モジュールの設計

FU14 ソフトウェア工学概論

第8回

吉岡 廉太郎

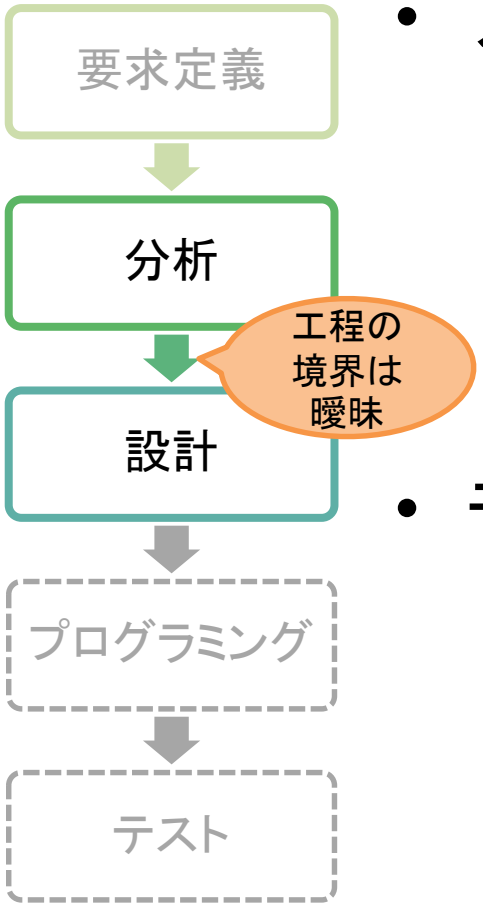
前回の内容

- アーキテクチャーの評価手法
  - 最適なアーキテクチャーを求める
  - 複数の案を比較・評価する
- 設計書
  - どのような検討・判断を行ったかの記録である
  - 妥当性を説得しなければならない
- アーキテクチャーのバリデーションと検証

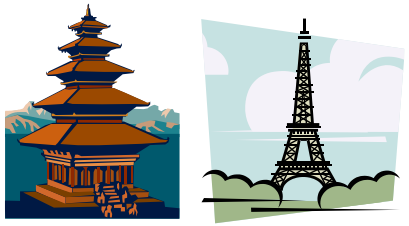
今日の内容

- モジュールの設計
  - 詳細設計とも呼ばれる
  - ソフトウェアユニットをモジュールに分割し、定義するプロセス
  - プログラムの作成に必要な情報をすべて決定する
- 設計の原則
  - 設計上の選択肢を考えるガイドライン

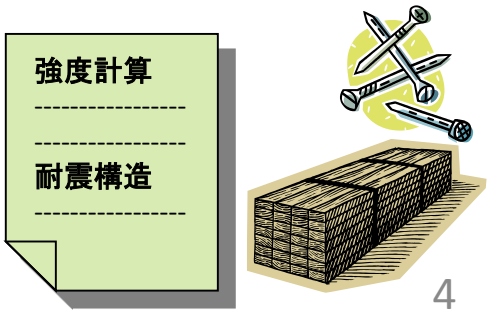
モジュール設計のプロセス



- アーキテクチャの設計
  - 全体の設計をソフトウェアユニットに分解し、機能を割り当てるおおまかな計画
  - 設計の指針：様式



- モジュールの設計
  - ソフトウェアユニットの詳細を決める
  - 設計の指針：なし



設計の原則

- システムの機能をモジュールに分解するためのガイドライン
- 原則が規準を定める
  - システムの分割単位
  - 各モジュールが公開／隠蔽する情報
- 主要な原則
  1. モジュール性
  2. インターフェース
  3. 隠蔽
  4. インクリメント開発
  5. 抽象化
  6. 一般化

“良い設計”が備える特徴とも言える

5.2.2

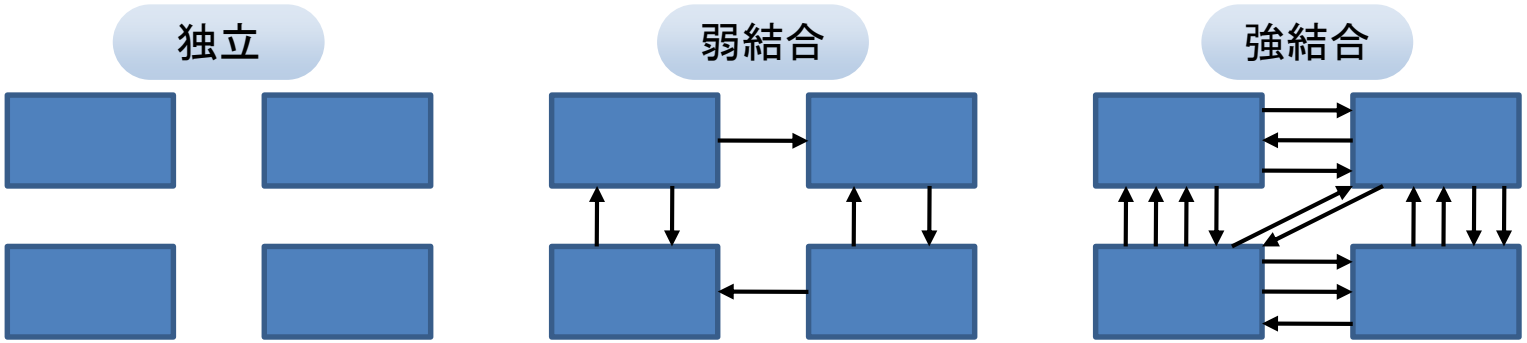
1．モジュール性

- システムの要素を、機能や振る舞いの単位に分離するという原則
  - 要素ごとに隔離することで問題発見が容易になる
  - 関心事の分離（separation of concerns）
    関心事＝機能、振る舞い、目的
- 極端な場合、各モジュールに一つの目的を持たせる
  - 第三者でも理解が容易で開発もしやすい
  - 問題の原因を見つけやすい
  - システムが変更しやすくなる
- 分離の度合いをはかる目安
  - 結合度と強度

5.2.3

結合度（coupling）

- 強結合
  - 二つのモジュールが互いに強く依存している状態
- 弱結合
  - モジュールに依存性はあるが相互接続が弱い状態
- 独立：モジュール間の相互接続が全くない状態

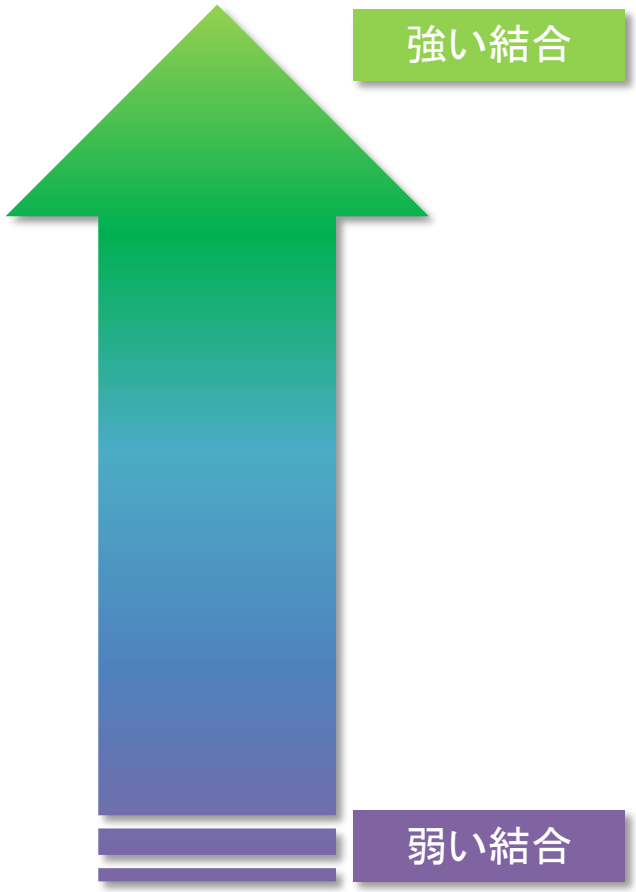


結合度

- モジュール同士の依存性を表す尺度
  - 参照の数
  - 受け渡されるデータの量
  - 制御の量
- 一般的には、完全に独立したモジュールのみでシステムを作ることはできない
  - 独立したものから、結合度が強いものまでの分布としてとらえる

結合の種類

- 内容結合（content coupling）
- 共有結合（common coupling）
- 制御結合（control coupling）
- スタンプ結合（stamp coupling）
- データ結合



強度・凝縮度（cohesion）

- 強度はモジュールの内部要素間の依存性を表す

強度の種類

偶発的（最悪のレベル）

- パーツ間に関連性がない場合に発生する。関連のない機能、プロセス、データが一つのモジュールに存在している。

論理的

- パーツ間の関連性がコードの論理構造を共有するというだけの場合に発生する

時間的

- モジュールの機能やデータが実行時に同じタイミングで使われるという場合に発生する。

手続き的

- 時間的強度に加え、各機能が関連したアクションや目的と関係がある場合に発生する。

強度の種類

通信的

- 同じデータに対する操作を行う場合に発生する

機能的（もっとも理想的）

- 一つの機能に必要（必須）な要素が一つのモジュールに含まれている場合に発生する

情報的

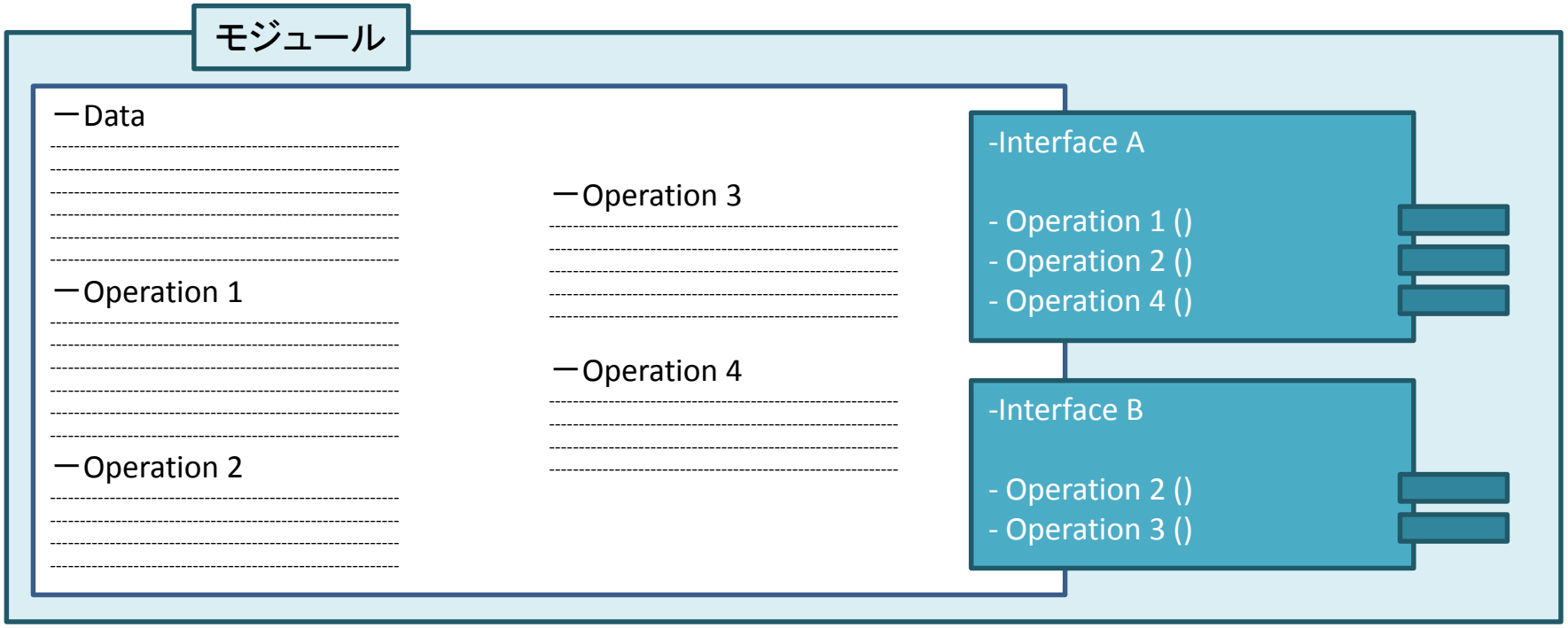
- 機能的強度をデータ抽象化やオブジェクト指向設計に適応した場合に発生する

2. インタフェース

- インタフェース
  - そのソフトウェアユニットが提供するサービスと、それをどのように要求できるかを定義する
  - 例えば、関数の名前、引数、戻り値など
- サービスを提供するための前提条件も定義する
  - 必要なリソース、操作の順序など
- そのユニットが環境に提供する機能とともに、環境に要求することも定義する

インタフェースの構成

- ソフトウェアユニットは複数のインタフェースを持てる
  - 環境に対する要求の違い
  - 提供するサービスの違い



インタフェースの定義

- インタフェース仕様
  - ソフトウェアユニットの外見上の属性を表す
- そのソフトウェアユニットを正しく利用するのに十分な情報を提供しなければならない
  - 目的
  - 事前条件
  - プロトコル（手順）
  - 事後条件
  - 品質属性

5.2.4

3. 情報隠蔽

- ソフトウェアユニットの外見を固定する
  - 将来変更される可能性のある設計上の都合を、外部からは見えなくする
  - インタフェースとインタフェース仕様を用いて、各ユニットを外部向け属性の集合として表す
- この原則に則った場合のモジュールの強度
  - データ表現を隠蔽したモジュールは情報的強度を持つ
  - アルゴリズムを隠蔽したモジュールは機能的強度を持つ
- 情報隠蔽の最大の利点
  - 各ソフトウェアユニットが弱結合になる

5.2.5

OO設計における情報隠蔽

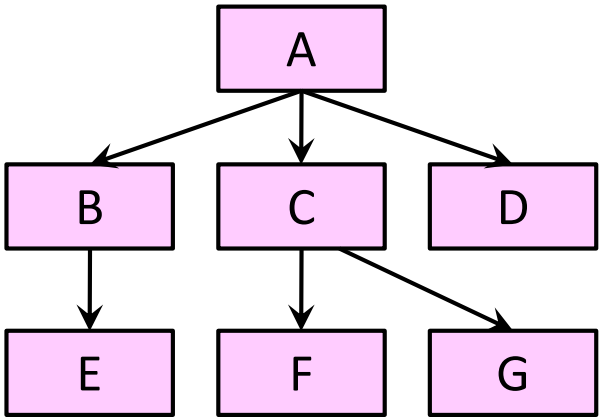
- OO（Object Oriented）設計では、システムをオブジェクトとその抽象型に分解する
  - 従って、各オブジェクト内のデータ表現は隠蔽される
  - インタフェースで定義されたアクセス関数を通してそのオブジェクトのデータを取得できる
  - データ表現の変更によるシステム全体への影響が少ない
- データ表現以外にもアルゴリズムを隠蔽する
- 隠蔽によって結合を完全に解決することはできない
  例）名前の変更
  - 特定の意味をもつオブジェクトでは影響を免れない
  - 一般性の高いオブジェクトであれば回避する方法もある
    →デザインパターン

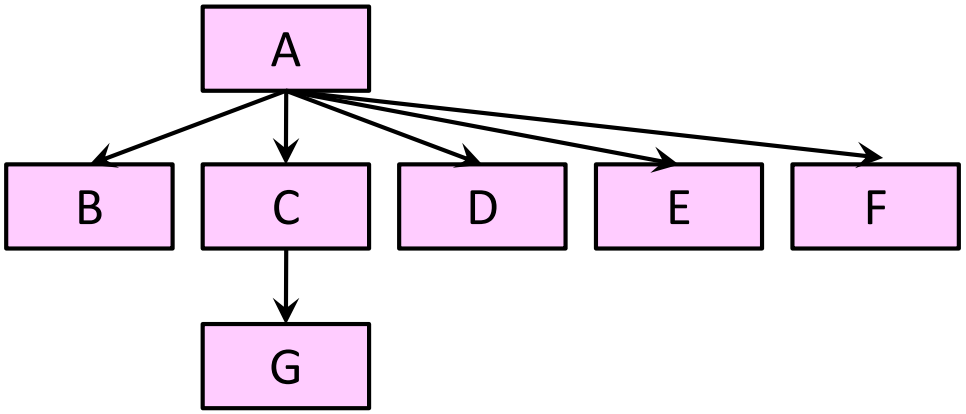
4. インクリメント開発

- 段階的な開発を可能にする設計
  - ソフトウェアユニット間の依存性に着目
  - 開発しやすい設計を目指す
- ユニット同士の参照関係を明らかにする
  - 依存しあうユニットを関係づける（ユースグラフ）
  - 開発とテストを段階的に進めることができるシステムの範囲を見つけられる

ユースグラフの例

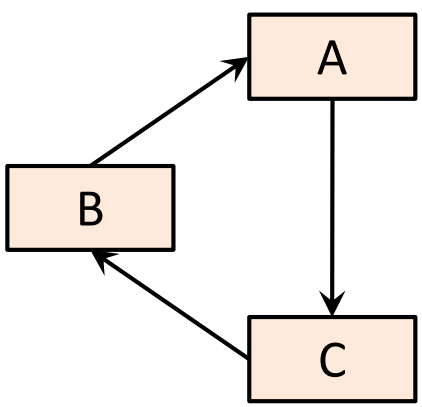
- 論理入力
  - あるソフトウェアユニットを利用するユニットの数
- 論理出力
  - あるソフトウェアユニットが利用するユニットの数

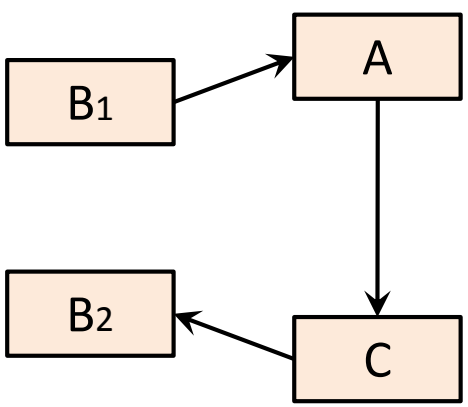


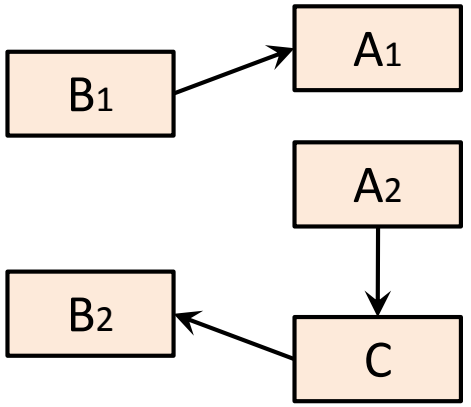


ユースグラフの例（2）

- サンドウィッチ
  - ユースグラフ内のサイクルを切る方法
  1. サイクルの一部になっているユニットを2つに分解する（新しいユニットは依存性を持たないようにする）
  2. この操作を繰り返し適用することで、相互依存や強連結のユニットも解決できる



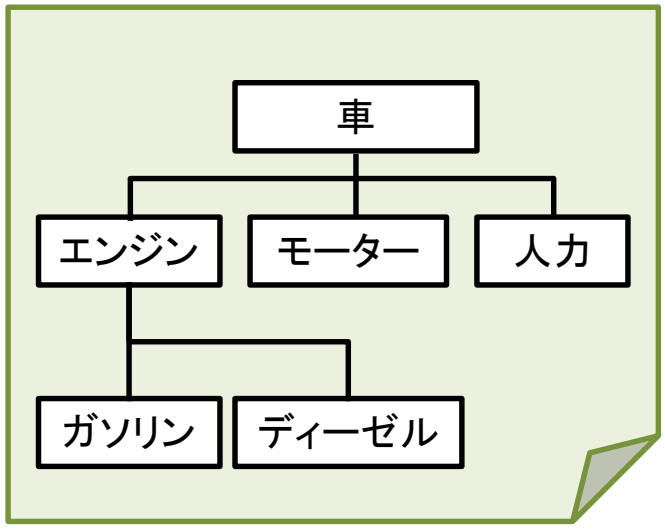




5. 抽象化

- 詳細を省くことによって他の詳細に焦点を当てるモデルや表現方法

- 何をどの程度省くかが肝心
  - しかし、一律の定義はない
  - 目的と対象によって異なる



抽象化の効果

- 例）配列　L　の要素をソートする機能の設計を考える

1. 最も高次元の記述：

```
Lを昇順に並び換える
```

2. 第二段階の記述：並び替えのアルゴリズムまで示す

```
DO WHILE I is between 1 and (length of L) – 1 :
  Set LOW to index of L(I), ……. ,  L (length of L)の最小値
  Interchange L(I) and L(LOW)
ENDDO
```

抽象化の効果(2)

3. 第三段階の記述:詳細な実装を示す

```
DO WHILE I is between 1 and (length of L) – 1
   Set LOW to current value of I
   DO WHILE J is between I+1 and (length of L)
     IF L(LOW) is greater than L(J)
         THEN set LOW to current value of J
     ENDIF
   ENDDO
   Set TEMP to L(LOW)
   Set L(LOW) to L(I)
   Set L(I) to TEMP
ENDDO
```

1～3の記述はそれぞれに長所・短所があることが分かります。

6. 一般化

- ソフトウェアユニットを活用しやすくなるような設計を目指す
  - 再利用を念頭にする
- 各ユニットを多くの状況で繰り返し利用する
  - 内容依存のある情報をパラメータ化する
  - 事前条件を無くす
  - 事後条件を簡素化する

一般化の度合い

- 例）関数のインタフェース

関数 SUM: INTEGER;
事後条件: returns 3つのグローバル変数の和

関数 SUM (a, b, c: INTEGER) : INTEGER;
事後条件: returns 引数の和

関数 SUM (a[] : INTEGER; len: INTEGER) : INTEGER
事後条件: returns 配列 a の要素 1..len の和

関数 SUM (a[] : INTEGER) : INTEGER
事後条件: returns 配列 a の全要素の和

本日のまとめ

- モジュール設計は、ソフトウェアユニットの詳細を決めるプロセス
- アーキテクチャー設計で決めたソフトウェアユニットを分解して、モジュールにしていく
- アーキテクチャーのように、設計指針となる様式はない
  - 解決策はより個別的
- 設計の原則：モジュールを検討する際のガイドライン
  - 6つの原則
- 常に設計を見直しながら進める
  - 設計書を書きながら
  - リファクタリング

次回講義の事前学習：
5.6.1, 5.6.2, 5.6.4