リファクタリング
オブジェクト指向

オブジェクト指向

データと振る舞いをセットでおこう

歴史を見てみよう

背景 [編集]

オブジェクト指向プログラミングという考え方が生まれた背景には、計算機の性能向上によって従来より大規模なソフトウェアが書かれるようになってきたということが挙げられる。大規模なソフトウェアが書かれコードも複雑化してゆくにつれ、ソフトウェア開発コストが上昇し、1960年代には「ソフトウェア危機 (software crisis)」といったようなことも危惧されるようになってきた。そこでソフトウェアの再利用、部品化といったようなことを意識した仕組みの開発や、ソフトウェア開発工程の体系化（**ソフトウェア工学** (software engineering) の誕生）などが行われるようになった。

このような流れの中で、プログラムを構成するコードとデータのうちコードについては手続きや関数といった仕組みを基礎に整理され、その構成単位をブラックボックスとすることで再利用性を向上し、部品化を推進する仕組みが提唱され**構造化プログラミング** (structured programming) として1967年にエドガー・ダイクストラ (Edsger Wybe Dijkstra) らによってまとめあげられた（プログラミング言語の例としてはPascal 1971年）。なお、それに続けて「しかしデータについては相変わらず主記憶上の記憶場所に置かれている限られた種類の基本データ型の値という比較的低レベルの抽象化から抜け出せなかった。これはコードはそれ自身で意味的なまとまりを持つがデータはそれを処理するコードと組み合わせないと十分に意味が表現できないという性質があるためであった。」といったように、ほぼ間違いなく説明されている。

そこでデータを構造化し、ブラックボックス化するために考え出されたのが、データ形式の定義とそれを処理する手続きや関数をまとめて一個の構成単位とするという考え方で**モジュール** (module) と呼ばれる概念である（プログラミング言語の例としてはModula-2 1979年）。しかし定義とプログラム内の実体が一対一に対応する手続きや関数とは異なり、データはその形式の定義に対して値となる実体（インスタンスと呼ばれる）が複数存在し、各々様々な寿命を持つのが通例であるため、そのような複数の実体をうまく管理する枠組みも必要であることがわかってきた。そこで単なるモジュールではなく、それらのインスタンスを整理して管理する仕組み（例えばクラスとその継承など）まで考慮して生まれたのがオブジェクトという概念である（プログラミング言語の例としてはSimula 1962年）

歴史を見てみよう

ソフトウェア危機(1960くらい)

マシンが強力になって

プロダクトも複雑になっていくけど

コードが全然整理できない！開発が困難。

ソフトウェア開発の進化

- 上から順に、if, forのみで制御
- 構造体でデータをまとめて整理

歴史を見てみよう

振る舞い

```
処理1()
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
}
```

Userのデータが整理された、Happy

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
  user.firstName + user.lastName
}

処理2()
   yyy
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

同じ処理増えた。

Code Smell
duplicate code !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
   fullName()
}
処理2()
   yyy
   fullName()
}
fullName(){
   user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

関数化！

remove duplicate !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
  fullName()
}
処理2()
  yyy
  fullName()
}
fullName(){
  user.firstName + user.lastName
}
```

ファイルB

```
処理3()
  xxx
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

同じ処理増えた。

Code Smell
duplicate code !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
  fullName()
}
処理2()
  yyy
  fullName()
}
```

ファイルB

```
処理3()
  zzz
  fullName()
}
```

ファイルC(Userの振る舞い)

```
fullName()
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

外部ファイルに関数化

remove duplicate !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
  fullName()
}
処理2()
  yyy
  fullName()
}
```

ファイルB

```
処理3()
  zzz
  fullName()
}
```

ファイルC(Userの振る舞い)

```
fullName()
  user.firstName + user.lastName
}
```

ファイルD

```
処理D()
  YYY
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

その関数あるの知らなかった！

Code Smell
duplicate code !!

歴史を見てみよう

データと振る舞いがそばにいれば

整理しやすいのではないか

クラス

- 構造体（データ）
- クラス(データと振る舞い）（フィールドとメソッド）

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
  fullName()
}
処理2()
  yyy
  fullName()
}
```

ファイルB

```
処理3()
  zzz
  fullName()
}
```

ファイルC(Userの振る舞い)

```
fullName()
  user.firstName + user.lastName
}
```

ファイルD

```
処理D()
  YYY
  user.firstName + user.lastName
}
```

構造体（データ）

```
struct User {
  firstName string
  lastName string
  }
```

その関数あるの知らなかった！

Code Smell
duplicate code !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
   user.fullName()
}

処理2()
   yyy
   user.fullName()
}
```

ファイルB

```
処理3()
   zzz
   user.fullName()
}
```

ファイルD

```
処理D()
   YYY
   user.fullName()
}
```

クラス（データと振る舞い）

```
class User {
   firstName string
   lastName string
   fullName() {
      user.firstName + user.lastName
   }
}
```

クラス化。メソッド化。
データを使う人が振る舞いを
意識しやすい

remove duplicate !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
   user.fullName()
}

処理2()
   yyy
   user.fullName()
}
```

ファイルB

```
処理3()
   zzz
   user.fullName()
}

処理E()
   ZZZ
   user.firstName() + user.lastName()
```

ファイルD

```
処理D()
   YYY
   user.fullName()
}
```

クラス（データと振る舞い）

```
class User {
   firstName string
   lastName string
   fullName() {
      user.firstName + user.lastName
   }
}
```

それでも知らなかった！

Code Smell
duplicate code !!

歴史を見てみよう

振る舞い

ファイルA

```
処理1()
  xxx
   user.fullName()
}

処理2()
   yyy
   user.fullName()
}
```

ファイルB

```
処理3()
   zzz
   user.fullName()
}

処理E()
   ZZZ
   user.fullName()
```

ファイルD

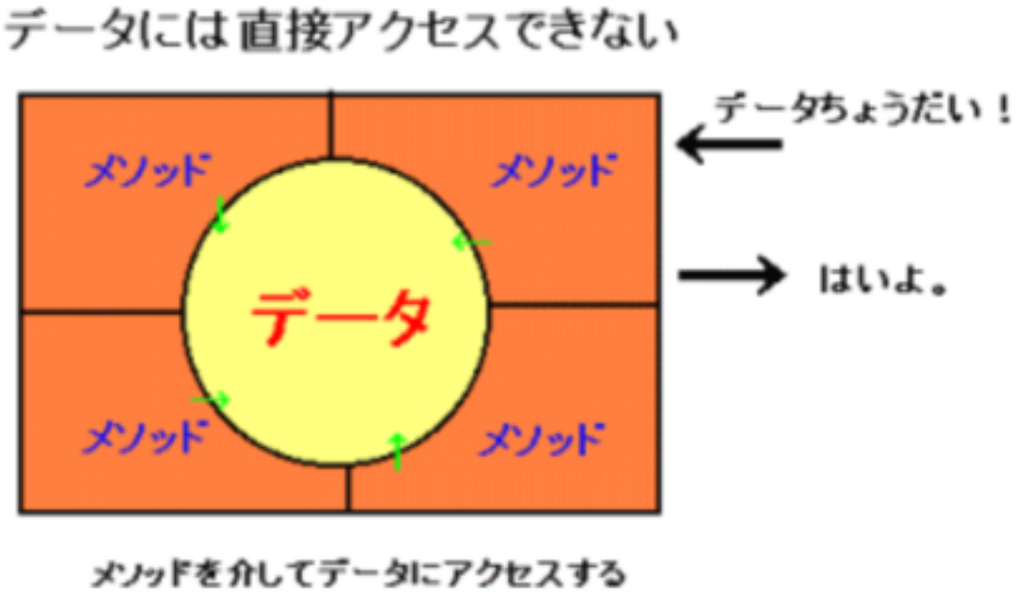
```
処理D()
   YYY
   user.fullName()
}
```

クラス（データと振る舞い）

```
class User {
   private firstName string
   private lastName string
  fullName() {
     user.firstName + user.lastName
   }
}
```

カプセル化。
必要な情報のみ開示。
remove duplicate !!

データと振る舞いはセット



重複コードを排除につながる、コードの整理

＋

人が理解しやすい

見てみよう、試してみよう

CODE SMELL

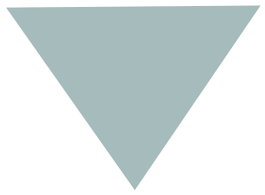
Feature Envy

あるべき場所から流出している！

DataClass（pure構造体）

子供オブジェクト

ここにあるべきものが流出している！



ソリューション：データと振る舞いはセット

参考：ドメインモデル貧血症

Martin Fowler's Bliki (ja)　　タグ　人気エントリー　PofEAA

ドメインモデル貧血症

Nov 25, 2003 [original] [source]

甲斐があります。 とはいえ、すべての振る舞いをサービスに押し込むと、どうしても トランザクションスクリプト になってしまいます。これでは、ドメインモデルのもたらすメリットを失います。

多くのOOエキスパートたちが、処理を行うレイヤをドメインモデルの一番上に置いて、サービスレイヤを作るよう推奨していることが、混乱のそもそもの原因です。 ただしこれは、振る舞いのないドメインモデルを作るということではありません。 そうではなくて、サービスレイヤの支持者は、振る舞いをたくさん含んだドメインモデルと一緒に使っています。

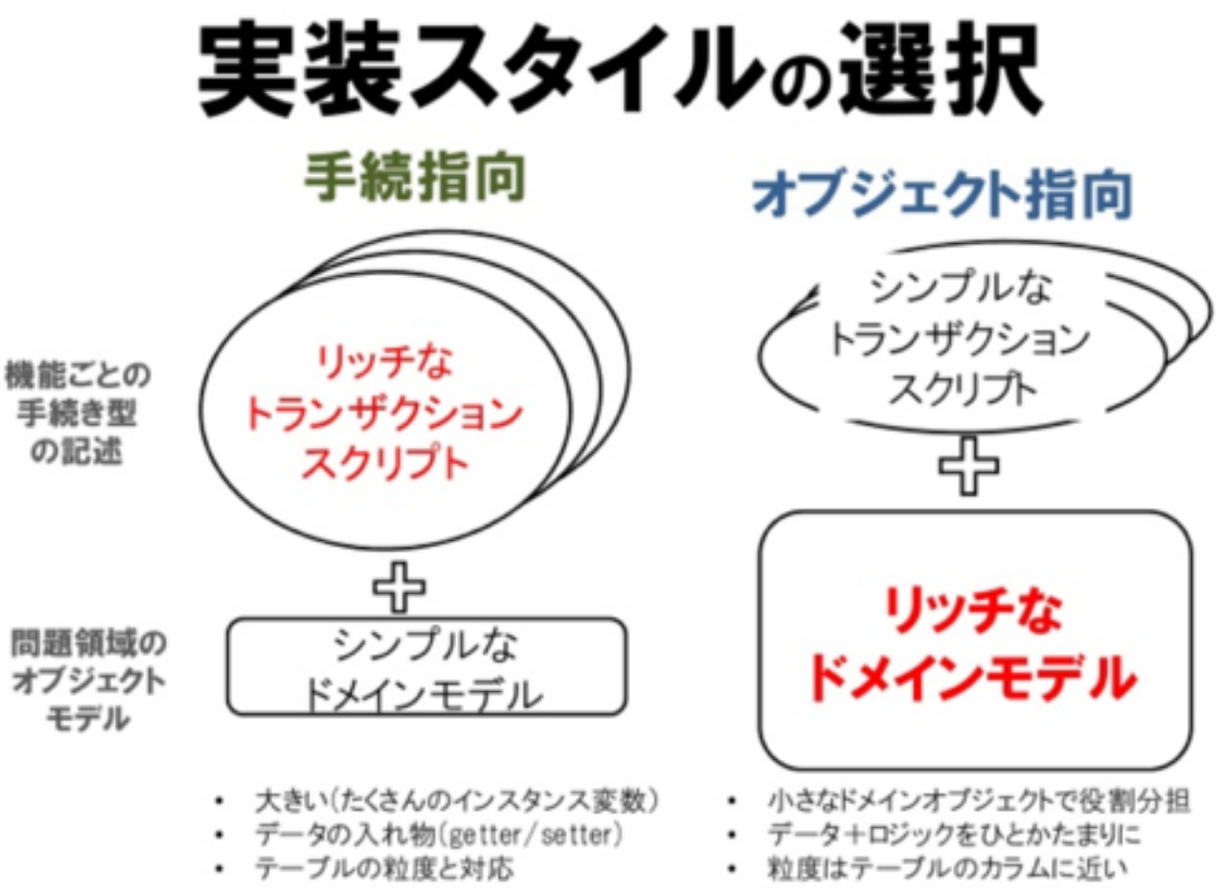
ここでのキーポイントは、サービスレイヤは薄い、という点です（キーとなるロジックはドメインレイヤに置かれています）。彼はこの点についてサービスパターンの中で何度も言及しています。

> *現在よくある過ちは、適切なオブジェクトに振る舞いを割り当てることを、あまりにも簡単に諦めてしまっていることです。徐々に手続き型プログラミングになっているのです。*

なぜこのアンチパターンが一般的になっているのかよく分かりません。 真のドメインモデルをちゃんと使っていないのが原因ではないかと思っています（特にデータ系からきた人たちが）。 それを助長している技術にも原因があります（J2EEのEntity Beanとか。だから私は POJO ドメインモデルが好きなのです）。

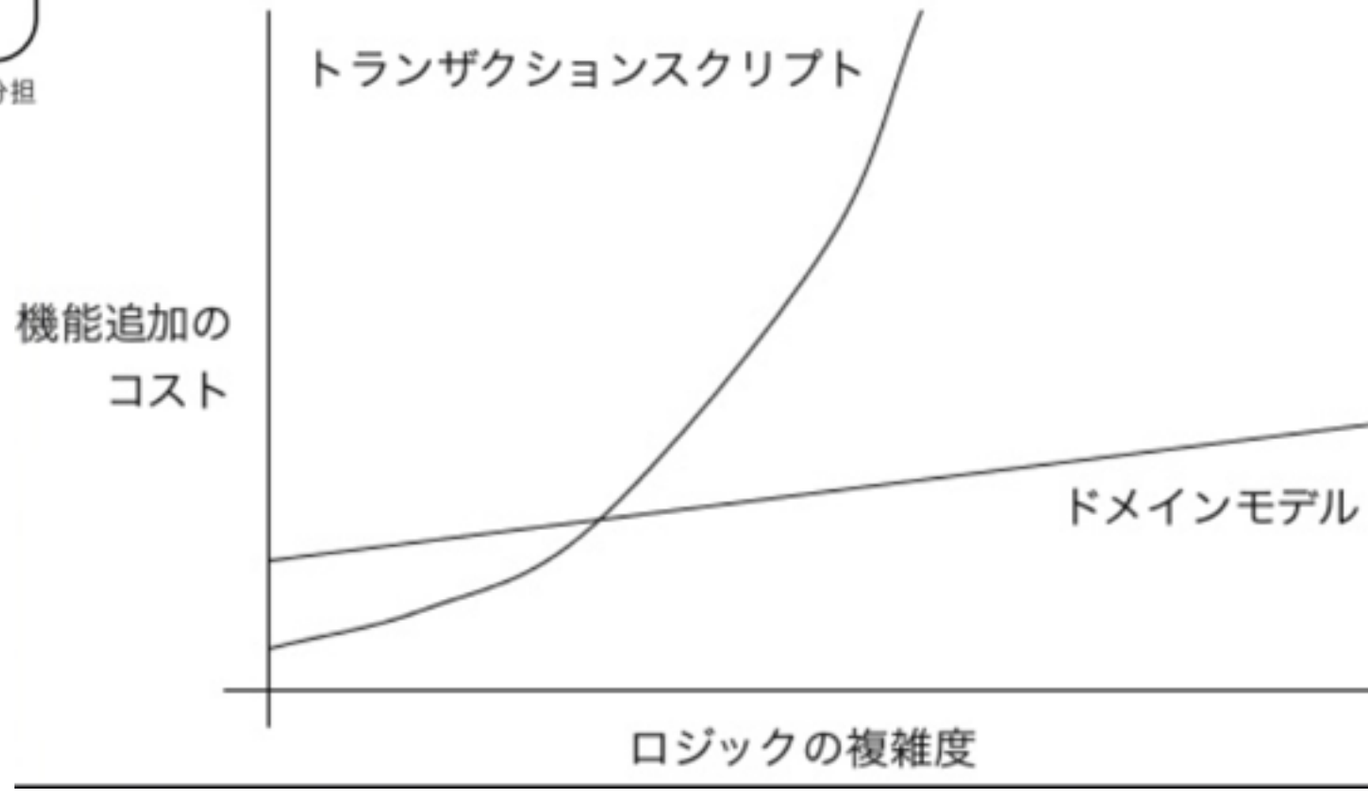
サービスの中に振る舞いを見つければ見つけるほど、ドメインモデルのメリットを奪っていくでしょう。サービスの中にすべてのロジックを埋めてしまうと、何も見えなくなってしまいます。

参考：トランザクションスクリプトとドメインモデル



※ https://www.slideshare.net/masuda220/rdra-ddd-agile

※PoEAA

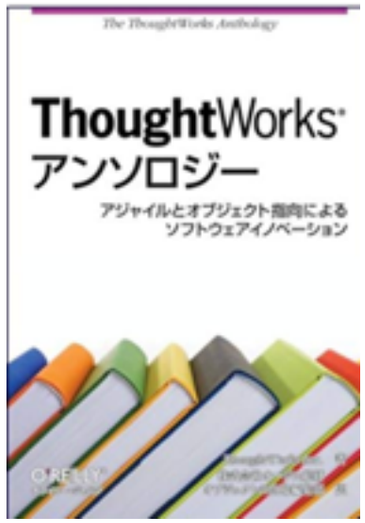


参考書籍

1999

2002

2003



2008

2008