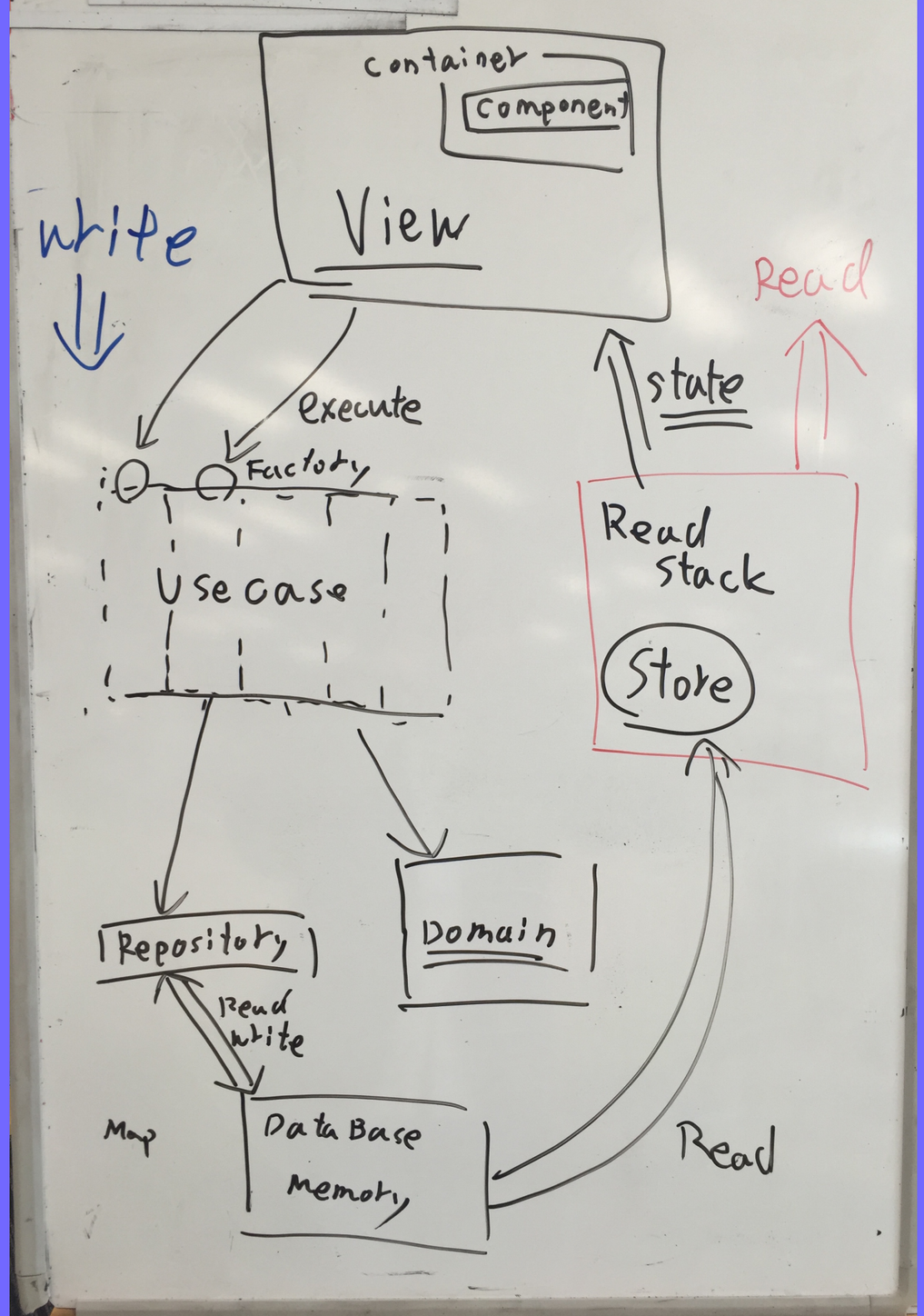
JavaScriptのアーキテクチャ

概要

- DDD/CQRSベースなレイヤードアーキテクチャ
  - Write(コマンド)/Read(クエリ)
  - イベントソーシングはやってない
- DTO(データ変換オブジェクト)はまだ入れてない
  - なのでDomainのインスタンスがViewにそのまま渡す(Read Only)
- テストが可能


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

実装

- azu/presentation-annotator: viewing presentation and annotate.

登場人

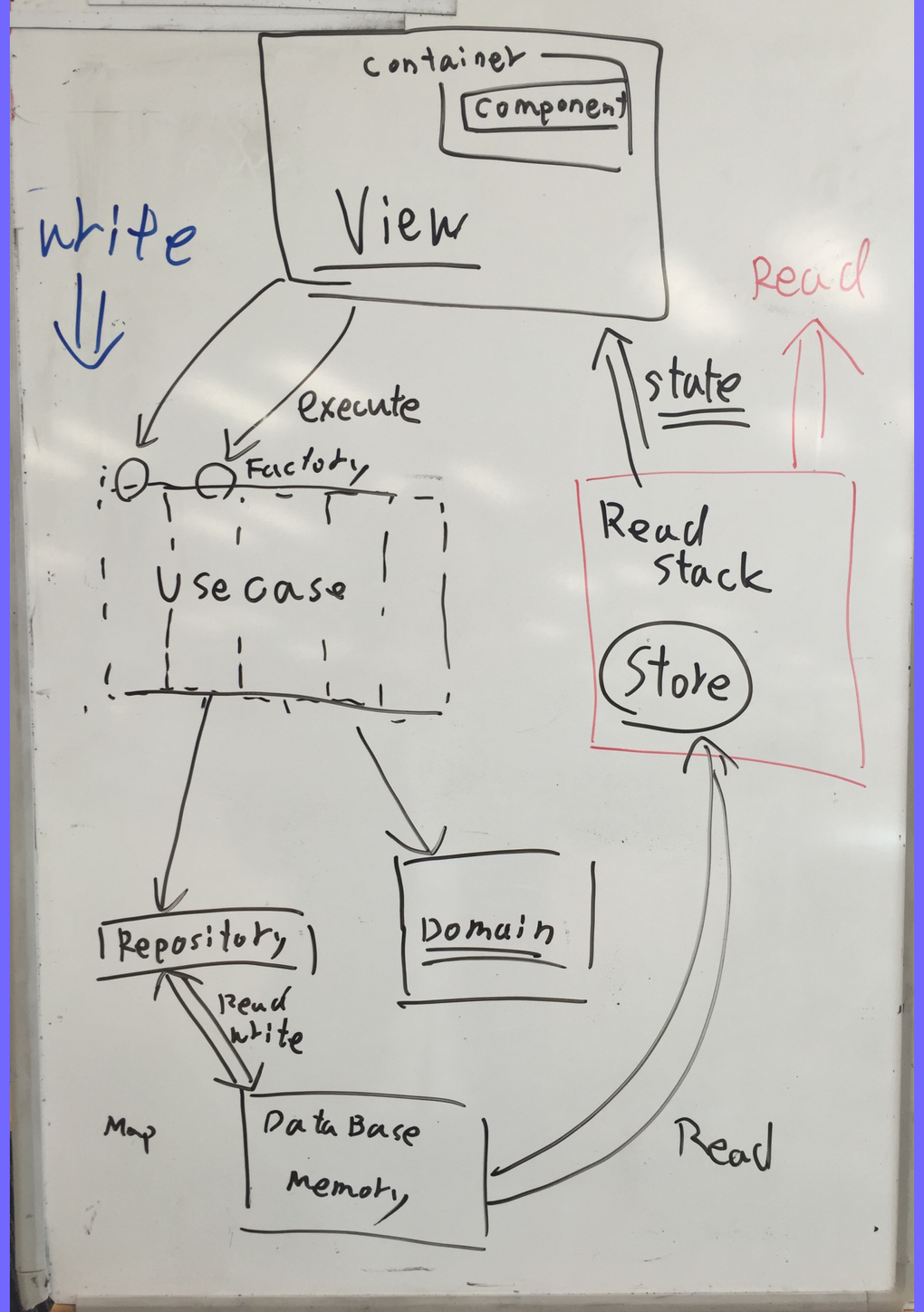
- View(React Component)
- Write Stack
  - UseCase
  - Repository
  - Domain
- Read Stack
  - Store
- DataBase(Memory)

View

- React Component
- CSS Architecture(別途)
  - Container + Project Componentベース

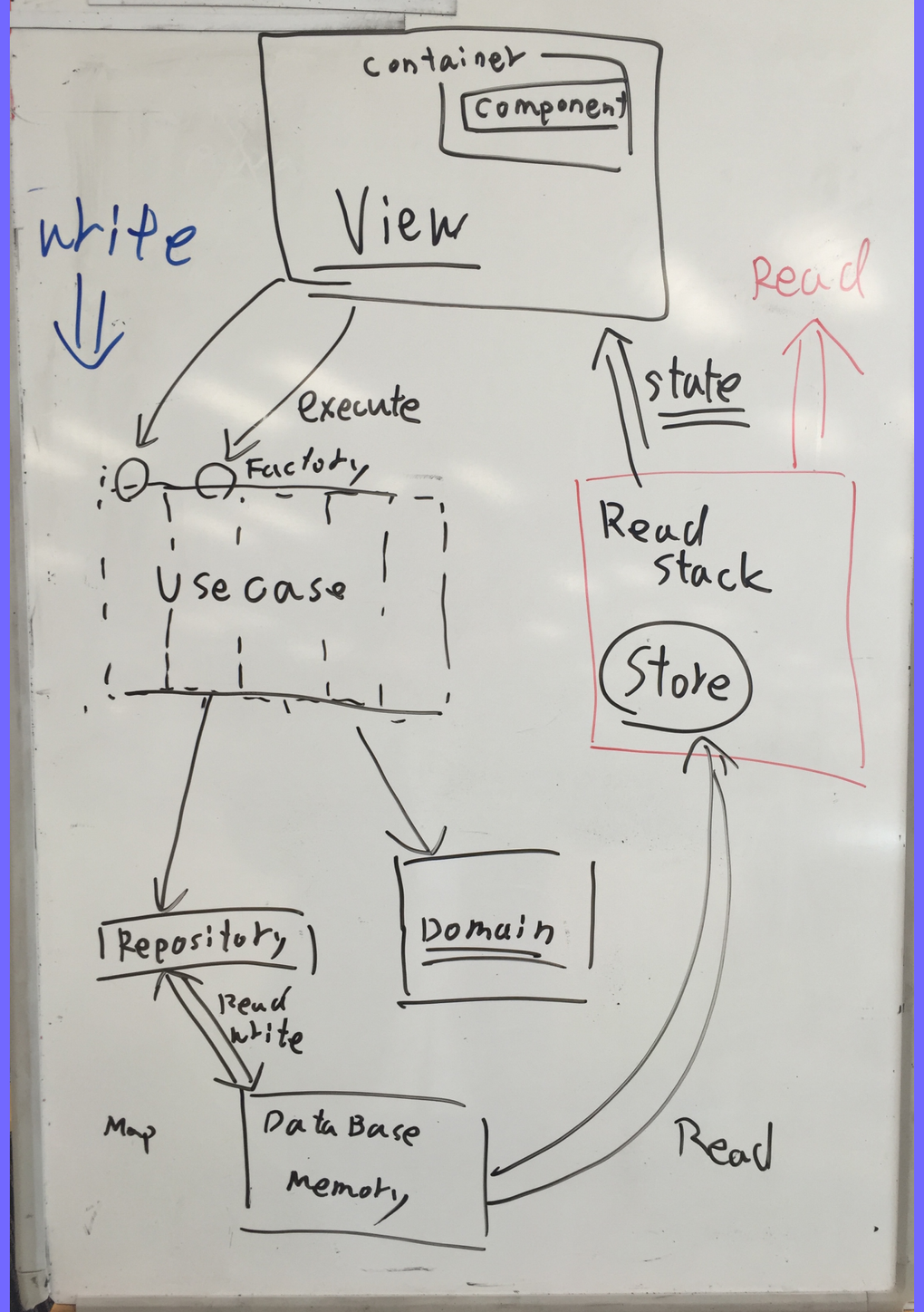
Write Stack ←

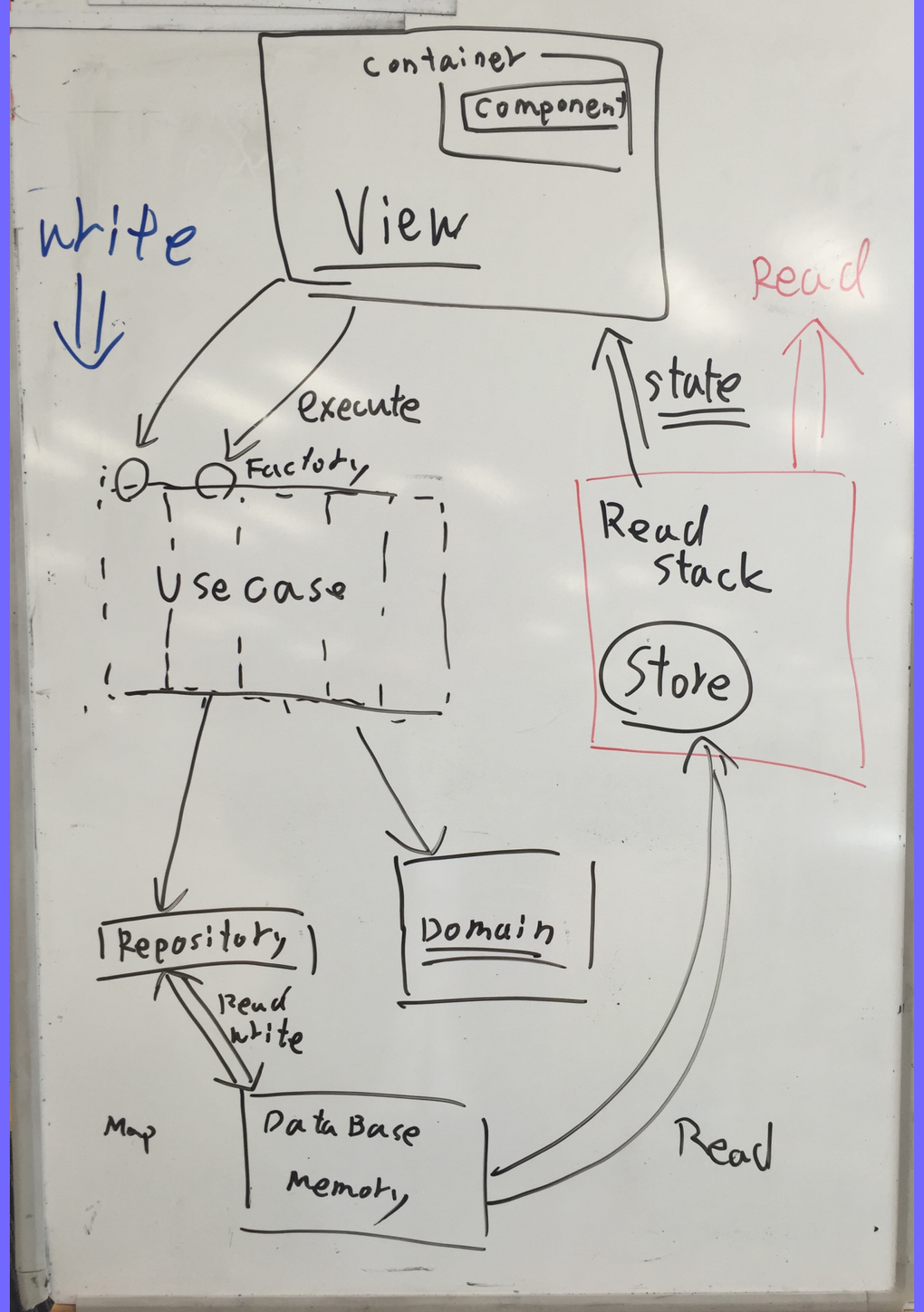
- ViewからUseCaseを発行(Actionと類似)
- UseCaseと1対1のFactoryを持ってる
  - テストのためというのがメインだが薄いレイヤーがViewとUseCaseの間に欲しかった
  - FactoryがRepositoryのインスタンスをUseCaseに渡す(依存関係逆転の原則)



Write Stack ←

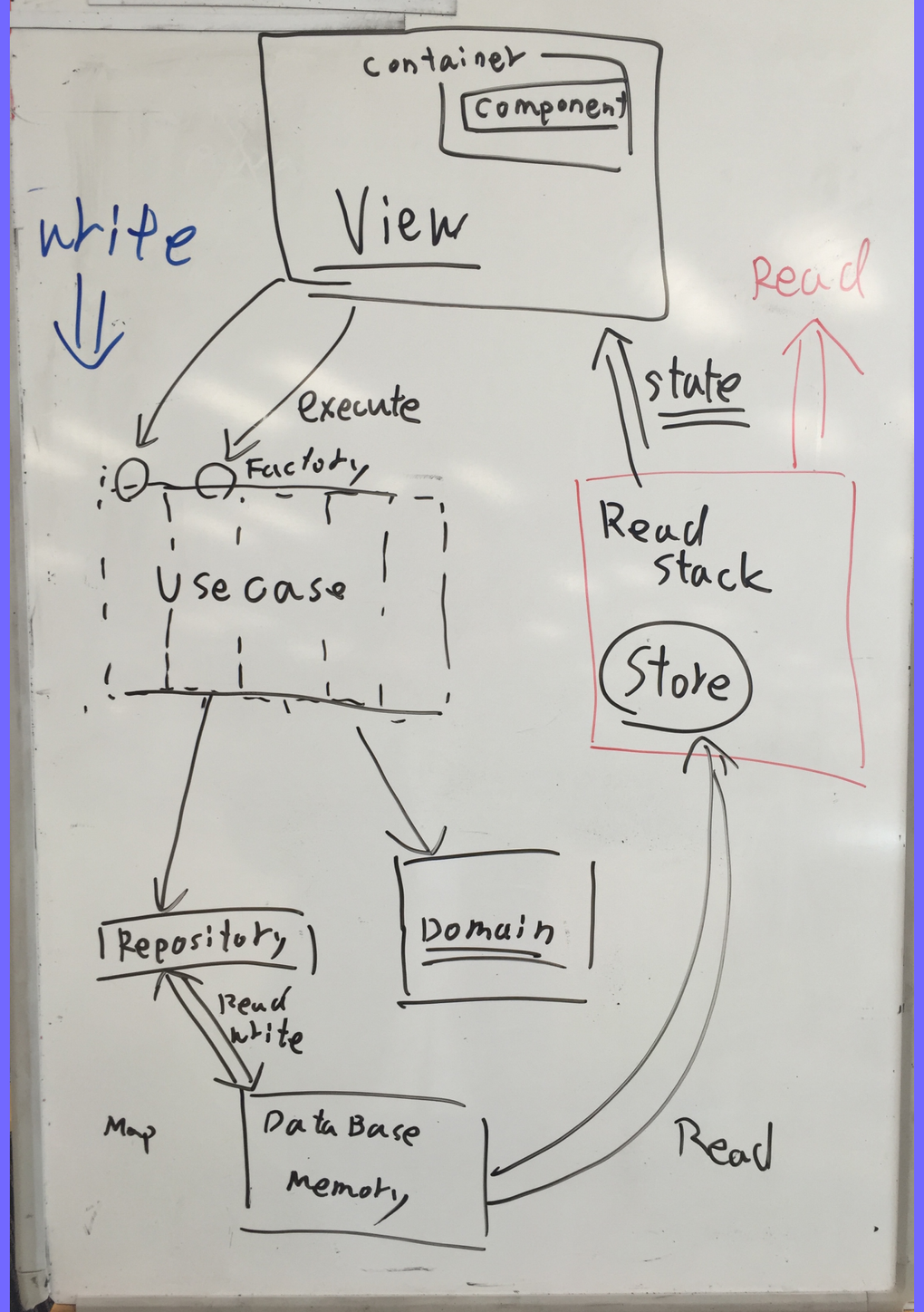
- UseCase
  - Repositoryから永続化されたDomainを取り出す
  - Domainモデルを使って操作する
  - Domainモデルを作ってRepositoryに保存する
- Domain
  - Domainモデル(EntityとValueObject)
  - ビジネスロジックのコア部分
    (UseCaseはこれを使う役)





Read Stack ➡

- Read Modelからなる層
  - 読み込み + 変換がメイン
  - 読み込みのみなので、インフラでありドメインでありアプリである
- Database
  - Write Stackで保存されているデータがある
  - Read Stackから取り出す

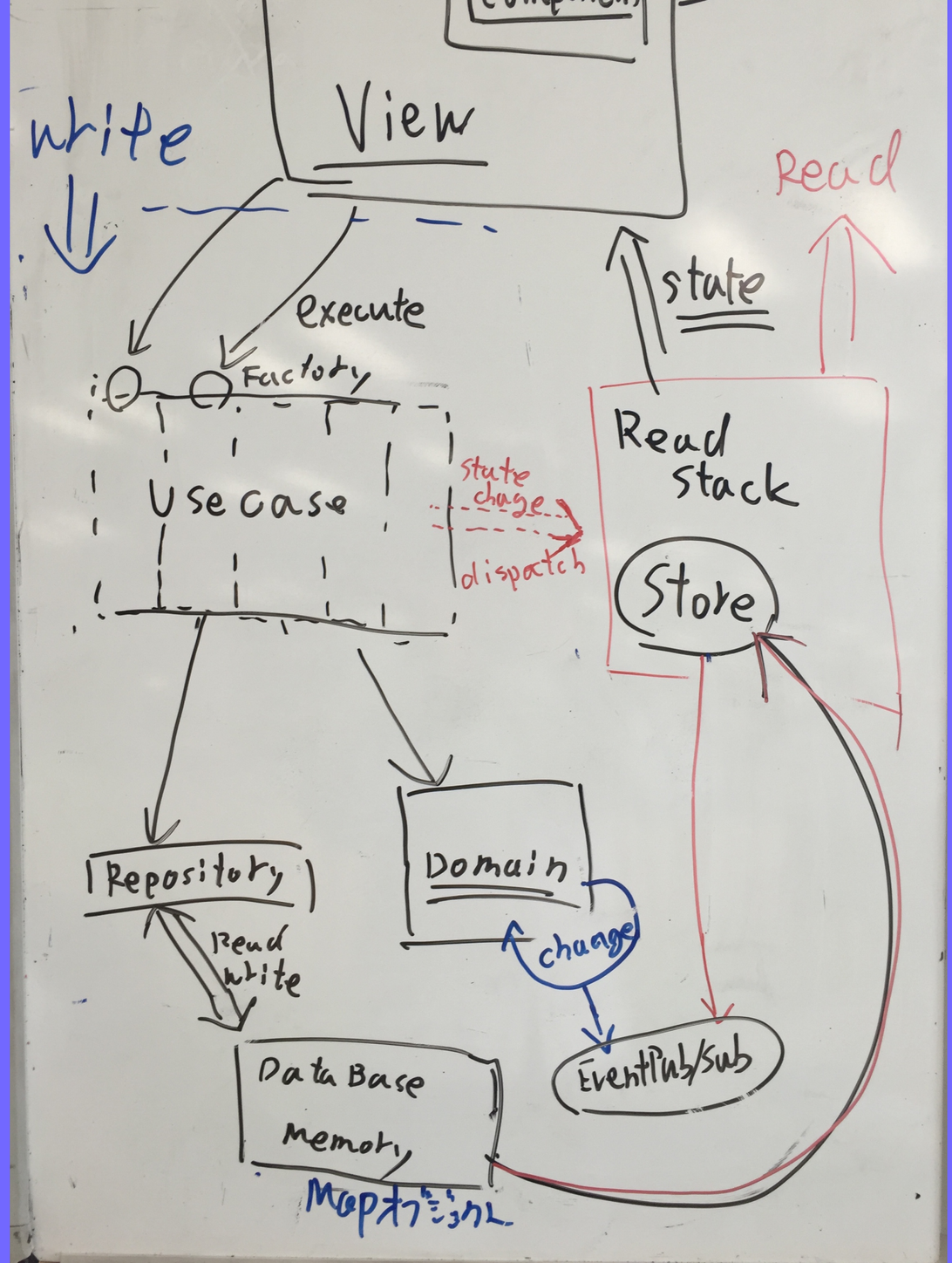


Read Stack ➡

- DataBaseから読み込んだデータをUIに向けて変換する
  - State, ViewModelに近い概念
- UIは必要になったらRead Stackからデータを取得する
  - 取得したデータを使ってUIを構築する

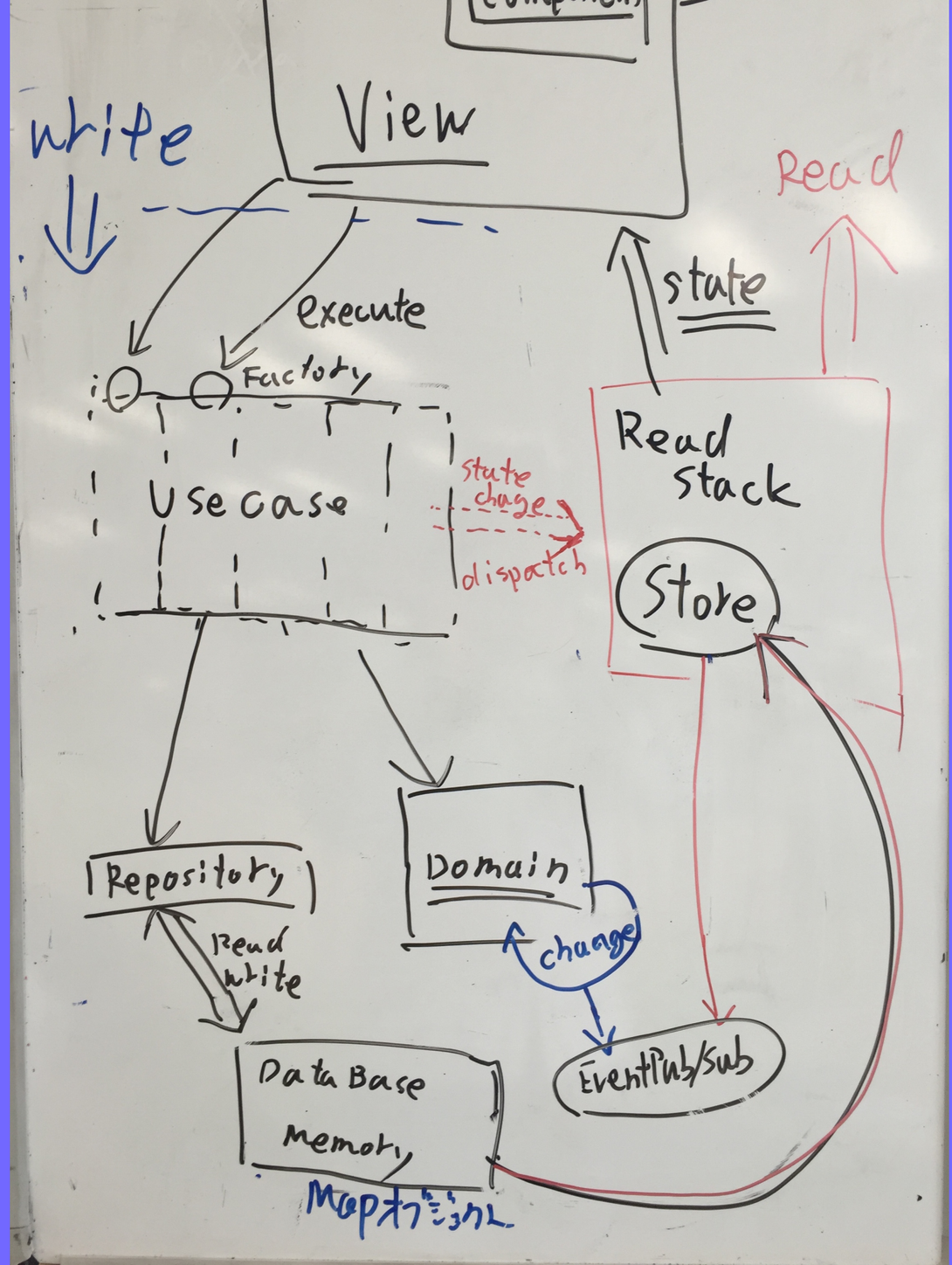
READ AND WRITE ↔

- WriteとReadの協調はどうやるの?
- WriteはDBに書き込み、ReadはDBから読み込む
- DBが変更したことは誰がどう伝えるのか
- Event Aggregator 😣
  - シングルトンなEventEmitter
  - 差し替え可能なのでテスト可能に



協調動作 ↔

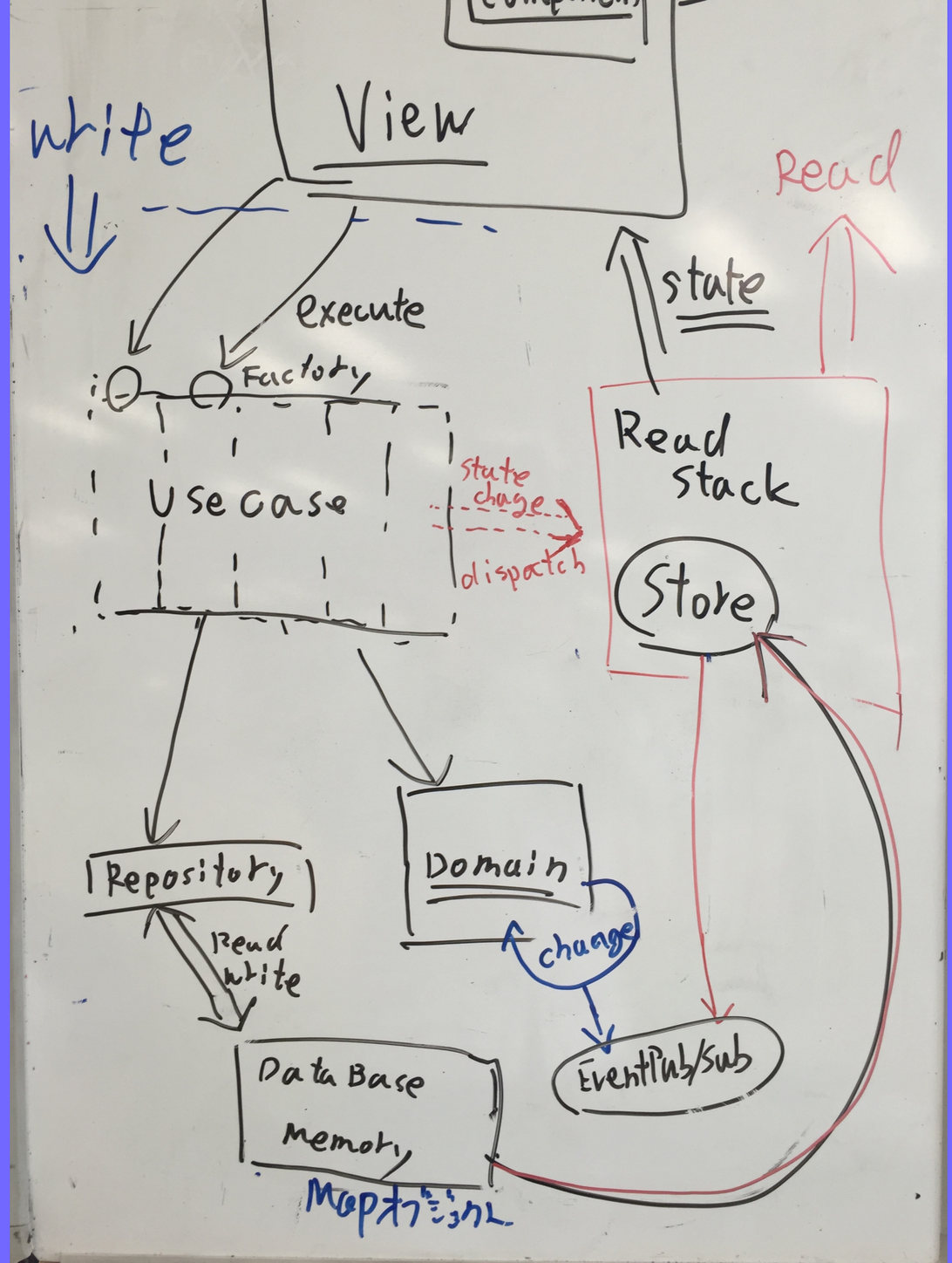
- EventPub/Sub(Event Aggregate)
  - シングルトンなDomain EventのPub/Sub
  - 実態は中にEventEmitterがいるだけ
  - 差し替え可能にしてテストできる
- ドメインモデルの変更を監視 <->> 監視してる人へ通知



```
import DomainEventEmitter from "./DomainEventEmitter";
export class DomainEventAggregator {
    constructor() {
        // テストはこれを置き換えちゃう
        this.eventEmitter = new DomainEventEmitter();
    }
    subscribe(EntityName, handler) {
        this.eventEmitter.subscribe(({type, value}) => {
            if (type === EntityName) {
                handler(value);
            }
        });
    }
    publish(EntityName, value) {
        this.eventEmitter.publish({
            type: EntityName,
            value
        });
    }
}
// シングルトン
export default new DomainEventAggregator();
```

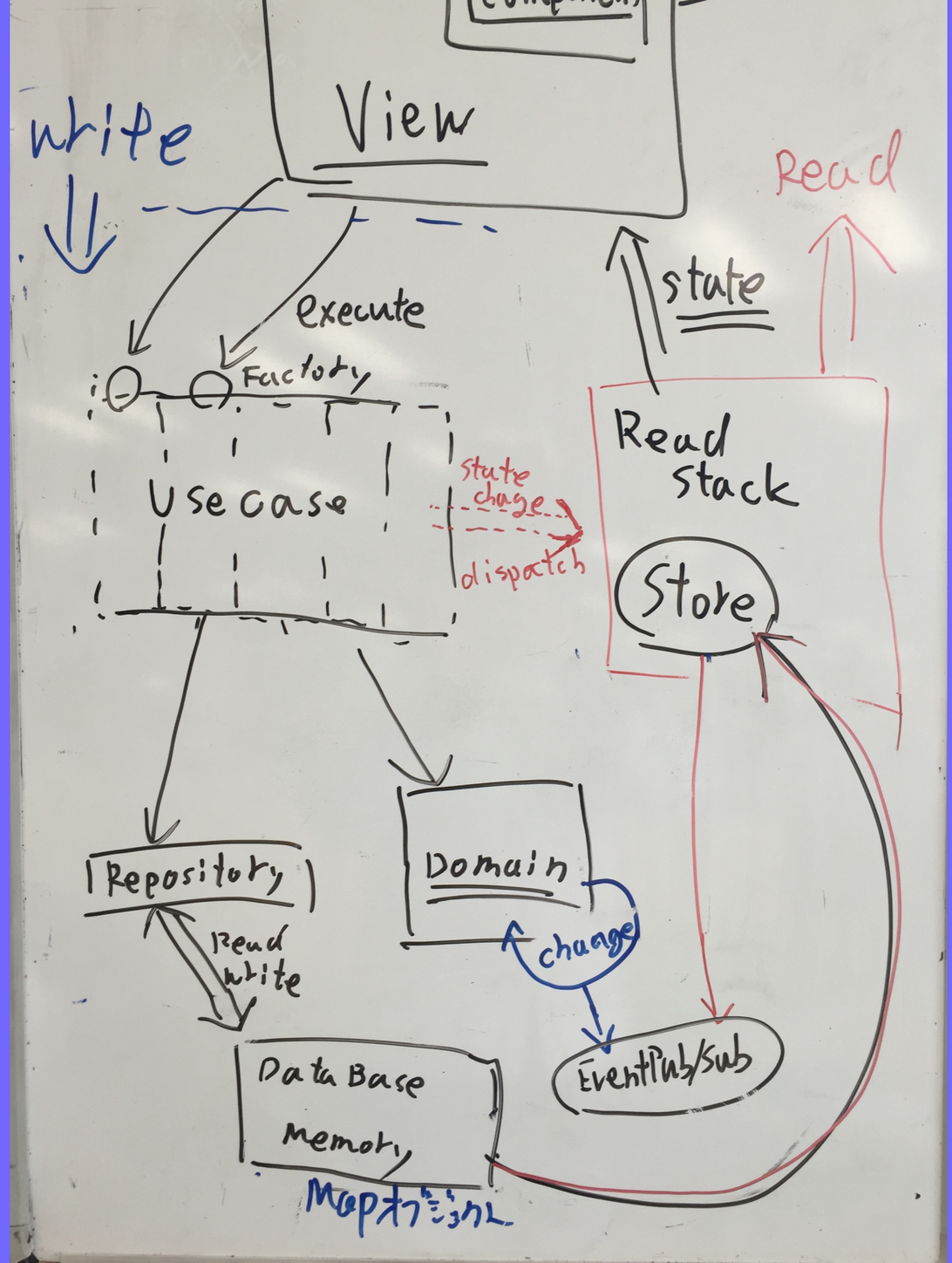
Read Stack

- Store
  - DataBaseから読み込んだDomain + State
  - Viewに向けた形に変換しても良い
- Read(Store)Aggregate
  - Storeをまとめて、一つのでっかいStateとして返すもの
  - Factory の対っぽい役割も持ってる
  - Single source of truth inspiered by Flux



Shortcut UseCase -> Read Store

- UseCaseからStoreへ値をdispatchする仕組みも備えてる
- Stateだけの変更をしたい場合にDomainを通さないで更新することができる



課題点

- Factory:UseCase=1:1 はテストのためのコストがある
  - ギリギリ許容範囲かなと思ってるがもう少し意味的に適切な方法があるかも
- ネーミングイマイチ Store Aggregator
- DomainEventAggregator どうなん?
- Shortcut UseCase -> Read Store 使い分けがわかりにくそう

課題点 - 使う方

- Repositoryが人によって解釈違う
  - 自分はDBへのget/add/findな薄いやつ(非同期で通信とはちょっとイメージ違う)
  - Repositoryのライフサイクルは誰が監視する?
- シーケンシャルな非同期はUseCaseでどうやるのか?
  - A -> B -> Cというユースケースを逐次的に処理する何かがいるのかどうか?

課題点 - ドメイン

- ドメインモデルはステートフル
  - ドメインモデルの作成、削除のライフサイクルは誰が管理し、それに紐づくイベントをちゃんと消す仕組みが必要(どこ?)
  - 安易なメモリリークが起きそう
- AドメインモデルがBドメインモデルを監視するというのはありそう(集約的な単位)
  - これを上手く実現するパターンが必要そう

例) 新しい機能を実装する

feat: marking specific page of presentation by azu · Pull Request #5 · azu/presentation-annotator

1. UseCaseを実装する
2. Storeを実装する
3. テストを実装する
4. Componentを実装する

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

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