ソフトウェア工学

2022/11/14

海谷 治彦

目次

- コーディングの入力は？
- Coding Standards
- イデオム
- 契約による設計 (Design by Contract)
- 防衛的プログラム
- 例外処理
- セキュアコーディング，設計等
- コーディングの統合戦略
- MDD

コーディングの入力は？

- アーキテクチャが決まりました.
  - 配置図等
- モジュール設計もできました.
  - クラス図等

- 無論，これらをプログラム群にしないと動きません.
  - システムは複数の要素(プログラム群，ハード群，人等)から構成されて，
  - それらが，相互に連携しあって要求を満たします.
- 本日は，プログラミングに関するソフトウェア工学の話題を取り上げます.

コーディング標準

- 会社や開発グループにおいて，言語毎に，どのようにコードを書くかを決めている場合が多い.
- これをコーディング標準と呼ぶ.

1. 段付け（インデント）の付け方
2. クラスや変数の命名の仕方
3. コメントの書き方

- 上記の1については，ツール等によって，自動チェックや修正，強制されるものもある.

なぜ標準化するのか？

- 作った人じゃない他人が読んだり直したりすることが多いから.
- 標準化することで，他人の読み書きの効率や品質を格段に向上させることができる.

例 Java標準

- Javaを提供する会社オラクル(旧Sun)によってきめられている.

https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

- 大抵，どの組織もこれに準拠する.
- ファイル名，フォルダ構成，インデント，コメント，変数クラス等の命名規則等がきまっている.
- これは守らないとアカン.

インデントの流儀

- C言語では多数の流儀がある模様.

◆Visual Studio マイクロソフト？

◆LLVM コンパイラ標準化プロジェクト

◆Google

◆Chromium オープンソースのブラウザ

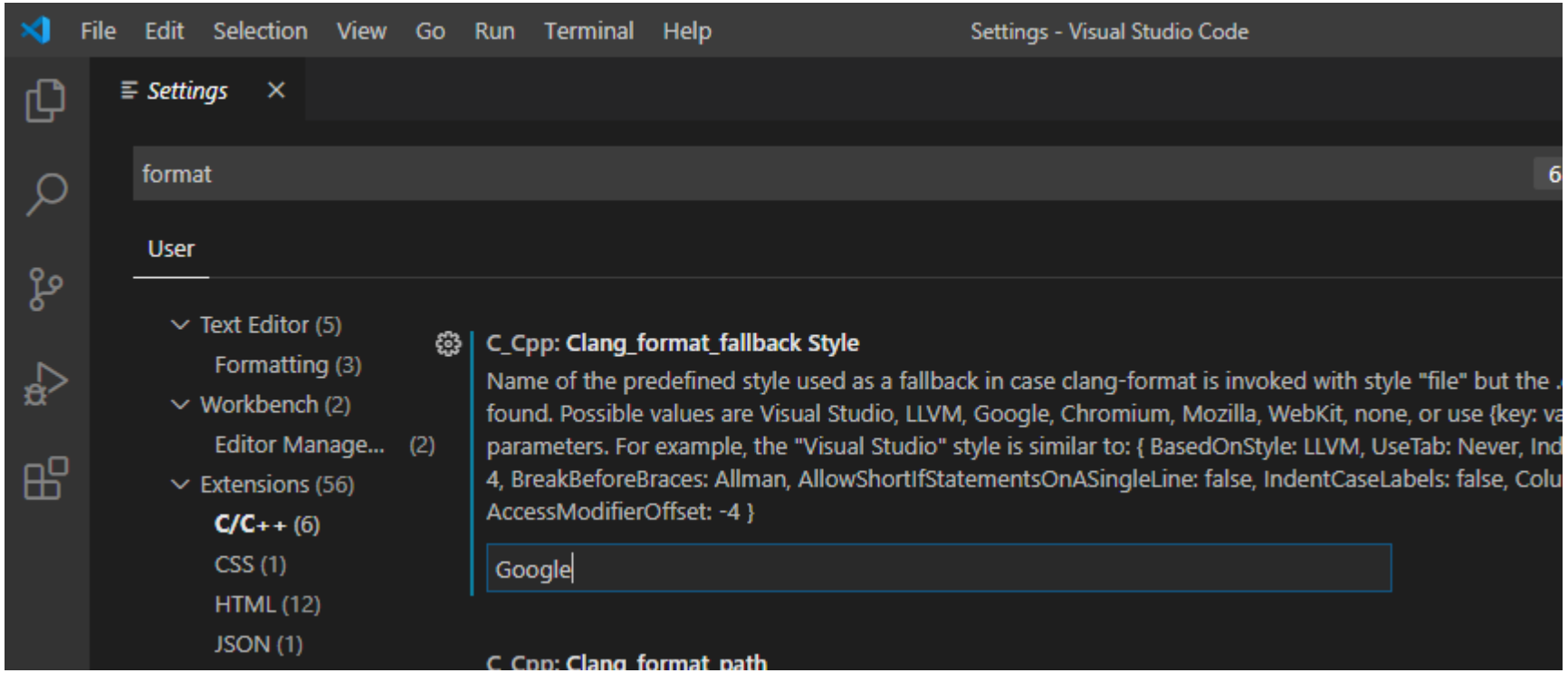
◆Mozilla firefoxとか作ってる

◆WebKit HTMLレンダリングエンジン

- とりあえず，VScodeでサポートされているものを列挙した.

VScodeでのフォーマット変更

- File > Preferences > Settings から，format で検索
- C/C++ を選択，再設定後，要再起動.



Visual Studio と Google はかなり違う

barchart.c

C: > LEC > webpage > html > se > se06sample > C barcha

```c
// list6-4x.c
#include <stdio.h>

int main(void)
{
  int i = 0; // 行を数える
  while (i < 10)
  {            // 行を書くループ
    int j = 0; // *の数を数える
    printf("%d ", i);
    while (j < i)
    { // *を書くループ
      printf("*");
      j++;
    }
    printf("\n");
    i++;
  }
  return 0;
}
```

barchart.c

C: > LEC > webpage > html > se > se06sample > C barchart.c

```c
// list6-4x.c
#include <stdio.h>

int main(void) {
  int i = 0;         // 行を数える
  while (i < 10) {   // 行を書くループ
    int j = 0;       // *の数を数える
    printf("%d ", i);
    while (j < i) {  // *を書くループ
      printf("*");
      j++;
    }
    printf("\n");
    i++;
  }
  return 0;
}
```

マニュアル自動生成

- コメント文からマニュアルを自動生成する機能がある言語もある.
- Java, Perl, Python も
- Javadoc, sphinx 等

- 多言語対応したツール doxgen もある.

- 関数やクラスの説明をコメントから自動生成したいなら，それぞれのツールのルールに従いマニュアルを書くこと.

Javadoc の例

trump

クラス Player

java.lang.Object
  └trump.Player

public abstract class **Player**
extends Object

プレイヤーを表すクラス。

フィールドの概要

protected Master	**master** 進行役
protected Hand	**myHand** 手札
protected String	**name** 名前
protected Rule	**rule** ルール
protected Table	**table** テーブル

コンストラクタの概要

Player(String name, Master master, Table table, Rule rule) コンストラクタ。

メソッドの概要

abstract void	**play**(Player nextPlayer) 順番を指名する。
void	**receiveCard**(Card card) カードを配る。
String	**toString**() プレイヤーの名前を返す。

コンストラクタの詳細

Player

```
public Player(String name,
              Master master,
              Table table,
              Rule rule)
```

コンストラクタ。

パラメータ:

name - 名前
master - 進行役
table - テーブル
rule - ルール

メソッドの詳細

play

```
public abstract void play(Player nextPlayer)
```

順番を指名する。実際の処理はサブクラスで記述すること。

パラメータ:

nextPlayer - 次のプレイヤー

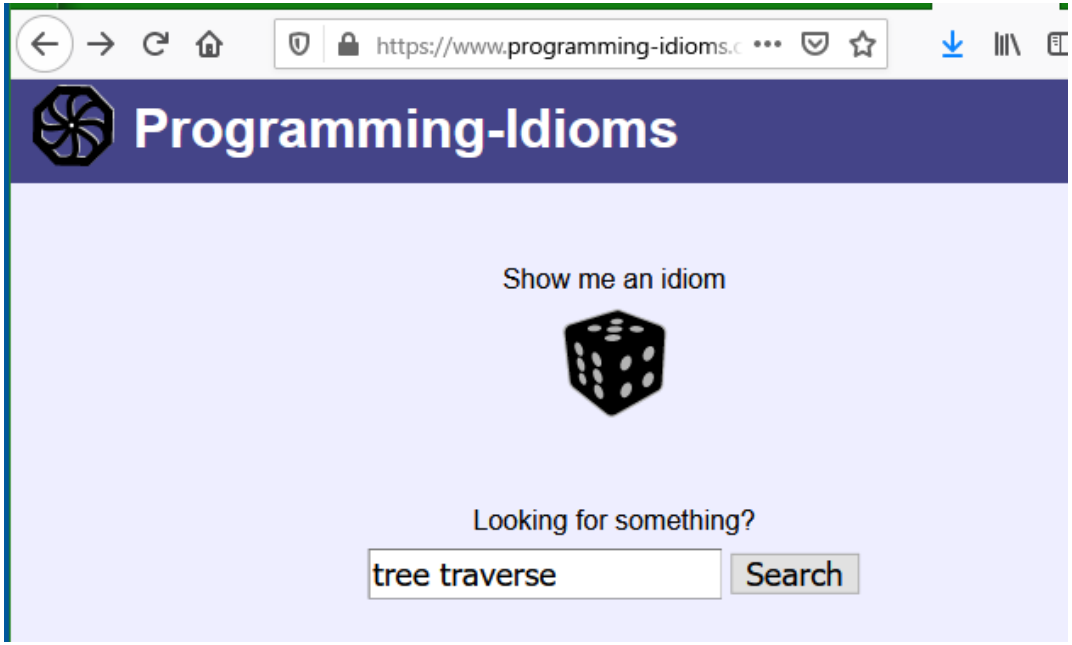
もとになってるソース

```java
package trump;

/**
 * プレイヤーを表すクラス。
 */
public abstract class Player
{
    /** 名前 */
    protected String name;

    /** テーブル */
    protected Table table;

    /** 手札 */
    protected Hand myHand = new Hand();

    /** 進行役 */
    protected Master master;

    /** ルール */
    protected Rule rule;

    /**
     * コンストラクタ。
     *
     * @param name 名前
     * @param master 進行役
     * @param table テーブル
     * @param rule ルール
     */
    public Player(String name, Master master, Table table, Rule rule)
    {
        this.name = name;
        this.master = master;
        this.table = table;
        this.rule = rule;
    }
```

イデオム

- 特定のアルゴリズム等を実装する際の言語毎の典型的な書き方.
- 無論，アルゴリズムを見ながら，自分でごりごり書いてもよいが，検索したほうが早いかも.

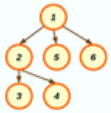
https://www.programming-idioms.org/ 等



検索結果の例

Idiom #18 Depth-first traversing of a tree

Call a function *f* on every node of a tree, in depth-first prefix order



C++ D Dart Go Haskell JS PHP Perl Python Ruby Rust

Idiom #16 Depth-first traversing of a binary tree

Call a function *f* on every node of binary tree *bt*, in depth-first infix order

D Dart Fortran Go Haskell Haskell JS Java Java Java PH

JS

```
function DFS(f, root) {
        f(root)
        if (root.children) {
                root.children.forEach(child => DFS(f, chil
d))
        }
}
```

関数等の呼び出し関係

- C等の手続き型言語では，関数を繰り返し呼び出すことで，処理が行われる．
- Java等のオブジェクト指向言語でも，メソッドを繰り返し呼ぶことで，処理が行われる．
- よって，関数やメソッドの呼び出し関係を明確にすることがプログラミングにとっては重要となる．
- 関数を呼ぶ側(Caller)と呼ばれる側(function)の関係について，大きく異なる二つの流儀がある．

1. Design by Contract 契約による設計
2. Defensive Programming 防衛的プログラム

事前条件，事後条件

- 関数や一連の処理は以下の二つの条件で仕様化されている.
- 事前条件: その関数等が呼び出される前に成り立っているべき条件
  - 基本的には入力値と状態値で表現できる.
  - 例 ルートを計算するには，正の整数を入力すること.
  - 最大公約数の計算は1以上の整数を2個以上入力すること.
- 事後条件: その関数等の処理が終わった後に成り立っている条件.
  - 入力値，出力値，状態値で表現できる.
  - 例 ルート計算では，結果^2 = 入力

二つの流儀の違い

- Design by Contract
  - 事前条件の整備は，関数等を呼び出す側の責任である.
  - よって，関数側ではチェックしない.（成り立っているのが当然として処理を書く）
- Defensive Programming
  - 事前条件の整備は，関数側でも呼び出す側でも両方でチェックしましょう.
  - どっちかが忘れてたり，ミスがあったりしても安心ダネ.

議論

- 関数等の責務の明確化という観点からは，明らかに Design by Contract のほうが正しい.
- ただし，現実的には Defensive Programming せざるを得ない.
- ほとんどのプログラム言語では，事前条件を記述してチェックする機構が無いため.
  - 処理と，その仕様を明示的に関連付ける機構が無い.

- ただ，むやみに関数と利用側，双方で条件チェックをするのは，明らかに冗長である.
- 加えて，「どっちかでやってるだろう」という緩みが欠陥を引き起こす.

例外処理

- 例外とは，非形式的には，システムの実行を中断させる正常でない出来事である.
  - 極端な例: あるプロセスに許可されてないメモリのアクセスで，OSがそのプロセスを停止させる等
- より厳密に以下のような定義が提唱されている.

- 関数等の失敗(failure): 関数等の呼び出しが事前条件および事後条件を満たさずに終了すること.
  - 事前および事後条件は，その関数の仕様に依存する.
- 例外(exception): 関数等の呼び出しが失敗となる実行時の出来事(event).

例

- 割り算 x/y
  - 事前条件は 割る数 y≠0
  - 実行時に出来事(event)として，x=10 y=0 を与えた
  - 事前条件を満たさないので例外が発生する.
- Webフォーム入力したメッセージ m を公開
  - 事前条件 メッセージ m は，Web上で特別な意味を持つ文字列ではないこと.
  - 実行時のeventとして，m に，怪しいサイトに誘導するJavaScriptを入力した.
  - 事前条件を満たさないので例外が発生する.

（通常はサニタイズしてJSを無効化しますが・・・）

例外を処理

- 関数等は，その振る舞いが明確に仕様として書かれているのは大前提.
- その仕様（通常，事前と事後条件）が満たされないと，例外が発生する.
- 発生した例外は，関数等を呼び出した側が受け取ることになる.
- 呼び出し側の処理の種類は以下の二種類しかない.

1. 条件を整え直して，再度，その関数等を呼ぶ.
2. その呼び出しでは対処不能として，失敗を宣言し，さらに上位の呼び出しがあれば，そちらに例外をたらいまわしにする.
   - mainまで戻ってしまえば，プログラム全体の実行が失敗となる.

実践的な例外処理記述

- 多くの言語では，try catch throw 等の語彙を使って，例外を処理する.
- tryブロック 例外が発生しうる関数等の呼び出しの範囲を区切る.
- catch 例外の種類毎の対処法を書く. ココで，条件を整え直して，再試行する記述があるのが好ましい.
- throw その呼び出しでは，手に負えない例外の場合は，さらに上位の呼び出しに，例外を投げる（たらいまわしする）ための命令.

例

```java
/**
文字列を入力.
@return 入力された文字列．不備があるとnullが返る.
*/
public static String readln(){
    String s;
    if(br==null) return null;
    try{
        s=br.readLine();
    }catch(IOException e){
        return null;
    }
    return s;
}
```

```java
/**
引数として与えられた配列の5番目の要素を返す.
引数の長さが足りない場合，自分では対処せず，
呼び出しもとに例外を投げて対処を委譲(たらいまわし)する.
*/
public String five(String[] a) throws Exception{
    String r=null;
    try{
        r=a[5-1];
    }catch(Exception e){
        System.err.println("Oh No! I (five) can't handle!");
        throw e;
    }
    return r;
}
```

例外を返り値としてはいけない

- 例外は，機能としては，関数等の呼び出し側への値の引き渡しと同じである．
- よって，構文上は，return と同様の役割を担える．
- しかし，例外は前述のような意味合いがあるので，そのような使い方はしてはいけない．
- 以下は悪い例

```
/**
 例外だって，メソッド呼び出し側に結果を通知するのにつかえるじゃん，
 ってことで，投げられる例外に計算(足し算)結果が埋め込まれているメソッド．
 これでメソッドの返り値なんていりませんね(笑)
 @throws 足し算の計算結果がメンバーresultとして埋め込まれている．
*/
public void add(int x, int y) throws AddException {
    AddException a=new AddException();
    a.result=x+y;
    throw a;
}
```

セキュアコーディング

- 今日の情報システムはリスクにさらされている.
- リスク = 情報資産 × 脅威 × 脆弱性
- システムで情報資産を扱わないのは，なかなか難しい.
  - クレジットカード番号や個人情報等
- 脅威の存在も開発者，運用者側では制御できない.
  - 世界中の人々が良き人になればよいのだが，実現は難しい.
- セキュアコーディングは，脆弱性をシステムに埋め込まないための実践といえる.

脆弱性とは

- 多数の定義があるが，
  - ✓想定した用途以外の動作をするように用いられる可能性

がもっともしっくりくる．

- 端的に言えば，システムの弱点といえる．
- 現代のシステムは部品化や継続的な開発が進んでいるため，脆弱性を含む場合が多い．

□脆弱性の分類

1. 既知のもの
2. 未知のもの

既知の脆弱性

- 特に外部のシステムやライブラリを利用するアプリの場合，その適切な利用制限を行わないことで脆弱性が生まれる.
  - ユーザー入力が直接にDBMSへのコマンドになってしまう等，いわゆる SQL injection
- 既知の脆弱性のカタログは存在する.
  - CVE Common Vulnerabilities and Exposures
  - CWE Common Weakness Enumeration
- しかし，それら全てをアプリ開発者がチェックして対処しているとはいえない.

未知の脆弱性に対して

- 原則，前もって対策するのは非常に難しい．
- 開発に余裕があるなら，システムが扱う資産(Asset)を起点として，その保護を洩れなく行うくらいのことしかできない．

情報セキュリティの基本

- セキュリティの基本は資産の保護である
- 情報に関しては以下の3つが重視される.

1. Confidentiality 秘密にすること
2. Integrity 不正改ざん等されないこと
3. Availability 必要時に必要な利用ができること

- 加えて，以下も重視される.

a. Authenticity
b. Accountability ログとりに相当
c. Non-repudiation 人が行為（普通は悪いこと）をしてないと言い張ることを防ぐこと.

セキュアコーディングの原則

1. Economy of mechanism
2. Fail-safe defaults
3. Complete mediation
4. Open design
5. Separation of privilege
6. Least privilege
7. Least common mechanism
8. Psychological acceptability

C/C++ セキュアコーディング アスキー 2006より

1. Economy of mechanism

- 単純で小さな設計を心掛けること.
- 仕組みを単純に.
- システムは小さく単純明快に設計する.

- Keep it short and simple (KISSの法則と呼ばれるらしい)

2. Fail-safe defaults

- 誤操作を安全に処理する.
- 誤操作・誤動作による障害が発生した場合でも常に安全側に制御するという考え方.
- 基本，システムが提供する機能を凍結するのが安全側・・・
  - とはいえ，Availability の観点からは問題ある.

- ユースケース記述の例外系列において「オチ」として，システムや，それが管理する資産が回復不能な状態にならないようにする.
  - データのクラッシュ，データが公開状態になる，飛行機の墜落等

3. Complete mediation

- セキュリティ機構が漏れない適用できるようにする.
- 保護機能が回避されないようにする.
- 一般には Defensive Programing が推奨される.
  - 機能提供側，利用側の双方でチェック.

4. Open design

- 設計内容を秘密にしてはいけない.
- 設計内容を知られても攻撃されないように設計すべき.
- 攻撃者が設計を知らないことは保護にはならない.

- 利用する防衛機構は公開すべき.
  - 当該の機構に問題があれば，第三者機関からの通報を期待できる.

5. Separation of privilege

- システムの機能実施のための権限を分離する.
- 例えば，ある機能と他の機能の実施に異なるカギ（パスワード等）が必要でもよい.
- 加えて，権限付与の判定を多段階および多重で行うべきである.
  - 複数のパスワードや，複数の異なる認証機構の利用（網膜と指紋と声紋の組み合わせ等）

- Defense in depth や Separation of duty のもとになる考え方.

6. Least privilege

- 最小限の権限(privilege)をユーザーや機能に与えること.
- 例えば，名前のリストを得る関数は，それ以外の情報，例えば年齢のリストにアクセスすることを禁止する等.

- プログラム中の関数レベルでアクセスコントロールがされているシステムは少ない.
- 利用者の種類によって，アクセス制御が異なるシステムは多い.

7. Least common mechanism

- 複数のユーザーが共有し依存する仕組みの規模を最小限にする.
- 共有ファイルやオブジェクトはなるだけ避ける.

- この意味からは，RBAC による役割の共有化もよくないのかも・・・

8 Psychological acceptability

- 利用者や開発者に受け入れらる保護設計にすべき.
- 基本的に「使いやすく」するしかない.
- セキュリティと使いやすさは，多くの場合，対立するが，そこはがんばって折り合いをつける.
  - 機能を呼び出す毎にパスワードを聞かれてたら，頭痛くなるし･･･しかし，保護のため必要なら仕方ないだろう.

実装の原則

- 前述は設計精神的なものだったが，以下は，実装（プログラミング）の際の原則群

1. Validate input
2. Sanitize data sent to other systems
3. Heed compiler warnings
4. Architect and design for security policies
5. Keep it simple
6. Default deny
7. Adhere to the principle of least privileage
8. Practice defense in depth
9. Use effective quality assurance techniques
10. Adopt a secure coding standard

1. Validate input

- 全ての信頼されないソースからの入力の妥当性を確認(validate)する.

- コマンドライン引数，ネットワーク，環境変数，ユーザー管理のファイル，ユーザー入力, DBからの値
- Webの場合: ユーザー入力，クッキー，HTMLヘッダーブロック等，クライアントから送られた物全て

2. Sanitize data sent to other systems

- 外部に渡すデータは渡した先で問題が起きないように加工する.
- Sanitize 消毒等によって衛生的にすること
- 受け取る外部システムによって，何が問題かは異なるので，異なる加工をすること.
  - 例: Webページに渡すなら，JavaScript等が入っているのは一般にヤバイ.
  - 例: DBに送るなら，SQL命令的なものはつぶす.
- その他，意図していない機能が呼び出されるような可能性はつぶしておく.
  - 引数型の違いによって，呼び出し関数を変更する言語等は特に注意.

3. Heed compiler warnings

- コンパイラの警告に注意を払え.
- コンパイラのチェックオプションはなるだけ多くつけよ.
- コンパイラに加えて，コードの静的，動的解析ツールによる分析もできるとよい.

4 Architect and design for security policies

- セキュリティポリシー実現のための設計と実装を行うこと.
- ポリシーには，資産の重要度のランク付けがされているようだ.
- それに従い，ランクの高いものから重点的に保護すべき.

5. Keep it simple

- 仕様に対して，それを単純に実現したコードにすること.
  - 仕様に求められたより簡単な方法で保護することを意味するのでは無い.
  - 要は妙に複雑で凝った実装はしない.
- その方が，脆弱性が含まれる可能性が結果として少ない.

6. Default deny

- 拒否をデフォルトとする.
- 基本，アクセスは出来ない状況で，権限や条件があった場合のみに，機能やデータへのアクセスができるようにする.

7. Adhere to the principle of least privilege

- 基本，設計原則 6. least privilege に同じ.
- 特に実行されたプログラムのプロセスが特権命令を実行可能な時間を最小化することらしい.
- むやみにシステムコールを呼ばない的な感じ.

8. Practice defense in depth

- 設計原則5 Separation of privilege と意図は同じ.
- 多段防御を実装すること.
- コードだけでなく，実行環境の選択等も含まれる.

9. Use effective quality assurance techs.

- 効果的な品質保証技術を用いる.
- 基本，テストを行うこと.
- レビューも有効.
- 予算があれば，数学的な手法による保証もあり.

セキュリティレビュー

- レビューとは，基本，コードや設計等の成果物を，手順に従って人がチェックすること.
- 会議形式で複数で行う場合もある.
- セキュリティの場合の留意点

1. ポリシーを満たしているか
2. セキュリティ要求に対する対策に漏れが無いか
3. 既知の脆弱性対策をうっているか

10 Adopt a secure coding standard

- セキュアコーディング標準を用いること.
- 例 JPCERT CC より
- Cの標準例
  - 配列: 境界外を指すポインタや配列添字を生成したり, 使用したりしないこと・・・
- Javaの標準例
  - データメンバはprivate宣言し, それにアクセスするためのラッパーメソッドを提供する.
    - JavaScriptだと, なかなか厳しい.

https://www.jpcert.or.jp/securecoding/

具体的なセキュリティ機能群

- 具体的には以下のような機能の例がある.
- 必要十分な機能リストではないので注意.

1. Authentication 認証
2. Authorization 認可
3. 暗号化
4. 入力の検証
5. 出力の無害化
6. 例外処理
   - ユースケース記述の例外系列のこと. Java 等の Exception handling のことではない.
7. ロギング

脅威モデリング

- 開発するシステムへの，どのような脅威があるかを分析するための作業の一種.
- テストやレビューと異なり，設計しか無い段階でも分析可能.
- 作業ステップ: 基本以下を繰り返す

1. 資産の把握
2. 全体像の把握
3. アプリの分解
4. 脅威の特定: STRIDE 等が役に立つ
5. リスクの格付け: 後述

STRIDE

- 脅威洗い出しのための仮想的な攻撃の帰結のカタログ.
- こんなことが起こるのは，どんな脅威があるからだろうと考える.

1. Spoofing Identity なりすまし
2. Tampering with data 改ざん
3. Repudiation 否認，利用者が行為を行った事実を否定すること．やってないって言うこと.
4. Information Disclosure 情報の漏洩
5. Denial of Service サービス妨害
6. Elevation of Privilege 権限昇格

コードの統合戦略

- コードは複数の関数やクラスから構成されている.
- これらを，1個の実行可能ファイルに統合しないと当然動かない.
  - いわゆるリンケージ（動的な場合も含めて）
- ばらばらの関数やクラスを作っていく順番として，以下の三つの流儀がある.

1. トップダウン
   - メイン関数/メソッドの部分から順に下請け部品まで作ってゆく.
2. ボトムアップ
   - トップダウンの逆，部品から作ってゆく.
3. 上記の併用

作っていく順番を何故気にする？

- ACMが教えろっていってるから.
  - はっきいって何をいまさら？と思う.
- あらかじめ，設計を行っていれば，ボトムアップだろうかトップダウンだろうがあまり関係ない.
  - クラス図やデータフロー図が書かれている.
- あえて意味があるとすれば動作テストのやり方が異なるくらい.

◆ボトムアップの場合: 単体テストをしやすい.

◆トップダウンの場合: 下請け部品のふりをする部品（スタブと呼ばれる）の準備の必要がある.

モデル駆動開発

- Model Driven Development (MDD)
- 夢の自動プログラミング ？
- 基本的にUML的なモデルを段階的にコードに変換してゆく自動化.
- PIMからPSMへの変換
  - Platform Independent Model
  - Platform Specific Model (例えばWindows上で等)
- そんな旨い話は・・・
  - 仕様に近いモデルもかなり厳密かつ数学的な記述が必要.
  - PSMに変換するにしても, Platform の詳細な Profile が予め必要.

それでも利点はある

- 仕様が変わったら，それを直接，コードに反映できる.
- ・・・・これくらい？

MBD と MDD の違い

- Model Based Design (MBE) というのがある.
- 考え方や理念はMDDを同じだが，MBEは機械制御の分野を主なターゲットにしている.
  - モーター制御，自動車，家電製品など.
- よって，MBE でのモデルは機械等の物理的な世界の模型のことを指す.
  - MATLAB 等の制御工学系のツールを利用する.
- 一方，MDD は普通のアプリケーション・ソフトウェアを対象としている.

- 現実的には MBD と MDD は別物と考えたほうがよい.

ローコード開発

- Low-code development
- あんまりプログラムを書かないでもアプリが作れる開発環境.
- No-code とまで言い張るものもある.
- 基本，GUIで画面にボタン等をぺたぺた張って，アプリができてしまう.
- ワークフローを与えることである程度，複雑なアプリもあまりコードを書かずに作り上げられる.
- クラウドベースのものがほとんど，大抵，サブスクリプション型の料金体系.
- 例: Google AppSheet, MS PowerApps

ローコード開発とMDD

- ある意味，実践的なMDDのプラットフォームとなっている.
  - 初期のMDDよりも，かなり安価で実際に適用する敷居も低い．手軽に使える.
- 基本的にデータモデル，振る舞いモデル（ワークフロー）を手作業でコードに落とす必要は無い.
  - よって，データや手順の変更をコードに反映させるのが容易である.
- モダンなフレームワーク同様，組み合わせる部品等の知識が多数必要で，修得がめんどくさい.
- 以前あったエンドユーザー・コンピューティングの考えを継承している.

本日は以上