# 設計原則、アーキテクチャパターン、アーキテクチャスタイルの違いって何？いつどう向き合ったらいいの？を考えてみる

PHP カンファレンス名古屋 2025　Feb 22, 2025.

v0.0.3

@katzumi(かつみ)

Press Space for next page →

# 自己紹介

katzumi（かつみ）と申します。

「障害のない社会をつくる」をビジョンに掲げている「LITALICO」という会社に所属しています

LITALICO

以下のアカウントで活動しています。

X katzchum

k2tzumi / katzumi

# お願い

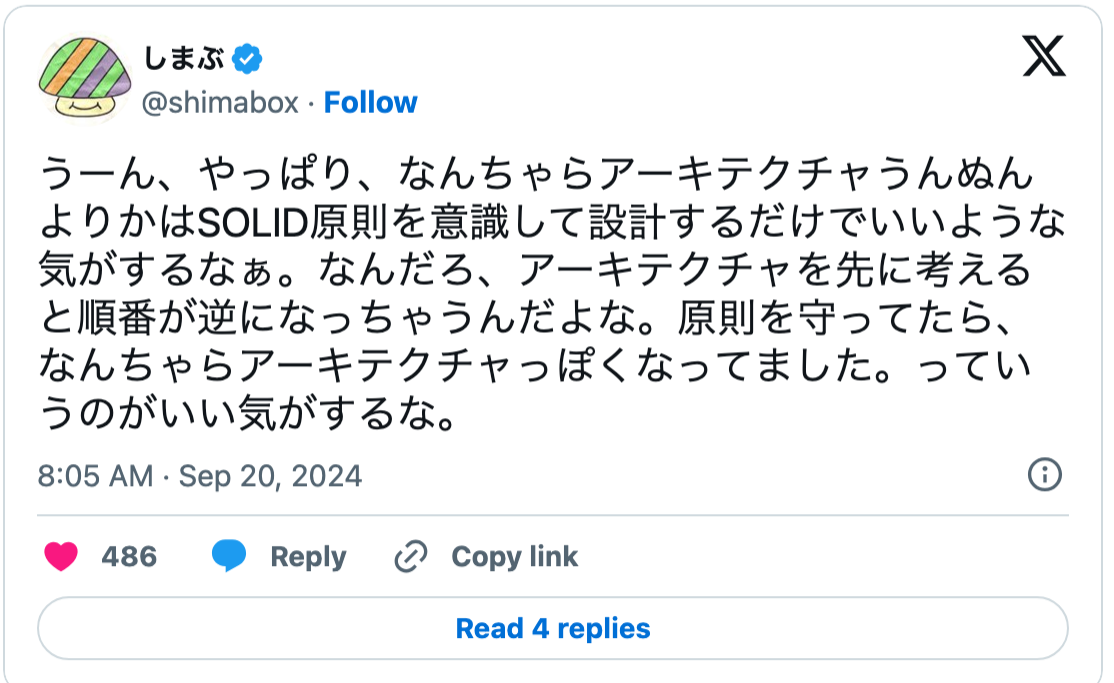
写真撮影、SNS での実況について

登壇者の励みになるので是非ともご意見やご感想など、フィードバック頂けると助かります mm
あとでスライドを公開します

#phpcon_nagoya #s

# 開発者が直面する「設計に関する情報過多」

この Post を見て「それな」と思ったのが、本セッションをしようと思い立ったきっかけになっています



# 今日お話すること・話さないこと

狙いは「設計概念を体系的に理解し、個々のパターンを把握するための助けとする」です

## 🚫セッションでは扱わないこと

- 個々の設計パターンの詳細な解説
- 具体的な実装方法やコード例の紹介
- 特定のアーキテクチャの採用判断基準

**個別の設計パターンやアーキテクチャの詳細は、巻末の参考資料参照📚**

## ✨セッションで得られること

- 設計に関する概念を体系的に理解する視点
- 各設計概念の位置づけと相互の関係性
- 設計概念の発展背景と解決してきた課題の理解

# 見慣れた風景

設計に関する用語が飛び交う日々...

- "このプロジェクトは Clean Architecture で..."
- "ここは ServiceLayer パターンを使って..."
- "SOLID の原則に従って設計しましょう"
- "マイクロサービスアーキテクチャに移行して..."

ドキュメントやブログで目にする
アーキテクチャやパターンの数々

# 開発者の本音

分かった気になっているけど...

- 具体的に何をすればいいの？
- この規模のプロジェクトに必要？
- いつ何を考えればいいの？
  - 設計の検討は早すぎ？遅すぎ？
  - どの段階で決めるべき？
- 既存のコードにどう適用する？
- トレードオフは何だろう？

# 開発者の本音

分かった気になっているけど...

- 具体的に何をすればいいの？
- この規模のプロジェクトに必要？
- いつ何を考えればいいの？
  - 設計の検討は早すぎ？遅すぎ？
  - どの段階で決めるべき？
- 既存のコードにどう適用する？
- トレードオフは何だろう？

**知識はあるのに、実践での判断に迷う**

ぜんぜんわからない

# 俺達は雰囲気で（ry

# 「雰囲気」の真意

「雰囲気で設計」の現実

- ソフトウェア開発に正解はない
  - 要件は常に変化する
  - チームごとに異なる制約がある
  - ビジネスの不確実性と向き合う
- 時には直感や経験に基づく判断も必要
  - 完璧な情報収集は現実的でない
  - スピードと質のバランス
  - チームの力量と成長

# 基礎となる原則を理解する

なぜ体系的な理解が必要なのか？

- 個別に学習・実践するには難易度が高すぎる
  - 新しい設計手法や考え方を次々と学ぶが、実践する機会がない
  - 表面的な理解に留まり、本質的な価値が掴めていない
  - 理想的な設計パターンやアーキテクチャの例を見ても、既存のプロジェクトにどう適用するか分からない
  - 様々な設計手法の中から、どれを選択すべきか判断できない
  - 各設計手法が生まれた背景や解決しようとした問題の理解が浅い
- 概念は互いに関連しており、体系的に理解することで、より効果的に活用することができる
- 概念の適用範囲や限界を理解し、状況に応じて適切な選択を行う為

# 基礎となる原則を理解する

なぜ体系的な理解が必要なのか？

- 個別に学習・実践するには難易度が高すぎる
  - 新しい設計手法や考え方を次々と学ぶが、実践する機会がない
  - 表面的な理解に留まり、本質的な価値が掴めていない
  - 理想的な設計パターンやアーキテクチャの例を見ても、既存のプロジェクトにどう適用するか分からない
  - 様々な設計手法の中から、どれを選択すべきか判断できない
  - 各設計手法が生まれた背景や解決しようとした問題の理解が浅い
- 概念は互いに関連しており、体系的に理解することで、より効果的に活用することができる
- 概念の適用範囲や限界を理解し、状況に応じて適切な選択を行う為

**近道はない**

＿人人人人人＿
＞　救世主　＜
￣Y^Y^Y^Y^Y^￣


# 「TECHNICAL MASTER はじめての PHP エンジニア入門編」

2024 年 12 月発刊！

豪華執筆陣！

# 「TECHNICAL MASTER はじめてのPHP エンジニア入門編」のおすすめ章

本セッションとも関連が高くて気に入るはずです

Chapter09: 設計原則とパターン

> *この章では、ソフトウェア開発における設計原則の重要性とアーキテクチャパターンの具体的な適用方法について詳しく解説します。*
>
> *まず、アーキテクチャの定義とその必要性を説明し、SOLID原則を通じて設計の基本概念を理解します。その後、さまざまなアーキテクチャパターン（MVC、 レイヤードアーキテクチャ、クリーンアーキテクチャ、ヘキサゴナルアーキテクチャ）を紹介します。*
>
> *また、設計パターンとアンチパターンをとりあげ、実際の開発における適用例や回避すべき設計の落とし穴についても解説します。*

# 超絶オススメなので是非！

きちんと学びたい＝学び直しにも

# 設計概念の階層構造

# 設計に関する用語の3つのカテゴリ

数多の用語がどういうカテゴリになるのか？

- ○○の原則・法則

  SOLID, KISS, YAGNI, DRY, コマンドクエリ分離原則(CQS) , デメテルの法則（最小知識の原則）,驚き最小の原則 [1]

  → 設計の原則
- ○○パターン

  MVC, ActiveRecord, Value Object, Server Session State, Domain Model, CQRS

  → アーキテクチャパターン
- ○○アーキテクチャ

  Layered, Clean, Hexagonal, Event Driven, Pipeline, Service Oriented

  → アーキテクチャスタイル

---

1. UNIX哲学 "Basics of the Unix Philosophy" の基本原則の一つ ↩

# 設計原則

SOLID, DRY 等

- 設計における普遍的な指針
- 個々の判断の基準となる考え方
- 文脈に依存しない基礎的なルール

# 設計原則

SOLID, DRY 等

- 設計における普遍的な指針
- 個々の判断の基準となる考え方
- 文脈に依存しない基礎的なルール

**最も具象的で個々のクラスやメソッドレベルの直接的な設計指針となる**

# アーキテクチャパターン

MVC, ActiveRecord 等

- 特定の問題に対する定型的な解決策
- 複数の原則を組み合わせた実践的な手法
- 文脈に応じて選択・適用する

# アーキテクチャパターン

MVC, ActiveRecord 等

- 特定の問題に対する定型的な解決策
- 複数の原則を組み合わせた実践的な手法
- 文脈に応じて選択・適用する

**抽象度が中程度なモジュールやコンポーネントレベルの設計パターン**

# デザインパターンとの違い

パターンにも色々ある

| 項目 | デザインパターン（GoF） | アーキテクチャパターン |
| --- | --- | --- |
| 設計レベル | クラス/オブジェクトの設計レベル | システム構造レベル |
| パターン数 | 23個のパターンで完結 | POSA [1]、PoEAA [2]、EIPで体系化 [3] |
| 適用範囲 | 言語非依存の解決策 | エンタープライズアプリケーション特有 |

1. Pattern-Oriented Software Architecture ↩
2. Patterns of Enterprise Application Architecture ↩
3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions ↩

# アーキテクチャスタイル

Layered, Event-Driven, Microservices 等

- システム全体の構造を定義する設計思想
- 複数のパターンを包含する包括的な方針
- ビジネス要件と技術要件を橋渡しする

# アーキテクチャスタイル

Layered, Event-Driven, Microservices 等

- システム全体の構造を定義する設計思想
- 複数のパターンを包含する包括的な方針
- ビジネス要件と技術要件を橋渡しする

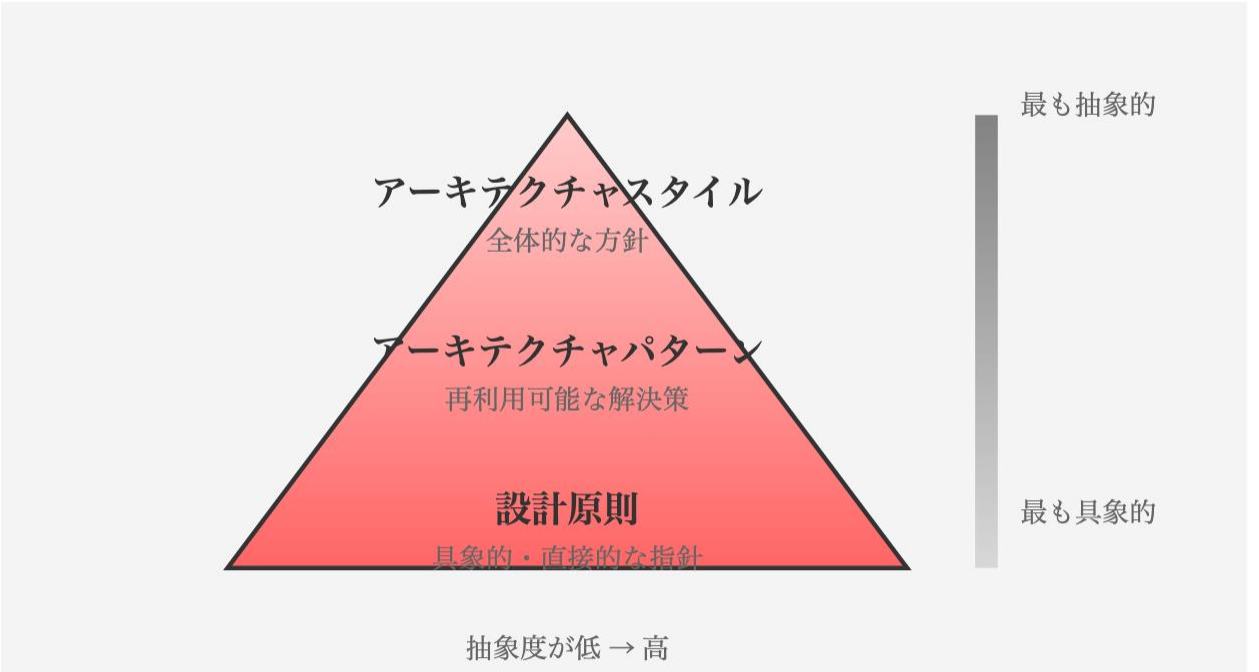
**最も抽象度が高く、システム全体の構造を決定する**

# 設計に関する3つの概念カテゴリ比較

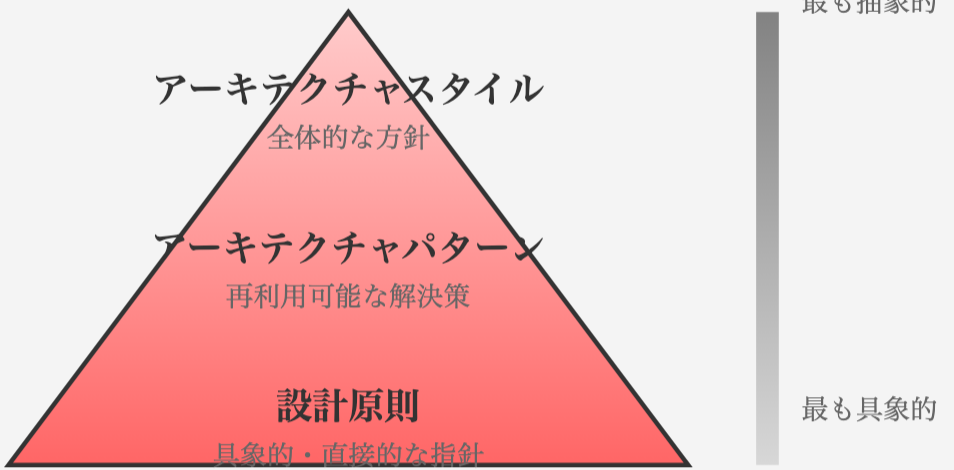
抽象度が低→高、影響度も小→大となる

| 項目 | 適用範囲 | 抽象度 | 決定の影響 |
| --- | --- | --- | --- |
| 設計原則 | メソッド/クラスレベルがメイン | 具象的（直接的な指針） | 局所的な影響 |
| アーキテクチャパターン | モジュール/コンポーネントレベル | 中間（再利用可能な解決策） | 中規模な影響 |
| アーキテクチャスタイル | システム/アプリケーションレベル | 抽象的（全体的な方針） | 全体的な影響 |

# 抽象度レベルピラミッド

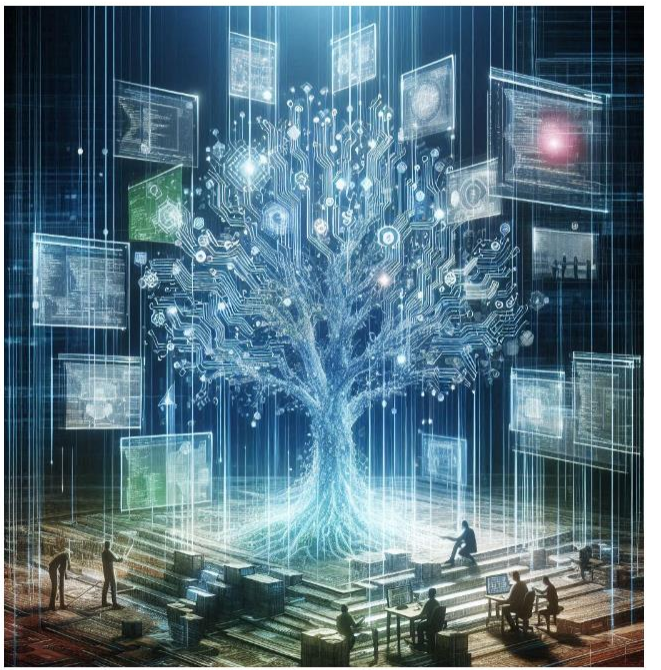


# 抽象度レベルピラミッド




**抽象度の上昇に伴い、影響範囲が広がっていく**
**スタイルは変更コストが高く、決定の重要度も高い**

# 設計課題の解決領域



# 設計原則の役割

ソフトウェア開発の品質向上と効率化を目指す

## 解決しようとした課題

- コードの重複等による保守性の低下
- 変更の影響範囲が予測できない
- テストが困難
- チーム間での品質にばらつきがある

## 目指すゴール

- 一貫した設計基準を提供し、コードの品質を均一に保つ
- 保守性を向上させ、長期的な維持管理を容易にする
- 技術的負債を削減する
- チーム内のコミュニケーションを円滑にする

# 設計原則の役割

ソフトウェア開発の品質向上と効率化を目指す

## 解決しようとした課題

1. コードの複雑性
   大規模なソフトウェアシステムの複雑さを管理し、理解しやすく保守しやすいコードを作成する。
2. 変更への対応
   ビジネス要件の変更や技術の進歩に柔軟に対応できるシステムを設計する。
3. 品質の向上
   バグの少ない、信頼性の高いソフトウェアを開発するための指針が求められた。
4. 開発効率の改善
   開発時間の短縮と、チーム間のコミュニケーション改善が必要。
5. 再利用性の促進
   コードの再利用を促進し、開発コストを削減する必要。

## 主な目的

1. コードの保守性向上
2. 再利用性の促進
3. 拡張性の確保
4. バグの減少
5. 開発効率の向上

# 設計原則の役割

ソフトウェア開発の品質向上と効率化を目指す

## 解決しようとした課題

1. コードの複雑性
   大規模なソフトウェアシステムの複雑さを管理し、理解しやすく保守しやすいコードを作成する。
2. 変更への対応
   ビジネス要件の変更や技術の進歩に柔軟に対応できるシステムを設計する。
3. 品質の向上
   バグの少ない、信頼性の高いソフトウェアを開発するための指針が求められた。
4. 開発効率の改善
   開発時間の短縮と、チーム間のコミュニケーション改善が必要。
5. 再利用性の促進
   コードの再利用を促進し、開発コストを削減する必要。

## 主な目的

1. コードの保守性向上
2. 再利用性の促進
3. 拡張性の確保
4. バグの減少
5. 開発効率の向上

**変更に強く理解しやすいコードにする**
**変更容易性と理解容易性を高くする**

# 設計原則による課題解決マトリクス

SOLID 原則での課題解決事例

| 課題 | 単一責任原則（SRP） | オープン/クローズド原則（OCP） | リスコフの置換原則（LSP） | インターフェース分離原則（ISP） | 依存性逆転原則（DIP） |
|---|---|---|---|---|---|
| コードの複雑性 | クラスの目的を明確化 | 継承やポリモーフィズムを活用 | 一貫した振る舞いの提供 | 必要な機能だけを提供 | 抽象化と依存関係の最小化 |
| 変更への対応 | 変更理由の一つに限定 | 新機能追加時に既存コードを変更しない | クライアントコードの変更を最小限に | インターフェースの分離 | 依存関係の逆転による柔軟性 |
| 品質向上 | 高凝集を実現 | 変更影響範囲の最小化 | クライアントコードとの結合を維持 | 高凝集のインターフェース | モジュール間の独立性向上 |

# アーキテクチャパターンの役割

複雑化するエンタープライズアプリケーションの設計と統合における共通課題に対処するための体系化

## 解決しようとした課題

- 似たような問題を何度も解決している
- 設計ノウハウが共有されない
- 実装方法の選択に時間がかかる
- チーム間で実装方法がバラバラ

## 目指すゴール

- 再利用可能な設計手法を提供し、効率的な開発を実現する
- 設計の標準化と一貫性を確保する
- チーム間のコミュニケーションを促進する

# アーキテクチャパターンの役割

複雑化するエンタープライズアプリケーションの設計と統合における共通課題に対処するための体系化

## 解決しようとした課題

1. 複雑性の管理
   大規模エンタープライズアプリケーションの複雑さを管理し、理解しやすく保守しやすいシステムを設計する必要。
2. 再利用性の向上
   共通のパターンを識別し、再利用可能なソリューションを提供することで、開発効率を向上させる必要。
3. システム統合の複雑さ
   異なるシステム間の統合が複雑化し、標準化されたアプローチが必要。
4. スケーラビリティとパフォーマンス
   増大するデータ量とユーザー数に対応できる、スケーラブルで高性能なシステムの設計が求められた。
5. 分散システムの課題
   分散システムにおける一貫性、可用性、パーティション耐性（CAP 定理）などの課題に対処する必要。
6. 技術の進化への対応
   急速に進化する技術環境に適応できる柔軟なアーキテクチャの設計が必要。

## 主な目的

- PoEAA（Patterns of Enterprise Application Architecture）
  1. エンタープライズアプリケーションの設計に関する知識の体系化
  2. 複雑なビジネスロジックとデータ処理の効率的な実装方法の提供
  3. スケーラブルで保守性の高いアプリケーション構築のための指針提供
- EIP（Enterprise Integration Patterns）
  1. 分散システム間の統合パターンの標準化
  2. メッセージングシステムを使用した効果的な統合方法の提供
  3. 複雑な統合シナリオに対する共通語彙の確立

# アーキテクチャパターンの役割

複雑化するエンタープライズアプリケーションの設計と統合における共通課題に対処するための体系化

## 解決しようとした課題

1. 複雑性の管理
   大規模エンタープライズアプリケーションの複雑さを管理し、理解しやすく保守しやすいシステムを設計する必要。
2. 再利用性の向上
   共通のパターンを識別し、再利用可能なソリューションを提供することで、開発効率を向上させる必要。
3. システム統合の複雑さ
   異なるシステム間の統合が複雑化し、標準化されたアプローチが必要。
4. スケーラビリティとパフォーマンス
   増大するデータ量とユーザー数に対応できる、スケーラブルで高性能なシステムの設計が求められた。
5. 分散システムの課題
   分散システムにおける一貫性、可用性、パーティション耐性（CAP 定理）などの課題に対処する必要。
6. 技術の進化への対応
   急速に進化する技術環境に適応できる柔軟なアーキテクチャの設計が必要。

## 主な目的

- PoEAA（Patterns of Enterprise Application Architecture）
  1. エンタープライズアプリケーションの設計に関する知識の体系化
  2. 複雑なビジネスロジックとデータ処理の効率的な実装方法の提供
  3. スケーラブルで保守性の高いアプリケーション構築のための指針提供
- EIP（Enterprise Integration Patterns）
  1. 分散システム間の統合パターンの標準化
  2. メッセージングシステムを使用した効果的な統合方法の提供
  3. 複雑な統合シナリオに対する共通語彙の確立

**共通言語としての機能**
**設計意図の明確な伝達**

# アーキテクチャスタイルの役割

ソフトウェアシステムの複雑化と多様化が大きく影響して発展

## 解決しようとした課題

- システム全体の一貫性が保てない
- スケーラビリティの要件に対応できない
- チーム間の連携が困難
- 長期的な進化が難しい

## 目指すゴール

- システム全体の構造を決定し、一貫した設計を提供する
- スケーラビリティを確保し、システムの成長に対応する
- チーム間の協調とコミュニケーションを促進する
- 長期的なシステムの進化を支援する

# アーキテクチャスタイルの役割

ソフトウェアシステムの複雑化と多様化が大きく影響して発展

## 解決しようとした課題

1. システム全体の複雑性管理
   大規模システムの全体構造を簡素化し、理解しやすくする必要。
2. スケーラビリティとパフォーマンスの最適化
   システム全体のスケーラビリティを向上させ、大規模なデータ処理や高負荷に対応する必要。
3. 分散システムの課題への対応
   ネットワーク遅延、部分的な障害、データの一貫性など、分散システム特有の問題に対処する必要。
4. 異種システム間の統合
   異なる技術や標準を使用するシステム間の効果的な統合方法が求められた。
5. 変更への適応性
   ビジネス要件や技術の変化に迅速に対応できる柔軟なアーキテクチャが必要。
6. 開発・運用の効率化
   システムの開発、デプロイ、運用を効率化し、継続的なデリバリーを可能にする必要。

## 主な目的

1. システムの全体的な構造を定義
2. コンポーネント間の関係を明確にする
3. アーキテクチャ特性をカバーする
4. 経験豊富なアーキテクト間の共通言語として機能する
5. システムの品質属性（パフォーマンス、スケーラビリティ、セキュリティなど）を達成する
6. コンポーネントや機能の再利用性を向上させる
7. システム内のコンポーネントや外部システムとの統合を容易にする
8. 将来的な変更や拡張に対する柔軟性を確保する
9. パフォーマンス、保守性、信頼性などのシステム品質を設計段階で担保する

# アーキテクチャスタイルの役割

ソフトウェアシステムの複雑化と多様化が大きく影響して発展

## 解決しようとした課題

1. システム全体の複雑性管理
   大規模システムの全体構造を簡素化し、理解しやすくする必要。
2. スケーラビリティとパフォーマンスの最適化
   システム全体のスケーラビリティを向上させ、大規模なデータ処理や高負荷に対応する必要。
3. 分散システムの課題への対応
   ネットワーク遅延、部分的な障害、データの一貫性など、分散システム特有の問題に対処する必要。
4. 異種システム間の統合
   異なる技術や標準を使用するシステム間の効果的な統合方法が求められた。
5. 変更への適応性
   ビジネス要件や技術の変化に迅速に対応できる柔軟なアーキテクチャが必要。
6. 開発・運用の効率化
   システムの開発、デプロイ、運用を効率化し、継続的なデリバリーを可能にする必要。

いっぱいあり、抽象的

## 主な目的

1. システムの全体的な構造を定義
2. コンポーネント間の関係を明確にする
3. アーキテクチャ特性をカバーする
4. 経験豊富なアーキテクト間の共通言語として機能する
5. システムの品質属性（パフォーマンス、スケーラビリティ、セキュリティなど）を達成する
6. コンポーネントや機能の再利用性を向上させる
7. システム内のコンポーネントや外部システムとの統合を容易にする
8. 将来的な変更や拡張に対する柔軟性を確保する
9. パフォーマンス、保守性、信頼性などのシステム品質を設計段階で担保する

# 相互補完の効果

各概念は独立ではなく、相互に補完

| 役割・効果 | 設計原則 | アーキテクチャパターン | アーキテクチャスタイル |
|---|---|---|---|
| 品質の確保 | - コードの品質基準<br>- 変更容易性の確保 | - モジュール構造の一貫性<br>- 再利用性の向上 | - システム全体の整合性<br>- 長期的な保守性 |
| 実装の指針 | - 具体的な実装の判断基準<br>- コードレベルの品質確保 | - コンポーネントの実装方針<br>- インターフェースの定義 | - システム構造の大枠決定<br>- 主要コンポーネントの特定 |
| 変更への対応 | - 局所的な変更の容易さ<br>- テスタビリティの確保 | - モジュール単位の変更管理<br>- 依存関係の制御 | - 大規模な変更の制御<br>- 進化の方向性の提供 |

# 相互補完の効果

各概念は独立ではなく、相互に補完

| 役割・効果 | 設計原則 | アーキテクチャパターン | アーキテクチャスタイル |
|---|---|---|---|
| 品質の確保 | - コードの品質基準<br>- 変更容易性の確保 | - モジュール構造の一貫性<br>- 再利用性の向上 | - システム全体の整合性<br>- 長期的な保守性 |
| 実装の指針 | - 具体的な実装の判断基準<br>- コードレベルの品質確保 | - コンポーネントの実装方針<br>- インターフェースの定義 | - システム構造の大枠決定<br>- 主要コンポーネントの特定 |
| 変更への対応 | - 局所的な変更の容易さ<br>- テスタビリティの確保 | - モジュール単位の変更管理<br>- 依存関係の制御 | - 大規模な変更の制御<br>- 進化の方向性の提供 |

**協調して機能することで、より堅牢で適応性の高いソフトウェア設計が可能**

# ボトムアップの重要性

下位の概念が上位の品質を支える

補完関係により以下を実現

- 一貫性のある設計判断が可能に
- 各レベルでの品質確保
- 変更に強いシステムの実現
- 長期的な保守性の向上

# ボトムアップの重要性

下位の概念が上位の品質を支える

補完関係により以下を実現

- 一貫性のある設計判断が可能に
- 各レベルでの品質確保
- 変更に強いシステムの実現
- 長期的な保守性の向上

**組み合わせにより実用的な設計を実現**

# 設計概念の歴史的発展

なぜこれらの設計概念が生まれて来たのか？

# 1960-1970年代: ハードウェア依存時代

設計原則の萌芽

- 構造化プログラミングの登場
- モジュール化、カプセル化の概念確立
- ソフトウェアの複雑性に対する初期の対応

# 1980-1990年代: パッケージソフトウェアの台頭

原則の体系化とパターンの出現

【設計原則の確立】

- 1988: CRC（Class-Responsibility-Collaboration）
- 1995: SOLID 原則（Robert C. Martin）
- 1999: DRY, YAGNI 等の原則

【デザインパターンの体系化】

- 1987: Kent Beck & Ward Cunningham がパターン言語を提唱
- 1994: GoF 本「Design Patterns」出版
- オブジェクト指向設計の知見を集約

# 2000年代：SaaS/Webサービスの時代

エンタープライズパターンの体系化

【アーキテクチャパターンの整理】

- 1996〜2009: Pattern-Oriented Software Architecture（POSA）シリーズ
  - 広範なシステムアーキテクチャパターン [1]を扱う
- 2002: Patterns of Enterprise Application Architecture (PoEAA)
  - エンタープライズアプリケーションの知見集約 [2]
- 2004: Enterprise Integration Patterns(EIP)
  - システム統合のパターン化

---

1. 基本的なアーキテクチャパターンから始まり、分散コンピューティングからリソース管理、パターン言語まで幅広いトピックをカバーしています ↩

2. アーキテクチャパターンのカタログとしてまとめられています ↩

# 2010年代以降：クラウドネイティブ時代

アーキテクチャスタイルの進化

- マイクロサービス
- イベント駆動アーキテクチャ
- サーバーレスアーキテクチャ

# 2010年代以降：クラウドネイティブ時代

アーキテクチャスタイルの進化

- マイクロサービス
- イベント駆動アーキテクチャ
- サーバーレスアーキテクチャ

**クラウド時代の新たな課題に対応**
**続々とXXXアーキテクチャが出現してくる**

# ソフトウェアの形態と開発スタイルの変遷

# 1960-1970年代：ハードウェア依存時代

専用ハードウェアでのみ動作する組み込み型システムを、少人数でウォーターフォール型の開発で作り上げる時代

【ソフトウェアの形態】

- メインフレーム専用のソフトウェア
- 顧客固有のカスタムシステム
- ハードウェアの付属品的な位置づけ

【開発スタイル】

- ウォーターフォール型
- 少人数での開発
- ハードウェア仕様に強く依存

# 1960-1970年代：ハードウェア依存時代

専用ハードウェアでのみ動作する組み込み型システムを、少人数でウォーターフォール型の開発で作り上げる時代

【ソフトウェアの形態】

- メインフレーム専用のソフトウェア
- 顧客固有のカスタムシステム
- ハードウェアの付属品的な位置づけ

【開発スタイル】

- ウォーターフォール型
- 少人数での開発
- ハードウェア仕様に強く依存

**利用形態が限定的。ワンオフで作りきり、修正ができない**

# 1980年代：パッケージソフトウェアの台頭

各 PC にインストールして利用するパッケージを、バージョン管理とリリース計画に基づくチーム開発で進める時代

【ソフトウェアの形態】

- パッケージソフトウェア
- クライアント/サーバーシステム
- 汎用的な業務アプリケーション

【開発スタイル】

- 複数バージョンの管理
- チーム開発の一般化
- 品質管理プロセスの確立

# 1980年代：パッケージソフトウェアの台頭

各 PC にインストールして利用するパッケージを、バージョン管理とリリース計画に基づくチーム開発で進める時代

【ソフトウェアの形態】

- パッケージソフトウェア
- クライアント/サーバーシステム
- 汎用的な業務アプリケーション

【開発スタイル】

- 複数バージョンの管理
- チーム開発の一般化
- 品質管理プロセスの確立

**利用形態が多様化。ソフトウェアも大規模化する。開発スタイルも複雑化。**

# 1990年代：エンタープライズ化

大規模な業務システムをコンポーネント単位で開発・カスタマイズし、組織全体に展開する反復型開発の時代

【ソフトウェアの形態】

- エンタープライズパッケージ
- Web ベースの ERP/CRM
- カスタマイズ可能なプラットフォーム

【開発スタイル】

- コンポーネントベース開発
- 反復型開発の導入
- グローバル開発チーム

# 1990年代：エンタープライズ化

大規模な業務システムをコンポーネント単位で開発・カスタマイズし、組織全体に展開する反復型開発の時代

【ソフトウェアの形態】

- エンタープライズパッケージ
- Web ベースの ERP/CRM
- カスタマイズ可能なプラットフォーム

【開発スタイル】

- コンポーネントベース開発
- 反復型開発の導入
- グローバル開発チーム

**パッケージソフトウェア時代からSaaS時代への重要な過渡期**
**カスタイマイズも入り、システムに求められる要件も高度化**

# 2000年代：SaaS/Webサービスの時代

ブラウザを通じて継続的に進化するサービスを、アジャイル開発による迅速な機能改善で実現する時代

【ソフトウェアの形態】

- SaaS モデル
- Web アプリケーション
- サブスクリプションベース

【開発スタイル】

- アジャイル開発
- 継続的デリバリー
- フィーチャー単位の開発

# 2000年代：SaaS/Webサービスの時代

ブラウザを通じて継続的に進化するサービスを、アジャイル開発による迅速な機能改善で実現する時代

【ソフトウェアの形態】

- SaaS モデル
- Web アプリケーション
- サブスクリプションベース

【開発スタイル】

- アジャイル開発
- 継続的デリバリー
- フィーチャー単位の開発

**Ruby on Railsの普及**
**新興のWebアプリケーションフレームワークがたくさん生まれた**

# 2010年代以降：クラウドネイティブ時代

クラウド上のサービス群を組み合わせて利用するシステムを、DevOps による自動化と分散開発で構築する時代

【ソフトウェアの形態】

- クラウドネイティブアプリケーション
- マイクロサービス
- API エコノミー

【開発スタイル】

- DevOps
- コンテナベース開発
- インフラのコード化

# 2010年代以降：クラウドネイティブ時代

クラウド上のサービス群を組み合わせて利用するシステムを、DevOps による自動化と分散開発で構築する時代

【ソフトウェアの形態】

- クラウドネイティブアプリケーション
- マイクロサービス
- API エコノミー

【開発スタイル】

- DevOps
- コンテナベース開発
- インフラのコード化

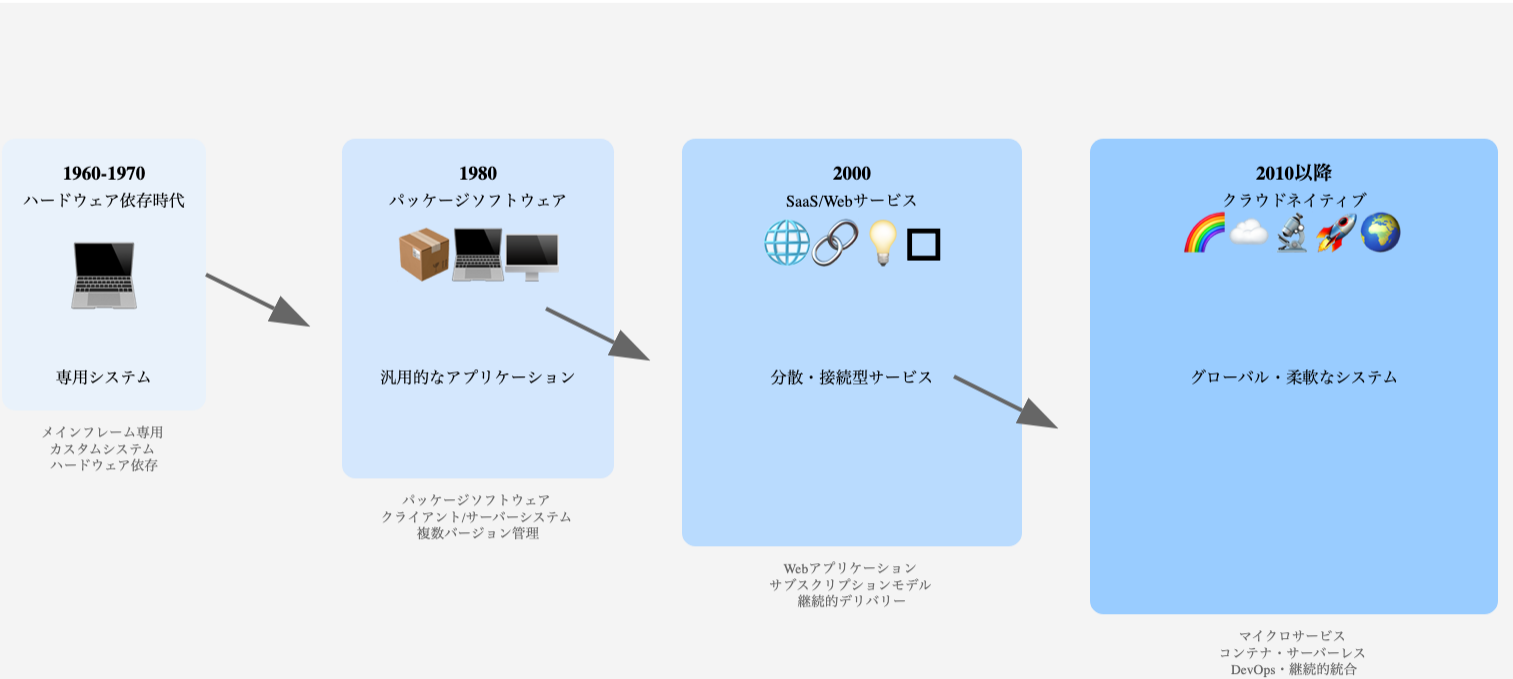
**インフラのコード化**
**モバイルアプリ化や、Single Page Applicationも台頭してきた**

# 変化の本質

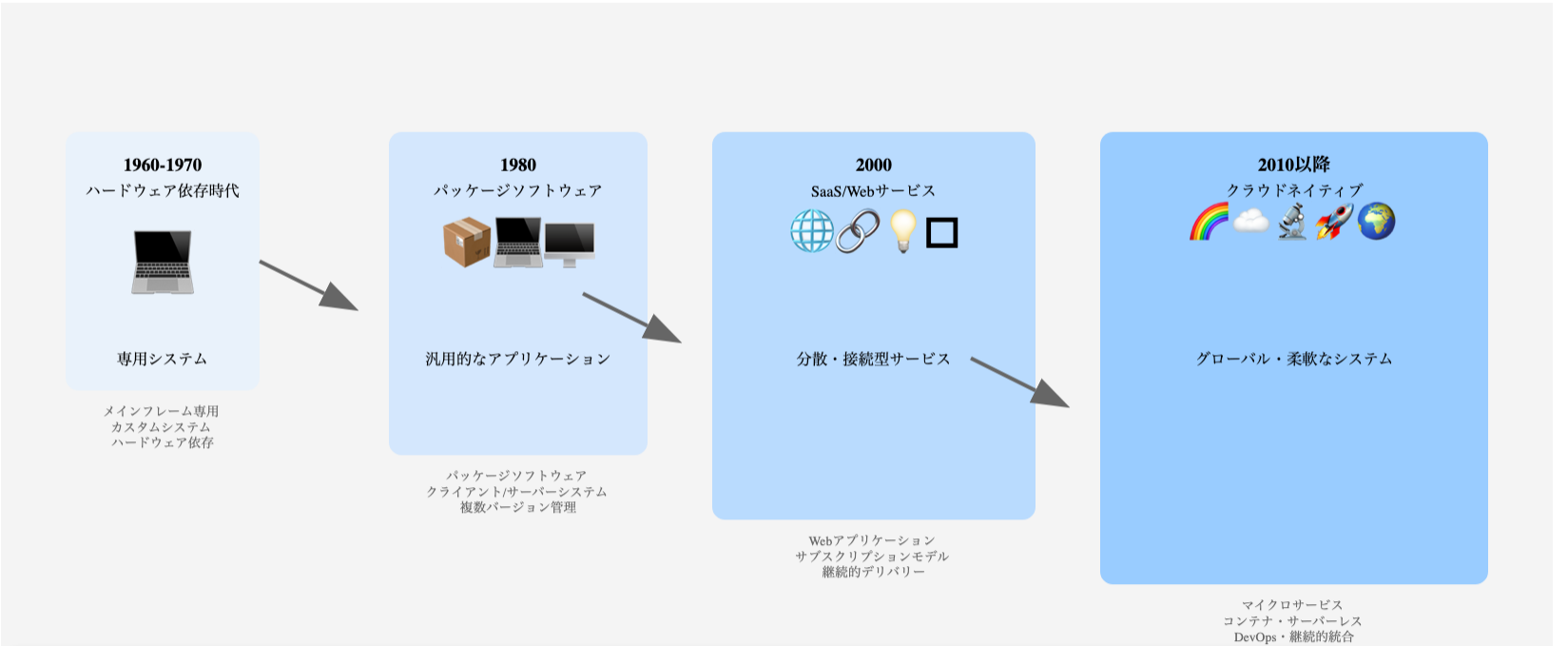
変化のまとめ

| 変化の方向性 | パッケージソフトウェア時代 | SaaS/Webサービス時代 | クラウドネイティブ時代 |
| --- | --- | --- | --- |
| 提供形態 | パッケージ販売 | サブスクリプション | 従量課金/サービス連携 |
| 開発スタイル | ウォーターフォール | 反復型/アジャイル | 継続的デリバリー |
| チーム構造 | 少人数チーム | 大規模チーム | 分散自律チーム |
| アーキテクチャ | モノリシック | コンポーネントベース | マイクロサービス |

# ソフトウェア開発の進化



# ソフトウェア開発の進化




**ソフトウェアはよりリッチで大規模化する**
**考慮する要素も増え、複雑さが増し、開発の難易度も高まり続けている**

# 変化の方向性

これらの変化に対応するため、設計概念も領域が拡大する

1. ビジネスモデルの変化
   - 一時販売 → 継続的な収益
   - カスタマイズ → セルフサービス
   - クローズド → オープン連携
2. 開発プロセスの変化
   - 長期計画 → 短期サイクル
   - 品質保証 → 継続的改善
   - 固定要件 → 柔軟な適応
3. 技術的な変化
   - 密結合 → 疎結合
   - 単一障害点 → 分散システム
   - 手動運用 → 自動化

なんで変化していったの？🤔

# ビジネスや環境の変化に追従するため

現在のソフトウェア開発と似ていませんか？

# 変化し続けるものを作り続けている 

設計概念もその中で生み出された

# 目的も同じ 

ビジネスの変化に追従できるソフトウェア設計が求められる

# 大きな泥団子を作っている場合ではない

茹でガエルになってはいけない

# 大きな泥団子

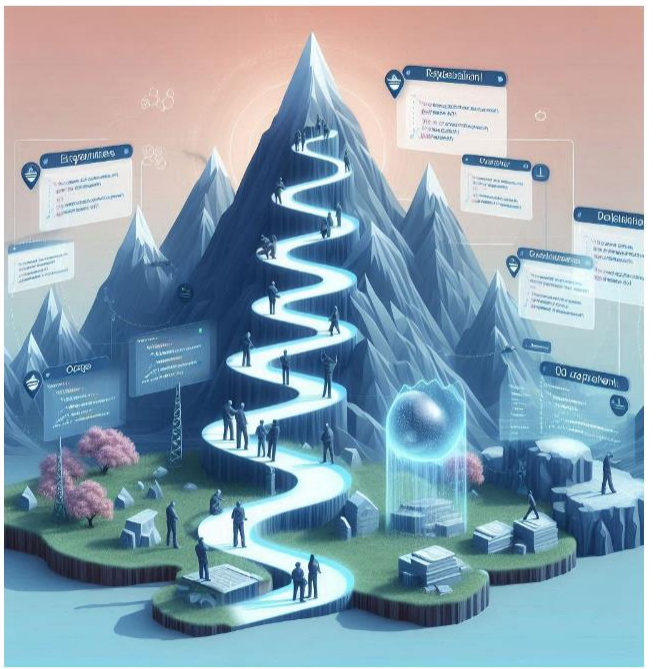
Big ball of mud

明確なアーキテクチャや設計思想がなく、場当たり的に継ぎ足し修正を繰り返した結果、まるで泥団子のように内部構造が複雑に入り組んでしまったソフトウェアシステムのこと

【特徴】

- 理解困難: システム全体の構造が把握しにくく、一部を修正するだけでも全体に影響が及ぶため、変更や機能追加が困難
- 保守性の低下: コードが整理されておらず、可読性が低いため、バグが発生しやすく、修正にも時間がかかる
- 技術的負債の蓄積: 場当たり的な修正を繰り返すことで、システムの内部構造がますます複雑化し、長期的な開発効率を著しく低下させる「技術的負債」が蓄積する

# 実践的なアプローチ



いつ、何を考えるか？💭

# 設計原則

# 常に意識し続けるもの🧠

常に意識し続けるもの🧠

# 設計原則

常に意識し続けるもの

- コーディング時の判断基準
- レビュー時の評価基準
- リファクタリングの指針
- 新規コード作成時のガイドライン

# アーキテクチャパターン

# フレームワーク全盛時代にイチから考えることは少ない🧩

# アーキテクチャパターンは戦術的に扱う

既に誰かが通ってきた道

# アーキテクチャパターンは戦術的に扱う

既に誰かが通ってきた道

フレームワーク選定
時に概ね決まる☑️

# アーキテクチャパターン

フレームワーク選定時に概ね決まる

- 問題解決のためのツールボックスとして捉える

  過去の経験から得られた成功事例に基づいて体系化されている。実践的な知識として活用
- MVC パターンはフルスタックフレームワークなら大体ある
- Laravel なら ActiveRecord パターン（Eloquent）
- Symfony なら Data Mapper パターン（Doctrine）
- 必要に応じて取捨選択

  フレームワークを薄く使う

# アーキテクチャスタイル

# アーキテクチャスタイルは戦略的に扱う

戦略は各システム・ビジネスの事情から生まれるもの

# アーキテクチャスタイルは戦略的に扱う

戦略は各システム・ビジネスの事情から生まれるもの

# 判断が一番むずかしい😵

なぜなら最も抽象度が高い設計判断になる。決定の影響が全体に及ぶ為

# 理想は「僕が考えた最強のアーキテクチャ」を考え抜いて作り込むこと💪

オレオレアーキテクチャを熟成したものがスタイルになる

# 但し、プロジェクト初期は過度に目指さない！⚖️

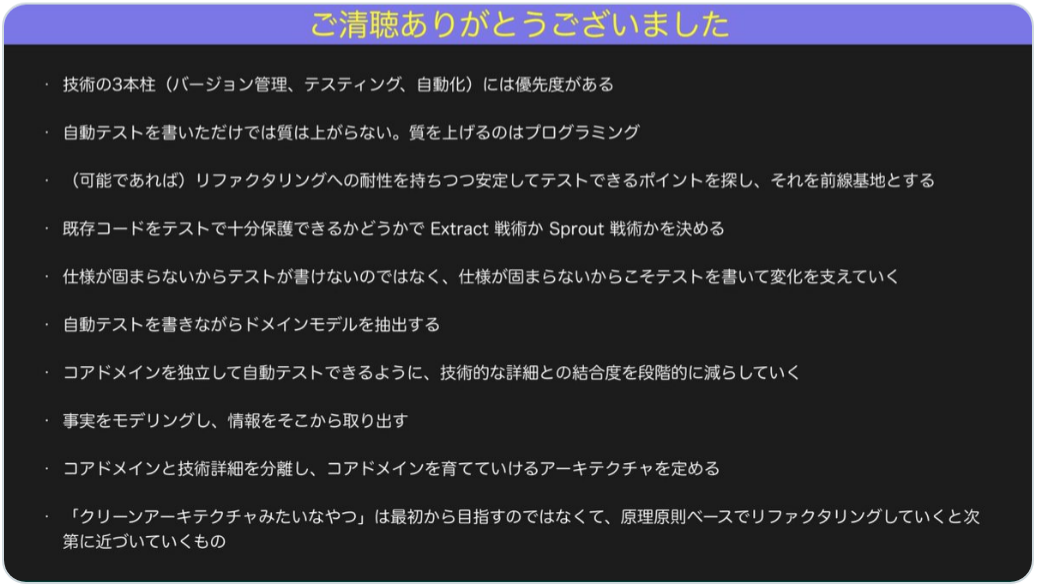
コアドメインの見極めが重要

# 原則ベースの継続的改善していくのがいいのでは？ 

**Takuto Wada**
@t_wada · **Follow**

"「クリーンアーキテクチャみたいなやつ」は最初から目指すのではなくて、原理原則ベースでリファクタリングしていくと次第に近づいていくもの"

これが伝わって欲しい



# 一つの事例

# 段階的にリアーキテクチャしたお話

サービス間でも DIP が適用できる！

326 USERS

## アーキテクチャレベルで依存性を逆転させたら最高だった話

はじめに LITALICO の @katzumi です。 2020 年に LITALICO へ Join して以来、ずっとレセプト業務の開発に携わってきました。 レセプト業務は複雑なドメインゆえミスが許されず、さらに３年に一度の大きな報酬改定があり、ロジックが大幅に変わります。 その改定作業は情報公開から実装完了までの期間が約３ヶ月…

テクノロジー 2024/12/19 08:19


zenn.dev/litalico

アーキテクチャ あとで読む 設計 開発 DDD architecture development

# 記事を通して伝えたいこと

今回は時間の都合ですべてはお話できません

- アーキテクチャはフラクタル構造を持つ
  同じパターンが異なる粒度で繰り返し現れる
- システムは階層的に構成される
  アプリケーション、パッケージ、コンポーネント、クラス、関数、式、文
- 各レイヤーには構造とアーキテクチャがあり、SOLID 原則が適用可能
  設計原則という基礎から、必然的にアーキテクチャが導かれる

# 記事を通して伝えたいこと

今回は時間の都合ですべてはお話できません

- アーキテクチャはフラクタル構造を持つ
  同じパターンが異なる粒度で繰り返し現れる
- システムは階層的に構成される
  アプリケーション、パッケージ、コンポーネント、クラス、関数、式、文
- 各レイヤーには構造とアーキテクチャがあり、SOLID 原則が適用可能
  設計原則という基礎から、必然的にアーキテクチャが導かれる

# アーキテクチャスタイルはいつ考えるべきか？

組織/ビジネスの転換点で検討する

【タイミング】

- チーム規模の拡大時
  チーム間の責務分離が必要に
  独立したデプロイの必要性
- ビジネス要件の変化時
  スケーラビリティ要件の変化
  新規ドメインの追加（新規プロジェクト立ち上げ時も）
- リファクタリングの契機
  技術的負債の蓄積。保守性の課題

【アプローチ方法】

- 小規模な時は設計原則に集中
- 必要なタイミングで戦略的に判断
- ビジネスの成長に合わせた漸進的な改善

# アーキテクチャスタイルはいつ考えるべきか？

組織/ビジネスの転換点で検討する

【タイミング】

- チーム規模の拡大時
  チーム間の責務分離が必要に
  独立したデプロイの必要性
- ビジネス要件の変化時
  スケーラビリティ要件の変化
  新規ドメインの追加（新規プロジェクト立ち上げ時も）
- リファクタリングの契機
  技術的負債の蓄積。保守性の課題

【アプローチ方法】

- 小規模な時は設計原則に集中
- 必要なタイミングで戦略的に判断
- ビジネスの成長に合わせた漸進的な改善

**オーバーエンジニアリングを避け、必要なタイミングで適切な投資判断**
**チームの成長を考慮した判断を行う**

# 設計概念の向き合い方

今回一番伝えたいこと

【ボトムアップの重要性】

- 設計原則という最も具象的な概念が基礎
- 原則の理解と実践が、より大きな設計の質を支える
- 基礎なしには、上位の概念も形骸化する

【アーキテクチャはフラクタル構造】

- 同じ設計原則が異なる粒度で現れる
- システムの規模に関係なく、同じ原則が品質を支える

# 設計概念の向き合い方

今回一番伝えたいこと

【ボトムアップの重要性】

- 設計原則という最も具象的な概念が基礎
- 原則の理解と実践が、より大きな設計の質を支える
- 基礎なしには、上位の概念も形骸化する

【アーキテクチャはフラクタル構造】

- 同じ設計原則が異なる粒度で現れる
- システムの規模に関係なく、同じ原則が品質を支える

**原則の深い理解がパターンやスタイルの適切な選択につながる**
**アーキテクチャスタイルを自然な帰結として捉える**

# まとめ

# 設計概念を実践で活かすために

相互関係性を理解し、バランスよく組み合わせる

設計原則を基礎として

- 日々のコーディングやレビューでの判断基準として活用
- チーム内での共通言語として定着させる
- 継続的な改善の指針として活用

アーキテクチャは段階的に

- 小規模なうちは原則に集中
- ビジネスの成長に合わせて発展
- 必要なタイミングで戦略的に判断

# 継続的な進化のために

アーキテクチャスタイルはシステム全体の構造やコード配置の設計指針となる

実践のポイント

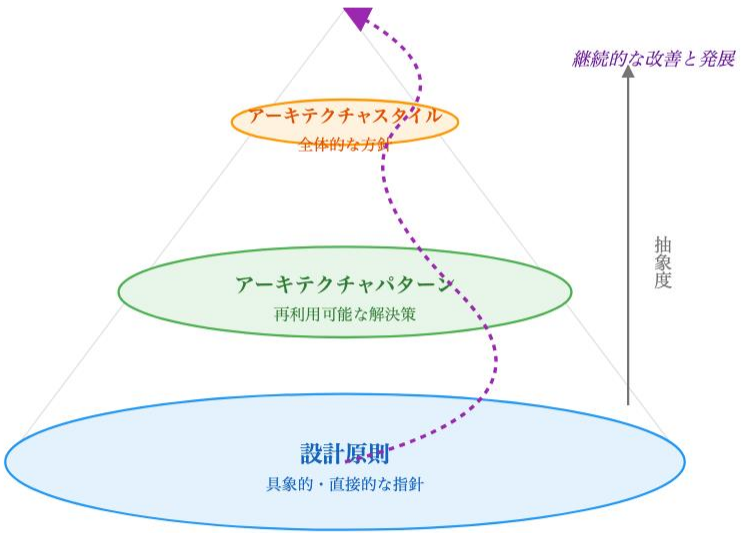
- 完璧な設計を目指すのではなく、変化に強い設計を目指す
- オーバーエンジニアリングを避け、必要に応じて段階的に改善
- チームの理解度とビジネスの要件をバランス

Key Message

- 設計原則という基礎が、より大きな設計の質を支える
- アーキテクチャは自然な帰結として導かれるもの
- 実践を通じた継続的な学びと改善を心がける

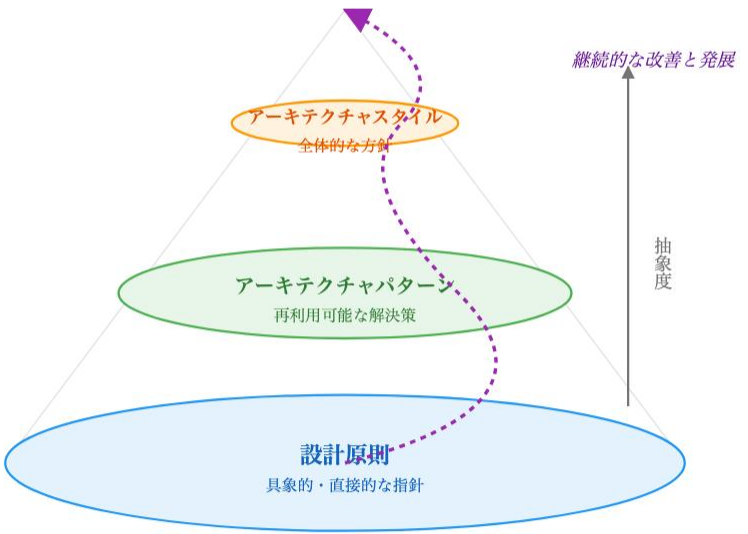
# 進化的アーキテクチャとなる

変化を前提としたシステム設計のアプローチ



# 進化的アーキテクチャとなる

変化を前提としたシステム設計のアプローチ




らせん状に進化するイメージ

# 付録

# 参考資料 & 書籍

紹介した書籍や、学習に使えるページ

【書籍】

- 「TECHNICAL MASTER はじめてのPHP エンジニア入門編」
- Pattern-Oriented Software Architecture
- Patterns of Enterprise Application Architecture
- Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions

【資料】

- Patterns of Enterprise Application Architecture - Martin Fowler’s Bliki (ja)
- Table of Contents - Enterprise Integration Patterns

ご清聴ありがとうございました