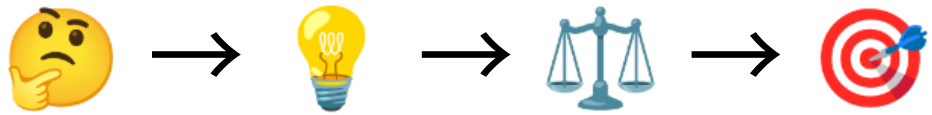
YAGNI の原則

より良いコードを書くための思考法

今日のお話



1. エンジニアリングにおける原則とは
2. YAGNI の原則とは？
3. メリット・デメリット
4. 具体例

1. エンジニアリングにおける原則とは

🧭 開発を導く羅針盤

- **DRY**: Don't Repeat Yourself
- **KISS**: Keep It Simple, Stupid
- **SOLID**: オブジェクト指向設計の 5 原則
- **YAGNI**: You Aren't Gonna Need It

原則 = 迷った時の判断基準

2. YAGNI の原則とは？

今必要ではない機能は実装しない

XP（eXtreme Programming）の基本原則

- 「将来使うかも」への過剰投資を避ける
- **現在の要件**にフォーカス
- シンプルさを保つ

3. メリット・デメリット

✅ YAGNI を守る

- **開発速度**: 2 倍速い
- **バグ数**: 1/4 に減少
- **保守性**: 高い
- **理解しやすさ**: ★★★★★

❌ YAGNI を破る

- **開発速度**: 半分以下
- **バグ数**: 4 倍増加
- **保守性**: 低い
- **理解しやすさ**: ★★

よくある誤解

「後で追加するのは大変では？」

実は70%の「将来必要」な機能は使われない

変更コスト < 無駄な実装コスト

4. 具体例：API 設計

❌ YAGNI 違反

```
class UserAPI {
  getUser(id) {/* 今使う */}
  createUser(data) {/* 今使う */}
  updateUser(id, data) {/* 今使う */}
  deleteUser(id) {/* 将来使うかも */}
  getUserStats(id) {/* 将来使うかも */}
  exportUserData(id) {/* 将来使うかも */}
  importUserData(data) {/* 将来使うかも */}
}
```

結果: 開発期間 2 倍、バグ 12 個

4. 具体例：API 設計

✅ YAGNI 適用

```
interface UserAPI {
  getUser(id string) {/* 実装 */}
  createUser(data string) {/* 実装 */}
  updateUser(id string, data string) {/* 実装 */}
  // 必要になったら追加
}
```

開発期間予定通り、バグ3個

リアルケーススタディ

EC サイト開発での比較

項目	YAGNI 適用	YAGNI 違反
開発期間	3 ヶ月	8 ヶ月
実装機能数	12 個	35 個
実際に使用	12 個（100%）	7 個（20%）
ユーザー満足度	85%	45%

YAGNI の実践ポイント

1. 要件を明確化

「本当に今必要？」を常に問いかける

2. 段階的開発

MVP → フィードバック → 機能追加

3. 削除する勇気

使われない機能は削除する

まとめ

シンプルさは最高の洗練

- **今必要な機能**だけに集中
- **70%の予想**は外れる
- **変更は怖くない**、むしろ健全

YAGNI で開発効率 2 倍アップ！

Let's YAGNI!

質問・ディスカッション