toCスタートアップにおけるDDDの取捨選択

株式会社NoSchool CTO / meijin

目次

1. TL; DR
2. 自己紹介・事業紹介
3. DDDやクリーンアーキテクチャに対するスタンス
4. 実際の取捨選択事例集
5. まとめ

TL; DR

TL; DR

DDDやクリーンアーキテクチャに関する議論は「要はバランス」に落ち着きがち

toCサービスで3年以上設計と向き合ってきた私が「具体的に何を考えてアーキテクチャのバランスを取ってきたか」という事例をまとめる

よくある取り組みや設計ルールに対して「弊社では無視している」「逆にめっちゃ力を入れている」&「その理由を事業・プロダクト・組織等々の観点から説明」

『DDDやクリーンアーキテクチャ原理主義に従わないことを恐れるな！』

自己紹介・事業紹介

自己紹介

職種・SNS

- 株式会社NoSchool CTO: マナリンク(https://manalink.jp/)の開発
- Twitter(X): 名人｜マナリンクCTO
- Zenn: https://zenn.dev/meijin
- 好きな言語はTypeScript、好きなHTTPヘッダーはCache-Control

趣味

- 将棋☗、カメラ📸、ラム酒🥃、個人開発💻、筋トレ💪、高校野球観戦⚾
- 個人開発：テストメーカー(https://test-maker.app/)

事業紹介

株式会社NoSchoolについて

- 2018年創業・2020年にマナリンクをリリース
- 2024年現在、社員10名超、エンジニア5名

マナリンクについて

- オンライン家庭教師が探せるメディア（先生600人超）
- 授業・売上・宿題等の管理ができるSaaS
- **『メディア的な機能とSaaS的な機能の両方が大事』**
- Web（Next.js×Laravel）、アプリ（React Native）

DDDやクリーンアーキテクチャに対するスタンス

DDDやクリーンアーキテクチャに対するスタンス

1. 「オンライン家庭教師」という過去に存在しない＆法的にも固まっていないドメインで事業をやるので **『ドメイン駆動開発』というより『ドメイン作りながら開発』**
2. メディア機能はパフォーマンス重視、SaaS機能はドメインロジック重視
3. 組織が4〜5人と小規模なので、強いルールを決めるより個人が自律的に意思決定できることを重視
4. 創業時からPHP＆Laravelを使っているので、PHPの限界やLaravelのルールとの競合を意識

以上から

- 機能群ごとに、設計で何を求めるかが変わる（なので一律でDDDやってますと言い辛い）
- ある程度以上の「完璧」を求めることのコスパがかなり悪い

実際の取捨選択事例集

取捨選択したもの（バランスor捨てたorこだわり）

1. 【バランス】`app/Domain`以下に機能群ごとに細かく分割し、各自で設計レベルを決定
2. 【捨てた】「ユビキタス言語」は策定しない
3. 【捨てた】「モデリング」は勝負所の機能だけやる
4. 【こだわり】CQRSパターンの徹底・EntityをRead Modelにしない
5. 【捨てた】結合テストは必須・単体テストは「自分を守るため」に書く
6. 【バランス】メンバーの「ここちょっと設計頑張りたいんすよ」は尊重する
7. 【捨てた】いろいろ言ったけど、大して重要じゃない機能の設計不備は割と許す

`app/Domain` 以下に機能群ごとに細かく分割

事業特性・組織特性

- マナリンク特有の「メディアとSaaSどっちも大事」
- フロント出身のメンバーもいるので、設計難度の低い領域を意図的に作れるとうれしい

Laravelにおける設計

- `app/Domain/Payment`, `app/Domain/TeacherSearch` といった感じで機能群ごとに分割
- 前者は決済系なのでいわゆる”DDD”、後者は先生検索機能なのでRead SQL中心

デメリット

- 開発者視点では、施策ごとに設計難度が上下する（ただしそれが**要件の難易度と近似**する）

「ユビキタス言語」は策定しない

事業特性

- マナリンクはITに慣れていない先生や親御さんが多数利用＆特定の法にほぼ依存しない
- 複雑な用語や仕様にすること自体がアンチパターンのため、用語統一の課題が生まれにくい

ユビキタス言語のデメリット

- 「なんかユビキタス言語を元に議論すると良いらしい」みたいに形から入ってしまいがち
- 事業特性が追いついていないのに策定しても、運用でどうせ風化する

今後

- 今後ずっと不要とは思っていない。課題が表面化してきたら検討します🔥

「モデリング」は勝負所の機能だけやる

事業特性・組織特性

- 新規の柱である先生検索はRead Heavy・継続の柱である指導支援機能がWrite Heavy
- （外から想像する以上に）「えっこの機能のこの情報がこっちの機能でいるの？」が多い
- 例：授業機能で集計された統計情報が、先生一覧のソート順などに影響する

モデリングとの向き合い方

- 契約が絡む機能（決済、授業記録など）だけ丁寧にモデリングをする
- Biz（CEO）にはDDDをやっていると声高に宣言していて、突如要件定義時にモデリングを始めても「ああ、あれか」となるようにしている

CQRSパターンの徹底

事業特性

- オンライン指導の透明化をミッションにしているから、想定外のReadが後から増えがち

課題

- EntityにRepositoryから使われるわけでもないGetterが増えたり、そもそもMutateやドメインロジックに関係ない無駄なデータを持ってしまい、ほぼDTOになる

CQRSパターンの徹底

- EntityをGet時に使わない。更新のための既存データのGetには使う
- `app/Domain/Hoge/UseCase/Query` 以下に特化型のClassが大量に生える

Queryの例

先生が見るカレンダーに表示されるデータ用のクエリ。授業予定もプライベートな予定も両方表示できる数少ない画面なので専用のQueryを用意。Query実装時は必ず唯一のpublicメソッド`handle`を持つことで多目的に使われないようにする。

```php
class MySQLGetCalendarQuery implements GetCalendarQueryInterface
{
    public function __construct(
      private HogeQueryInterface $hogeQuery, private FugaQueryInterface $fugaQuery
    ) {}
    /**
     * 中略
     */
    public function handle(int $teacherId, Carbon $startAt, Carbon $endAt): array
    {
        $lessonSchedules = $this->hogeQuery->handle($teacherId, $startAt, $endAt);
        $hogeSchedules = $this->fugaQuery>handle($teacherId, $startAt, $endAt);

        return [...$lessonSchedules, ...$hogeSchedules];
    }
}
```

結合テストは必須・単体テストは自分を守るため

単体テストを必須化するデメリット

- 設計のアグレッシブな変更に弱い→2次リリース以降でゴリっと改善できない
- 設計より外枠の、大きな要件の変更に対して”現状維持したい”という余計な引力が働く
- 単体と結合っていう分け方がそもそもおかしい、みたいなことに向き合わないといけない

マナリンクでの開発ルール

- バックエンドのAPIテストは必須。フロントエンド全般と、バックエンドの単体テストは任意
- 開発していてTDD的に進めたい＆今後改修時に条件網羅で困るといった事情を各自判断して単体テストやフロントエンドテストを実装
- ※私（CTO）が恐らく世の標準よりかなり”メンバーに任せる”寄りの意思決定者

メンバーの「設計頑張りたいんすよ」は尊重する

これまでの話を受けて逆張り

- 私自身「新しい技術を試したい」「新しい作戦を試したい」欲求は強くあれと思っている
- Too Muchは原則避ける指向性だが、メンバーから「ここは設計頑張りたい」と言われたら尊重することが多い

そのための技術選定でありアーキテクチャ

- `app/Domain/Hoge`で切り分ける作戦・何が何でもAPIテストは必須化するラインなど、ギリギリ耐える出口戦略を普段から採るようにしている
- 開発者自身が最も解像度高いので”設計頑張りたい”という勘は尊重した方が良い
- 『ハードルを下げるのは技術選定の役目、ハードルを越えるのは各メンバーの役目』

大して重要じゃない機能の設計不備は割と許す

現レビュー体制

- （まだ小規模なので）原則CTOがレビュー
- 形だけドメインオブジェクトを使っている・ORMをViewに近いところで使っているなど

とはいえ

- 一定の信頼関係の上で、コメント等で補足をしつつ、リリースが近いのでor要件の不透明度が高いといった理由で"汚い"コードも通すことは多い
- サクッとHogeModelの特定のカラム取りたいだけなのに専用のQuery作って〜とか窮屈なので許すなど。ただし、さすがに許容できないラインも設ける。`$hogeModel->fuga->piyo`といった感じでネストして呼び出すコードはDB構造に強く依存するのでNGなど。

まとめ

まとめ

1. DDDやクリーンアーキテクチャを厳守しなくても機能は作れるし、事業は進む
2. かといってカオスでメンテナンス性ゼロなコードになるのを恐れる気持ちもわかる
3. 機能群でディレクトリ切るとか、単一publicメソッドなクラスを量産するといった、ちょっと理想的ではないけど、めっちゃ事故にはならない、誰でも60点以上は採れるようなラインを目指す方が現実的なことは多いのでは？
4. 本LTが、目の前の事業や組織やプロダクトに対して、本当に今の開発ルールでいいのか？と自問自答するきっかけになると幸いです

おわり