ClearCode

リーダブルコードを読み解こう

7章 制御フローを読みやすくする

須藤功平

株式会社クリアコード

schoo

2015-04-03

質問（1）

前回の授業は…

- ✓ A：参加した
- ✓ B：参加できなかった
  （都合が合わなかった、知らなかったなど）
- ✓ C：知っていたが参加していない

質問（2）

プログラミングについて

- ✓ A：未経験
- ✓ B：学習中（schooや学校、独学など）
- ✓ C：趣味・仕事でたまに書く
  （趣味でWebサイトを作っている、職業がデザイナーなど）
- ✓ D：趣味・仕事でバリバリ書く
  （趣味でOSSを開発している、職業がエンジニアなど）

質問（3）

リーダブルコード（本）を…

- ✓ A：読んだ
- ✓ B：読んでいる
- ✓ C：まだ読んでいない

内容

- ✓ 自己紹介
- ✓ リーダブルコードとは
- ✓ 実例で考えよう
- ✓ 実際の改善にチャレンジ！
- ✓ まとめ
- ✓ 質疑応答

自己紹介（1）

- ✓ リーダブルコードの「解説」の著者
  http://www.clear-code.com/blog/2012/6/11.html

自己紹介（2）

- クリアコードの代表取締役
  - 「クリア」な（意図が明確な）
    「コード」を大事にする
    ソフトウェア開発会社

自己紹介（3）

Kouhei Sutou
kou

ClearCode Inc.
Tokyo, Japan
kou@clear-code.com
http://www.clear-code.com/
Joined on 3 Oct 2008

127	55	0
Followers	Starred	Following

Organizations

Contributions | Repositories | Public activity | Edit profile

Popular repositories

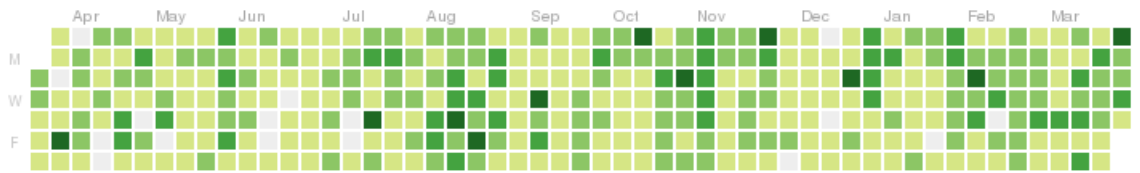
- bundle-milkode — Make all gems installed by Bundler milkable — 7★
- segv-handler-gdb — Dump C level backtrace by GDB on SEGV — 5★
- mruby-pp — pp for mruby — 4★
- misawa — TODO: one-line summary of your gem — 3★
- rubyinstaller — R… for Windows - Building recipes — 5★

Repositories contributed to

- groonga/groonga — An embeddable fulltext search engine. Groon… — 250★
- ruby-gnome2/ruby-gnome2 — A set of bindings for the GNOME-2.x libraries … — 116★
- mroonga/mroonga — A MySQL pluggable storage engine based o… — 84★
- project-hatohol/hatohol — A unified manager of monitoring software — 60★
- droonga/droonga-engine — …e engine … a core component in Droo… — 5★

毎日コードを書いている

Contributions



Summary of Pull Requests, issues opened, and commits. Learn more.
Less More

Contributions in the last year	Longest streak	Current streak
7,322 total	147 days	42 days
Mar 25, 2014 – Mar 25, 2015	July 12 – December 5	February 13 – March 26

リーダブルコードとは（1）

> 本書の目的は、君のコードをよくすることだ
>
> ［「はじめに p. x」より引用］

リーダブルコードとは（2）

> その中心となるのは、コードは理解しやすくなければいけないという考えだ
>
> ［「はじめに p. x」より引用］

リーダブルコードとは（3）

> 「コードを理解する」というのは、変更を加えたりバグを見つけたりできるという意味
>
> ［「1.2 読みやすさの基本定理 p. 3」より引用］

リーダブルコード

読む人が…

- ✓ 変更できるコード
- ✓ バグを見つけられるコード

↓

読む人視点！

何をしているコード？

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

「優れた」コードって何？ p. 2より

何をしているコード？

```
for (Node* node = list->head;
     node != NULL;
     node = node->next)
    Print(node->data);
```

「優れた」コードって何？ p. 2より

どちらがリーダブル？

```
// どちらがリーダブルコード？どうして？
// リーダブルコード：変更できる・バグを見つけられるコード
// A                                    // B
Node* node = list->head;                for (Node* node = list->head;
if (node == NULL) return;                    node != NULL;
while (node->next != NULL) {                 node = node->next)
    Print(node->data);                      Print(node->data);
    node = node->next;
}
if (node != NULL)
    Print(node->data);
```

「優れた」コードって何？ p. 2より

実例で考えよう

7章
「制御フローを読みやすくする」
より

7.1 例：式の並び順（1）

```
/* A */
if (length >= 10)
/* B */
if (10 <= length)
```

どちらがリーダブル？

7.1 例：式の並び順（2）

```
/* A */
while (bytes_received < bytes_expected)
/* B */
while (bytes_expected > bytes_received)
```

どちらがリーダブル？

7.1 式の並び順の指針

- 左側
  - 「調査対象」の式。変化する。
  - `length, bytes_received`
- 右側
  - 「比較対象」の式。変化しにくい。
  - `10, bytes_expected`

7.1 指針に沿った並び順

```
/*   ↓調査対象。変化する。  */
if (length >= 10)
/*              ↑比較対象。変化しない。  */
/*      ↓調査対象 */
while (bytes_received < bytes_expected)
/*                          ↑比較対象 */
```

7.1 指針の理由

自然言語の並び順に近い

```
/* もし長さが10以上なら */
if (length >= 10)

/* もし10が長さ以下なら */
if (10 <= length)
```

番外：別の指針

- 左側
  - 小さい値
- 右側
  - 大きい値

左から右にいくほど
大きくなることは自然
（例：数直線）

番外：指針に沿った並び順

使う比較演算子は「<」と「<=」

```
if (10 <= length)

while (bytes_received < bytes_expected)
```

7.1：まとめ

- 条件式内の並び順を工夫するとリーダブル度があがる
- 指針：
  - 左側：変化する式
  - 右側：変化しにくい式

7.2 例：処理の順番（1）

```
/*      どちらがリーダブル？      */
/* A */                  | /* B */
if (a == b) {             | if (a != b) {
  /* ケース1 */           |   /* ケース2 */
} else {                  | } else {
  /* ケース2 */           |   /* ケース1 */
}                         | }
```

7.2 例：処理の順番（1）

```
/*     A：条件が肯定形だから     */
/* A */                  | /* B */
if (a == b) {            | if (a != b) {
  /* ケース1 */          |   /* ケース2 */
} else {                 | } else {
  /* ケース2 */          |   /* ケース1 */
}                        | }
```

7.2 例：処理の順番（2）

```
/*      どちらがリーダブル？      */
/*  （ケース1の方が処理が長い）  */
/* A */                 | /* B */
if (a == b) {           | if (a != b) {
  /* ケース1 */         |   /* ケース2 */
  /*   ...   */         |  } else {
  /*   ...   */         |   /* ケース1 */
} else {                |   /*   ...   */
  /* ケース2 */         |   /*   ...   */
}                       | }
```

7.2 例：処理の順番（2）

```
/*   B：単純な処理の条件が先    */
/*    if/elseを一望しやすい     */
/* A */              | /* B */
if (a == b) {       | if (a != b) {
  /* ケース1 */     |   /* ケース2 */
  /*   ...   */     |  } else {
  /*   ...   */     |   /* ケース1 */
} else {            |   /*   ...   */
  /* ケース2 */     |   /*   ...   */
}                   | }
```

7.2 指針

- 否定形は肯定形にして書く
  - `if (!debug)`より`if (debug)`
- 単純な処理の条件を先に書く
  - `if`と`else`を一望できて読みやすい
- 関心を引く条件を先に書く
- 目立つ条件を先に書く

7.2 指針：注意

- ✓ 同時に満たせないことがある
- ✓ 自分で優劣を判断すること
- ✓ 優劣は明確になることが多い

7.2 判断してみよう

```
if (!url.HasQueryParameter("expand_all")) {
  response.Render(items);
  // ...
} else {
  for (int i = 0; i < items.size(); i++) {
    items[i].Expand();
  }
  // ...
}
```

7.2 !を外して順番を逆に

```
// 関心を引く条件「expand_all」を優先
//  （肯定形になったのはおまけ）
if (url.HasQueryParameter("expand_all")) {
  for (int i = 0; i < items.size(); i++) {
    items[i].Expand();
  }
  // ...
} else {
  response.Render(items);
  // ...
}
```

7.2 まとめ

- 処理する条件の順番次第でリーダブル度があがる
- 指針：
  - 否定形は肯定形にして書く
  - 単純な条件を先に書く
  - 関心がある・目立つ条件を先に書く

7.5 関数から早く返す

「ガード節」

```
public boolean Contains(String str, String substr) {
  if (str == null || substr == null) // ガード節
    return false;
  if (substr.equals(""))             // ガード節
    return true;
  // 大事な処理だけに興味があるなら↑までは無視可能。
  // 大事な処理は↓から。
  // ...
}
```

7.5 ガード節なし

```
public boolean Contains(String str, String substr) {
  if (str == null || substr == null) {
    return false; // 異常値
  } else {
    if (substr.equals("")) {
      return true; // 特別値
    } else {
      // 大事な処理はここ。ネストが深い。
      // ...
    }
  }
}
```

7.5 ガード節の効果

- ✓ 以降の処理が単純になる
  - ✓ 異常値・特別値を考慮しなくてよい
  - ✓ →考える事が減る
- ✓ ネストが浅くなる
  - ✓ →コードの見通しがよくなる

リーダブルコードにつながる！

番外：ifとreturnの使い方

ifとreturnの使い方
ククログ(2012-03-28)
http://www.clear-code.com/blog/2012/3/28.html

番外：パス

同じ処理でも流れ（パス）は違う

```
if 条件          | return if !条件
  サブ処理       |
end              | サブ処理
```

番外：パスとリーダブル

- パス
  - 処理を実行するときの流れ
  - コードを読むときの流れでもある
- 読みやすい流れ
  - →リーダブル！

番外：例（1）

```
def add_comment(post, user, text)
  if post.hidden_from?(user)
    report_access_error
  else
    comment = Comment.create(text)
    post.add(comment)
  end
end
```

番外：例（1）コメント付き

```
# コメントを追加するメソッドだな！
def add_comment(post, user, text)
  if post.hidden_from?(user) # ユーザーに見えない投稿なら…
    report_access_error
    # エラー処理か。ifの後にもなにか処理はあるのかな？
  else
    comment = Comment.create(text)
    post.add(comment)
    # ユーザーに見えるときだけコメント追加か。
    # こうやってコメントを追加するんだな。
  end # ifの後にはなにも処理はないんだな。
end
```

番外：例（1）return付き

```
# コメントを追加するメソッドだな！
def add_comment(post, user, text)
  if post.hidden_from?(user) # ユーザーに見えない投稿なら…
    report_access_error # エラー処理をして、
    return # それで終了か。
  end

  # エラー処理が終わったからここからは通常の処理だな。
  comment = Comment.create(text)
  post.add(comment) # こうやってコメントを追加するんだな！
end
```

番外：例（1）まとめ

- ✓ 異常系と正常系の扱いを変える
- ✓ 異常系はすぐにreturn
  - ✓ エラー処理だけというのをアピール
- ✓ 正常系はネストしない
  - ✓ 正常系の方が重要なコード
  - ✓ 重要なコードは見やすい場所に

番外：例（2）

```
def prepare_database(path)
  if not File.exist?(path)
    return Database.create(path)
  end

  Database.open(path)
end
```

番外：例（2）コメント付き

```
# データベースを準備するんだな
def prepare_database(path)
  if not File.exist?(path) # なかったら…
    # 作ってすぐに終了か。あれ、作るのも普通の「準備」なような…
    return Database.create(path)
  end

  # あったら開くのか。これも「準備」だよなぁ。
  # なにか特別な意図があるんだろうか…
  Database.open(path)
end
```

番外：例（2）returnなし

```
# データベースを準備するんだな
def prepare_database(path)
  if File.exist?(path) # あったら…
    Database.open(path) # 開いて
  else                          # なかったら…
    Database.create(path) # 作るのか
  end
end
```

番外：例（2）まとめ

- ✓ 正常系同士の扱いを変えない
  - ✓ 変えると意図があると勘違いする
  - ✓ 「Aの処理は何か特別なのかも…」
- ✓ なんでもガード節にしない
  - ✓ 特別なケースや異常系のときだけ

他の例

- ✓ 三項演算子
- ✓ ネストを浅くする
- ✓ …
- ✓ （詳細は本を買ってください）

実際の改善にチャレンジ！

```
// 読みやすい制御フローにして投稿・よい投稿に「いいね！」して応援

function positiveIntegerSum(n) { // n以下の正の整数の合計を返す
  if (0 >= n) {                  // ヒント:
    return -1; // エラー         // 1. 条件式の並び順
  } else {                       //    * 左に変化するもの
    var total = 0;               //    * 右に変化しにくいもの
    for (; n > 0; n--) {         // 2. 処理する条件の順番
      total += n;                //    * 肯定形で書く
    }                            //    * 単純な条件を先に書く
    return total;                //    * 関心がある条件を先に書く
  }                              //    * 目立つ条件を先に書く
}                                // 3. ガード節
```

まとめ（1）

- ✓ リーダブルコードとは
  - ✓ 変更できるコード
  - ✓ バグを見つけられるコード
  - ✓ ↑は**読む人視点**

まとめ（2）

- 「読みやすい制御フロー」を考えた
  - 条件式内の並び順
    - →左に変化するもの・右に変化しにくいもの
  - 処理する条件の順番
    - →肯定形で書く・単純な条件を先に書くなど
  - ガード節
    - →特別なケースはすぐにreturn

まとめ（3）

- ✓ 実際の改善にチャレンジした
  - ✓ 「**読む人**が理解しやすいか？」をとことん考えたはず

> *行数を短くするよりも、他の人が理解するのにかかる時間を短くする。*
>
> *［「7章 制御フローを読みやすくする p. 89」より引用］*

これから（1）

- これからも**読む人**のことを考えてコードを書こう
- **読む人**のことを考えるには？
  - 読む経験をたくさん積む
  - たくさんコードを読もう

これから（2）

- たくさんコードを読むコツ
  - コードから学ぶ気持ちで読む
  - ×悪いこと探し
  - ○いいこと探し
- 読むコード
  - オススメは自分が使っているOSS
  - ↑動作を知っているから読みやすい

悪いコード

- ✓ 見つけやすい
  - ✓ 異質
  - ✓ リーダブルじゃない

よいコード

- 見つけにくい
  - リーダブルだから
  - すーっと理解できてひっかからない
- これからのチャレンジ
  - 意識して見つけよう！

これから（3）

「解説」を読む

http://www.clear-code.com/blog/2012/6/11.html

- ✓ 本文：**個人**で
  リーダブルコードを書く方法
- ✓ 解説：**チーム**で
  リーダブルコードを書く方法