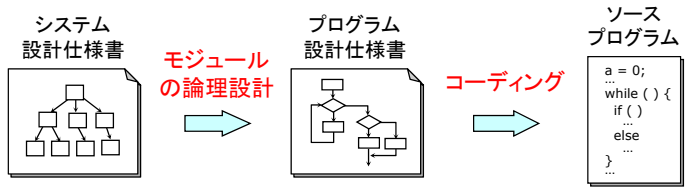
モジュール設計

モジュールの設計



- システム設計仕様書(system design specification)
  - ✓ モジュール構成図
  - ✓ モジュール機能仕様書
  - ✓ モジュールインタフェース仕様

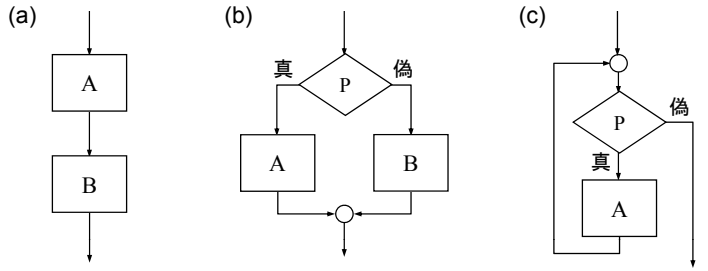
  } モジュールの外部特性の定義
- プログラム設計仕様書(program design specification)
  - ✓ モジュールの論理設計: 個々のモジュールの内部構造を決定
- ソースプログラム(source program)
  - ✓ コーディング: 具体的なプログラミング言語による記述

モジュールの論理設計

- モジュールの内部論理の設計 ≅ プログラミング
  - ✓ 従来:
    メモリの使用領域と処理時間の最小化
    → 生産性向上の阻害
  - ✓ 現在:
    理解しやすいプログラムの作成
    → 構造化プログラミング(structured programming) [Dijkstra]
      - 論理(アルゴリズム)の構造化
        (1) NSチャート, PAD
        (2) プログラム記述言語
        (3) 構造化コーディング

構造化プログラミング

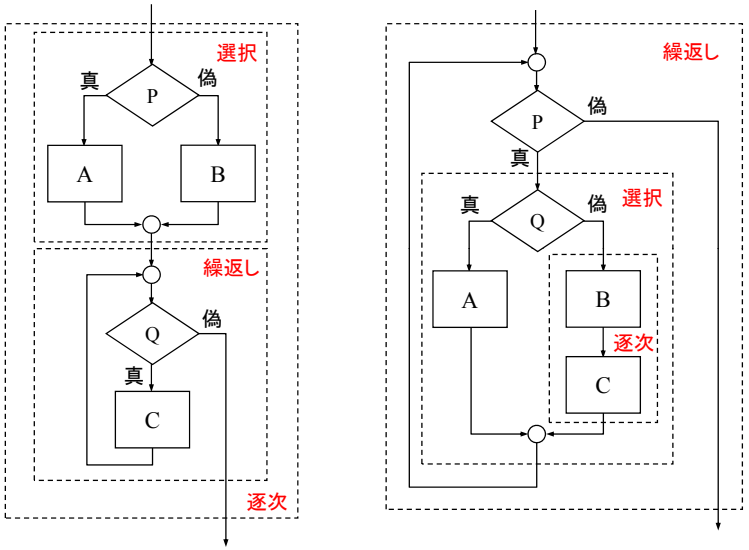
- 手続き型プログラムの論理を3つの基本制御構造の組合せで表現
  (a) 逐次(sequence):
  命令(命令群)A, Bを順番に実行
  (b) 選択(selection, if-then-else):
  条件Pの真偽により命令A, Bの実行を選択
  (c) 繰返し(iteration, do-while)
  条件Pが真である間，命令Aを繰返し実行



構造化定理

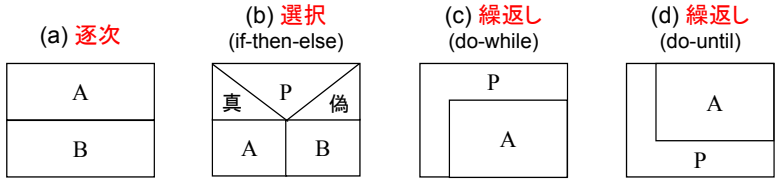
- 構造化定理[Böhm, Jacopini]:
  - ✓ すべての適正プログラムの論理は3つの基本制御構造 (逐次，選択，繰返し)で記述可能
- 適正プログラム(proper program):
  - ✓ プログラムの制御の流れに対し，1つの入口と1つの出口を持つ
  - ✓ プログラムの制御の流れにすべての命令が関係する

論理構造の例

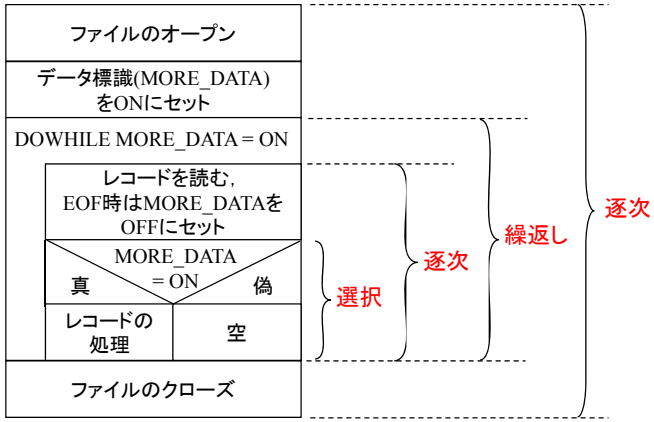


NSチャート

- フローチャート(flow chart):
  - ✓ プログラムの制御の流れを表現
  - ✓ 構造化の原則を破りやすい
- NSチャート(ナッシ・シュナイダーマン・チャート)[Nassi, Shneiderman]
  - ✓ 4つの制御論理構造を固有の記号で記述
  - ✓ 構造化の原則を破る制御の流れを記述することは不可能
  - ✓ データの存在範囲を把握しやすい



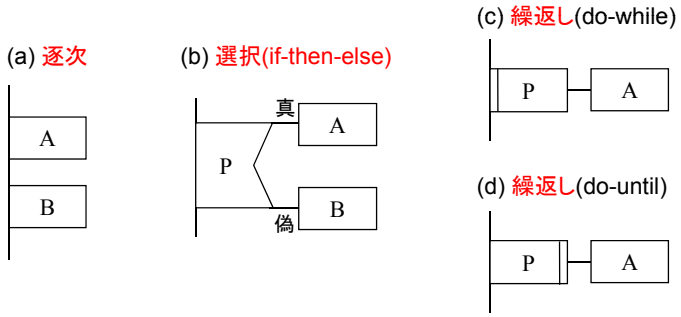
NSチャートの例



PAD

- PAD(problem analysis diagram)[二村]:
  - ✓ プログラムの論理を処理の実行順序と処理のレベルの2次元で表現
  - ✓ 木構造チャート

    e.g., SPD(structured programming diagram)
    HCP(hierarchical and compact description chart)
    YACII (yet another control chart II)



プログラム記述言語

- プログラム記述言語(PDL: program description language)
  - ✓ プログラムの論理をトップダウンに記述

    1) 論理の全体の流れを文で表現

    2) 各文を対象プログラム言語のレベルまで詳細化
  - ✓ 擬似言語(高水準言語)
    - 表現は自由
    - 最低限の規約(IF-THEN-ELSEやDO-WHILE，字下げ)

プログラム記述言語

```
在庫データ照会
  データ宣言
  初期設定
  DO WHILE トランザクションあり
    妥当トランザクションを得る
    IF 妥当トランザクション THEN
      IF トランザクション = A THEN
        入出庫活動データを得る
      ELSE
        IF トランザクション = B THEN
          在庫状況データを得る
          IF 在庫量 < 最低保有量 THEN
            発注メッセージを作成する
          END IF
        ELSE
          エラーメッセージを表示する
        END IF
      END IF
    ELSE
      エラーメッセージを表示する
    END IF
  END DO
終わり
```

```
妥当トランザクションを得る
  EOF標識のリセット
  トランザクション読取り
  IF NOT EOF THEN
    許可番号チェック
    IF 該当許可番号あり THEN
      トランザクション形式チェック
      IF 形式 OK THEN
        トランザクション標識をOKにセット
      ELSE
        エラーメッセージの印刷
      END IF
    ELSE
      エラーメッセージの印刷
    END IF
  ELSE
    EOF標識のセット
  END IF
終わり
```


構造化コーディング

- プログラムの論理を3つの基本制御構造をそのまま命令に変換
  - ✓ 逐次，選択，繰返し
- goto文の使用を制限
  - ✓ 例外処理
  - ✓ モジュールからの脱出
  - ✓ ループからの脱出
  - ✓ 重複コードの排除
- コーディング規約
  - ✓ 字下げ(indentation): 制御範囲の明確化
  - ✓ データ名や関数の名前の付け方

コーディング例

基本構造		C [Ritchie, 1972]	Pascal [Wirth,1970]	COBOL [1959]	Fortran [Backus,1955]
逐次		A; B;	A; B	A. B.	A B

選択		if (p)
		A;
		else
		B;
			if p then
			A
			else
			B;
				IF p
				A
				ELSE
				A
					IF (p) THEN
					A
					ELSE
					B
					END IF

繰返し	DO-WHILE	while (p) A;	while p do A;	PERFORM A UNTIL NOT p.	10 IF (p) GOTO 20 A GO TO 10 20 CONTINUE
繰返し	DO-UNTIL	do A; while (p);	repeat A until p;	PERFORM A. PERFORM A UNTIL p.	10 CONTINUE A IF (p) GO TO 20 20 GO TO 10

プログラミングパラダイム

- プログラミング(programming)
  - ✓ 計算機を使って解くべき問題をプログラムとして記述すること
- プログラミングパラダイム(programming paradigm)
  - ✓ プログラムの作り方に関する規範
  - ✓ 設計手順やプログラム構造およびプログラムの記述方法を規定するもの
  - ✓ プログラミングの際に，何に着目して問題を整理するのか，何を中心にプログラムを構成するのかの方向付けを与えるもの

プログラム = アルゴリズム + データ構造
programs = algorithms + data structures [Wirth, 1976]

アルゴリズム = 論理 + 制御
algorithm = logic + control [Kowalski, 1979]

プログラミングパラダイムの例

- 手続き型プログラミング(procedural programming)
  - ✓ コンピュータの処理手順を文で記述
  - ✓ 構造化プログラミング
    Fortran, COBOL, Algol, BASIC, PL/I, Pascal, C, Ada
- 関数型プログラミング(functional programming)
  - ✓ 入出力関係を表現する関数とその呼出しで記述
    Lisp (λ算法; lambda calculus), Scheme, ML
- 論理型プログラミング(logic programming)
  - ✓ 入出力関係を述語論理(事実と規則)で記述
    Prolog (導出原理; resolution principle)
- オブジェクト指向プログラミング(object-oriented programming)
  - ✓ データとその操作をカプセル化したオブジェクトとその間のメッセージ通信で記述
    Smalltalk, C++, Java, C#
- アスペクト指向プログラミング(aspect-oriented programming)
  - ✓ オブジェクトをまたがる横断的な関心ごと(cross-cutting concern)をアスペクトにまとめて記述し，後で織り合わせ(weaving)
    AspectJ, Hyper/J, DemeterJ(adaptive programming)

プログラムの記述例

- 手続き型プログラミング

Cのプログラム(繰返し)

```
int fact = 1, i;
for (i = 1; i <= n, i++)
   fact = i * fact;
```

Cのプログラム(再帰)

```
int fact(int n) {
   if (n == 0) return(1);
      return (n * fact(n-1));
}
```

- 関数型プログラミング

関数 *fact ()*の定義

$$fact(x) = \text{if } x=0 \text{ then } 1 \text{ else } x \times fact(x-1)$$

Lispプログラム

```
(DEFUN fact(N)
   (COND ((ZEROP N) 1)
   (T (TIMES N (FACT(SUB1 N)))))
```

- 論理型プログラミング

階乗 *x!* の定義

$$fact(0,1)$$

$$fact(x, y) \leftarrow sub(x, 1, w) \wedge fact(w, z) \wedge times(x, z, y)$$

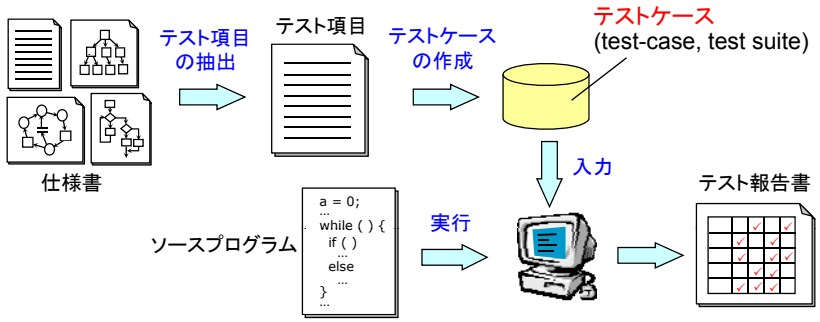
$$\exists [fact(3, y)]$$

Prologのプログラム

```
fact(0, 1).
fact(X, Y) :- sub(X, 1, W),
              fact(W, Z), times(X, Z, Y).
?- fact(3, Y).
```

ソフトウェアテスト

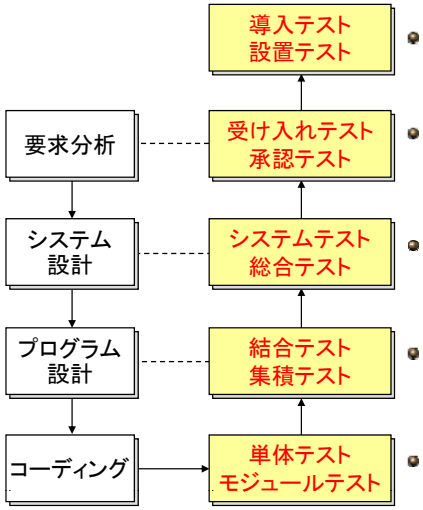
ソフトウェアテスト



- プログラムが仕様書に定義した振る舞いを満たすかどうかの検査
  - ✓ 故障(failure): 要求に反した振る舞い，間違った動作
  - ✓ 障害(fault)/欠陥(defect): 故障の原因，故障を引き起こすソフトウェア内部の誤り
  - ✓ 障害(failure): ソフトウェアが障害を持つことになった開発者の誤り

「テストはエラーの存在を示すことはできるが，エラーが存在しないことは示せない」

ソフトウェアのテスト工程



- 実際の稼働環境での運用による検査
- 利用者が要求仕様に基づき実施する検査
- プログラム全体に対するエラーの検出
  機能，性能，例外，限界，操作，耐久など
- モジュール間インタフェースに関するエラーの検出
- モジュール内に存在する論理エラーの検出

単体テスト

- 単体テスト(unit test):
  モジュール内部に存在するエラーを検出

(a) ブラックボックステスト(block-box test)

テストデータを与えて，実行結果を観察することでエラーを検出

- ✓ プログラムの外部仕様(機能)に着目
- ✓ プログラムの詳細(内部構造や内部論理)を無視

同値分割法，限界値分析

(b) ホワイトボックステスト(white-box test)

テストデータを与えて，実行のようすを追跡することでエラーを検出

- ✓ プログラムの内部仕様(構造や論理)に着目
- ✓ 制御の流れに基づくテスト網羅

テスト網羅技法

(c) コードレビュー(code review)

- ✓ コードウォークスルー(walk-through)：非形式的，正当性に関するコメント
- ✓ インスペクション(inspection)：形式的，リストとコードとの照合

同値分割法

- 同値分割法
  プログラムの入力領域を同値クラスに分類することでテストケースを作成

(1) 同値クラスの識別
機能仕様の入力条件を満足する範囲(有効同値クラス)と
満足しない範囲(無効同値クラス)に分割

例)

入力条件	有効同値クラス	無効同値クラス
文字数	4 ~ 8	3以下，9以上
文字の種類	英字と数字の組合せ	英字のみ，数字のみ

(2) クラスに基づくテストケースの作成
(2a) 有効同値クラスを検査するテストケースを作成
e.g., amku5ge
(2b) 1つの無効同値クラスと残りの同値クラスを検査する
テストケースを作成
e.g., xy9, jdsi5enjcd, abcdef, 123456

限界値分析法

- 限界値分析法
  入出力条件の境界値を詳しくテストするテストケースを作成

(1) 入出力条件の識別
機能仕様の入出力条件に着目し、境界を判別する

例)

条件	1～64の数字
境界	1と64

(2) 境界に基づくテストケースの作成
上記の例の場合
0,1,2,63,64,65

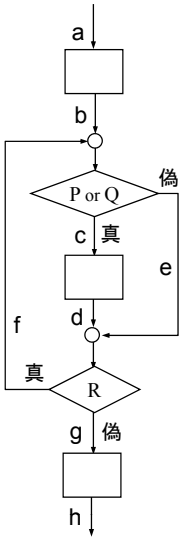
テスト網羅技法(1)

- 命令網羅，節点網羅(statement coverage, C0 coverage)
  - ✓ プログラム中のすべての文を1回以上実行
    例) P or Qが真，Rが偽 (パス: abcdgh)

  網羅(coverage) = 実行した文 / 全文

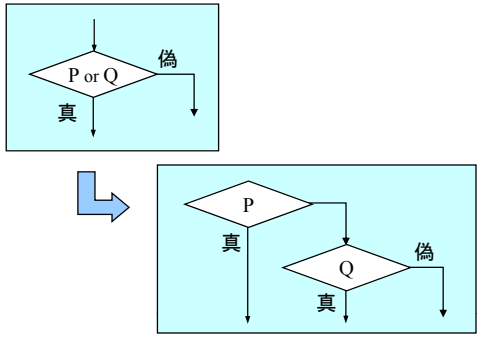
- 枝網羅，分岐網羅(edge coverage, C1 coverage)
  - ✓ プログラム中のすべての枝を1回以上実行
    例) P or Qが真，Rが真(パス: abcf)
    P or Qが真，Rが偽(パス: abcdgh)
    P or Qが偽，Rが真(パス: abef)
    P or Qが偽，Rが偽(パス: abegh)

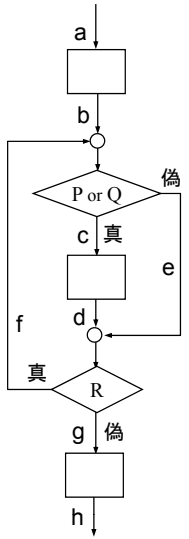
  網羅(coverage) = 通過した枝 / 全枝



テスト網羅技法(2)

- 条件網羅(condition coverage)
  - ✓ プログラム中のすべての判定条件を1回以上実行
    e.g., PとQを区別
    Pが真，Qが偽 or 偽
    Pが偽，Qが真 or 偽



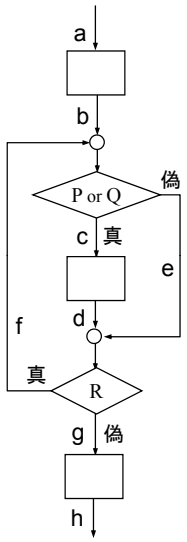


テスト網羅技法(3)

- パス網羅(path coverage)
  - ✓ 判定条件間の依存性(条件の組合せ)を考慮
  - ✓ プログラム中のすべてのパスを1回以上実行

  e.g., abcdgh + abcdfcdgh + abegh
  + abefegh + abcdfegh + …

網羅(coverage) = 実行したパス / 全パス



結合テスト

- 結合テスト(integration test):
  モジュールインタフェース(パラメータや共通データ)に関するエラーを検出

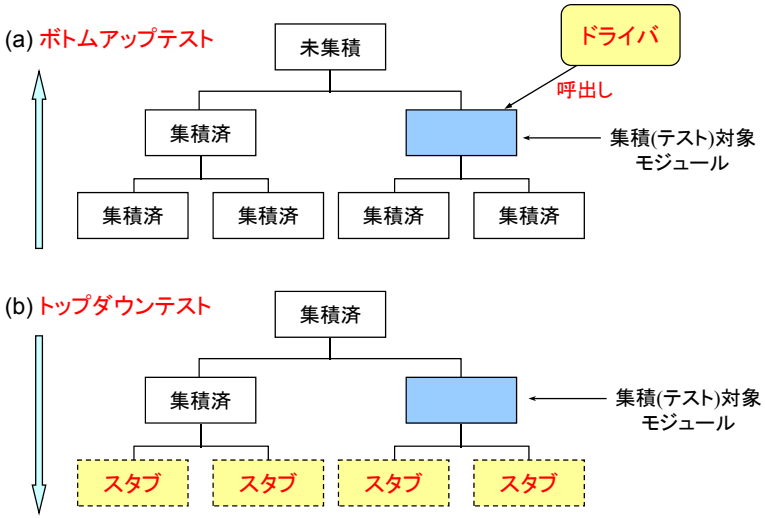
(a) ボトムアップテスト(bottom-up test)
- ✓ モジュール階層図の最下位モジュールからテスト開始
- ✓ テストドライバ(仮のメインプログラム)が必要
- ✓ 初期段階から並行にテスト可能

(b) トップダウンテスト(top-down test)
- ✓ モジュール階層図の最上位モジュールからテスト開始
- ✓ プログラムスタブ(身代わりモジュール)が必要
- ✓ インターフェースエラーが早期に発見可能

(c) 混合テスト(mixed integration)/サンドイッチテスト(sandwich test)
- ✓ ボトムアップテストとトップダウンテストの統合

(d) ビックバンテスト(big-bang test)
- ✓ すべての構成要素を単独でテスト後，一括して結合してテスト

ボトムアップテストとトップダウンテスト



演習: トップダウンプログラミング

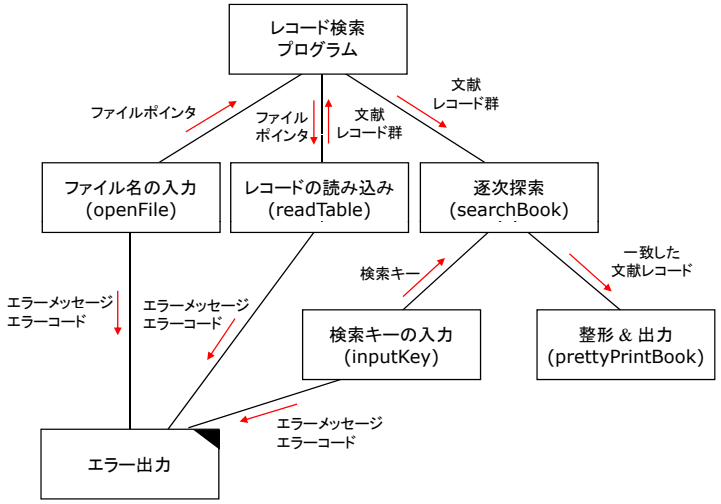
トップダウンプログラミング: トップダウンテストに基づくプログラミング

逐次マッチングによる検索プログラム

- 文献簡易目録ファイル名（最大100バイト）をプロンプトメッセージ出力の後，キーボードから受け取り，このファイルのすべてのレコードを配列に読み込んで，総文献件数を出力した後，入力された姓の読み（ローマ字）が前方一致で該当するすべてのレコードを検索し，姓の読み，著者，書名，出版社，出版年，ISBN番号をそれぞれ1行ずつ画面に表示するプログラムを作成せよ．検索は，会話型で行われ，終了コード（/）が入力されるまで繰り返すものとする．

- 文献簡易目録ファイルは，姓の読み，著者，書名，出版社，出版年，ISBN番号の6つの項目が空白文字で区切られてならんでおり，各項目の最大長はどれも200バイトを超えない．文献レコードの区切りは改行になっており，最大レコード数は2000件を超えないものとする．

演習: モジュール構成図



演習: ステップ(1) (データ構造の決定とメイン手続きの作成 &テスト)

```
#define RECORD_NUM  2000
#define FIELD_SIZE  201

typedef struct _BookRecord {
    char yomi[FIELD_SIZE];     /* 姓の読み */
    char author[FIELD_SIZE];   /* 著者 */
    …
} BookRecord;

main( )
{
    BookRecord bookTable[RECORD_NUM];
      /* 文献レコード群 */
    FILE *fp;   /* ファイルポインタ */
    int num;    /* 総文献件数 */

    fp = openFile( );
    num = readRecord(fp, bookTable);
    fclose(fp);
    searchBook(book, num);
}
```

作成 & テスト対象

```
FILE *openFile( )
{
    printf("####  Open File ####¥n");
    return(NULL);
}
```

スタブ(stub)

```
int readRecord(FILE *fp, BookRecord bookTable[ ])
{
    printf("####  Read Records ####¥n");
    return(0);
}
```

```
void searchBook(BookRecord book[ ], int num)
{
    printf("####  Retrieve Books  ####¥n");
}
```


演習: ステップ(2-a) (openFile手続きの作成 & テスト)

```
#define FILENAME_SIZE  101

FILE *openFile( )
{
    char filename[FILENAME_SIZE];  /* ファイル名 */
    FILE *fp;  /* ファイルポインタ */

    /* プロンプトの表示 */
    printf("文献簡易目録ファイル> ");

    /* ファイル名の入力 */
    /* 本来はファイル名の長さを検査する */
    scanf("%s", filename);

    /* ファイルポインタの取得(ファイルオープン) */
    if ((fp = fopen(filename, "r")) = NULL)

        /* ファイルオープンに失敗したとき，エラー出力 */
        printError("Cannot open file %s¥n", filename, 1);

    return(fp);
}
```

```
void printError(char *msg, char *str, int no)
{
    printf("#### Error ####¥n");
}
```

スタブ

演習: ステップ(2-b) (readRecord手続きの作成 & テスト)

```
int readRecord(FILE *fp, BookRecord bookTable[ ])
{
    int num;  /* 総文献件数 */

    /* 総文献件数の初期化 */
    num = 0;

    /* ファイルの終わりまで */
    while (!feof(fp)) {

        /* 文献レコードを読み込む */
        if (fscanf(fp, "%s %s %s %s %d %s¥n",
          bookTable[num].yomi, bookTable[num].author, … ) == 6)
            /* 文献件数をかぞえる */
            num++;
        else
            /* 各フィールドが正常に読み込めなかったとき，エラー出力 */
            printError("Format error %s¥n", book.Table[num].yomi, 1);
}

    /* 総文献件数の表示 */
    printf("Total number of books = %d¥n", num);
    return(num);
}
```

演習: ステップ(2-c) (searchBook手続きの作成 & テスト)

```
void searchBook(BookRecord book[ ], int num)
{
   char key[FIELD_SIZE];
   int i;

   while (1) {
      /* 検索キーの入力 */
      inputKey(key);

      /* 検索キーが"/"のとき，ループから脱出 */
      if (strcmp(key, "/") == 0) break;

      /* 逐次マッチング */
      printf("--------¥n");
      for (i = 0; i < num; i++) {

         /* 一致したレコードを整形して出力 */
         if (strcmp(book[i].yomi, key, strlen(key)) == 0) {
            prettyPrintBook(book[i]);
            printf("--------¥n");
         }
      }
   }
}
```

```
void inputKey(char key[ ])
{
   printf("####  Input Key ####¥n");
   strcpy(key, "/");
}
```

```
void prettyPrintBook(BookRecord book)
{
   printf("####  Pretty Print Books ####¥n");
}
```

スタブ

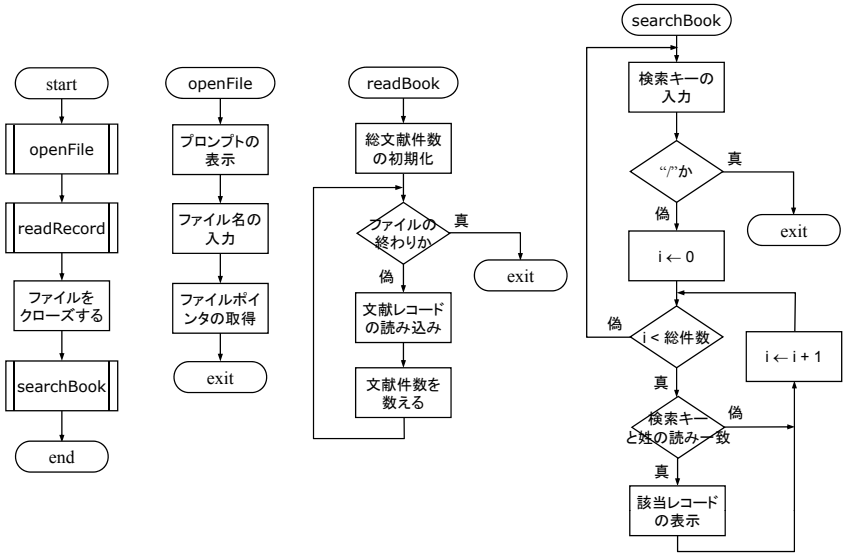
演習: ステップ(3) (残り手続きの作成 & テスト)

```
void inputKey(char key[ ] )
{
   char key[システムが許す最大サイズ];

   /* 検索キーの入力 */
   printf(" 検索文字> ");
   scanf("%s", key);

   while (strlen(key) >= FIELD_SIZE) {

      /* 入力検索キーが長すぎる */
      printError("Invalid input key %s ¥n", key, 0);

      /* 検索キーの再入力 */
      printf(" 検索文字> ");
      scanf("%s", key);
   }
}
```

```
void prettyPrintBook(BookRecord book)
{
   printf("姓の読み: %s¥n", book.yomi);
   printf("著者: %s¥n", book.author);
   ...
}
```

```
void printError(char *msg, char *str, int no)
{
   /* エラーメッセージの出力 */
   fprintf(stderr, msg, str);

   /* エラーコードが0でないとき，終了 */
   if (no != 0) exit(no);
}
```


演習: フローチャート(全体)



システムテスト

- システムテスト(system test):
  顧客の要求をシステムが満たしているかどうかを検査
  テスト計画書(test plan)を作成

(a) 機能テスト(function test)

- ✓ 統合システムの機能が要求仕様通りに稼動するかどうかを検査
  原因結果グラフ法

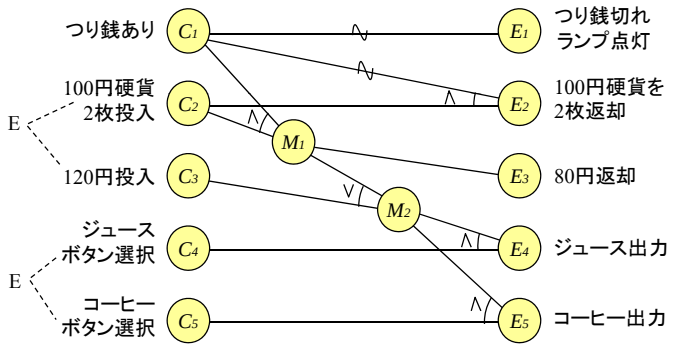
(b) 性能テスト(performance test)

- ✓ 非機能要求を評価
  過負荷テスト(stress test)，容量テスト(volume test)，
  構成テスト(configuration test)，互換性テスト(compatibility test)，
  セキュリティテスト(security test)，タイミングテスト(timing test)，
  環境テスト(environment test)，品質テスト(quality test)，
  回復テスト(recovery test)，保守テスト(maintenance test)，
  文書化テスト(documentation test)，ユーザビリティテスト(usability test)

原因結果グラフ法

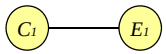
- 原因(入力)と結果(出力)の因果関係に着目し，テストケースを作成

(1) 原因結果グラフ(CEG: cause-effect graph)の作成

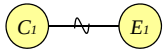


原因結果グラフ

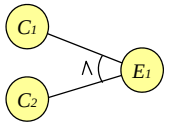
(a) 肯定(C_1であればE_1)



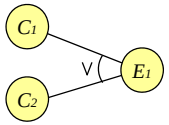
(b) 否定(C_1でなければE_1)



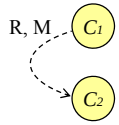
(c) 論理積(C_1かつC_2であればE_1)



(d) 論理和(C_1またはC_2であればE_1)

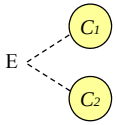


(e) 必要(R), マスク(M)



R (require): 一方が成立すれば他方も成立
M (mask): 一方が成立すれば他方は不成立

(f) 排他的論理和(E), 包含(I), 1つのみ(O)



E (exclusive): 同時には成立しない
I (include): 少なくとも一方は成立
O (only one): 常に一つだけ成立

決定表

(2) 決定表(decision table)の作成

	原因/結果	テスト項目					
		1	2	3	4	5	6
入力	(C_1) つり銭あり	×	○	○	×	○	×
	(C_2) 100円硬貨2枚投入	○	×	○	×	○	○
	(C_3) 120円投入	×	×	×	○	×	×
	(C_4) ジュースボタン選択	○	○	○	○	×	×
	(C_5) コーヒーボタン選択	×	×	×	×	○	○
出力	(E_1) つり銭切れランプ点灯	✓			✓		✓
	(E_2) 100円硬貨2枚返却	✓					✓
	(E_3) 80円返却			✓		✓	
	(E_4) ジュース出力			✓	✓		
	(E_5) コーヒー出力					✓	

e.g.,「1」: つり銭なしの状態で，200円を投入して，ジュースのボタンを押した場合，
つり銭切れランプが点灯しており，200円が返却される．

形式手法とソフトウェア検証

形式手法

- 論理(logic)，代数(algebra)，集合論(set theory)などの数学に基づく形式化(formalization)をソフトウェア開発に取り入れること

  仕様の厳密性，プログラムの正しさの検証などに貢献

  - ✓ 機能の形式化
  - ✓ データの形式化
  - ✓ 形式的仕様記述言語Z (Z notation)

機能の形式化

- 入出力条件による形式化(論理的仕様で表現されることが多い)
  - ✓ システムの機能を入出力時に成立する条件で表現
- 関数による形式化(関数型仕様)
  - ✓ 入力を出力に変換する関数として表現

例) 整数xとyの最大公約数zを求める

入出力条件による定義
入力条件: $integer(x) \land integer(y) \land x > 0 \land y > 0$
出力条件: $interger(z) \land divide(z, x) \land divide(z, y) \land$
$\forall w.(integer(w) \land (divide(w, x) \land divide(w, y) \Rightarrow z \geq w))$
ただし，$divide(a, b)$はbがaによって割り切れることを意味

関数$gcd(x, y)$の定義
$gcd(x, y) = gcd(x, y \bmod x) = gcd(x \bmod y, y)$
$gcd(x, y) = gcd(y, x)$
$gcd(x, 0) = gcd(0, x) = x$
ただし，$a \bmod b$はaをbで割った余りを指す
関数変換と見ると，左辺を右辺に書き換えることを意味

データの形式化

- 抽象データ型(ADT: abstract data type)
  - ✓ データ構造を，それに対する演算(operation)の組により定義
  - ✓ 演算の仕様(インタフェース)と内部実装を分離し，公開演算子を通してのみデータにアクセス可能(データのカプセル化: encapsulation)
  - ✓ 内部状態を隠蔽(情報隠蔽: information hiding)
- 代数的仕様(algebraic specification)
  - ✓ データ型を代数と見なし，代数を公理で記述することで，演算の意味を定義

例) スタック(stack)の代数的仕様記述

型種(sort):	公理(axioms):
Stack(integer)	s: Stack
	z: integer
演算子(operators):	$pop(push(z, s)) = s$
init: → Stack	$pop(init) =$ **stack-error**
push: integer × Stack → Stack	$top(push(z, s)) = z$
pop: Stack → Stack	$top(init) =$ **stack-error**
top: Stack → integer	$empty(init) =$ **true**
empty: Stack → bool	$empty(push(z, s)) =$ **false**

形式的仕様記述言語Z

- 集合論に基づくデータの型付け

例) 基本型$NAME$と$DATE$に対する$\{ NAME, DATE \}$の仕様

BirthdayBook
$known: \mathbb{P}\ NAME$
$birthday: NAME \pfun DATE$

$known = \mathbf{dom}\ birthday$

$known$: 名前の集合
$birthday$: 誕生日の集合
$A \pfun B$: AからBへの部分関数

スキーマ(schema)

InitBirthdayBook
$BirthdayBook$

$known = \emptyset$

AddBirthdayBook
$\Delta BirthdayBook$
$name?$: NAME
$date?$: DATE

$name? \notin known$
$birthday' = birthday \cup \{ name? \mapsto date? \}$

FindBirthday
$\Xi BirthdayBook$
$name?$: NAME
$date!$: DATE

$name? \in known$
$date! = birthday\ (name?)$

暗黙条件: $known' = \mathbf{dom}\ birthday'$
$a \mapsto b$: 対(a, b)

ソフトウェア検証

- ソフトウェア検証(verification & validation)
  ソフトウェアが要求される品質を満たし，信頼できることを確認

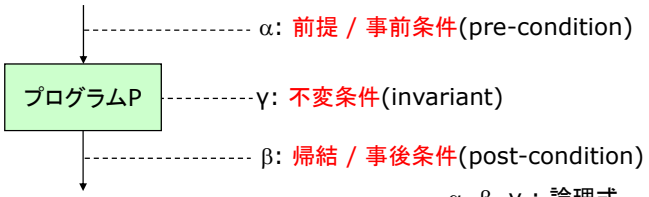
(a) 仕様検証
- ✓ モデル検査(model checking)
  モデルが時相論理式(temporal logic formula)を満たすかどうかを自動的に検査
  - 安全(safety): 望ましくない事象が決して起こらないこと
  - 活性(liveness): 望む事象がいつかは起こること
- ✓ レビュー(review)

(b) プログラム検証
- ✓ 動的検証: テスト，プロファイル分析，網羅度計測，表明検査
- ✓ 静的検証: 正当性検証(Hoare論理)，型検査，
  記号実行，制御フロー解析，データフロー解析

プログラム検証

- テスト
  エラーの存在を示すことはできるが，エラーが存在しないことは示せない
- 正当性(correctness)証明
  エラーが存在しないことを数学的に証明
  正当性: (1) プログラムが必ず停止する(停止性)
  (2) 答えが必ず正しい(部分正当性: partial correctness)
  - ✓ 公理的意味論(axiomatic semantics)に基づく方法
    - 帰納表明法(inductive assertion method) [Floyd]
    - ホーア論理(Hoare logic) [Hoare]
  - ✓ 定理証明器(theorem prover)を利用

公理的意味論

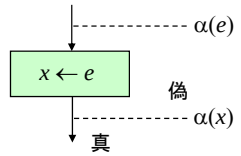


- ✓ Pの実行前にαが成立するならば，Pの実行後にβが成立する
  例) 10個の要素を持つ配列が入力されると，ソートされた配列
  (配列添字が大きい方が，その値が大きい) が出力される
- ✓ Pの実行中は必ずγが成立する
  例) 配列の要素の数は変わらない

契約による設計(DbC: design by contract)で採用
契約: 事前条件を満たした状態でプログラムを実行した場合は，
事後条件を満たすこと状態を実現することを約束

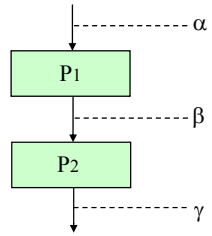
検証条件(1)

(1) 代入文



$\{\alpha(e)\}\ x \leftarrow e\ \{\alpha(x)\}$
$x \leftarrow e$の実行前に$\alpha(e)$が成立するならば，
$x \leftarrow e$の実行後に$\alpha(x)$が成立する

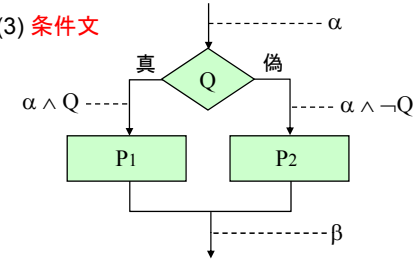
(2) 複合文



$$\frac{\{\alpha\}\ P_1\ \{\beta\},\ \{\beta\}\ P_2\ \{\gamma\}}{\{\alpha\}\ P_1; P_2\ \{\gamma\}}$$

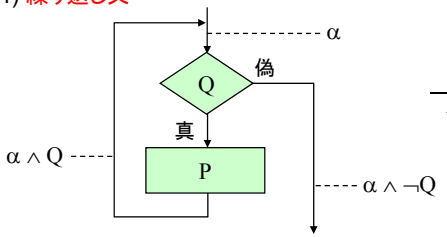
検証条件(2)

(3) 条件文



$$\frac{\{\alpha \wedge Q\}\ P_1\ \{\beta\},\ \{\alpha \wedge \neg Q\}\ P_2\ \{\beta\}}{\{\alpha\}\ \textbf{if}\ Q\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \{\beta\}}$$

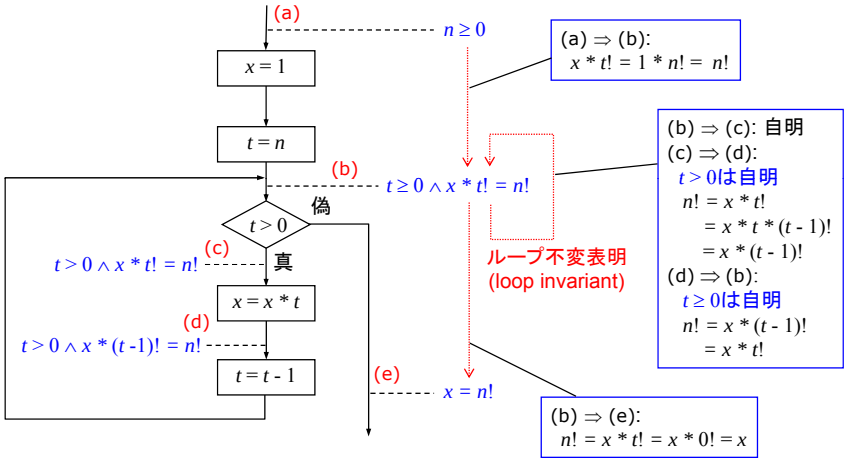
(4) 繰り返し文



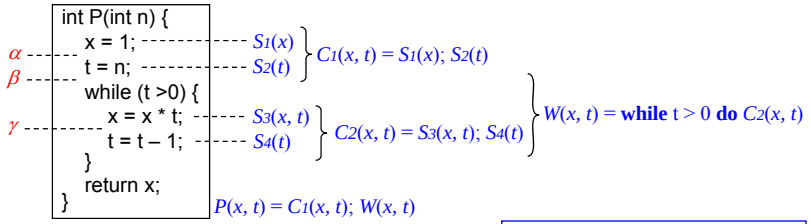
$$\frac{\{\alpha \wedge Q\}\ P\ \{\alpha\}}{\{\alpha\}\ \textbf{while}\ Q\ \textbf{do}\ P\ \{\alpha \wedge \neg Q\}}$$

帰納表明法

表明(assertion): プログラムの各時点で成立する変数間の関係式



ホーア論理



$\{x = n!\} = \{x * t! = n! \wedge t = 0\}$
$= \{x * t! = n! \wedge \neg(t > 0)\}$

$\{n \geq 0\}\ P(x, t)\ \{x = n!\}$
$\Leftrightarrow \{n \geq 0\}\ C_1(x, t); W(x, t)\ \{x = n!\}$
$\Leftrightarrow \{n \geq 0\}\ C_1(x, t)\ \{\beta\},\ \{\beta\}\ W(x, t)\ \{x = n!\}$
$\Leftrightarrow \{n \geq 0\}\ S_1(x); S_2(t)\ \{\beta\},\ \{\beta\}\ \textbf{while}\ t > 0\ \textbf{do}\ C_2(x, t)\ \{x = n!\}$
$\Leftrightarrow \{n \geq 0\}\ S_1(x)\ \{\alpha\},\ \{\alpha\}\ S_2(t)\ \{\beta\},\ \{\beta\}\ \textbf{while}\ t > 0\ \textbf{do}\ C_2(x, t)\ \{x * t! = n! \wedge \neg(t > 0)\}$
$\Leftrightarrow \{n \geq 0\}\ x = 1\ \{\alpha\},\ \{\alpha\}\ t = n\ \{x * t! = n! \wedge t > 0\},\ \{x * t! = n! \wedge t > 0\}\ C_2(x, t)\ \{x * t! = n!\}$
$\Leftrightarrow \{n \geq 0\}\ x = 1\ \{\alpha\},\ \{\alpha\}\ t = n\ \{x * t! = n! \wedge t > 0\},\ \{x * t! = n! \wedge t > 0\}\ S_3(x, t); S_4(t)\ \{x * t! = n!\}$
$\Leftrightarrow \{n \geq 0\}\ x = 1\ \{\alpha\},\ \{\alpha\}\ t = n\ \{x * t! = n! \wedge t > 0\},$
$\{x * t! = n! \wedge t > 0\}\ S_3(x, t)\ \{\gamma\},\ \{\gamma\}\ S_4(t)\ \{x * t! = n!\}$
$\Leftrightarrow \{n \geq 0\}\ x = 1\ \{\alpha\},\ \{\alpha\}\ t = n\ \{x * t! = n! \wedge t > 0\},$
$\{x * t! = n! \wedge t > 0\}\ x = x * t\ \{\gamma\},\ \{\gamma\}\ t = t - 1\ \{x * t! = n!\}$
$\Leftrightarrow \ldots$

型検査

- 型(type): ものの集まり，ものを分類するための仕組み
  プログラミング言語処理系では，変数や式の取り得る値を規定するもの
  例) int x; 変数xの値は-2147483648から2147483647の整数である
  boolean p; 変数pの値は真(true)か偽(false)
- 型検査(type checking)
  例) 1 + 2: 型安全である (int型とint型の加算)
  1 + true: 型安全でない (int型とboolean型の加算)
  - ✓ 型に関して不適切な演算や操作が行われることのないプログラム
    = 型安全なプログラム
  - ✓ 型安全でないプログラムは実行時エラーを引き起こす可能性あり
    → 信頼性の低下

  型推論(type inference)を利用して，プログラム実行前に実行時エラーの可能性を発見
- 静的な型付けに基づく言語(強く型付けされた言語)
  すべての変数や式の静的な型がコンパイル時に決定可能
  例) C, C++, Java, ML

ソフトウェア保守と再利用

ソフトウェア保守

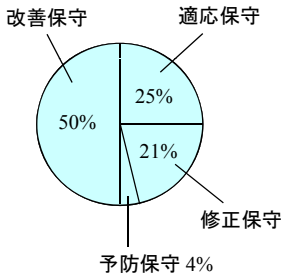
- ソフトウェア保守(software maintenance):
  現行のソフトウェアを維持・管理する作業
  ソフトウェアは変更を受け入れ可能，部品の磨耗なし
  ↔ ハードウェア保守

(a) 修正保守(corrective maintenance)
故障に対するソフトウェアの修正

(b) 適応保守(adaptive maintenance)
システムやハードウェアの進化や変更に
応じて発生する変更

(c) 改善保守(perfective maintenance)
機能追加や使いやすさの向上のための変更
維持・管理のし易さを向上させるための変更

(d) 予防保守(preventive maintenance)
故障を未然に防ぐための訂正，潜在的な障害の修正

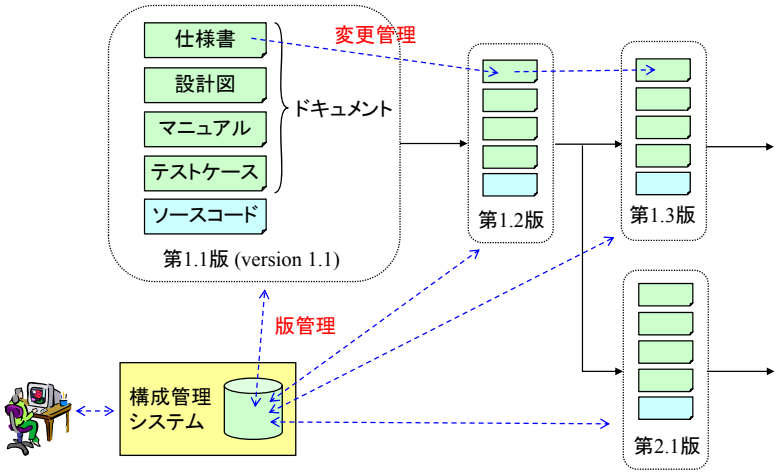


Lientz and Swanson (1981)

保守技法

- 構成管理(configuration management)
  - ✓ バージョン(version)とリリース(release)を管理
- 影響分析(impact analysis)
  - ✓ 保守による変更あるいは追加による影響が及ぶ範囲(モジュール)を把握
  - ✓ 変更に関連するリスクの評価
- 回帰テスト(regression test)
  - ✓ 変更されなかった部分がもとの通り正常に動作するかどうかを確認
  - ✓ 変更前に適用されたテストデータを実行
- プログラムスライシング(program slicing)
  - ✓ プログラム中の特定の文sの値に影響を与える，あるいは，
    文sの値が影響を与える文を抽出する手法
  - ✓ スライス(slice)：抽出された文の集合
    - 文sの値に影響を与える文集合を逆方向スライス(backward slice)と
      文sの値が影響を与える文集合を順方向スライス(forward slice)
    - 静的解析に基づく静的スライス(static slice)と
      実行時の情報に基づく動的スライス(dynamic slice)

構成管理



プログラムスライシング

```
a = 0;
b = a + 1;
```
データ依存関係(data dependence)
変数aの値の定義が変数aの値の参照に到達

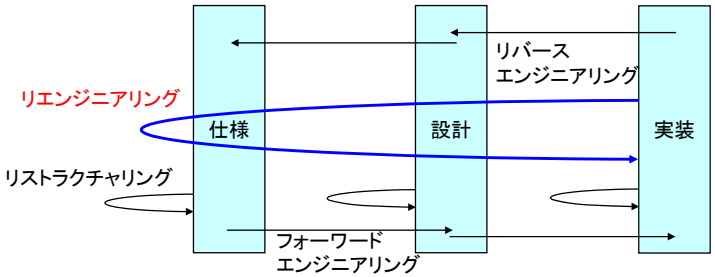
```
if (a < 0) {
  b = 10;
}
```
制御依存関係(control dependence)
文"b = 10"の実行は，if文の判定の結果に依存

```
1: int func(int data[]) {
2: int sum = 0;
3: int prod = 1;
4: int i = 0;
5: while (i < data.length()) {
6:   sum = sum + data[i];
7:   prod = prod * data[i];
8:   i++;
9: }
```
もとのプログラム

```
1: int func(int data[]) {
2: int sum = 0;
3:
4: int i = 0;
5: while (i < data.length()) {
6:   sum = sum + data[i];
7:
8:   i++;
9: }
```
文6のsumに関する静的逆方向スライス

ソフトウェアリエンジニアリング

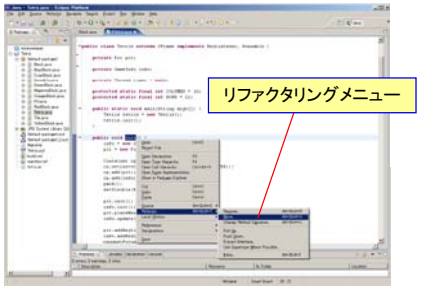
- ソフトウェアリエンジニアリング(software reengineering)
  リバースエンジニアリング + フォーワードエンジニアリング
- リバースエンジニアリング(reverse engineering)
  - ✓ ソースコードから設計図や要求仕様を回復(recovery)
- フォーワードエンジニアリング(forward engineering)
  - ✓ 従来と開発と同方向
- リストラクチャリング・再構成(restructuring)
  - ✓ 内部表現の単純化，構造化



リファクタリング

- ソフトウェアリファクタリング(software refactoring)
  factoring: 因数分解
  - ✓ リストラクチャリングの一種
  - ✓ 既存ソフトウェアの設計の理解性や変更容易性を向上させることを目的とした上で，そのソフトウェアの外部からみた挙動(振る舞い)を変えずに，内部構造を再構成すること
  - ✓ 大きな(複雑な)設計変更を一連の小さな変換により実現

例) 関数の名前変更
変数の名前変更
関数の移動
変数の移動



ソフトウェア再利用

- ソフトウェア再利用(software reuse):
  ソフトウェアシステム中の任意の要素を繰り返し使用する作業
  - ✓ 文書，コード，設計，要求，テストケース，...

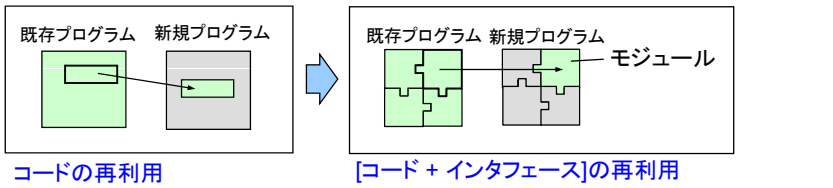
(a) 生産者側での再利用(producer reuse): 再利用可能な要素を作成
vs. 消費者側の再利用(consumer reuse): 再利用可能な要素を使用

(b) ブラックボックス再利用(black-box reuse): 修正なしで利用
vs. ホワイトボックス再利用(white-box reuse): 一部変更して利用

(c) 構成的再利用(compositional reuse):
再利用可能な要素を組み合わせてシステムを構築
vs. 生成的再利用(generative reuse):
実際に使用する要素を再利用可能な要素から生成

(d) 垂直的再利用(vertical reuse):
同一プロジェクトや同一アプリケーション領域での再利用
vs. 水平的再利用(horizontal reuse)
プロジェクトや領域を横切る再利用

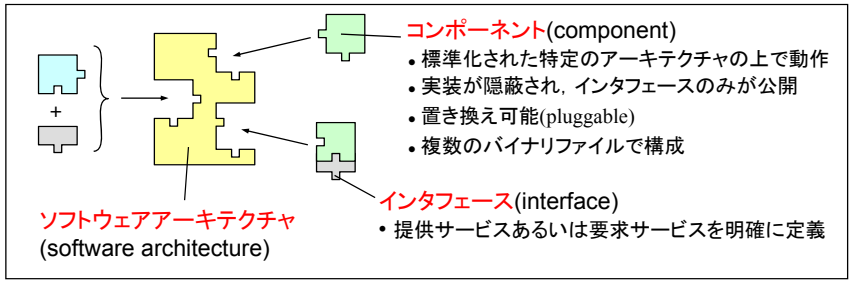
コンポーネントウェア



コードの再利用

[コード + インタフェース]の再利用

[(コード) ＋ インタフェース + 設計]の再利用



コンポーネント(component)
- 標準化された特定のアーキテクチャの上で動作
- 実装が隠蔽され，インタフェースのみが公開
- 置き換え可能(pluggable)
- 複数のバイナリファイルで構成

インタフェース(interface)
- 提供サービスあるいは要求サービスを明確に定義

ソフトウェアアーキテクチャ
(software architecture)

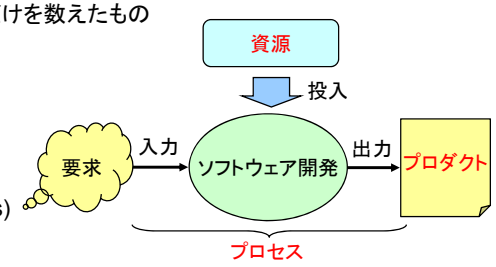
ソフトウェア開発管理

開発計画

- 開発計画の構成要素
  - ✓ 開発の目的(開発プロジェクトの目的)
  - ✓ 開発の目標(システム利用者の要望，業務運営上の方針)
  - ✓ 開発対象業務(開発対象の範囲と機能)
  - ✓ 開発システムの基本構成(SW構成, HW構成, NW構成)
  - ✓ 開発システムの運用方針(管理運用者，業務上の制約)
  - ✓ 開発工数と開発コスト:
    ソフトウェアメトリクス(metrics, 定量的評価尺度)
    工数見積もり技法
  - ✓ 開発スケジュール(作業進捗管理): ガントチャート
  - ✓ 開発体制, 開発環境, 開発方法(方法論やツール)
  - ✓ 成果物の管理方法(構成管理方法)
  - ✓ リスク管理(risk management)
    - リスク衝撃(risk impact): 否定的事象に関連する損失
    - リスク確率(risk probability): 否定的事象が起こる可能性
    - リスク制御(risk control): 否定的事象の影響を最小・回避するアクション

ソフトウェアメトリクス

- プロダクトメトリクス(product metrics)
  - ✓ ソースコードの規模(行数)
    - LOC (lines of code)
    - NCNB (non-comment non-blank) LOC: コメントや空行を除いた行数
    - ステップ数: プログラム命令だけを数えたもの
  - ✓ 複雑さ
    - McCabeの尺度
- 資源メトリクス(resource metrics)
  - ✓ 作業時間
  - ✓ 開発者の能力
- プロセスメトリクス(process metrics)
  - ✓ 作業効率, 生産性
  - ✓ プロセス成熟度
    - CMM(capability maturity model, 能力度成熟モデル)
    - SPICE(software process improvement and capability determination)
    - ISO9000：品質の目標と制約を満たすために取るべき動作の仕様化
      ISO9000-3: ISO9001のソフトウェア開発向け文書

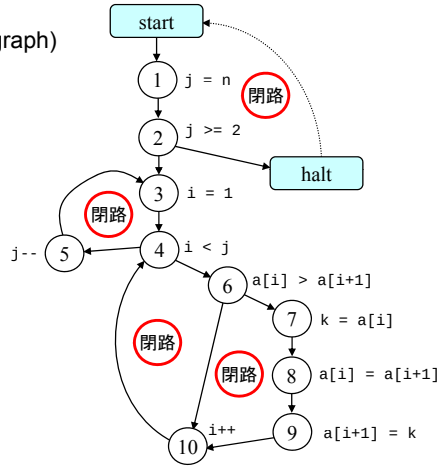


プログラムの複雑さ

- McCabeのサイクロマティック複雑度(cyclomatic complexity)
  プログラムの流れを有向グラフ(CFG)で表現し，
  一次独立な閉路の数で複雑度を測定
  CFG: 制御フローグラフ(control flow graph)

例) ソートプログラム

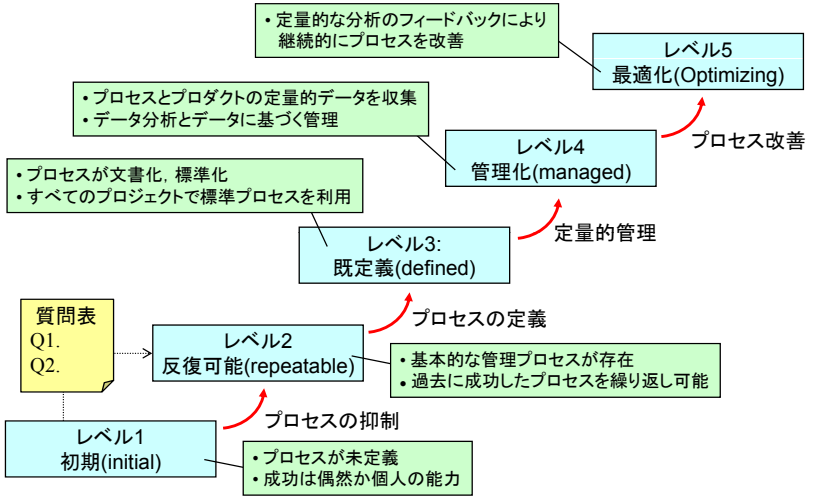
```
for (int j = n; j >=2; j--) {
  for (int i = 1; i < j; i++) {
    if (data[i] > data[i+1]) {
      int k = a[i];
      a[i] = a[i+1];
      a[i+1] = k;
    }
  }
}
```



複雑度 = 閉路の数 = 4

LOC = 9
ノード数 = 10

CMM



工数見積もり

- 類似法
  - ✓ 過去あるいは他の開発プロジェクトの実績を適用
- 標準タスク法
  - ✓ 標準的な作業(タスク)ごとに開発工数の基準を設定しておき，作業を積み上げることで全体の開発工数を算出
- COCOMO(Constructive Cost Model) [Boehm]
  - ✓ 開発ソフトウェアの規模(予測規模)から開発工数を算出
- COCOMO 2.0
  - ✓ FP法とソフトウェアの規模から開発工数を算出
- ファンクションポイント法(FP法：function point)
  - ✓ 入力や出力などの機能数から規模を算出
  - ✓ 開発工数への変換は未提示

標準タスク法

標準タスクAの作業日数

規模＼複雑度	単純	普通	複雑
大	1	2	3
中	1.5	3	5
小	2	4	7

プロジェクトXにおける標準タスクAの工数

規模＼複雑度	単純	普通	複雑
大	1×10	2×5	3×0
中	1.5×10	3×30	5×5
小	2×0	4×10	7×10

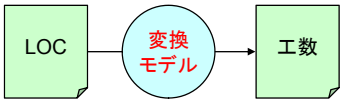
プロジェクトXにおける標準タスクAの件数

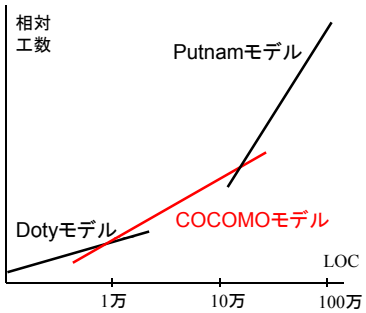
規模＼複雑度	単純	普通	複雑
大	10	5	0
中	10	30	5
小	0	10	10

10 + 10 + 0 + 15 + 90 + 25 + 0 + 40 + 70
= 260 (標準タスクAの総工数)

プロジェクトXの総工数
= 標準タスクAの総工数
+ 標準タスクBの総工数
+ 標準タスクCの総工数
+ ....

COCOMO





- 基本COCOMO
  - ✓ 開発規模(類似法で算出)のみから算出，開発の初期段階で利用
- 中間COCOMO
  - ✓ 要求分析結果により判明した影響要因で調整して算出
- 詳細COCOMO
  - ✓ 設計結果により判明した影響要因で調整して算出

COCOMO (cont'd)

- COCOMO開発モード

(a) 組織モード: 少人数で行う小規模システム

(b) 半組込みモード: 一般の業務システム

(c) 組込みモード: 厳しい制約を持つ大規模システム

例) 半組込みモードでの変換モデル

開発工数 (人月) = 3.0 × (LOC)$^{1.12}$ × 調整要因

開発期間(月) = 2.5 × (開発工数)$^{0.35}$

調整要因: 製品の複雑度，データベースサイズ，分析者の能力，経験，...

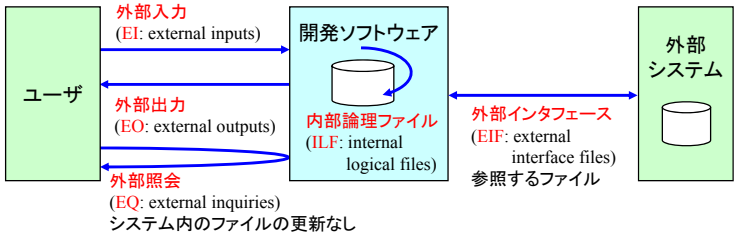
- COCOMO 2.0

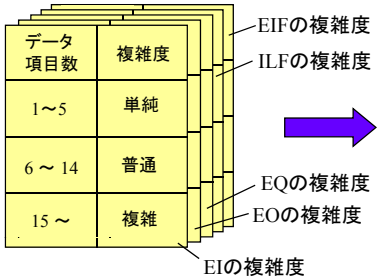
(a) アプリケーション組み立てモデル
  - ✓ GUIビルダーでの開発やプロトタイピングのような初期段階で適用
  - ✓ オブジェクトポイント法(オブジェクトの数)で規模を算出

(b) 初期設計モデル
  - ✓ システム構造が決定される前に適用
  - ✓ FP法に基づき機能数で規模を算出

(c) ポストアーキテクチャモデル
  - ✓ システム構造が決定された後に適用
  - ✓ FP法に基づく機能や行数で規模を算出

ファンクションポイント法(1)





複雑度別の機能数

機能＼複雑度	単純	普通	複雑
EI	10	12	14
EO	11	13	15
EQ	1	3	5
ILF	2	4	6
EIF	3	5	7

ファンクションポイント法(2)

複雑度別の機能数

機能＼複雑度	単純	普通	複雑
EI	10	12	14
EO	11	13	15
EQ	1	3	5
ILF	2	4	6
EIF	3	5	7

重み付け係数

機能＼複雑度	単純	普通	複雑
EI	×3	×4	×6
EO	×4	×5	×7
EQ	×3	×4	×6
ILF	×7	×10	×15
EIF	×5	×7	×10

機能＼複雑度	単純	普通	複雑
EI	10×3	12×4	14×6
EO	11×4	13×5	15×7
EQ	1×3	3×4	5×6
ILF	2×7	4×10	6×15
EIF	3×5	5×7	7×10

30 + 48 + 84
+ 44 + 65 + 105
+ 3 + 12 + 30
+ 14 + 40 + 90
+ 15 + 35 + 70
= 700 (未調整FP)

ファンクションポイント法(3)

システム特性	ポイント
1 データ通信	0
2 分散処理	0
3 パフォーマンス	4
4 高負荷環境	4
5 トランザクション量	2
6 オンラインデータ入力	1
7 エンドユーザの作業効率	2
8 マスターデータベースのオンライン更新	2
9 内部処理の複雑さ	3
10 再利用を考慮した設計	4
11 導入の容易性	1
12 運用の容易性	3
13 複数サイトでの使用	0
14 変更の容易性	2
合計	28

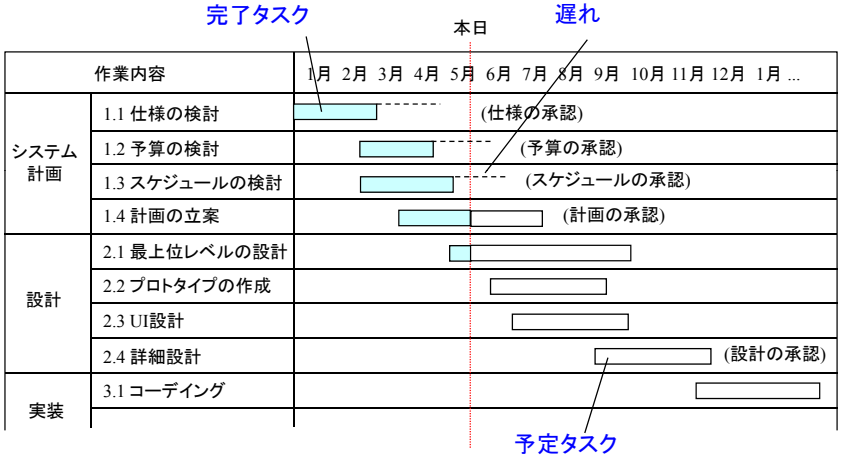
0: まったく関係ない
1: ほとんど影響を受けない
2: 適度に影響を受ける
3: 平均的な影響を受ける
4: 大きな影響を受ける
5: 非常に大きな影響を受ける

調整用係数
= 0.65 × (0.01 × 28)
= 0.182
FP
= 0.182 × 700
= 127.4

調整用係数 = 0.65 + (0.01 × システム特性の合計)
FP = 調整用係数 × 未調整FP

ガントチャート

- ガントチャート(Gantt chart)
  作業予定の明示 + 作業進捗の追跡

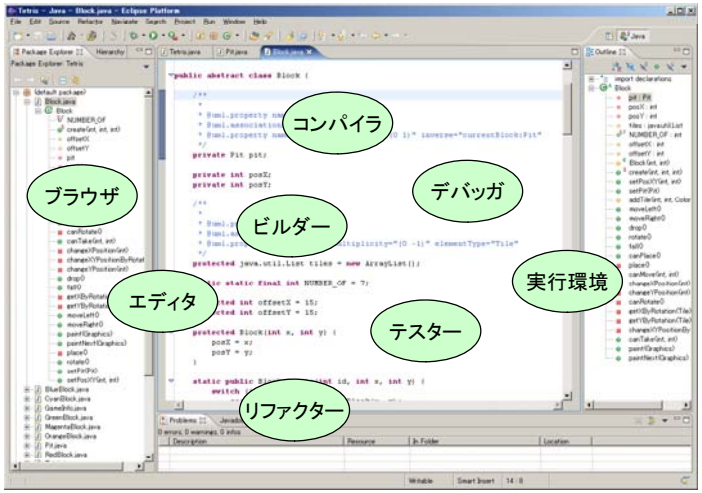


ソフトウェア開発環境

- バッチ型プログラミングツール(1950年代～60年代)
  例) 高級言語コンパイラ
- 対話型プログラミングツール(70年代)
  例) ドキュメント作成支援, エディタ, デバッガ
- 統合プログラミング環境(80年代後半)
  例) 構造化技法支援, ビジュアル化,
  CASE(computer-aided software engineering) [1986]
- 統合開発環境/自動化(90年代)
  例) 統合型CASE(全工程を支援)
  リポジトリ(repository):
  開発情報(企業モデル, データモデル, DFD, モジュール構成図,
  状態遷移図, プログラム設計書など)を集中管理するための保管庫
  cf. ライブラリ(library): ソースコードの保管庫
- オープン化(90年代後半～)
  例) コンポーネントウェア, 分散開発環境(CVS)

ソフトウェア統合開発環境Eclipse

- 統合開発環境(IDE: Integrated Development Environment)

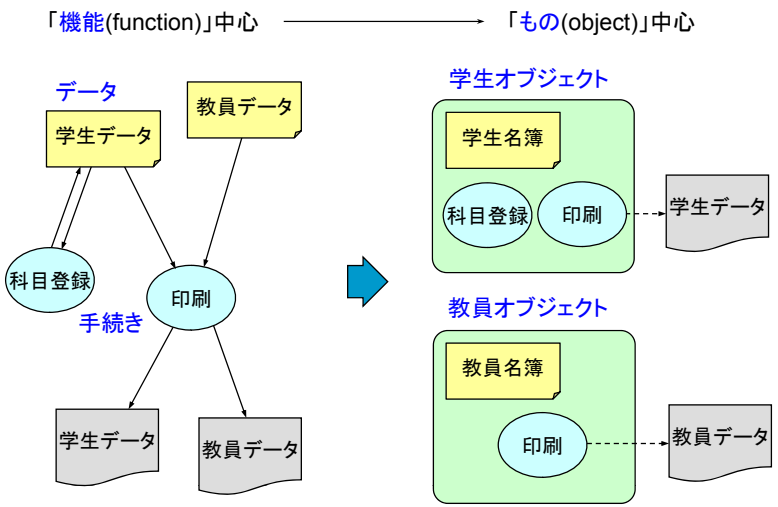


オブジェクト指向の概要

オブジェクト指向

- オブジェクト(object)
  - ✓ 実世界の「もの」や「役割」などの事柄(thing)を抽象化したもの
    (物理的なもの, 概念的なもの
    - 状態(state)
    - 振る舞い(behavior)
    - 識別性(identity)
- オブジェクト指向(object-oriented)
  - ✓ 実世界モデルをソフトウェアで直接的に表現する方法
  - ✓ オブジェクトを構成単位としてソフトウェアを構築する枠組み
  - ✓ コンピュータで取り扱う問題の中に存在する対象を
    そのままプログラミングの基本単位として表現する方法
- オブジェクト指向ソフトウェア開発(OO software development)
  - ✓ オブジェクト指向分析(OOA: OO analysis)
  - ✓ オブジェクト指向設計(OOD: OO design)
  - ✓ オブジェクト指向プログラミング(OOP: OO programming)

オブジェクト指向(cont'd.)



オブジェクト指向の基本概念I

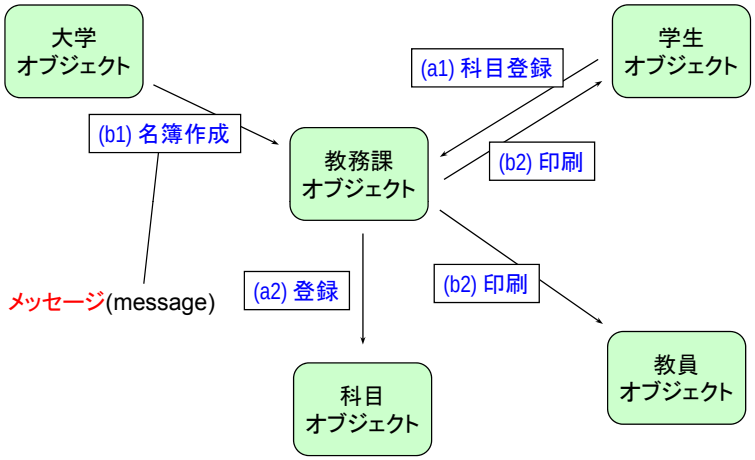
Object-based

- メッセージパッシング(message passing) → 分散協調型計算モデル
  オブジェクトに対するメソッド呼出し(method invocation)
  = オブジェクトに対する操作実行の依頼

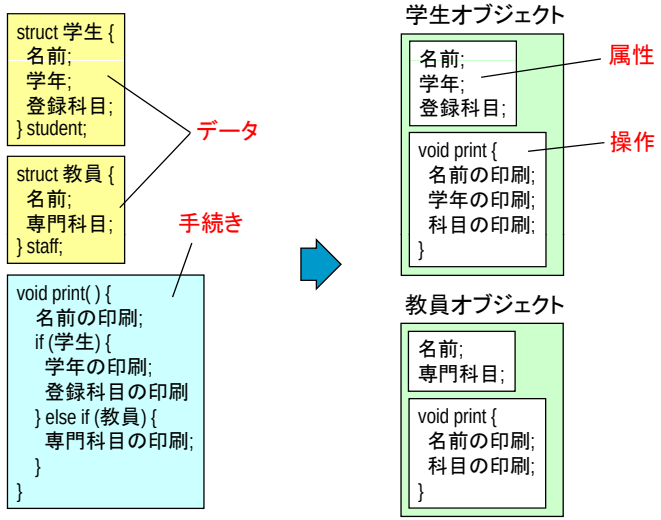
- モジュール化(modularization)
  データ(属性)と手続き(操作)をグループ化
  - 凝縮度(cohesion)と結合度(coupling)
- カプセル化(encapsulation)
  ≈ 情報隠蔽(information hiding)
  インタフェースと実装の分離
  インタフェースを介したデータ操作

} データ抽象化 (data abstraction)

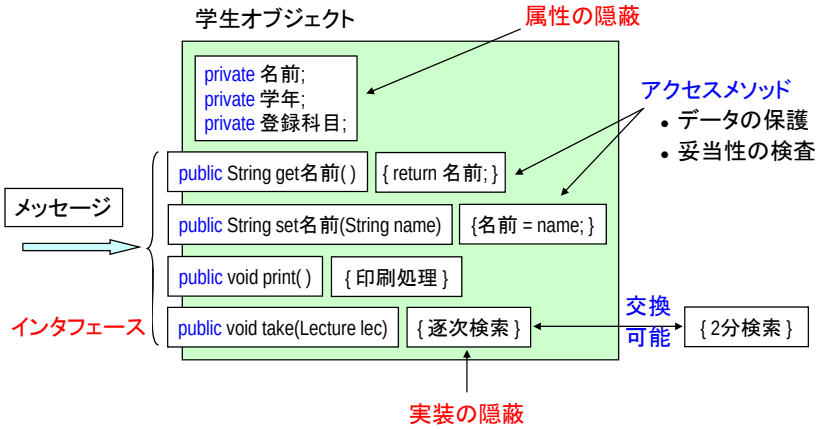
メッセージパッシング



モジュール化



カプセル化



インタフェース(interface) ≈ 型(type): どのようなメッセージを受け取るのか

オブジェクト指向の基本概念II

Object-oriented

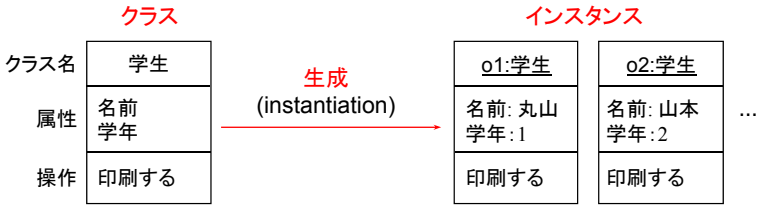
- クラスとインスタンス
  データ構造や手続きの同じオブジェクトをクラス(class)として定義
- 継承(inheritance)
  クラス間に成立する概念の汎化(特化)関係
  上位クラスの属性と操作の引継ぎ = 差分プログラミング
- 多相性/多態性(polymorphism)
  操作の多重定義と操作の動的束縛(dynamic binding)
  動的束縛: 操作を実行するインスタンスを受信側で決定

クラスとインスタンス

クラス(class): 共通の属性(attribute)と操作(operation)を持つ
オブジェクトを抽象化したテンプレートのようなもの

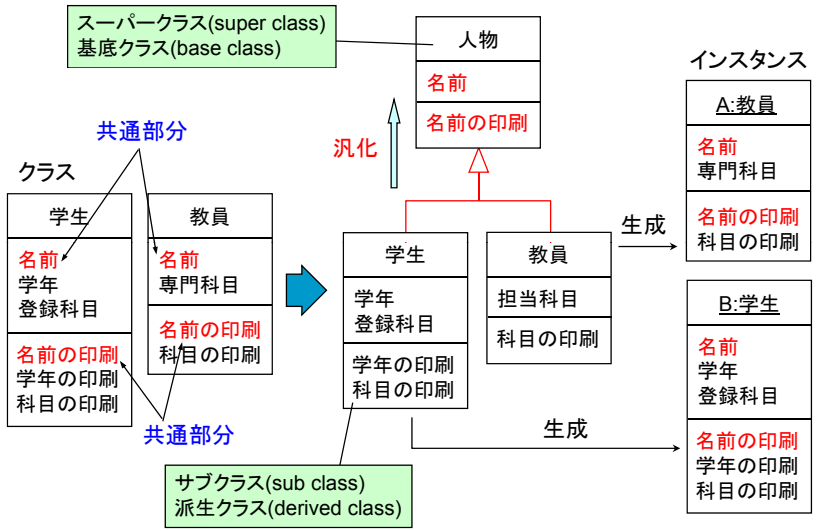
属性 = メンバ変数(member variable), インスタンス変数(instance variable)

操作 = メンバ関数(member function), メソッド(method)



生成されたオブジェクト = クラスのインスタンス(instance, 実体)

継承



多相性