2023年度
ソフトウェア工学
ソフトウェアモジュール

2023年12月11日

渥美 紀寿 (情報環境機構)

京都大学

講義スケジュール

- 01回 (10/02 (Mon))
  ソフトウェア工学概説
- 02回 (10/16 (Mon))
  ソフトウェアプロセスモデルとライフサイクル
- 03回 (10/23 (Mon))
  要求分析
- 04回 (10/30 (Mon))
  システム設計
- 05回 (11/06 (Mon))
  品質管理とプロジェクト管理
- 06回 (11/13 (Mon))
  実践的ソフトウェア工学：クリティカルシステム
- 07回 (11/20 (Mon))
  実践的ソフトウェア工学：アクセシビリティ
- 08回 (11/27 (Mon))
  実践的ソフトウェア工学：ビジネス創生
- 09回 (12/04 (Mon))
  実践的ソフトウェア工学：
  契約としての要求仕様
- 10回 (12/11 (Mon))
  ソフトウェアモジュール
- 11回 (12/18 (Mon))
  ソフトウェアテスト
- 12回 (12/25 (Tue))
  形式手法
- 13回 (01/15 (Mon))
  ソフトウェアメトリクス
- 14回 (01/22 (Mon))
  ソフトウェアの保守と発展

第01〜05回: 伊藤, 第06〜09,回: 星野, 第10〜14回: 渥美

参考図書

- 鯵坂 恒夫著: ソフトウェア工学入門, サイエンス社

ソフトウェアモジュール

モジュールとは

- (工業製品などで)組み換えを容易にする規格化された構成単位
- 建造物などを作る際の基準とする寸法
  また，その寸法の集合西洋古典建築では円柱の基部の直径または半径，日本建築では柱の太さまたは柱と柱の間の長さ
- 歯車の歯の大きさを表す値
  ミリメートルで表したピッチ円の直径を歯数で割ったもの
  (三省堂 大辞林)
- ソフトウエアやハードウエアを構成する部分のうち，独立性が高く，追加や交換が容易にできるように設計された部品

ソフトウェアモジュール

- 独立性が高く，追加や交換が容易にできるよう設計された部品
- モジュールを利用してシステムを構築
  - 大規模で複雑なソフトウェアシステムを
  - どのような部品を
  - どのように組み合わせて
  - 構成するか

ソフトウェアモジュールの例

- 手続き指向
  - 関数，データ型
- オブジェクト指向
  - クラス
- アスペクト指向
  - アスペクト

⇒ 分割の基準・観点が異なる

モジュール化

- 個々に名前がつけられ識別可能なコンポーネントに分割すること
  - 扱いやすい小さい問題に分割
  - 単一モジュールで構成された巨大なプログラムを理解することは難しい
- 要素間の関係
  - モジュール内： 相互依存性を持つ
  - モジュール間： 独立している
- 目的： 複雑性の緩和
- ゴール： プラグ&プレイ

モジュール化の目的

プログラムを理解する際に，同時にたどり続けなければならない要因の数を減らす

- 適切にモジュール化すると
  - モジュール化された部分がはっきり定義される
  - 境界はインタフェースによって結合される
    - モジュールで必要なデータが明確に
- 適切にモジュール化しないと
  - モジュールの機能が複雑
  - モジュール間の結合が複雑

モジュールの独立性

- 良いプログラム設計
  - プログラムを単に階層構造にモジュール化するのではない
  - 各モジュールが他のすべてのモジュールからできるだけ独立するように，プログラムを階層構造にモジュール化する
- 独立性を高めるために
  - モジュール間の関連性(結合度)を最小に
  - モジュール内の部品の関連性(凝集度)を最大に

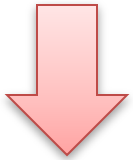
モジュールの大きさ

- モジュールが大きすぎると
  - 同時に理解する要素が多くなり，理解することが難しくなる
- モジュールが小さすぎると
  - 個々のモジュールは容易に理解できるが，他のモジュールとの関連が多くなり，全体を把握することが難しくなる

巨大なモジュール

大規模なシステムを数個の巨大なモジュールに分割した

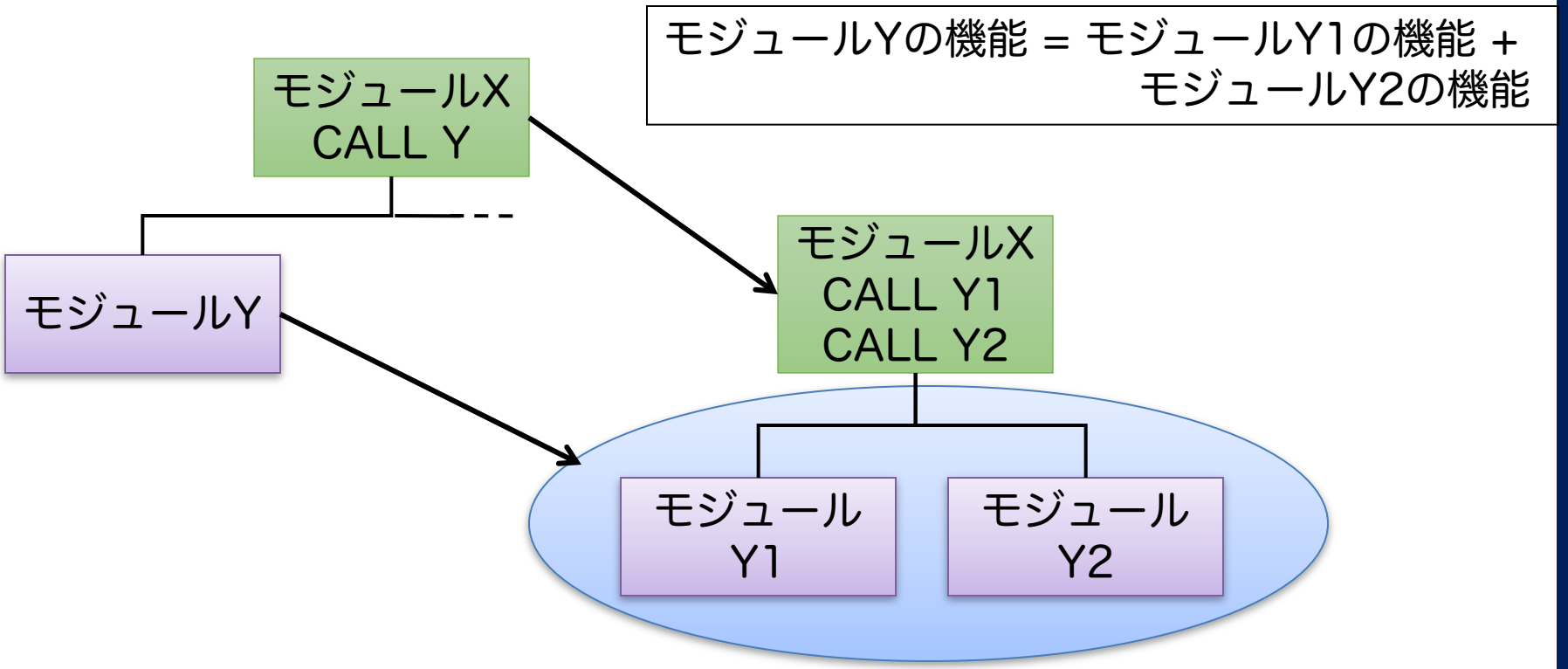
- モジュール間の独立性が高ければ問題無い？



- モジュールの開発が困難
- 他のモジュールの追加，交換が困難
- 保守が困難

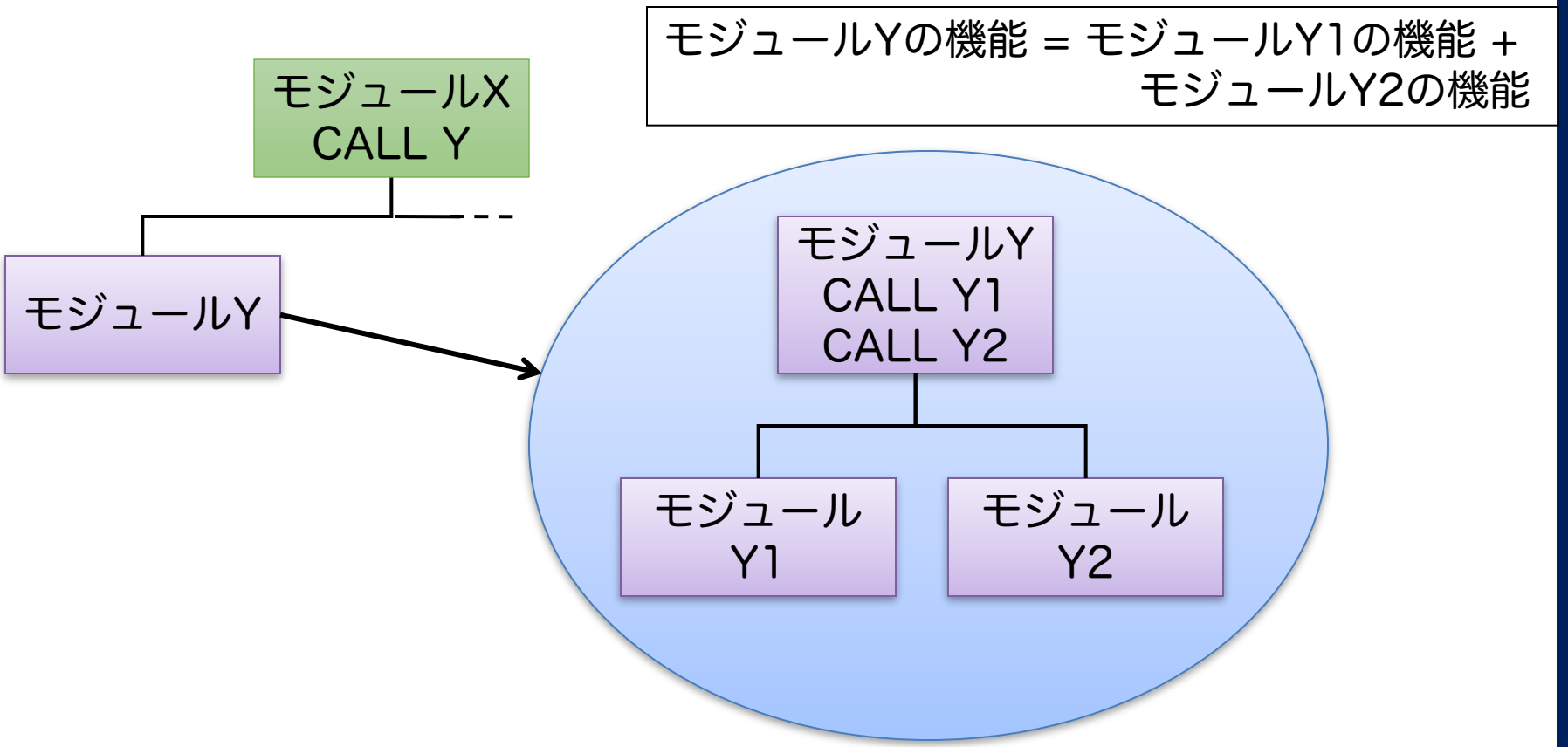
モジュール分割方法 (幅の分割)

- 機能の大きさからの制約に依存して行われる
- 処理手順を手続き的に保ち，並列的に構成する



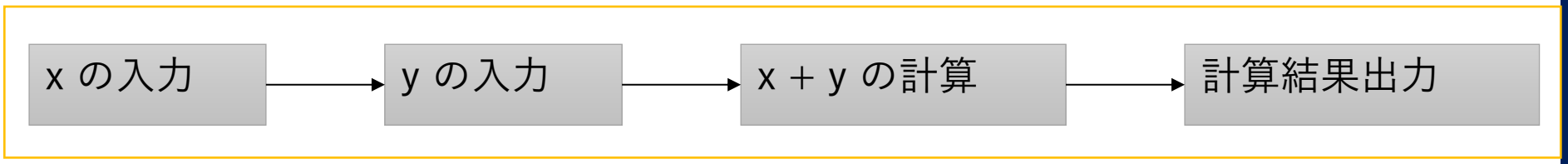
モジュール分割方法 (深さの分割)

- 対象となるモジュールを段階的に詳細化
- 個々のモジュールが持つ機能の独立性を確保



モジュール分割の例 (分割なし)

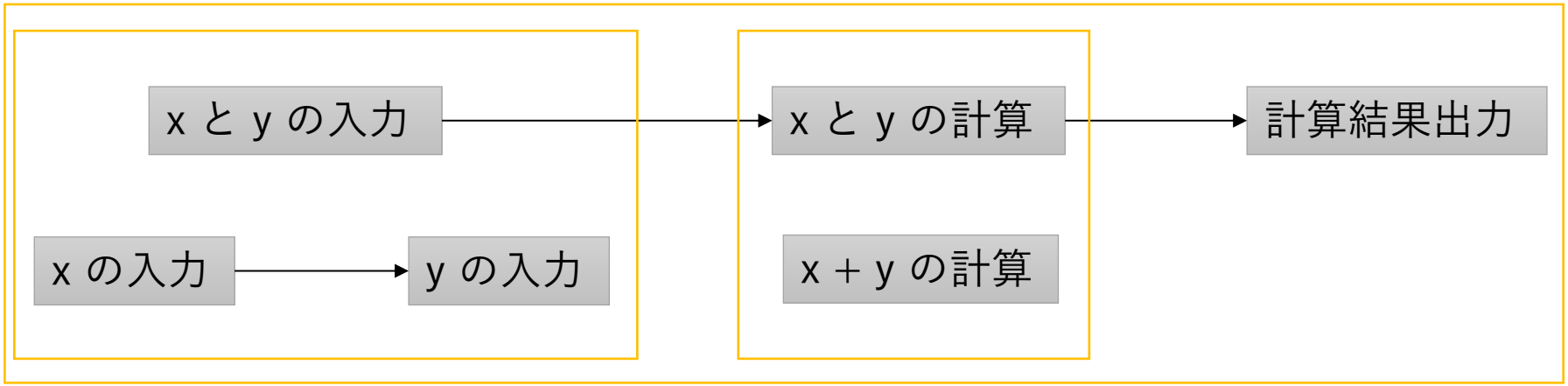
```
#include <stdio.h>
int main(void) {
    int x, y, ans;
    printf("input x: ");
    scanf("%d”, &x);
    printf("input y: ");
    scanf("%d”, &y);
    ans = x + y;
    printf("ans: %d¥n", ans);
    return 0;
}
```



モジュール分割の例 (幅の分割)

```c
#include <stdio.h>
int calc(int x, int y) {
    return x + y;
}
void input(int *x, int *y) {
    printf("input x: ");
    scanf("%d", x);
    printf("input y: ");
    scanf("%d", y);
}
```

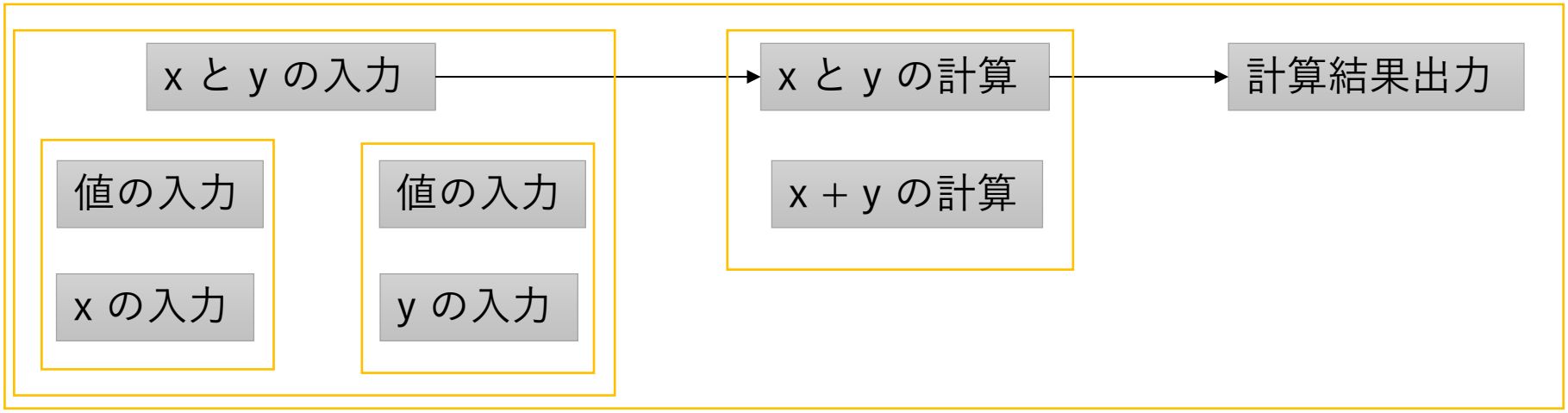
```c
int main(void) {
    int x, y;
    input(&x, &y)
    ans = calc(x, y);
    printf("ans: %d¥n", ans);
    return 0;
}
```



モジュール分割の例 (深さの分割)

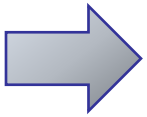
```c
#include <stdio.h>
int calc(int x, int y) {
    return x + y;
}
void input(char c, int *x) {
    printf("input %c: ", c);
    scanf("%d", x);
}
```

```c
void input_xy(int *x, int *y) {
    input('x', x);
    input('y', y)
}
int main(void) {
    int x, y;
    input_xy(&x, &y)
    ans = calc(x, y);
    printf("ans: %d¥n", ans);
    return 0;
}
```



モジュール分割

- 深さの分割をし過ぎると
  - 縦に深い構造に
  - 制御が複雑に
- 幅の分割をし過ぎると
  - 横に広い構造に
  - 機能の独立性が不明確に



モジュールの独立性を保ちながら，
両極端にならないように中間的
構造へ最適化

構造化技法とオブジェクト指向

- 構造化技法
  - システム全体の機能を段階的詳細化し，階層的に組み立てる
  - 機能とデータを別々に実現

- オブジェクト指向
  - システム全体をオブジェクト間の相互作用で表現
  - オブジェクトとはデータと処理を一纏めにしたモジュール（カプセル化)

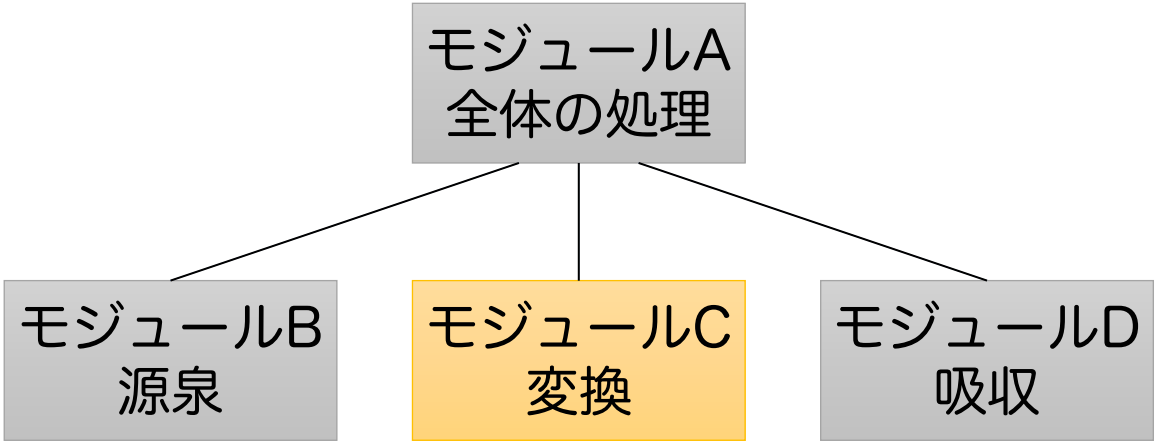
構造化技法における モジュール分割

構造化設計におけるモジュール分割技法

- 源泉/変換/吸収分割 (STS分割)
- トランザクション分割 (TR分割)
- 共通機能分割

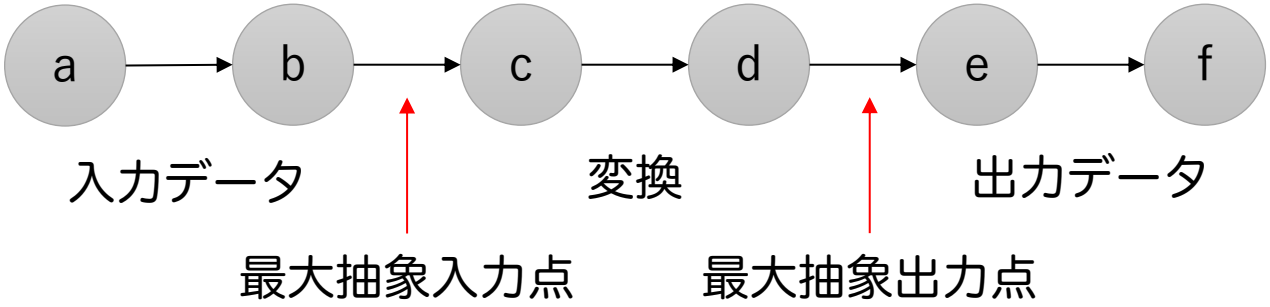
源泉/変換/吸収分割

- STS分割
  (Source/Transform/Sink decomposition)
- 機能を入力から出力への変換とみなして分割



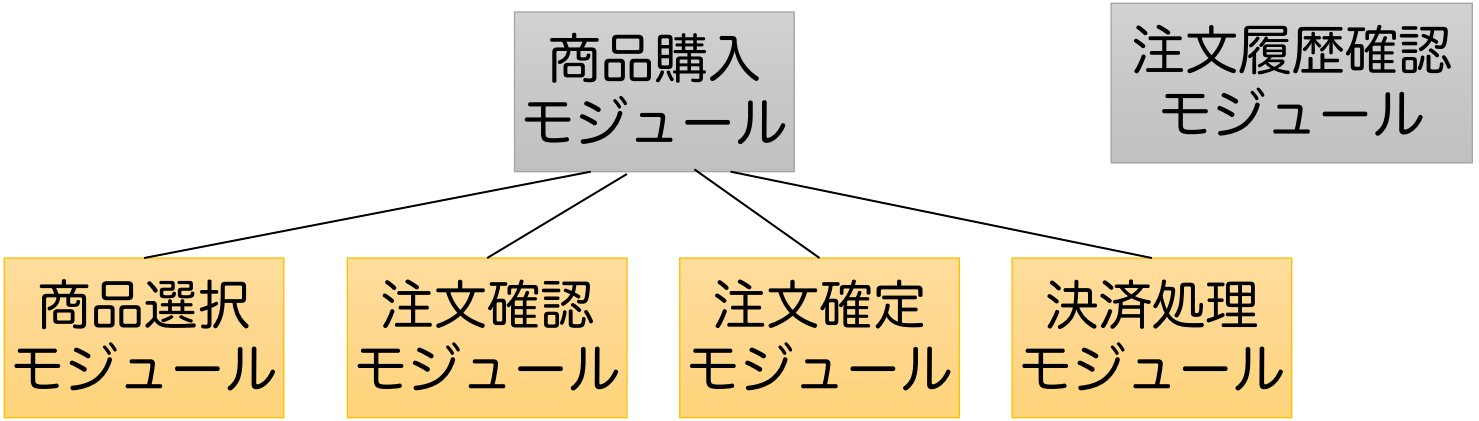
源泉/変換/吸収分割の手順

- データフロー図において主要なデータの流れを抽出
- 最大抽象点の発見
  - 最大抽象入力点
    - 入力側から入力データを順方向にたどり，
      入力データとはいえなくなる点
  - 最大抽象出力点
    - 出力側から出力データを逆方向にたどり，
      出力データとはいえなくなる点



トランザション分割

- TR分割 (transactional decomposition)
- トランザクション処理ごとに分岐先モジュールを定義
  - トランザクション
    データに対する切り離せない一連の処理の単位
- ex) オンラインショッピングでの商品購入と
    注文履歴確認



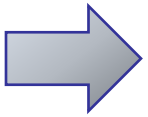
共通機能分割

- functional decomposition
- 複数のモジュールに含まれる共通の従属機能を取り出して定義

- ex)
  ログ出力
  エラーメッセージ出力

モジュール間の関係

- 上位モジュールと下位モジュールは，データの受け渡しにより関連付けられる
- 上位モジュール
  - 処理すべきデータを下位モジュールへ受け渡す
- 下位モジュール
  - 受け取ったデータを手続きに従って処理し，結果を上位モジュールに返す



モジュール分割によって，
入出力インタフェースが必要に

段階的詳細化

- 与えられた問題に対して，徐々に細部の手順を定める

- 自然言語で大まかな処理を表す
- 処理を詳細化する
- 詳細化された個々の処理に対して 2 を繰り返す
- プログラミング言語に置き換える

プログラムの段階的作成

- 例題

  2を最初の素数として，はじめの1,000個の素数の表を出力する

オブジェクト指向における モジュール分割

オブジェクト指向の視点

- 現実世界のものやもの同士の関係を表現
- ものをオブジェクトとして捉える
- オブジェクト間の繋りをメッセージパッシングとして捉える
- 類似したオブジェクトをクラスとして抽象化

オブジェクト指向の特徴

- 抽象データ型
  - データ(属性)とそれに対する操作(メソッド)群でデータを表現
- カプセル化による情報隠蔽
  - 不要なデータを外部に見せない
  - 必要な操作のみ公開
- 継承
  - データや操作を引き継ぐ
- 多相性
  - 操作の受け手によって処理内容が変わる

クラスとインスタンス

- クラス
  - データ (フィールド・属性)
  - メソッド (操作)
- インスタンス (オブジェクト)
  - クラスから生成した要素
  - 実行される実体

モジュール化：何のモジュール？

```
struct {
    int a[100];
    int b;
} s;
```

```
int f(int c, int d, int *e) {
      int r = 0;
      switch(d) {
          case 0:
              if (s.b < 99) {
                  s.b++;
                  s.a[s.b] = c; r = 1;
              }
              break;
          case 1:
               if (s.b > 0) {
                   *e = s.a[s.b];
                   s.b--; r = 1;
              }
              break;
      }
      return r;
}
```

モジュール化：何のモジュール？

```
struct {
    int data[MAX_SIZE];
    int top;
} stack;
```

ちょっとだけ改良

```
int stack_operator(int push_data, int op,
                                  int *pop_data) {
    int result = 0;
    switch(op) {
        case PUSH:
            if (stack.top < MAX_SIZE - 1) {
                stack.top++;
                stack.data[stack.top] = push_data;
                result = 1;
            }
            break;
        case POP:
            if (stack.top > 0) {
                *pop_data=stack.data[stack.top];
                stack.top--;
                result = 1;
            }
            break;
      }
      return result;
}
```

モジュール化：良い例

- スタックを表すデータ型

```
typedef struct {
    int data[MAX_SIZE];
    int top;
} stack;
```

- スタックに対する pop 操作
  - スタック *s からデータを pop し，その値を *data に格納
  - スタックが空の場合 0 を，そうでなければ 1 を返す
  - `int pop(stack *s, int *data);`
- スタックに対する push 操作
  - スタック *s にデータ data を push する
  - スタックがあふれた場合　0 を，そうでなければ 1 を返す
  - `int push(stack *s, int data);`

モジュール化：Stack の例

```
struct {
    int data[MAX_SIZE];
    int top;
} stack;
```

```
int push(stack *s, int data) {
    int result = 0;
    if (s->top < MAX_SIZE - 1) {
        s->top++;
        s->data[s->top] = data;
        result = 1;
    }
    return result;
}
```

```
int pop(stack *s, int *data) {
    int result = 0;
    if (s->top > 0) {
        *data=s->data[s->top];
        s->top--;
        result = 1;
    }
    return result;
}
```

オブジェクト指向言語によるStack

Stack.java

```
class Stack {
    private final int MAX_SIZE = 1024;
    private int[] data;
    private int top;
    Stack() {
         data = new int[MAX_SIZE];
         top = 0;
    }
    int push(int d) {
        if (top > MAX_SIZE) return 0;
        data[top] = d;
        top++;
        return 1;
    }
    int pop() {
        if (top == 0) return 0;
        top--;
        return data[top];
    }
    boolean isEmpty() {
        if (top > 0)
             return false;
        else
             return true;
     }
}
```

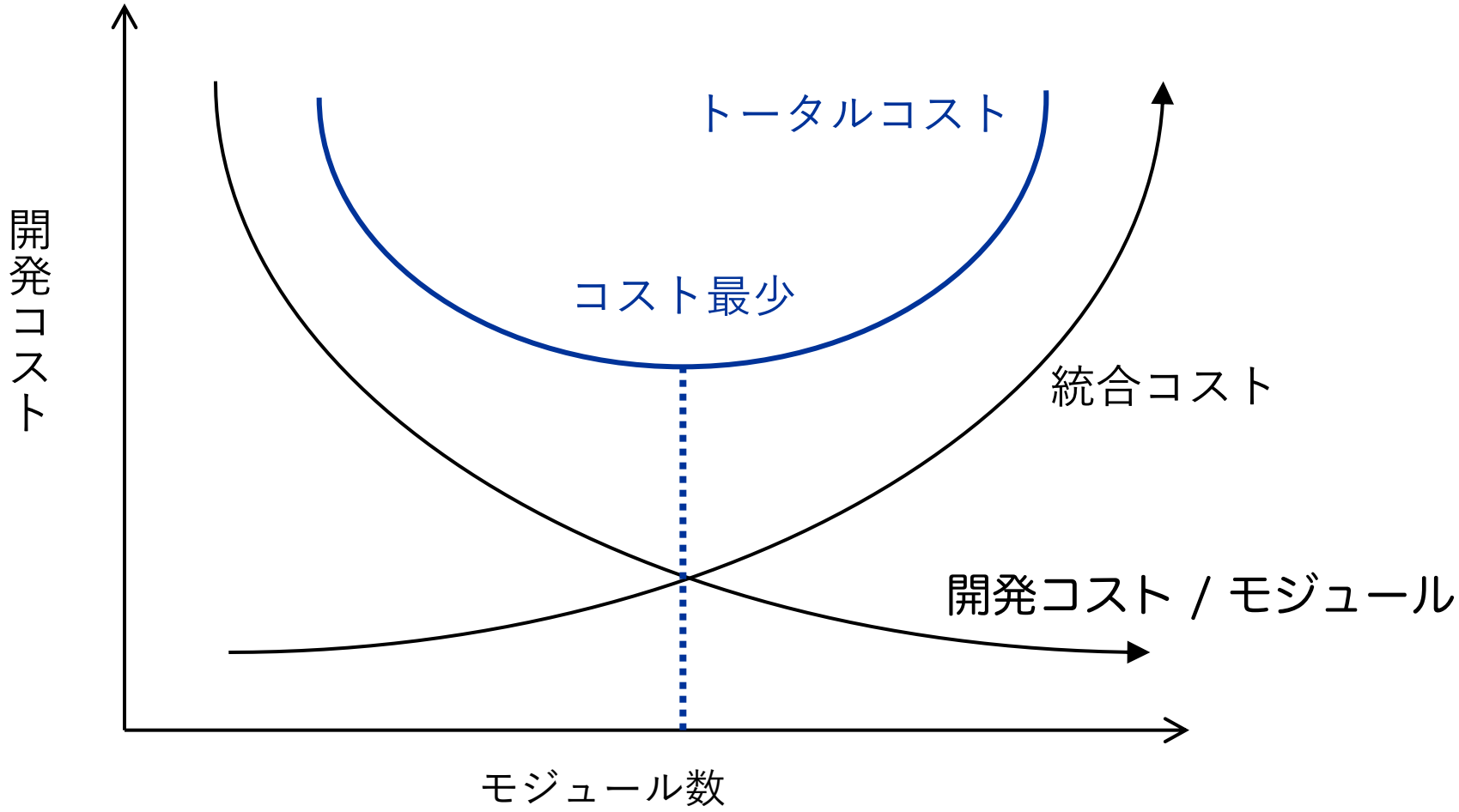
モジュール化されたソースコード

- 識別子 (関数，変数，データ型など)に意味のある名前を付ける
- 処理を名前が付けられる単位でモジュール分割

モジュール化のメリット と デメリット

- メリット
  - 条件付き「適切にモジュール化を行った場合」
  - 構成要素間の調整や擦り合わせにかかる コスト（トキ，ヒト，モノ，カネ）の削減
  - モジュールの再利用
  - システム全体に対する変化の影響の局所化
  - 分業の促進
- デメリット
  - 冗長性確保によるシステム・パフォーマンスの低下
  - モジュールをまたがる変化への対応が困難

モジュール開発コスト vs 統合コスト

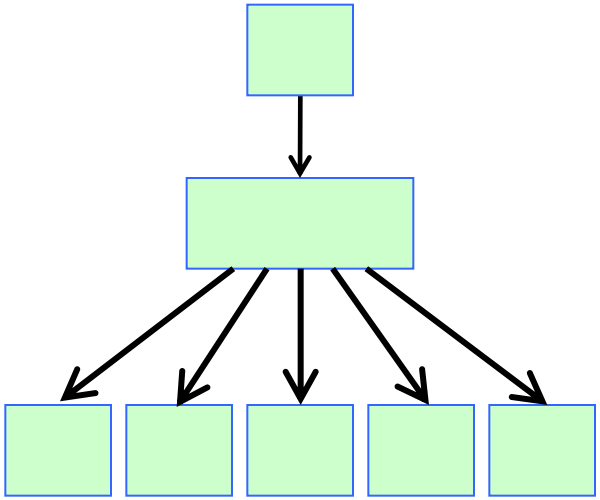


失敗事例

- 肥大化
- なんでも屋
- スパゲッティ
- 無秩序な構造
- 物理駆動
- 余計な関係

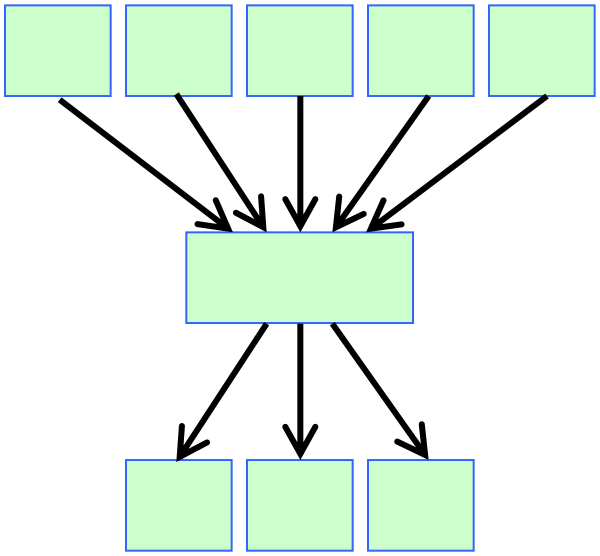
肥大化

- 1つのモジュールに多くの機能が入っている
  - モジュールの複雑度が高い
  - ファンアウト数が多い
  - 手順的，時間的凝集度と低い
- 内部の実際の処理の流れが中心となっている



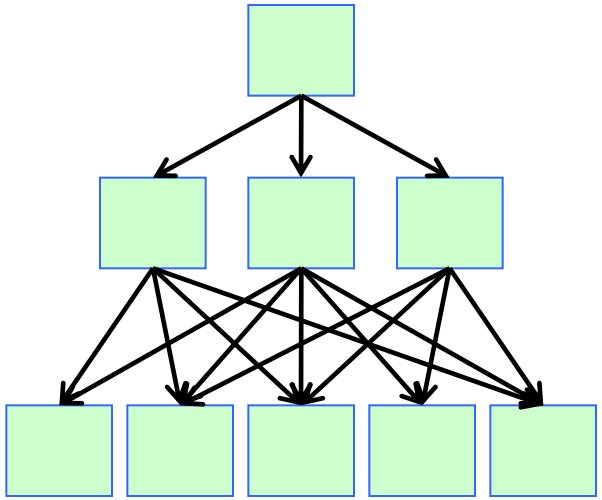
なんでも屋

- 複数のモジュールから呼び出されている
  - 機能を選択する引数があり，内部的に多くのデータを抱えている可能性がある
- モジュール単一の目的を持っていれば問題ない



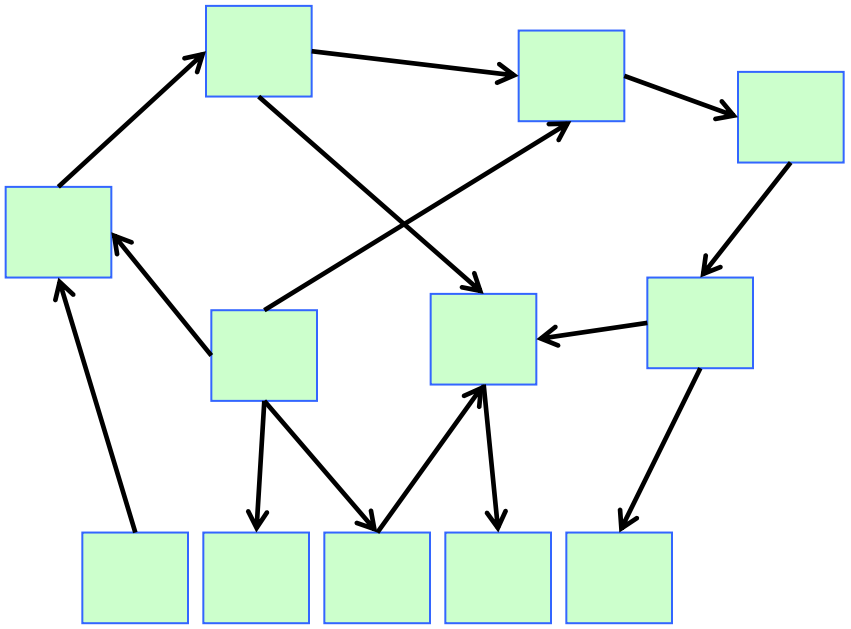
スパゲッティ

- モジュール間の呼び出し関係が複雑に絡み合っている
  - それぞれのモジュールの凝集度が低い
  - システム形状のバランスが悪い
  - 共通となる処理がまとめられていない



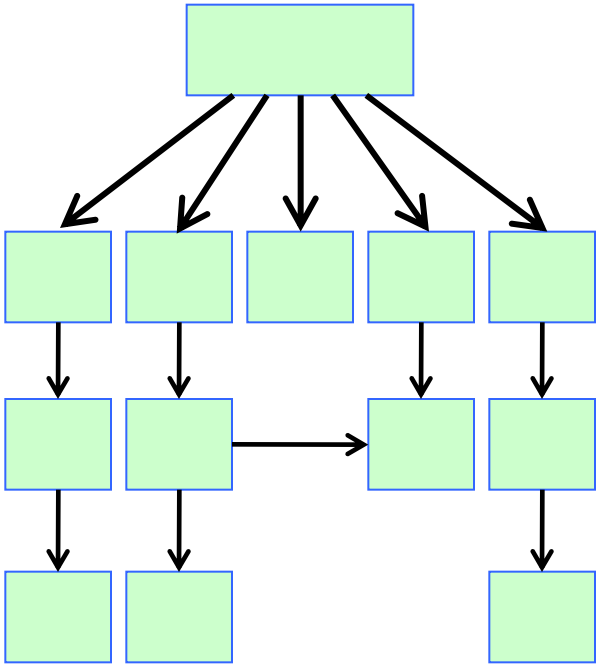
無秩序な構造

- モジュール間の関係の意味が見いだせない
  - 制御が中心の流れになっている
  - 凝集度が低い
  - 各階層で抽象度が統一されていない



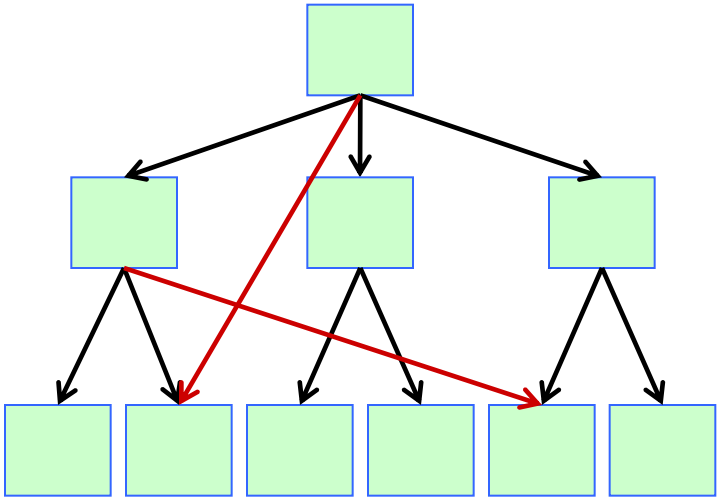
物理駆動

- モジュールの流れが複数列になって，
  モジュール間の呼び出しが発生している
  - 機能分析ではなく，ハードウェアのイベントが機能を支配している



余計な関係

- 全体としての調和がとれていない関係がある
  - 全体構造を決めた後に仕様の不備から余計な関係が生まれてしまう
  - 要求分析（要求モデリング，分析モデリング）に不備がある



モジュールの再利用

- 適切にモジュール化すれば，様々なプログラムで利用可能
  - ライブラリ (関数，クラス)
- 汎用的に利用できるようにモジュール化すれば再利用しやすいが…
  - 何でもできるモジュールは使えない
- 再利用可能なモジュールを見付けるのは困難

モジュールの再利用例

- ライブラリ
  - 言語ごとに標準で用意されたライブラリ
  - 特定の用途向けに開発されたライブラリ
- コンポーネント
  - COM，.NET コンポーネント，JavaBeansなど
- フレームワーク
  - Apache Struts，Ruby on Rails (Webフレームワーク)
  - .NET Framework
  - JavaFX，Qt，Quartz (GUIフレームワーク)
- サービス

まとめ

- 適切なバランス (幅，深さ)
- 適切な抽象度
- 適切な依存関係
- 分割する粒度
  - 名前が付けられる
- モジュール分割技法
  - 構造化技法
  - オブジェクト指向