Clean Architecture

クリーンアーキテクチャ Clean Architecture

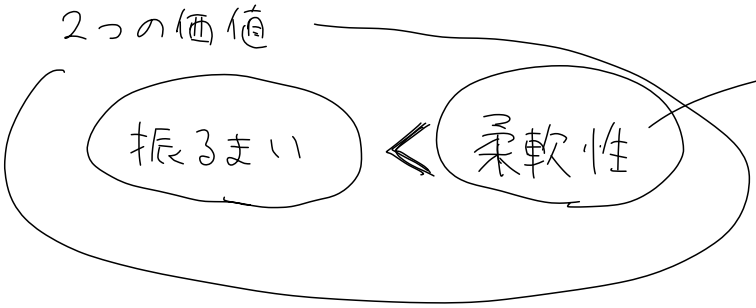
達人に学ぶソフトウェアの構造と設計

Robert C. Martin 著
角 征典、髙木正弘 訳

アーキテクチャのルールはどれも同じである！

ASCII DWANGO

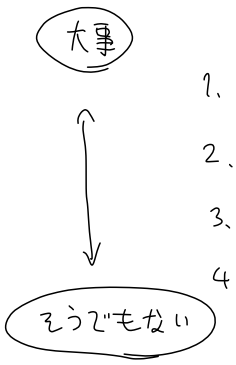
第2章　2つの価値のお話



変更の難易度は 変更の形状でなく
変更のスコープに比例.

↳ 形状にとらわれないアーキテクチャ
が重要.

重要性 × 緊急性 のマトリクス

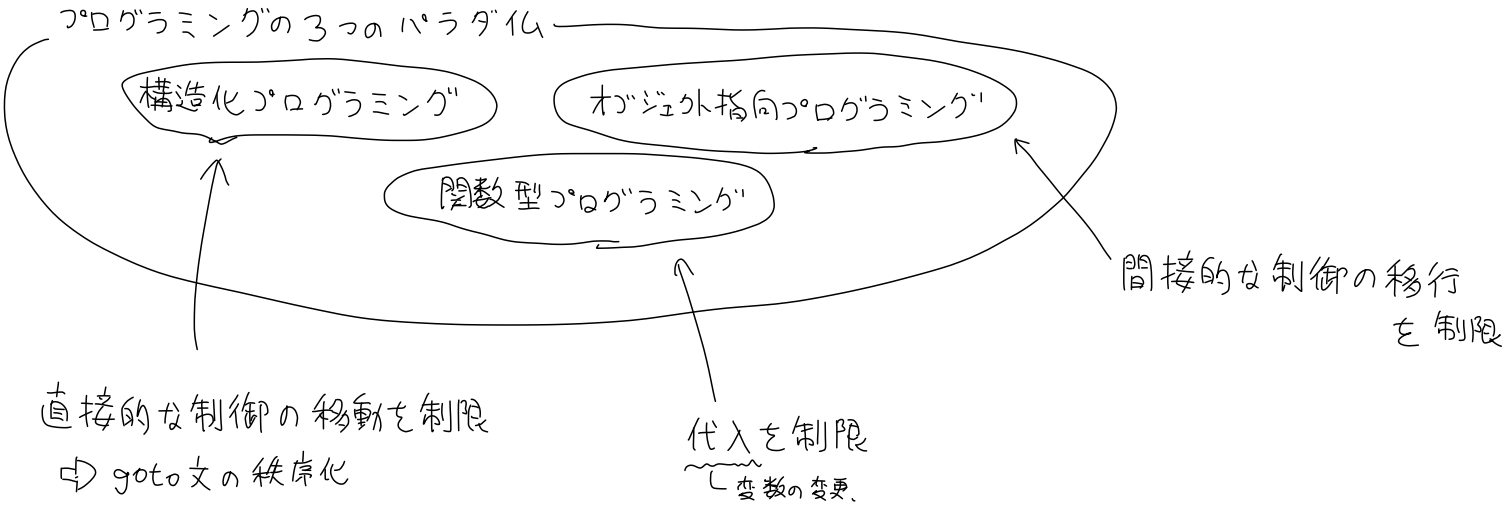


1. 緊急 かつ 重要
2. 緊急ではないが重要
3. 緊急だが重要ではない
4. どちらでもない

☆ 3. を 1. に昇格させた
考え方をしないこと!

⇨ アーキテクチャは 2. に概当.
ソフトウェアエンジニアは 3 な実装を主張する
ビジネスパーソンと戦い, 2 を死守すべき.

第3章　パラダイムの概要



※ 3つのパラダイムは何を「すべきでない」かを示す。

今後のキーワード

- コンポーネントの分離
- データ管理
- 機能

第4章　構造化プログラミング

ダイクストラ(Dijkstra)：オランダ初のプログラマ.

数学の証明と同じようにプログラムの正統性を証明する.

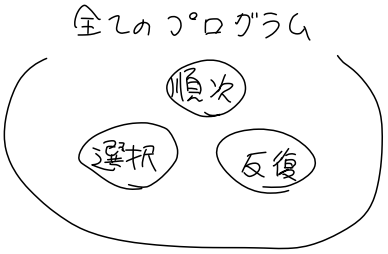
→ 無秩序なgoto文の使用は証明の不可能性を生み出すことを発見.

⇩

現代のメジャー言語ではgoto作用を極部分的に採用.

これこそが構造化プログラミング

→ goto文を用いないことでプログラムが機能上再帰的に分割可能になる.



★Dijkstraの証明は結局普及せず

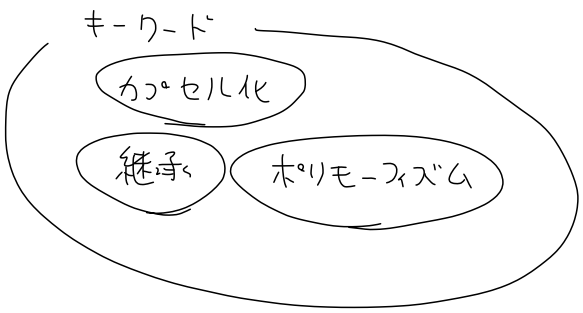
⇩

数学的手法に代わり科学的手法が定着.

→ 十分な労力をかけたテストでバグが存在しなければ科学的にプログラムは正しい(反証不可能)とする手法.

第5章　オブジェクト指向プログラミング

オブジェクト指向 ＝「ポリモーフィズムを使用することで，システムにあるすべてのコードの依存関係を絶対的に制御する能力」



カプセル化

→ データや関数を取りまとめ，外界との境界線を定ぎ．
それぞれのデータを外界に見せるか否かを制御する．

C言語

ヘッダーと実装が完全に分離していたため，完ぺきなカプセル化が可能．

C++

コンパイラの技術的にヘッダーと実装が一体．
⇨ 変数へのアクセスは防げるが存在は知られる．

Java / C#

ヘッダーと実装を分離しない

→ 言語の構文レベルでカプセル化を表現．

※ カプセル化はOO言語ではむしろ弱体化している

継承

C

手法的には完ぺきな継承が可能だが，トリック的．

C++ / Java / C#

便利な継承機能を持つ．

ポリモーフィズム

C

「ポインタを初期化するときはポインタを経由して関数を呼び出す」
という規則をつかうと全員が準守すれば成立.

OO言語

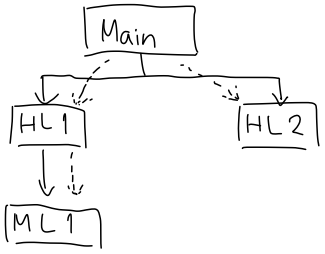
上記規則が言語レベルで実現.

☆ ポインタ使用上の安全性が格段に上がった

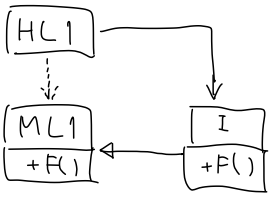
⇩同時に間接的な制御の移行に制限.

依存関係逆転

典型的な呼び出しツリーでは制御の流れと
ソースコードの依存関係の方向が一致.



OO言語では逆転が可能.



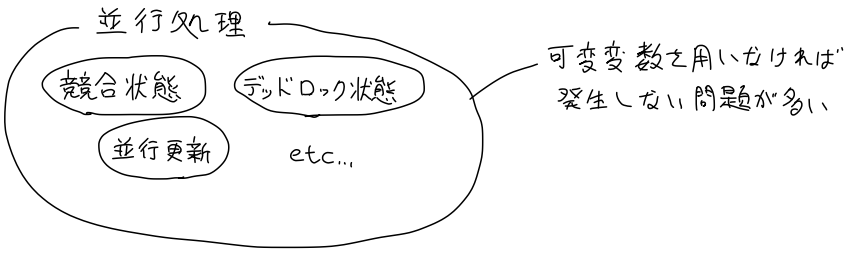
Interfaceを挟むことによっていかなる部分も流れを逆転させられる.

⇩例えばUIやDBをビジネスロジックから分離させられるように.

→ 独立デプロイ可能性, 独立開発可能性の担保.

第6章　関数型プログラミング

関数型プログラミングでは 変数が変化しない.



⇨ 可変コンポーネント と 不変コンポーネントの分離

→ 単純な 比較と置換のアルゴリズムでも実現できるが 大規模になると管理が困難.

☆ 可能な限り不変コンポーネントに処理を持たせる.

⇨ イベントソーシング

☆ 状態を保存するのではなく,「状態を変化させる操作」を保存する.

→ リソースが多く必要にはなるが 完全に不変化できる.

第7章 SRP：単一責任の原則

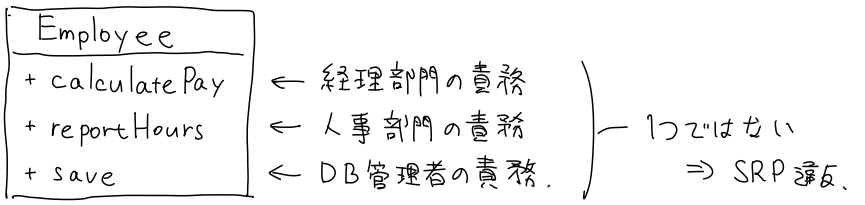
S O L I D

モジュールはたったひとつのアクターに対して責務を負うべきである。

≒ モジュールを変更する理由はたった1つだけであるべきである

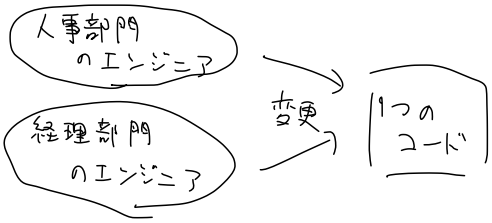
≒ モジュールはたったひとりのユーザやステークホルダーに対して責務を負うべきである。

〈NG例〉



★内部的に別のアクターの責務のコードに依存すると変更時にミスを犯しやすい。

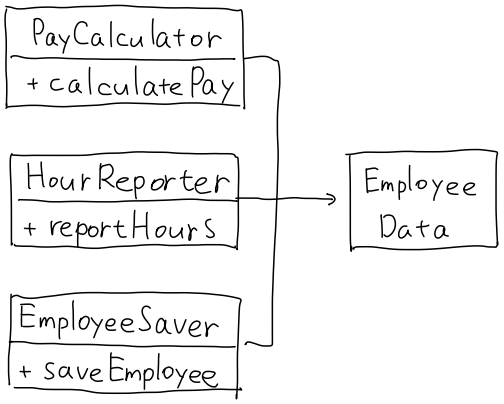
★ エンジニアリング的にも



という状況が起こる可能性が高くなり、コンフリクトによるバグを誘因する。

〈解決策〉

★ データと関数を分離すればOK。

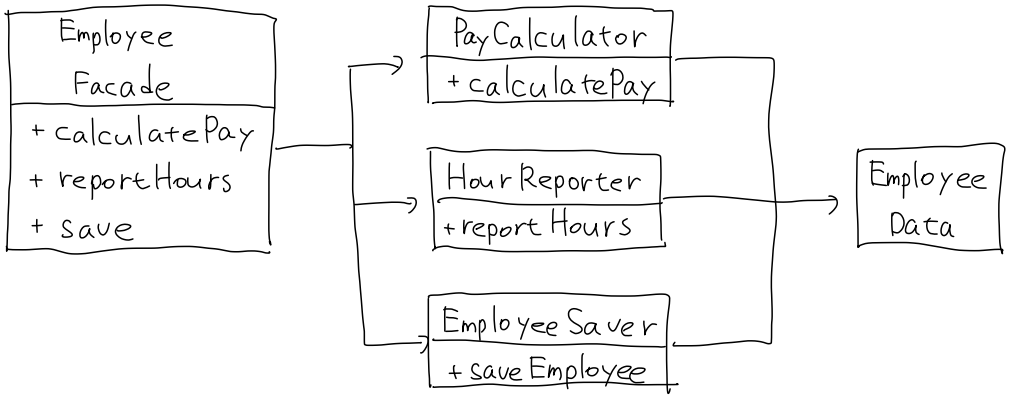


- 3つのクラスをインスタンス化して単一で保持し続けなければいけない弱点。

Facade パターン

Facadeの責務

・実行したいメソッドを持つクラスを
インスタンス化して処理を委譲するだけ



※ 一部分だけFacadeとするのもアリ.

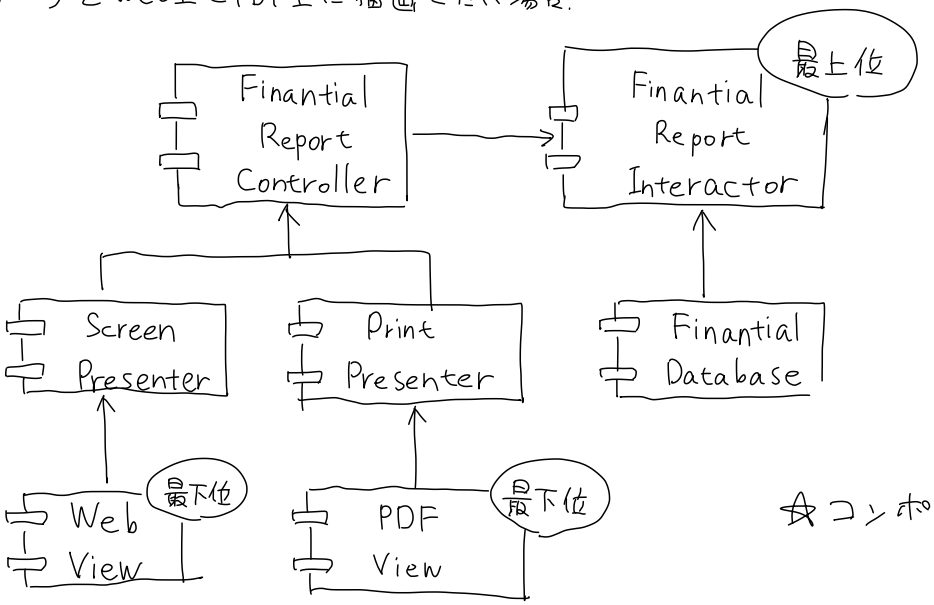
・元のEmployeeクラスにsaveとdataのみ残し, calculatePay / reportHoursのみはFacade化するなど

第8章　OCP：オープン・クローズドの原則　SOLID

ソフトウェアの構成要素は拡張に対しては開いていて，修正に対して閉じていなければいけない．

〈OK例〉

・財務データをweb上とPDF上に描画したい場合．



☆コンポーネントの依存関係は常に一方通行．

☆変更の可能性が高いコンポーネントほど上位に配置すべき．
例）WebViewを変更してもScreenPresenter以下には何ら影響を与えない．

☆Interactorはアプリケーションのビジネスルールを含んでおり，最上位レベルの方針を含んでいるため，他の全てが変更されたとしても影響を受けない．

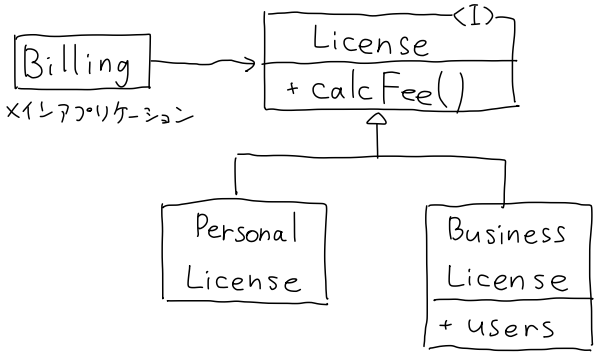
☆周辺的な処理ほど下位コンポーネントに任せる．

Interfaceを使って依存関係をコントロールするのがめちゃくちゃ重要

第9章　LSP：リスコフの置換原則　　SOLID

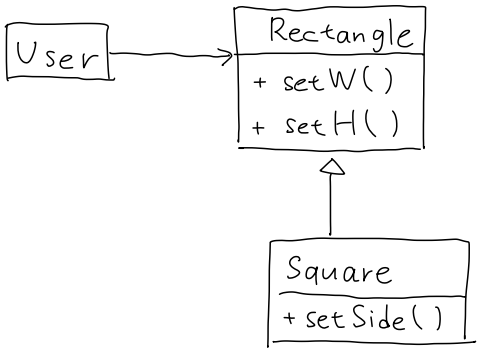
S型のオブジェクトo1の各々に，対応するT型のオブジェクトo2が1つ存在し，Tを使って定義されたプログラムPに対してo2の代わりにo1を使ってもPの振る舞いが変わらない場合，SはTの派生型．

〈OK例〉



☆ BillingはPersonalLicenseやBusinessLicenseには依存していないのでLSPを満たしている．

〈NG例〉



☆ UserがSquareをRectangleとして扱おうとするとHとWが一致しない状態になり得るので置換不可能．
⇒ LSPを満たしていない．