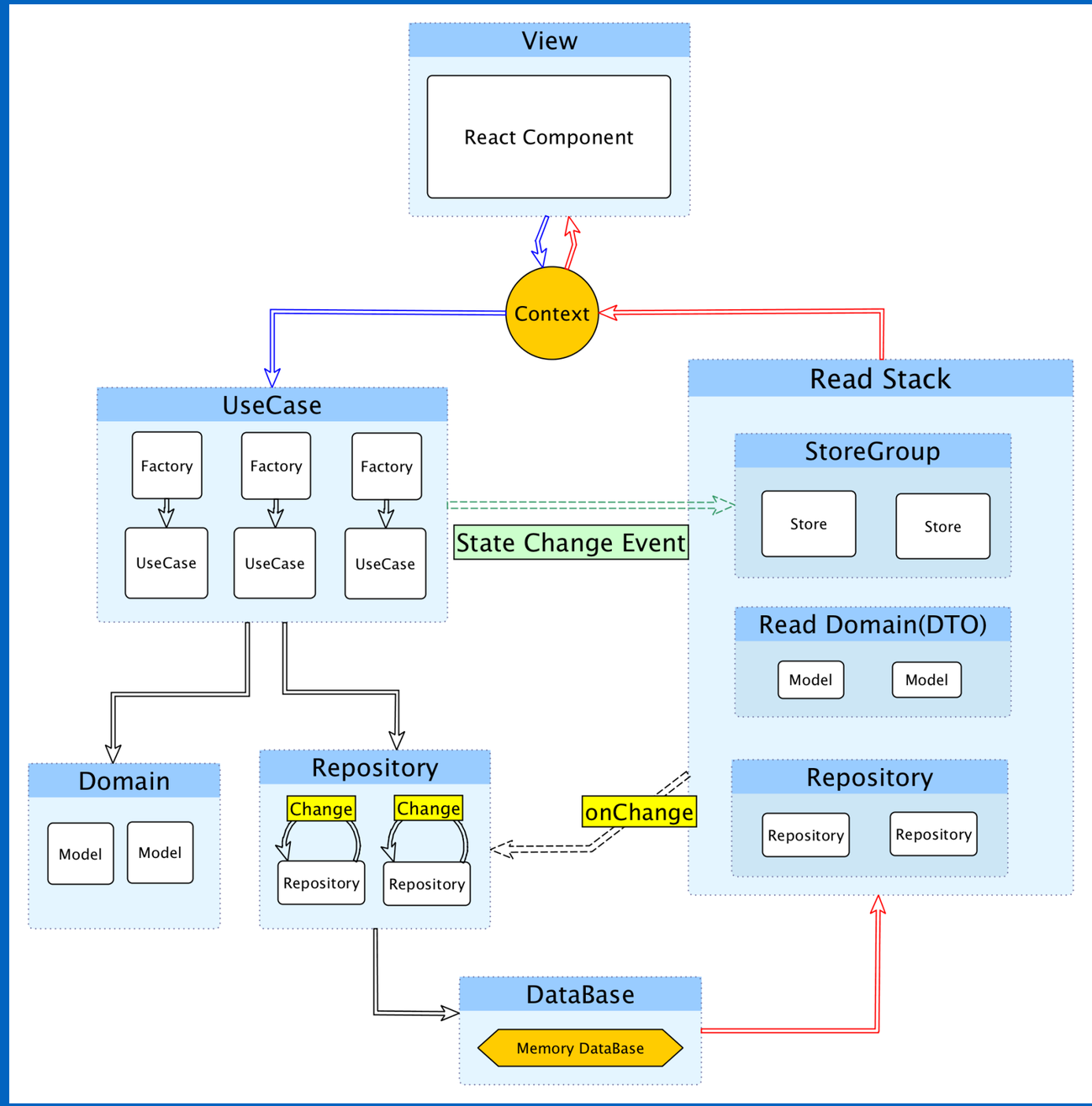
古くなったデータを扱うことが問題かどうかは、状況次第です。

何よりもまず、古いデータを使用するからには理由があるはずです。多くの場合は、スケーラビリティを向上させるために書き込みアクションを高速化することが理由となります。スケーラビリティが重要でなければ、古いデータと結果整合性に甘んじる理由はないでしょう。ましてや、ほんのいっとき古いデータを表示することが許されないアプリケーションなどほとんど存在しないはずです。もっとも、「ほんのいっとき」がどれくらいかは状況によります。

via .NETのエンタープライズアプリケーションアーキテクチャ　第2版 p299

UseCase -> Store

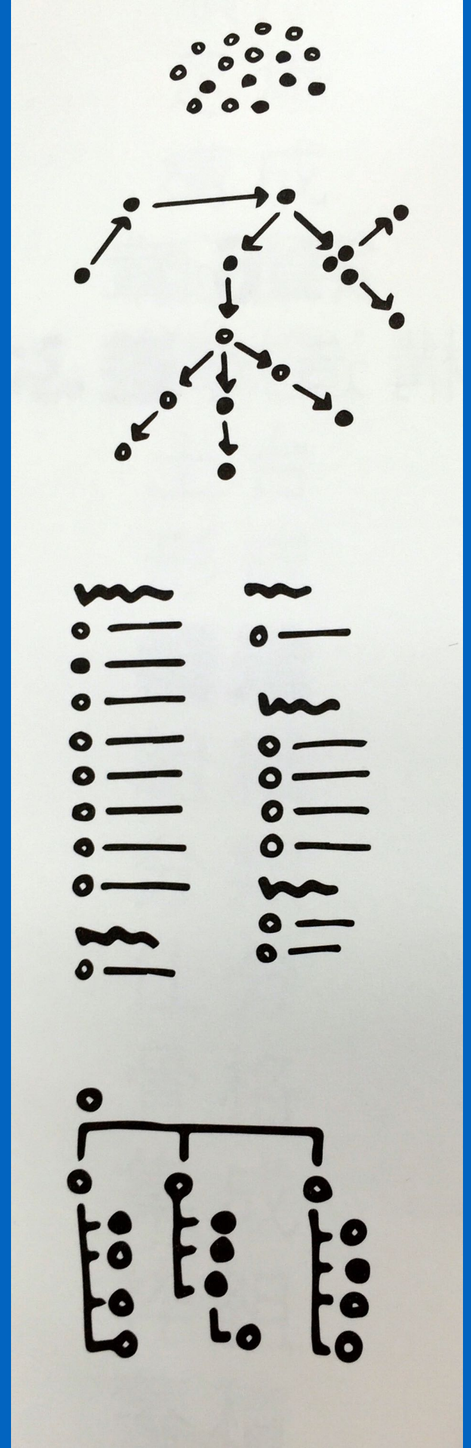
- UseCaseからdispatchしたイベントが、Storeに届く横のルート
  - 抜け穴感があるので慎重に取り扱いたい
- FluxやReduxはこのルートが基本的な流れ
  - 図の上半分がよく見る流れ



構造化の考え方

ものごとを構造化するための方法はたくさんある

今日からはじめる情報設計, p131



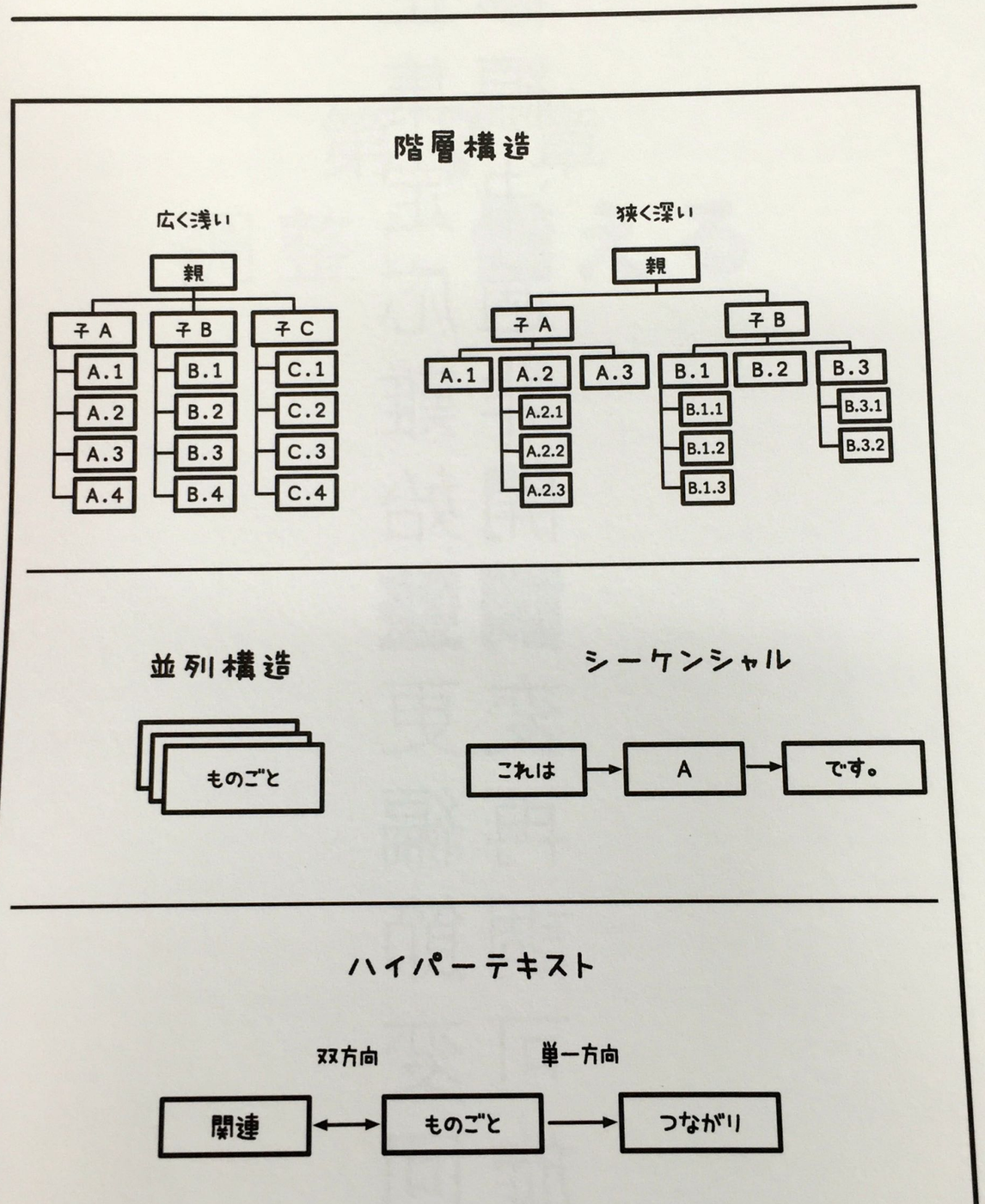
分類法(タクソノミー)

- 分類法(タクソノミー) は構造化の手法
- 分類法を組み合わせて形状を作る
  - UIに反映する形となったもの。 e.g.) ページ、Repositoryとか
- 曖昧な分類と正確な分類はそれぞれメリット、デメリットがある
  - 曖昧さは明確性を犠牲にし、正確性は柔軟性を犠牲にする
- ファセット = 主キーで分類する

分類の結果

- 分類法は
  - 並列的構造
  - 階層的構造
- どちらかになる p143

p143 今日からはじめる情報設計



実装したもの

- azu/presentation-annotator: viewing presentation and annotate.
  - この話は New framework by azu · Pull Request #7 · azu/presentation-annotator で加えた変更を元にしているので、masterは書かれている事と違うコードになっている可能性があります。

参考書籍

- 今日からはじめる情報設計
- オブジェクト開発の神髄
- .NETのエンタープライズアプリケーションアーキテクチャ　第2版

参考

- CQRS + ES
  - CQRS+ESをAkka Persistenceを使って実装してみる。
  - 最新DDDアーキテクチャとAkkaでの実装ヒントについて // Speaker Deck
- DDD クリーンアーキテクチャ
  - DDD + Clean Architecture + UCDOM Essence版 // Speaker Deck
  - Scalaで学ぶヘキサゴナルアーキテクチャ実践入門 // Speaker Deck
- レイヤー設計とか、オブジェクト指向とか、DDDとか、その辺 - まっつんの日記

参考

- [Android] – これからの「設計」の話をしよう – NET BIZ DIV. TECH BLOG
- CQRSの小さな演習(1) 現実の問題 - 考える場所

参考 MVVM

- MVVMパターンとは？
- 塹壕よりLivetとMVVM
- MVVMのModelにまつわる誤解 - the sea of fertility
- MVVMパターンの常識 ― 「M」「V」「VM」の役割とは？ ― @IT
- 開発者が知っておくべき、6つのUIアーキテクチャ・パターン ― @IT