



# システム開発技術

## ◆ モジュール結合度とは

- 定義  
あるモジュールが他のモジュールに **どの程度依存しているか** を示す指標。
- 考え方
  - 結合度が強い ... モジュール同士がべったり依存 → 修正に弱くなる
  - 結合度が弱い ... モジュール同士が独立 → 修正や再利用がしやすい

## ◆ 結合度の種類（強い→弱い順）

教科書的には以下の7段階で説明されます（強い＝悪い、弱い＝良い）。

1. 内容結合 (**Content coupling**)
  - 他モジュールの内部処理や変数に直接アクセスしている
  - 例：モジュールAがモジュールBのローカル変数を書き換える
  -  最悪のパターン
2. 共通結合 (**Common coupling**)
  - 複数モジュールが同じグローバル変数を共有している
  - 修正の影響が広がりやすい
3. 外部結合 (**External coupling**)
  - 外部のデータ形式やプロトコルに依存
  - 例：ファイルフォーマットや通信プロトコルに強く依存
4. 制御結合 (**Control coupling**)
  - 他モジュールの動作を制御するためにフラグなどを渡す
  - 例：引数に「mode=1なら処理A、mode=2なら処理B」と書く
5. スタンプ結合 (**Stamp coupling / Data-structured coupling**)
  - 必要なデータだけでなく構造体やレコード全体を渡す
  - 例：社員番号だけ使うのに「社員情報構造体」を丸ごと渡す
6. データ結合 (**Data coupling**)
  - 必要なデータだけを引数として渡す
  - 例：calculateTax(income)
  -  一般的で健全なやり方
7. 無結合 (**No coupling**)
  - まったく依存関係がない
  - 完全に独立したモジュール



## ◆ モジュール強度（凝集度）とは

- モジュール内の処理やデータの **まとまりの強さ** を表す
- 強いほど「そのモジュールが一貫した目的を持つ」→ 修正しやすく、再利用しやすい
- 弱いと「なんでも詰め込んだ雑多なモジュール」→ 修正の影響が広がりやすい

## ◆ モジュール強度の種類（弱い → 強い）

教科書的には **7段階** に整理されます。

1. 偶発的強度 (**Coincidental Cohesion**)
  - たまたま処理をまとめただけ

- 例：ログ出力処理と数値変換処理を同じモジュールに入れる
-  最悪のケース
- 2. 論理的強度 (Logical Cohesion)
  - 論理的に似た処理をまとめているが、目的はバラバラ
  - 例：「入力処理」モジュールに、キーボード入力・マウス入力・音声入力など全部入れる
- 3. 時間的強度 (Temporal Cohesion)
  - 実行タイミングが同じものをまとめただけ
  - 例：プログラム開始時の初期化処理を全部ひとまとめ
- 4. 手続き的強度 (Procedural Cohesion)
  - 一連の処理手順だからまとめた
  - 例：ファイルを開く → 読む → 閉じるを全部1つのモジュールにする
- 5. 通信的強度 (Communicational Cohesion)
  - 同じデータを使う処理をまとめた
  - 例：顧客データを参照して「住所を更新」「請求先を変更」する処理
- 6. 順序的強度 (Sequential Cohesion)
  - ある処理の出力が次の処理の入力になるためまとめた
  - 例：データを読み込む → 解析する → 出力する
- 7. 機能的強度 (Functional Cohesion)
  - モジュールが **1つの明確な機能だけ** を実現
  - 例：「税額を計算する」モジュール
  -  最も望ましい形

## 図一覧

### ◆ 構造化チャートとは

- プログラムをモジュール（部品）に分けて、その 階層構造と関係 を表す図
- トップダウン設計で用いられる
- UMLのクラス図やシーケンス図とは違い、「処理モジュールの構造と流れ」に特化
- 👉 ざっくり言うと「プログラムの部品図」みたいなもの

### ◆ DFD（データフロー図）とは

- システム内で データがどのように処理され、どこに流れていくか を表現する図
- 「処理の流れ」ではなく「データの流れ」に注目
- 利用者とのやり取りや外部システムとのインターフェースを整理するのに有効
- 👉 つまり「データ中心のシステムの見取り図」

### ◆ シーケンス図とは

- オブジェクト間のメッセージのやり取りを、時間の流れに沿って表す図
- 縦方向：時間の流れ（上から下へ）
- 横方向：登場するオブジェクトやコンポーネント
- 「誰が誰にメッセージを送って、どう応答するか」がわかる

👉 「システムの動作シナリオ」を視覚的に確認できる図

### ◆ コミュニケーション図とは

- オブジェクト間のやり取り（メッセージ交換）を、関係構造に基づいて表現する図
- 「どのオブジェクトがどのオブジェクトとやり取りしているか」がメイン
- メッセージの順序は番号を付けて表す（時系列の軸は持たない）

👉 「関係性（つながり）」に注目するのがポイント。

### ◆ 状態遷移図とは

- 対象（オブジェクトやシステム）がとりうる状態と、それを変化させるイベントを表した図
- 「今どの状態か」「何をきっかけに別の状態へ移るか」を整理できる
- イベント駆動型のシステムや、状態によって挙動が変わるものに有効

👉 フロー（処理の流れ）ではなく、状態の変化 に注目

### ◆ ペトリネットとは

- 状態遷移を表すグラフモデル
- 並行処理や同期・非同期処理を表すのに適している
- 「状態遷移図」の拡張版のようなもの
- 1960年代に Carl Adam Petri が提案

👉 OSのプロセス管理や通信プロトコル、ワークフローのモデリングなどで使われる

### ◆ 配置図とは

- システムの物理的構成 を図で表すもの
- どのハードウェア（サーバ、PC、スマホなど）に、どんなソフトウェア（アプリ、コンポーネント）が配置されるかを示す
- ネットワーク構成や分散システムの設計で利用される

👉 「システムがどこで動くか」を示す図

### ◆ コンポーネント図とは

- ソフトウェアの「物理的な部品構造」を表す
- モジュール、ライブラリ、サービスなどを「コンポーネント」として描く
- それらが どのように依存・接続してシステムを構成しているか を示す

👉 「配置図」がハードウェア寄りなのに対し、コンポーネント図はソフトウェア寄り

### ◆ アクティビティ図とは

- 処理の流れを可視化する図
- UMLでフローチャートに相当する
- 条件分岐、並行処理、開始・終了を表現できる
- 業務フローやシステム処理の流れを共有するのに使う

👉 「システムや業務のワークフロー図」

### ◆ ユースケース図とは

- システムの外側から見た 利用者の視点の機能一覧 を表す
- 「誰（アクター）が」「何をする（ユースケース）」の関係を図示
- 要件定義や外部設計の初期段階で使われる

👉 「システムは何をするのか」を利用者目線で整理する図  
(UMLの一つ)

### ◆ E-R図とは

- 実体 (Entity) ... システムで管理したい対象 (人・物・事柄)
- 属性 (Attribute) ... 実体を持つ性質 (名前、住所、価格など)
- 関係 (Relationship) ... 実体同士の関連 (注文する、所属する など)

👉 「何をデータベースに入れるか」「どう結びつけるか」を整理するための図

### ◆ スタブ (Stub)

- 意味  
上位モジュールをテストするときに、呼び出される下位モジュールの代用品。
- 役割  
本物の下位モジュールがまだ完成していないときに「ダミー処理」を返す。

👉 上位から下位へテストするトップダウンテストで使う。

### ◆ ドライバ (Driver)

- 意味  
下位モジュールをテストするときに、呼び出す上位モジュールの代用品。
- 役割  
本物の上位モジュールが未完成でも、テスト対象を呼び出せるようにする。

👉 下位から上位へテストするボトムアップテストで使う。

### ◆ インспекション (Inspection)

- 意味  
フォーマルな (形式的な) レビュー手法。
- 特徴
  - 事前にレビュー対象 (設計書やソースコード) を配布し、参加者が準備してから会議でチェックする
  - チェックリストを使い、誤り・欠陥を体系的に洗い出す
  - ファシリテータや記録係など 役割を分担 して行う
  -

### ◆ ウォークスルー (Walkthrough)

- 意味  
開発者 (作成者) がレビュー対象を 説明しながら順に読み進める 手法。
- 特徴
  - 開発者が主導して「ここでこう処理します」と説明
  - 参加者が疑問点や誤りを指摘する
  - 比較のカジュアルで、教育・知識共有にも役立つ

## ◆ ラウンドロビン (Round Robin)

- 意味  
レビュー参加者が順番にレビュー対象を読んでいく方式。
- 特徴
  - 複数人が交代で読み進めることで、全員が積極的に参加できる
  - 主に小規模開発や教育的な場で利用

## ◆ 3つの比較

手法	主導	形式	特徴	適用シーン
インスペクション	ファシリテータ（進行役）	フォーマル	役割分担・チェックリスト必須・欠陥発見に特化	大規模開発、品質重視
ウォークスルー	作成者	カジュアル	作成者が説明しながら確認	教育、初期段階の確認
ラウンドロビン	参加者全員	中間	順番に読んで確認	小規模開発、参加意識向上

## ◆ ホワイトボックステストの5つの網羅性

### 1. 命令網羅 (Statement Coverage)

- プログラム中の **すべての命令（文）** を少なくとも **1回** 実行する
- 最も基本的な網羅基準
- 例：if 文の条件が真のケースだけ通れば、偽のケースを通さなくても命令網羅は達成

### 1. 分岐網羅 (Branch Coverage / Decision Coverage)

- **全ての分岐 (true / false)** を少なくとも **1回** 通す
- if 文や switch 文の全ての選択肢を実行する
- 命令網羅より強い基準

### 1. 条件網羅 (Condition Coverage)

- 複合条件を構成する **各条件式を true / false にする**
- 例：if (A && B) の場合、A と B をそれぞれ true/false にする

### 1. 分岐／条件網羅 (Decision/Condition Coverage)

- **分岐網羅 + 条件網羅を組み合わせたもの**
- 各条件が true/false になること、かつ分岐全体も true/false になることを確認

### 1. 条件組合せ網羅 (Condition Combination Coverage / Multiple Condition Coverage)

- **複合条件のすべての組み合わせを実行する**
- 例：if (A && B) の場合 → (A=true,B=true), (A=true,B=false), (A=false,B=true), (A=false,B=false) の4通りを網羅

- 最も厳密な基準（テストケースが爆発的に増える）

総バグ数は  $(N_A * N_B) / N_{AB}$