

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

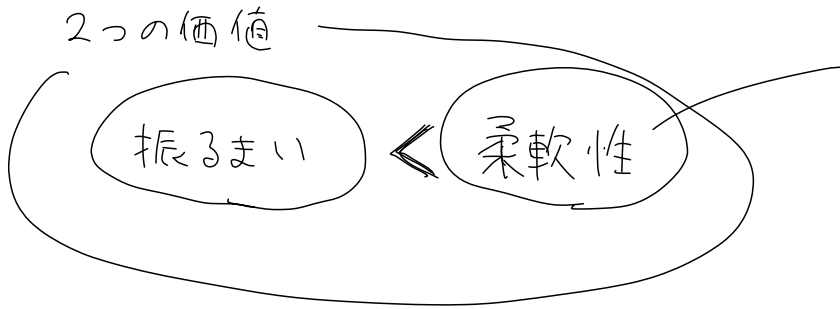
Robert C. Martin 著  
角 征典、高木正弘 訳

アーキテクチャの  
ルールはどれも  
同じである！

ASCII  
DWANGO



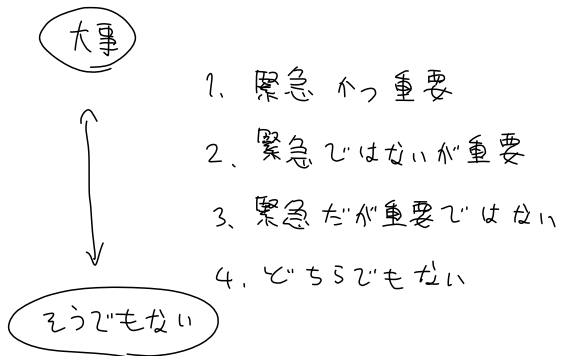
## 第2章 2つの価値のお話



変更の難易度は変更の形状でなく  
変更のスコアに比例.

↳ 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

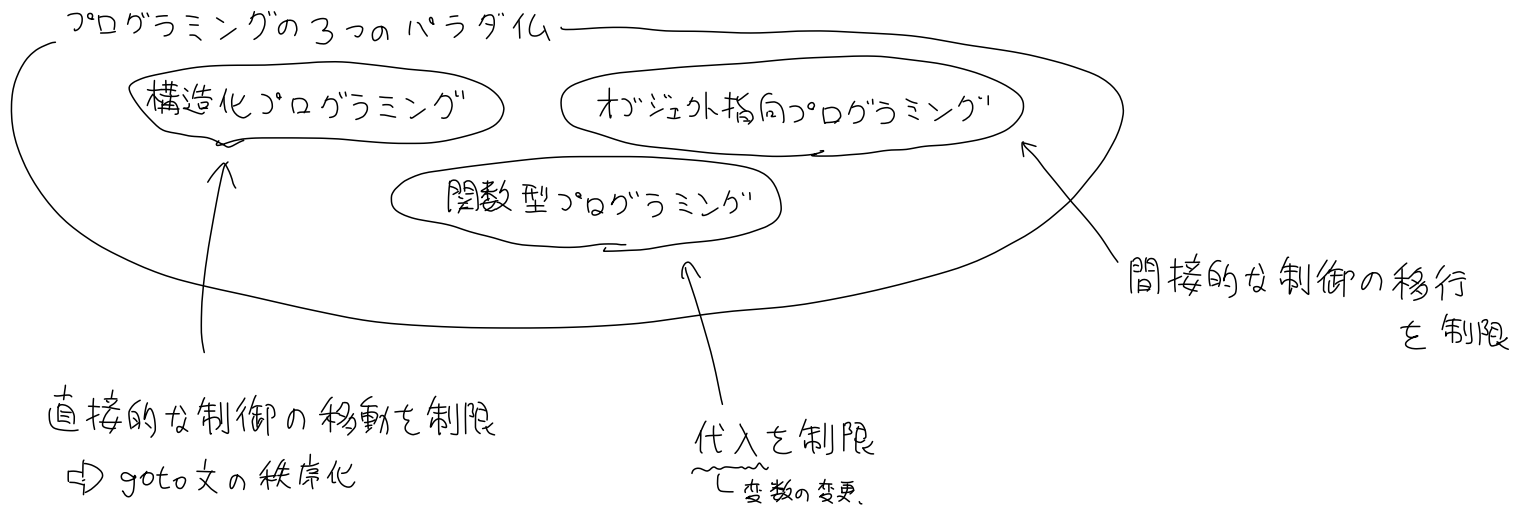


★ 3. を 1. に昇格させた  
考え方をしないこと!

⇒ アーキテクチャは 2. に相当.

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ー.

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.

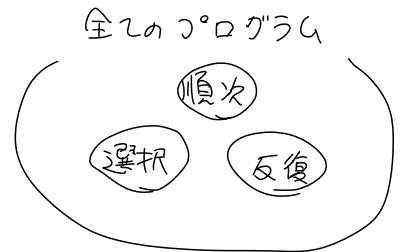


現代のXジャー言語では goto 作用を極部分的に採用.



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

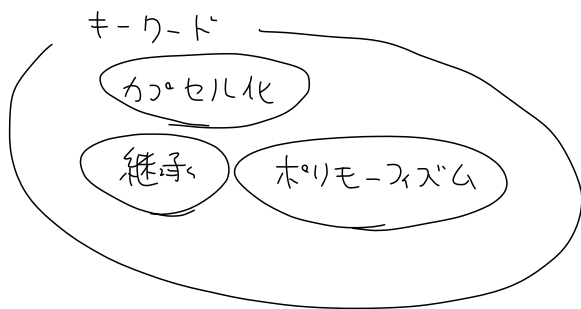


数学的手法に代わり 科学的手法 が席卷.

→ 十分な労力をかけたテストでバグが存在しなかった  
科学的にプログラムは正しい(反証不可能)となる手法.

# 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め、  
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、完璧なカプセル化が可能。

### C++

コンパイルの技術的にヘッダーと実装がくっ着く。

⇒ 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C言語ではむしろ弱体化している

## 継承

### C

手法的には完璧な継承が可能だが、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

①

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。  
という規則をつづらうと全員が遵守すれば成立。

OO言語

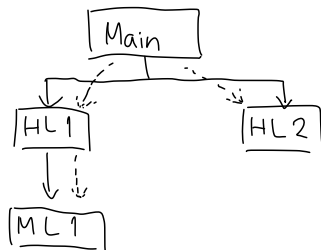
上記規則が言語レベルで実現。

★ ポインタ使用上の安全性が格段に上がった

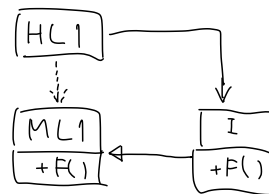
⇒ 同時に間接的な制御の乱行に制限。

## 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。



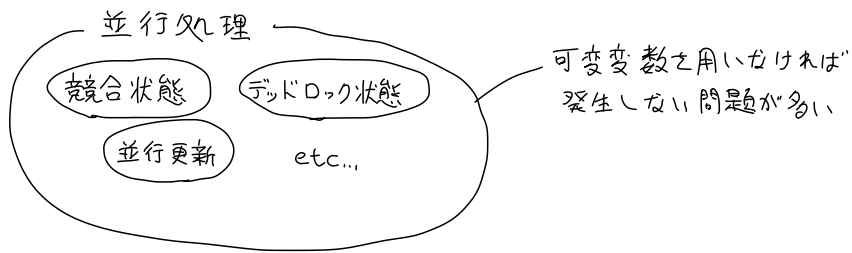
Interfaceを抜き出して、いかなる部分も流れを逆転させられる。

⇒ 例えば UI や DB を ビジネスロジックから分離させるように。

→ 独立な実装可能性、独立な関数可能性の担保。

# 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。

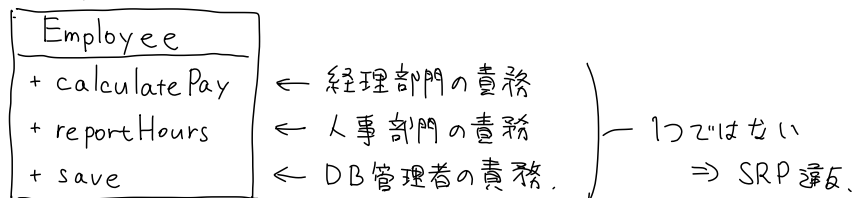
# 第7章 SRP: 単一責任の原則.

モジュールはた、たひとつのアクターに対し責務を負うべきである.

≡ モジュールを変更する理由はた、た1つだけであるべきである

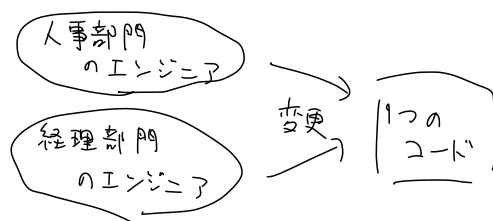
≡ モジュールはた、たひとりのユーザーやステークホルダーに対し責務を負うべきである.

<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.

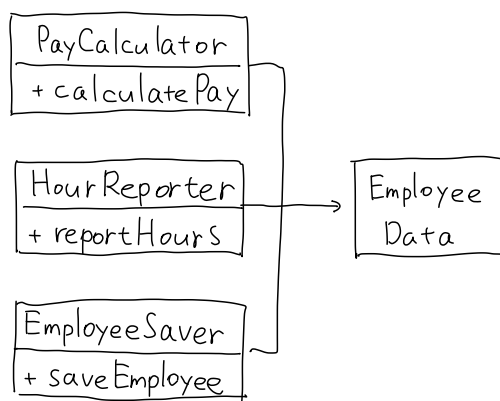
★ エンジニアリング的にも



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.



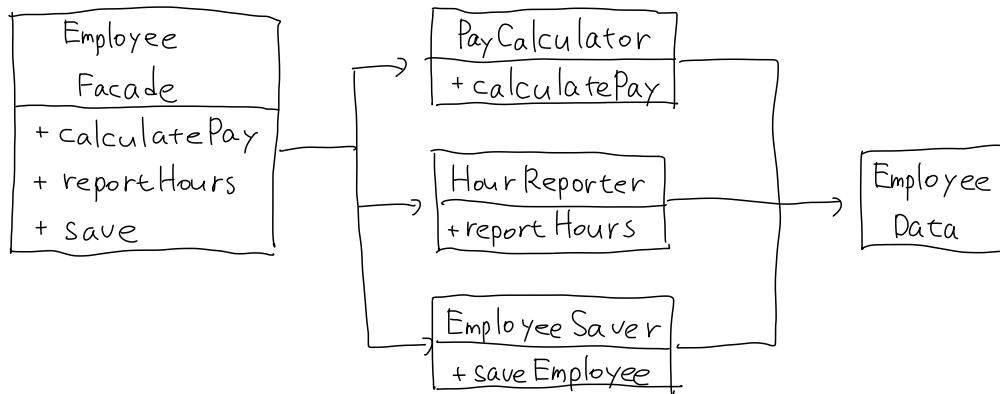
● 3つのクラスをインスタンス化して単一で保持し続けなければならない弱点.



## 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 とするのもあり。

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