オブジェクト指向設計入門

小井土 亨

2006.4.28版

アジェンダ

- 設計の基本
- 設計の原則
- モデルレビュー手法
- テスト駆動開発

設計の基本

1. 設計とは
2. 設計の基本概念
3. 設計の原則
4. 設計品質特性
5. 良いクラスの特徴
6. 設計とテスト容易性

設計とは

- 設計とは
  - 要求、分析の結果をどのように実現するか解決させる活動
  - 設計は、問題を解く作業なので、唯一の正解はなく、あるのは最適解
- 設計の基本
  - 分析の全ての機能を正しく実装する
  - ソフトウェアの完全な姿を提供すべきである
  - 設計結果を利用する人が読みやすく理解しやすい
- 設計の難しさ
  - 利用者の視点から開発者の視点変換
    - 視点変換によってギャップの発生
  - ますます複雑さが増すシステム
  - 相反する要求項目
    - パフォーマンスとセキュリティなど
  - 予測することが難しい要求の変更や機能強化

設計の基本概念　その１

- アーキテクチャ設計
  - ソフトウェアの全体的な構造を一貫性のある概念で設計する
  - システムを論理的な側面から階層構造を決定する
  - 論理的な階層構造を物理構造に対応付ける
- 抽象化
  - 階層的に複数の抽象レベルで設計を行う
  - 抽象レベル間では、手続きとデータが抽象化される
- パターン
  - 特定の状況下で発生する問題について、立証された解法であるパターンを適応する

設計の基本概念　その2

- モジュール化
  - モジュールと呼ばれる個々に名前付けされたコンポーネントに分割する
- 情報隠蔽
  - モジュールは、自分が持つ情報（手続きやデータ）について、必要としない他のモジュールから参照できないように設計する
- 機能独立性
  - モジュールを「だた一つの目的を持った」機能を実現するために設計し、他のモジュールとは必要される最低限の相互作用を行う

設計の基本概念　その3

- 詳細化
  - 高い抽象レベルからそのレベルにあった詳細化を階層的に行う
  - 下位に行くほど、より詳細な設計とする
- リファクタリング
  - 機能や振る舞いを変えることなく、コンポーネントの構造を最適化する
  - 重複をなくす、非効率なアルゴリズムの改修、よりシンプルな構造にするなどを目的に行う

設計の原則

- プログラムをモジュールに分割する
  - 分割することで、プログラムが複雑になるのを避ける
  - モジュール間の依存関係を明確にする
- モジュール間の結合度が低い
  - モジュール間の結合度が低いほうが良い
  - 修正や問題が影響する範囲が少なくなる
- モジュールの凝集度が高い
  - 各モジュール内の凝集度を高くすることで、モジュール間の結合度が低くなる
- 再利用率
  - モジュール分割し、モジュール間の結合度を低くして再利用率を上げる

良い設計とは (設計品質特性)

- 正確性
  - 仕様を正しく満たしていること
- 理解性
  - 設計結果が読みやすく理解しやすいこと
- 一貫性 (統一性)
  - 設計上の個々の概念が首尾一貫して、ぶれがない
- 変更容易性
  - 機能強化などに伴う変更が容易であること
- 頑健性
  - 誤った使い方に対して、システムが適切に対処できること
  - システムの一部のバグが全体に波及しないこと
- 移植性
  - いろいろなハードウェア、ソフトウェア環境へ容易に移植できること
- 効率性
  - 実行効率、資源効率ともに十分実用に適応していること

悪い設計の兆候

- 硬さ
  - 変化しにくいシステム
  - 1つの変更が全体に影響し、多くの変更が必要になる
- もろさ
  - 1つの変更が、論理的には関連のない箇所まで正常動作しなくなる
- 扱いにくさ
  - 正しいことをすることが難しいシステム
- 不透明さ
  - 読みにくく、わかりにくい
- 不必要な複雑さ
  - 本質的に意味を持たない構造が含まれているシステム
- 不必要な繰り返し
  - 同じような構造が繰り返し現われ、共通化できる部分がされていない
- 移植性のなさ
  - 再利用できる部分を切り出すことができない

設計品質特性　正確性

- 正確性とは
  - 仕様を正しく満たしていること
- ポイント
  - 設計の論理的な領域で検証する
    - 仕様（論理的な領域）をマップした領域と実装領域を分離する
  - レビューを受ける
    - レビュー者に分かる設計にする
      - 言葉が大切である
      - ドメインの単語を利用する
- テストの役割
  - テストによって仕様を満足していることを確認する
  - 論理的な視点で検証を行う
    - テストの視点は、論理的な側面からの視点である

設計品質特性　理解性

- 理解性とは
  - さまざまな関係者が設計の結果を十分理解しやすいこと
- ポイント
  - 論理的側面の設計によって理解しやすくなる
  - 既に理解している概念を利用する
    - 現在使用している単語を利用する
    - 知っていることをメタファ（比喩）として利用する
  - 統一された言葉を利用している
    - 同じ単語が同じ概念で利用されている
- テストの役割
  - テストによって処理の流れを明示して、理解を助ける

設計品質特性　一貫性(統一性)

- 一貫性とは
  - 設計上の個々の概念が首尾一貫していること
- ポイント
  - 粒度を合わせる
    - モジュールやAPIなどの粒度を合わせることで一貫性が生まれる
    - 階層化する場合、階層内では粒度を合わせる
  - 統一した概念や名前を使用する
- テストの役割
  - 利用者の視点から粒度の確認を行なう
  - 統一性のチェックには、利用者からの視点が有効である

設計品質特性　変更容易性

- 変更容易性とは
  - 機能強化などに伴う機能追加や変更が容易であること
- ポイント
  - 以下の2つの設計方法を状況に応じて実施する
    - 変更を予測した設計を行う
    - 変更に合わせて再設計（リファクタリング）を行う
- テストの役割
  - テストによって、変更の影響が波及していないことを確認する
  - 再設計（リファクタリング）の実施にはテストが必要
    - 再設計によって、既存の機能が正常に動作することをテストによって確認する

設計品質特性　頑健性

- 頑健性とは
  - 誤った使い方に対して、システムが適切に対処できること
  - システムの一部のバグが全体に波及しないこと
- ポイント
  - システムの各サブシステム間を密結合としない
    - サブシステムが独立して動作するようにする
- テストの役割
  - テストによって、頑丈なシステムであることを確認する

設計品質特性　移植性

- 移植性とは
  - いろいろなハードウェア、ソフトウェア環境へ容易に移植できること
- ポイント
  - 再利用できる部分が切り出されている
  - 環境に依存する部分としない部分を分離する
    - 環境に依存しない部分が依存する部分に依存していないこと
- テストの役割
  - テストによって、環境の影響を確認する
  - テストによって、移植の検証を行なう

良いテストの条件

- テスト目的が明確である
  - 何をテストするか明確である
  - その目的も明確で、更にひとつであること
- テストの判定が正しい
  - テストの成功、失敗が正しく判断されている
- テストを独立して実行することができる
  - テストが他のテストに依存することなく、独立している
- 繰り返し実行することができる
  - 何度でも繰り返して、テストを実行することができる
- テストを実行しても、状態が変化しない
  - テストを実行し、成功した場合でも失敗した場合でも、テストを実行する前と後で何も変わらない

良いテストの効果

- メソッドが一つの機能を実現している
  - メソッドが明確な一つの機能だけを提供する
- メソッドの結果を提供する
  - 外部に対して、メソッドの処理結果を判断できるような何らかの方法を提供する
- クラスやメソッドの独立度が高いこと
  - クラスやメソッドが、他のクラスやメソッドとの依存が低い
- 特定の環境への依存度が低い
  - 特定のファイルやデータベース構造などに対する依存度が低い

良い設計の特徴

- 完全性と十分性
  - システムが目的を達成するために必要な機能のみを含んでいる
- 原子性
  - 1つのサービスを実現することに焦点があたっている
- 高凝集度
  - 少数に絞られた責務を持ち、その責務を実装するためだけの機能を持つ
- 低結合度
  - 他の部分との連携が必要最低限である

設計品質とテスト容易性

❖ 良いテストの効果と良い設計の相関関係

完全性	良いテストは目的が明確で、必要な機能を包含している
原始性	良いテストは特定の目的だけに絞られている
高凝集度	良いテストはテスト対象を特定の責務に限定している
低結合度	良いテストは独立して実行することができる

テスト容易性が設計品質の指針

変わる企業IT

- 企業のITのコモディティ化
  - IT Doesn’t Matter
    - ニコラス・G・カーのレポート
  - ITは差別化の道具ではなく、標準装備（必須）の道具
  - ITは戦略ではなく、ビジネス戦略を実現する道具
- 企業がITに求めるものは
  - 高い品質
  - 容易に変更できるIT
    - ビジネスのスピードに追随するIT

変更容易性が設計品質の重要な指針

設計の原則
オブジェクト指向設計

1. **設計の5つの原則**
2. GRASP**パターン**

設計の難しさ

- 要求や分析の変更による再設計
  - 要求や分析が変わることで、再設計をする必要がある
- 再設計の要因
  - クラスを明確に規定してインスタンスを生成
  - 特定のオペレーションに依存
  - プラットフォームに依存
  - オブジェクトの表現や実装への依存
  - アルゴリズムへの依存
  - 密な結合
  - サブクラス化による機能拡張
  - 簡単に修正できないクラスの修正が必要になる

設計の原則　単一責任の原則

- SRP (The Single Responsibility Principle)
  - クラスは、1つの役割 (責任) だけを持つ
  - 1つの理由のみによって、クラスが変更される
  - 役割を単独でテストできる
  - 役割＝変更理由＝テスト単位
  - 凝集 (cohesion) と同意
    - 凝集とは、モジュールに含まれる要素間の機能的な結び付き
- 原則の適用条件
  - 役割の変更が確実に想定される場合
    - 理由なく適応すると、「不必要に複雑な設計」となる
  - 役割をテストする必要がある場合
- 問題の予兆 (もろい設計)
  - ある役割の変更に伴い、本来関係ないと思われる役割のクラスに変更の影響がある
  - 独立したテストが困難なクラスがある

設計の原則　オープン･クローズドの法則

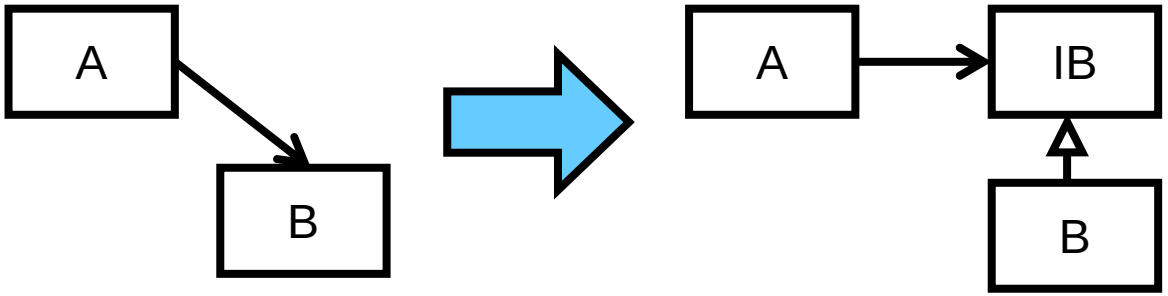
- OCP (The Open-Closed Principle)
  - ソフトウェアの構成要素 (クラス、モジュール、関数など) は拡張に対して開いていて、修正に対して閉じていなければならい
  - 拡張に対して開いているとは
    - 構成要素の振る舞いを拡張できる
  - 修正に対して閉じているとは
    - モジュールの振る舞いを拡張しても、既存のモジュールは変更の必要がない
- 原則の適用
  - 長く使い続けられるシステムを開発する
  - システムに機能の追加や変更し、成長させる必要がある
    - 確実でない変化に対応するための準備は「無意味に複雑な設計」となる
- 問題の予兆 (硬い設計)
  - ちょっとした変更でさえ、修正箇所と依存関係のある全ての箇所を修正する必要がある
  - 機能の追加や修正を行なったときに、それに伴って修正しなければならないif文やswitchが数多くある
  - コピー＆ペーストで、大量の同じコード存在する

設計の原則　リスコフの置換原則

- LSP (The Liskov Substitution Principle)
  - 派生クラスは、その基本クラスと置換え可能でなければならない
  - 派生クラスは、基本クラスが許可する全てを許可しなければならい
- 原則の適応条件
  - 基本クラスの派生型を作成する場合
  - 契約による設計
    - 派生クラスは、基本クラスの事前条件より等しいか弱いか
    - 派生クラスは、基本クラスの事後条件より等しいか強いか
    - 単体テストによって契約を明記する
- 問題の予兆 (もろい設計)
  - 派生クラスが増えた場合に基本クラスに変更が必要
    - 派生クラスごとの分岐処理が基本クラスに存在する
  - 基本クラスの機能を派生クラスが無力がする
    - 基本クラスの機能を派生クラスで何もしない動作と置き換える

設計の原則　依存性逆転の原則

- DIP (The Dependency Inversion Principle)
  - 上位モジュールは、下位モジュールに依存してはならない
  - 抽象 (目的) に依存させる
  - 実装の詳細に依存させない



- 原則の適応条件
  - レイヤ構造によって、システムの複雑性を低くする
  - 変更耐えられるコードを構築する必要がある
  - コードの保守が想定される
- 問題の予兆 (もろい設計)
  - 下位モジュールの変更が上位モジュールに影響する

設計の原則　インターフェイス分離の原則

- ISP (The Interface Segregation Principle)
  - クライアントに、クライアント依存しないメソッドへの依存性を強制してはならない
  - インターフェイス内の強い関連があるものをグループ化して分離する
- 原則の適応条件
  - 変更に強いシステムを開発する
    - 多目的なインターフェイスを目的ごとに分離し、不用意な結びつきを避ける
    - クラス間の依存度を下げて、低結合度を実現する
  - 目的ごとのテストを作成する
- 問題の予兆 (もろい設計)
  - クライアントが利用していないメソッドの変更に影響される
  - インターフェイスの実装クラスの変更が広範囲に影響する
  - インターフェイスが部分的 (グループ単位) にしか強い関連を持っていない

GRASPパターン

- **責務とは**
  - **必要な情報を持つ**
  - **情報に関連した振る舞いを持つ**
  - **関連する相手を知っている**
  - **責任の範囲**
- GRASP**パターンとは**
  - General Responsibility Assignment Software Pattern
  - **責務の割り当ての原則**
  - 9**のパターンから構成される**
    - **情報エキスパート、生産者、コントローラ**
    - **疎結合性、高凝集性**
    - **多相性、純粋架空物、間接化、バリエーション防護**

GRASPパターン　代表的なパターン

- 情報エキスパート　パターン
  - 責務を遂行するために必要な情報を持つクラスに責務を割り当てる
    - 責務を満たすためのメソッドを実装する
  - 良いカプセル化を実現する
    - 情報と関連の深い処理をまとめる
- 生産者　パターン
  - あるクラスのインスタンスをどのクラスが生成すべきか
  - 関係が密であるクラスにインスタンス生成の責務を割り当てる
  - 密なクラスとは
    - 全体クラスと部分クラス
      - 生成は全体クラスが行なう
  - 依存関係を少なくなるように責務パターンを利用する
- コントローラ　パターン
  - コントローラ　クラスにシステム　イベントを処理する責務を割り当てる
  - コントローラ　クラスとは
    - システムやサブシステムを表すクラスを用意し、外部からのメッセージを受け取る
    - 現実世界には対応しないクラスである

GRASPパターン

- 疎結合性
  - 結合性を疎に保つように責務を割り当てる
- 高凝集性
  - 凝集性が高くなるように責務を割り当てる
- 多相性
  - クラスのバリエーションがある場合は、ポリモーフィズムを利用して各クラスに責務を割り当てる
- 純粋架空性
  - 便宜的なクラスを作成して責務を割り当てる
- 間接化
  - ほかのコンポーネントとの間にクラスを配置し、間接的に接続するような責務を割り当てる
- バリエーション防護
  - 不安定な部分の周囲に安定したインターフェイスとなるクラスを配置して責務を割り当てる

モデルレビュー手法

1. モデリングとは
2. モデルレビューの基本的な技
3. 設計モデルのレビュー

設計のモデリング

- モデリングとは
  - 概念（仕組み）を明示して伝えるため
  - 複雑な設計の意図を関係者に理解しやすくするため
  - 関係者間の理解の共有や合意形成が開発を正しい方向に進める
- モデルには洗練が必要
  - 再設計、リファクタリング
- 注意
  - モデリングを目的にしない
    - モデルを作ることが目的ではなく、目的は設計を明確にして正しく実装できるようにすること
  - モデル以外の設計書も必要

モデルレビュー

- レビューとは
  - レビュー対象が正しいか検証すること
- 何を検証するのか
  - モデルが正しいか検証する
  - 書きたいことモデルが一致していることを確かめる
  - 作成者の意図が正しく表現されているか確かめる
  - 理解しやすいモデルになっているか検証する
- レビュー方法
  - 読んでみる
  - 数えてみる
  - 絵としてみる
- レビューポイント
  - 名前
  - 関係

レビューの基本的な技

- 読んでみる
  - モデルを日本語で読んでみる
  - 日本語として不自然であれば何らかの問題がある
- 数えてみる
  - モデル要素の数を数えて、モデルの複雑度などを検証する
  - 閾値の基本は、人間の認知できる範囲
    - マジックナンバーは、7±2
- 絵としてみる
  - ビジュアルに表現ざれたモデルを絵と見ることで複雑性を検証する
  - 複雑では、保守性や拡張性がないものである
  - 偏りがある場合、何らかも問題が内在されている

設計モデルのレビュー　クラス図

- クラスの責務を検証
  - メソッド名から責務が理解できる
  - メソッド名からクラスの目的が理解できる
- クラスの分割の適切さを検証
  - パブリックなメソッドが3個から9個
  - メソッドが少ない
    - メソッドをクラスとして分割している
    - クラスの内部が構造化されていない
  - メソッドが多い
    - クラスに過度の責務が割り当てられている
    - クラスが大きすぎる
    - メソッドをグループ化して、クラスを分解する

設計モデルのレビュー　パッケージ図（クラス図）

- パッケージの大きさを検証
  - クラスの数が5から9
  - クラスの数が少ない
    - パッケージの統合を検討する
  - クラスの数が多い
    - サブパッケージの導入を検討する
- パッケージツリーを検証
  - 上から「パッケージ名のパッケージ名のパッケージ名」とので繋げて読む
  - 自然に読めなかったら
    - ツリー構造に問題がある

設計モデルのレビュー　クラス間の関係

- クラスの依存線を検証
  - 矢印の方向に「AはBに依存します」と読む
  - 依存線は密な結合になるので、不必要なら削除する
- 継承関係を検証
  - 「下位クラス名は上位クラス名」の一種です
  - 日本語として不自然である
    - 複数のクラスの属性やメソッドを単に上位クラスに集めている
    - 実装の再利用をしようとしている
    - 継承関係に混乱をきたしている
- コンポジット関連を検証
  - 「全体側のクラス名とコンポジットされているクラス名の寿命は運命共同体です」と読む

設計モデルのレビュー

- 正しさの基準
  - 設計モデルは、問題をどのように解決するかの視点
  - 設計には、「唯一の解」はなく「最適解」がある
    - 変化するシステムで常に「最適解」にするには、再設計が必要
- 設計モデルの品質特性
  - 形式基準
    - 文法に誤りがないこと
    - 実行可能であること
  - 内容基準
    - 最適な解が選択されていること

設計モデルのレビュー　品質特性

- 標準準拠性
  - 命名規則に準拠している
  - 用語集に準拠している
  - ガイドラインに準拠している
- 形式的網羅性
  - すべての要素に名前が付いている
  - すべてのクラスやインターフェイスに責務が割り当てられている
- 形式的無矛盾性
  - インターフェイスとクラスが区別されている
- 明確性
  - 責務を第三者が読んで、意味の取り違えがない
  - インターフェイスが仕様化（テスト可能）されている

設計モデルのレビュー　内容基準

- **合目的性**
  - 解決すべき問題の解となっている
- **正当性**
  - 選択した設計に論理的説明が可能
- **妥当性**
  - 可変部分に対するモジュール化が行なわれている
  - 技術的リスクに対する回避策がモデル化されている
- **内容的網羅性**
  - すべてのサービスに対する構造と振る舞いが設計されている
- **内容的無矛盾性**
  - 複数の可変部分に対する優先順位と設計モデルに矛盾がない
- **拡張性**
  - 機能追加に対して、単純な追加や変更で対応可能である
- **保守性**
  - データが増えた場合に、属性のクラスのみを修正すればよい
- **合理性**
  - 共有クラスと共通クラスが使い分けられている
- **追跡性**
  - 要求、分析モデルとマッピングが可能である

テスト駆動開発

1. アジャイル開発とは
2. テスト駆動開発とは
3. 作業手順
4. テストの効果

求められているソフトウェア開発

- ビジネスの変化
  - 顧客のニーズに合わせて、ビジネスを柔軟に変化させることが成功の鍵
  - ビジネスの変化に合わせて、ソフトウェアも素早く対応（変更）することが求められている
  - 多くの変化は予測不可能
- ソフトウェア開発のリスク対策
  - ビジネスチャンスには、リスクがある
  - リスクを軽減する開発方法論が必要

アジャイル・ソフトウェア開発　必要項目

- ニーズへの的確な対応
  - 早期に動作するソフトウェアを顧客に提供し、フィードバックを繰り返す
- 変更コストを一定に保つ
  - 常に変更しやすい状態に保つ
    - 変更しやすい設計 (シンプルな設計)
    - 設計の改善 (リファクタリング)
- リスクの軽減
  - システム状態を可視化して、発生した問題を素早く対処する
- 高い品質
  - 機能の正しさ、実行速度、使い勝手など

アジャイル・ソフトウェア開発の答え

- アジャイル・ソフトウェア開発の原則 (抜粋)
  - 早期かつ頻繁にソフトウェアを納品する
  - 顧客のビジネス競争の優位性を確保する
    - 継続的に、要求の変更に対応する
  - 顧客とは、契約交渉ではなく協調
  - より効果的なことを探るために見直しを定期的に行う

変化に強く、継続的で迅速なソフトウェア開発
アジャイル・ソフトウェア開発

テスト駆動開発

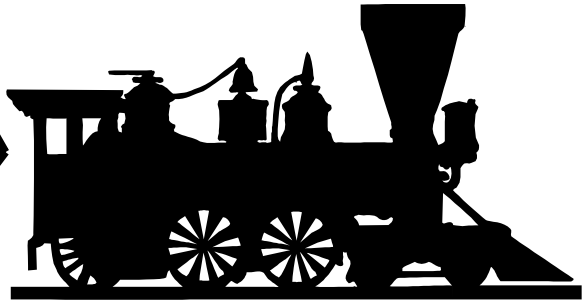
継続的なリリース
継続的なテストが必要

設計の改善
自動テストによる確認が必須

リスクの軽減
テストで進捗と状態を可視化

高い品質
テストによる確認が必要

テストで
ソフトウェア開発を牽引



誤解と実際

- 誤解
  - ✕ 設計せずに、すぐに実装を行う
  - ✕ 実装を始める前に、全てのテストを作成する
  - ✕ コードが唯一のドキュメントである
- 実際
  - ○ コードを重要なドキュメントのひとつと位置づける
  - ○ 要件をテストとい形で明確化する
  - ○ 自動テストを作成してから実装を行う
  - ○ テストを単位として、小さいステップでソフトウェア開発を進める

コードの位置づけ

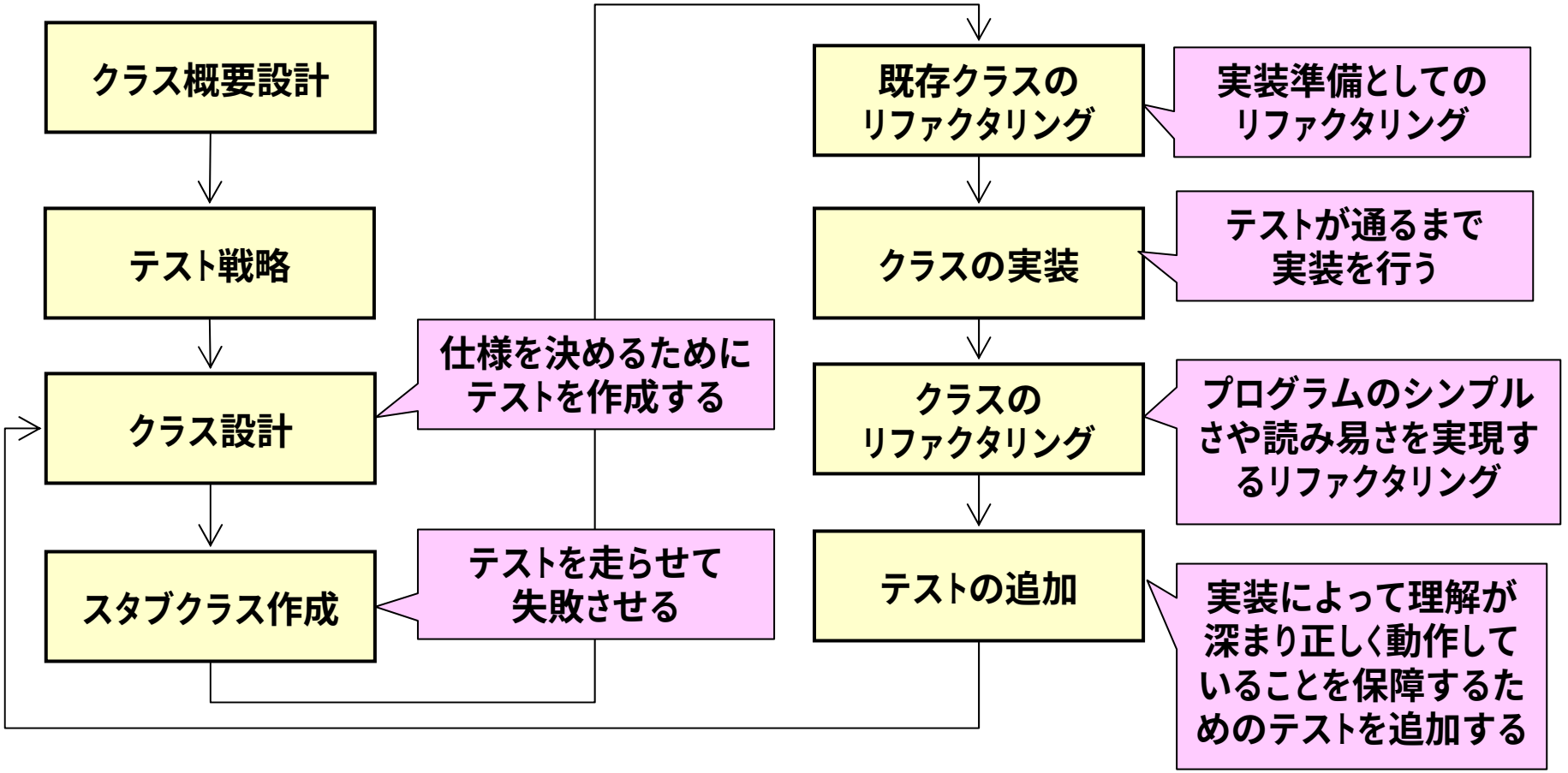
- コードは重要なドキュメントのひとつ
  - 目標「動作するきれいなコード」
  - 詳細かつ正確なドキュメント
  - ドキュメントとして認識して、クリアで読みやすくする
- テストコードは貴重なリソースのひとつ
  - 継続的な開発で繰り返し利用される
  - 運用で問題が発生した場合
    - テストを追加する
    - 既存のテストを実行し、修正を確実に行う

テストの位置づけ

- テストの目的
  - 振る舞い (Behaviour) を定義する
  - 仕様を定義する
- テストの効果
  - 品質が維持される

テストは動作可能な設計書 (一部)

作業手順


クラス概要設計
テスト戦略
クラス設計
スタブクラス作成
仕様を決めるために
テストを作成する
テストを走らせて
失敗させる
既存クラスの
リファクタリング
クラスの実装
クラスの
リファクタリング
テストの追加
実装準備としての
リファクタリング
テストが通るまで
実装を行う
プログラムのシンプル
さや読み易さを実現す
るリファクタリング
実装によって理解が
深まり正しく動作して
いることを保障するた
めのテストを追加する

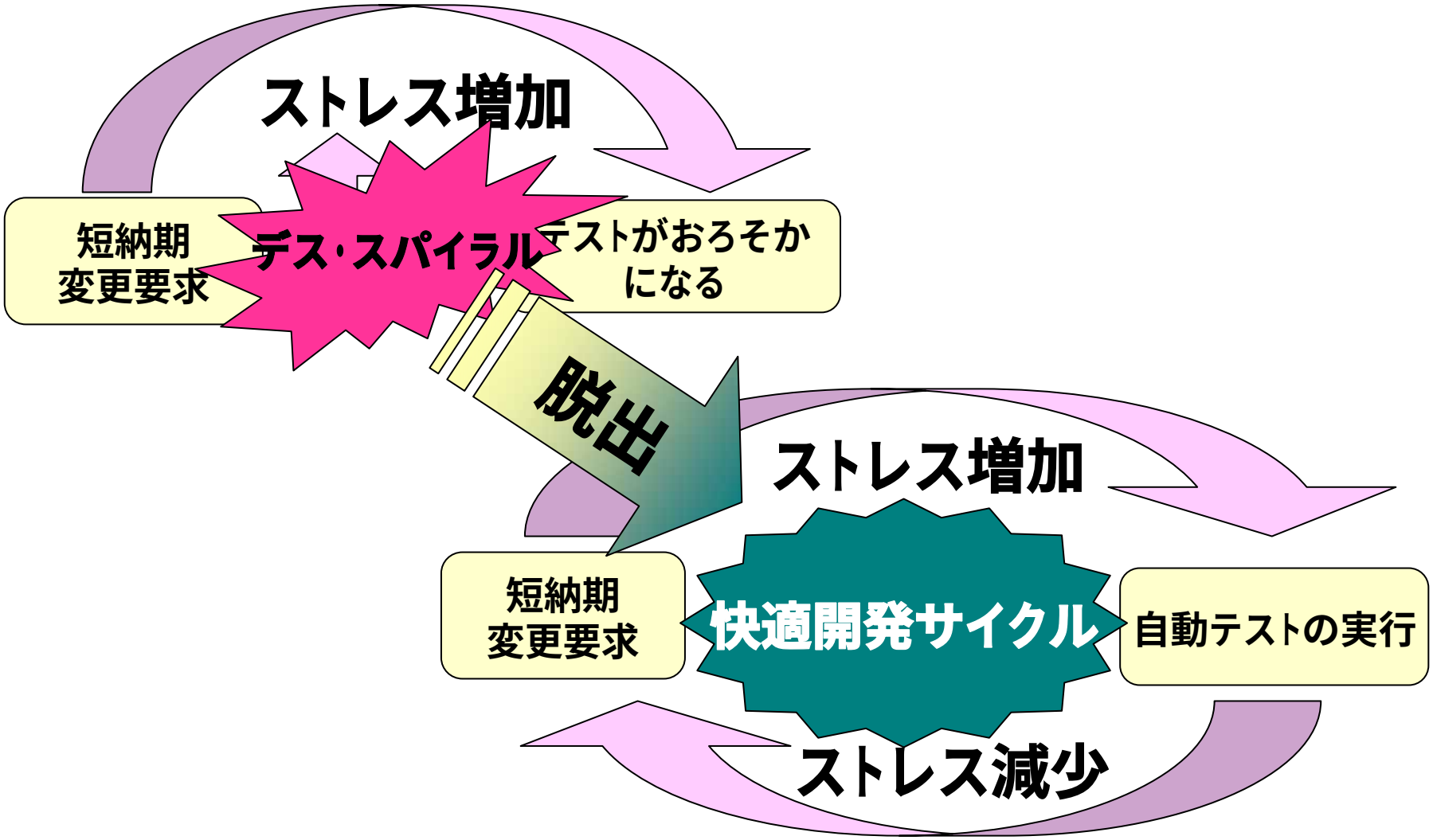
プログラミング手順

- プログラミングの順番
  - レッド
    - 動作しないテストを少しだけ作成する
  - グリーン
    - テストを直ぐに動作させる
    - 最初のコードはきれいでなくても良い
  - リファクタリング
    - 重複や分岐のないきれいなコードにする
- 結果
  - プログラムコードの欠陥密度を低下させる

テストの効果　ー　高凝集で疎結合

- 効果
  - システムが極めて高凝集で疎結合のコンポーネントから構成される
- 理由
  - テストは、明確な一つの目的だけを検証する
    - クラスとメソッドが一つの明確な目的を持つ
  - テストを独立した状態で実行する
    - クラスとメソッドの独立性が高くなる
  - テストは繰り返し実行される
    - テスト対象クラス以外の状態を変化させない

テストの効果　ー　デススパイラルからの脱出



テストの効果　ー　見える化（予測可能）

- ゴールの見える化
  - テストが通過する（ゴール）まで実装作業を行う
- プログラム異常の見える化
  - システムの変更を分単位で自動チェックし、自動インテグレーション（ビルドとテストの実行）を行う
  - 短時間で検証されるので、直ぐに対処可能
- 進捗の見える化
  - 受入れテスト（要求）が通過した数で進捗を明確にする

ギャップをコントロールする

- ギャップの認識
  - プログラミング中の決定とフィードバッグ（テストの実行結果）とのギャップを認識する
- ギャップの制御
  - 小さいステップで進めて、具体的に学習し、ギャップを大きくしない
  - 設計の決定を追加（改善）するために、リファクタリングを行う

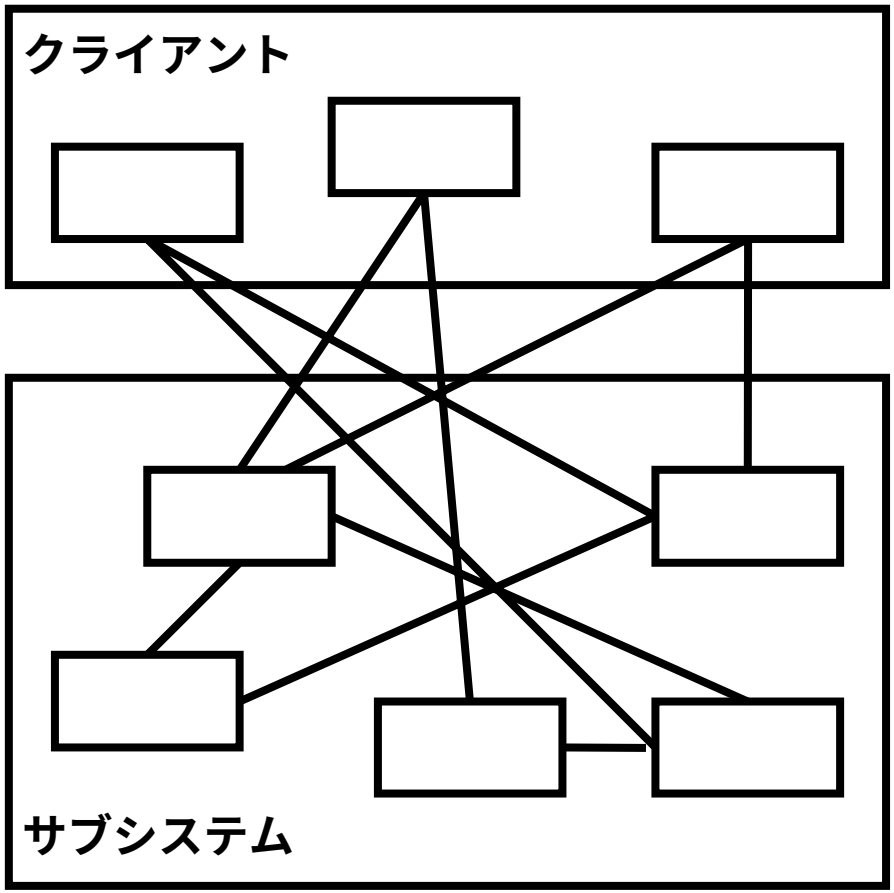
演習1

- レストランなどの飲食店を例として設計を考える
  - お客とオーダー係と調理係がいる場合

  - 多くのお客に対応するためには、どのような設計変更が有効か？

演習2　構造の最適化

- Façade
  - **依存関係を少なくするように構造を変更してください。**



演習3　会議室の予約システム

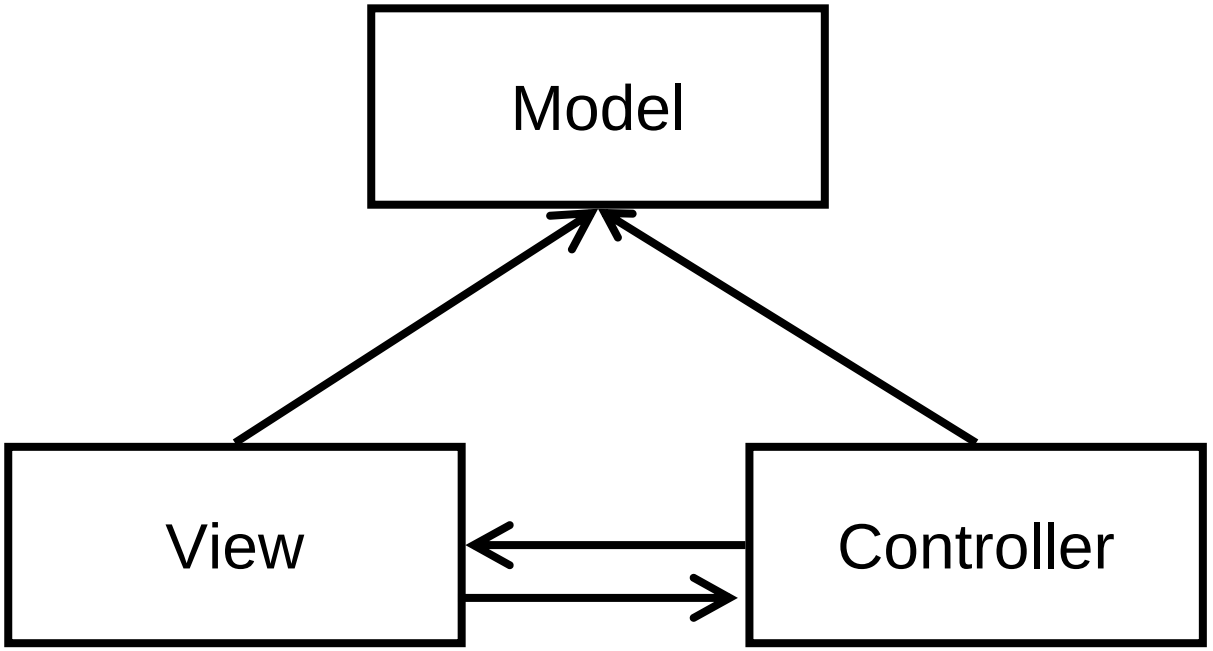
- 会議室の種類
  - 会議室は、A、B、Cの3つがある
  - 応接室は、第1、第2がある
- ユーザーストーリーについて
  - 従業員が会議室を予約する
    - 会議室の予約状況を確認する
    - 会議室が空いている場合は予約を行なう
  - 応接室を予約する
    - 応接室を予約できるのは、課長以上の従業員である

演習4　大学のゼミの履修管理

- 登場すると思われるクラス
  - 教授、ゼミ、学生
  - 建物、部屋
- ユーザーストーリーについて
  - 学生がゼミに登録する
    - 学生がゼミに空きがあるか確認する
    - 空きが合った場合は、ゼミに登録することができる
  - 成績証明書の発行
    - 学生は、ゼミに関する成績証明書の発行を請求する
    - 成績証明書を取得する

資料　基本的な構造パターン

- MVC



参考文献

- 実践ソフトウェアエンジニアリング　ロジャー　S･プレスマン
- 森口 繁一編『ソフトウェア品質管理ガイドブック』日本規格協会(1990)
- アジャイルソフトウェア開発の奥義　ロバート･C･マーチン
- 日経ソフトウエア「正しく学ぶソフトウェア設計」
- 実践UML　パターンによる統一プロセスガイド　クレーグ･ラーマン
- UMLによる　オブジェクト指向モデリング　セルフレビューノート　荒井玲子
- UML　モデリングの本質　児玉公信
- テスト駆動開発入門　ケント･ベッグ