

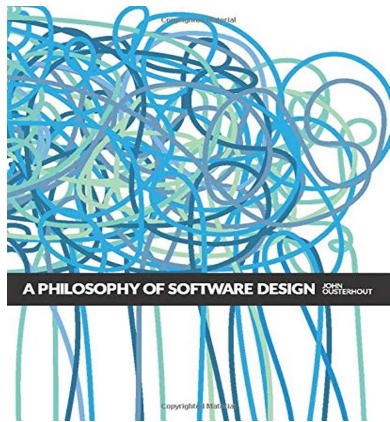
勉強会 - 2022/06

本の紹介 : A PHILOSOPHY OF SOFTWARE DESIGN

me

背景

- (積んでいた) 本を消化したい
 - パラパラ読むより, スライド化したほうが記憶に残るかも
 - そもそも洋書なので, パラパラ読めるものではなかった (個人差)
 - 内容としても, 資料化しておいて社内に置いておいて良いだろう
 - そこそこ有名な本らしい ミーハーなので買いました
- 本について
 - タイトル : “A PHILOSOPHY OF SOFTWARE DESIGN”
 - John K. Ousterhout 著
 - スタンフォード大学の教授
 - 初版2018年
 - 今は2nd Edition(Kindleあり)
 - 198ページ, 全22章
 - 比較的平易な英語で書かれている (?)



本の概略

- ソフトウェア開発におけるComplexity(複雑さ)への向き合い方について
 - システムが巨大化するにつれ, Complexityは必ず生じる
- Complexityを認識する能力==ソフトウェア設計力
 - 新システムを設計するときどのように Complexityを回避するか？
 - 既存システムを見たときどのように Complexityを認識するか？
 - シンプルな設計にするためのポイントが分かれば対処できる
- Complexityの種類と, それぞれの対処法を紹介
 - 複雑さを抑えるためのテクニックを記述
 - コードレビュー時に引用すると良い (とのこと)

本の概略

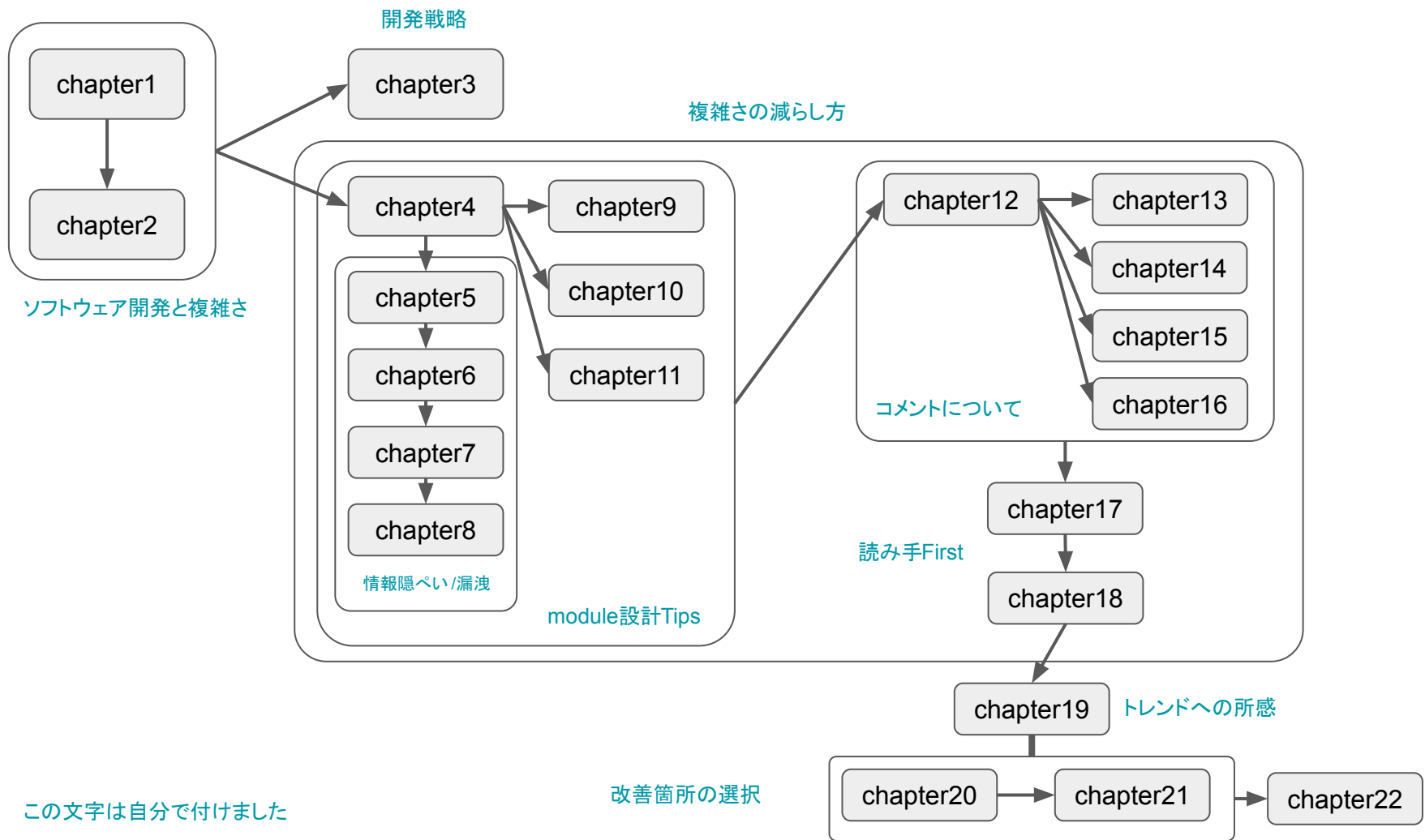
- 抽象的な話がメイン
 - 「設計は純粋な創作でありパズルのようなもの 楽しい 😊」と書いている

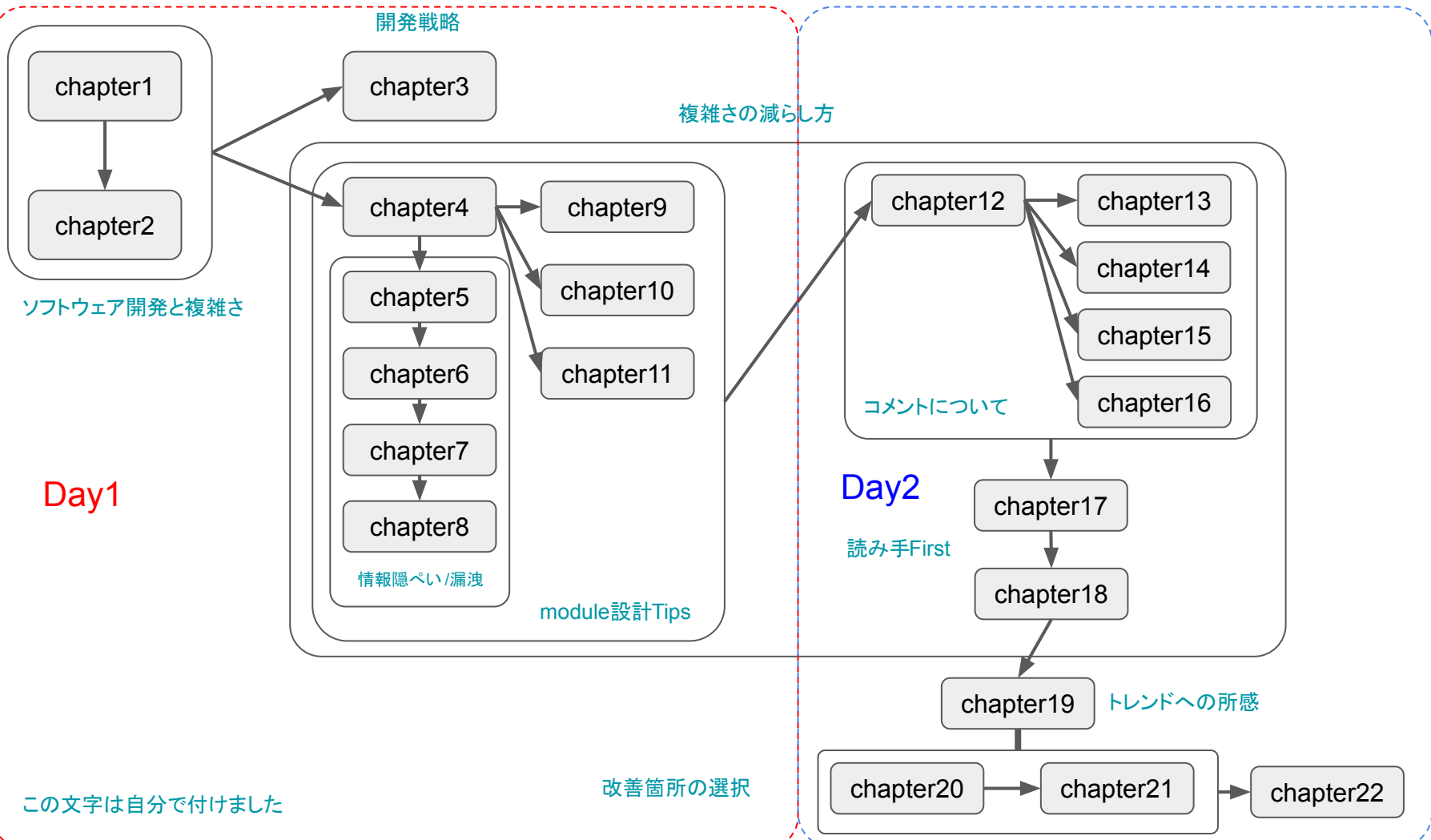
その他

- 2回に分けて紹介します(60+30)
 - 著者の1意見に過ぎない
 - 時間の関係上、詳細は省略している箇所があります
 - スライドは 60枚程度
- 和訳が間違っていたらすみません😓
 - 訳が微妙な英単語はそのまま紹介
 - 最後に、Kindle洋書の和訳について少しコラムがあります
- 試験的にGoogleスライドで作ってみました
 - 絵文字が入れやすいので嬉しい 😊💧💣👏🔥🚀
- 紹介程度なので、軽めに聞いておいてください

章立てと構成

1. Introduction
2. The Nature of Complexity
3. Working Code Isn't Enough
4. Modules Should Be Deep
5. Information Hiding (and Leakage)
6. General-Purpose Modules are Deeper
7. Different Layer, Different Abstraction
8. Pull Complexity Downwards
9. Better Together Or Better Apart?
10. Define Errors Out Of Existence
11. Design it Twice
12. Why Write Comments? The Four Excuses
13. Comments Should Describe Things that Aren't Obvious from the Code
14. Choosing Names
15. Write The Comments First
16. Modifying Existing Code
17. Consistency
18. Code Should be Obvious
19. Software Trends
20. Designing for Performance
21. Decide What Matters
22. Conclusion





Day1

Day2

この文字は自分で付けました

改善箇所の選択

Day1: Deep Module

複雑さとDeep module

ソフトウェア開発の現状 [Chapter1]

- ソフトウェア開発における1番の制約？
 - 我々が作っているシステムに対する理解力
 - 多くの機能を持ち複雑になるにつれ、全容把握が困難になる
- ウォーターフォールは最終的に複雑さが爆発四散する
 - 全体の設計を途中で変えられないため
 - アジャイルならば段階的に見直しができるので良い
- 開発者は設計を改善する機会を窺い、複雑さ解消を検討すべき
 - ソフトウェア開発は継続的行為ですよ

Complexity(複雑さ) [Chapter2]

- ソフトウェアシステムの構造に関する, 理解や変更を困難にするようなもの
 - 複雑さの欠片がincrementalに蓄積され, やがて取り返しのつかないことに ... 😱
- 全体の複雑性
 - 各部分に開発者が費やす時間の総和 ($C = \text{Sigma}(\text{cost_player} * \text{time_player})$)

$$C = \sum_p c_p t_p$$

- 複雑さは実装者よりも読み手のほうが体感する 😞
 - コードが読みにくいと指摘された場合は注意すべき
- 巨大システムでも理解が容易で変更が簡単なら, 複雑ではない
 - 優れた設計!




複雑さの種類

- (1) Change amplification (変更の増幅)
 - 単純な変更でも、変更箇所がたくさんある
- (2) Cognitive load (認知負荷)
 - タスク完了のために、開発者が知る必要のある知識の量
 - たくさんのAPI, たくさんのモジュールの依存関係
 - 複雑性はコードの行数に依存しない
 - コードの行数を多くすることでかえって Cognitive loadが減る場合もある
- (3) Unknown unknowns (未知の未知)
 - どこから手を付けてよいかわからない
 - 変更後にバグが出て初めてこれが発覚する
 - 3つの中で最悪 🙄
- システムが明確かつ理解しやすくなれば、それは良い設計である

複雑さを引き起こすもの・向き合い方

- 引き起こすもの
 - Dependency(依存関係)
 - 単体で理解が困難であること
 - Obscurity(あいまいさ)
 - 一般的すぎる変数名 (count, time)
 - 単位は？
 - 説明がドキュメントになればコードを読まないといけない
- 向き合い方
 - 複雑性の排除
 - コードをシンプル・明確にすることで複雑さを排除する
 - 特殊ケースの削除による分岐の単純化
 - 複雑性の隠ぺい
 - 開発者は複雑さの影響を部分的に被るだけで済む
 - カプセル化

開発戦略: Tactical vs Strategic [Chapter3]

-  Tactical Programming
 - 何かを早く機能させるような戦略で実装すること
 - 近視眼的で良い設計になることは少ない
 - ひとたびこれで進めると, 方針変更するのは困難
 - tactical tornado  短期間にtacticalな開発で機能を量産すること
 - tornado通過後はコードが複雑になり (破壊), 復興するヒーローが必要
-  Strategic Programming
 - 将来的な拡張性やメンテナンス性も鑑みて実装すること
 - 大局的な視点
 - 機能するコードだけでは不十分
 - 優れた設計を実現するためには, 投資の考え方が不可欠
 - 総開発時間の10~20%程度が良い
 - 優れたエンジニアが集まりやすくなる
 - 優れたエンジニアは, コードがクリーンなほうが良いため

長期スパンでは, Strategicが優れている

- 良い設計を目指し, 動くことはその副産物 とする考えでもよいかも
 - ！？

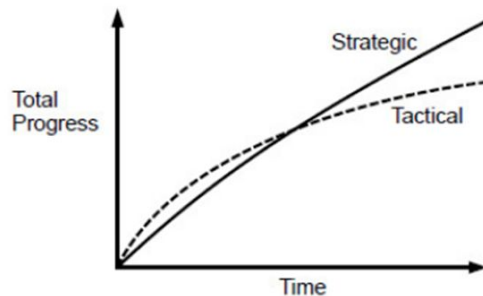


Figure 3.1: At the beginning, a tactical approach to programming will make progress more quickly than a strategic approach. However, complexity accumulates more rapidly under the tactical approach, which reduces productivity. Over time, the strategic approach results in greater progress. Note: this figure is intended only as a qualitative illustration; I am not aware of any empirical measurements of the precise shapes of the curves.

Module [Chapter4]

- 「複雑さの隠ぺい」へのアプローチの1つ
 - クラス, サブシステム, サービス
 - Interface + implement(実装) で成り立つ
 - モジュール同士は, ある程度の依存関係が存在する
 - 良いモジュール設計の目標は, 依存関係を最小限にすること
 - 良いモジュール設計は, 複雑さを減らすことができる
- Interfaceと実装の分離
 - IFは別モジュール作業者がそのモジュールを使うために知るべき機能で構成されるのが理想
 - `std::map`のユーザは, 追加・取り出しなど方法が分かればよい
 - 二分木の中身を知る必要はない
 - 最良のモジュール: interfaceが実装よりもはるかに単純なモジュール
 - 他モジュールに与える複雑さを抑制できる
 - 他モジュールに影響を与えることなく内部の変更がしやすい

Interfaceが持つ情報

- Interfaceは2つの情報を持つ(formal/informal)
 - formal information: シグネチャ, 引数の名前や型の情報
 - プログラムで解析可能
 - informal information: 機能内容や先に呼ぶべきモジュールの情報
 - プログラムで解析不可能 コメント書くしかない
- informal informationは, formal informationより複雑
 - 明確になることで, Unknown unknownsが軽減されて良い

Abstraction(抽象化)

- 抽象化：エンティティを簡略化したviewのこと
 - 重要でないものの詳細を削る
 - interfaceはモジュールの機能の簡略化した viewである
 - 抽象化できるならばできるだけ、優れたものになる
- 抽象化が失敗する可能性
 - (1) 重要でない要素を残してしまう
 - 他モジュール開発者が IFを確認するときの Cognitive loadが増えてしまう
 - (2) 重要な要素を削る
 - Obscurity(あいまいさ)が増し、複雑さが増大してしまう
 - “False abstraction” と呼ぶ

🎯 Deep Module

- 最高のmoduleのあり方は？
 - シンプルなinterfaceでありながら多くの機能を有するのが良い！
 - 適切な抽象化による情報隠ぺいの結果, シンプルな interfaceを実現
 - deep module と呼ぶ
 - 最小の導入コストと最大の機能
 - 対: Shallow module
- しかし, 慣習としては「classは小さく浅いほうが良い」とされている
 - classitis症候群
 - JavaのfileStream

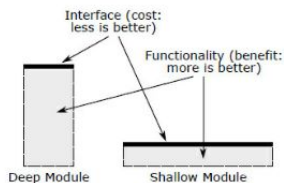


Figure 4.1: Deep and shallow modules. The best modules are deep: they allow a lot of functionality to be accessed through a simple interface. A shallow module is one with a relatively complex interface, but not much functionality: it doesn't hide much complexity.

Deep Moduleの例: Unixにおけるファイル I/O

- 5つだけのシンプルなシグネチャ
 - 実装としては数十万行のコードが必要
 - ディスク上でのファイルの表現方法
 - 権限処理
 - メモリキャッシュによる効率化
- 使用者は↑の詳細を気にせずに使用することができる
 - Deep module 😊

```
int open (const char *path, int flags, mode_t permissions) ;  
ssize_t read (int fd, void *buffer, size_t count) ;  
ssize_t write (int fd, const void *buffer, size_t count) ;  
off_t lseek (int fd, off_t offset, int referencePosition) ;  
int close (int fd) ;
```

Day1: Deep Module

どうやってDeep Moduleを設計するか？

Information hiding and leakage (情報の隠ぺいと漏洩) [C5]

- 「情報の隠ぺい」はDeep Module作成の重要テクニックの1つ
 - IFをシンプルにして, 内部情報 (実装)を露出させない
 - 例:効率的なアクセスを実現するようなデータ構造とアルゴリズム
 - 隠ぺいするメリット
 - Cognitive loadを軽減する
 - 実装が変わっても, 外部システムへの変更の波及を抑えることができる
- 情報の漏洩
 - 設計上の情報・知識が, 複数のモジュールに影響すること
 - 1つの設計変更が複数モジュールに影響する
 - 例:temporal decomposition (時間分解)
 - 時間軸でモジュールを分けてしまう
 - ファイルの読み込み/修正/書き込み を別class化してしまうこと
 - 読み込みと書き込みは同じファイルフォーマットの知識が必要になる

モジュールを general-purpose(汎用的)にする [Chapter6]

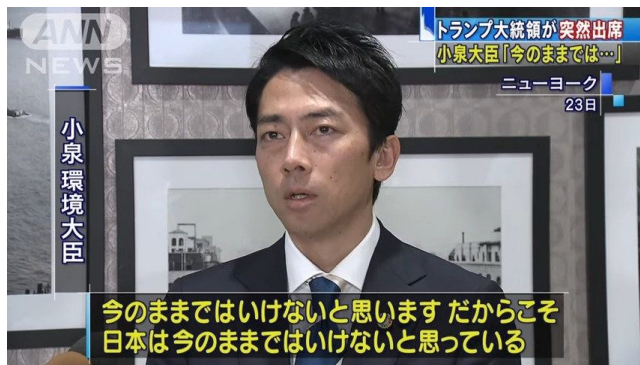
- 専門的なモジュールにすると、複雑さが増しやすい
 - 過度な専門化が複雑さを生み出す最大の原因かも (筆者調べ)
- 汎用的なコードはシンプル・クリーンで理解しやすい
 - deep module作成のテクニックの1つ
 - 意外にも実装のコードが少なくなりがち (本当?)
- 「やや」汎用的にするのが良い
 - 汎用的にしすぎてかえって現在の目的にそぐわない場合があるので
- 汎用化されたIFであるか確認する質問
 - 現在のニーズを満足する最も単純な IFは何か？
 - このメソッドを使うシチュエーションはいくつあるか？
 - このAPIは現在のニーズにおいて使いやすいものであるか？

「やや汎用的」なIFにする方法

- (1) 専門的なコードを上下レイヤどちらかに押しやる
 - 専門的なコードは完全には除外できない
 - 汎用的な箇所と分離できれば良い
 - 上位クラスに専門性を押しやる場合
 - 例: テキストエディタ
 - テキストクラスを汎用的にして, 特殊なキー操作は UIクラスに置く
 - バックスペースキーの動作
 - 下位クラスに専門性を押しやる場合
 - 例: デバイスドライバ
 - 数多のデバイスタイプをサポートする必要がある
 - デバイスごとにAPIを定義
 - 各デバイスの専門性が漏れないようにできる
- (2) 特殊ケースを排除する
 - if文による分岐が減らせシンプルになる (後述)



レイヤごとに異なる抽象化をする [Chapter7]

- 異なる層が同じような抽象化をしていると, path-throughが増えがち
 - 似たようなシグネチャを呼ぶだけのメソッド
 - 複雑さが増してしまうので良くない
 - Cognitive loadが増加する
 - Shallow module
 - 追加機能があるならば, そのメソッドは問題ない
 - Decoratorにも同じような議論ができる



path-through

複雑さを下位に押しやる [Chapter8]

- どうしても避けられない複雑な処理への対応
 -  モジュール利用者に処理させるべき？
 -  モジュール内部で処理すべき？
 - 実装者が苦しもう 😊
 - シンプルなinterfaceは、シンプルな実装よりも重要
 - 複雑さを内部に隠ぺいする
- 押しやって良い場合
 - 下位に押しやる複雑さが、モジュールが提供する機能と関連する場合
 - 下位に押しやることでシンプルになる場合

実装を同じ箇所にするか？ 分けるか？ [Chapter9]

- 良く飛び交う質問 どうやって判断する？
 - 目標:システムの複雑さを減らすこと
 - 良い抽象化にならないならば分けるべきではない
 - 細かく分けると追跡が難しくなるし、interfaceの数も増えてしまう
 - いくつかのモジュール間で情報漏洩の可能性もある
 - メソッド自体の行数は減るが、それが複雑さを減らすわけではない(『CleanCode』より)
- まとめる場合
 - 情報が共有される場合
 - interfaceをシンプルにしたい場合
 - 重複を排除したい場合
- 分離する場合
 - 機能が完全に独立している場合
- まずdeep moduleを作ってその後に分割を検討する で良い

例外を消し去る [Chapter10]

- 例外の存在は複雑さを増大させる
 - 投げるのは簡単だけど...
 - 異常系処理のコードは、通常コードよりも記述が難しい
 - 例外処理が別の例外を引き起こすことも 😬
 - 動作確認, デバッグも難しい
- 例外処理をする箇所をどこまで減らせるか？が重要
 - 過剰防衛しないほうが良い (筆者の見解)
 - Tclの設計で痛い目にあっただけ
 - ※Tcl(Tool Command Language ティクル)
 - <https://www.freesoftnet.co.jp/webfiles/tclkits/doc/tcl.html>
 - ※変数宣言が不要らしい

例外ハンドラを減らす4つの方法

- (1) 存在しないエラーを定義する
 - 例外を投げずに、「何もしない」をする
- (2) 例外マスキングをする
 - 例: TCPのネットワークプロトコル
 - パケット損失時に再送処理をする
 - クライアント側は再送されたことを知らずに過ごせる
- (3) 例外の集約
 - 1つの処理で多くの例外を処理したい
- (4) ただクラッシュさせる
 - アプリケーションでやれることがあまりない場合
 - 例1: メモリ不足(どうせアプリケーションが落ちる)
 - 例2: ネットワークソケットが開けない

Design it Twice [Chapter11]

- 初回の設計が最良であることなどほとんどない
 - 何回か設計してみよう
- 賢い人ほどDesign it Twiceが難しい
 - 初回設計案で進めがち



これはRun it Twice

Day1 まとめ①

- Complexity(複雑さ)を軽減したい [Chapter1, 2]
 - Change amplification (変更の増幅)
 - Cognitive load (認知の負荷)
 - Unknown unknowns (未知の未知)
- どうやって減らすか？
 - Strategic Programming [Chapter3]
 - tactical tornado🌀はやめたほうがよい
 - Deep moduleを作ろう [Chapter4]
 - シンプルなinterfaceで豊富な機能
 - 複雑な処理は実装に隠ぺいされている
 - モジュール間の依存関係を減らすことができる

Day1 まとめ②

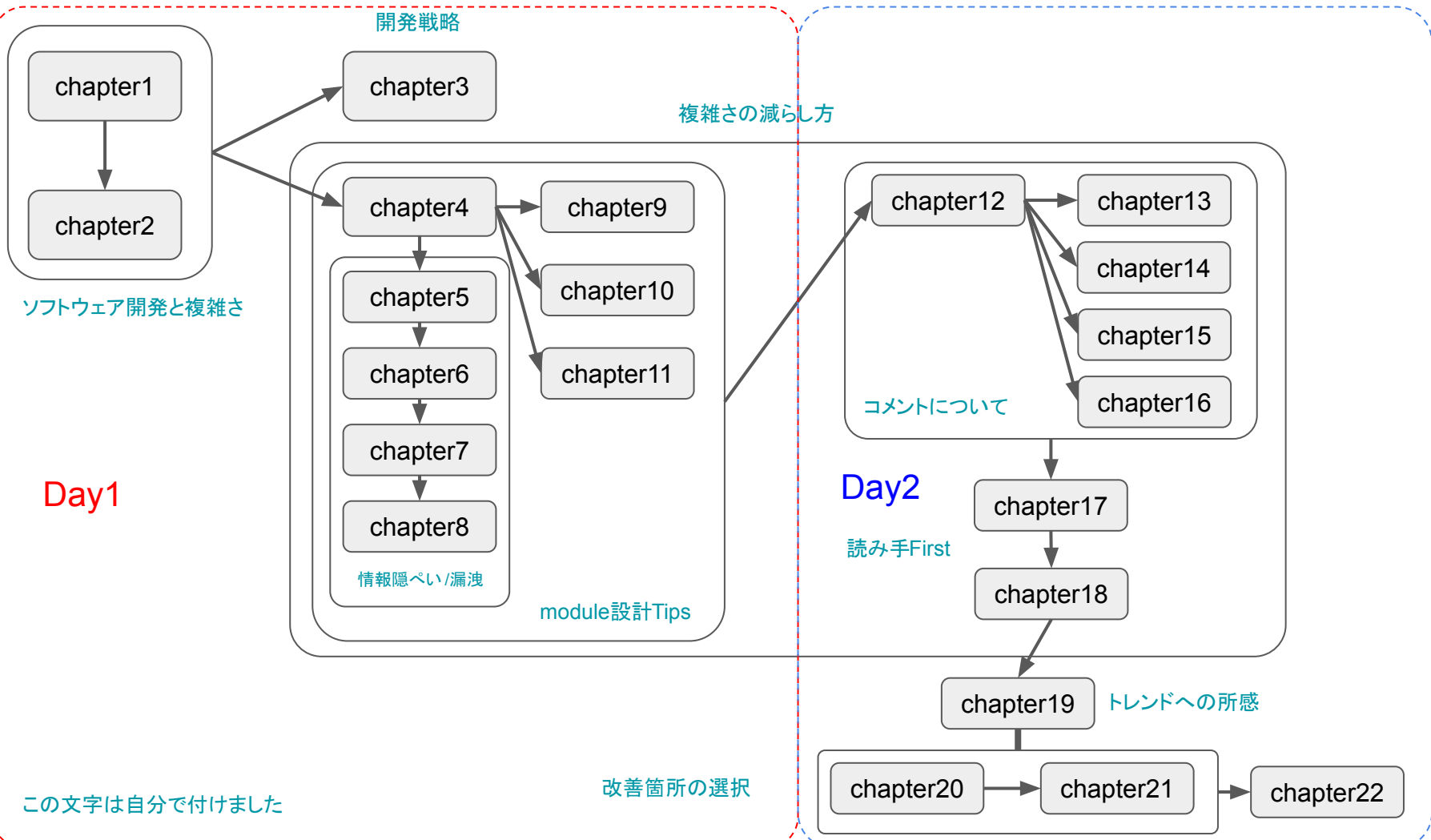
- Deep module 設計テクニック
 - 目標:「シンプルなinterfaceだが豊富な機能」を実現する
 - テクニック(インデント付けたけど, 全部フラットで良いかもしれない 🤔)
 - 情報の隠ぺい/漏洩を意識する [Chapter5]
 - 複雑さを下位レイヤに押しやる [Chapter8]
 - 実装を分けるか? 同じ箇所にするか? [Chapter9]
 - まずdeep moduleを作り, 分割はその後検討する
 - 抽象化してみる [Chapter5]
 - やや汎用化する [Chapter6]
 - レイヤ毎に異なる抽象化をする [Chapter7]
 - 例外を減らす [Chapter10]
 - 2回設計してみる [Chapter11]

Day1 感想

- 確かに自分も関数を分けがち
 - 行数が短くなれば多分見やすいだろうと思っていた
- EasyとSimpleは違う
 - Simpleにするのは難しいことが多い
- あくまで著者の主張
 - 開発現場との乖離はある程度あると思う
 - とはいえ息の長い開発の場合には参考になる気がする
 - 例外処理の扱いは以前話題になった
- deep moduleの狙い？
 - システムが巨大化するにつれ、モジュールは必ず増える
 - deepにしておくことでモジュール総数を減らせる, ということが本質かも
 - 200個のdeep moduleと400個のshallow moduleはどちらが認知負荷低い？
 - 開発担当領域を決めればマシになるのかも

Day2: 最後まで+α

読み手ファーストなコメント



Day1 まとめ①

- Complexity(複雑さ)を軽減したい [Chapter1, 2]
 - Change amplification (変更の増幅)
 - Cognitive load (認知の負荷)
 - Unknown unknowns (未知の未知)
- どうやって減らすか？
 - Strategic Programming [Chapter3]
 - tactical tornado🌀はやめたほうがよい
 - Deep moduleを作ろう [Chapter4]
 - シンプルなinterfaceで豊富な機能
 - 複雑な処理は実装に隠ぺいされている
 - モジュール間の依存関係を減らすことができる




Day1 まとめ②

- Deep module 設計テクニック
 - 目標:「シンプルなinterfaceだが豊富な機能」を実現する
 - テクニック(インデント付けたけど, 全部フラットで良いかもしれない 🤔)
 - 情報の隠ぺい/漏洩を意識する [Chapter5]
 - 複雑さを下位レイヤに押しやる [Chapter8]
 - 実装を分けるか? 同じ箇所にするか? [Chapter9]
 - 抽象化してみる [Chapter5]
 - やや汎用化する [Chapter6]
 - レイヤ毎に異なる抽象化をする [Chapter7]
 - 例外を減らす [Chapter10]
 - 2回設計してみる [Chapter11]
 - まずdeep moduleを作り, 分割はその後検討するのが良い [Chapter9]

なぜコメントを書くのか？ [Chapter12]

- コメントやドキュメントは、抽象化のために重要
 - これらで補足することで複雑さを隠すことができる
 - メソッドを読めば分かるけど、それはコストが大きい
 - Cognitive loadが大きい
 - 事例: メソッドを使うために実装を読む必要がある 😞
 - メソッドの抽象化ができていない証拠
 - interfaceを読むだけにしたい
- コードからは分からないような設計思想の伝達をコメントに委ねる
 - Chapter13

コメントを書かない!?!?

- よくある言い訳4選
 - (1) 良いコードはそれ自体がドキュメントになるから
 - 「アイスクリームは健康によい   
」くらいの神話
 - 例えば:なぜこの実装に至ったのか？に対して答えられない
 - (2) 時間がないから
 - 新機能開発に追われた時に起こりがち (耳が痛い)
 - (3) 古くなったときにミスリードを誘うから
 - 多大(≡頻繁)な更新コストがあるわけでもない
 - コードとドキュメントを近くに置くことである程度是正可能
 - (4) 役に立ったコメントを見たことがないから
 - 良い書き方は何か？を知ればあなたも役に立つコメントが書ける
- 良いコメントを書きましょう でもどうやって？

コードから分からない事柄をコメントで描写する [C13]

- 良いコメントは, シンプルな形で上位(神様?)の視点を提供する
 - // 事前に呼び出すべきメソッドは〇〇です
 - // このメソッドが呼び出された後, ネットワークトラフィックは△△に制限されます
 - 実装を読まずにこの情報を得られることは良いこと
 - 他モジュール開発者が interface+コメント だけでモジュールを理解できる
- コメントの種類
 - (1) 💖 Interface (or Class) について
 - (2) 💖 データ構造について
 - (3) 関数, メソッドなどの実装について
 - (4) Cross-moduleコメント
 - モジュールの依存関係を記す

良いコメントを書くために

- 規約の策定
 - (1) 何をコメントするか？
 - 書く内容も定まるので、コメントを書くハードルが下がる
 - (2) どの形式で書くのか？
 - ドキュメントツールの使用
 - 従うように書くことで、一貫性が確保でき読みやすくなる
 - Doxygen <http://www.doxygen.jp/>
 - pdoc3 <https://pypi.org/project/pdoc3/>
- 実装と同じことは書かない
 - 実装をみてそのコメントが書けるなら不要
 - 同じ単語をコメントで書いている場合は注意

```
const auto volume=calculate_volume(p); //体積を求める
```

良いコメントを書くために

- コメントを書くレベル
 - 高レベル視点のコメントは抽象的な理解や直感を助ける
 - 低レベル視点のコメントはより精細な把握を助ける
- how ではなく, what と why を書く
 - 変数がどう操作されるかではなく, 何を表しなぜこのようになるのか記述する
- 読み手にとって明白になるようにコメントを書く

変数の命名 [Chapter14]

- 平凡な名前は曖昧さや誤解を生むのでやめよう
 - 複雑さにつながる
 - count😞, cnt😞, c😡
 - getCount()😱
 - 例: block→fileBlock or diskBlock?
 - 読み手次第で変わってしまう
- 良い変数名の指標
 - 明確で一貫性がある
 - 余分な箇所がない
 - 開発者間で共通認識がある
- 定義した箇所から遠くにあるほど、説明的な変数名にすべき

Go言語の例

- Go言語では変数名は短い
 - <https://github.com/golang/go/wiki/CodeReviewComments#variable-names>
 - 変数名が短いほうが、関数のふるまいなどが分かりやすい
 - （変数名はノイズということ？）
 - メソッド内ならそれでよいが、遠くにある場合は説明的にすべき
 - グローバル変数とかもこれに該当する

初めにコメントを書く [Chapter15]

- 開発プロセスの最初にコメントを書こう
 - 最初にコメントを書くのは結構楽しい
 - 最後に書こうとしても、どうせ忘れている
 - 結局コードを繰り返すようなコメントにしかない
 - 「Q:なぜ途中に書かないの?」「A:まだコードが変化するから」
 - コードが安定するころには手遅れ
- デザインツールとしてコメントを使う
 - 上手くコメントが書けないならば、その設計は怪しいかも 😊💧
 - コメントが長いinterfaceは、適切な抽象化ができていないかも 😊💧
 - 簡潔なコメントができたならば、良い設計なので自信持って良いかも 😊
 - 推奨する作業順序
 - IF→重要なPublicメソッド→メソッド実装→追加メソッドのコメント

既存コードの変更時に気を付けること [Chapter16]

- 既存コードを追加開発する場合, tacticalになりがち
 - 少ない実装で機能追加を達成しようとしてしまう
 - Strategicであり続けよう
- 理想: 機能追加後に最初からそれを含んだ設計になっていること
 - 設計が悪くなっているなら本末転倒
 - 商用コードでは両立は難しい ...
- コメントの重複はなくそう
 - 古いコメントは刈り取ろう
- コメントはコードに書こう
 - コミットログだけに書くんじゃないよ

一貫性 [Chapter17]

- 一貫性は複雑さを減らす
 - 規律あるコーディングスタイルを導入しよう
 - 迷わなくて済むので, Cognitive loadを減らすことができる
 - 違和感に気が付きやすくなり, バグの発見頻度が上がる
 - 一貫性を適用する場面
 - 変数名
 - コーディングスタイル
 - #includeの順序 など
 - フォーマッタも使おう
 - interface
 - .size()
 - design patternの採用

コードを明快にしよう [Chapter18]

- 理解しやすさは読み手の判断
 - 変数名の選択, 一貫性が特に重要
- イベント駆動はコードの流れを追いにくする
 - ハンドラが呼ばれるタイミングはコメントで補足する
- ソフトウェアは読み手のためにデザインされるべき
 - 読み手にとって明快でない場合, 足りない情報がある
 - 抽象化して情報を減らしても良い
 - 読み手が考えないといけなことを減らす
 - for文のことを考えたくない

```
boost::push_back(results, hogs);
```

```
for(const auto& hoge : hogs)  
{  
    results.emplace_back(hoge);  
}
```


トレンドへの所感 [Chapter19]

- OOP(オブジェクト指向プログラミング)
 - 継承はchange amplificationを軽減するが、親 classとの依存関係は情報漏洩を引き起こす
- Agile
 - 機能開発にフォーカスしすぎるので tacticalになりがち
 - incrementalな開発は良いことだが、incrementalな抽象化を目指すべき
- Unit Test
 - リファクタ時の安心が得られる
 - 良い設計のための開発を促進するので良い
- Design Pattern
 - 適用は基本的には良いことだが、無理に当てはめないように気を付ける
- Getter and Setter
 - shallow moduleになる(=deepにならない)のでやめたほうがよい

シンプルな設計とアプリパフォーマンス [Chapter20]

- シンプルな設計の方が、パフォーマンスも良いことが多い
 - パフォーマンスのために複雑さを許容する場合は要検討
 - (枝刈り処理など?)
- パフォーマンス改善の時に考えること
 - クリティカルパスへの対処から考え始める
 - コストが高い処理を把握しておく
 - ネットワークのレイテンシ

重要事項を決める [Chapter21]

- 本当に重大な問題を精査する
 - やみくもに対応したり, 重要なものを見落とさないようにする
 - interfaceが大量に増えたり.....
 - (優先順位をつけて対処に当たろう)

感想

- 英語の技術書を1冊読めて良かった
 - 英語苦手なので嬉しかった 😊
 - ところどころ和訳をせざるを得なかった 😞
 - 読んでいて英文が滑る 対策募集中です
 - 本を読むということは、勉強会の準備にしては大変すぎるのでやらないほうがよい
 - ワンオペ 複数人で手分けして読むのはとても良いと思います
- 本について
 - 「開発時のつらみ」を分析して、それぞれの対策がまとまっていた
 - 脳内が整理されたように思う 🧠
- リーダブルコードの次の本として良いかも
 - リーダブルコードは実装レベルでの話、本書は設計レベルでの話
 - 実戦的なのは前者

感想

- 最近「シンプルな実装とは何か？」みたいな話が出ている
 - 例外処理を減らすなどは良いアプローチだと思う
 - 「抽象的にする」箇所は boost の range 記法と近い思想がありそう
- Tactical tornado について
 - とはいえ、まずは実装を完遂できることが大切だと思う
 - まず愚直に書いて、道中見直せばよいと思う
 - 実装力はつけるしかない 毎日トレーニング パワー 💪

Reference

1. <https://zenn.dev/tatane616/scraps/8b6d877aebd9b4>
2. <https://qiita.com/immrshc/items/73f9a9c5d7453273e371>
3. <https://ky-yk-d.hatenablog.com/entry/2022/01/04/100000>
4. https://budougumi0617.github.io/2021/12/17/review_aposd/
5. https://speakerdeck.com/yosuke_furukawa/a-philosophy-of-software-design-qian-ban