ソフトウェアアーキテクチャ研修

yamamoto kiyohiro

2022/09/05

研修の前に…

問

「良い設計」とは何か？

研修の目的

「良い設計」について考えるための土台を作る

キーワードの認識、基礎知識の獲得

今日の内容

- ソフトウェアアーキテクチャとは
  - ソフトウェアアーキテクチャの目的
  - ソフトウェアアーキテクチャの法則

- プログラミングパラダイム
  - 構造化プログラミング
  - オブジェクト指向プログラミング
  - 関数型プログラミング

- ソフトウェア設計における原則
  - プログラミングパラダイム
  - 凝集度と結合度
  - SOLID 原則

- 設計パターン紹介
  - MVC
  - DDD
  - クリーンアーキテクチャ

ソフトウェアアーキテクチャとは

ソフトウェアアーキテクチャとは

ソフトウェアアーキテクチャ（英: Software Architecture）は、ソフトウェアコンポーネント、それらの外部特性、またそれらの相互関係から構成される。また、この用語はシステムのソフトウェアアーキテクチャの文書化を意味することもある。ソフトウェアアーキテクチャの文書は開発依頼主とのコミュニケーションを容易にするもので、概要レベルの設計に関する早期の決定を促し、プロジェクト間でのコンポーネントとパターンの設計を再利用することを可能にする[1]。

https://ja.wikipedia.org/wiki/ソフトウェアアーキテクチャ

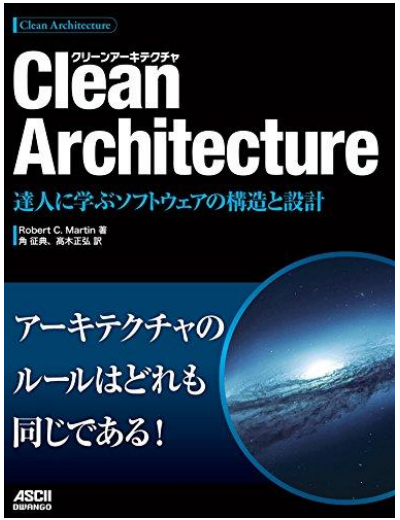
ソフトウェアアーキテクチャとは

ソフトウェアアーキテクチャとは、抽象化と問題の分割によって複雑性を減らすことを主に念頭に置いたものである。ただし、今までのところ、「ソフトウェアアーキテクチャ」という用語に関して、万人が合意した厳密な定義は存在しない[4]。

https://ja.wikipedia.org/wiki/ソフトウェアアーキテクチャ

ソフトウェアアーキテクチャの目的

> ソフトウェアアーキテクチャの目的は、求められるシステムを
> 構築・保守するために必要な人材を最小限に抑えることである
>
> — *Robert C. martin*



https://itkonekt.com/2019/12/19/robert-c-martin-uncle-bob-2/

ソフトウェアアーキテクチャの法則

Q. この研修で最強のアーキテクチャが作れるようになる？

> アーキテクチャには正解も間違いもない。
>
> ただトレードオフがあるだけだ。
>
> — *Neal Ford*

24.4 別れの挨拶 より

ソフトウェアアーキテクチャの法則

ソフトウェアアーキテクチャはトレードオフがすべてだ。

ソフトウェアアーキテクチャの第一法則

↓必然的帰結↓

アーキテクトが、トレードオフではない何かを見出したと考えているなら
まだトレードオフを特定していないだけの可能性が高い。

「どうやって」よりも「なぜ」の方がずっと重要だ。

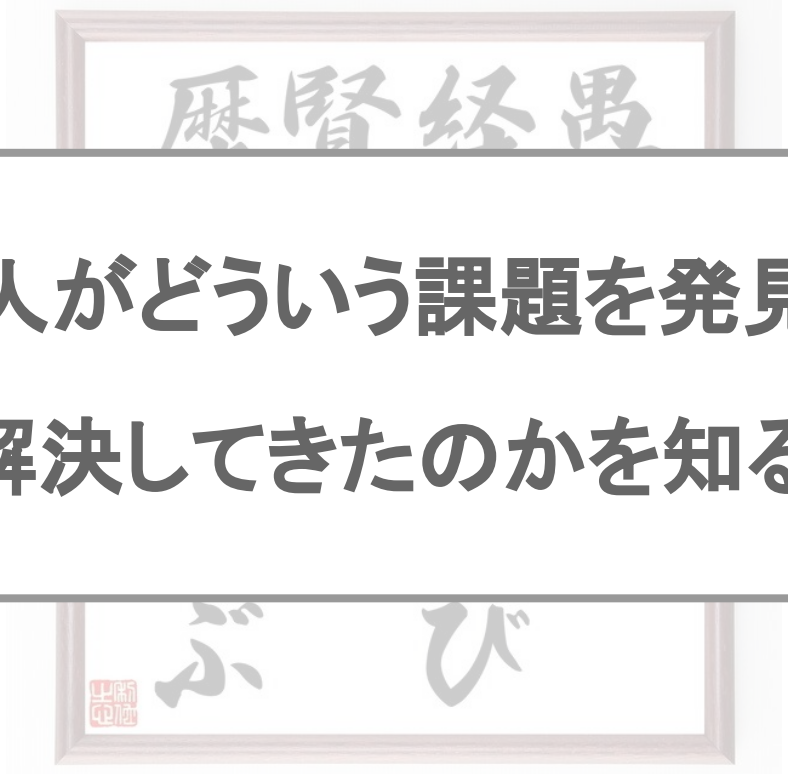
ソフトウェアアーキテクチャの第二法則

ソフトウェアアーキテクチャの基礎　1章　イントロダクション　より

トレードオフの見つけ方

では、どのようにトレードオフを見つければよいのか？

トレードオフの見つけ方



先人がどういう課題を発見し

解決してきたのかを知る

プログラミングパラダイム

プログラミングパラダイム

例 [編集]

この節には独自研究が含まれているおそれがあります。問題箇所を検証し出典を追加して、記事の改善にご協力ください。議論はノートを参照し

比較されるものは横に並べてある。括弧内はそれを用いている例である。

- 構造化プログラミング - 非構造化プログラミング
- 命令型プログラミング - 宣言型プログラミング
- メッセージ送信プログラミング（アクターモデル）
- 手続き型プログラミング - 非手続き型言語
- イベント駆動型プログラミング
- シグナルプログラミング
- スタック指向プログラミング
- クラスベースプログラミング - プロトタイプベースプログラミング ※オブジェクト指向プログラミングの中での分類
- 並行論理プログラミング
- 制約プログラミング
- 論理プログラミング
- 解集合プログラミング（en:Answer Set Programming）
- 制約論理プログラミング
- 並行プログラミング
- 並行制約プログラミング
- 関数型プログラミング
- コンポーネント指向プログラミング (OLE)
- アスペクト指向プログラミング (AspectJ)
- 契約プログラミング
- リフレクティブプログラミング
- データフロープログラミング
- リアクティブプログラミング（英語版） (スプレッドシート)

https://ja.wikipedia.org/wiki/プログラミングパラダイム

プログラミングパラダイム

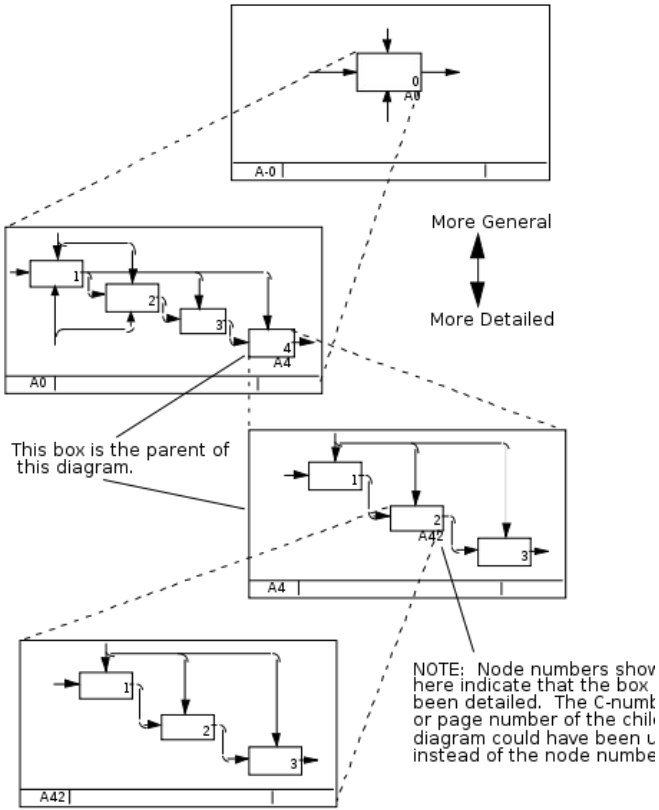
中でも特に重要な3つのパラダイム

- 構造化プログラミング
- オブジェクト指向プログラミング
- 関数型プログラミング

それぞれ簡単に特徴を見てみる

構造化プログラミング

構造化プログラミング



https://ja.wikipedia.org/wiki/構造化プログラミング

構造化プログラミング

パラダイムの要約

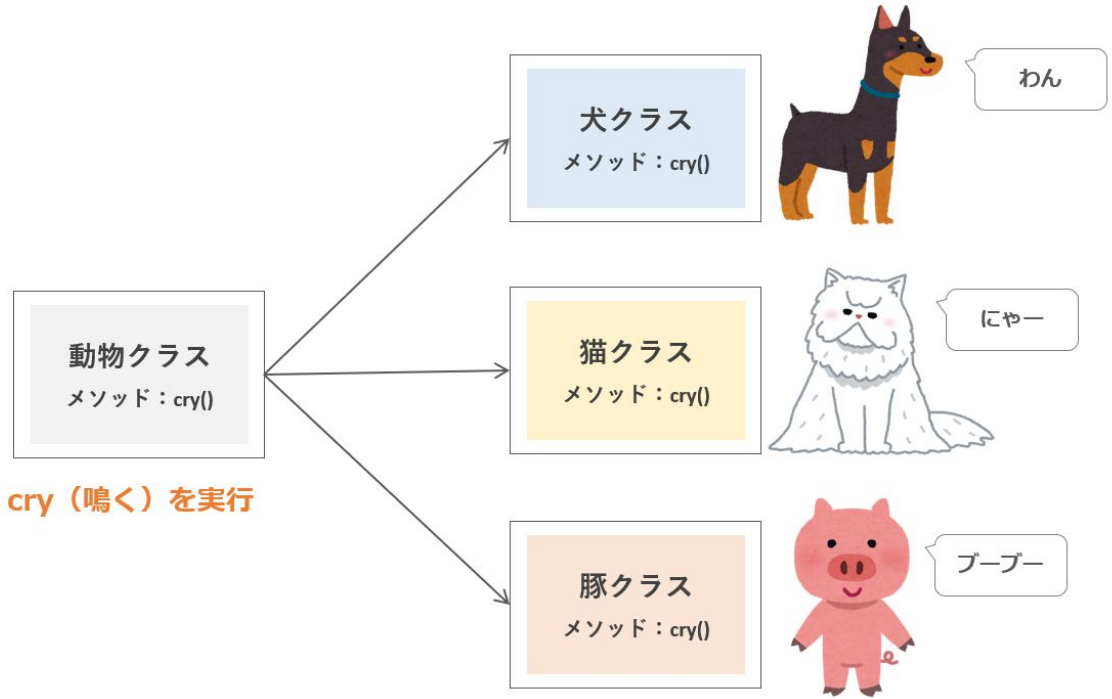
「構造化プログラミングは
直接的な制御の移行に規律を課すものである」

制限のないジャンプ（goto文）を
if/then/else や do/while/until などに置き換えた

これにより分割統治が可能となりテストもしやすくなった

オブジェクト指向プログラミング

オブジェクト指向プログラミング



https://medium-company.com/ポリモーフィズム/

オブジェクト指向プログラミング

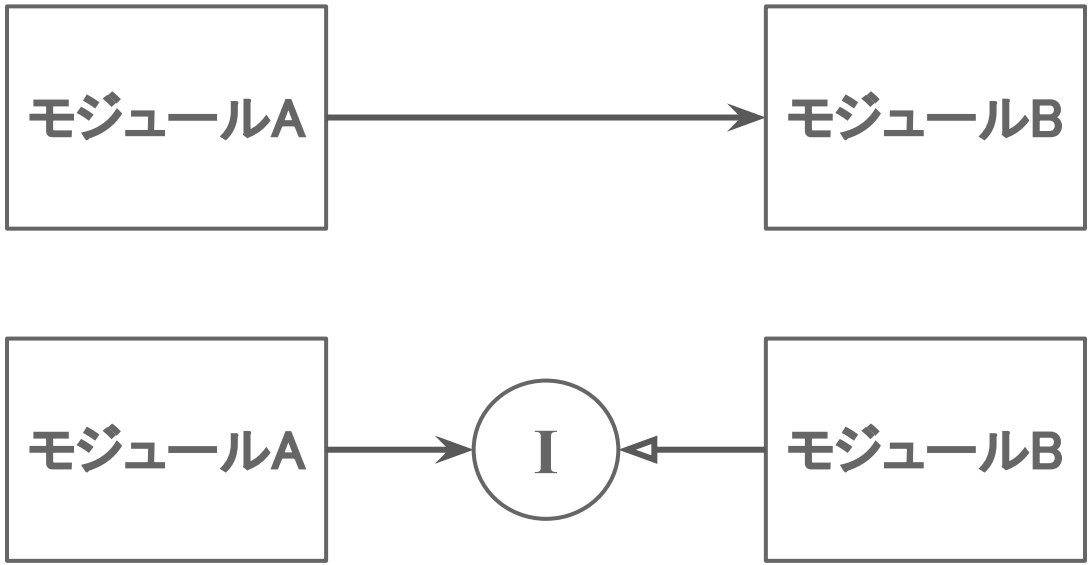
パラダイムの要約

「オブジェクト指向プログラミングは
間接的な制御の移行に規律を課すものである」

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ソースコードの依存関係を絶対的に制御する能力

これにより「プラグインアーキテクチャ」を利用できる

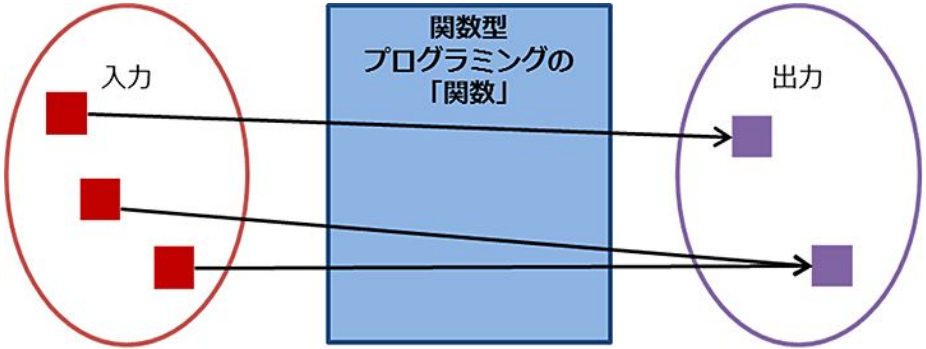
オブジェクト指向プログラミング



依存関係逆転

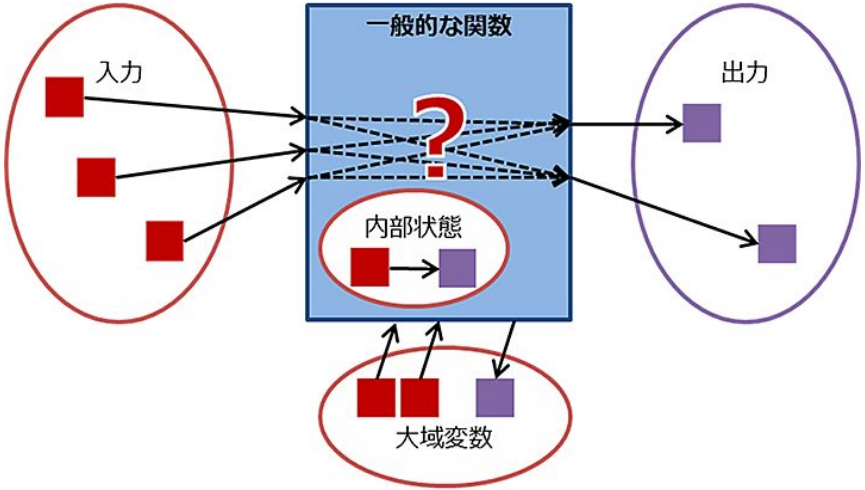
関数型プログラミング

関数型プログラミング



「関数」の出力は、入力だけで決まる

↑ 関数型プログラミングの「関数」



関数の出力は、入力だけでなく内部状態や大域変数の影響を受ける

↑ 一般的な関数

https://www.nttdata.com/jp/ja/data-insight/2013/032101/

関数型プログラミング

パラダイムの要約

「関数型プログラミングは
代入に規律を課すものである」

競合状態、デッドロック状態、並行更新の問題の原因が全て可変変数にある

→ 不変なコンポーネントと可変なコンポーネントに分離する考えを得る

これらのパラダイムは

いつ頃のものなのだろうか

プログラミングパラダイムまとめ

- 構造化プログラミング
- オブジェクト指向プログラミング
- 関数型プログラミング

これらのパラダイムは 1958~1968 の 10 年間に発見された

それから何十年も重要なパラダイムは追加されていない

ツールもハードウェアも変わったが、**ソフトウェアの本質は変わらない**

これら本質を理解することが、良い設計の理解に繋がる

ソフトウェアアーキテクチャにおける原則

原則の紹介

・構造化プログラミング

→ 凝集度と結合度

・オブジェクト指向プログラミング

→ SOLID 原則

原則としてはこの辺も有名だが、割愛

DRY：重複を避けろ
（*Don't Repeat Yourself*）

YAGNI：必要になるまで実装するな
（*You Aren't Gonna Need It*）

KISS：不必要な複雑性を避けろ
（Keep It Simple Stupid）

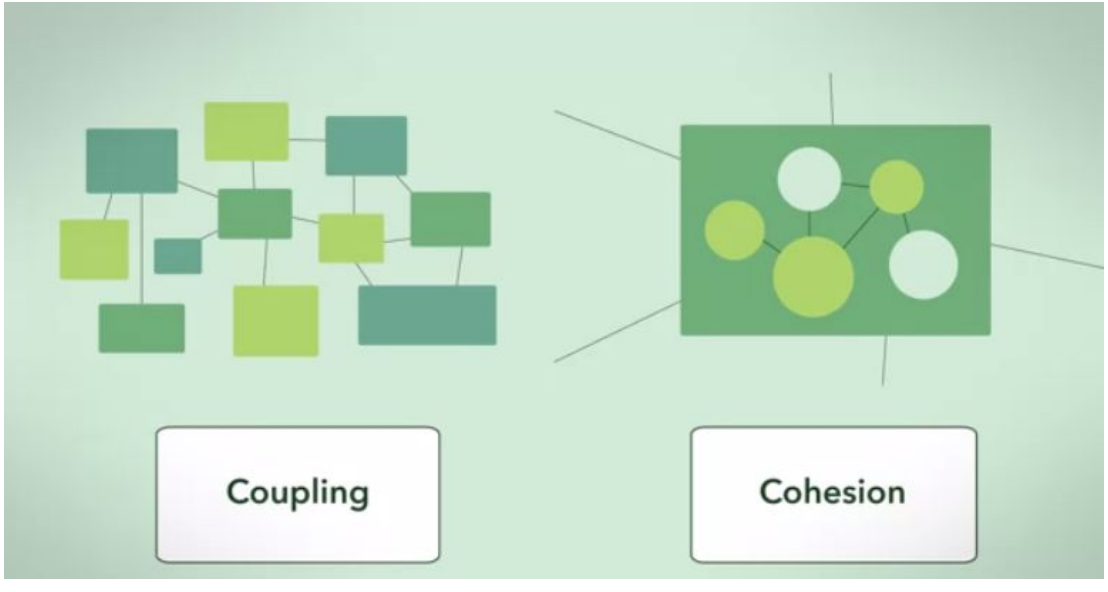
それぞれのパラダイムの中で特に有名な原則（考え方）を紹介

凝集度と結合度

凝集度と結合度

凝集度(Cohesion):モジュール内の協調に関する指標

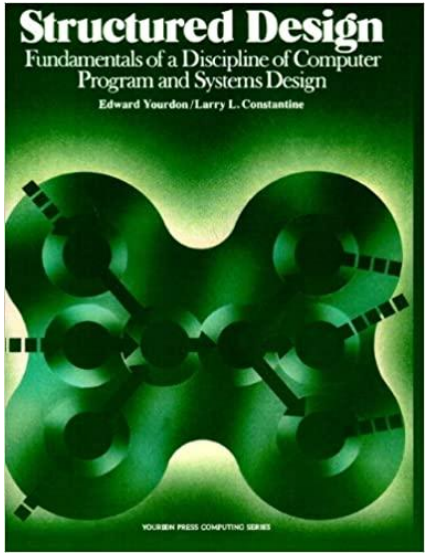
結合度(Coupling) :モジュール間の連携に関する指標



凝集度と結合度

1979年に発表された
構造化プログラミングに関する論文
（構造化設計）の中で提唱されたもの

原則ではないが
設計における様々な場面で有用な指標



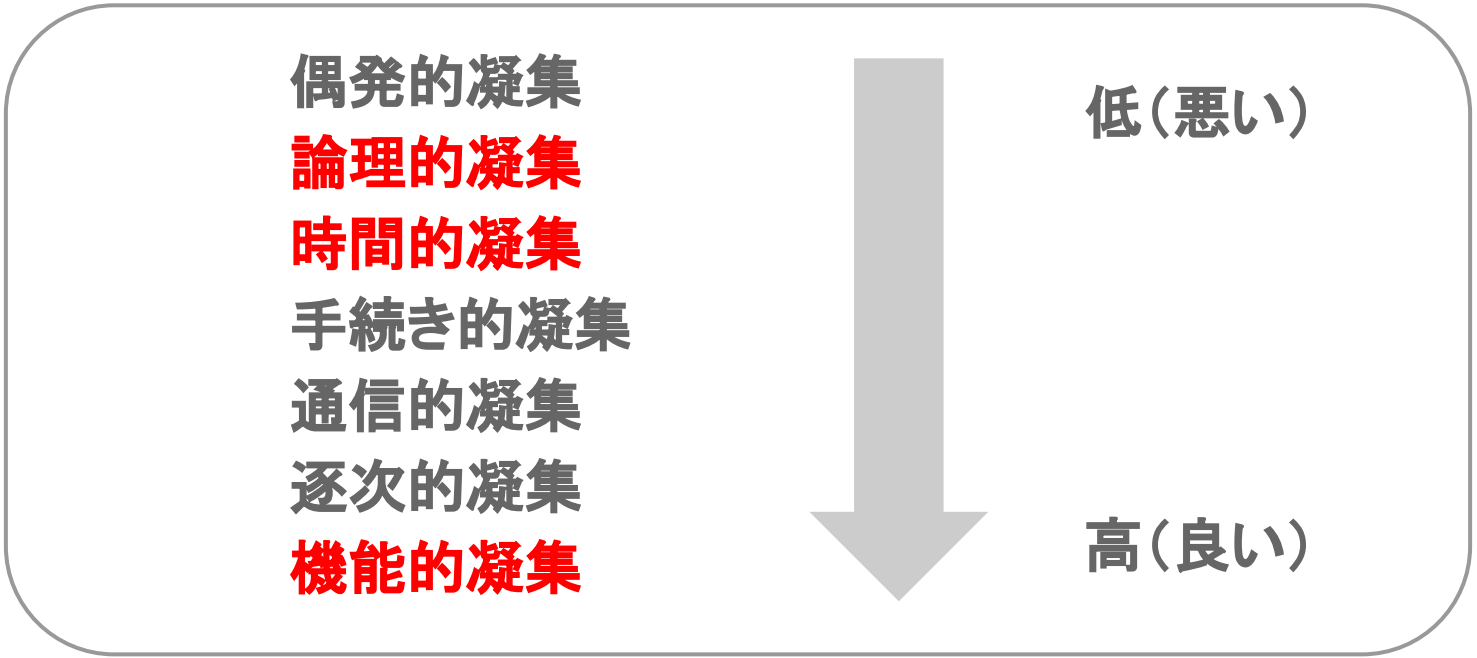
凝集度

凝集度

凝集度：モジュール内の協調に関する指標

凝集度

凝集度：モジュール内の協調に関する指標



凝集度:論理的凝集

論理的凝集:フラグによる分岐、内部処理を把握する必要がある

```
function 論理的凝集(flag: bool) {
    if (flag) {
        hoge()
    } else {
        fuga()
    }
}
```

凝集度：時間的凝集

時間的凝集：順序は関係なく近い時間の処理をまとめたもの

```
function 時間的凝集(): Config, DB, Logger {
    config = loadConfig()
    db = connectDB()
    logger = getLogger()
    return config, db, logger
}
```

凝集度：機能的凝集

機能的凝集：単一の機能に関する処理

```
function 機能的凝集(p1, p2: Point): number {
    dx = p2.x - p1.x
    dy = p2.y - p1.y
    return Math.sqrt(dx * dx + dy * dy)
}
```

例：論理的凝集を時間的凝集に

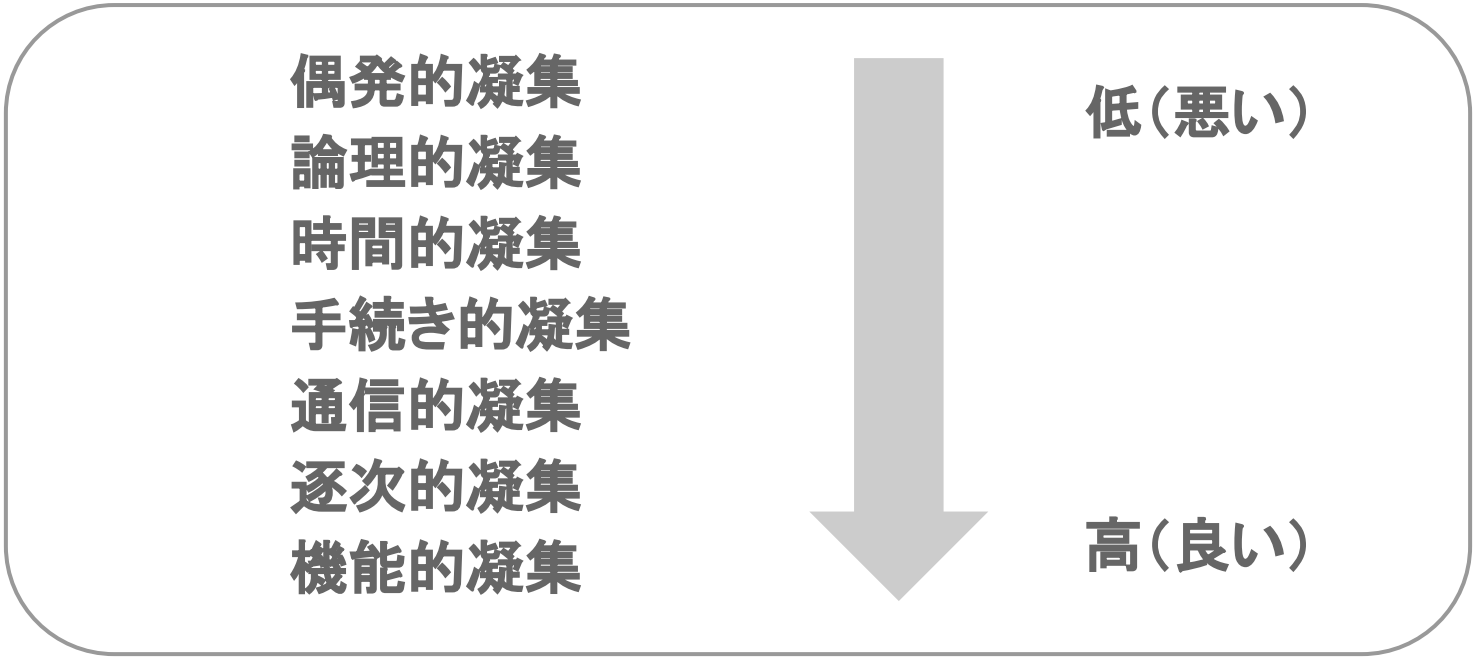
```
function sampleAB(isA: bool) {
    a()
    b()
    if (isHoge) {
        c()
    }
    d()
}
```



```
function sampleA() {
    a()
    b()
    c()
    d()
}

function sampleB() {
    a()
    b()
    d()
}
```

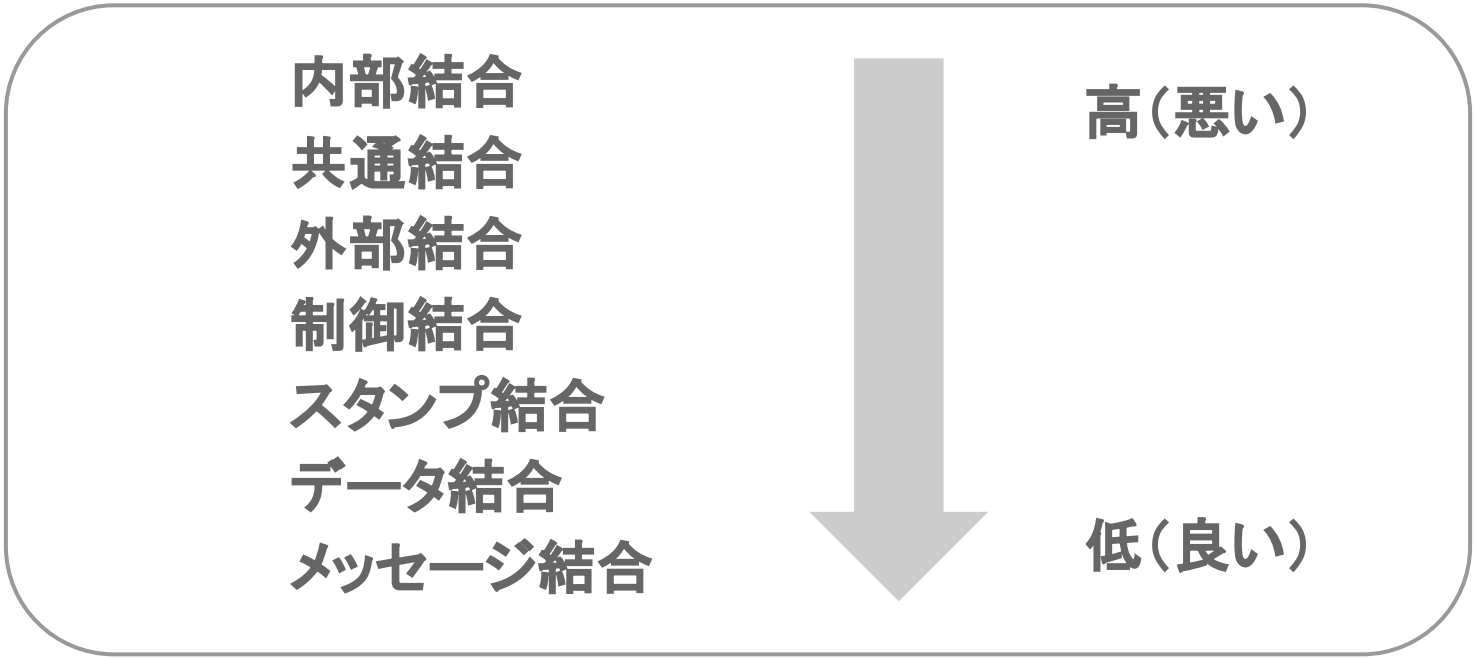
凝集度（再掲）



結合度

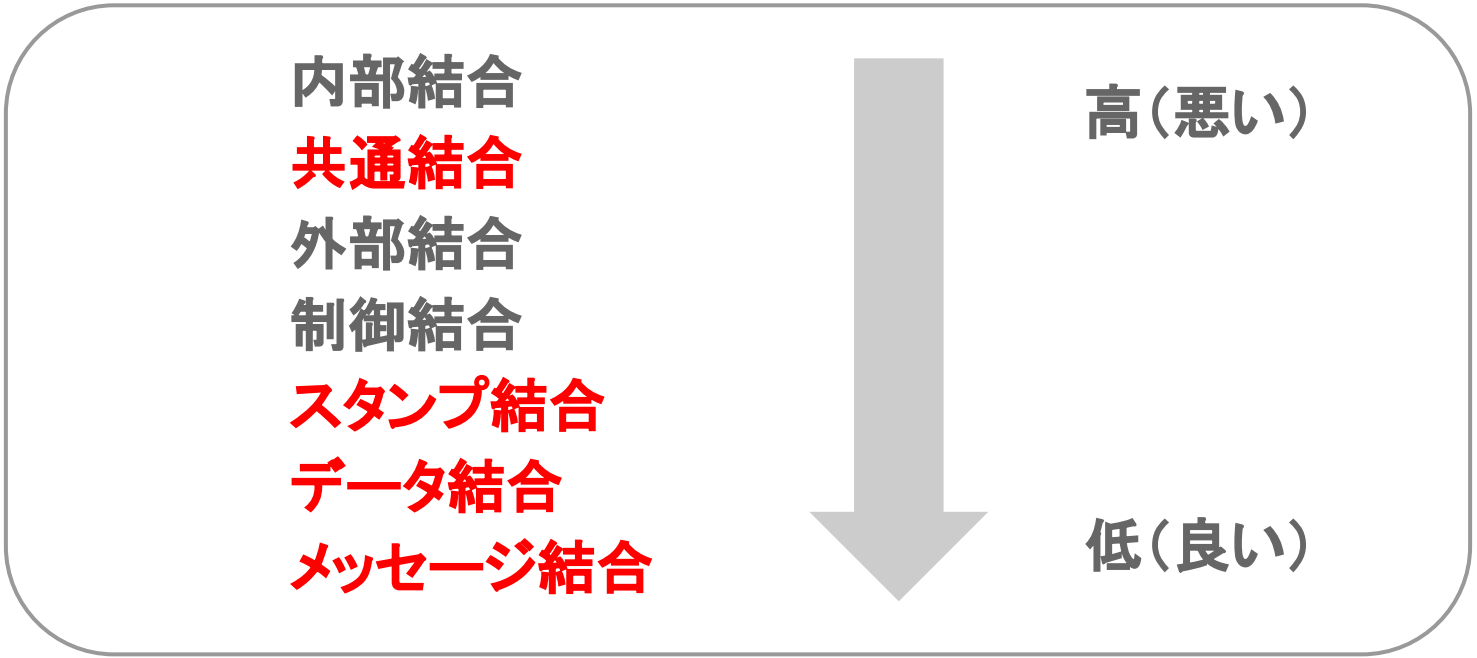
結合度

結合度：モジュール間の連携に関する指標



結合度

結合度：モジュール間の連携に関する指標



結合度:共通結合

共通結合:グローバルなデータを共有する結合

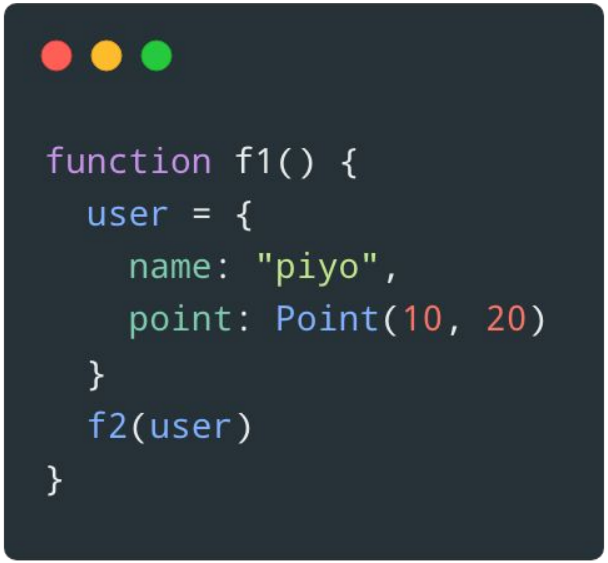
```
point = getPoint()

function moveRight() {
  if (point.x == limit) throw new Error('これ以上移動できない!');
  point.x++
  loadData(point)
}

function warp(dx: number) {
  point.x += dx
}
```

結合度:スタンプ結合

スタンプ結合:構造体を共有する結合



```
function f1() {
  user = {
    name: "piyo",
    point: Point(10, 20)
  }
  f2(user)
}
```

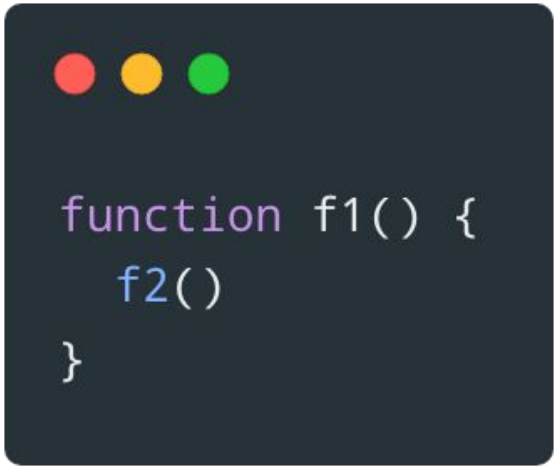
結合度:データ結合

データ結合:必要な要素のみを共有する結合

```
function f1() {
  f2(10, "123")
}
```

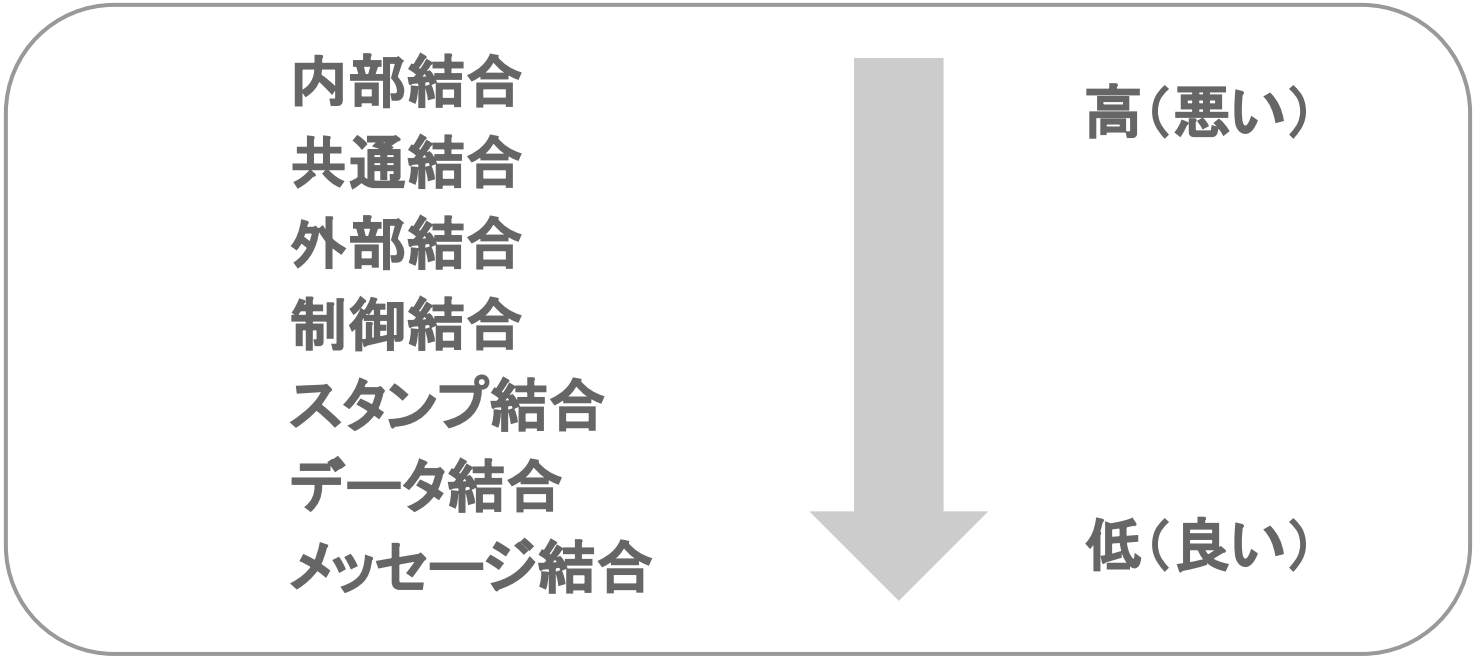
結合度：メッセージ結合

メッセージ結合：タイミングのみ通知する結合



```
function f1() {
  f2()
}
```

結合度（再掲）



SOLID 原則

SOLID 原則

2000年に発表されたレポート
「Design Principles and Design Patterns」
の中で発表された様々な原則の一部

XP 提唱者ロバート・C・マーチンが2005年に書いた
「The Principles of OOD(Object Oriented Design)」
という記事にて5つの頭文字から SOLID としてまとめた(らしい)

SRP : 単一責任の原則

Single Responsibility Principle

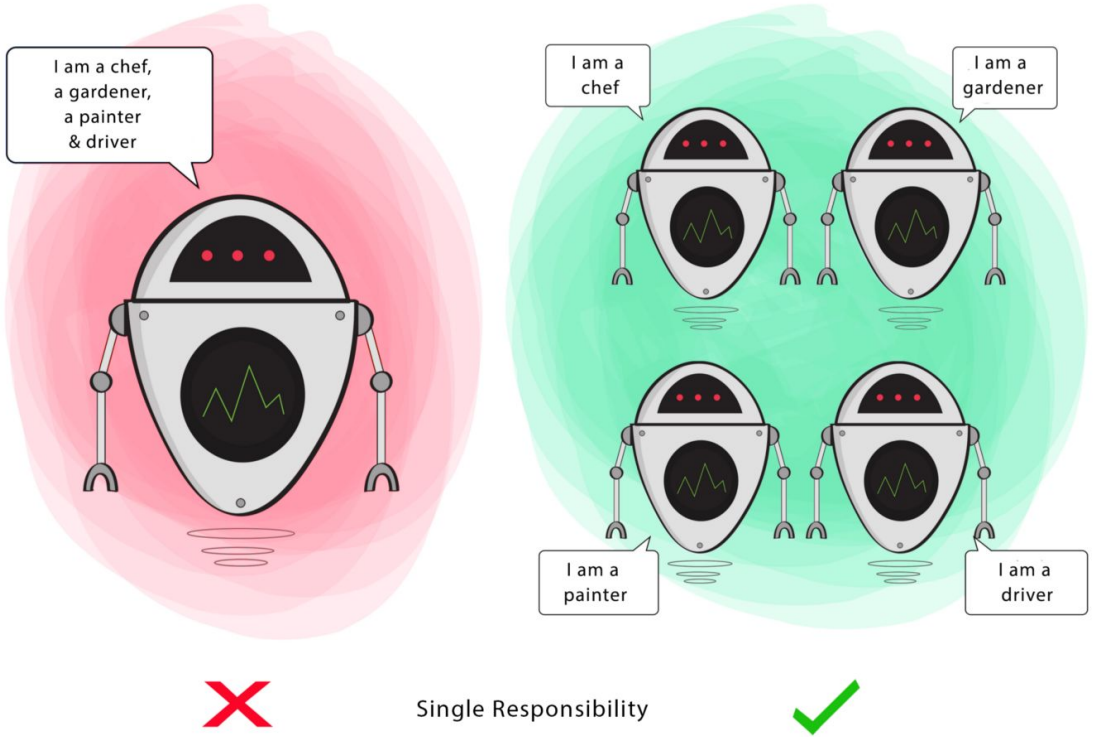
オブジェクトはたったひとつのアクターに対して

責務を負うべきである

→ 立場の違うAとBが同じモジュールを使うと

Bへの変更でAも影響受ける

SRP：単一責任の原則



https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

OCP : オープン・クローズドの原則

Open Closed Principle

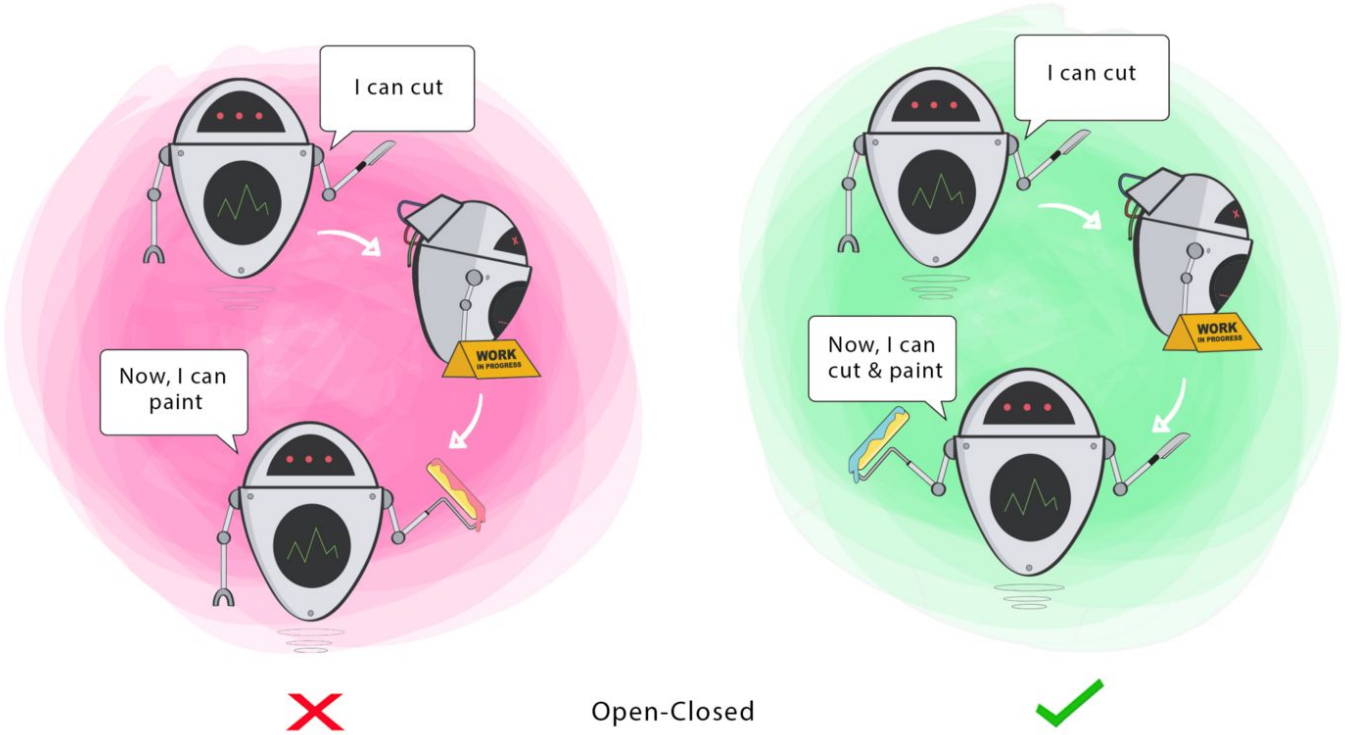
ソフトウェアの構成要素は拡張に対しては開いていて

修正に対して閉じていなければならない

→ ソフトウェアの振る舞いは

既存の成果物を変更せず拡張できるようにすべき

OCP : オープン・クローズドの原則



https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

LSP:リスコフの置換原則

Liskov Substitution Principle

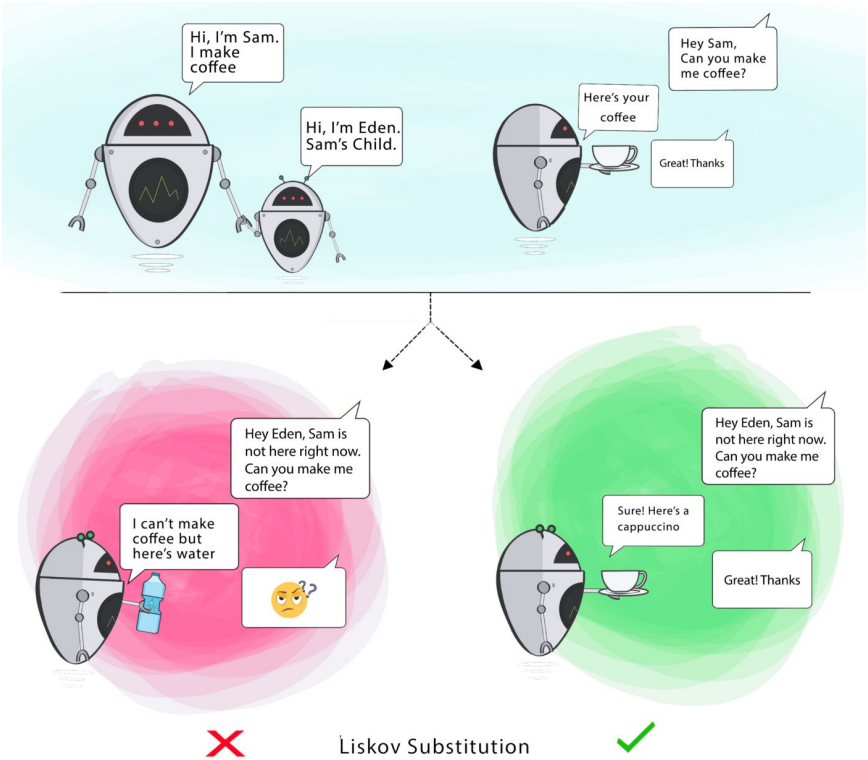
S が T の派生型であれば

T 型のオブジェクトが使われている箇所は全て

S 型のオブジェクトで置換可能である

LSPはアーキテクチャのレベルにも適用できるし、適用すべき

LSP:リスコフの置換原則



https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

ISP：インターフェイス分離の原則

Interface Segregation Principle

インターフェイスを分離し、最小限の依存ですむようにせよ

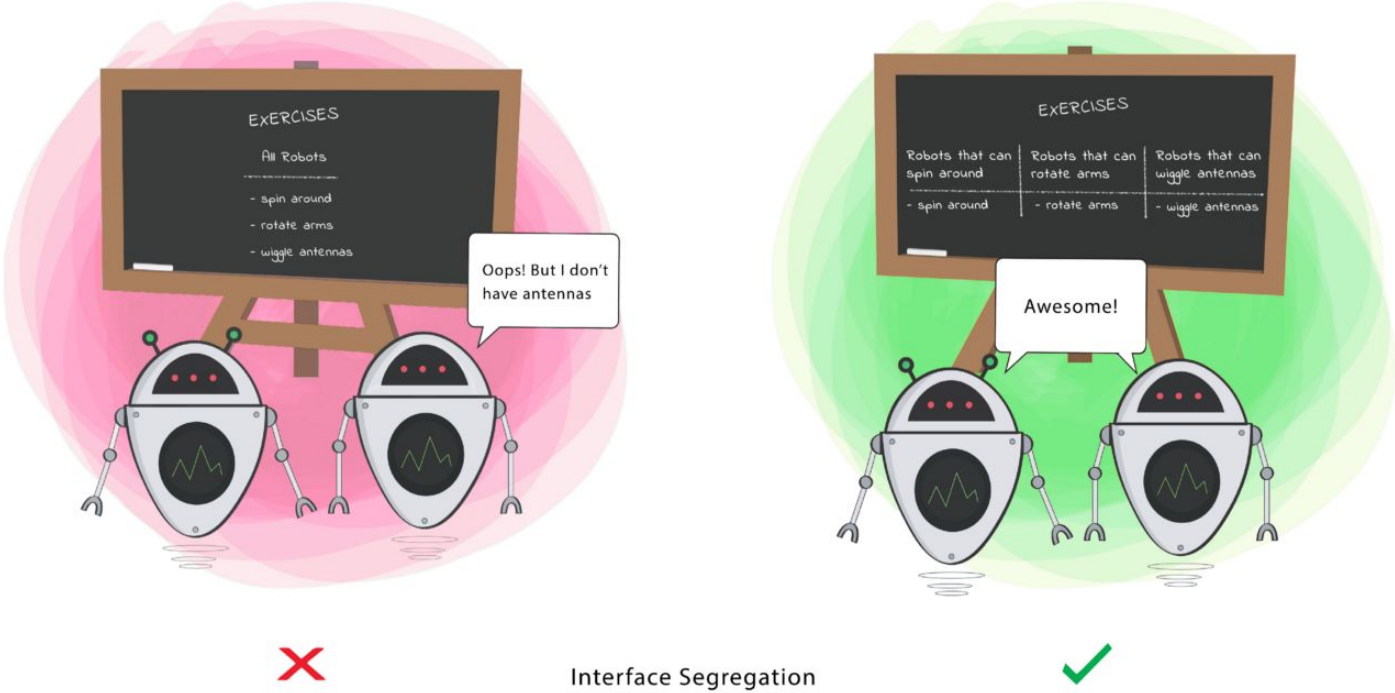
アーキテクチャレベルにも適用できる

システムS → フレームワークF → データベースD

のような依存の場合

Dの変更（障害）はSの変更（障害）となる

ISP：インターフェイス分離の原則



https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

DIP：依存関係逆転の原則

Dependency Inversion Principle

変化しやすい具象要素には依存したくない

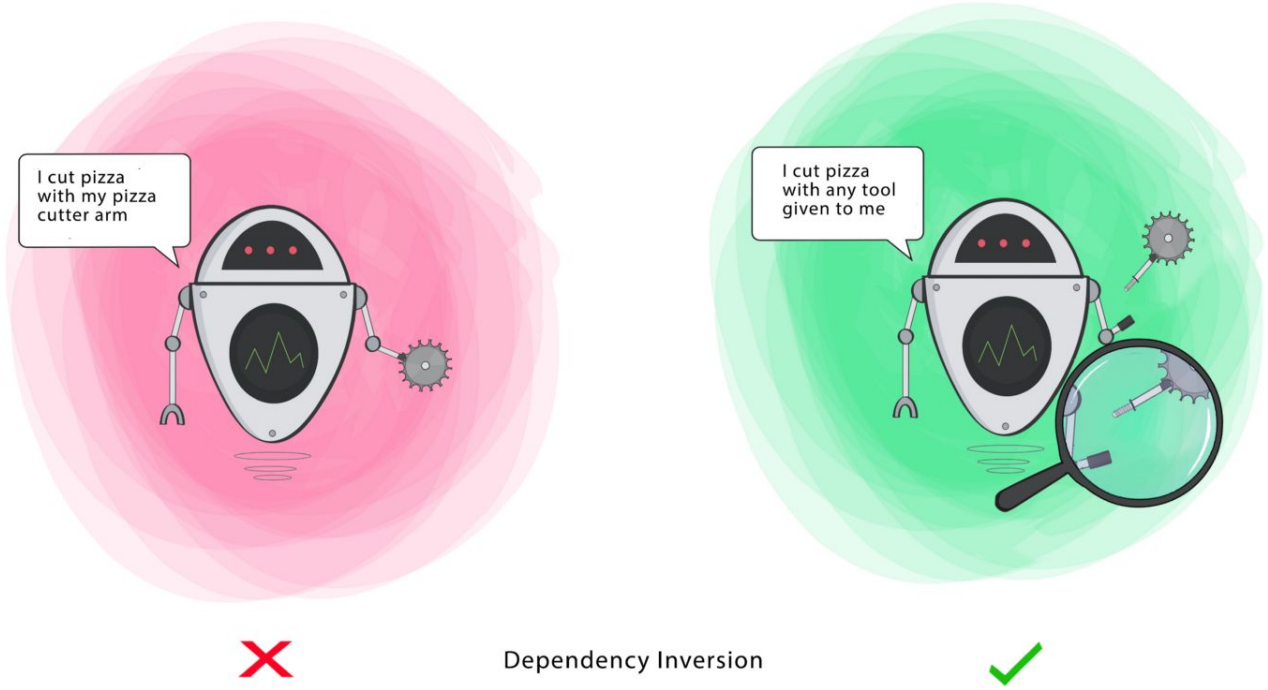
（String など変化しにくいものは OK）

そのようなものはインタフェースなどの抽象宣言にしたい

ただし誰が具象クラスを生成するかという問題がある

大抵のOO言語では Abstract Factory パターンを使う

DIP：依存関係逆転の原則



https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

設計のパターン紹介

設計のパターン

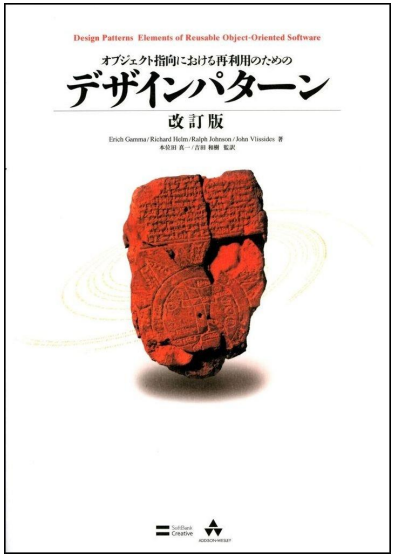
設計を行うと、似たようなパターンが見えてくる事がある

→ デザインパターン

時代によっても流行り廃りがある

アーキテクチャに関して

特に有名なやつを少し紹介



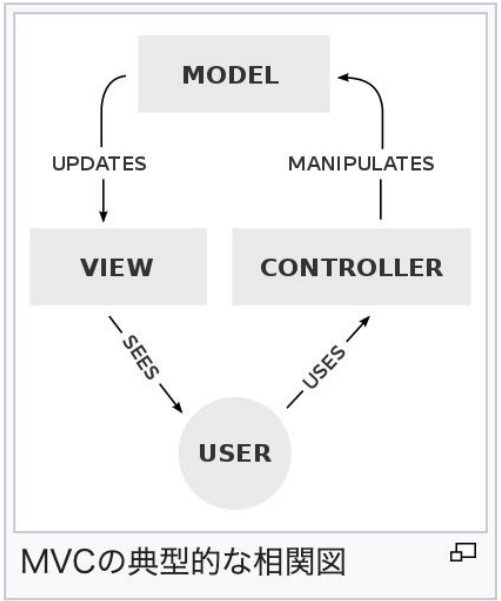
MVC

MVC

UI に注目したデザインパターン
（設計の原則では入出力に関しては言及してこなかった）

もともと SmallTalk におけるウィンドウプログラムの
設計のために提唱された（1980 年代の話）

機能でコンポーネントを3つに分類しようという発想



MVCの典型的な相関図

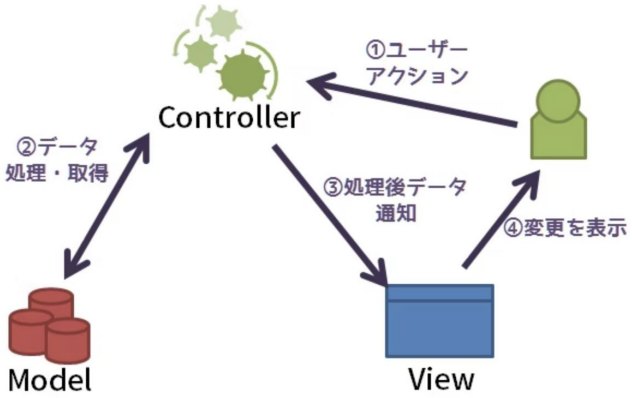
https://ja.wikipedia.org/wiki/Model_View_Controller

MVC2

UI に注目したデザインパターン
（Web 版）

サーバーとクライアントに分かれたため
Model と View が直接つながらなくなった

Web フレームワークを使った開発など
だいたいこれ



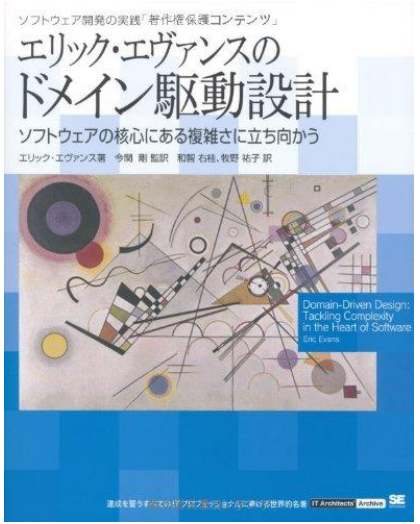
https://qiita.com/shinkuFencer/items/f2651073fb71416b6cd7

DDD

DDD

ドメインモデルに注目したデザインパターン
（パターンというより開発全体の方法論）

モデリングから劇的な利益を得られたプロジェクトを例に、パターンを抽出する



- ドメインの中核となる複雑さと機会に焦点を当てる
- ドメイン専門家とソフトウェア専門家のコラボレーションでモデルを探求する
- 明示的にそれらのモデルを表現するソフトウェアを書く
- 境界付けられたコンテキストの中のユビキタス言語で話す

戦術的 DDD

DDD におけるモデル表現のためのパターン

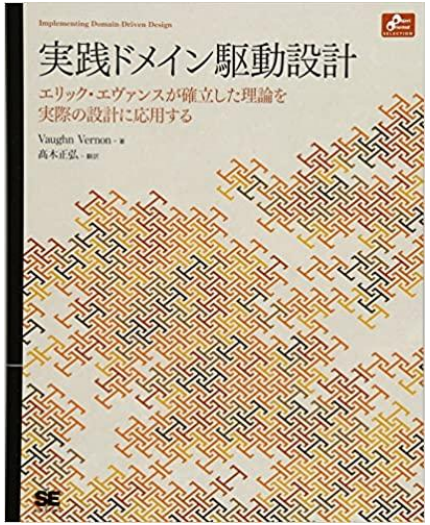
DDD はモデルをコードに落とし込むことを目的とする
それをサポートするパターンがいくつかある

- Entity
- Value Object

} オブジェクトとして表現するモデル

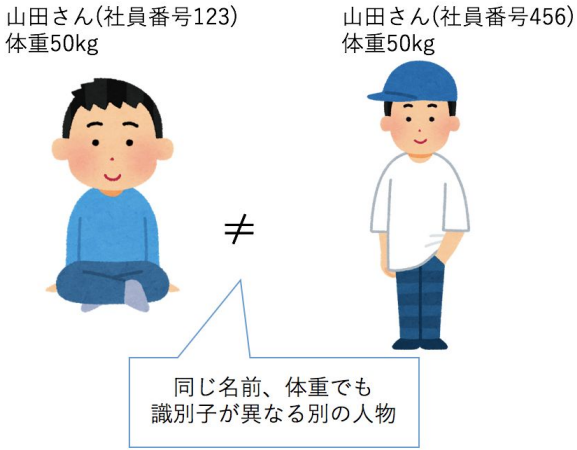
- Domain Service
- Domain Event

} オブジェクトとして表現しないモデル



戦術的 DDD

	Entity	ValueObject
同一性判定	識別子が同一であれば同一	保持する属性が全て同一であれば同一
可変性	可変 生成されてから、変異するというライフスパンを持つ	不変 生成されたら、あとは破棄されるのみ



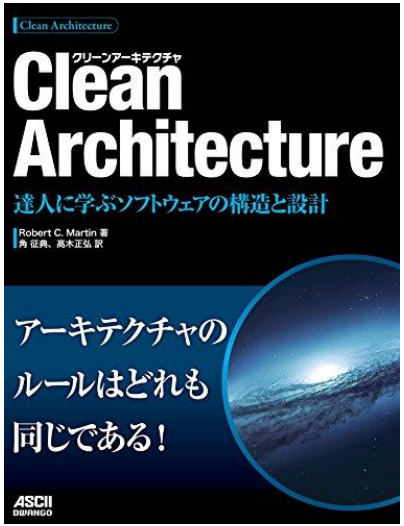
https://little-hands.hatenablog.com/entry/2018/12/09/entity-value-object

CleanArchitecture

CleanArchitecture

アーキテクチャのルールに注目したデザインパターン
（これまでに提唱された様々な設計パターンから共通の考え方を抽出する試み）

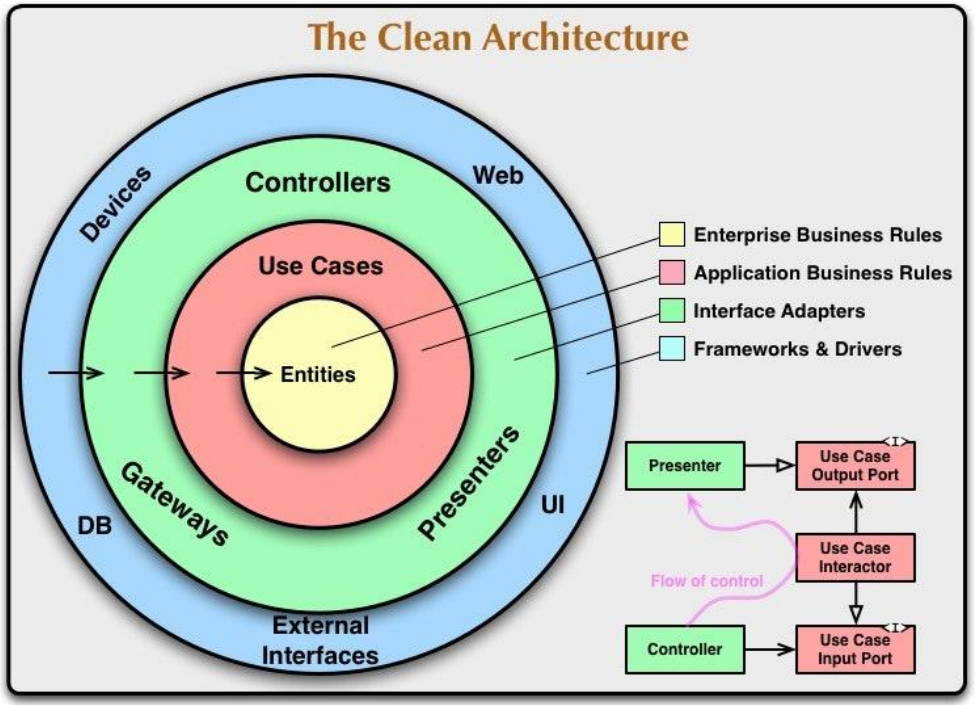
> アーキテクチャのルールはどれも同じである！
>
> — *Robert C. martin*



・関心事でレイヤーを分ける（境界を引く）
・レイヤー間の依存性は一方向にする

この２点のルールを守れば柔軟なシステムができるとの主張

CleanArchitecture



「レイヤーに分けて、より重要な方向に依存させよう」の例

まとめ

まとめ

ソフトウェアアーキテクチャに正解はない

トレードオフの理解には知識が必要

常に学び、常に実践し、そして常に設計しよう