ドメイン駆動設計入門

- 設計はなぜ必要か？
  - リーダビリティ
    - 全員が理解できる
    - 設計書を読めば分かる
  - 変更容易性
    - 後から簡単に変更できる
    - 長期的に運用できる
      - 逆に
        - 1週間ほどで捨てるようなプロジェクトや1回限り動作すれば良いシステムには設計は不要
          - ソフトウェアの二つの価値
            - 機能
              - すぐに動くかどうか
                - すぐに動くものを作るのは誰でもできる
                - 長期的に運用できるのとは雲泥の差がある
              - コードさえ書ければ可能
              - 重要視されやすい
            - 設計
              - 後から変更が容易にできるかどうか
              - 経験と知識が必要
              - 軽視されやすい
  - テスタビリティ
    - 小さく作り、組み合わせる
    - テストをしやすい設計は良い設計
      - テスト駆動設計（TDD）

ドメイン駆動設計とは？

- ドメインとは？
  - ソフトウェアを適用する領域のこと
  - 重要なのは
    - ドメインが何かではなく
    - ドメインに含まれるものは何かをいうこと
  - 物流システムを作るなら、トラックの積載容量や燃費は適用されるべき領域だが、トラックの語源は別に不要よねという話
- ドメイン駆動設計って何？
  - ソフトウェア開発手法の一種
  - ドメインの知識に焦点を当てた設計手法
  - 知識をコードに埋め込むことを実現する
    - だから
      - ドメインやドメインロジックを何よりも大切にし、ドメインオブジェクトとその他雑多な処理を分けて混合しないようにしてる
      - 開発者はドメインを純粋で有用な構造として維持するために大量の分離とカプセル化を実装する必要があると批判されているが
        - 保守性などの利点を提供する
  - 当たり前を当たり前に実践するための手法
    - ソフトウェアを適用する領域と向き合い、
    - そこに渦巻く知識に焦点を当てる

よく観察し、よく表現すること
なぜ、今ドメイン駆動設計か？
提唱されたのは2003年ごろ
一昔前はサービスをいち早く世に出すことこそが重要だった
片道ロケットのようなもの
ドメイン駆動設計は重厚で鈍いものと敬遠されてた
それに比べドメイン駆動設計は
飛行機のようなもの
速度は圧倒的に見劣りするが
安定運用できる
当時ドメイン駆動設計を取り入れたが
あまり効果は見られなかった
しかし
時が流れ先人たちによって撒かれた種が芽吹いたと言われてる
システムを長期的に運用したいときに
ドメイン駆動設計が生きてくる
そもそもモデルとは？
現実の事象あるいは概念を抽象化した概念のこと
全て忠実に再現することはできないため、必要に応じて取捨選択を行う（ドメインによる）
ドメインモデルとは？
ドメインの概念をモデリングして得られたモデルのこと
あくまでも概念を抽象化した知識にとどまる
ドメインの概念の射影
ドメインエキスパート（物流会社の人）と開発者が協力してドメインモデルを作り上げる必要がある
コミュニケーションスキルも大事
そこがおろそかになると全く違ったドメインモデルが構築され、それに従ってできたドメインオブジェクトも徒労に終わる
ドメインオブジェクトとは？
ドメインモデルをソフトウェアで動作するモジュールとして表現したもの
知識をコードで表現したのがドメインオブジェクト
ドメインモデルはあくまでも概念を抽象化した知識に留まる
何かしらの媒体で表現されることで問題解決の力を得る
ドメインオブジェクトの選別
どのドメインモデルをドメインオブジェクトとして実装するのかは重要な問題
利用者の問題の解決に何ら関わりのないドメインモデルをドメインオブジェクトとして実装したところでそれは徒労
ドメインオブジェクトを採用するメリット
コードのドキュメント性が高まる
無口なコードに比べコードを饒舌にする努力をすればそれらのコードはドキュメントのようになる
ドメインにおける変更をコードに伝えやすくする
ただのデータ構造体であるならルールの変化をコードに反映するのは極めて困難な作業
ドメインオブジェクトにルールやふるまいを記述することはドメインからドメインモデルへ伝播した変化をドメインオブジェクトまで到達させるために必要なこ

- と
- とにかくコードを饒舌にする努力をしろという話
- ドメインとドメインモデルとドメインオブジェクトの関係性
  - イテレーティブ（反復的）な関係
    - ドメインの変化はドメインモデルを媒介にして連鎖的にドメインオブジェクトまで伝えられる
    - ドメインオブジェクトがドメインに対する態度を変化させることもある
      - プログラムは人の曖昧さを受け入れない
      - ドメインに対する曖昧な理解による実装の障害を解決するにはドメインモデルを見直し、最終的にはドメインの概念に対する捉え方を変える必要が出てくる
      - ドメインに対する鋭い洞察は実装時にも得られるもの
- どうやって知識を表現する？
  - 値オブジェクト
    - 値オブジェクトとは？
      - 業務で使う単位や値のルールをクラスとして表現したもの
      - ふるまいを定義できるという特徴
        - オブジェクトに対する操作をふるまいとして一処にまとめることで、値オブジェクトはルールを語るドメインオブジェクトらしさを帯びる
        - 値オブジェクトはデータを保持するコンテナではなく、ふるまいを持つことができるオブジェクト
        - 例えば
          - 加算を行うふるまいをAddメソッドとして実装したりできる
    - そもそも値の性質とは
      - 不変である
        - 普段やっているのは値の変更ではなく「代入」
      - 交換が可能である
      - 等価性によって比較される
        - =等価であれば同一とみなす
    - 値オブジェクトにする基準
      - 基本はコンテキストによる
      - あるとすれば
        - そこにルールが存在しているか
        - それ単体で取り扱いたいか
      - 例えば
        - 氏名には姓と名で構成されるというルールがある
        - 氏名は単体で取り扱っているが
        - 姓だけを取り出したり、名だけを利用するシーンはない場合
        - 値オブジェクトにするのは氏名だけでいいよね
    - 値オブジェクトを採用するモチベーション
      - より説明的になる
        - 製品番号の例
          - プリミティブな値で"a2020-100-1"
          - 値オブジェクトで表すと、productCodeに"a2020"、branchに"100"、lotに"1"で構成される
        - 自己文書化を推し進める
      - 不正な値を存在させない
        - ガード節で不正な値を弾くことができる

- 誤った代入を防ぐ
  - タイプ不一致エラーとして検知する
  - 値オブジェクトを作ることで型の恩恵を受けることができる
- ロジックの散在を防ぐ
  - ルールをまとめることは変更箇所をまとめることと同義
- まとめ
  - 値オブジェクトのコンセプトは
    - システム固有の値を作る
  - システムにはそのシステムならではの値が必ずあるはず
    - プリミティブな値だけでソフトウェアを構築することは可能だが、、
    - 汎用的すぎて表現力が乏しくなってしまう
  - ドメインに存在するさまざまなルールは値オブジェクトに記述され
    - コードがドキュメントとして機能し始める

エンティティ

- エンティティとは？
  - 値オブジェクトと同じくドメインモデルを実装したドメインオブジェクト
  - 属性ではなく同一性によって識別されるオブジェクト
    - 同一性とは？
      - それをそれたらしめる所以
      - 人の例
        - ある人の体重が昨日と今日で異なっていたとしても別人になったというわけではない
        - 同姓同名の人間同士が同じ人間問うわけではない
  - 値オブジェクトと対をなすドメインオブジェクト
- エンティティの性質について
  - 可変である
    - 値オブジェクトは不変なオブジェクトであった
    - 変更の表現の仕方
      - 値オブジェクト
        - 交換（代入）によって表現していた
      - エンティティ
        - ふるまいを通じて属性を変更する
        - 具体的にはChangeNameというふるまいを定義してそれを通じて属性を変更する
  - 同じ属性であっても区別される
    - 同姓同名の人が同一と見做されるのは現実世界ではおかしいみたいな話
    - 同一かどうかは識別子によって区別される
      - user_idとか
  - 同一性により区別される
    - 属性ではなく識別子のみを比較することによって同一性の比較ができる
- 判断基準
  - ライフサイクルと連続性の存在
    - 例えばユーザというのは作成されて生を受け、削除されて死を迎えるというライフサイクルをもち、連続性のある概念
  - 基本的にシステムにとってライフサイクルを表現することが無意味な場合は
    - ひとまずは値オブジェクトとして取り扱うと良い
    - 可変なオブジェクトは取り扱いに慎重さが要求される厄介なもの

- 不変にして置けるものは出来るだけ不変なオブジェクトのままにして取り扱いたい
- 捉え方によっては値オブジェクトとエンティティのどちらにもなりうる
  - 車にとってのタイヤとタイヤ工場にとってのタイヤの具体例
- メモ
  - railsで言ったら
    - modelの役割が多すぎるので
      - modelを
        - entityとリポジトリに分けれる
- ドメインサービス
  - そもそもサービスというのは2種類ある
    - ドメインサービス ← ここではこっち
    - アプリケーションサービス
  - ドメインサービスとは？
    - 不自然さを解消するオブジェクト
      - 例えば
        - ユーザが他のユーザとの重複を確認しているのは不自然
        - それなら、ユーザサービスという別のオブジェクトがユーザ同士を比べて重複を確認するという方が自然
    - 値オブジェクトやエンティティと異なり、自身の振る舞いを変更するようなインスタンス特有の状態を持たないオブジェクトのこと
  - どんなときに使う？
    - 値オブジェクトやエンティティに定義すると不自然に感じる操作があるとき
    - 可能な限りドメインサービスを避ける
    - ユーザの重複確認の例
    - 物流システムの物流拠点の例
  - ドメインサービスを濫用すると？
    - 実は全てのふるまいをドメインサービスに移すことも理論上可能
    - ドメインオブジェクトを無口なオブジェクトに変容させてしまう
    - ドメインモデル貧血症を引き起こす
  - まとめ
    - 複数のドメインオブジェクトを横断するような操作には不自然に思える振る舞いが存在する
      - そんなときはドメインサービス
    - 便利すぎる存在ゆえ、濫用に気をつけろ
      - さもなくば、活き活きと自分のルールを語っていたドメインオブジェクトが無口になってしまう
    - ドメインの概念を表現するには、値オブジェクト、エンティティ、それで不自然な場合はドメインサービスを利用する
- どうやってアプリケーションを実現する？
  - リポジトリ
    - リポジトリとは？
      - データの保管庫
      - データを永続化し再構築するといった処理を抽象的に扱うためのオブジェクト
      - データにまつわる処理を分離する
      - データストアへのアクセスは下位レベルの操作
        - ソフトウェアからすれば、MySQLであろうがNoSQLであろうが何でもいい

- リポジトリを利用するだけでデータストアの差し替えやテストの実施など、柔軟に切り替えを行える
- 処理の意図を鮮明にする
- リポジトリの責務
  - ドメインオブジェクトの永続化や再構築を行うこと
- リポジトリの定義
  - インターフェース
    - 抽象的な振る舞いが定義されている
    - ドメインオブジェクトはまずここにお願いをする
    - ロジックを組み立てる分にはインターフェースのみで十分
  - 実体部分
    - SQLなど技術基盤に依存した処理を記述
    - 実際にデータストアに保存したり更新したりする
- まとめ
  - データストアに対する細かい操作をリポジトリに任せるとビジネスロジックが鮮明になるよねというお話
  - このリポジトリを工夫すればテストを気軽に実施できるようになるよね
- アプリケーションサービス
  - アプリケーションサービスとは？
    - ドメインオブジェクトが行うタスクの進行を管理し、問題の解決に導くもの
    - ユースケースを実現するオブジェクト
      - ユーザ機能におけるユースケースにはどんなのがある？
        - ユーザを登録する
        - ユーザ情報を確認する
        - ユーザ情報を更新する
        - 退会する
  - 気をつけるべきこと
    - ドメインのルールを流出させない
      - アプリケーションサービスはあくまでもドメインオブジェクトのタスク調整に徹するべき
      - ドメインのルールは記述してはいけない
    - 凝集度を高める
      - そもそも凝集度とは？
        - モジュールの責任範囲がどれだけ集中しているかを測る尺度
          - その結果高まるもの
            - 堅牢性
            - 信頼性
            - 再利用性
            - 可読性
      - 全てのインスタンス変数は全てのメソッドで使われるべき
      - 凝集度を高めるために分離して、まとまりが分かりづらくなったら？
        - パッケージを利用する
          - ディレクトリ構造に反映されることも多い
          - 要するにユーザに関する操作関連をまとめたらいい
  - アプリケーションサービスのインターフェース
    - より柔軟性を担保するために用意することがある
    - 具体的には？

- before
  - Client
    - UserRegisterService
- after
  - Client
    - IUserRegisterService （インターフェース）
      - UserRegisterService
      - MockUserRegisterService
      - ExceptionUserRegisterService
- クライアントはアプリケーションサービスの実体を直接呼び出すのではなくインターフェースを操作して処理を呼び出すようになる
- クライアント側の利便性を高める
  - モックオブジェクトを利用してプロダクション用のアプリケーションサービスの実装を待たずして開発を進めることができる
  - クライアント側の処理を実際にテストしたいときにも差し替えて簡単に対応できる
- まとめ
  - アプリケーションサービスはドメインオブジェクトの力をまとめあげてドメインの問題を解決に導くよ
  - アプリケーションサービスはドメインオブジェクトの操作に徹することでユースケースを実現するよ
  - そのときドメインのルールが記述されないように気をつけなさいよ
- ファクトリ
  - ファクトリとは？
    - オブジェクトの生成に関わる知識がまとめられたオブジェクトのこと
    - 前提として複雑な道具はその生成過程も...
      - 複雑になってる
    - 複雑な手順を必要とするオブジェクトの生成をモデルを表現するオブジェクトに実装すると
      - オブジェクトの趣旨がぼやける
    - オブジェクトを生成する責務を持ったオブジェクトが必要よねって話
  - 分かりやすい例としては
    - 採番処理
      - そのまま記述すると
        - 高レベルな概念であるオブジェクトに低レベルなデータベース操作が記述されてしまう
        - かといって採番テーブルを用意するのも面倒だし直感的でない
      - ファクトリを用いると
        - まずファクトリのインターフェースを用意する
          - これはコンストラクタの代わりとして利用される
        - ユーザ登録処理では、userのインスタンス作成をファクトリ経由で行えるようになる
        - Userクラスのコンストラクタにはデータベース操作を記述しなくて良くなる
          - 結果
            - オブジェクトが鮮明になる
    - 車
      - そのまま記述すると
        - 本体、エンジン、タイヤ..などなど様々なもの組み合わせる処理が必要
          - これがCarオブジェクトに記述されると

モデルを表現するはずのオブジェクトに複雑な生成処理の記述が大半を締めてしまいコードがぼやける
Clientからしたら車を扱いたいのにその生成処理も視界に入るのは嫌だ
ファクトリを用いると
Carには複雑な生成を行うファクトリのインターフェースを記述するだけにとどまる
コードの趣旨が鮮明になる
ファクトリ生成の動機付け
本来初期化は
コンストラクタの役目
コンストラクタは単純である必要がある
単純でなくなるときファクトリを定義せよ
つまり
「コンストラクタ内で他のオブジェクトを生成するかどうか」
もしコンストラクタが他のオブジェクトを生成するようなことがあれば...
オブジェクトが変更されるシアにコンストラクタも変更しなければいけなくなる恐れが
主張としては
×
ただ漠然とインスタンス化をするのではなく
○
ファクトリを導入すべきか検討する習慣を身につけるべきである
まとめ
ファクトリはオブジェクトのライフサイクルの始まりの役割を果たす
複雑な処理を伴うオブジェクトの生成にファクトリをしようすることで
コードの論点が明確になる
全く同じ生成処理が散在することを防ぐ
ファクトリによって生成処理をカプセル化することで
ロジックの意図が明確になる
柔軟性を確保できる
知識を表現する、より発展的なパターン
集約
集約とは？
シンプルに言えば
変更の単位
データを変更するための単位として扱われるオブジェクトの集まり
集約にはルートとなるオブジェクトが存在する
全ての操作はルートを介して行われる
これによって
集約内部への操作に制限がかかる
集約内の不変条件は維持される
集約は不変条件を維持する単位として切り出され、オブジェクトの操作に秩序をもたらす
集約の基本構造
境界
集約に何が含まれているのかを定義
集約の外部から境界の内部のオブジェクトを操作してはいけない
ルート

集約に含まれる特定のオブジェクト
集約内部のオブジェクトに対する変更は集約ルートが責任を持って行う
そうすることで
集約内部の不変条件を保てる
オブジェクトの操作に関する基本的な原則
外部から内部のオブジェクトに対して直接操作するのではなく
それを保持するオブジェクトに依頼する形をとる
そうすることで
コードが直感的になる
不変条件を維持することができる
「デメテルの法則」という
どんなもの？
オブジェクト同士のメソッド呼び出しに秩序をもたらすガイドライン
ソフトウェアのメンテナンス性を向上させる
コードをより柔軟なものへ導く
集約が成し遂げようとしてることと同じこと
メソッドを呼び出すオブジェクトは4つ
オブジェクト自身
インスタンス変数
引数として渡されたオブジェクト
直接インスタンス化したオブジェクト
主張としては
オブジェクトのフィールドに直接命令をするのではなく、それを保持するオブジェクトに対して命令を行、フィールドは保持しているオブジェクト自身が管理すべき
例えばでいうと...
車を運転するときにタイヤに対して直接命令しないよねって話
タイヤを動かして前に進みたい時はそれを保持する車に対して命令を行って、タイヤの管理は車がすべきよねってこと
このルールに至る理由
フィールドがゲッターを通じて公開されていると
本来オブジェクトに記述されるべきルールがどこかで漏れ出すことを防げない
IsFullメソッドの具体例
内部データを隠蔽するために
そもそもの話
オブジェクトの内部データは無闇やたらに公開すべきものではない
しかし
完全に非公開にしてしまうとリポジトリがインスタンスを永続化しようとしたときに困ってしまう
例として、
UserクラスのIdやNameが非公開になっているとコンパイルエラーが起きる
アプローチとして
ルールによる防衛
ゲッターを使わないように紳士協定をひく
しかしこれは制限力が低い
通知オブジェクトを使う

- オブジェクトの内部データを非公開にしたまま、外部に対して引き渡すことが可能に
- これデザパタで出てきたような..

- 集約をどう区切るかについて
  - これはとても難しいテーマ
  - 方針として最もメジャーなものは
    - 変更の単位で境界を引く
      - これはあえて違反してみると理由がわかりやすい
        - CircleとUserという集約があるとする
        - サークル集約からユーザ集約の変更をすると
          - 影響はリポジトリに現れる
            - サークルリポジトリのロジックの多くがユーザの更新処理に汚染される
            - 追加されたコードとほぼ同じコードがユーザのリポジトリにも存在する
          - これらは本来の変更の単位を超えて変更を行なったため発生した問題
      - リポジトリは変更の単位である集約ごとに用意する
  - そもそもできてしまうことを問題視する考え方もある
    - 変更しないことを不文律として課すよりも有効な手段は？
      - インスタンスを持たないという選択肢
        - インスタンスを持たなければメソッドを呼び出しようがない
        - インスタンスを持たないけれど、保持しているように見せかけるものとして識別子がある
        - サークル集約をユーザ集約を直接保持するのではなく識別子をインスタンスの代わりに持たせる
          - そうすることで
            - 不意にメソッドを呼び出して変更してしまうことはなくなる
            - メモリの節約にもつながる
  - まとめ
    - 集約とはオブジェクトが持つ不変条件を守る境界
    - システマチックに定義できるものではない
      - ドメインに渦巻く概念はそのほとんどが連なっている
      - そこに境界線を引くことは難しいことである
    - 集約の境界線を引くことは
      - ドメインの概念を捉え
      - そこにある不変条件を導き出し
      - ドメインとシステムを両天秤にかけながら最適解を目指すような作業

- 仕様
  - 仕様とは？
    - あるオブジェクトがある評価基準に達しているかを判定するオブジェクトのこと
    - 例
      - サークルの話
        - サークルは人数が30人を超えてはいけないというルールがある
        - それを評価するためのメソッドとしてIsFull()というメソッドを作成した
        - この時点では単純な条件なので問題は生まれない
        - この条件が複雑になってくると話は別
          - プレミアムユーザの登場
          - プレミアムユーザが10人以上だとサークルの人数は50人に引き上げられるという条件付加

- これらの評価メソッドをアプリケーションサービスに記述するとどうなるか
  - そもそもサークルが満員かどうかの確認はドメインの重要なルール
  - サービスにドメインのルールに基づくロジックが流出してしまっている状態
  - ドメインのルールはドメインオブジェクトに定義するという原則に反する
- 解決策として
  - 仕様と呼ばれるオブジェクトを利用する
    - 具体的には
      - 複雑な評価手順をオブジェクトに埋もれさせず切り出す
    - メリットとして
      - コードの趣旨が明確になる
      - 複雑な評価手順はカプセル化される
- 仕様とリポジトリを組み合わせる
  - 仕様は単独で取り扱う以外にもリポジトリと組み合わせて活用する手法が存在する
    - リポジトリに仕様を引き渡して、仕様に合致するオブジェクトを検索する手法
  - 検索処理の例
    - 検索処理の中には重要なルールを含むものがある
    - このような検索処理をリポジトリのメソッドとして定義すると。。
      - 重要なルールがリポジトリの実装クラスに記述されることになってしまう！！
    - 解決策
      - 重要なルールを仕様オブジェクトとして定義し、リポジトリに引き渡す
        - 結果
          - 重要なルールがリポジトリの実装クラスに漏れ出すことを防げる
    - おすすめサークルの例
      - おすすめする条件
        - 直近1ヶ月以内に結成
        - メンバーが10人以上
      - これの条件をサークルリポジトリに書いたら重要なルールがリポジトリの実体クラスに漏れ出しちゃう
      - 解決策
        - 仕様として定義する
        - アプリケーション内で、リポジトリからサークルをfindAllして取得する
        - 取得した全サークルを仕様オブジェクトに定義したIsSatisfiedByというメソッドに渡してフィルタリングする
        - パフォーマンス的に問題があるのでコレクションが行われるまでクエリを発行しない等の工夫は必要
          - railsでクエリ発行のタイミングに癖があるのはこれのためか(?)
- まとめ
  - オブジェクトの評価はそれ単体で知識になりうる
  - 仕様は評価の条件や手順をモデルにしたオブジェクトのこと
  - オブジェクトの評価をオブジェクト自身にさせることが常に正しいとは限らない
  - 仕様のように外部のオブジェクトに評価させる手法の方が素直なコードになるよ
  - 読み取り操作は単純ながら最適化を求められることも多い
    - ドメインという考え方を棚上げにしてクライアントが利用しやすい形で提供することもあると覚悟せよ

大切なこと

- 依存関係のコントロール

- ソフトウェアを柔軟に保つために必要なこと
  - 特定の技術的要素への依存を避ける
  - 変更の主導権を主たる抽象に移す
- 依存関係逆転の原則
  - 定義
    - 上位レベルのモジュールは下位レベルのモジュールに依存してはならない、どちらのモジュールも抽象に依存すべきである
    - 抽象は、実装の詳細に依存してはならない。実装の詳細が抽象に依存するべきである
  - 抽象に依存せよ
    - レベルとは？
      - 入出力からの距離を示すもの
      - 低レベルといえば
        - 機械に近い具体的な処理
          - データストアを操作する処理など
      - 高レベルといえば
        - 人間に近い抽象的な処理
          - アプリケーションサービスなど
    - アプリケーションサービスは上位レベルのモジュールでありながら、データストアの操作を行う下位レベルのモジュールに依存しているのは原則に反しているって話
      - そんな時は抽象型を導入しよう（挟もう）
  - 主導権を抽象に
    - これができてないと...
      - 低レベルなモジュールの変更が重要な高レベルのモジュールに波及してしまう
        - 例えば、データストアの変更を理由にビジネスロジックを変更するみたいなことは起きてほしくないよねって話
    - インターフェースを宣言し、低レベルのモジュールはそのインターフェースに合わせて実装を行う
      - より重要な高次元の概念に主導権を握らせることが可能に
- どうやってコントロールしていく？
  - Service Locatorパターン
    - どんなパターン？
      - 返却されるインスタンスを事前に登録しておき、ServiceLocaterと呼ばれるオブジェクトに依存解決先となるオブジェクトを事前に登録しておき、インスタンスが必要となる各所でServiceLocaterを経由してインスタンスを取得するパターン
    - メリット
      - リポジトリのインスタンス化を行うコードが随所に点在しなくなり修正箇所が簡潔になる
    - 問題点
      - 依存関係が外部から見えづらくなる
        - インスタンス化してるとこだけ見たら事前に登録してるかどうかが分からんからエラー起こしてしまうよねって話
      - テストの維持が難しくなる
        - 新たなテストコードが追加されたときにテストが破壊される
          - それ自体が問題ではなく、テストを実行するその時までわからないことが

問題
依存関係の変更に自然と気づき、テストコードの変更を余儀なくさせる強制力を持たせることができなければいけない
IoC Containerパターン
まずDependency Injectionパターンについて知る必要あり
依存するモジュールを外部から注入してる
テストを実施するために開発者はコンパイルエラーを解消することを余儀なくされる
この強制力が素晴らしい
デメリット
依存するオブジェクトのインスタンス化をあちこちに記述しなければいけなくなる
Dependency Injectionパターンのデメリットを解決するために出てきた
ごめんちょっとここら辺意味わからん
まとめ
依存関係は自然に発生するもの
依存が発生することは避けられないが、依存の方向性はコントロールしようね
コントロールしないと硬直するよ
主要なロジックを技術的な要素に依存させないようにしようね
柔軟なソフトウェアが保てるぜって話
整合性のお話
整合性とは？
矛盾がなく一貫性のあること
ズレがないこと
どうやって守るか
ユニークキー制約
データベースの特定のカラムが唯一無二のものであることを保証する機能のこと
じゃあもう重複確認なんかせずこれだけしとけばよくね？
あかん理由
重複に関するルールをコードから読み取れないようになる
特定の技術基盤に依存するようになってしまう
あくまでセーフティとして活用されるべき機能
トランザクション
ある操作の完了時にコミットが行われるまでデータストアに変更はされない仕組み
低次元レベルであるトランザクション処理をアプリケーションサービスに書きたくない
ビジネスロジックに書くべきは
整合性が必要とされる処理であることを明示的に主張すること
それを実現するためにあるパターン
トランザクションスコープ
AOP
ユニットオブワーク
トランザクションが引き起こすロックについて
どれほどロックされるかは常に念頭に置くべきこと
できる限り小さくすべき
広範囲に渡ると..?

それに比例するように処理が失敗する可能性が高まる
ロックを狭める指針
一度のトランザクションで保存するオブジェクトを一つに限定する
そのオブジェクトをなるべく小さくする
アーキテクチャ
アーキテクチャとは？
知識を記述すべき箇所を示す方針のこと
ドメインのルールが流出することを予防するのと同時に1箇所にまとめるように促す
大事なこと
ドメイン駆動設計にとってアーキテクチャは決して主役ではない
ドメイン駆動設計にとってのアーキテクチャの立ち位置
前提として
ビジネスロジックを正しい場所に配置し続けることは熟練開発者であっても難しいもの
開発者に強い自制心を促す以外の方法を考える必要がある
その解決策としてのアーキテクチャがある
アーキテクチャは方針である
何がどこに記述されるべきかといった疑問に対する解答を明確にし、ロジックが無秩序に点在することを防ぐ
これはドメイン駆動設計の本質である「ドメインを捉え、うまく表現する」ことに集中するために必要なこと
ドメイン駆動設計がアーキテクチャに求めること
ドメインオブジェクトが渦巻くレイヤーを隔離すること
ソフトウェア特有の事情からドメインオブジェクトを防衛すること
ドメイン駆動設計にとってはドメインが隔離されることのみが重要
ドメインの本質に集中する環境を用意することが重要
アーキテクチャに従ったからといってドメイン駆動設計を実践したことにはならない
全ての詳細がドメインに対して依存するようにすることはソフトウェアの方針を最も重要なドメインに握らせることを可能にしてい
三つの代表的アーキテクチャ
レイヤードアーキテクチャ
いくつかの層が積み重なる形で表現される
四つの層で構成されたものが代表的
プレゼンテーション層
ユーザインタフェースとアプリケーションを結びつける役割
主な責務は
表示
システム利用者にわかるように表示を行い
解釈
システム利用者の入力を解釈する
アプリケーション層
ドメイン層の住人を取りまとめる層
アプリケーションサービスが挙げられる
ユースケースを実現するための進行役になる
ドメイン層
最も重要な層

ソフトウェアを適用しようとしている領域で問題解決に必要な知識を表現している
ドメイン層に本来所属すべきドメインオブジェクトの隔離を促す
他の層へ流出しないようにする
インフラストラクチャ層
他の層を支える技術的基盤へのアクセスを提供する層
主に
アプリケーションのためのメッセージ送信を行うモジュール
ドメインのための永続化を行うモジュール
原則
依存の方向は上から下
上位のレイヤーは自身より下位のレイヤーに依存することが許される
ヘキサゴナルアーキテクチャ
六角形をモチーフにしてる
コンセプト
アプリケーションとそれ以外のインターフェースや保存媒体は付け外しできるようにする
アプリケーション以外のモジュールは差し替え可能
インターフェースや保存媒体が変更されるような事態が起きても
コアとなるアプリケーションにその余波は及ばない
例
ゲーム機
コントローラーやモニター
どんなコントローラーでも動作する
どんなモニターに繋いでも動作する
アプリケーション
クリーンアーキテクチャ
四つの同心円で説明される
コンセプト
ユーザインターフェースやデータストアなどの詳細を端に追いやり、依存の方向を内側に向けることで
詳細が抽象に依存するという依存関係逆転の原則を達成する
ヘキサゴナルアーキテクチャと目的は同じ
じゃあどこが違う？
コンセプトを実現する具体的な実装方針が明示されている点
まあ要するに
ビジネスルールをカプセル化したモジュールを中心に捉え、依存の方向を絶対的に制御している
アーキテクチャの共通点
一度に多くのことを考えすぎないこと
アーキテクチャは方針を示し、各所で考える範囲を狭めることで集中を促す
まとめ
ドメイン駆動設計においてアーキテクチャは主役じゃない
ドメインに集中するための方針
ドメインの隔離を促すことができればどんなアーキテクチャでもいい