良いプログラムとは?

美しいソースコードとは:
次の七つを満たしたソースコードである。

1. **意図を表現:**
   - 意図が表現されていること。
   - 意図の理解が容易であること。
   - 意図以外の記述が少ないこと。
   - How (どうやってやるか) でなく What (何をやるか) が記述されていること。
   - できれば、Why (なぜやるか) も記述されていること。
2. **単一責務:**
   - (型やメソッドや変数などの) プログラムの単位が唯一の仕事を記述していること。
   - 且つ、(なるべくなら) その仕事がそのプログラム単位内で記述されつくされていること (=高凝集: high cohesion)。
3. **的確な名前:**
   - (型やメソッドや変数の) 名前が、それの (唯一の) 仕事を (一言で必要十分に) 表現していること。
   - 同じものは同じ名前で、違うものは違う名前で表現されていること。
     - 既知の名前を別の意味で使用しない。
       - 問題領域の言葉を意味を変えて使用しない。
   - それであるものとそれでないものの区別が付くこと。
4. **Once And Only Once:**
   - 同じ意図のものが重複して書かれていないこと。
5. **的確に記述されたメソッド:**
   - メソッド内が同じ抽象度の記述の集まりで構成されていること。
   - メソッド内が、自然な粒度で記述されていること (話し言葉のように)。
   - ほどよい量 (記述が多過ぎない)。
6. **ルールの統一:**
   - 全体が同じルールに従っていること。
7. **Testable:**
   - 正しい記述であることが分かるようになっていること。
   - 検証 (verification) が容易であること。

良いプログラムとは?

ソース
URL: https://ja.wikipedia.org/wiki/%E6%9C%AC%E7%89%A9%E3%81%AE%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9E%E3%81%AFPascal%E3%82%92%E4%BD%BF%E3%82%8F%E3%81%AA%E3%81%84

プログラムを「自然言語で説明」できることは重要だが、ときに「数式や図で説明」できることも重要だ。C# で、コメントとして「自然言語で説明」だけでなく「数式や図の

プログラムを「自然言語で説明」できることは重要だが、ときに「数式や図で説明」できることも重要だ。C# で、コメントとして「自然言語で説明」だけでなく「数式や図の説明」も (HTMLの <math> や <img> か何かで) 入れられるようにしてはどうか。Visual Studio で見られるような。

「コメントがなくても意図まで掛けるように言語を含めたプログラミングツールが進化していく方向」が好ましい。文芸的プログラミングの方向かな。「ドキュメントをコンパイルしたら動く」でも同じだけど。プログラム コードと同じ人が読むドキュメントは別々に書いて管理しない方が良いと思う。

「進化の方向性を見定める」重要

委譲

委譲
delegation
https://ja.wikipedia.org/wiki/%E5%A7%94%E8%AD%B2
http://codezine.jp/article/detail/3710

「それはこのクラスの役割なので」
「委譲するかどうかと『このクラスの役割』かどうかは別問題」

オブジェクトの役割
= 外部に向けてどのようなサービスを提供するか

サブルーチンへの委譲
Main()

Sub();

オブジェクトへの委譲

良いプログラムとは?

ソース
URL: https://ja.wikipedia.org/wiki/%E6%9C%AC%E7%89%A9%E3%81%AE%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9E%E3%81%AFPascal%E3%82%92%E4%BD%BF%E3%82%8F%E3%81%AA%E3%81%84

良いプログラムとは?

「ソフトウェアの品質とは、優れたプログラムが備えるべき七つの属性の集合で
ある」
七つの属性を平均的なプロジェクトでの優先順にあげると以下の通り。

信頼性 (reliability)
使用性 (usability)
理解容易性 (understandability)
変更容易性 (modifiability)
効率 (efficiency)
検証性 (testability)
移植性 (portability)

「動くのが重要」なのはその通り。その上で保守性が重要。

ソフトウェア開発は複雑さとの戦い

ソフトウェアのエントロピー

要求が大きくなり、実行環境も多様化

どんどん複雑に

複雑さの解消を行う手段

基本は「分けること」

10000行 より 100行×100

「どう分ける」ともっともシンプルか。

・分かりやすさ
・変更が起こったときの複雑さ
・テストしやすさ

複雑性の排除
　どうやったらシンプルにできるか
　　問題の切り分け
　　サブルーチンに分ける
　　クラスに分ける

　　分け方が大事
　　それぞれがシンプルになるように
　　単一の問題を扱う
　　一つの問題を分散させない
　　例. プログラム単位は、単一責務の法則に従う
　　例. デバッグ時には、発生個所をソースコードレベルで特定 → 再現するミニマムコード
　　境界がくっきりとするように
　　インタフェイスを明確に

データは何かを考える

Why → What → How
　How から考えては駄目

設計 ⊃ モデリング

複雑な問題をシンプルにする

問題として扱う部分だけを表現
問題として扱わない部分から問題として扱う部分を「分ける」

ここで扱う問題は何か?
問題の切り分け
関心の分離
説明責任

「なんかうまく動かない」→ 問題の切り分け →「この行が動かない」

シンプルに考えるコツ

ソフトウェア開発のコツ
問題を分けて考えること
※ いかに分けるか? ここが腕の見せ所。

ソフトウェアの原則

構造化手法
オブジェクト指向
アスペクト指向

オブジェクト指向入門

OOとは?
OOで何が良くなる?
オブジェクトとは?

オブジェクト指向の基本的なアイディア

分散処理
それぞれの役割をきちんとこなす

オブジェクト
メッセージを受けて何かする

オブジェクト指向の三大要素

カプセル化
継承
ポリモーフィズム

抽象化
差分プログラミング

それぞれどういうこと?

設計課題
・機能要求
・ユースケース モデルによって表現
・ユーザーから見えるシステムの提供する
サービス

・非機能要求
・システム化に対する機能要求でない要求
・ここは再利用可能にして欲しい
・この部分は将来拡張可能にして欲しい
・既存システムと同じミドルウェアを使って
欲しい

場の定義 (横分割)
・レイヤ分割
・抽象化のレベルで分割

・再利用性の高いのどのレイヤ?
・抽象化による再利用性
・クラス: オブジェクトの抽象化によるオブ
ジェクトの再利用
・パターン: モデルの抽象化によるモデルの
再利用

・場の定義 (縦分割)
・論理ティア
・論理的なオブジェクトの配置空間
・MVC (View, Controller, Model)
・View: 表示を制御するクラス

・Controller: 要求に応じて Model を制御するクラス
・Model: ドメイン (ビジネス ロジックとエンティティ) のクラス
・BCE (Boundary, Control, Entity)
・Boundary: システムの境界線上に位置するクラス
・Control: 制御に関わるビジネス ロジックを持つクラス
・Entity: システムの実体 (永続データ) を持つクラス

・物理ティア
・Web 層
・ビジネス ロジック層
・データベース層

・特性によって開発担当者をティア毎に分けることが可能
・Boundary: 画面デザイナ
・Control: ビジネス ロジックに詳しい開発者
・Entity: データベースに詳しい開発者

・変更箇所の局所化
・Boundary -- Control -- Entity
多 <----------> 少 (変更頻度)

・レイアリング

C++ や C#.NET における論理ティアの表現
・namespace を使用

○ UML とは
・UML の経緯
・特徴
・どんな図があるか

クラス
オブジェクト

関連と集約
多重度

良いプログラムとは

原則

「Once and Only Once」†
「一度、たった一度だけ」同じものを重複して書かない、ということ。これを守らないと、変更によって同じ修正を複数箇所で行うことになる。

最も重要な原則 †
↑
高凝集 (high cohesion) 且つ 疎結合 (low coupling) †
↑
高凝集 (high cohesion) †
クラスは一つの明確な目的と名前を持ち、不整合のない首尾一貫した中身を持っているべき。

↑
疎結合 (low coupling) †
クラスは単純なインタフェイスを持ち、関係のないクラスとは出来るだけ話さないようにすべき。

↑
関心事の分離 (separation of concerns) †
複数の関心事が一つのクラスの中に混在しないようにする。或る関心事を局所的にする (他の関心事とは分けてカプセル化しておく)。

↑
Robert C. Martin の Principles of OOD †
Object Mentor - pood

http://www.objectmentor.com/courses/pood/

↑
OCP (Open-Close Principle) †
開放と閉鎖の法則。

ソフトウェア モジュールは、変更に対して閉じており、拡張に対して開いてい

るべき。

↑
LSP (The Liskov Substitution Principle) †
リスコフの置換原則。

スーパークラス (継承元のクラス) をサブクラス (継承したクラス) で置き換えることが可能であるべき。

スーパークラスへの参照 (やポインタ) を使うメソッド (関数) は、それのサブクラス のオブジェクトを、それとは知らずに使えなければならない。

↑
DIP (The Dependency Inversion Principle) †
依存関係逆転の原則。

上位モジュールが下位モジュールに依存すべきではない。上位モジュールも下位モジュールも抽象概念に依存すべき。

抽象が詳細に依存すべきではない。詳細が抽象に依存すべきである。

↑
ISP (The Interface Segregation Principle) †
インタフェース分離の原則。

クライアントは自分が使わないメソッドに依存することを強制されない。

↑
REP (The Reuse/Release Equivalency Principle) †
再利用・リリースの粒度の原則。

再利用の粒度はリリースの粒度であるべき。

↑
CCP (The Common Closure Principle) †
共通閉鎖の原則。

パッケージ内のクラス群は或る修正に対して閉じている方が良い。従って、将来の変更に対して同じような修正が予想されるクラスは同一パッケージに入れた方が良い。

↑
CRP (The Common Reuse Principle) †
パッケージ再利用の原則。

パッケージ内のクラスは一緒に再利用されるべき。

↑
ADP (The Acyclic Dependencies Principle) †
循環依存禁止の原則。

パッケージ間の依存関係は依存してはならない。

↑
SDP (The Stable Dependencies Principle) †
安定依存の原則。

パッケージの依存関係は、依存元が依存先より安定している方が良い。

↑
SAP (The Stable Abstraction Principle) †
安定抽象の原則。

抽象的な方がより安定しているべき。

↑
SRP (Single Responsibility Principle) †
単一責務の原則。

一つのクラスは一つの責務を持つべき。

クラスに変更が起こる理由は一つであるべき。

良い抽象には良い名前がつく。

↑
Karl J. Lieberherr †
↑
The Law of Demeter †
デメテルの法則。

オブジェクト中の全てのメソッド (関数) は、以下のいずれかの種類のオブジェ

クトのメソッドのみを呼び出すべきである。

それ自身
メソッドに渡されたパラメータ (引数)
それが生成したオブジェクト
直接保持しているコンポーネント オブジェクト
Introducing Demeter and its Laws by Brad Appleton

http://www.cmcrossroads.com/bradapp/docs/demeter-intro.html

↑
Bertrand Meyer †
↑
IOP (Inside-Out Principle) †
内側から外側への原則。

中から外へ向って設計せよ。

モデルを先に設計し、ユーザーインタフェイスは後で設計せよ。

↑
OCP (Open-Close Principle) †
上記参照。

↑
その他 †
↑
Edelman's Law †
エーデルマンの法則。

他人と話すな。(Don't talk to strangers.)

クラスは関係のないクラスには話しかけないようにせよ。

↑
Design by Contract †
契約による設計。

↑
Hollywood Principle †
ハリウッドの法則。

"Don't call us. We'll call you." (貴方から呼ばないで下さい．必要な時に私が呼びますから)

ソフトウェア開発におけるライブラリとしての「フレームワーク」の性格を表す言葉。ユーザーは、フレームワークから呼ばれる必要な部分だけの個別のプログラムを書くことでアプリケーションを作成する。ユーザー プログラムは、フレームワークから制御されることとなる。

↑
GRASPパターン †
Expert
Creator
Low Coupling
High Cohesion
Controller
Polymorphism
Pure Fabrication
Indirection
Don't Talk to Strangers
↑
Layering Principles †
http://martinfowler.com/bliki/LayeringPrinciples.html http://capsctrl.que.jp/kdmsnr/wiki/bliki/?LayeringPrinciples

掲示板などでの初心者の典型的な質問:
教えてください!!! (至急)
「×× したいのですが、いまいちうまくいきません。どなたか分かる方、私にもわかるように教えてください。できれば、具体的なソースコード付きがいいです」

どこが問題?

問題解決能力

科学でいうモデル化と同様。
物理の場合

名前を付けるとはモデリング

名前を付ける →

概念を抽出する作業

その概念とそれ以外との境界を決める
「概念を切り出す」
「問題を切り出す」
「問題とそれ以外を分ける」

例.
サブルーチンを切り出す
＝処理の中の「或るまとまり」を名前を付けることで概念を切り出すこと

Name and Conquer (定義攻略)
Divide and Couquer (分割攻略)

【本日の偏見】

相手のメール一覧に並んだ自分のメールの姿を想像できずに、「お疲れ様です。」とか「ご確認ください。」のような件名をつけてメールを送る人は…

プログラミングが下手。

どういうコンテキストが共有されてて、そのコンテキストの中でどういうネーミングがユニークで分かりやすいか、という視点に欠けてる人はプログラミングに向かない (偏見)。

「件名を利用するのは自分じゃなくて相手」

モデル
図で描いても良い

UML入門

クラス図
フローチャート図

名前付けは、顧客視点で

設計の様々な視点
→
様々なモデル

・角度によって
建築の場合

ソフトウェア開発の場合

・高さによって

鳥の視点、虫の視点

例. ソフトウェア工学は鳥の視点、テーラリングは虫の視点

例. インタフェイスの名前が全体の名前になる
テレビ、電話

コーディング:
「コンピュータに何をやるかを教える」
↓ 視点の変換
「自分がどうしたいのかを自分や他人にも伝える」

名前付けは How でなく What で

ソースコードの説明責任

情報の提供側に説明責任がある

ソースコードは設計を語るべき
ソースコードは意図を語るべき

それ自身が語る
例. アフォーダンス

設計の見える化

C# で書いたモデル
図で描くとしたら? もっとも意図が伝わるシンプルな図は?
頭の中のモデルにもっとも近いものは?
→ 分かりやすさ。

設計と実装を対応させる

シンプルであること → 意図以外のことを排除 → モデル化

モデルとは?
関心の外のものを取り去ってシンプルにしたものがモデル。

「自分の設計モデル」と「自分の実装コード」が乖離?

C#でモデル駆動開発（Model Driven Development）

名前付けはモデリング (＝設計行為)

名前付けはコミュニケーション ツール
コミュニケーションには質が重要

「声に出して読みたいソースコード」のススメ

詳細に伝えるのと正確に伝えることは違う

S/N比

「ある人の付けた名前＝その人の概念モデル」

おかしな名前
その名前を付けた人の頭の中で、きちんと概念が整理できてない
その人自身が、概念とそれ以外のものの境界を把握していない

名前の変更＝モデルの変更（設計の見直し）＝リファクタリング

名前を付けることによって、「概念モデル」が出来上がっとんねん。名前が付いて初めて、概念とその範囲が確立すんねん。
「頭の中にある『それ』をどう呼ぶようにするか」を決めるということは、自分の中にしかなかった、ある関心の範囲の概念、つまり暗黙知でしかなかった概念を、他人にも分かるようにする、つまり形式知化する作業なんや。「線分追加コマンド」の例でいうと、「線分追加コマンド」という名前がないうちは、その概念を他

人さんに伝えんのは困難なことなんや。名前が決まって初めてコミュニケーションに使える。せやから、これはコミュニケーションのためのモデリングやということもできるんや。

自然言語でどう呼ぶか → メソッドなどの名前
メソッドなどの名前が長くなったとき → メソッドなどを抽象的にする → 単機能&シンプルになる → 名前が短くなる

Naming Driven Development

問題記述→ソリューションの記述
→開発を駆動

名前付けのプラクティス：

1. 概念と名前を一致させる
2. 同じ概念には同じ名前を付けて、異なった概念には違う名前を付ける
3. 1つの独立した概念のみを表す名前を付ける
4. 抽象的な概念には抽象的な名前、具体的な概念には具体的な名前を付ける
5. 抽象的すぎて伝わりにくい概念は、メタファ（例え）で表す*5

*5 例えば、Command（コマンド）クラスのインスタンスを作るためのクラスの名前をCommandFactory（コマンド工場）とする、など。

名前付けのアンチ・プラクティス（やったらあかんこと）：

1.

名前の後ろに数字を付ける
例：Calc1、Calc2、Calc3……

2. 省略する
例：（GetNameを）GetNm、（Initializeを）Intl

3. 意味不明の名前
例：TheFunction

4. 名前に、種類（クラス、メソッドなど）や型名を入れる
例：MainClass、FirstMethod、intNumber、doubleValue

5. 統一感がない
例：命名規則がなく、名前の付け方がばらばら

命名のプロセス - kawasima
https://scrapbox.io/kawasima/%E5%91%BD%E5%90%8D%E3%81%AE%E3%83%97%E3%83%AD%E3%82%BB%E3%82%B9

分かりやすいとは:

・分かりやすい
・分かりにくい

どちらでも選べる。
分かりやすい方を選ぶのが得

シンプルな記述がよい。
シンプルなことと簡単であることは違う。
シンプルに表現したり、シンプルな表現を理解するのが、簡単だとは限らない。
例えば、抽象的な記述はシンプルでかつ難解なことが多い。

[Programming][Memo]
Write simple code.

[Programming][Memo]
There's a difference between simplicity and ease.
Writing and reading simple code aren't always easy.
Abstract code isn't easy but simple.

保守性の基本

ひとつの変更 → ひとつの修正

コードが重複しない

どの名前が美しいかは、最終的には、感性による。
主観。

客観的な理論付けは、色々あるが。

カイゼンのループ
PDCA

検証
障害の大きさ＝エラー×エラーの滞在時間
設計は要求分析へのフィードバック
実装が設計へのフィードバック

サービスとしてのプログラム
視点の変換

このドキュメントは
このソースコードは

このドキュメントは顧客をどう幸せにする?
このクラスは顧客クラスをどう幸せにする?
このメソッドは顧客メソッドをどう幸せにする?

顧客の価値を上げることにコミット

エンジニアとしての質

カイゼンのループ
PDCA

カイゼンとは、
「良いことを行うこと」→「良いと思うことを試し続けること」
フィードバックがキモ
角度が重要

行動を変化させること

"XP is about social change. Social change is changing yourself." †
XP とは人と人とのつながりを変えることである。人と人とのつながりを変える
とは、あなた自身が変わること。

Kent Beck の「Extreme Programming Explained: Embrace Change, 2nd
Edition」(邦訳: XPエクストリーム・プログラミング入門―変化を受け入れる
第二版) より。

3K
子供にエンジニアを勧められる
ともに成長していることが信じられる

Developer 2.0
企業の考える5年後のスキルと個人の考える5年後のスキル

ビジョンと現状
「カメがウサギに勝てたのは、ウサギを目指したからではなく、ゴールを目指し
たからである」

技術習得のレベル
段階 1 : 無知の段階 ― その技術について聞いたこともない
段階 2 : 気がかりな段階 ― その技術について文献を読んだことがある
段階 3 : 見習いの段階 ― その技術について 3 日間のセミナーに通った
段階 4 : 実践しようとする段階 ― その技術を実際のプロジェクトに適用しよ

うとしている
段階 5 : 職人の段階 — その技術を仕事の上で自然に自動的に使っている
段階 6 : 名人の段階 — その技術を完全に消化していて、いつルールを破るべきかを知っている
段階 7 : エキスパート — 専門書を著作し、講演し、その技術を拡張する方法を探究する
↑
「何故開発がうまく出来ないか。それは識らないことが在るから」
何を作れば良いのか識らない
どうやって作れば良いのか識らない
何が使えるのか識らない
情報が何処にあるのか識らない
↑
「解決策が分らないのではない。問題が分っていないのだ」
チェスタートン

三大要素
ビジョンと現状と問題
→ ToDo (プラクティス)

ToBe - AsIs → ToDo

Why → What → How と考える

Clean Code

Aha! 体験
やってみないと気付きは少ない

実践からのフィードバックがカイゼンに

ドッグイヤー
周りの人が前に進んでいるのにじっとしているのなら、それは下がっているのと同じ

「芸は砂の山」

答えをもって聞かない

QoEL

素直さ重要

実習

NUnit
TestDriven.NET

TDDによる実習

```
class 仕様記述
{
    void Verification
    {
        ○○であるべき;
    }
}
```

static 仕様Verificator
dynamic 仕様Verificator

リファクタリング

モデリング コンテスト
グループ モデリング

複雑さのグラフ

良書
達人プログラマー システム開発の職人から名匠への道 アンドリュー・ハント、
デビッド・トーマス
リファクタリング―プログラムの体質改善テクニック マーチン・ファウラー
アジャイルソフトウェア開発の奥義 ロバート・C・マーチン
UMLモデリングの本質 児玉公信
オブジェクト指向における再利用のためのデザインパターン
エリック・ガンマ、ラルフ・ジョンソン、リチャード・ヘルム、ジョン・ブリシ
ディース

- smell code
- インタフェイスについて class Figure -------- start end -------- SetStart(position) SetEnd(position)
- class Figure -------- start end --------
- SetStart(position) SetEnd(position) Move MoveTo
- 外からは: どんなインターフェイスを持って欲しいか 実装 (内部構造) に口出ししない
- ソースコードが重複するのは偶然じゃない
  - 理由がある
  - 同じ責務を複数の部分が持っている可能性を疑うべき
- コメント
  - ソースコードで表現できないことを書く
  - Why を書く
  - 「何故そのコードを選んだか」はソースコードでは表現できない
  - 図を描きたい
  - ドキュメントとしてのコメント
    - 使う側から参照できる
    - Contract や例外など
  - https://twitter.com/epitwit/status/723306702678843392
    - それが"なに(what)" かはクラス名/関数名で、"どうやって(how)"はコードそのもので表現する。残るは"なぜ(why)"そうしたかだが、それをコメントに記すのであるよ。
- 何故適切な名前をつけるべきなのか?
- [Programming] 「同様のコード」が複数の場所に出てきたり、異なった名前空間、クラス、メソッド内などに「同じ名前」が出てくるのは偶然じゃない。そこには共通した関心事がある。一つの関心事が複数にコードに渡っていることを意味している。

一時変数で受ける

```
double Average(double a, double b, double c)
{
    var sum = a + b + c;
    return sum / 3.0;
}
// Average は「a と b と c を足し、これを sum とする。この sum を 3 で割ったもの。」

double Average(double a, double b, double c)
{
    return (a + b + c) / 3.0;
}
```

```
// Average は「a と b と c を足し、3 で割ったもの。」

double Sum(double a, double b, double c)
{
    return sum = a + b + c;
}
// Sum は「a と b と c を足したもの。」
double Average(double a, double b, double c)
{
    return sum(a, b, c) / 3.0;
}
// Average は「a と b と c の Sum を 3 で割ったもの。」
```

ベクトルで考える

中点 = (*a* + *b*) / 2;

cx = ax + bx / 2;
cy = ay + by / 2;

ではなく。

この話は a や b が二次元ベクトルか三次元ベクトルかに依存しない (二つの事象は独立)
プログラムでこの二つの事象が独立 (依存関係がない) = 疎結合なのは、「プログラムのテクニックとして」ではない。
本来独立した事象なので、プログラムで依存関係が生まれる理由がない (本来密結合ではない、本質的に疎結合)。
疎結合なものはプログラムでも疎結合になるのは自然。

「人の考え方から乖離したプログラムは判りにくい」

論理的に考える力
国語力重要

最低限 施工事例でのタグ対応
タグ
　種類

フォルダー or タグ

クエリー

　複数タグでの検索
　「を全部含む』のいずれか」

PC版、iPad版、...

業務分析

→ 要件

種類

顧客ごと
フォルダー分け

　タグを付けて検索したい

検索条件

「この画面で全部」

　タグとは?

何故仕様が複雑化するのか
シンプルにならない理由

　業務分析 →
　タグの定義

キラー クエスチョン
　「どう使うか?」
　「それは何か?」

1から10までを加えた数を返す:
```
return 55;
return 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
return Enumerable.Range(1, 10).Sum();
```
実行スピードを上げる
保守性が下がる
サブルーチンの実行速度を気にしたら、サブルーチンの内部に依存

Concernはアプリケーションを綺麗に保てる良い手段。だが、そこでゴールではなく、更にリファクタリングを重ねること。

[Programming] 人は必ず間違うので、「如何に間違ってる時間を短くできるか」が重要。一秒でも早く。例えば、文法ミス程度なら、実行せずとも明示的にコンパイルしなくとも、タイプした途端にエラーがその箇所に示されると良い。そう Visual Studio みたいにね。

サブルーチンは、「似たような処理をまとめる」為じゃなく、「名前を付けて抽象化する」為にあるんだよね。ただ「似たような処理をまとめた」らスパゲッティ化する。

抽象化することによって、なるべく常にシンプルな問題だけを解くようにプログラミングする。

抽象化というのは:「どういう手順でそれをやるのか」ばかり考えると、プログラムは複雑化する。なので、「つまり、結局、ここでは何をやればいいのか」と考える癖をつけて、なるべくそっちを書くようにする。

例えば「平均を求める」問題を解くときに、「えっと先ずループを回して…」なんて考えない。「平均は、合計を個数で割ったもの」、次に「合計」とは…、「数」とは…、と解いていく。ここでサブルーチンが出てくる。

そうしてサブルーチンを使って、処理を抽象化していくと、「ネストした」ループなどと云うものは自然に作らなくなる。初心者には、そういうことを教えた方が良い。「似たような処理をまとめて再利用」は難し過ぎ。

大切なのは、「何故それを人は複雑に感じるのか」と云うこと。「プログラミングは複雑で難しい」のだから、人が頭の中で考えられる限界を超えた複雑なコードは書いちゃだめ。

人は一度に複雑なことを細部まで完璧に考えられない。それがサブルーチンがある理由。

[Programming] 「本質的に複雑な処理を書いたメソッドが複雑になるのは仕方がない」について。「本質的に複雑」というのが「複雑さが不可避」の意味であるならトートロジーで無意味。また、本質的だと決めつけて諦める前に「避けられない複雑性」が存在しうるかどうか考察すべき。

「無理に短いメソッド群に分解」が何を意味してるかによると思う。「本質的に長い記述を要する」んなら、そも不適切な分割しかできない筈。「分割により依存関係が増し、より複雑になるだけ」なら、もちろんそ

んな分割はすべきじゃない。 > twitter.com/tomooda/status...

メソッド分割は、「どんな語彙で記述したいか」の問題だと思う。私の場合「あるメソッドから小さなメソッドを切り出して、新たなメソッドのコールに変える」のは主に「元のメソッドをその語彙で記述したいから」。
> https://twitter.com/Fujiwo/status/362426886955089921

例えば、if (name.Length > 0) ... じゃなくて if (IsValid(name)) ... と IsValid を作ってまで書くのは「このロジックをその語彙で書きたい」時。つまり「その抽象度で書きたい」時。 > bit.ly/19wi0Zj

「では、どの抽象度で書きたいのか?」といえば、それは「人がそれを記述するときにもっとも自然な抽象度」。例えば、「最寄りの空港に行く」というメソッド内部は、それを自然言語で「もっとも簡潔に」記述するときの粒度で書きたい。 > twitter.com/Fujiwo/status/...

何故かと云うと、「それが我々には解りやすいから」。
> http://twitter.com/Fujiwo/status/362492899126022144

Fujiwo:それに、「技量が一番下の人に合わせる」で生産性が上がったりしない。技量が高い人たちが多くのコードを生産しているので、そこの足を引っ張っても全体最適化にならない。
[http://twitter.com/Fujiwo/status/364204822263644160]
Fujiwo:プログラミングで難しいのは、複雑になってしまうところ。そしてそれこそが保守性 (含むコードの可読性) の一番の敵。初心者には特に。LINQ等の「複雑さを回避する仕組み」を除外したって解決しない。前にも書いたけどLINQ使うだけなら学習コストも大したことないし、問題ない。
[http://twitter.com/Fujiwo/status/364204207001174018]
Fujiwo:保守性のためなら、初心者には「if とか switch をなるべく使うな」とか「ネストは一段階までにしろ」とか言う方が良い。 > twitter.com/Fujiwo/status/... #hokunet
[http://twitter.com/Fujiwo/status/364199850474086400]
Fujiwo:「保守性の為に if (x == 0) じゃなくて if (0 == x) と書け」とか「保守性の為に if の中は単文でも {} 付けろ」とか「くだらないこと」言ってそう > 「LINQ禁止とか言う人」。 > twitter.com/Fujiwo/status/... #hokunet
[http://twitter.com/Fujiwo/status/364199224985923584]
Fujiwo:だいたいプログラミングで初心者が保守性を下げるのは「そんなところ」じゃない。LINQ禁止で何かが僅かでも解決すると思ってる人が現場に口を出さない方が吉。 > twitter.com/Fujiwo/status/... #hokunet [http://twitter.com/Fujiwo/status/364196378483752962]
Fujiwo:いまだに「LINQ禁止」なんてあるのか。「普通の人」は LINQ 使った方が生産性・保守性が上がるんだから、「ごくごく一部の分からない人」に使い方を教えて皆で使った方が得なのに。 > twitter.com/bouzuya/status... #hokunet
[http://twitter.com/Fujiwo/status/364195391253655553]

[Programming] バグ報告するときは、再現条件を言ってあげるのが親切。このときの再現条件は必要十分条件にするのが基本。つまり「○○してたら落ちた」より『この動作環境でこの手順でやったとき』だけ必ず落ちます」が親切。

[Programming] 良いプログラミング言語は:「プログラムがこういう風に書かれるべき」を邪魔しない。プログラマーの「こう書きたい」に「変な記述を足さない」。更に、プログラマーに「あ。そうか。俺こう書きたかったんだ」と気付かせる。

二重ループ
「複数の名前から指定文字列を見つける」は一重ループ。

例えば、そのプログラムを図で「もっとも判りやすく」表現するとしたら、型名をいちいち書いたやつをメインの図にするか? ということ。そして、何故そうしないのか? ということだと思う。書きやすさの為に var を使ってる訳じゃない。

[Programming] なんでそんなに型名を書きたがるのかよく分からない。ロジック読むときに型名なんか要らなくない? そっちをデフォルトのソースコードのビューにしといて、型名チェックしたい場合はツールでチェックすれば良いのでは。

[Programming] 「名前」は必要にして十分であるのが良い。先ず、表そうとする概念がシンプルであること。そして、名前はその概念以外を表さず、且つ、概念を余さず表すように付ける。これは高凝集を意味することになるので、良い名前付けはプログラムを高凝集にする。
名前付けによって、「それであるものとそれでないものを分ける』その境界線をくっきりさせる」ということ。
一旦何かをそう呼ぶことにしたら、別のものはそう呼べない。名前は使いまわせない。

[Programming] 美しいソースコードの指標 1: 継続して作業しやすいこと。ソースコードは最初の1行を書いたとき以降、その寿命を終える迄インクリメンタルに書かれるので、将来に渡って書き加えるのが容易であることが重要。
[Programming] 美しいソースコードの指標 2: シンプルであること。そのソースコードを読む人が扱える複雑さの限界を超えないこと。
[Programming] 美しいソースコードの指標 3: 検証が容易であること。ソースコードが実現すべきことを実現しているかどうか検証することが容易であること。

[Memo][Programming] プログラムは、最初の一行を書いてからのコーディングはずっと変更なので、拡張性のない (変更が容易でない) プログラムを書くのは常に非効率。

【プログラミング研修】 各部分 (クラスやメソッド等) の仕様を well-defined にする (それが何であるかを明確にし、輪郭をくっきりさせる=それであるものとそれでないものの境界を明らかにする) ことが大切だと度々強調している。

【プログラミング研修】 現状がどれだけ仕様から外れているかを調べるのが検証 (verification)。仕様-現状=バグ。仕様が曖昧だとちゃんと検証できないし、どれがバグかが定まらない。

【プログラミング研修】逆に (謎)、「(その時点の) 検証 (テスト) を通れば、(その時点での) 仕様通りである」と定義すれば、テストケースを徐々に追加していってそれを一つずつ通していくことで、インクリメンタルに検証可能な仕様記述をすることが可能になる。

【プログラミング研修】各部品の責務の範囲をくっきりさせることが重要。話は変わるが、PowerPoint の資料作るときも、その資料全体、各スライド、各図の責務の範囲をくっきりさせると良い、などと言ってみた。

【プログラミング研修】 各部品の範囲を「過不足なく」表すような名前を付けることが重要。複雑すぎる名前になったときは、範囲の切り取り方を疑う。逆にいうと、良い名前を付けることで、適切に範囲を切り取る。言葉の選び方重要。プログラマーは日頃から言葉を大切に扱うと良い。などと言った。

【プログラミング研修】 使う側から見て、それが何であり、どういう責務を持っているか、を名前で表す。それがどうやって実現されているか (=実装) は表さないようにする。境界=インタフェイスを決めるときは外から見る。
使う側から見ると、インタフェイスの名前=それの名前になる。

ツールについて
　技術はツールとともに語られるべきか (C# の良し悪しは Visual Studio とともに語られる)

https://twitter.com/Fujiwo/status/430222359467786240
『マクロを組んで仕事をしていたら、先輩に「ズルしてる」と怒られた』 http://bit.ly/1eLCcnQ 系
https://twitter.com/Fujiwo/status/430220980829437953
[Programming] 「静的型付けもオブジェクト指向も関数型もコンパイラーもIDEも、紙上デバッグ&ハンド・アセンブル以外は全部甘え」と云うのを思い付いた。
紙テープもテレタイプも甘え。8個のトグルスイッチをセットしてプッシュボタン。

"Real Programmers Don't Use Pascal" (本物のプログラマはPascalを使わない)
https://ja.wikipedia.org/wiki/%E6%9C%AC%E7%89%A9%E3%81%AE%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9E%E3%81%AFPascal%E3%82%92%E4%BD%BF%E3%82%8F%E3%81%AA%E3%81%84
「本物のプログラマは、(FORTRANで) 戸惑うことなく5ページにもわたるDOループを書かなくてはならない」
→「本物のプログラマは、コードを書かない」

良いコ-ドへの道―普通のプログラマのためのステップアップガイド:連載|gihyo.jp … 技術評論社 http://gihyo.jp/dev/serial/01/code

[Link] コーディング技術にこだわり過ぎるとITエンジニアの地位は向上しない http://bit.ly/1bv4wdf とその反論: SIerって終わってんな http://bit.ly/1bv4y4X

[Programming] モジュール (メソッド、オブジェクト) 名には (そのモジュールのクライアントから見た) What を書き、実装には How を書き、コメントに Why を書く。How を書くときはこっちがクライアントで他のモジュールの What を書く。

[Link] 「何がわからないか言ってみなさい』何がわからないかがわかりません」問答
http://bit.ly/1qZOHCh 「分からないことと分かることを分けられる」ことが「分かる」ことの基本。なので、「何がわからないか言えるようにする」ことが「教える」ことの基本。
「何が分からないか言えるようになりましょう」といつも教えている。「何が分からないか分からない」状態は自己解決が難しい状態。そこから出ると自力で先に進める。「問題が認識できないと解けない』問題が認識できれば半分解けたようなもの」

Reading: 複雑さについて思うこと。 - 結城浩の連ツイ http://rentwi.textfile.org/?598379314262900736s

[Link] 関数の適切な長さとは? マーチン・ファウラー氏は、長さより意図と実装の分離、そしてよい関数名が重要だと指摘 | Publickey
http://www.publickey1.jp/blog/17/post_262.html

[Programming] 名前付け

概念の範囲

インタフェース

責務

ユニーク

・基礎知識編
　・原理

・
状態
「何の」

・モノ - コト - モノ

・コメントなしソースコードでは What How は表現できるが Why は表現できない コメントが必要
・複数のパラダイムに親しもう
・プログラミング言語は思考のフレームワーク

言語は思考のフレームワーク
　言語によって
　　語彙が異なる
　　よく使う言い回しが違う
　　論理構造が違う
言葉による思考が左右される

・その文脈での語彙を用いる

・美しいソースコードとは
　・図で説明するならどう説明すると判りやすいか。

・設計の考え方
　・モデルとは
・分かりやすさとは
　　・理解 (しているモデル) と乖離していると分かりにくい
・意図とソースコードの乖離について
・変数で受ける理由
・文章を書くように

分析について
　物事をどう捉えるか
　　2つのものと観るか1つのものと観るか
　　　2つのものと観ると、2つのプログラムになる
　　　　(...何か例...)
　　どう捉えるかによって名前が変わる

・バグなのかバグじゃないのか
・契約プログラミング
・バグと検証 (Validation とvarification)

・インタフェース
・インタフェースが名前になる
・委譲する
・独立性

・問題の切り分け
　・クラスやサブルーチンが存在する理由
　・それに名前が付けば分けられる

・ガード条件

・正常処理と例外処理

・解 → 問題 = ToBe - AsIs
「以前からプログラミングに興味があって」っていう新人ITエンジニア。プログラミング経験を聞くと全くない、とのこと。ずっとやってこなかったんなら、多分ずっと興味がなかったんじゃないのかな。

「知っている」と「分かっている」の差は大きい。私は自分が「分かっている」かどうかを検証するのに、テストケースとして「分からない人に分かるように説明できるかどうか」や「知識として実際に利用できるか」を用いることにしている。
・分かるということ。

考えのまとめ方
- 自然言語で書く
- 図で描く

・[Memo] プログラミングは、人 (自分も含む) とのコミュニケーション。機械とのコミュニケーションと考えると雑になる。

- [Programming] 「この地図を画像としてダウンロードして印刷しといてね」みたいなのも抽象化。戻り値を一旦画像として変数に受けて、印刷メソッドに渡してる。変数に入れるか入れないか、どんな変数に入れたかで文脈が変化する。
  - 実装に落ちる前の段階で抽象化されてる。実装が抽象化してるんじゃない。
  - 欲を言えば、プログラミング言語は「こう書きたい」を助けるものであって欲しい。「この言語の場合はこう書くしかないよね」みたいなのは必要悪に感じる。

- 或る塊を簡潔なコメントで説明できる場合は、そこをその名前のサブルーチンで切り出せる。
- [Programming] 新人プログラマーに基本を教えてるところ。「使う側視点で何を作るか決め (インタフェイスとテストケースの定義)、然る後に作る (インタフェイスを実装してテストを通す)。」
- [Programming] 新人プログラマーに基本を教えてるところ。「メソッドの内部は人がシンプルに考えるときの粒度で設計し、その儘の粒度を保って実装する」
- [Programming] C++研修中。C++ よりもプログラミングそのものの方がずっと教えるのが難しいことを再認識。型の概念やアルゴリズムの書き方、その他。

- [Programming]　(クラスやメソッド、メソッド内の処理などを書く際に) 難しいポイントは、「抽象化」して考えることと、「曖昧性なく言語化すること」のようだ。
- [Programming] 設計は、プログラムを記述する語彙を作る作業でもあると思う。クラスやメソッド、変数を設計するときに、関心事を切り出して (或いは分割して) 名前を付ける。それは、プログラムを記述する語彙を作ってることになる。
- [Programming] 設計によって「自分が書きたいように書けるようにする」。
- [Programming] サブルーチン (関数、メソッド) を作るのは「共通の処理をまとめて再利用するため」というより、「処理のまとまりに名前を付けて抽象化し、プログラムを記述する語彙とするため」。クラスや変数など他のシンボルも同様。
  名前付けによって、「どういう語彙で記述するのか」を決められる。
- C++研修中。「気付き」を奪わないように注意。「分かる」のは本人にしかできない。正解を告げれば分かれる訳ではない。正解を覚えるのと分かるのは別のこと。
- プログラミング研修中。「判ろうとする人」は伸びるが、「正解を聞いて正解を覚える」タイプの学習をしてきた人は伸びが遅いようだ。一つの正解を聞いても他に使えない。プログラミングは応用問題ばかりなのでこれはまずい。さて、どうしようかな。
- 「正解を覚える」タイプの人は、「どこかにある正解」を教えてもらうまで正解が分からない。それまでは「教わってないから知らない」。プログラマーの主な仕事はまだ書かれてないコードを作りだすこと。逆に既にあるコードは書かない。
- プログラミングは「既に解かれた問題は解かなくて良い」という位のものなので。或る問題の解き方をまんま覚えたって仕方ない。
- IT技術者に必要な学習は、正解を知ること、テキストに書いてある正解を覚えること、ではない。問題を見つけて、その問題を解くこと。
- プログラミングで特に大切なのは、「やりたい事を曖昧性なく言葉にする能力」だと思う。
- プログラミング研修中。『あなたのメソッドの書き方は良くない』と言われて、自分の問題を『メソッドの書き方が良くない』だと捉えてしまい、『自分のメソッドの書き方を良くしよう』を解決法だと思うと具体策がないので何も解決しない。問題の捉え方を間違えると解決できない」と言ってみた。
- プログラミング研修中。「理解するためには、どういう質問をすれば良いか」が結構重要な気がする。「質問する練習」をしてみようかな。「理解することがどういうことかを理解するための訓練」として効果的かも知れない。
- プログラミング研修中。「理解したかどうかを検証する方法」を持つことが重要だと言ってみた。例えば、私の場合は「他の人に説明できるか』実務に役立てられるか」等で検証する。これらのテストケースを通ったら、「理解できた」ことにする、というもの。
- プログラミング研修中。例えば、教わったことをその儘言えたとしても「理解してる」とは限らない。説明を記憶することと理解することは別。オウムがオウム返しできても理解できてる訳じゃない。オウム返しで理解したかどうかは検証できない。
- (プログラミング研修で話してることは、 http://www.slideshare.net/Fujiwo/20140213-developers-summitlt1 … にも少し書いてます)
- プログラミング研修中。仕様と実装の視点切り替えが難しいようだ。例えば、メソッドの外部仕様は呼ぶ側の要求に基づいてデザインされたインタフェイスに基づき、メソッドの内部実装はその外部仕様に基づく (クラス等のもっと大きい粒度でも同様) のだが、そこを分けて考えられないようだ。

- 新人プログラマーの教育中。理解型の学習ではなく記憶型の学習をしてきた人の「分かる」は「記憶にある」、「分からない」は、「見聞きしたことがない」なので、厄介。これだと、すでにある正解しか答えられない。正解を作りだす作業には向かない。理解型学習について説明。
- 新人プログラマーの教育中。理解型の学習ではなく記憶型の学習をしてきた人の「分かる」は「記憶にある」、「分からない」は、「見聞きしたことがない」なので、厄介。これだと、すでにある正解しか答えられない。正解を作りだす作業には向かない。理解型学習について説明。
- 要は「メソッドの中で何をするか (What) と、どうやってするか (How)」を分離して考えるということ。What に先ずフォーカスすることが重要。
- 1.メソッドの仕様を書く 2.仕様に基づきテストを書く 3.メソッドの宣言を書く 4.メソッドを実装 5.テストを通す、という順番で「視点を変える練習」してもらってるところ。
- なるべく What にフォーカスしてられるプログラミング言語/環境が、良い言語/環境。
- 「メソッドの外部仕様はメソッドを使う人の立場で、内部実装はコードを読む人の立場で」
- [プログラミング研修]「日付って言葉を知らない子供に日付の意味を説明してみる』size と volume の違いを説明してみる」みたいな演習。これは「名前付け」の演習。クラス名やメソッド名を丁寧に付ける為。
- プログラミング研修。「型やメソッド、変数等の名前を英語で付けるときに、例えば、"大きさ"の意味を辞書で調べて"size"と書いてあったから"size"と付ける、みたいな安易なのは NG。ニュアンスまで考えて意味が近いものを適切に選ぶ。というか、英語を勉強するように。」
- ところで、プログラミングを始めてから英語を復習するのはまだ楽だが、国語力がない人がプログラミングをマスターするのは本当に大変だなー、というのが最近の実感。厳密に自分の語彙を作り、それで何をやるかを記述する、という作業なので。
- 「プログラミング=設計+実装+テスト」で、且つこの中で一番省略ができないのが設計。「設計にはプログラミング経験が必要か否か」という問い自体奇妙。「ユーザー視点の外部仕様の一部にはプログラミング経験不要かも」は同意。> http://kuranuki.sonicgarden.jp/2013/01/post-109.html
- プログラミング=設計+実装+テスト。
  ※ この場合のテストは開発者テスト

  臨機応変に、

  設計 → 実装 → テスト → ...
  実装 → 実装 → 設計 → ...
  テスト → 実装 → 設計 → ...

  のように短く何度も繰り返す。
  設計と実装とテストでは視点が変化するため、プログラムを多面的に捉えられる。

- 7つの習慣
- サーバー視点からクライアント視点への転換。基本全てのメソッドやクラスはこうやって作るのが良いと思う。テストファースト等が良い例。> https://twitter.com/tagomoris/status/496878628202741762 何かAPIを考えるときは、そのAPIを使ったコードをどういうふうに書きたいか、の例をまず適当に書くと良いことが多い

- 設計と制約 何が良い設計かは制約に依る
- [Programming]「作成期間中は、なるべく何時もプログラムが起動可能な状態を保ちつつインクリメンタルに機能を追加していく。ある程度完成に近づく迄起動出来ないのは宜しくない」
- 責務を分割 (委譲)
  - 責務の周りの境界線を「くっきり」させる
  - 「外部から『シンプルに」内と外が区別できるように
  - 責務を説明したときに、内部でやってることや「○○や△△なんかをやってる」という説明になるときは境界線がくっきりしてない
- 図で描くと、複数の四角を線で結んだもの
- 「プログラミングのやり方」
- 或る関数の中の内部ロジックを書くのに、他の関数の内部ロジックを見なきゃ書けない、という時点でプログラミングの方法がおかしい。
- [Link] Goの変数名が短い理由 - Qiita bit.ly/1qDUgKu「読む人にとって、そこの文脈における一番判りやすい (目にしてから理解する迄の時間が最短な) 名前」が良いと思う。書くのに長いかどうかは大した問題じゃない。IDEが何とかする。
- kohta ishikawa @_kohta
  - プログラミング得意じゃないから偉そうなこと言えないんだけど、コードを読む(意図を理解する)作業は逆問題で解が一意に定まらないので、コメントは解釈の分岐を可能な限り減らす情報を書くとよいと思っている。実践できているかは別問題。
  - twitter.com/_kohta/status/616612103353339905
- モノ-コト-モノ分析
- メソッド名は動詞、クラス名、変数名は名詞
- [Programming] ソースコードの悪い匂いの一つに "Noisy" というのを加えたい。
- 文脈は型でなく、オブジェクトの名前で記述される
- Retweeted Jxck (@Jxck_):

  「守破離」でいう「守」でコケて諦めたのを「離」とは言わない。「逃」だ。
  ある批判的な言説を見たとき、「逃」か「離」かを見極められないと、振り回される。FUD とか典型。
  そして、それを見極められない人が正しく知識を身につけるのは難しい。が、見極めるには正しい知識がいる。難しい。

  リスペクトが必要
- return a + b; と var sum = a + b; return sum; は異なるロジック。
- 日本語の単語にしろ、英単語にしろ、漢字にしろ、言葉はニュアンスとして「それが意味する範囲」として捉える。それが国語力に繋がると思う。

きれいな設計を身に付けるためのSandi Metzルール
http://gihyo.jp/dev/clip/01/tech_information/vol75/0003

JavaScriptの関数で何ができるのか、もう一度考える
https://sbfl.net/blog/2016/12/26/javascript-function-revisited/

メソッドからの責務の切り出し方

責務ごとに明確に分割
それぞれに名前を付ける
→ 名前で表現
→ コメントで表現
→ 外に追い出す

それぞれに名前を付ける

責務のスコープを考える
そのメソッドだけの責務
→ 変数
→ コメントで責務に分割
そのクラスの責務
→ クラス内のメソッド、プロパティ
そのクラスの外の責務
他のクラス
汎用的
→ 汎用クラスへ
スコープは小さい方が良い

そのプログラムは、「どの問題」に対する解なのか?
問題が一般的 → プログラムも一般解 → 一般的な層に書かれるべき

[Programming] 概念の領域に命名して、はじめてそれを概念として認識できるようになる。概念の命名によって語彙を作ることで、その語彙でその概念に関して記述ができるようになる。

何を作るか決めないと、できたかどうか分からない。

ソフトウェア原則
http://objectclub.jp/technicaldoc/object-orientation/principle/

オブジェクト指向の法則集
http://objectclub.jp/community/memorial/homepage3.nifty.com/masarl/article/oo-principles.html

「インターフェースに対してプログラミングする」という設計原則

DRYと不当な抽象化によるコストについて | POSTD
http://postd.cc/on-dry-and-the-cost-of-wrongful-abstractions/
https://twitter.com/arton/status/793979503789965312
https://twitter.com/arton/status/793979926810460160
https://twitter.com/arton/status/793982077335048192

良いコードの書き方 - Qiita
https://qiita.com/alt_yamamoto/items/25eda376e6b947208996

サービス提供側のモジュールがクライアント側のモジュールの仕様に依存しないように

```
例.
void client(const char* userName)
{
    // ...省略...
    int userNameLength = GetTextLength(userName); // OK
    int userNameLength = GetUserNameLength(userName); // NG
    // ...省略...
}

int GetTextLength(const char* text) // OK
{
    // ...省略...
}

int GetUserNameLength(const char* text) // NG
{
    // ...省略...
}
```

マジックナンバーについて
　理由
　どこに定義を置くか

■ 新人研修「コードを書くときの心得」
Q. 綺麗なコードってどういうの?
A. 変更や追加が楽で、テストが楽なコード。
■ 新人研修「コードを書くときの心得」
Q. コードが綺麗でなくても、動けばいいんじゃないの?
A. 動くのは最低限のことで当たり前。
追加や変更を繰り返しても「動き続ける」ことが大切。

コードの綺麗さは、ソフトウェアの寿命に貢献。
■ 新人研修「コードを書くときの心得」
Q. コードって手順を書くんですよね?
A. まあそうだけど、コードは、意図を語るべし。
プログラムには、なるべく HOW (どうやってやるか) でなく WHAT (何をやるか) を書く。
ちなみに WHY (何故このロジックを選んだかなど) はコメントで。
■ 新人研修「コードを書くときの心得」
Q. メソッド名は、そこで何やってるかを付ければ良いんですよね。
A. 呼ぶ側の視点で命名 (インターフェイスは外から見る)。
×: LoadXmlAndCreateItemList() // メソッドの中で何をやるかが名前になっている
○: GetItemList() // サービスを受ける側の視点
■ 新人研修「コードを書くときの心得」
Q. 省略せずに適切なメソッド名を付けろっていうけど、そうすると名前が長くなっちゃう。
A. 各メソッドを単機能にしてかつ汎用的にすると、短く適切な名前に。
×: GetOldStaffSalaryAverageAndPrint();
○: Print(Staffs.Where(staff => staff.IsOld).Select(staff => staff.Salary).Average());
■ 新人研修「コードを書くときの心得」
Q. メソッドって重複した処理をまとめるためにあるんですよね。
A. メソッドは:
・ひとかたまりの処理を分離して、独立させる。
・それに名前を付けて、抽象化する。
・名前を付けることで、認識できるようにする (ジョシュア ツリーの悟り → Name and Conquer)。
・汎用化することで、再利用できるようにする。
■ 最後におまけ: var について
Q. 型推論の var を使うとコードが分かりにくくなるって聞いたんですが、var を使うコツってありますか?
A. var を使うと分かりにくくなるようなコードを書かない。

**「ソフトウェアの品質とは、優れたプログラムが備えるべき七つの属性の集合である」

七つの属性を平均的なプロジェクトでの優先順にあげると以下の通り。

-信頼性 (reliability)
-使用性 (usability)
-理解容易性 (understandability)
-変更容易性 (modifiability)
-効率 (efficiency)
-検証性 (testability)

-移植性 (portability)

最も重要な原則

↑

高凝集 (high cohesion) 且つ 疎結合 (low coupling)

↑

高凝集 (high cohesion)

クラスは一つの明確な目的と名前を持ち、不整合のない首尾一貫した中身を持っているべき。

↑

疎結合 (low coupling)

クラスは単純なインタフェイスを持ちは、関係のないクラスとは出来るだけ話さないようにすべき。

↑

Robert C. Martin の Principles of OOD

Object Mentor - pood

http://www.objectmentor.com/courses/pood/

↑

OCP (Open-Close Principle)

開放と閉鎖の法則。

ソフトウェア モジュールは、変更に対して閉じており、拡張に対して開いて いるべき。

↑

LSP (The Liskov Substitution Principle)

リスコフの置換原則。

スーパークラス (継承元のクラス) をサブクラス (継承したクラス) で置き換えることが可能であるべき。

スーパークラスへの参照 (やポインタ) を使うメソッド (関数) は、それのサブクラス のオブジェクトを、それとは知らずに使えなければならない。

↑

DIP (The Dependency Inversion Principle)

依存関係逆転の原則。

上位モジュールが下位モジュールに依存すべきではない。上位モジュールも下位モジュールも抽象概念に依存すべき。

抽象が詳細に依存すべきではない。詳細が抽象に依存すべきである。

↑

ISP (The Interface Segregation Principle)

インタフェース分離の原則。

クライアントは自分が使わないメソッドに依存することを強制されない。

↑

REP (The Reuse/Release Equivalency Principle)

再利用・リリースの粒度の原則。

再利用の粒度はリリースの粒度であるべき。

↑

CCP (The Common Closure Principle)

パッケージ内のクラス群は或る修正に対して閉じている方が良い。従って、将来の変更に対して同じような修正が予想されるクラスは同一パッケージに入れた方が良い。

↑

CRP (The Common Reuse Principle)

パッケージ再利用の原則。

パッケージ内のクラスは一緒に再利用されるべき。

↑

ADP (The Acyclic Dependencies Principle)

循環依存禁止の原則。

パッケージ間の依存関係は依存してはならない。

↑

SDP (The Stable Dependencies Principle)

安定依存の原則。

パッケージの依存関係は、依存元が依存先より安定している方が良い。

↑

SAP (The Stable Abstraction Principle)

安定抽象の原則。

抽象的な方がより安定しているべき。

↑

SRP (Single Responsibility Principle)

一つのクラスは一つの責務を持つべき。

クラスに変更が起こる理由は一つであるべき。

良い抽象には良い名前がつく。

↑

Karl J. Lieberherr

↑

The Law of Demeter

デメテルの法則。

オブジェクト中の全てのメソッド (関数) は、以下のいずれかの種類のオブジェクトのメソッドのみを呼び出すべきである。

- それ自身
- メソッドに渡されたパラメータ (引数)
- それが生成したオブジェクト
- 直接保持しているコンポーネント オブジェクト

Introducing Demeter and its Laws by Brad Appleton

http://www.cmcrossroads.com/bradapp/docs/demeter-intro.html

↑

Bertrand Meyer

↑

IOP (Inside-Out Principle)

中から外へ向って設計せよ。

モデルを先に設計し、ユーザーインタフェイスは後で設計せよ。

↑

OCP (Open-Close Principle)

↑

その他

↑

Edelman's Law

他人と話すな。(Don't talk to strangers.)

クラスは関係のないクラスには話しかけないようにせよ。

↑

Design by Contract

契約による設計。

「インターフェースに対してプログラミングする」
http://promamo.com/?p=2241

初級プログラマくらいの人向けに「プログラミングは論理的思考が必要っていうけど、なんぞそれ」みたいな、説明に使うための文章を書いてみています。| やっとむでぽん
http://d.hatena.ne.jp/yach/20161006#p1