誰でも編集可能にしてあるのでどんどん書き換えてください。

読書会の進め方は、対象ページの全文を読み上げる形にしようと思います。ソースコードは読み上げません。声に出したほうが時間はかかりますが細かいところを見逃さなくなると思うからです。

やり方については後々考えていきましょう。

読書メモ

第1章から第4章までは読書会では飛ばします。後から読み直すと学びがあるかもしれません。

第1章 > ソフトウェアアーキテクチャの目的は、求められるシステムを構築・保守するために必要な人材を最小限に抑えることである。
> 崩壊したコードを書くほうがクリーンなコードを書くよりも常に遅い。

第2章 > ソフトウェア開発チームには、機能の緊急性よりもアーキテクチャの重要性を強く主張する責任が求められる

第3章
第4章 著者はGOTO文がまだまだ使われていた1970年代からのプログラマ

第5章　オブジェクト指向プログラミング

> だが、オブジェクト指向（OO: Object Oriented）とは何だろうか？ひとつの答えに「データと関数の組み合わせ」がある。...あるいは、カプセル化、継承、ポリモーフィズム

Wikipediaの「オブジェクト指向プログラミング」が網羅的によくまとめていて、カプセル化、継承、ポリモーフィズムを含む7つの特徴をあげています。

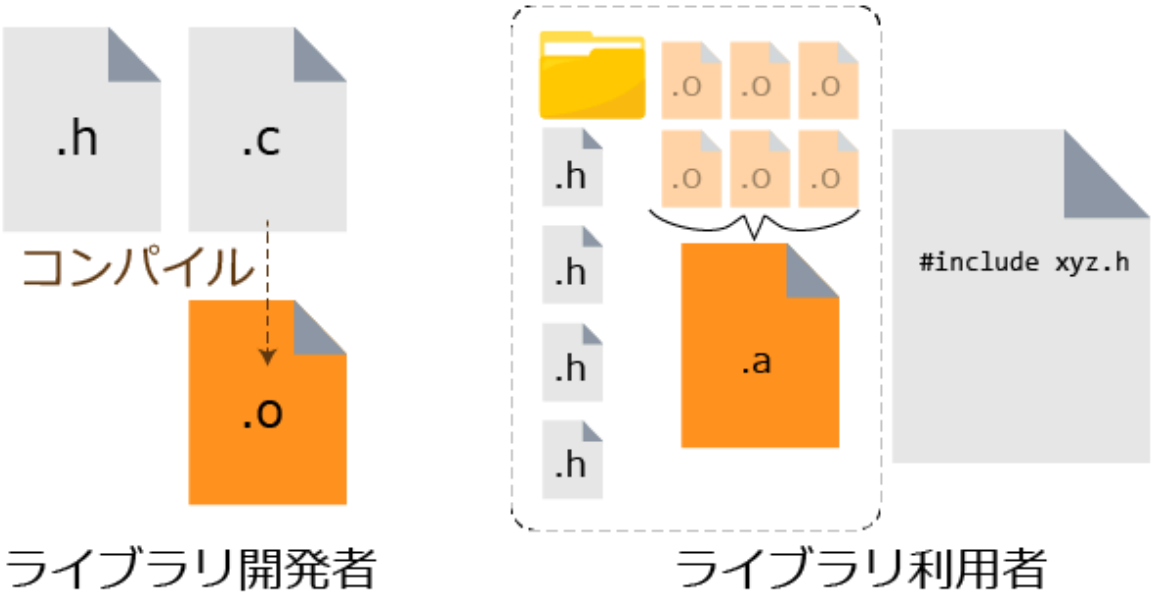
> もうひとつのよくある答えに「現実の世界をモデル化する方法」がある。

Animalクラスを継承したCatクラスがあって、Catクラスは「ニャー」と鳴く(print)だけみたいな例がよくあって、何がしたいのかわからんという話がTwitterなどでよく話題に登ります。現実のネコは「ニャー」と鳴くだけでなく様々な動作をしますし、そもそも現実のネコをモデル化したソフトウェアで何がしたいのか使いみちがわかりません。

カプセル化とは？

> C 言語でも完ぺきなカプセル化は実現できる。

ここでは完全なカプセル化である「実装を隠す(C言語)」と弱いカプセル化であるprivate,protectedをつかった「アクセスを制限する」の2つがでてきています。商用ライブラリが頻繁に使われていた環境では実装の隠蔽に大きな意味がありました。OSSが当たり前の世界で生きていると実装を隠す完全なカプセル化の利点は分かりづらいと思います。



継承とは？

```
int main(int ac, char** av) {
  struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
  struct NamedPoint* upperRight = makeNamedPoint
    (1.0, 1.0, "upperRight");
  printf("distance=%f\n",
    distance(
            (struct Point*) origin,
            (struct Point*) upperRight));
}
```

リスト5-7を見るとNamedPoint* originで宣言されたものがPoint*にキャストされて使われています。Javaではキャストを明示的に行うのではなく

```
class MyClass {
  public void superCoolMethod(argument: BaseArgument) { ... }
}
```

というコードがあればsuperCoolMethod()にclass DerivedArgument extends BaseArgumentの引数を渡すことが出来ますね。

> 要するに、NamedPoint は Point の純粋な上位集合であり

上位集合はSuper set。ここはちゃんとした定義をせず「上位集合」なる数学的用語を用いるのは誤解をまねくだけのような気がします…

> 2 つのフィールドの順序が Pointと同じになっているからだ

Cで継承っぽい動作を実現するために必要なトリックです

ポリモーフィズムとは？

```
struct FILE {
  void (*open)(char* name, int mode);
  void (*close)();
  int (*read)();
  void (*write)(char);
  void (*seek)(long index, int mode);
};
```

これはJavaのinterfaceと似た働きをする構造体です。Cの関数ポインタを使っています。

> OO 言語がポリモーフィズムを提供してくれたわけではないが、それを安全かつ便利にしてくれたのは OO 言語だ。

ポリモーフィズムのパワー

UnixとLinuxはCで書かれているという話。Cで書かれているOSがデバイスドライバでポリモーフィズムを実現するために上記のような仕組みが使われたと思われます。Linuxのコードどこに行けば読めるのか知りません…

依存関係逆転

(リチャード)個人的には依存関係逆転とかDependency Injectionっていうと私が混乱するので、毎回脳内で”Programming to interface”に変換しています。

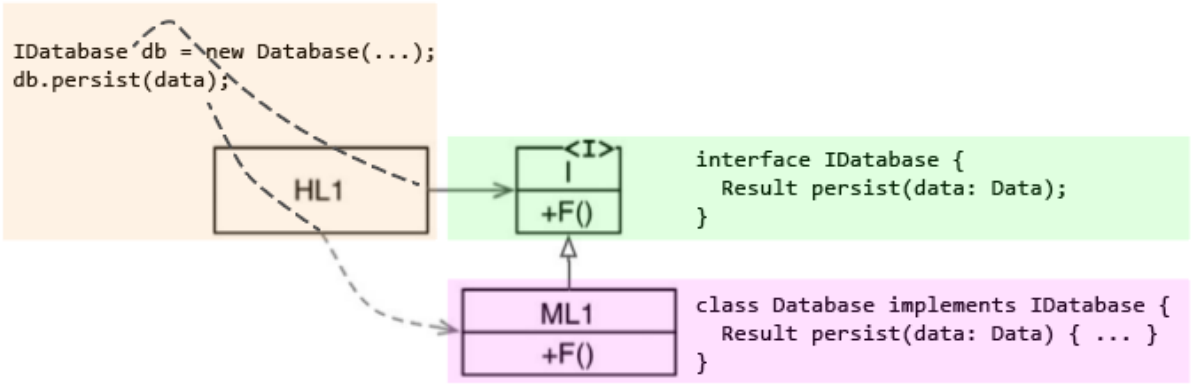


図 5-2　依存関係逆転

実用上はinterfaceに対する実装を入れ替えることで得られるメリットには私は懐疑的です。大きなモジュールだったらinterfaceを統一するだけでも大変ですし、SLF4JやJDBCがinterfaceであるからと言って実装を入れ替える作業をしたい人はあまりいないと思います。反対に小さなモジュールだったら同じinterfaceに対する実装を入れ替えるのは容易かもしれませんが、モジュールが小さいため得られるメリットは少なくなるでしょう。
多くの場合依存性逆転・Dependency Injectionはテストのために用いられることが多いと思います。

まとめ

> 「 OO とは何か？」...答えは明らかだ。OO とは「ポリモーフィズムを使用することで、システムにあるすべてのソースコードの依存関係を絶対的に制御する能力」である。

最近(でもない?)「オブジェクト志向 不要」論を唱える人がいて、そのままGoogleで検索するとそんな記事がいくつかヒットします。このへんなど。特に継承については濫用するとソースコードを保守・変更しづらくし使い所が難しい、あるいは使うなという事を言う人もいます。Scalaでは型クラスだけを利用することで(Java的な「普通の」)継承を一切使わないというスタイルもあります。この章ではオブジェクト指向で嬉しいことってなんだっけ？というところから始まってBob Martinさんの結論が述べられています。

第6章　関数型プログラミング

> 1930 年代に Alonzo Church が発明したラムダ計算にもとづいている
サラッと触れているけど、ラムダ計算は大変に難しいトピックです。

整数の二乗

> 無限リストの要素は、アクセスされるまで評価されないからだ。

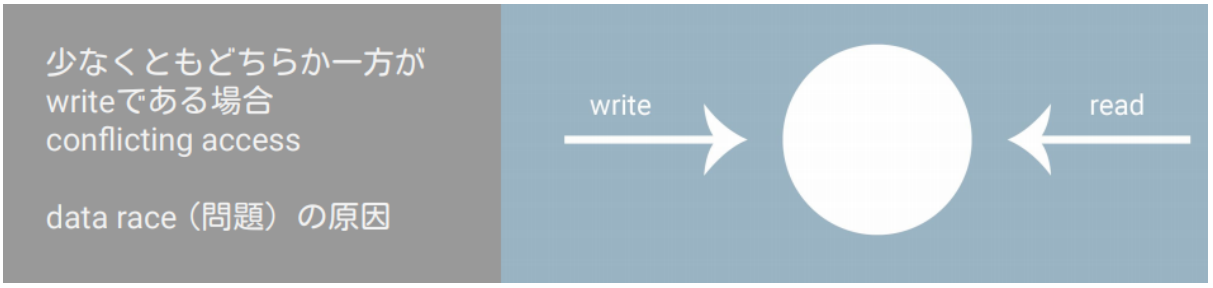
遅延評価ですね。Scalaはデフォルトが正格評価で、一部遅延評価用のクラスがあります。遅延評価の機能を持ったMonixなどのライブラリもあります。

>「関数型言語の変数は変化しない」

関数の中で値を変えることができる可「変」という意味じゃなくて数学の関数のようにf(x)のxを外から与える段階で変えられるという意味。ラムダ計算をしっかり勉強するとこの辺の話が詳しく出てきます。

不変性とアーキテクチャ

Oracle Currency guideやJava Memory Modelにとても関連が深い話。不変ではなく可変(書き込み可能)な変数は平行・非同期処理ではとても扱いにくいです。すべてが不変な変数なら43ページあるJava Memory Modelがほとんど必要なくなるくらい単純になります。それくらい不変変数は効果が大きいです。
平行・非同期処理がなぜ重要かは私の資料にとてもよくまとまっていますw 以下はその私の資料から抜粋した図です。



可変性の分離

不変な変数のみで構成されるアプリケーションのほうが、特に平行・非同期環境で扱いやすいですよという話。上に同じ。ただしアプリケーションのすべてを不変な変数で構成するのは難しく、むしろ現代のアプリケーションは内部状態が増えていると言えなくもないので、可変変数の適切な隔離と関数が重要となりそうです。

イベントソーシング

イベントソーシングも結構大きなトピックでここにあるようにサラッと解説しただけではわからないと思います。ちなみにScala/JavaのOSSであるAkkaにはイベントソーシングのためのモジュールがあります。

第III部

リチャードが読む。
> SOLID原則の目的は...ソフトウェア構造の定義に役立つ。

目的がないと、たまたま頭に思い浮かんだ原則を押し付けあうだけの一貫性のないコードレビューになります。原則は目的を達成するためにある。

> 単一責任の原則 … 依存関係逆転の原則

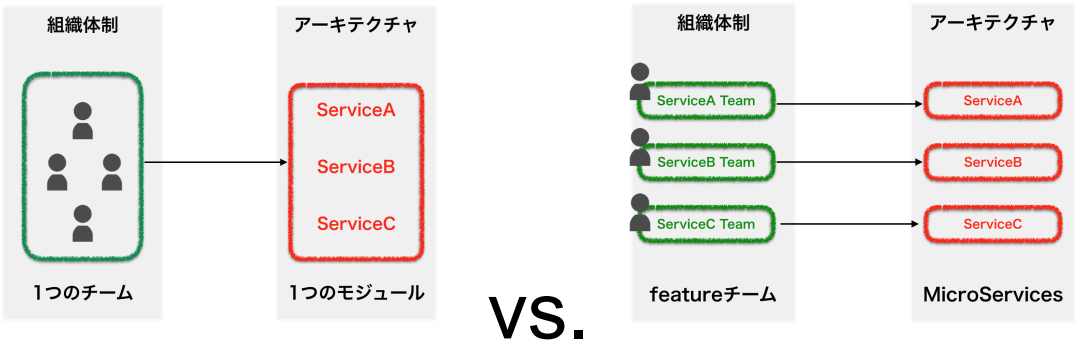
コンウェイの法則が言及されていることに注目。第7章でちょっと触れます。

第7章 単一責任の原則

Wikipediaによると、実はボブ・マーティン氏自身が最初に提唱したらしい。もともとは凝集度と結合度というアイデアに基づいて提案された原則。この本によって新たにコンウェイの法則が文脈に加わったことに注目してください。

参考:

- コンウェイの法則と逆コンウェイの法則から組織構造を考える
- なぜMicroservicesか?



> SOLID 原則のなかで最も誤解されがちなのが … 違反している例を見るのが一番だ。

モジュール = ソースファイル。

> 症例1: …これらのメソッドをひとつの Employee クラスに入れると…も呼び出されていることには気づかなかった。

アクターが異なる…財務部門向けアプリケーションと人事部門向けアプリケーションがソースファイルを共有しているというケースは現実から外れすぎていないですかね?

> 症例2: …
読み飛ばします

> 解決策 -> まとめ
読み飛ばす。クラス図だけ見るとよいのでは。

現実に適用するには「アクターをどこまで細分化できるのか？」などモヤモヤが残る章だと思います。後の章で凝集性と結合度の話が出てくるので、そこを読めばもっとイメージがクリアになるかもしまれません。

ゴッド・クラスのような明らかに有害なものはわかりやすいですが、現実では意見の分かれるケースをどう扱うかが大事になると思います。原則を押し付けるのではなく、原則の目的を理解するのが大事かなと。

- 変更に強いこと
- 理解しやすいこと
- コンポーネントの基盤として、多くのソフトウェアシステムで利用できること

単一責任原則では特に「変更に強いこと」「理解しやすいこと」が大事そうですね。Pull Request上げるときに影響範囲の調査が大変なのはイヤだし、Pull Request上げる前にソースコードが読みづいらのは困ります。

普段ソースコードを触ってレビューをしていれば、コードの質が時間が経つにつれ悪化しているのかが感覚としてわかるはずです。それを改善するために使えるのが原則であり「この原則に従ってないからこのコードはダメ」と、たまたま思いついた原則を挙げて批判するのは良くないです。一貫性大事、目的を見失わないこと大事。

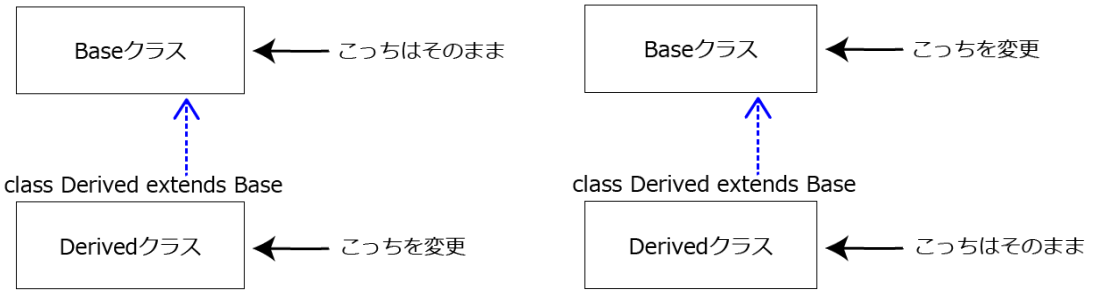
参考:

- 実況中継シリーズ「開発現場で役立たせるための設計原則とパターン」#builderscon 2018
- 単一責任の原則(SRP)についての見解と方法論 (途中まではいい感じの記事なのに途中からわけがわからなくなった)

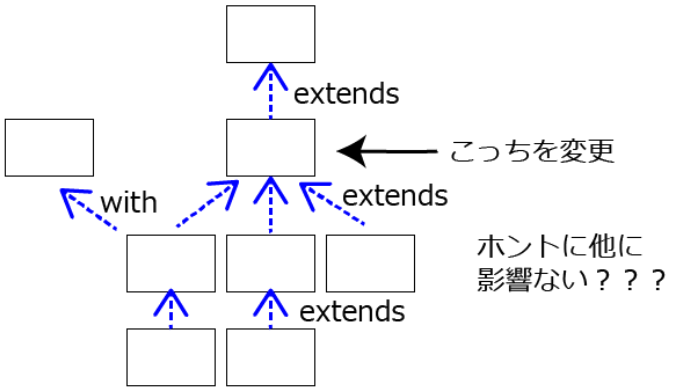
第8章 オープン・クローズドの原則

読んでると長くなりそう。結論だけ述べるにとどめます。

継承を考えた時Baseクラスを変更したときDerivedクラスは変更しなくても良いですよ、Derivedクラスを変更するときはBaseクラスを変更しなくても良いですよということ。そしてDerivedクラスはBaseクラスを変更することなく新しいクラスを追加できますよということ。



反対に言うとこの原則の目的が達成できなくなる、変更に際して影響範囲の調査が困難な場合は継承の仕組みをつくりなおす、あるいは継承自体やめることも必要かもしれません。



こういう複雑な継承・実装の関係があるときに、「オープン・クローズドの原則を満たすためにメソッドを移動してリファクタリング！」って繰り返してたら、「それを繰り返すってことはオープン・クローズドになってないじゃん」っていうやつ。

参考:

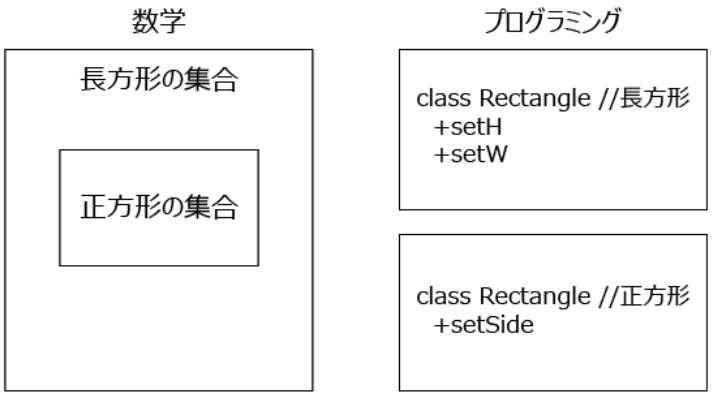
- 実況中継シリーズ「開発現場で役立たせるための設計原則とパターン」#builderscon 2018

第9章 リスコフの置換原則

継承を使うときは、常にBaseクラスとして使われることを考えようということ。継承の階層が深くなると問題が表面化しやすい。

> 正方形・長方形問題

正方形・長方形問題は数学的な部分集合で表せることと、ソフトウェアの目的として何をしたいかを混同した例のようです。DDD読書会で出てきた「モデルは目的に合わせて選ぶ」(例: ピザの分割モデルと3Dワイヤフレームモデルの話など)ことを思い出すと良いでしょう。ソフトウェアの目的に合わないなら数学的にはきれいに見える表現も、目的を達成するモデルにはなりません。



> リスコフの置換原則(LSP)とアーキテクチャ

Baseクラスだけじゃなくてインターフェースの場合も同様。これも継承が深くなると問題になりやすい。

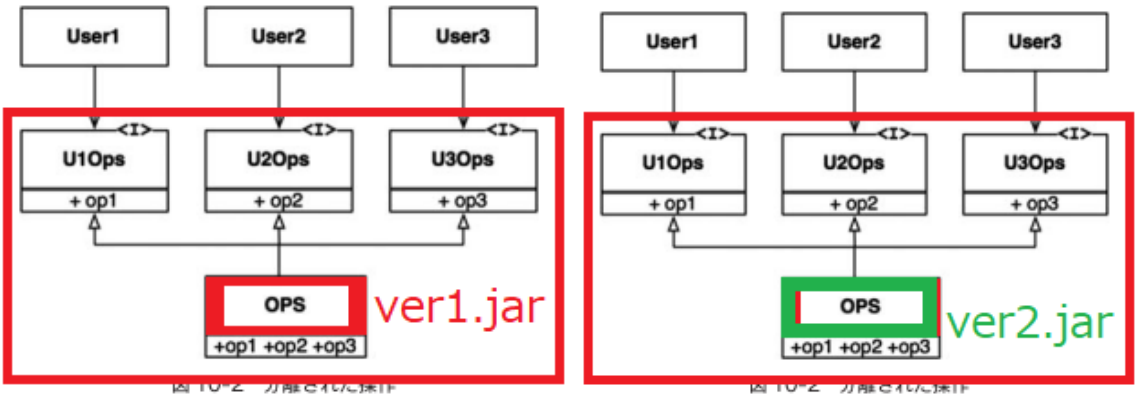
> リスコフの置換原則（LSP）違反の例

よみとばす。タクシー配車サービスの話はあまりいい例ではない。個別の商用サービスが全て従うべき「業界標準タクシー配車API」というものは普通存在しない。普通にJavaとかのInterfaceとそれをimplementしたクラスの一例をもう一つ出したほうが良かった。

第10章 インターフェース分離の法則

> (最初) … インターフェイス分離の原則（ISP）と言語との関係

User1 ~ 3を含むソースコードをver1.jarとリンクしてコンパイル
-> user.jarができる
-> ver1.jarの新しいバージョンver2.jarがリリースされる
-> user.jarは何も変えることなく、ランタイムのリンケージをver2.jarに差し替えることができる



interfaceはver1.jarに含まれる(user.jarに含まれない)にもかかわらず、user.jarの再コンパイルが必要ないことに注目してください。

しかしこれはver1.jar -> ver2.jarでinterfaceであるU1Ops ~ U3Opsが変更されなかった時の話で、interfaceが変更されてしまうとbinary compatibility問題が生じる可能性があることに留意してください。
https://twitter.com/RichardImaokaJP/status/1151284445816614912

第11章 依存性逆転の原則

ボブおじ大好きなDIの話。本書の先の部分でもDI激推しでしたね。

> 依存したくないのは、システム内の変化しやすい具象要素だ。開発中のモジュールや、頻繁に変更され続けているモジュールがその対象になる。

冒頭は全部読まなくてよい。↑の部分だけが大事そう。

> 安定した抽象

読む

> Factory

よみとばす。これは各自調べてくれで良さそう。