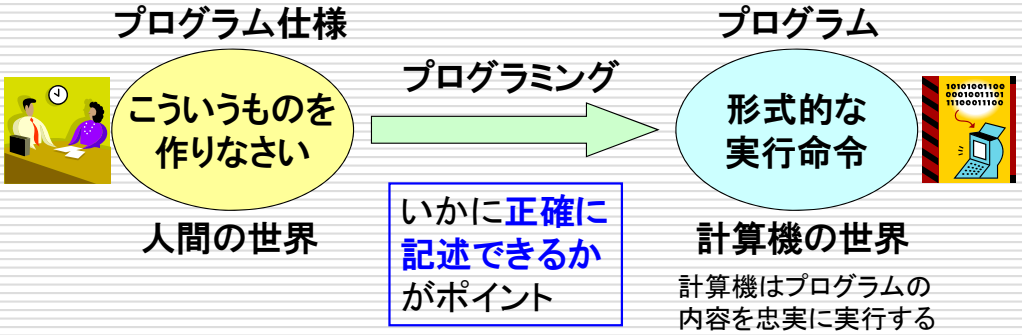
ソフトウェアテスト
[9] プログラミング技術

Software Testing
[9] Programming Tips and Techniques

あまん ひろひさ
阿萬 裕久（AMAN Hirohisa）
aman@ehime-u.ac.jp

プログラミングの位置付け

- 「**プログラム仕様**」をコンピュータ上で**実現**するのがプログラミングの役割



位置付け(いちづけ): positioning
プログラム仕様(プログラムしよう): program specification
実現する(じつげんする): implement
形式的な(けいしきてきな): formal
実行命令(じっこうめいれい): instruction
計算機(けいさんき): computer
正確に記述(せいかくにきじゅつ): to describe correctly

ソフトウェア開発 ≠ プログラミング

- 「プログラミングこそがソフトウェア開発である」と**思われがち**
  - 小規模開発の場合，要求分析・設計は開発者の頭の中だけで行われ，**いきなりプログラミングから**開始される
  - しかし，それでは**大規模開発では歯が立たない**
    - もちろんプログラミング能力は大切であるが設計や管理を軽んじてはいけない

例えば，スマートフォン用のOS：数百万行以上
印刷したらA4用紙で十万枚以上になる（高さ10m超）

思われがち（おもわれがち）: likely to be considered
小規模開発（しょうきぼかいはつ）: small-scale development
要求分析（ようきゅうぶんせき）: requirements analysis
設計（せっけい）: design
開発者の頭の中だけで行う（かいはつしゃのあたまのなかだけでおこなう）:perform it in only the developer's mind (head)
大規模開発（だいきぼかいはつ）: large-scale development
歯が立たない（はがたたない）: too tough
軽んじる（かろんじる）:treat it lightly

プログラミングで大事なこと

- 仕様を正しく反映した（**誤りの無い**）プログラムを書くこと
  - バグを作り込むというミス（エラー）を犯さないよう注意
- **分かりやすい**プログラムを書くこと
  - 「自分が分かればよい」という考え方は良くない
  - **他人でも分かるプログラムでなければ保守は無理**
    （書いた本人でも半年後には理解不能なことも）

仕様を正しく反映する（しようをただしくはんえいする）: reflect the specification correctly

誤りの無い（あやまりのない）: error free

半年後（はんとしご）: six month later

理解不能（りかいふのう）:incomprehension, hard to understand

プログラムの書き方・作り方

1. 見やすく・分かりやすく
2. 間違いを防ぐように
3. シンプルな構造に
4. 環境に依存しないように
5. コメント文を適切に
6. 変更しやすいように

書き方（かきかた）: how to write
作り方（つくりかた）: how to make
見やすく（みやすく）: easy to read
分かりやすく（わかりやすく）: easy to comprehend
間違い（まちがい）を防ぐ（ふせぐ）: prevent a mistake
環境に依存（かんきょうにいぞん）: depend on the environment
適切に（てきせつに）: properly

(1)見やすく・分かりやすく

- インデント(字下げ)を付け，ブロックの始まりと終わりが視覚的に分かるように

書き忘れ防止策として，“{” を書いたら “}” も書き，その間にコードを挿入していくとよい

```
sum = 0;
for ( i = 0; i < n; i++ ){
    if ( a[i] > 0 ){
        sum += a[i];
    }
}

printf("%d\n", sum);
```

- 意図的に空行を入れ，実行内容の切り替わりをアピール

視覚的に分かる(しかくてきにわかる): understand it visually
書き忘れ(かきわすれ): miss
防止策(ぼうしさく): measure to prevent
挿入(そうにゅう): insert
意図的に(いとてきに): by intention
空行(くうぎょう): blank line
切り替わり(きりかわり): switching

変数名には意味をもたせる

- その**変数が何を表しているのか**が分かるような名前が望ましい

  index, position, count
  あるいは略して idx, pos, cnt

  ただし，数行程度のプログラムで
  内容を見ればすぐに意味が分かる
  場面ならばさほど気にしなくてもよい
  （例）前ページの変数 `i` など

- Yes/No を問う場合は**疑問形**も使われる

  is_empty, hasNext

  単語の区切りには
  アンダースコア(_)
  または大文字を使うとよい

  （例）集合が空なら 1
  さもなくば 0

  （例）リストで次のノードが
  存在すれば 1 さもなくば 0

意味を持たせる（いみをもたせる）: give meaning
数行程度（すうぎょうていど）: a few lines
疑問形（ぎもんけい）: question
集合（しゅうごう）が空（から）: the set is empty
次（つぎ）のノードが存在（そんざい）する: the next node exists

変数名，関数名での スネークケースとキャメルケース

- スネークケース（snake case）
  - アンダースコア（ _ ）で単語をつなげて書く
  - （例）file_name, get_file_size

C言語の場合
スネークケース
が多く使われる

- キャメルケース（camel case）
  - 単語の切れ目に大文字を使う
  - （例）fileName, getFileSize

変数名（へんすうめい）: variable name
関数名（かんすうめい）: function name
単語（たんご）: word, term
大文字（おおもじ）: capital letter

関数名では機能と戻り値を意識する

□ その関数が**何をするのか**，**何を戻り値とするのか**が分かるような名前が望ましい

（例）

```
int get_length (char str[ ]){
   ・・・文字列の長さを調べる
}
```

```
void move_to (int x, int y){
   ・・・(x,y) へ移動
}
```

```
void sort (int array[ ], int length){
   ・・・ array をソーティング
}
```

単語の区切りには
アンダースコア(_)
または大文字を使うとよい

機能（きのう）: functionality
戻り値（もどりち）: return value
意識する（いしきする）: pay attention
文字列の長さ（もじれつのながさ）: length of string
移動（いどう）: move
ソーティング : sorting

【演習1】関数名を考えよ

- ヘルプを表示する

  void [　　] ( )

- うるう年(leap year)かどうかを調べる

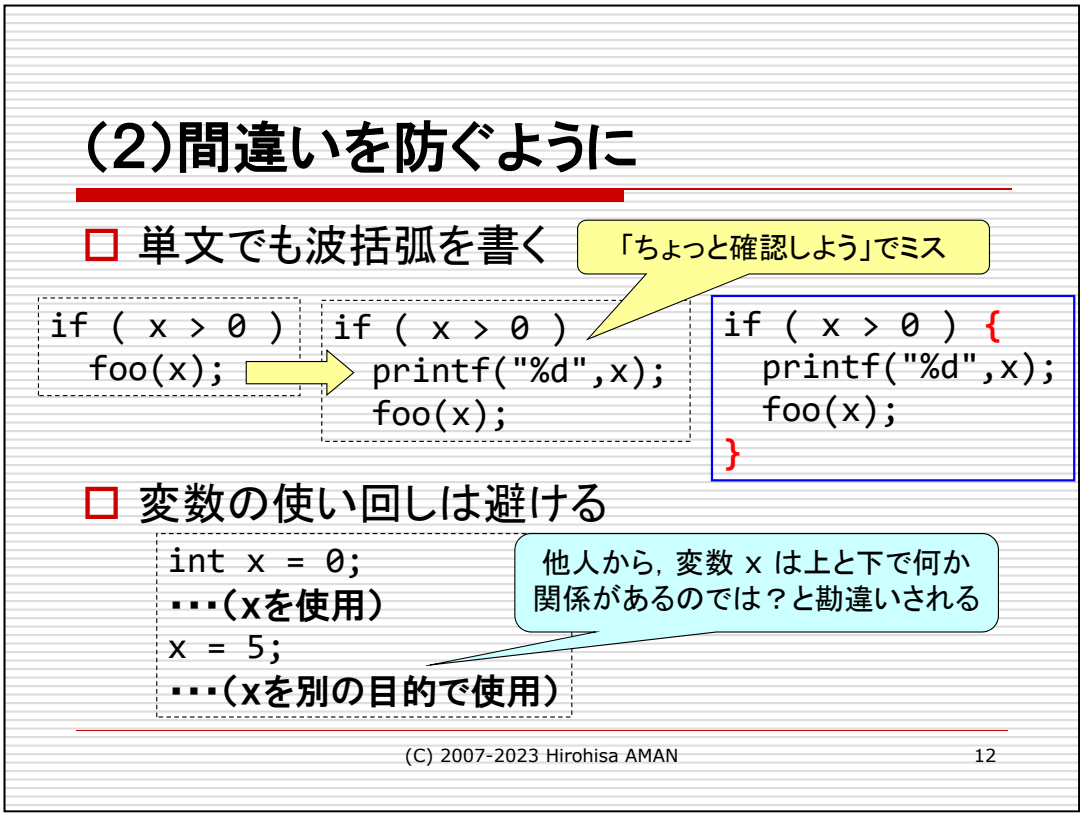
  int [　　] (int year)

- 配列の内容を別の配列へコピーする

  void [　　] (int src[], int dest[], int length)

  コピー元　コピー先　長さ

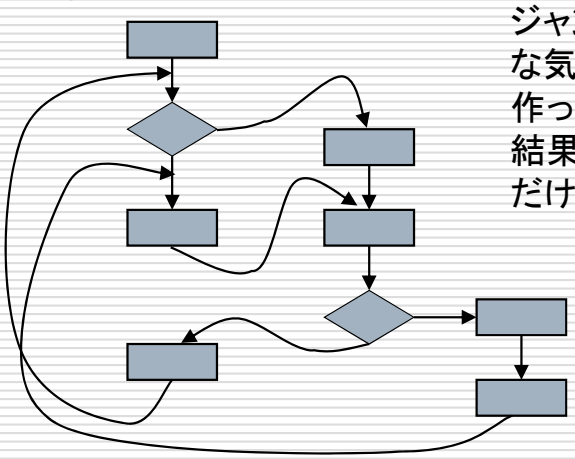
配列(はいれつ): array



単文(たんぶん): single statement
波括弧(ちゅうかっこ): brace
使い回し(つかいまわし): sharing
避ける(さける): avoid
勘違い(かんちがい)される: get misunderstood

(3)シンプルな構造に

- goto は使わない



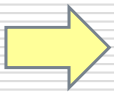
goto 文は好きな位置にジャンプできるので便利な気がするが，それは作っている途中だけで結果的には複雑になるだけである

この種のプログラムを「スパゲティプログラム」といい，ダメなプログラムの代名詞となっている

条件分岐が多くならないように

- if, for, while 文が多くなればなるほど，理解しにくく間違いが増える

```
sum = 0;
scanf("%d", &x);
do{
  if ( x > 0 ){
    sum += x;
    scanf("%d", &x);
  }
} while( x > 0 );
```



より良いプログラム（動きは同じ）

```
sum = 0;
scanf("%d", &x);
while( x > 0 ){
  sum += x;
  scanf("%d", &x);
}
```

試行錯誤の上，左のようなプログラムになったのかもしれないが，必要以上に複雑化してしまっている

条件分岐（じょうけんぶんき）: conditional branch

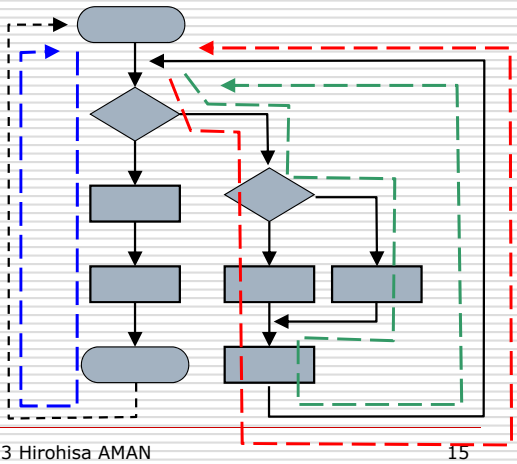
試行錯誤（しこうさくご）: trials and errors

サイクロマティック数(cyclomatic number)

- フローチャートの中の**独立したループ**の個数

プログラム構造の複雑さを評価する尺度として知られている

この数が10を超えると
保守性がかなり悪い
という研究報告がある



独立したループ(どくりつしたループ): independent loop
プログラム構造(プログラムこうぞう): program structure
複雑さ(ふくざつさ): complexity
尺度(しゃくど): measure
保守性(ほしゅせい): maintainability

(4)環境に依存しないように

- 計算機環境の違いが影響しないようにする
  - C言語の場合，変数に割り当てられるバイト数が処理系によって異なることがある（例えば，int 型が 2 バイトだったり 4 バイトだったり）

    sizeof(int) と書いたりするのはそのためである

  - OSが違うと #include で読み込むヘッダ（ライブラリ）も違うことがある

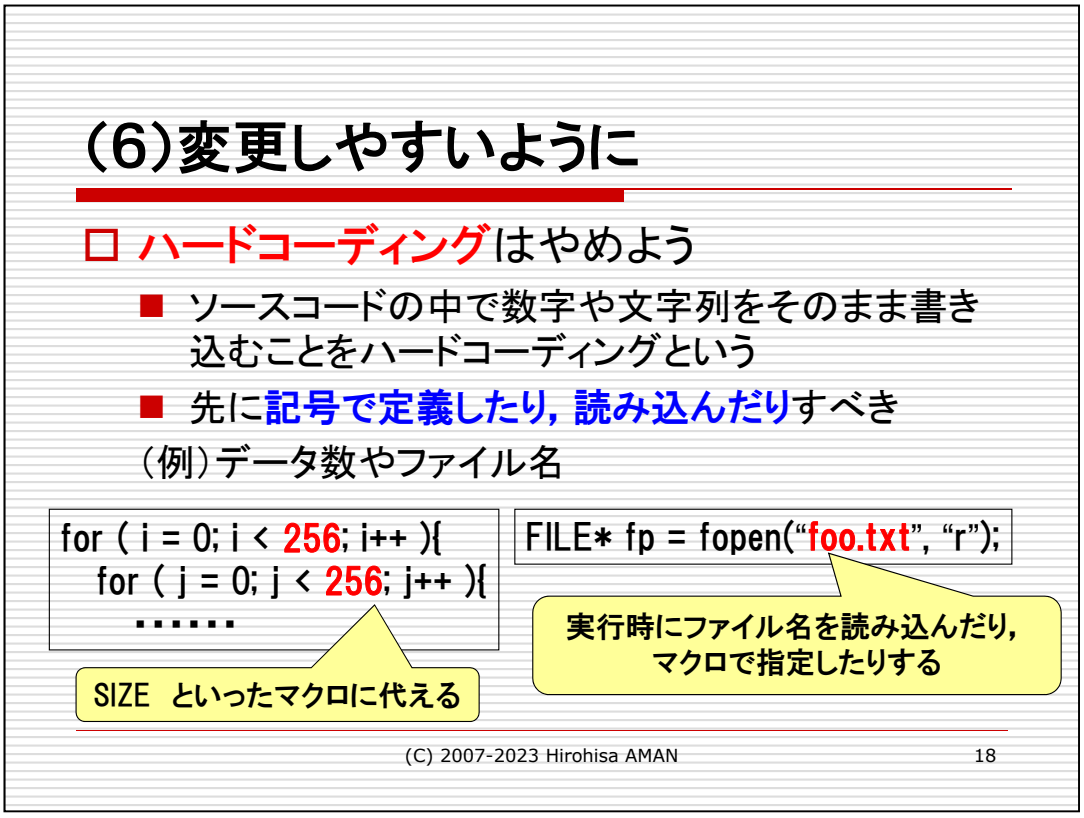
    少し専門的だが #ifdef 等を使って切り替えている

計算機環境（けいさんきかんきょう）: computing environment
処理系（しょりけい）: processing system, compiler

(5)コメント文は適切に

- 意味のあるかたまりごとに機能説明をコメントとして入れるのが望ましい
- ただし，単に書けばよいというわけではない
  - いちいちコメントを書かなければ理解できないようなプログラムもまた問題である
    - 関数名や変数名を分かりやすくして，プログラムをシンプルな構造にしておけばそれだけで十分な場合も多い

意味（いみ）のあるかたまり: meaningful chunk (code fragment)
機能説明（きのうせつめい）: description of the functionality
いちいちコメントを書く（かく）: write comments for each statement



ハードコーディング: hard coding
数字(すうじ): number
文字列(もじれつ): string
記号(きごう): symbol
定義(ていぎ)する: define

【演習2】ハードコーディングについて

- 例えば，配列の長さやデータファイルの名前は次のようにマクロで与えることが推奨される

```
#define SIZE 256
#define DATA_FILE "foo.txt"
```

- こうすることの利点（メリット）を考えよ

推奨（すいしょう）される: recommended

10-minutes rest break

10分休憩

マクロを使う場合の慣例

- C 言語においてマクロを定義する場合，その名前はすべて大文字にするのが一般的

```
#define SIZE 256
```

- 一方，変数名はすべて小文字にしておく

```
int size;
```

- そうすることで両者が混在していていても，それが変数なのかマクロなのかがすぐに分かる

慣例（かんれい）: convention
混在（こんざい）: mixture

リファクタリング（refactoring）

- ソースコードの**機能や意味を変えることなく**，コードの持つ**読みやすさや保守性を向上**させるように修正することをリファクタリングという
  - パフォーマンス（実行速度，メモリ使用量）の向上も含まれる
- いろいろなリファクタリングのパターンがあるので，興味のある人は調べてみるとよい

読みやすさ（よみやすさ）: readability
保守性（ほしゅせい）: maintainability
向上させる（こうじょうさせる）: improve
実行速度（じっこうそくど）: execution speed
メモリ使用量（メモリしようりょう）: memory usage

XP(eXtreme Programming)

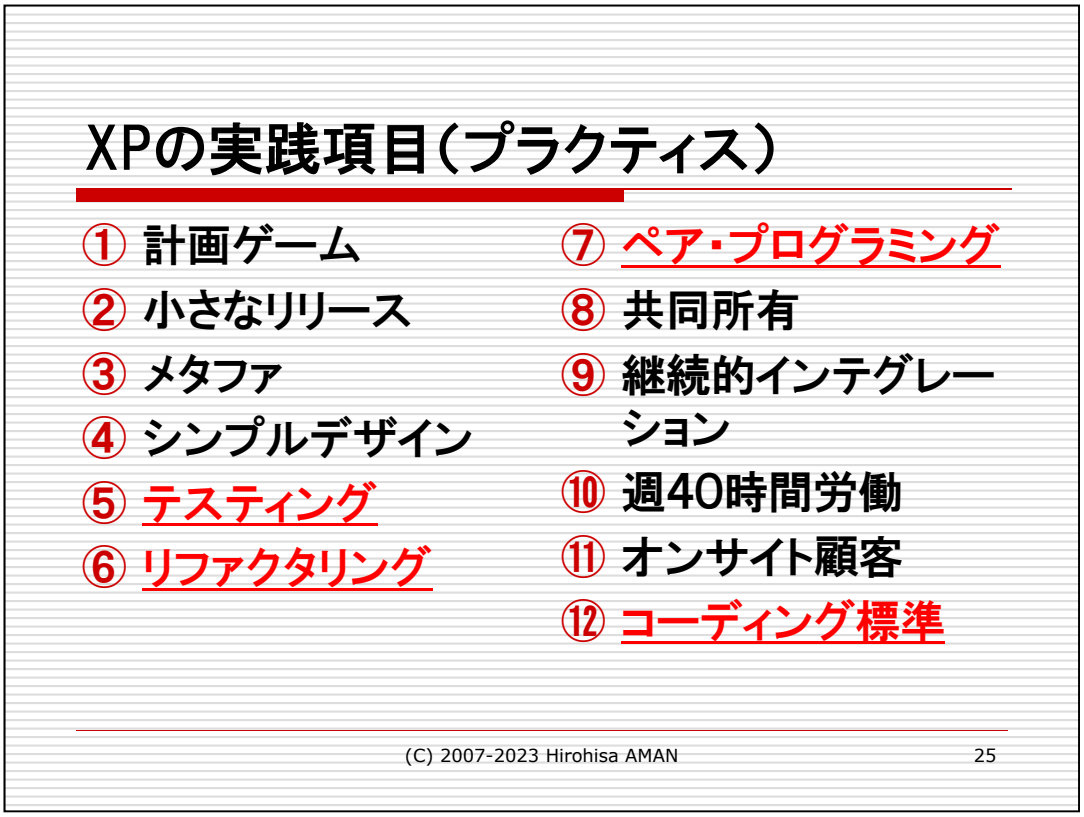
□ 少人数での短期間開発に適した開発スタイル

- 現代の IT 業界の変化は目まぐるしい(いわゆる「ドッグイヤー」である)
- 特に小規模開発の場合，開発は長くても数ヶ月で終わらなければならない

小規模開発では，大規模とは異なったスタイルで機敏な(アジャイル，agile)開発が望まれる

※ ドッグイヤー： 犬の寿命は人間の数分の一であり，数倍の早さで時間が経過

少人数(しょうにんずう): small-group
短期間開発(たんきかんかいはつ): short-term development
IT 業界(IT ぎょうかい): IT industry
目まぐるしい(めまぐるしい): dizzying
ドッグイヤー: dog year … seven times faster than human
機敏(きびん): agile



実践（じっせん）: practice

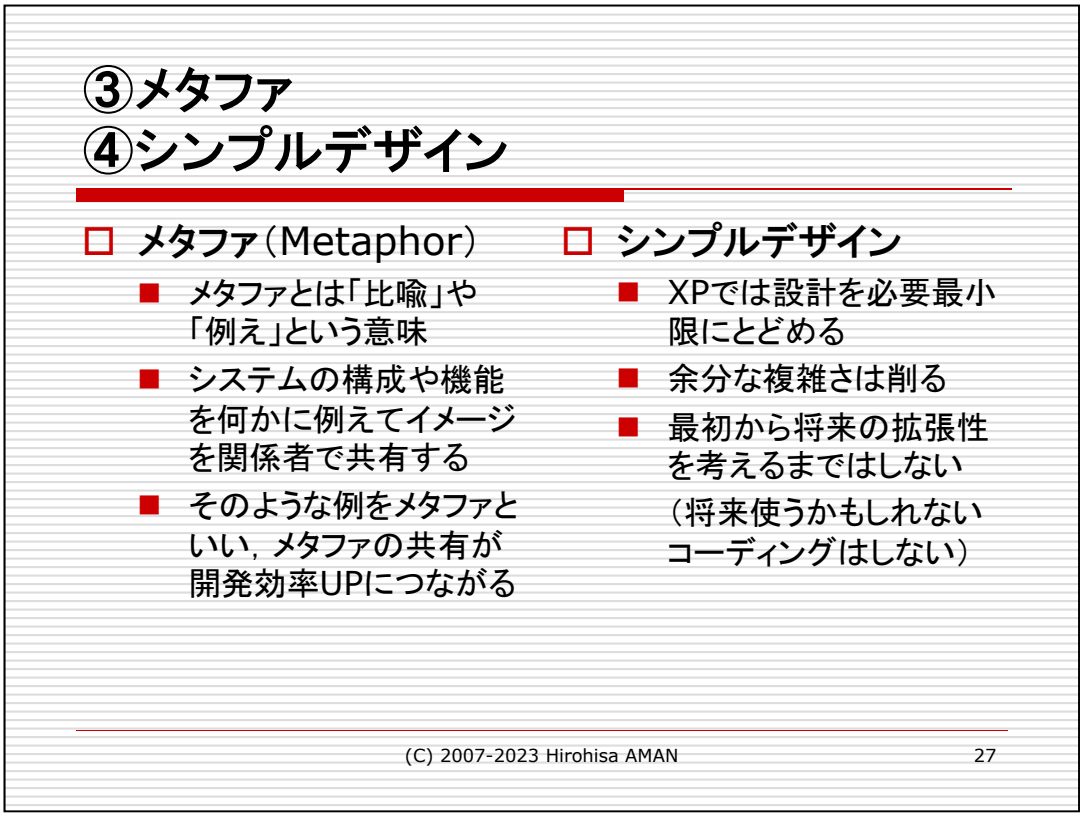
①計画ゲーム
②小さなリリース

- **計画ゲーム**
  顧客との間で計画を調整していく
  計画は最初の一度きりではなく，状況の変化に応じて**開発の途中でも継続的に**行う
  （短いスパンで開発を繰り返し，計画を調整していく）

- **小さなリリース**
  数ヶ月の単位でリリースを行い，**顧客からのフィードバックを得ながら**開発していく
  必要な機能から順に開発・改善していく

※リリース：ソフトウェアを顧客・利用者へ納入したり公開したりすること
※フィードバック：　意見やバグ報告を開発現場へ反映させていくこと

計画（けいかく）: planning
顧客（こきゃく）: customer, client
調整（ちょうせい）: adjusting
一度きり（いちどきり）: only once
継続的（けいぞくてき）: continuously



比喩（ひゆ）: metaphor
必要最小限（ひつようさいしょうげん）: minimum necessary
余分な（よぶんな）: extra
拡張性（かくちょうせい）: extensibility

⑤テスティング

- テストを継続的に行う
- テストファースト
  - 先にテストプログラムを作成してからプログラムを書く
  - 「テストプログラムは仕様書である」
  - プログラムを作る者は，最低でも単体テストを用意すべき（そして，書き換えるたびに実行する）

単体テスト（たんたいテスト）: unit test

プログラムを書き換えるたびに（かきかえるたびに）: whenever you changed the program

テストファーストの例

- 指定された1バイトの16進数を10進数に変換する関数 convert を作ろうとしている

先にテストプログラムを作る

```
.........
if ( convert("00") == 0 ){
   printf("OK\n");
}
.........
```

```
convert("01") == 1
convert("ff") == 255
convert("FF") == 255
convert("A0") == 160
......
convert("000") == ERROR
convert("fff") == ERROR
convert("-1") == ERROR
convert(NULL) == ERROR
convert("xyz") == ERROR
.......
```


16進数（16しんすう）: hexadecimal
10進数（10しんすう）: decimal

⑥リファクタリング

第 10 回(Week 10)で演習(Exercise)を行います

- 機能や意味はそのまま保持しながら，プログラムをより良いものへ変化させる(**洗練化**)

- リファクタリングの専門書では，プログラムを書き換えた方がよいような「**怪しい**」コードを「**不吉なにおい(code smell)**」と呼んでいる

不吉なにおいの例

- 「重複したコード」

  同じようなコードが2か所以上で見られたら，1か所にまとめることを考える

```
…
for( i = 0; i < 10; i++ ){
  printf("%d\n", a[i]);
}
…
for( j = 0; j < 100; j++ ){
  printf("%d\n", b[j]);
}
```

```
print_array(int x[], int size){
  int i;
  for( i = 0; i < size; i++ ){
    printf("%d\n", x[i]);
  }
}
```

```
…
print_array(a, 10);
…
print_array(b, 100);
```


重複したコード（ちょうふくしたコード）: duplicated code

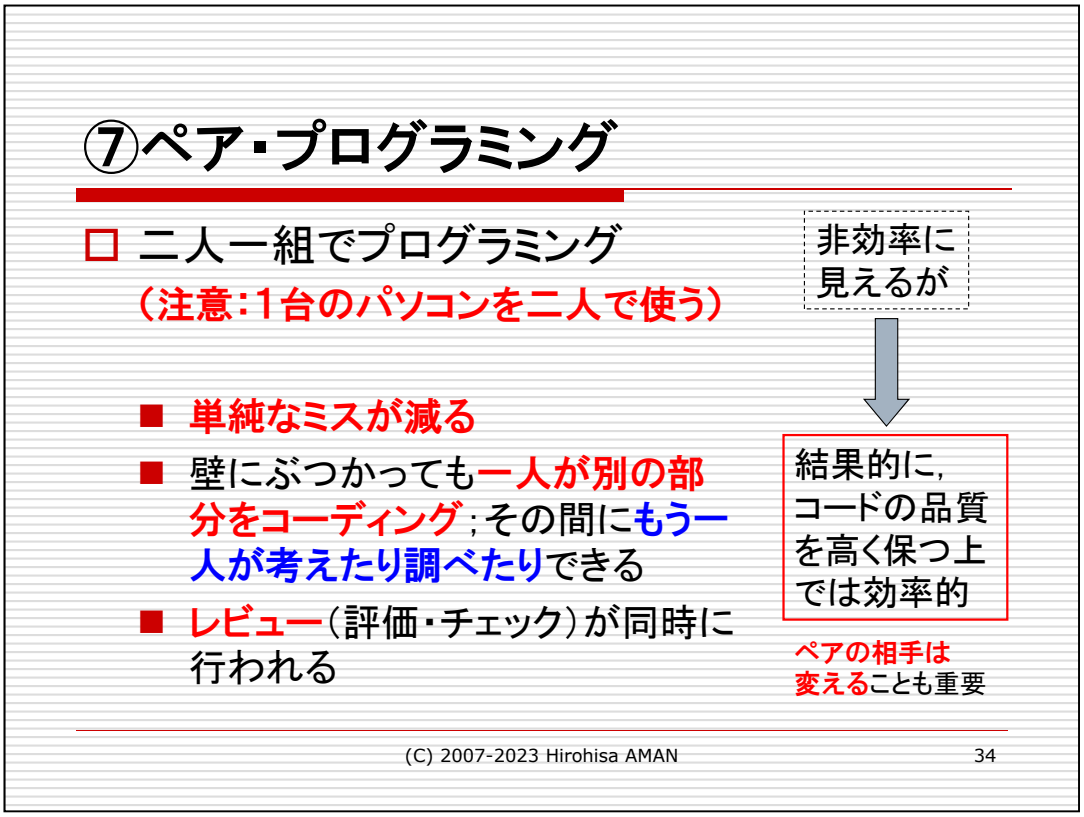
【演習3】リファクタリングを考えよ

長さ4の配列 bin に4ビットの2進数が格納されていて，これを10進数に変換するプログラムを作っている．

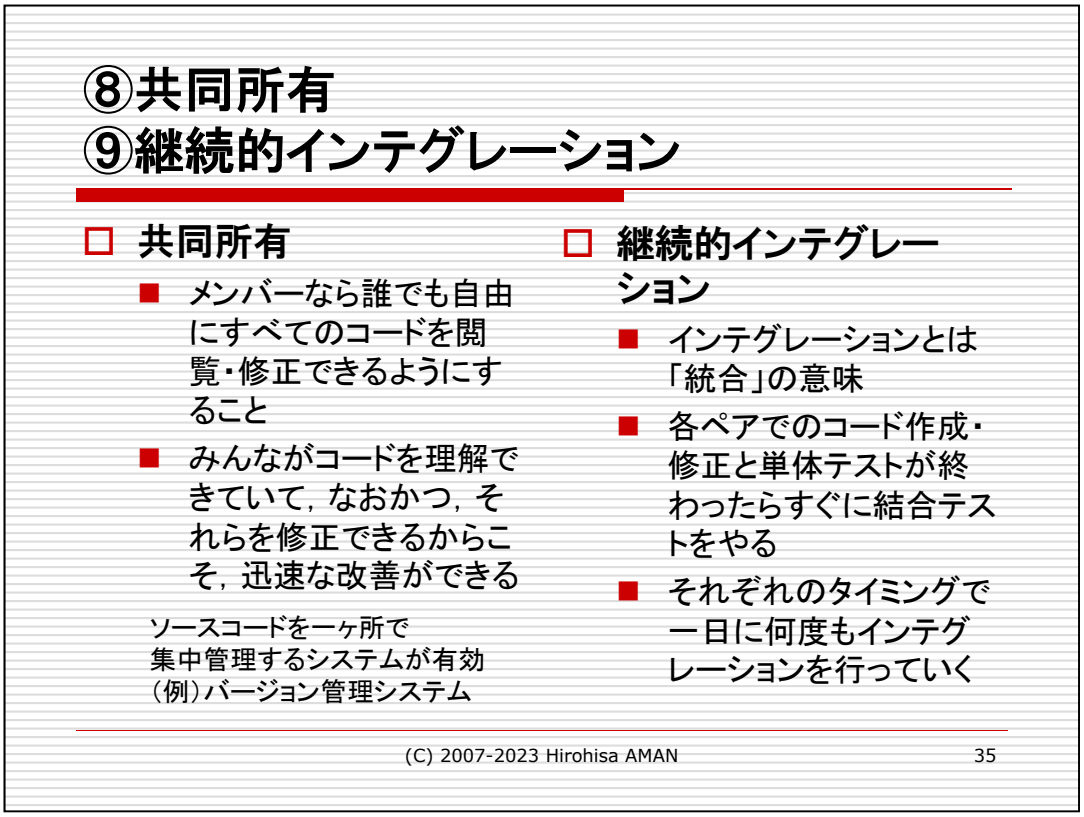
①は良くないので②にしてみたが，もっと良いものにできないか？

①
```
x = bin[3]*8 + bin[2]*4
  + bin[1]*2 + bin[0]*1;
```

②
```
x = 0;
for ( i = 0; i < 4; i++ ){
  if ( i == 0 ) p = 1;
  if ( i == 1 ) p = 2;
  if ( i == 2 ) p = 4;
  if ( i == 3 ) p = 8;
  x += bin[i] * p;
}
```



壁（かべ）にぶつかる: meet an obstacle



共同所有（きょうどうしょゆう）: share

閲覧（えつらん）: browse

迅速な改善（じんそくなかいぜん）: quick improvement

集中管理（しゅうちゅうかんり）: central control

継続的インテグレーション（けいぞくてきインテグレーション）: continuous integration, CI

⑩週40時間労働
⑪オンサイト顧客

週40時間労働

- １日８時間×５日間の労働時間を「目標」にする
- 疲れていたり，残業では生産性は上がらない

どちらか一人の疲労はペア・プログラミングの崩壊を招く

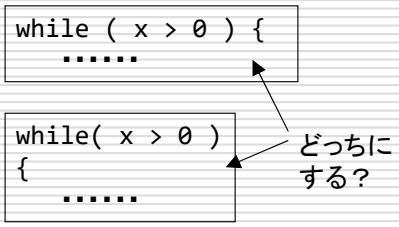
オンサイト顧客

- オンサイトとは「現場での」という意味
- 顧客の代表を開発チームに入れる，あるいはいつも近くにいてもらう
- アジャイルな開発を行う上で，顧客との頻繁な打ち合わせは大事

崩壊（ほうかい）: collapse

⑫コーディング標準

- ソースコードの書き方（スタイル）は人によってまちまち
- 我流の書き方は他人にとってわかりにくい
- 組織でルールを決めておく
  - 変数名の付け方
  - コメントの書き方
  - 括弧の付け方
  - インデントの付け方

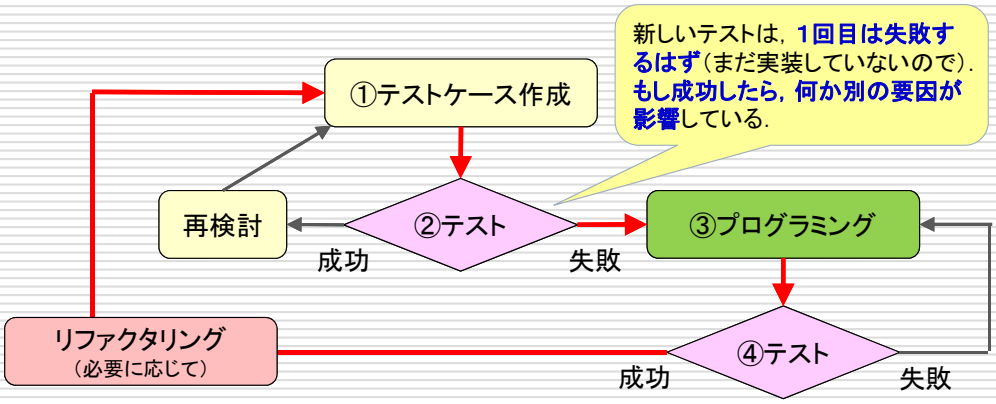


コーディング標準（コーディングひょうじゅん）: coding standard
まちまち: differ from person to person
我流（がりゅう）: one's own style

（テスト ＋ プログラミング）テスト駆動開発

第 11 回（Week 11）で演習（Exercise）を行います

「**先にテストケース**を作り，**次に，それに合格するようなプログラム**を書く」ことを**繰り返す手法**



テスト駆動開発（テストくどうかいはつ）: test driven development
合格する（ごうかくする）: pass
成功（せいこう）: success
失敗（しっぱい）: fail

テスト駆動開発の特徴

- 目標が明確になる
  - 用意したテストケースについて正しく動作するようにプログラムを書くことになる
  - **テストケースが具体的な仕様書**の役目を果たす
- バグの見逃しを防ぐ
  - テストを何度も繰り返しながらプログラミングが進むため，**途中でバグを混入しても気付きやすい**

目標が明確（もくひょうがめいかく）: clarify the goal
具体的（ぐたいてき）: concreate
バグの見逃し（バグのみのがし）: overlooking bugs

まとめ

- プログラミング作法： **分かりやすさ**が大事

- アジャイル開発の基礎
  - **テストファースト**
  - リファクタリング
  - **ペア・プログラミング**
  - **コーディング標準**

プログラミングといえども
個人プレイではなく
チームプレイ（の感覚）
で臨むと良い

宿題(homework)

“[09] quiz” に答えなさい (明日の 13 時まで)

Answer “[09] quiz”
by tomorrow 13:00 (1pm)

注意:quiz のスコアは final project の成績の一部となります
(Note: Your quiz score will be a part of your final project evaluation)

【演習1】関数名を考えよ(解答例)

- ヘルプを表示する

  void print_help ( )

（別解）
printHelp
isLeapYear
copyArray

- うるう年(leap year)かどうかを調べる

  int is_leap_year (int year)

- 配列(array)の内容を別の配列へコピーする

  void copy_array (int src[], int dest[], int length)

  あるいは array_copy

【演習2】ハードコーディングについて（解答例）

```
#define SIZE 256
#define DATA_FILE "foo.txt"
```

①変更に強い

配列の長さを変更する場合，256 を書き換えればOK（いちいち置換しているとミスを誘発）

もしも「SIZE-1」の意味で「255」が書かれていたら変更漏れに

②意味を把握しやすい

数字や文字列だけではその意味を誤解される場合もある（同じ「256」でも「あっちとこっちは別の意味」ということもありえる）

③設定管理をやりやすい

固有の設定を一ヶ所（プログラムの先頭部分）にまとめることができるため，変更・確認が容易

置換（ちかん）: replacement
ミスを誘発（ゆうはつ）: cause a mistake
変更漏れ（へんこうもれ）: omission of modification
意味を把握（いみをはあく）: comprehend the meaning (intention)
設定管理（せっていかんり）: configuration management
固有の設定（こゆうのせってい）: setting specific to the program

【演習3】リファクタリングを考えよ（解答例）

```
x = 0;
for ( i = 0; i < 4; i++ ){
  if ( i == 0 ) p = 1;
  if ( i == 1 ) p = 2;
  if ( i == 2 ) p = 4;
  if ( i == 3 ) p = 8;
  x += bin[i] * p;
}
```

```
x = 0;
p = 1;
for ( i = 0; i < 4; i++ ){
  x += bin[i] * p;
  p *= 2;
}
```

p=1 を初期値として1回目はそのまま使用し，それ以降では p を2倍にしていく