

この5年間で「ひどいコード」をたくさん集めてきた（その多くは自分たちのものだけ）。そして、なぜそのコードがひどいのか、それを改善するにはどんな原則や技法が使えるのか調べてみた。そこからわかったのは、すべての原則はたった1つのテーマから生じているということだ。

鍵となる考え方

コードは理解しやすくなればいい。

これがコードを書く上でいちばん大切な原則だ。これからこの原則を目々のコーディングのさまざまな場面に当たる方法を紹介していきたいと思う。でもその前に、この原則のことをもっと詳しく説明しよう。どうしてこの原則がそんなに大切なだろうか。

1.1 「優れた」コードって何？

プログラマというのは（ほくたちもそうだけど）、何となく直感でプログラミングのことを決めていることが多い。例えば、このようなコードのほうが、

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

以下のコードよりも優れているとみんな思っている。

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

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

（どちらも動作は全く同じなのに）

でも、どちらが優れているかわからないことが多い。例えば、このようなコードは、

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

以下のコードよりも優れているのだろうか。

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

前者のほうが簡潔だ。でも、後者のほうが安心できる。「簡潔」と「安心」はどちらが大切なことなんだろう？ 君ならどうやって決めるだろうか？

1.2 読みやすさの基本定理

さつき見たようなコードをたくさん調査してみたら、読みやすさの基準となるものにたどり着くことができた。これはどんな基準よりも大切なものなんだ。すごく大切な「読みやすさの基本定理」と呼ぶことにしよう。

鍵となる考え方

コードは他の人が最短時間で理解できるように書かなければいけない。

これってどういう意味だと思う？ そのまんまの意味だ。例えば、同僚にコードを読んでもらって、彼が理解するまでにかかる時間を計測するとしよう。この「理解するまでにかかる時間」という数値を最短にするってことだ。

それから「理解する」っていう言葉には、高いハードルを設けてある。「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だ。他のコードと連携する方法も理解しておかなければいけない。

もしかすると、こんなふうに考えているかもしれないね。「他の人が理解できるって、誰が得するんだよ？ このコードを使ってるのはオレだけなんだぞ！」 でもね、たとえ君ひとりのプロジェクトだったとしても、この目標には取り組むだけの価値があるんだ。「他の人」というのは、自分のコードに見覚えのない6か月後の「君自身」かもしれない。君のプロジェクトに途中から誰かが参加しないとも言い切れない。「使い捨てのコード」が他のプロジェクトで再利用される可能性だってある。

1.3 小さなことは絶対にいいこと？

問題を解決するコードは短いほうがいい（「13章 短いコードを書く」参照）。2,000行のクラスのほうが5,000行のクラスよりも理解するまでにかかる時間は短いはずだ。

でも、短ければいいってもんじゃない！ このような1行のコードは、

```
assert((!bucket = FindBucket(key))) || !bucket->IsOccupied());
```

以下のような2行のコードを理解するよりも時間がかかることが多い。

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

コメントをつけると「コードが長く」なるけど、そのほうが理解しやすいこともある。

```
// "hash = (65599 * hash) + c" の高速版
hash = (hash << 6) + (hash << 16) - hash + c;
```

コードは短くしたほうがいい。だけど、「理解するまでにかかる時間」を短くするほうが大切だ。

1.4 「理解するまでにかかる時間」は競合する？

「それじゃあ、他の条件は？ コードを効率化するとか、設計をうまくやるとか、テストしやすいとか、いろいろあるじゃん？ そういうのは理解しやすさと競合しないわけ？」そんなことを考えるかもしれないね。

でも、他の目標とは全く競合しないんだ。高度に最適化されたコードであっても、もっと理解しやすくできるはずだ。それに、理解しやすいコードというのは、優れた設計やテストのしやすさにつながることが多い。

本書では、いろんな状況に「読みやすさ」を当てはめる方法を紹介している。どうすればいいかわからなくなったときは、本書で紹介する規則や原則よりも「読みやすさの基本定理」を最優先に考えて欲しい。コードを見たらすぐにリファクタリングしあくなるプログラマもいるだろうけど、常に一歩下がって「このコードは理解しやすいだろうか？」と自問自答してみることが大切だ。理解しやすいコードになってか

ら、次のコードを書き始めてもいいんじゃないかな。

1.5 でもやるんだよ

想像上の誰かが自分のコードを理解しやすいかなんて考えるのは大変なことだ。これまでとは違う脳ミソを回転させなきゃいけない。

でも、この目標を（ぼくたちみたいに）受け入れたら、君はきっと優秀なプログラマーになれるはずだ。自分の仕事に誇りを持ち、周囲のみんなが喜んで使ってくれるような、バグの少ないコードを作り出せるようになる。さあ、始めよう！

名前をつけるときには、それが変数であっても、関数であっても、クラスであっても、同じ原則を当てはめることができる。名前は短いコメントだと思えばいい。短くてもいい名前をつければ、それだけ多くの情報を伝えることができる。

鍵となる考え方

名前に情報を詰め込む。

プログラムに使われる名前というのはハッキリしないものが多い。例えば、`tmp`なんかがそうだ。でも、`size`や`get`みたいに一見すると問題がなさそうな名前であっても、情報が含まれていないことがある。

これから情報を詰め込んだ名前のつけ方を紹介する。本章は、以下の6つのテーマで構成されている。

- 明確な単語を選ぶ
- 汎用的な名前を避ける（あるいは、使う状況を選ぶ）
- 抽象的な名前よりも具体的な名前を使う
- 接尾辞や接頭辞を使って情報を追加する
- 名前の長さを決める
- 名前のフォーマットで情報を伝える

2.1 明確な単語を選ぶ

「名前に情報を詰め込む」には、明確な単語を選ばなければいけない。「空虚な」単語は避けるべきだ。

例えば、「`get`」はあまり明確な単語ではない。

```
def GetPage(url):
    ...
```

「`get`」という単語からは何も伝わってこない。このメソッドはページをどこから取ってくるのだろう？ ローカルキャッシュから？ データベースから？ インターネットから？ インターネットから取ってくるのであれば、`FetchPage()`や

`DownloadPage()` のほうが明確だ。

次は、`BinaryTree` クラスの例だ。

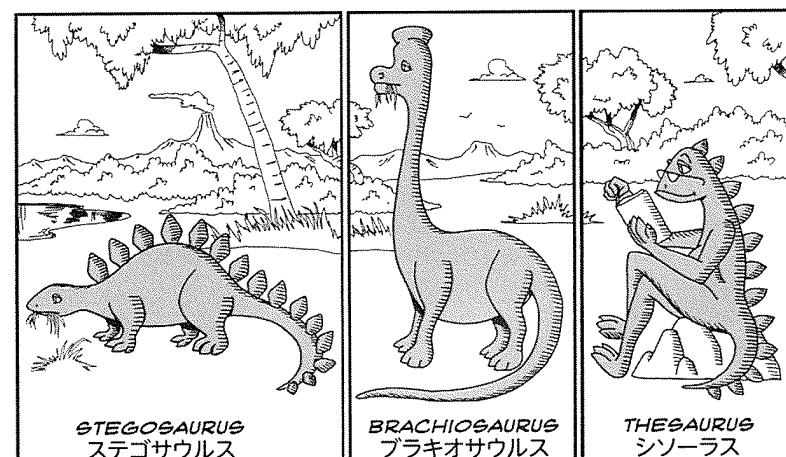
```
class BinaryTree {
    int Size();
    ...
};
```

この `Size()` メソッドは何を返すのだろうか？ ツリーの高さ？ ノードの数？ ツリーのメモリ消費量？ `Size()` では何も情報が伝わらない。目的に適した明確な名前をつけるなら、それぞれ `Height()`・`NumNodes()`・`MemoryBytes()` になるだろう。もう1つの例は、とある `Thread` クラスだ。

```
class Thread {
    void Stop();
    ...
};
```

`Stop()` という名前でもいいけど、動作に合わせてもっと明確な名前をつけたほうがいいと思う。例えば、取り消しができない重い操作なら、`Kill()` にするといい。あとから `Resume()` できるなら、`Pause()` にしてもいいかもしれない。

もっと「カラフル」な単語を探す



シソーラス（類語辞典）を使って調べよう。友達にもっといい名前がないかと聞いてみよう。英語は豊かな言語だから、選べる単語はたくさんあるはずだ。

例として、いくつかの単語と状況に合わせて使えるもっと「カラフル」なバージョンの単語を紹介しておこう。

単語	代替案
send	deliver, dispatch, announce, distribute, route
find	search, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

ただし、やりすぎはいけない。PHPには、文字列を `explode()` する関数がある。カラフルな名前だし、何かを分割する様子がうまく表現できている。だけど、`split()` と何が違うんだろう？（2つの関数は別物だけど、名前からその違いはわからない）

鍵となる考え方

気取った言い回しよりも明確で正確なほうがいい。

2.2 tmp や retval などの汎用的な名前を避ける

`tmp`・`retval`・`foo`のような名前をつけるのは、「名前のことなんて考えていません」と言っているようなものだ。このような「空虚な名前」をつけるのではなく、エンティティの値や目的を表した名前を選ぼう。

例えば、`retval` を使っている JavaScript の関数があるとする。

```
var euclidean_norm = function (v) {
  var retval = 0.0;
  for (var i = 0; i < v.length; i += 1)
    retval += v[i] * v[i];
  return Math.sqrt(retval);
};
```

いい名前が思いつかなかったら、戻り値に `retval` とつけたくなる。でも、`retval` には「これは戻り値です」以外の情報はない（戻り値なのは当たり前だ）。

いい名前というのは、変数の目的や値を表すものだ。ここでは、`v` の 2 乗の合計を表しているわけだから、`sum_squares` という名前がいいだろう。こうしておけば、変数の目的を事前に伝えることができるし、バグの発見にも役立つはずだ。

例えば、ループ内の処理を間違えて以下のように書いたとする。

```
retval += v[i];
```

変数名が `sum_squares` だったら、バグを見つけやすい。

```
sum_squares += v[i]; // 合計する「square (2乗)」がない。バグだ！
```

アドバイス

`retval` という名前には情報がない。変数の値を表すような名前を使おう。

ただし、汎用的な名前に意味がないわけではない。汎用的な名前をうまく使った例を見ていく。

tmp

2 つの変数を入れ替える古典的な例を考えてみよう。

```
if (right < left) {
  tmp = right;
  right = left;
  left = tmp;
}
```

このような場合は、`tmp` という名前で全く問題ない。この変数の目的は、情報の一時的な保管だ。しかも、生存期間はわずか数行である。`tmp` という名前で「この変数には他に役割がない」という明確な意味を伝えている。つまり、他の関数に渡されたり、何度も書き換えられたりしない、ということだ。

でも、以下の `tmp` は単なる怠慢だ。

```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += " " + user.email();
...
template.set("user_info", tmp);
```

生存期間は短いけど、この変数にとっていちばん大切なことは「一時的な保管」ではない。これをわかりやすくするには、`user_info`のような名前に変えるといいだろう。

以下の例では、名前の一部に `tmp` を使っている。

```
tmp_file = tempfile.NamedTemporaryFile()
...
SaveData(tmp_file, ...)
```

ファイルオブジェクトなので、ただの `tmp` ではなく `tmp_file` という名前になっている。これを `tmp` にしていたら、どうなっていただろうか。

```
SaveData(tmp, ...)
```

この行だけを見ると、`tmp` がファイルなのか、ファイル名なのか、データなのかがわからない。

アドバイス

`tmp` という名前は、生存期間が短くて、一時的な保管が最も大切な変数にだけ使おう。

ループイテレータ

`i`・`j`・`k`・`iter`などの名前は、インデックスやループイテレータでよく使われている。汎用的な名前だけど、これだけで「ぼくはイテレータ」という意味になるので問題ない（それ以外の目的に使うとまぎらわしくなるのでやめよう！）

でも、`i`・`j`・`k` よりいい名前がある。例えば、クラブに所属しているユーザを調べるループを見てみよう。

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
```

```
if (clubs[i].members[k] == users[j])
    cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

if 文にある `members[]` と `users[]` のインデックスが逆になっている。そこだけ見ると問題がなきそうに見えるので、バグを見つけるのが難しい。

```
if (clubs[i].members[k] == users[j])
```

イテレータが複数あるときには、インデックスにもっと明確な名前をつけるといいだろう。`i`・`j`・`k` ではなく、説明的な名前 (`club_i`・`members_i`・`users_i`) にするのだ。あるいは、もっと簡潔なもの (`ci`・`mi`・`ui`) でもいいだろう。こうすればバグが目立ちやすくなる。

```
if (clubs[ci].members[ui] == users[mi]) # バグだ！最初の文字が違ってる。
```

インデックスの最初の文字は、配列の名前の最初の文字と同じになるのが正しい。

```
if (clubs[ci].members[mi] == users[ui]) # OK。最初の文字が同じだ。
```

汎用的な名前のまとめ

これまで見てきたように、汎用的な名前が役に立つこともある。

アドバイス

`tmp`・`it`・`retval` のような汎用的な名前を使うときは、それ相応の理由を用意しよう。

ただし、単なる怠慢で使われていることが多い。いい名前が思いつかなかったら、`foo`のような意味のない名前を使いたくなってしまうものだ。だけど、少しでも時間を使っていい名前を考える習慣をつけるようにすれば、すぐに「命名力」の高まりを感じられるようになるだろう。

2.3 抽象的な名前よりも具体的な名前を使う



変数や関数などの構成要素の名前は、抽象的ではなく具体的なものにしよう。

例えば、`ServerCanStart()` という名前のメソッドがあったとする。任意の TCP/IP ポートをサーバがリッスンできるかを確認するメソッドである。でも、`ServerCanStart()` という名前はちょっと抽象的だ。具体的な名前にはすれば、`CanListenOnPort()` になるだろう。これならメソッドの動作をそのまま表している。

以下の 2 つの例は、この考え方を詳しく示したものだ。

例：DISALLOW_EVIL_CONSTRUCTORS

これは Google 社から持ってきたコードだ。C++ のクラスでは、コピー・コンストラクタと代入演算子を再定義しないと、デフォルトの設定が使われてしまう。お手軽なのはいいけれど、デフォルトのメソッドだとメモリリークなどの問題につながる可能性がある。みんなが気づかない「舞台裏」で実行されてしまうからだ。

そのため Google 社では、こうした「悪の」コンストラクタを許可しない規約を作り、マクロを使って対応している。

```
class ClassName {
private:
    DISALLOW_EVIL_CONSTRUCTORS(ClassName);

public:
    ...
};
```

このマクロは以下のように定義されている。

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

このマクロをクラスの `private:` に置くと、2 つのメソッドがプライベートになる。こうすれば、間違ってメソッドを使うことはない。

でも、`DISALLOW_EVIL_CONSTRUCTORS` という名前があまりよくない。「EVIL（悪の）」という言葉には、必要以上に強い意志を感じてしまう。そんなことよりも、このマクロが「許可していないもの」を明確にするほうが大切だ。実際には `operator=()` メソッドも許可していない。このメソッドは「CONSTRUCTORS（コンストラクタ）」ではないのだ！

長年使用されてきた名前だったけど、最終的にはあまり刺激的ではなく、より具体的な名前に変更されることになった。

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) ...
```

例：--run_locally

ほくたちのプログラムには、`--run_locally` という名前のコマンドオプションがあった。このオプションをつけると、プログラムがデバッグ情報を印字するようになる。ただし、動作は遅くなってしまう。このオプションは、デスクトップなどのローカルマシンでテストをするときに使っていた。リモートサーバで動かすときにはパフォーマンスが重要になるので使わなかった。

`--run_locally` の名前の由来はわかったと思う。でも、これには問題があった。

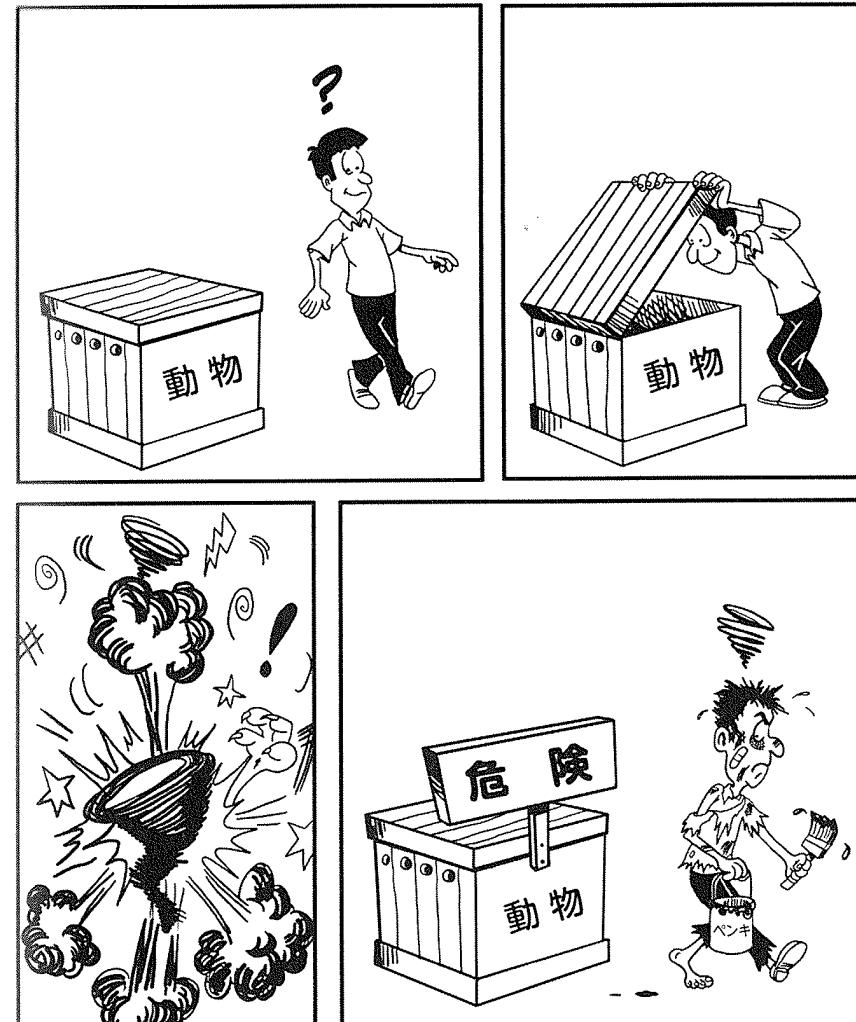
- まず、新しいチームメンバがこのオプションを理解できなかった。ローカルで動かすときには使っていたようだけど（名前から推測したのだろう）、どうしてこれが必要なのかはわからていなかつた。
- リモートで動かすときにデバッグ情報を印字したいこともある。そんなときは、リモートで動いているプログラムに`--run_locally` オプションを渡す。リモートなのに「locally」ってなんだか変な感じだ。すごくまぎらわしい。
- ローカルでパフォーマンステストをしたいこともある。そんなときは、動作の遅くなるロギング機能は使いたくない。だから、ローカルで動かすけど、`--run_locally` は使わない。

`--run_locally` に環境の名前が入っているのが問題なのだ。`--extra_logging` という名前のほうが直接的で明確だ。

でも、`--run_locally` の機能がロギングだけじゃなかったらどうしよう？ 例えば、ローカル用の特別なデータベースを設定して使う必要があったとしよう。それなら両方の機能を表現できている`--run_locally` でもいいような気がしてくる。

でも、名前と目的が合っていない。あいまいだし間接的だ。ここでは、`--use_local_database` のようなオプションを用意するといいだろう。オプションは2つに増えるけど、それぞれの意味は明確だ。直交する概念は無理に1つにまとめようとせずに、別々に使えるようにするといい。

2.4 名前に情報を追加する



前にも言ったけど、名前は短いコメントのようなものだ。変数名に詰め込める情報はあまり多くない。だけど、名前につけた情報は変数を見るたびに目に入ってくる。

だから、もし絶対に知らせなきゃいけない大切な情報があれば、「単語」を変数名に追加すればいい。例えば、16進数の文字列を持つ変数について考えてみよう。

```
string id; // 例: "af84ef845cd8"
```

ID のフォーマットが大切なら、名前を `hex_id` にするといい。

値の単位

時間やバイト数のように計測できるものであれば、変数名に単位を入れるといいだろう。

例えば、ウェブページの読み込み時間を計測する JavaScript のコードを見てみよう。

```
var start = (new Date()).getTime(); // ページの上部
...
var elapsed = (new Date()).getTime() - start; // ページの下部
document.writeln("読み込み時間：" + elapsed + " 秒");
```

このコードは間違っていないように見える。でも、これではうまく動かない。`getTime()` が秒ではなく、ミリ秒を返すからだ。

変数名に `_ms` を追加すれば明確になる。

```
var start_ms = (new Date()).getTime(); // ページの上部
...
var elapsed_ms = (new Date()).getTime() - start_ms; // ページの下部
document.writeln("読み込み時間：" + elapsed_ms / 1000 + " 秒");
```

時間以外にもプログラミングで使う単位はたくさんある。以下の表は、単位のない仮引数と、単位を追加したよりよいバージョンの仮引数を示したものだ。

関数の仮引数	単位を追加した仮引数
<code>Start(int delay)</code>	<code>delay → delay_secs</code>
<code>CreateCache(int size)</code>	<code>size → size_mb</code>
<code>ThrottleDownload(float limit)</code>	<code>limit → max_kbps</code>
<code>Rotate(float angle)</code>	<code>angle → degrees_cw</code>

その他の重要な属性を追加する

変数名に追加する情報は単位だけではない。危険や注意を喚起する情報も追加したほうがいい。

例えば、セキュリティ問題の多くは、プログラムの受信するデータが安全ではないことが原因で発生している。このようなデータには、`untrustedUrl` や `unsafeMessageBody` などの変数名を使うといいだろう。データを安全にする関数を呼び出したあとは、変数名を `trustedUrl` や `safeMessageBody` にするといい。

以下の表は、情報を変数名に追加したほうがいい例を示している。

状況	変数名	改善後
<code>password</code> はプレインテキストなので、処理をする前に暗号化すべきである。	<code>password</code>	<code>plaintext_password</code>
ユーザが入力した <code>comment</code> は表示する前にエスケープする必要がある。	<code>comment</code>	<code>unescape_comment</code>
<code>html</code> の文字コードを UTF-8 に変えた。	<code>html</code>	<code>html_utf8</code>
入力された <code>data</code> を URL エンコードした。	<code>data</code>	<code>data_urlenc</code>

すべての変数名に `unescape_` や `_utf8` などの属性を追加しろということではない。変数の意味を間違えてしまったときにバグになりそうなところにだけ使うことが大切だ。特にセキュリティのバグのような深刻な被害が出るところに使うといいだろう。基本的には、変数の意味を理解してもらわなければ困るところに属性を追加しておこう。

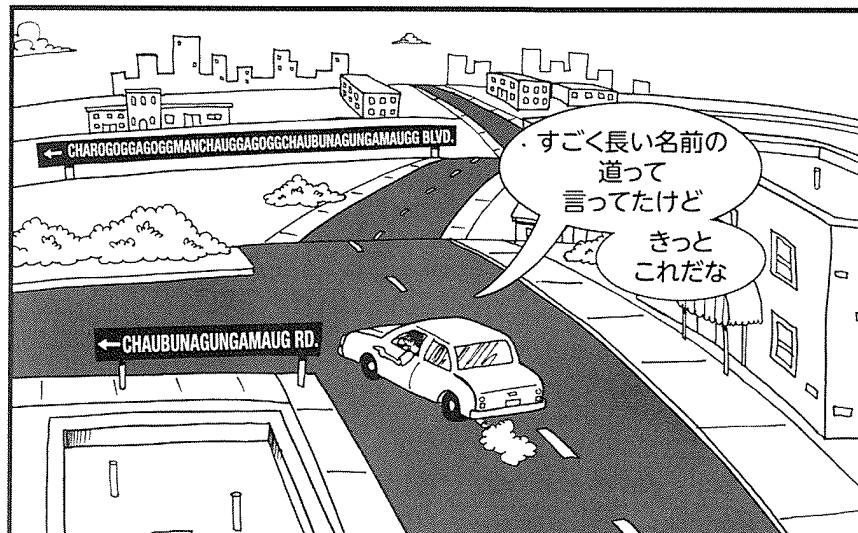
これってハンガリアン記法なの？

ハンガリアン記法というのは、Microsoft 社で広く使われていた命名規則だ。すべての変数名の接頭辞に「型」をつける。例えば、こんな感じだ。

変数名	意味
<code>pLast</code>	あるデータ構造の最後の要素を指すポインタ (p)
<code>pszBuffer</code>	ゼロ終端 (z) の文字列 (s) バッファを指すポインタ (p)
<code>cch</code>	文字 (ch) のカウント (c)
<code>mpcopx</code>	カラーのポインタ (pco) から X 軸のポインタ (px) を指すマップ (m)

これも「名前に属性を追加」しているけれど、もっと厳密で規律のあるシステムだ。一方、本節でぼくたちが提唱しているのは、もっと大まかで規律の緩いシステムだ。必要なときにだけ変数の大切な属性を見つけ出して、それを読みやすくして名前に追加する。これをハンガリアンならぬ「イングリッシュ記法」と呼んでもいいだろう。

2.5 名前の長さを決める



いい名前を選ぶときには、「長い名前を避ける」という暗黙的な制約がある。以下のような識別子は、誰もが嫌がるだろう。

```
newNavigationControllerWrappingViewControllerForDataSourceOfClass
```

長い名前は覚えにくいし、画面を大きく占領してしまう。折り返しが必要になれば、コード行が無駄に増えてしまう。

だけど、「長い名前を避ける」を真に受けてしまったら、1つの単語（あるいは1文字）だけの名前になってしまう。それじゃあ、どのあたりに線を引けばいいのだろう？ `d · days · days_since_last_update` のなかから1つを選ぶにはどうすればいいだろう？

それは、変数の使い方によって違ってくる。でも、ガイドラインはある。

スコープが小さければ短い名前でもいい

長期休暇よりも短期でどこかへ行くときのほうが荷物は少ないはずだ。それと同じで、識別子の「スコープ」（その名前が「見える」コードの行数）が小さければ、多くの情報を詰め込む必要はない。すべての情報（変数の型・初期値・破棄方法など）が見えてるので、変数の名前は短くていい。

```
if (debug) {
    map<string,int> m;
    LookUpNamesNumbers(&m);
    Print(m);
}
```

`m` という変数名にはあまり情報が含まれていない。でも、大丈夫。コードを理解するのに必要な情報がすぐそばにあるからだ。

`m` がクラスのメンバ変数やグローバル変数ならどうだろう。以下のようないふにするかもしれない。

```
LookUpNamesNumbers(&m);
Print(m);
```

このコードは読みにくい。`m` の型や目的がよくわからないからだ。

識別子のスコープが大きければ、名前に十分な情報を詰め込んで明確にする必要がある。もくじりもどかしい

長い名前を入力するのは問題じゃない

長い名前を避ける理由はいくつもある。でも、「入力しにくい」というのは、もはや理由にならない。プログラミングに使うテキストエディタには「単語補完」機能がついている。この機能を知らないプログラマは意外と多い。まだ使ったことがないなら、本書をひとまず横に置いて、すぐに試してみて欲しい。

1. 名前の最初の1文字を入力する。
2. 単語補完のコマンドを実行する（以下参照）。

3. 補完された単語が正しくなければ、正しい名前が出現するまでコマンドを実行し続ける。

これがビックリするほど正確なんだ。どんなファイル形式でも、どんな言語でもうまくいく。それにどんなトークンにも使える。コメントを入力しているときでも大丈夫だ。

エディタ	コマンド
Vi	Ctrl-p
Emacs	Meta-/ (ESC を押してから /)
Eclipse	Alt-/
IntelliJ IDEA	Alt-/
TextMate	ESC

頭文字と省略形

プログラマは頭文字や省略形を使って名前を短くすることがある。例えば、クラス名を `BackEndManager` じゃなくて `BEManager` にしたりする。このように情報を圧縮すると混乱の元になるだろうか？

ほくたちの経験からすると、プロジェクト固有の省略形はダメだ。新しくプロジェクトに参加した人は、暗号みたいに見えて怖いと思うだろう。しばらくすると、それを書いた人ですら暗号みたいで怖いと思うようになる。

新しいチームメイトはその名前の意味を理解できるだろうか？ 理解できるなら問題ない。

プログラマは、`evaluation` の代わりに `eval` を使う。`document` の代わりに `doc` を使う。`string` の代わりに `str` を使う。だから、新しいチームメイトも `FormatStr()` の意味は理解できる。でも、`BEManager` の意味は理解できない。

不要な単語を投げ捨てる

名前に含まれる単語を削除しても情報が全く損なわれることもある。例えば、`ConvertToString()` を短くして `ToString()` にしても、必要な情報は何も損なわれていない。同様に、`DoServeLoop()` を `ServeLoop()` に変えても明確さは同じだ。

2.6 名前のフォーマットで情報を伝える

アンダースコア・ダッシュ・大文字を使って、名前に情報を詰め込むこともできる。例えば、Google 社のオープンソースプロジェクトで使っている C++ のフォーマット規約を見てみよう。

```
static const int kMaxOpenFiles = 100;

class LogReader {
public:
    void OpenFile(string local_file);

private:
    int offset_;
    DISALLOW_COPY_AND_ASSIGN(LogReader);
};
```

エンティティごとに異なるフォーマットを使っている。一種のシンタックスハイライトと言えるかもしれない。これでコードが読みやすくなっている。

このフォーマットはよく使われているものだ。クラス名は `CamelCase` (キャメルケース) で、変数名は `lower_separated` (小文字をアンダースコアで区切ったもの)。でも、その他の規約にはビックリするかもしれない。

例えば、定数は `CONSTANT_NAME` ではなく `kConstantName` になっている。`MACRO_NAME` のような `#define` マクロと簡単に区別できるからだ。

クラスのメンバ変数は、一見普通の変数のように見えるけど、`offset_` のように最後の文字がアンダースコアになっている。ほくたちも最初は奇妙に思ったけど、普通の変数と区別できるので便利だ。例えば、長いメソッドを読んでいると、以下のようなコードが目に入ってくる。

```
stats.clear();
```

普通なら「`stats` はクラスのメンバ変数かな？」これはクラスの内部状態を変えるコードなんだろうか？」などと考えるだろう。メンバ変数を `member_` にする規約を守っていれば、「メンバ変数じゃない。この `stats` はローカル変数だ」とすぐにわかる。`クラスのメンバ変数なら stats_` になるからだ。

その他のフォーマット規約

プロジェクトや言語によって使えるフォーマット規約は違ってくる。なかにはより多くの情報を詰め込める規約もある。

例えば、『JavaScript: The Good Parts』(Douglas Crockford, O'Reilly, 2008)[†]では、「コンストラクタ」(newを使って呼び出される関数)は大文字で始め、通常の関数は小文字で始めるように著者が提唱している。

```
var x = new DatePicker(); // DatePicker() は「コンストラクタ」関数
var y = pageHeight();     // pageHeight() は通常の関数
```

その他にも JavaScript では、jQuery のライブラリ関数 (\$だけの関数) を呼び出したときには、変数名の頭に \$をつけるというものがある。

```
var $all_images = $("img"); // $all_images は jQuery のオブジェクト
var height = 250;           // height は違う
```

こうすれば、\$all_images が jQuery のオブジェクトだと明確にわかる。

最後に HTML と CSS の例を紹介しよう。HTML タグの id や class などの属性名には、アンダースコアやハイフンも妥当な文字として使うことができる。ただし、id の区切り文字にはアンダースコアを、class の区切り文字にはハイフンを使う規約が有力だ。

```
<div id="middle_column" class="main-content"> ...
```

以上のような規約を使うかどうかは、自分自身やチームで決めるといい。どんなものを使うにしても、プロジェクトで一貫性を持たせることが大切だ。

2.7 まとめ

本章のテーマは「名前に情報を詰め込む」だった。つまり、名前を見ただけで情報を読み取れるようにすることだ。

以下は、ぼくたちが説明したヒントだ。

- 明確な単語を選ぶ。——例えば、Getではなく、状況に応じて Fetch や Downloadなどを使う。
- tmp や retval などの汎用的な名前を避ける。——ただし、明確な理由があれば話は別だ。
- 具体的な名前を使って、物事を詳細に説明する。——ServerCanStart() よりも CanListenOnPort() のほうが明確だ。
- 変数名に大切な情報を追加する。——ミリ秒を表す変数名には、後ろに _ms をつける。これからエスケープが必要な変数名には、前に raw_ をつける。
- スコープの大きな変数には長い名前をつける。——スコープが数画面に及ぶ変数に 1~2 文字の短い暗号めいた名前をつけてはいけない。短い名前はスコープが数行の変数につけるべきだ。
- 大文字やアンダースコアなどに意味を含める。——例えば、クラスのメンバ変数にアンダースコアをつけて、ローカル変数と区別する。

[†] 訳注『JavaScript: The Good Parts——「良いパート』によるベストプラクティス』(ダグラス・クロックフォード著、水野貴明訳、オライリー・ジャパン)

前章では、名前に情報を詰め込むことについて触れた。本章では、また違った話題に触れたいと思う。それは「誤解される名前に気を付けろ」だ。

鍵となる考え方

名前が「他の意味と間違えられることはないだろうか？」と何度も自問自答する。

ここではクリエイティブになって欲しい。積極的に「誤解」を探していくのだ。そうすれば変更が必要なあいまいな名前が見つかるだろう。

本章に登場する例では、誤解する可能性を「ひとりごと」で確認してから、名前を選ぶようにしている。

3.1 例：filter()

データベースの問い合わせ結果を処理するコードを書いているとしよう。

```
results = Database.all_objects.filter("year <= 2011")
```

この results には何が含まれているだろうか？

- 「year <= 2011」のオブジェクト
- 「year <= 2011」ではないオブジェクト

どちらかよくわからないのは、filter があいまいな言葉だからだ。これでは「選択する」のか「除外する」のかわからない。filter という名前は避けるべきだ。簡単に誤解を招いてしまう。

「選択する」のであれば、select() にしたほうがいい。「除外する」のであれば、exclude() にしたほうがいい。

3.2 例：Clip(text, length)

段落の内容を切り抜く関数があるとしよう。

```
# text の最後を切り落として、「...」をつける
def Clip(text, length):
    ...
    if len(text) > length:
        text = text[:length] + "..."
```

Clip() の動作は 2 つ考えられる。

- 最後から length 文字を削除する (remove)
- 最大 length 文字まで切り詰める (truncate)

後者のような気もするけど、確実なことは言えない。読み手に疑問を抱かせるよりも、関数名を Truncate(text, length) に変えるほうがいい。
それから、length という名前もダメだ。max_length にしたほうが明確になる。
これで終わりじゃない。max_length もいろんな解釈ができる。

- バイト数
- 文字数
- 単語数

前章で触れたように、ここでは名前に単位をつけたほうがいい。この場合は「文字数」を意味しているので、max_length ではなく max_chars にするといいだろう。

3.3 限界値を含めるときは min と max を使う

ショッピングカートには商品が 10 点までしか入らないとしよう。

```
CART_TOO_BIG_LIMIT = 10

if shopping_cart.num_items() >= CART_TOO_BIG_LIMIT:
    Error("カートにある商品数が多すぎます。")
```

このコードには古典的な「off-by-one エラー[†]」のバグがある。これを修正するには >= を > に変えればいい。

```
if shopping_cart.num_items() > CART_TOO_BIG_LIMIT:
```

(あるいは CART_TOO_BIG_LIMIT を 11 に変えてもいい。) だけど、ここでの根本的

[†] 訳注 境界条件の判定に関するエラー (参考：http://ja.wikipedia.org/wiki/Off-by-one_エラー)。

な問題は、`CART_TOO_BIG_LIMIT` という名前があいまいなことだ。これでは「未満（限界値を含まない）」なのか「以下（限界値を含む）」なのかがわからない。

アドバイス

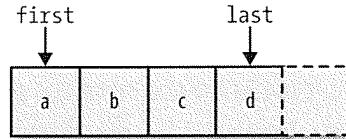
限界値を明確にするには、名前の前に `max_` や `min_` をつけよう。

この場合は、`MAX_ITEMS_IN_CART` という名前にするべきだ。こうすればコードがより明確になる。

```
MAX_ITEMS_IN_CART = 10
```

```
if shopping_cart.num_items() > MAX_ITEMS_IN_CART:
    Error("カートにある商品数が多すぎます。")
```

3.4 範囲を指定するときは first と last を使う



「未満」と「以下」の例は他にもある。

```
print integer_range(start=2, stop=4)
# これが印字するのは、[2,3]？ それとも [2,3,4]？ (あるいはその他?)
```

`start` は適切な名前だ。でも、`stop` は複数の意味に解釈できる。

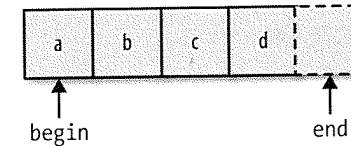
包含的な範囲を表すのであれば（終端を範囲に含めるのであれば）、`first` と `last` を使うのがいい。例えば、以下のようになる。

```
set.PrintKeys(first="Bart", last="Maggie")
```

`stop` ではなく `last` を使えば、包含していることが明確になる。

意味的に「正しく聞こえる」のであれば、`first` と `last` 以外にも `min` と `max` を使って包含的範囲を表すことができる。

3.5 包含／排他的範囲には begin と end を使う



包含／排他的な範囲は便利だ。例えば、10月16日に開催されたイベントをすべて印字したいとする。以下のように書くほうが、

```
PrintEventsInRange("OCT 16 12:00am", "OCT 17 12:00am")
```

以下のように書くよりも簡単だ。

```
PrintEventsInRange("OCT 16 12:00am", "OCT 16 11:59:59.9999pm")
```

ここに使う仮引数の名前は何がいいだろうか？ プログラミングの命名規約では、包含／排他的範囲に `begin` と `end` を使うことが多い。

でも、`end` は少しあいまいだ。例えば、「本の終盤 (the end of the book) を読んでいる」の「`end`」は包含的だ。残念ながら英語には「ちょうど最後の値を超えたところ」を意味する簡潔な言葉がない。

`begin` と `end` の対はイディオムになっている（少なくとも C++ の標準ライブラリではこれが使われている。また、配列がこのように「スライス」されることも多い）ので、これが最善の選択と言える。

3.6 ブール値の名前

ブール値の変数やブール値を返す関数の名前を選ぶときには、`true` と `false` の意味を明確にしなければいけない。

以下は危険な例だ。

```
bool read_password = true;
```

「`read`」をどう「読む」かになるけど（ギャグじゃないよ）、これには2つの解釈の仕方がある。

- パスワードをこれから読み取る必要がある。
- パスワードをすでに読み取っている。

ここでは「read」を避けるべきだろう。代わりに、`need_password` や `user_is_authenticated` を使ったほうがいい。

ブール値の変数名は、頭に `is`・`has`・`can`・`should`などをつけてわかりやすくすることが多い[†]。

例えば、`SpaceLeft()` という名前は数値を返すように聞こえる。ブール値を返したいのであれば、`HasSpaceLeft()` という名前にしたほうがいい。

それから、名前を否定形にするのは避けたほうがいい。例えば、

```
bool disable_ssl = false;
```

ではなく、肯定形にしたほうが声に出て読みやすい（それに短くて済む）。

```
bool use_ssl = true;
```

3.7 ユーザの期待に合わせる

たとえ別の意味で使っていたとしても、ユーザが先入観を持っているために誤解を招いてしまうことがある。こういうときは「負けを認めて」、誤解されない名前に変えたほうがいい。

例：get^{*}()

多くのプログラマは、`get` で始まるメソッドはメンバの値を返すだけの「軽量アクセサ」であるという規約に慣れ親しんでいる。この規約を守らなければ、誤解を招く可能性がある。

以下は、やってはいけない例だ。コードは Java で書いている。

```
public class StatisticsCollector {
    public void addSample(double x) { ... }
```

[†] 註注 Ruby のメソッドや Scheme には名前の後ろに ? をつける慣習がある（例：`authenticated?`）。また、Lisp には p をつける慣習がある（例：`authenticatedp`）。

```
public double getMean() {
    // すべてのサンプルをイテレートして、total / num_samples を返す。
}
```

`getMean()` は過去のデータをすべてイテレートして、その場で平均値を計算する実装になっている。データが大量にあったらものすごいコストだ！ そのことを知らないプログラマは、コストが高いとは思わず `getMean()` を呼び出してしまうだろう。

コストの高さが事前にわかるように、このメソッドは `computeMean()` などの名前に変えるべきだろう（あるいは、コストの高くない実装に変えるべきだろう）。

例：list::size()

C++ の標準ライブラリの例を挙げよう。以下のコードには見つけにくいバグが存在する。これが原因でサーバの速度がめちゃくちゃ遅くなったりする。

```
void ShrinkList(list<Node>& list, int max_size) {
    while (list.size() > max_size) {
        FreeNode(list.back());
        list.pop_back();
    }
}
```

この「バグ」が発生しているのは、`list.size()` の計算量が $O(n)$ であることを作者が知らなかったからだ。リンクリストのノード数を事前計算をせずに順番にカウントしているので、`ShrinkList()` 全体の計算量が $O(n^2)$ になっている。

このコードは技術的に「正しい」し、ユニットテストもすべて成功している。でも、要素数が 100 万個のリストを `ShrinkList()` に渡したら、終了までに 1 時間以上かかるてしまう！

「そんなの呼び出し側の責任だよ。事前にドキュメントをちゃんと読むべきだ」なんて思うかもしれない。もちろん君が正しい。でもこの場合は、`list.size` に時間がかかりすぎるのが問題だ。C++ には他にもコンテナがあるけど、どれも一定時間で終了する `size()` メソッドを持っている。

`size()` という名前が `countSize()` や `countElements()` だったら、このような問題は起きなかつたはずだ。C++ の標準ライブラリの作者は、`vector` や `map` などのコン

テナに合わせてメソッド名を `size()` にしたのだと思う。でも、そのことが原因で、他のコンテナと同じ高速な操作と間違えられるようになってしまった。なお、最新の C++ 標準では、`size()` の計算量を $O(1)$ にすることが定められている。

ウィザードは誰だ？

OpenBSD をインストールしていたときのことだ。ディスクのフォーマット中に難解なメニューが表示されて、ディスクパラメータを選択せよと言われた。オプションには「ウィザードモード」があった。ユーザに優しいオプションが見つかったので、安心してそれを選んだ。すると、インストーラが終了して、プロンプトが表示された。ディスクフォーマットのコマンドを手入力せよと言っている。ここから脱出することもできなかった。「ウィザード」とは、ユーザ自身のことだったんだ！

3.8 例：複数の名前を検討する

名前を決めるときには、複数の候補を検討すると思う。最終的に決める前に、それぞれ長所について話し合うのが普通だ。以下の例は、この選考過程について示したものだ。

高トラフィックのウェブサイトでは、ウェブサイトの変更によってビジネスがどのくらい改善できるかを調べる「実験」をすることが多い。以下の例は、実験用の設定ファイルだ。

```
experiment_id: 100
description: "フォントサイズを 14pt に上げる"
traffic_fraction: 5%
...
```

設定ファイルには、属性と値のペアが 15 個ほど定義されている。同じような実験をするときには、設定ファイルの大部分をコピペしなくてはいけない。

```
experiment_id: 101
description: "フォントサイズを 13pt に上げる"
[以下、experiment_id 100と同じ]
```

既存の設定ファイルを他の実験でも使えるようにしたい（これは「プロトタイプ継承」パターンと呼ばれる手法だ）。そうすれば、以下のように書ける。

```
experiment_id: 101
the_other_experiment_id_I_want_to_reuse: 100
[以下、変更が必要な情報だけ書き換える]
```

ここで考えなければいけないのは、「`the_other_experiment_id_I_want_to_reuse`（再利用したい実験の ID）の名前を何にするか？」だ。

以下の 4 つの名前を検討してみよう。

1. template
2. reuse
3. copy
4. inherit

どの名前もぼくたちには適切に思える。新しい機能を追加したのは、ぼくたち自身だからだ。でも、この機能を知らない人が見たらどうなるかを想像しなければいけない。それぞれの名前を調べていこう。他の誰かが誤解する可能性を考えるんだ。

1. まずは、`template` から考えよう。

```
experiment_id: 101
template: 100
...
...
```

`template` にはいくつかの問題がある。まず、「これはテンプレートだ」なのか「このテンプレートを使っている」なのかがわかりにくい。次に、「テンプレート」という言葉は、抽象的なものに何かを「埋め込」んで、具体的なものにするために使うものだ。テンプレートに使う実験のことを「本物」の実験ではない抽象的なものと誤解する人がいるかもしれない。この状況で `template` を使うには意味があいまいすぎるのだ。

2. reuseはどうだろう。

```
experiment_id: 101
reuse: 100
...
...
```

reuseは悪くない言葉だ。でも、文字だけを見ると「この実験は100回再利用できる」と誤解される可能性もある。名前を reuse_id に変えたほうがいいだろう。でも、reuse_id: 100 のことを「この実験の再利用idは100だ」と誤解するユーザがいるかもしれない。

3. copyについて考えてみよう。

```
experiment_id: 101
copy: 100
...
...
```

copyはいい名前だ。でも、copy: 100だけでは、「この実験を100回コピーする」なのか「これは100回めのコピーだ」のかがわからない。他の実験を参照している言葉だということを明確にするには、copy_experiment という名前に変えるといい。今までのところ、これが最善の名前だ。

4. 最後に inheritを検討しよう。

```
experiment_id: 101
inherit: 100
...
...
```

inheritという言葉はプログラマにはなじみがある。何かを継承するというのは、新たに変更を加えるという意味だ。クラスを継承すれば、そのクラスのメソッドとメンバがすべて手に入り、それらを変更したり新しく追加したりできる。現実の世界で考えてみても、身内の財産を継承するというのは、その財産を自分で好きなように保有したり売却したりできるという意味だ。

ただし、他の実験から継承していることは明確にしておこう。名前を inherit_from や inherit_from_experiment_id に変えるといいだろう。

以上の検討結果から、最善の名前は copy_experiment と inherit_from_

experiment_id ということになった。その理由は、何が起きるかを明確に表していて、誤解を生む可能性が低いからだ。

3.9 まとめ

最善の名前とは、誤解されない名前である。つまり、君のコードを読んでいる人が、君の意図を正しく理解できるということだ。英語の単語は、filter・length・limit のように、プログラミングに使うには意味があいまいなものが多い。

名前を決める前に反対意見を考えるなどして、誤解されない名前かどうかを想像してみよう。最善の名前というのは、誤解されない名前である。

上下の限界値を決めるときには、max_ や min_ を前に付けるといい。包含的範囲であれば、first や last を使うといいだろう。包含／排他的範囲であれば、begin と end がイディオムなのでそれを使う。布尔値に名前をつけるときには、それが布尔値だとわかるように is や has などの単語を使う。disable_ssl のような否定形は避ける。

単語に対するユーザの期待にも注意する。例えば、get() や size() には軽量なメソッドが期待されている。

雑誌のレイアウトには多くの考えが詰め込まれている。段落の長さ。横幅。記事の順番。表紙。優れた雑誌というのは、ページを飛ばして読めるようになっているし、順番に読めるようになっている。

優れたソースコードは「目に優しい」ものでなければいけない。本章では、コードを読みやすくするための余白・配置・順序について説明しよう。

具体的には、ぼくたちが使っている3つの原則についてだ。

- 読み手が慣れているパターンと一貫性のあるレイアウトを使う。
- 似ているコードは似ているように見せる。
- 関連するコードをまとめてブロックにする。

美しさと設計

本章では、コードの単純な「美しさ」の改善を扱う。こうした変更は取り組みやすいし、コードがすごく読みやすいものになる。大きなりファクタリング（新しい関数やクラスの導入など）がもっとうまくいくようになることが多い。ぼくたちは、美しさと優れた設計は独立した考え方だと思っている。できれば両方を追求してもらいたい。

4.1 なぜ美しさが大切なのか?



以下のようなコードを使わなければいけないとしよう。

```
class class StatsKeeper {
public:
// double を記録するクラス
void Add(double d); // とすばやく統計を出すメソッド
private: int count; /* それまでの 個数
/* public:
double Average();

private: double minimum;
list<double>
past_items
;double maximum;
};
```

これを理解するには時間がかかると思う。では、以下のキレイなバージョンならどうだろう。

```
// double を記録するクラスと
// すばやく統計を出すメソッド
class StatsKeeper {
public:
void Add(double d);
double Average();

private:
list<double> past_items;
int count; // それまでの個数

double minimum;
double maximum;
};
```

見た目が美しいコードのほうが使いやすいのは明らかだ。考えてみれば、プログラミングの時間のほとんどはコードを読む時間なのだッ! さっと流し読みができる、誰にとっても使いやすいコードだと言えるだろう。

4.2 一貫性のある簡潔な改行位置

とあるJavaのコードを書いているとしよう。任意の速度のネットワークに接続したときに、プログラムがどのように動くかを評価するコードだ。TcpConnectionSimulatorクラスのコンストラクタには4つの仮引数がある。

1. 接続速度 (Kbps)
2. 平均遅延時間 (ms)
3. 遅延「イライラ」時間 (ms)
4. パケットロス率 (%)

このコードには TcpConnectionSimulator のインスタンスが3つ必要だ。

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi = new TcpConnectionSimulator(
        500, /* Kbps */
        80, /* millisecs latency */
        200, /* jitter */
        1 /* packet loss % */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Kbps */
            10, /* millisecs latency */
            0, /* jitter */
            0 /* packet loss % */);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(
            100, /* Kbps */
            400, /* millisecs latency */
            250, /* jitter */
            5 /* packet loss % */);
}
```

横幅80文字に合わせるために（これは会社のコーディング標準なのだ）余計な改行が入っている。その結果、t3_fiber の見た目が他と違って残念なことになってい

る。コードの「シルエット」が変なので、自然と t3_fiber に目が向いてしまう。それに「似ているコードは似ているように見せる」の原則も守られていない。

コードの見た目を一貫性のあるものにするには、適切な改行を入れるようにしよう（それからコメントも整列させよう）。

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(
            500, /* Kbps */
            80, /* millisecs latency */
            200, /* jitter */
            1 /* packet loss % */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Kbps */
            10, /* millisecs latency */
            0, /* jitter */
            0 /* packet loss % */);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(
            100, /* Kbps */
            400, /* millisecs latency */
            250, /* jitter */
            5 /* packet loss % */);
}
```

このコードには一貫性があり、楽に目を通すことができる。でも、縦に長くなっているし、同じコメントが3回も繰り返されている。

このクラスを簡潔に書いたら以下になる。

```
public class PerformanceTester {
    // TcpConnectionSimulator(throughput, latency, jitter, packet_loss)
    // [Mbps] [ms] [ms] [percent]

    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(500, 80, 200, 1);
```

```

public static final TcpConnectionSimulator t3_fiber =
    new TcpConnectionSimulator(45000, 10, 0, 0);

public static final TcpConnectionSimulator cell =
    new TcpConnectionSimulator(100, 400, 250, 5);
}

```

コメントを最上部に移動して、仮引数を一行で書くようにした。数値の右隣からコメントがいなくなったけど、より簡潔な表組みに「データ」が並ぶようになった。

4.3 メソッドを使った整列

人事データベースがあるとしよう。以下の関数が提供されている。

```

// 「Doug Adams」のような partial_name を「Mr. Douglas Adams」に変える。
// それができなければ、error に説明文を入れる。
string ExpandFullName(DatabaseConnection dc, string partial_name, string* error);

```

この関数は実例を使ってテストしている。

```

DatabaseConnection database_connection;
string error;
assert(ExpandFullName(database_connection, "Doug Adams", &error)
    == "Mr. Douglas Adams");
assert(error == "");
assert(ExpandFullName(database_connection, " Jake Brown ", &error)
    == "Mr. Jacob Brown III");
assert(error == "");
assert(ExpandFullName(database_connection, "No Such Guy", &error) == "");
assert(error == "no match found");
assert(ExpandFullName(database_connection, "John", &error) == "");
assert(error == "more than one result");

```

見た目が美しくない。長すぎて折り返されているところもある。シルエットは不細工だし一貫性のあるパターンがない。

これは改行の位置を変えなければいけない。でも、「assert(ExpandFullName(database_connection...))」や「error」などの文字列が何度も登場して邪魔をしている。このコードを本当の意味で改善するには、ヘルパー・メソッドを使う必要がある。

```

CheckFullName("Doug Adams", "Mr. Douglas Adams", "");
CheckFullName(" Jake Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy", "", "no match found");
CheckFullName("John", "", "more than one result");

```

これなら引数の異なる4つのテストがあることがよくわかる。「面倒な仕事」はすべて CheckFullName() に隠されることになったけど、このメソッドも悪くはない。

```

void CheckFullName(string partial_name,
                  string expected_full_name,
                  string expected_error) {
    // database_connection はクラスのメンバになっている。
    string error;
    string full_name = ExpandFullName(database_connection, partial_name, &error);
    assert(error == expected_error);
    assert(full_name == expected_full_name);
}

```

ここでの目標はコードの見た目を美しくすることだったけど、この変更によってうれしい副作用がもたらされることになった。

- 重複を排除したことでコードが簡潔になった。
- テストケースの大切な部分（名前やエラー文字列）が見やすくなった。以前は、database_connection や error などのトークンに囲まれていて、コードを見て「飲み込む」のが難しかった。
- テストの追加が簡単になった。

この話の教訓は、コードの「見た目をよく」すれば、表面上の改善だけではなく、コードの構造も改善できるということだ。

4.4 縦の線をまっすぐにする

縦の線をまっすぐにすれば、文章に目を通しやすくなる。

列を「整列」させれば、コードが読みやすくなることがある。例えば、前節の CheckFullName() 引数は、空白を使って整列できる。

```
CheckFullName("Doug Adams" , "Mr. Douglas Adams" , "");
CheckFullName(" Jake Brown ", "Mr. Jake Brown III" , "");
CheckFullName("No Such Guy" , "" , "no match found");
CheckFullName("John" , "" , "more than one result");
```

CheckFullName() の 2 番めと 3 番めの引数がわかりやすくなった。

以下の例では、複数の変数を定義している。

```
# POST のパラメータをローカル変数に割り当てる
details = request.POST.get('details')
location = request.POST.get('location')
phone = request.POST.get('phone')
email = request.POST.get('email')
url = request.POST.get('url')
```

もう気づいたかもしれないけど、3 つめの定義でタイプミスをしている（request が equest になっている）。コードを整列しておけば、こういうミスが見つけやすくなる。

wget のコードでは、利用可能なコマンドラインオプションが（100 個以上ある）、以下のように並べられている。

```
commands[] = {
    ...
    { "timeout",      NULL,           cmd_spec_timeout },
    { "timestamping", &opt.timestamping, cmd_boolean },
    { "tries",        &opt.ntry,       cmd_number_inf },
    { "useproxy",     &opt.use_proxy,  cmd_boolean },
    { "useragent",    NULL,           cmd_spec_useragent },
    ...
};
```

こうしておけば、次から次へと楽に流し読みできる。

整列すべきなのか？

縦の線が「視覚的な手すり」になれば、流し読みが楽にできるようになる。これは「似ているコードは似ているように見せる」のいい例だ。

でも、これが好きではないプログラマもいる。整列やその維持に手間がかかるとい

うのだ。1 行だけ変更したいのに、他の行も（それも空白だけ）変更しなければいけないので、「差分」が増えるという人もいる。

でも、試しにやってみてはどうだろうか。ほくたちの経験では、プログラマが心配するほどの手間にはならない。もし手間になるようだったら、そのときは止めればいい。

4.5 一貫性と意味のある並び

コードの並びがコードの正しさに影響を及ぼすことは少ない。例えば、以下の 5 つの変数の定義はどんな順番で並べても構わない。

```
details = request.POST.get('details')
location = request.POST.get('location')
phone = request.POST.get('phone')
email = request.POST.get('email')
url = request.POST.get('url')
```

であれば、ランダムに並べるのではなく、意味のある順番に並べるといい。例えば、こんな感じだ。

- 対応する HTML フォームの <input> フィールドと同じ並び順にする。
- 「最重要」なものから重要度順に並べる。
- アルファベット順に並べる。

どの並び順を選ぶにしても、一連のコードでは同じ並び順を使うべきだ。並び順を変えてしまうと、あとでわかりにくくなる。

```
if details: rec.details = details
if phone:   rec.phone   = phone   # あれ、'location' はどこ？
if email:   rec.email  = email
if url:     rec.url    = url
if location: rec.location = location # なんで 'location' が下にあるの？
```

4.6 宣言をブロックにまとめる

人間の脳はグループや階層を1つの単位として考える。コードの概要をすばやく把握してもらうには、このような「単位」を作ればいい。

例えば、フロントエンドサーバ用のC++のクラスがあるとする。メソッド宣言は以下のようになっている。

```
class FrontendServer {
public:
    FrontendServer();
    void ViewProfile(HttpServletRequest* request);
    void OpenDatabase(string location, string user);
    void SaveProfile(HttpServletRequest* request);
    string ExtractQueryParam(HttpServletRequest* request, string param);
    void ReplyOK(HttpServletRequest* request, string html);
    void FindFriends(HttpServletRequest* request);
    void ReplyNotFound(HttpServletRequest* request, string error);
    void CloseDatabase(string location);
    ~FrontendServer();
};
```

特にひどいコードというわけではない。でも、メソッドの概要をすぐに把握できるような配置にはなっていない。すべてのメソッドを1つの大きなブロックにまとめのではなく、論理的なグループに分けてあげるといいだろう。例えば、こんな感じだ。

```
class FrontendServer {
public:
    FrontendServer();
    ~FrontendServer();

    // ハンドラ
    void ViewProfile(HttpServletRequest* request);
    void SaveProfile(HttpServletRequest* request);
    void FindFriends(HttpServletRequest* request);

    // リクエストとリプライのユーティリティ
    string ExtractQueryParam(HttpServletRequest* request, string param);
    void ReplyOK(HttpServletRequest* request, string html);
    void ReplyNotFound(HttpServletRequest* request, string error);
};
```

```
// データベースのヘルパー
void OpenDatabase(string location, string user);
void CloseDatabase(string location);
};
```

これで概要が把握しやすくなった。コード行は増えたけど、ずっと読みやすくなつたと思う。グループに分けたことで、最初に4つのグループを把握して、あとから必要になったときに詳細を読めるようになった。

4.7 コードを「段落」に分割する

文章は複数の段落に分割されている。それは、

- 似ている考えをグループにまとめて、他の考えと分けるためだ。
- 視覚的な「踏み石」を提供できるからだ。これがなければ、ページのなかで自分の場所を見失ってしまう。
- 段落単位で移動できるようになるからだ。

これと同じ理由で、コードも「段落」に分けるべきだ。例えば、以下のようなひと塊りのコードは誰も読む気がしない。

```
# ユーザのメール帳をインポートして、システムのユーザと照合する。
# そして、まだ友達になっていないユーザの一覧を表示する。
def suggest_new_friends(user, email_password):
    friends = user.friends()
    friend_emails = set(f.email for f in friends)
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends
    return render("suggested_friends.html", display)
```

わかりにくかもしれないけど、この関数はいくつかの手順で成り立っているので、

コードを段落に分割するといい。

```
def suggest_new_friends(user, email_password):
    # ユーザの友達のメールアドレスを取得する。
    friends = user.friends()
    friend_emails = set(f.email for f in friends)

    # ユーザのメールアカウントからすべてのメールアドレスをインポートする。
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)

    # まだ友達になっていないユーザを探す。
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email_in=non_friend_emails)

    # それをページに表示する。
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends

    return render("suggested_friends.html", display)
```

段落ごとに要約コメントを追加した。これでコードにざっと目を通せるようになった（「5章 コメントすべきことを知る」参照）。

文章と同じように、コードを分割する方法もいろいろある。長い段落が好きなプログラマもいれば、短い段落が好きなプログラマもいるだろう。

4.8 個人的な好みと一貫性

最終的には個人の好みになってしまうこともある。例えば、クラス定義の開き括弧の位置がそうだ。

```
class Logger {
    ...
};
```

または、

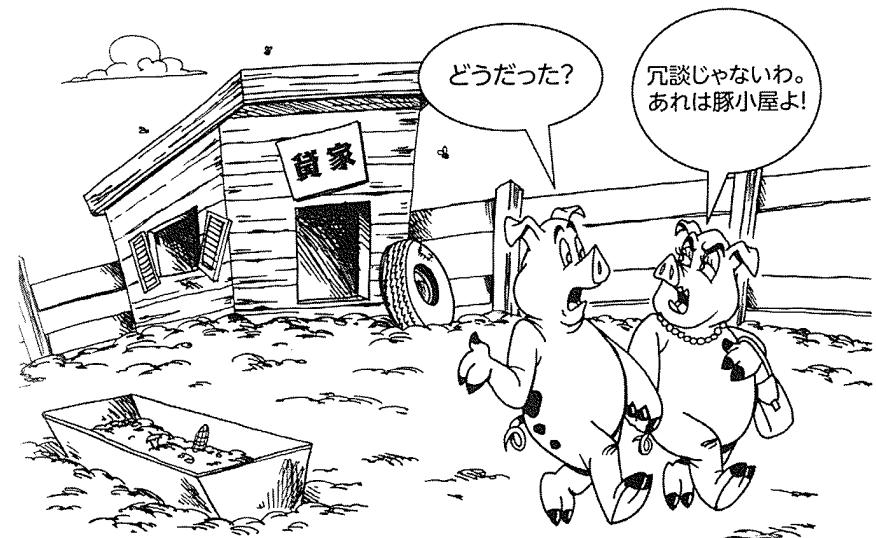
```
class Logger
{
    ...
};
```

どちらを選んだとしても、コードの読みやすさに大きな影響はない。でも、この2つのスタイルを混ぜてしまうと、すごく読みにくいものになってしまう。

ぱくたちは「間違った」スタイルを使っているプロジェクトに数多く携わってきた。でも、そこではプロジェクトの規約にしたがうようにした。一貫性のほうが大切なことだからだ。

鍵となる考え方

一貫性のあるスタイルは「正しい」スタイルよりも大切だ。



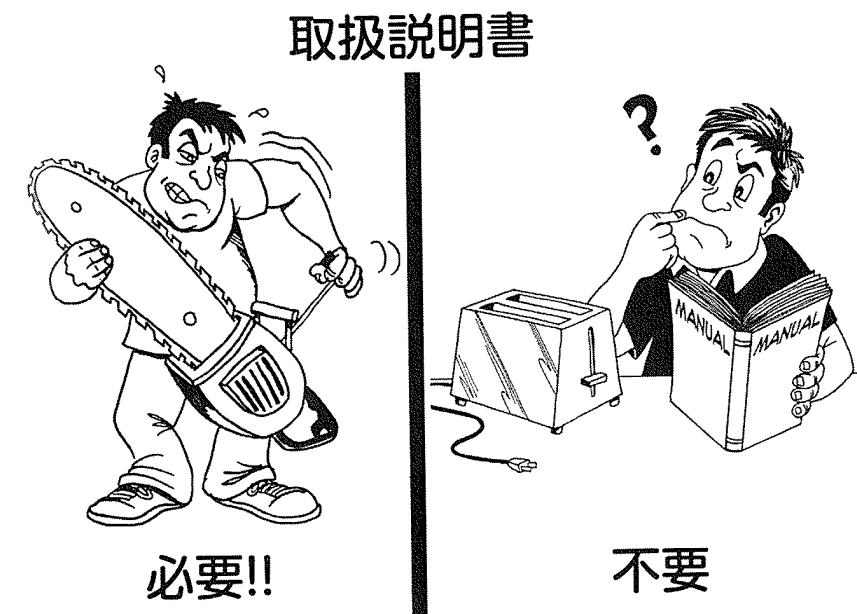
4.9 まとめ

誰もが美しいコードを見るのが好きだ。一貫性と意味のあるやり方でコードを「整形」すれば、すばやく簡単にコードを読むことができる。

ここで紹介した技法をまとめよう。

- 複数のコードブロックで同じようなことをしていたら、シルエットも同じようなものにする。
- コードの「列」を整列すれば、概要が把握しやすくなる。
- ある場所で A・B・C のように並んでいたものを、他の場所で B・C・A のように並べてはいけない。意味のある順番を選んで、常にその順番を守る。
- 空行を使って大きなブロックを論理的な「段落」に分ける。

5章 コメントすべきことを知る



本章の目標は、コメントすべきことを知ってもらうことだ。コメントの目的は「コードの動作を説明する」ことだと思っているかもしれない。でもそれは、目的のごく一部でしかない。

鍵となる考え方

コメントの目的は、書き手の意図を読み手に知らせることである。

コードを書いているときには、君の頭のなかに大切な情報がたくさんあると思う。でも、誰かが君のコードを読むときには、その情報は失われてしまう。コードを読む人が持っているのは、目の前にあるコードだけだ。

本章では、頭のなかにある情報をいつ書き出せばいいのかという例を数多く紹介している。コメントに関してよく言われるようなことには触れていない。もっと興味深くて「注目されていない」側面に集中した。

本章では、以下の話題を取り上げる。

- コメントするべきでは「ない」ことを知る。
- コードを書いているときの自分の考えを記録する。
- 読み手の立場になって何が必要になるかを考える。

5.1 コメントするべきでは「ない」こと



コメントを読むとその分だけコードを読む時間がなくなる。コメントは画面を占領してしまう。言い換えれば、コメントにはそれだけの価値を持たせるべきなんだ。それでは、価値のないコメントと価値のあるコメントの違いは何だろうか？

以下のコードに書かれたコメントには価値がない。

```
// Account クラスの定義
class Account {
    public:
        // コンストラクタ
        Account();
```

```
// profit に新しい値を設定する
void SetProfit(double profit);

// この Account から profit を返す
double GetProfit();
};


```

新しい情報を提供するわけでもなく、読み手がコードを理解しやすくなるわけでもない。全く価値がない。

鍵となる考え方

コードからすぐにわかるることをコメントに書かない。

この「すぐに」が大切だ。以下の Python のコードのコメントについて考えてみよう。

```
# 2番めの '*' 以降をすべて削除する
name = '*' .join(line.split('*')[2:])
```

厳密に言えばこのコメントも「新しい情報」を提供していない。コードを見ればどのように動くかはわかる。でも、コードを理解するよりも、コメントを読んだほうが早く理解できる。

コメントのためのコメントをしない



「宿題に出したコードの関数には必ずコメントをつける」 学生たちにこんなことを教う大学教授がいる。そんなことを言わされたら、コメントをつけていない裸の関数に罪悪感を抱き、関数の名前と引数をそのまま文章形式でコメントに書き直すようになってしまう。

```
// 与えられた subtree に含まれる name と depth に合致した Node を見つける。
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

これは「価値のないコメント」だ（関数宣言とはほぼ同じだ）。このコメントは削除するか改善すべきだろう。

コメントをつけたければ、もっと大切なことを説明したほうがいい。

```
// 与えられた 'name' に合致した Node か NULL を返す。
// もし depth <= 0 ならば、'subtree' だけを調べる。
// もし depth == N ならば、'subtree' とその下の N 階層まで調べる。
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

ひどい名前はコメントをつけずに名前を変える

コメントはひどい名前の埋め合わせに使うものではない。例えば、CleanReply()という関数につけたコメントがある。

```
// Reply に対して Request で記述した制限を課す。
// 例えば、返ってくる項目数や合計バイト数など。
void CleanReply(Request request, Reply reply);
```

このコメントは「クリーン (clean)」の意味をわかりやすく説明しているだけだ。「制限を課す (enforce limit)」という言葉を関数名に入れたほうがいい。

```
// 'reply' を 'request' にある項目数やバイト数の制限に合わせる。
void EnforceLimitsFromRequest(Request request, Reply reply);
```

この関数名は「自己文書化」されている。関数名はいろんなところで使用されるのだから、優れたコメントよりも名前のほうが大切だ。

ひどい名前の関数にコメントをつけた例をもう 1 つ挙げよう。

```
// レジストリキーのハンドルを解放する。実際のレジストリは変更しない。
void DeleteRegistry(RegistryKey* key);
```

`DeleteRegistry()` は危険な関数のように見える（レジストリを削除する！？）。でも、コメントにある「実際のレジストリは変更しない」は、危険な関数ではないと言っている。

以下のような自己文書化した名前をつけたほうがいいだろう。

```
void ReleaseRegistryHandle(RegistryKey* key);
```

通常は「補助的なコメント」（コードの読みにくさを補うコメント）が必要になることはない。プログラマはこのことを「優れたコード > ひどいコード + 優れたコメント」と言っている。

5.2 自分の考えを記録する

何をコメントすべきでないかはわかった。それじゃあ、何をコメントすべきかを考えていこう（みんな考えてないことが多いよね）。

優れたコメントというのは「考えを記録する」ためのものである。コードを書いているときに持っている「大切な考え方」のことだ。

「監督のコメント」を入れる

映画のDVDにはよく「監督のコメント」がついてくる。映画の製作者が自分の考え方や物語について語ってくれるので、作品がどのように作られたのかを理解するのに役立つ。これと同じように、コメントにはコードに対する大切な考え方を記録しなければいけない。

例えば、こんな感じだ。

```
// このデータだとハッシュテーブルよりもバイナリツリーのほうが40%速かった。
// 左右の比較よりもハッシュの計算コストのほうが高いようだ。
```

コメントから情報を得ることができるので、下手に最適化しようとして無駄に時間を使う必要がなくなる。

例は他にもある。

```
// ヒューリスティックだと単語が漏れることがあるが仕方ない。100%は難しい。
```

このコメントがなければ、失敗するテストケースに無駄な時間をかけることになるかもしれない。あるいは、バグだと思って修正したくなるだろう。
コードが汚い理由をコメントに書いてもいい。

```
// このクラスは汚くなってきてている。
// サブクラス 'ResourceNode' を作って整理したほうがいいかもしれない。
```

このコメントはコードが汚いことを認めている。そして、誰かに修正を促している（簡単な仕様書もついている）。コメントがなければ、コードが汚くて誰も近づかなかっただろう。

コードの欠陥にコメントをつける

コードは絶えず進化しているので、その過程で欠陥を生む運命にある。その欠陥を文書化することを恥ずかしがってはいけない。例えば、改善が必要なときは以下のように書いておこう。

```
// TODO: もっと高速なアルゴリズムを使う
```

コードが未完成のときは、以下のように書いておこう。

```
// TODO(ダステイン): JPEG以外のフォーマットに対応する
```

プログラマがよく使う記法がいくつかある。

記法	典型的な意味
TODO:	あとで手をつける
FIXME:	既知の不具合があるコード
HACK:	あまりキレイじゃない解決策
XXX:	危険！ 大きな問題がある

これらの記法をいつどのように使うかについては、チームの規約があるかもしれない。例えば、`TODO:` は大きな問題に使って、小さな問題には `todo:`（小文字）や

maybe-later: (あとで直す) を使うなどだ。

大切なのは、これからコードをどうしたいのかを自由にコメントに書くことだ。そういうコメントを書くことで、コードの品質や状態を知らせたり、さらには改善の方向を示したりできるのだ。

定数にコメントをつける

定数を定義するときには、その定数が何をするのか、なぜその値を持っているのかという「背景」が存在する場合が多い。例えば、コードの中で以下のような定数を見かけることがあるだろう。

```
NUM_THREADS = 8
```

このコードにはコメントは必要なさそうだ。でも、これを書いたプログラマには何か考えがあるかもしれない。

```
NUM_THREADS = 8 # 値は「>= 2 * num_processors」で十分。
```

これなら値の決め方がわかる（1だと小さすぎて、50だと大きすぎる）。値は厳密じゃなくてもいい場合もある。この場合もコメントが役に立つ。

```
// 合理的な限界値。人間はこんなに読めない。
const int MAX_RSS_SUBSCRIPTIONS = 1000;
```

値がいい感じに設定されているので、それ以上調整する必要がないこともある。

```
image_quality = 0.72; // 0.72ならユーザはファイルサイズと品質の面で妥協できる。
```

こんなコメントはつけないと思っているかもしれないけど、すごく役に立っている。

なかには名前が明確なのでコメントが必要ない定数もある（SECONDS_PER_DAYなど）。だけど、ぼくたちの経験からすると、コメントをつけて改善できる定数が多い。定数の値を決めたときに頭のなかで考えていたことを記録することが大切なのだ。

5.3 読み手の立場になって考える

本書で使っている技法は、他の人にコードがどのように見えるかを想像するものだ。「他の人」というのは、プロジェクトのことを君のように熟知していない人のことである。どこにコメントをつければいいかを判断するのにこの技法が役に立つ。

質問されそうなことを想像する



何か質問はありますか?
あの看板のこと以外で

他人のコードを読むと「えっ？ これって何なの？」と疑問に思うところがあるはずだ。こういうところにコメントをつければいい。

例えば、Clear() の定義を見てみよう。

```
struct Recorder {
    vector<float> data;
```

```

...
void Clear() {
    vector<float>().swap(data); // えっ？ どうして data.clear() じゃないの？
}

```

このコードを見た C++ プログラマは、「どうして単純に `data.clear()` せずに空のベクタをスワップするんだ？」と疑問に思うだろう。このようにしているのは、ベクタのメモリを解放してメモリアロケータに戻す方法がこれしかないからだ。これはあまり知られていないことである。つまり、ここにコメントをつけるべきなのだ。

```
// ベクタのメモリを解放する（「STL swap 技法」で検索してみよう）
vector<float>().swap(data);
```

ハマりそうな罠を告知する



関数やクラスを文書化するときには、「このコードを見てビックリすることは何だろう？ どんなふうに間違えて使う可能性があるだろう？」と自分に問いかけるといい。基本的にはコードを使うときに直面する問題を「前もって」予測したい。例えば、ユーザにメールを送信するコードを書いたとしよう。

```
void SendEmail(string to, string subject, string body);
```

この関数の実装では、外部のメールサービスに接続している。その接続には 1 秒以上かかる。このことを知らないウェブアプリケーション開発者が、HTTP リクエストの処理中に誤ってこの関数を呼び出してしまうかもしれない（メールサービスがダウンしていると、ウェブアプリケーションが「ハング」してしまう）。

このような不幸を防ぐには、「実装の詳細」についてコメントを書くべきだ。

```
// メールを送信する外部サービスを呼び出している（1分でタイムアウト）
void SendEmail(string to, string subject, string body);
```

以下はまた別の例だ。`FixBrokenHtml()` という関数があるとする。これは閉じタグのない壊れた HTML を修復するためのものだ。

```
def FixBrokenHtml(html): ...
```

この関数は間違いなく動く。ただし、対応の取れていないタグのネストが深いと処理に時間がかかるてしまう。HTML によっては何分もかかることがある。

ユーザが使った「あと」で気づくよりも、使う「前」に告知するほうがいい。

```
// 実行時間は O( タグの数 * タグの深さの平均 ) ので、ネストの深さに気を付ける。
def FixBrokenHtml(html): ...
```

「全体像」のコメント



新しいチームメンバーにとって、最も難しいのは「全体像」の理解である。クラスはどのように連携しているのか。データはどのようにシステムを流れているのか。エントリーポイントはどこにあるのか。システムを設計した人は、こうしたことについてコメントを書かないことが多い。あまりにも密接にシステムに関わりすぎているからだ。

こんな思考実験をしてみよう。新しくチームに参加した人がいるとする。彼女は君の隣に座っている。彼女にはコードに慣れてもらわなければいけない。

君は彼女にコードのことを教える。ファイルやクラスを指して、こんなことを言うだろう。

- 「これはビジネスロジックとデータベースをつなぐグルーコードです。アプリケーションから直接使ってはいけません」
- 「このクラスは複雑に見えますけど、単なるキャッシュです。システムのことは関知していません」

簡単な会話だけど、ソースコードを読んだだけでは得られない情報が手に入った。

これは高レベルのコメントに書くべき情報だ。

以下は、ファイルにつけたコメントの例だ。

```
// このファイルには、ファイルシステムに関する便利なインターフェースを提供
// するヘルパー関数が含まれています。ファイルのパーミッションなどを扱います。
```

なにも大量の正式文書を書けと言ってるわけではないので、ビビらないで欲しい。短い適切な文章で構わない。何もないよりはマシだ。

要約コメント

関数の内部でも「全体像」についてコメントするのはいい考えだ。低レベルのコードをうまく要約したコメントを紹介しよう。

```
# 顧客が自分で購入した商品を検索する
for customer_id in all_customers:
    for sale in all_sales[customer_id].sales:
        if sale.recipient == customer_id:
            ...
            ...
```

コメントがないとコードを読んでいる途中で意味がわからなくなる（「all_customers をイテレートするのはわかるけど……何のために？」）。

このような要約コメントは、関数の内部にある大きな「塊」につけてもいい。

```
def GenerateUserReport():
    # このユーザのロックを獲得する
    ...
    # ユーザの情報をDBから読み込む
    ...
    # 情報をファイルに書き出す
    ...
    # このユーザのロックを解放する
```

関数の処理を箇条書きでまとめたものなので、詳細を調べなくても関数の概要が把握できる。

握できるようになっている（塊を関数に分割できるならそうしよう。前にも言ったけど、ひどいコードに優れたコメントをつけるよりも、優れたコードのほうがいい）。

WHAT・WHY・HOWをコメントに書くべきか？

「コメントには WHAT ではなく（あるいは HOW ではなく） WHY を書こう」というアドバイスを耳にしたことがあるかもしれない。確かにキャッチャーな言葉だけど、少し単純化しちゃって、人によって受け取り方が違うかもしれない。

ぼくたちからのアドバイスはこうだ。「コードを理解するのに役立つものなら何でもいいから書こう」これなら、WHAT でも HOW でも WHY でも（あるいは 3 つ全部でも）書ける。

5.4 ライターズブロックを乗り越える

プログラマの多くはコメントを書きたがらない。コメントをうまく書くのは大変だと思っているからだ。こうした「ライターズブロック[†]」を乗り越えるには、とにかく書き始めるしかない。自分の考えていることをとりあえず書き出してみよう。生煮えであっても構わない。

例えば、ある関数を作っていて、「ヤバい。これはリストに重複があったら面倒なことになる」と思ったとする。それをそのまま書き出せばいい。

// ヤバい。これはリストに重複があったら面倒なことになる。

ね、簡単でしょう？ 変なコメントでもないよね（何もないよりはマシだ）。ただ、ちょっとあいまいな表現かもしれない。言い回しをもっと詳細な言葉に置き換えるといいだろう。

- 「ヤバい」は「注意：これには気を付けて」という意味だ。
- 「これ」は「入力を処理するコード」という意味だ。
- 「面倒なことになる」は「実装が難しくなる」という意味だ。

[†] 訳注 行き詰まってしまって、文章が書けないこと。

書き換えたコメントは以下のようになる。

// 注意：このコードはリストの重複を処理できません（実装が難しいので）。

コメントを書く作業は、3 つの簡単な手順に分解できる。

- 頭のなかにあるコメントをとにかく書き出す。
- コメントを読んで（どちらかと言えば）改善が必要なものを見つける。
- 改善する。

コメントを書くようになれば、手順 1 の品質が次第によくなっていく。最終的には、修正の必要がなくなるだろう。それに、早めにショットチューコメントを書いていれば、最後にまとめて大量のコメントを書くような事態に陥ることはない。

5.5 まとめ

コメントの目的とは、コードの意図を読み手に理解してもらうことである。本章では、コードについてほんやりと考えていたことをハッキリと理解して、実際に書き出すことをやってみた。

コメントすべきでは「ない」こと：

- コードからすぐに抽出できること。
- ひどいコード（例えば、ひどい名前の関数）を補う「補助的なコメント」。
コメントを書くのではなくコードを修正する。

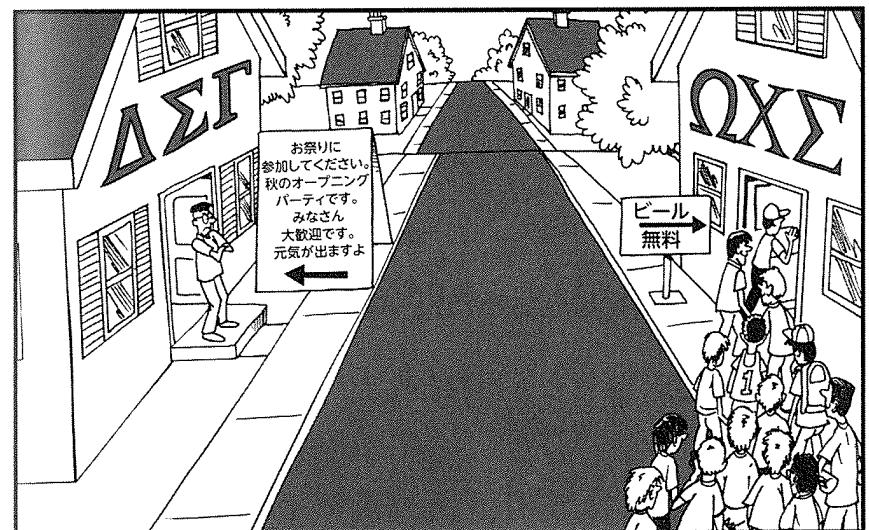
記録すべき自分の考え方：

- なぜコードが他のやり方ではなくこうなっているのか（「監督コメントリーリー」）。
- コードの欠陥を TODO: や XXX: などの記法を使って示す。
- 定数の値にまつわる「背景」。

読み手の立場になって考える：

- コードを読んだ人が「えっ？」と思うところを予想してコメントをつける。
- 平均的な読み手が驚くような動作は文書化しておく。
- ファイルやクラスには「全体像」のコメントを書く。
- 読み手が細部に捕らわれないように、コードブロックにコメントをつけて概要をまとめること。

6章 コメントは正確で簡潔に



前章では、何をコメントに書くべきかを説明した。本章では、どうすればコメントを正確で簡潔に書けるかを説明する。

コメントを書くのであれば、正確に書くべきだ（できるだけ明確で詳細に）。また、コメントには画面の領域を取られるし、読むのにも時間がかかるので、簡潔なものでなければいけない。

鍵となる考え方

コメントは領域に対する情報の比率が高くなればいけない。

これからそのやり方を見ていこう。

6.1 コメントを簡潔にしておく

以下は、C++ の型定義につけたコメントの例だ。

```
// int は CategoryType。  
// pair の最初の float は 'score'。  
// 2つめは 'weight'。  
typedef hash_map<int, pair<float, float> > ScoreMap;
```

どうして3行も使って説明しているのだろう？ これなら1行で説明できないだろうか？

```
// CategoryType -> (score, weight)  
typedef hash_map<int, pair<float, float> > ScoreMap;
```

3行分の領域が必要なこともあるだろう。でも、ここには必要ない。

6.2 あいまいな代名詞を避ける

古典漫才の「Who's on First?[†]」からもわかるように、代名詞は物事を複雑にしてしまう。

読み手は代名詞を「還元」しなければいけない。場合によっては、「それ」や「これ」が何を指しているのかよくわからないこともある。以下に例を挙げよう。

[†] 訳注 アボット&コステロの「一塁手は誰？」という漫才。一塁手の名前が「Who (誰・ダレ)」なので、「一塁手は誰なの?」「一塁手はダレだよ」「こっちが誰かと聞いてるんだよ」「だからダレだよ」……という会話が延々と繰り返される。

```
// データをキャッシュに入れる。ただし、先にそのサイズをチェックする。
```

「その」が指しているのは、「データ」かもしれないし「キャッシュ」かもしれない。コードを読み進めていけば、どちらを指しているのかはわかるだろう。でも、そうしないとわからないのであれば、何のためにコメントを書いているのだろう？

まぎらわしいようであれば、名詞を代名詞に「代入」してみるといい。先ほどの例であれば、「それ」を「データ」に変えてみるのだ。

```
// データをキャッシュに入れる。ただし、先にデータのサイズをチェックする。
```

これが最も簡単な改善だ。あるいは、文章全体を書き換えて「それ」を明確にすることもできる。

```
// データが十分に小さければ、それをキャッシュに入る。
```

6.3 歯切れの悪い文章を磨く

コメントを正確にすることと簡潔にすることは両立することが多い。

以下は、ウェブクローラの例だ。

```
# これまでにクロールした URL かどうかによって優先度を変える。
```

この文章は問題なさそうに見える。でも、以下と比較してみるとどうだろうか。

```
# これまでにクロールしていない URL の優先度を高くする。
```

こちらのほうが単純だし短いし直接的だ。さらには、クロールしていない URL の優先度が高いという、先のコメントでは言及されていない情報も含まれている。

6.4 関数の動作を正確に記述する

ファイルの行数を数える関数を書いているとしよう。

```
// このファイルに含まれる行数を返す。  
int CountLines(string filename) { ... }
```

あまり正確なコメントではない。「行」にはさまざまな意味があるからだ。コー

ナーケースを考えてみよう。

- "" (空のファイル) は、0 行なのか 1 行なのか。
- "hello" は、0 行なのか 1 行なのか。
- "hello\n" は、1 行なのか 2 行なのか。
- "hello\n world" は、1 行なのか 2 行なのか。
- "hello\n\r cruel\n world\r" は、2 行なのか 3 行なのか 4 行なのか。

最も単純な実装は、改行文字 (\n) を数えるものだ (Unix の wc コマンドと同じ方法だ)。この実装に適したコメントがこれだ。

```
// このファイルに含まれる改行文字 ('\n') を数える。
int CountLines(string filename) { ... }
```

最初のコメントと比較すると、それほど長くはないのに、伝わる情報は格段に増えている。改行文字がない場合は、0 を返すことがわかる。また、キャリッジリターン (\r) が無視されることもわかる。

6.5 入出力のコーナーケースに実例を使う

慎重に選んだ入出力の実例をコメントに書いておけば、それは千の言葉に等しいと言える。

例えば、文字列の一部を除去する関数があるとする。

```
// 'src' の先頭や末尾にある 'chars' を除去する。
String Strip(String src, String chars) { ... }
```

このコメントはあまり正確ではない。以下のような質問に答えることができないからだ。

- chars は、除去する文字列なのか、順序のない文字集合なのか？
- src の末尾に複数の chars があったらどうなるのか？

以上の質問に答えられる適切な実例はこうだ。

```
// ...
// 実例: Strip("abba/a/ba", "ab") は "/a/" を返す
String Strip(String src, String chars) { ... }
```

この実例は、Strip() のすべての機能を「見せて」いる。上記の質問に答えられないような簡単な実例では役に立たないので注意しよう。

```
// 実例: Strip("ab", "a") は "b" を返す
```

実例を使った関数をもう 1 つ紹介しよう。

```
// 'v' の「要素 < pivot」が「要素 >= pivot」の前に来るよう配置し直す。
// それから、「[v[i] < pivot]」になる最大の 'i' を返す（なければ -1 を返す）。
int Partition(vector<int>* v, int pivot);
```

このコメントは実に正確だ。でも、視覚化が少し難しい。実例を使ってより詳しく説明してみよう。

```
// ...
// 実例: Partition([8 5 9 8 2], 8) の結果は [5 2 | 8 9 8] となり、1 を返す。
int Partition(vector<int>* v, int pivot);
```

ぼくたちが選んだ実例には、大切な点がいくつかある。

- pivot をベクタの要素と同じにして、エッジケースを示している。
- ベクタに重複要素 (8) を入れることで、これが入力値として受け入れられるることを示している。
- 結果のベクタはソートしていない。ここでソートしていたら、ソートされているはずと読み手が誤解する可能性がある。
- 戻り値が 1 になるので、ベクタの要素に 1を入れていない。

6.6 コードの意図を書く

前章でも言ったけど、コメントというのはコードを書いているときに考えていたことを読み手に伝えるためのものだ。でも、コードの動作をそのまま書いているだけで、何も情報を追加していないコメントが多い。

そんなコメントを見てみよう。

```
void DisplayProducts(list<Product> products) {
    products.sort(CompareProductByPrice);

    // list を逆順にイテレートする
    for (list<Product>::reverse_iterator it = products.rbegin(); it != products.rend();
        ++it)
        DisplayPrice(it->price);

    ...
}
```

このコメントは直下のコードをそのまま説明しているだけだ。もっといいコメントを考えてみよう。

```
// 値段の高い順に表示する
for (list<Product>::reverse_iterator it = products.rbegin(); ... )
```

このコメントは、プログラムの動作を高レベルから説明している。プログラマがコードを書いたときに考えていたことに近い。

ところで、このプログラムにはバグがある！ `CompareProductByPrice` 関数（ここには書いていない）が、すでに値段の高い順にソートしているのだ。したがって、このコードは作者の意図に反したものになっている。

このことはつまり、2番めのコメントのほうが優れているということだ。最初のコメントは、バグのことはさておき、技術的には正しい（逆順にイテレートする）。でも、作者の意図（値段の高い順に表示する）が、実際のコードと矛盾していることに気づきやすいのは、2番めのコメントのほうである。ここではコメントが冗長検査の役割を果たしているわけだ。

究極的には、ユニットテストが最高の冗長検査になるだろう（「14章 テストと読みやすさ」参照）。だからといって、プログラムの意図を説明するコメントをつける

のが無駄になるわけではない。

6.7 「名前付き引数」コメント

以下のような関数呼び出しを目撃したとしよう。

```
Connect(10, false);
```

数値とブール値が渡されているけど、何のことだかよくわからない。

Python のような言語であれば、引数を名前付きで渡せる。

```
def Connect(timeout, use_encryption): ...
```

名前付き引数で関数を呼び出す

```
Connect(timeout = 10, use_encryption = False)
```

これは C++ や Java のような言語ではできない。でも、インラインコメントを使えば同じような効果が得られる。

```
void Connect(int timeout, bool use_encryption) { ... }
```

// 引数にコメントをつけて関数を呼び出す

```
Connect(*timeout_ms = */ 10, /* use_encryption = */ false);
```

最初の「名前」が `timeout` ではなく、`timeout_ms` になっていることに注意して欲しい。本来であれば、仮引数の名前を `timeout_ms` にすべきだけど、何らかの理由で変更できないのであれば、このようにして手っ取り早く名前を「改善」できる。

特にブール型の引数では、値の前に `/* name = */` を置くのが大切だ。値の後ろにあるとまぎらわしい。

// これはやってはいけない！

```
Connect( ... , false /* use_encryption */ );
```

// これもやってはいけない！

```
Connect( ... , false /* = use_encryption */ );
```

これでは、`false` が「暗号化する」のか「暗号化しない」のかよくわからない。

このようなコメントは必要ないことが多い。でも、よくわからない引数を説明する

ときには、こうするのが手っ取り早い（し簡潔である）。

6.8 情報密度の高い言葉を使う

プログラミングの経験が何年かあれば、同じ問題や解決策が何度も繰り返し登場することに気づいていると思う。こうしたパターンやイディオムを説明するための言葉や表現がある。このような言葉を使えば、コメントをもっと簡潔にできる。

例えば、以下のようなコメントがあったとしよう。

```
// このクラスには大量のメンバがある。同じ情報はデータベースにも保管されている。ただし、
// 速度の面からここにも保管しておく。このクラスを読み込むときには、メンバが存在してい
// るかどうかを先に確認する。もし存在していれば、そのまま返す。存在しなければ、データベー
// スから読み込んで、次回のためにデータをフィールドに保管する。
```

こうではなく、以下のように書けばいい。

```
// このクラスの役割は、データベースのキャッシュ層である。
```

他にも以下のようなコメントは、

```
// 所在地から余分な空白を除去する。それから「Avenue」を「Ave.」にするなどの整形を施す。
// こうすれば、表記がわずかに違う所在地でも同じものであると判別できる。
```

以下のようにできる。

```
// 所在地を正規化する（例："Avenue" -> "Ave."）。
```

このように多くの意味を含んだ言葉や表現が数多く存在する。例えば、「ヒューリスティック」・「ブルートフォース」・「ナイーブソリューション」などがそうだ。コメントが長くてくどいと感じたら、こうした表現を使えないか確かめよう。

6.9 まとめ

本章では、小さな領域にできるだけ多くの情報を詰め込んだコメントを書くことについて説明した。具体的なヒントを以下に挙げる。

- 関数の動作はできるだけ正確に説明する。
- コメントに含める入出力の実例を慎重に選ぶ。
- コードの意図は、詳細レベルではなく、高レベルで記述する。
- よくわからない引数にはインラインコメントを使う（例：Function(/* arg = */ ...)）。
- 多くの意味が詰め込まれた言葉や表現を使って、コメントを簡潔に保つ。

- 複数のものを指す可能性がある「それ」や「これ」などの代名詞を避ける。

条件やループなどの制御フローがないコードは読みやすい。他の場所に飛んだり枝分かれしたりするのは複雑なので、コードがすぐにわかりにくくなってしまう。本章では、コードの制御フローを読みやすくすることについて説明する。

鍵となる考え方

条件やループなどの制御フローはできるだけ「自然」にする。コードの読み手が立ち止まつたり読み返したりしないように書く。

7.1 条件式の引数の並び順

以下の2つのコードはどちらが読みやすいだろうか。

```
if (length >= 10)
```

または、

```
if (10 <= length)
```

ほとんどのプログラマは最初のほうが読みやすいと言うだろう。それでは、以下の2つならどうだろうか。

```
while (bytes_received < bytes_expected)
```

または、

```
while (bytes_expected > bytes_received)
```

これも最初のほうが読みやすい。それはなぜだろう？ どんな原則があるのだろう？ $a < b$ と $b > a$ のどちらが読みやすいかなんて、どうやって決めるのだろう？ ぼくたちが便利に使っている指針を教えよう。

左側	右側
「調査対象」の式。変化する。	「比較対象」の式。あまり変化しない。

この指針は英語の用法と合っている。英語で「もし君が1年間で10万ドル以上稼

げるならば」や「もし君が18歳以上ならば」と言うのは自然だ。でも、「もし18歳が君の年齢以下ならば」と言うのは不自然だ。

`while (bytes_received < bytes_expected)` が自然なのもこのためだ。`bytes_received` は調査対象の値なので、ループを実行するたびに増えていく。`bytes_expected` は比較対象の値なので、より「安定」したものである。

今も「ヨーダ記法」は便利なの？

プログラミング言語のなかには、ifの条件部分で代入できるものがある（例えば、CやC++などがそうだ。Javaではできない）。

```
if (obj = NULL) ...
```

これは以下のことを意図したバグであることが多い。

```
if (obj == NULL) ...
```

このようなバグを防ぐために、引数の並び順を変えるプログラマもいる。

```
if (NULL == obj) ...
```

こうすれば、==を間違えて=と書いたとしても、`if (NULL = obj)` がコンパイルできないので安全だ。

ただし、順序を逆にするとコードが不自然で読みにくくなる（ヨーダが「Not if anything to say about it I have[†]」と言うのと同じだ）。現代のコンパイラーは`if (obj = NULL)` と書くと警告を出してくれる。したがって、「ヨーダ記法」は過去のものになりつつあると言えるだろう。

まあが、でもいいくらい

[†] 訳注 バルバティーンとの会話に登場するセリフ。「今日でジェダイは死に絶える」「さてどうかのう。そう決めつけるのはまだ……早い」

7.2 if/else ブロックの並び順



if/else 文のブロックは、並び順を自由に変えることができる。例えば、以下のように書くのと、

```
if (a == b) {
    // 第1のケース
} else {
    // 第2のケース
}
```

以下のように書くのは同じことだ。

```
if (a != b) {
    // 第2のケース
} else {
    // 第1のケース
}
```

これまであまり深く考えなかったかもしれないけど、この並び順には優劣がある。

- 条件は否定形よりも肯定形を使う。例えば、if (!debug) ではなく、if (debug) を使う。
- 単純な条件を先に書く。if と else が同じ画面に表示されるので見やすい。
- 関心を引く条件や目立つ条件を先に書く。

この優劣は衝突することもあるので、そのときは自分で判断しなければいけない。でも、優先度は明確に決まることが多い。

例えば、URL にクエリパラメータ expand_all が含まれているかどうかを判断して、response を構築するウェブサーバがあるとしよう。

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

最初の行を見ると、expand_all のことが頭に浮かんでしまう。「ピンクの象のことを考えないように」と言われても、ピンクの象のことを考えてしまうのと同じだ。「ピンクの象」という独特な言葉が「考えないように」を吹き飛ばしてしまうのだ。

ここでは expand_all がピンクの象だ。関心を引く条件なので（それに肯定形なので）、これを先に処理しよう。

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

否定形の条件であっても、単純で関心や注意を引く場合もある。そういうときは、それを先に処理しよう。

```
if not file:
    # エラーをログに記録する
else:
    # ...
```

状況によって判断基準は変わってくるのだ。

まとめると、ここで挙げたようなことに注意して欲しい。if/else がおかしな順番にならないように気を付けよう。

7.3 三項演算子

C 言語などでは、条件 ? a : b という条件式が書ける。これは、if (条件) { a } else { b } を簡潔に書いたものだ。

読みやすさの点から言うと、これには議論の余地がある。支持者は、複数行が1行にまとまるのでいいと言う。反対者は、読みにくいしデバッガでステップ実行するのが難しいと言う。

三項演算子が読みやすくて簡潔な例を挙げよう。

```
time_str += (hour >= 12) ? "pm" : "am";
```

三項演算子を使わないと以下のようになる。

```
if (hour >= 12) {
    time_str += "pm";
} else {
    time_str += "am";
}
```

長くて冗長な感じがする。この場合は、三項演算子のほうが読みやすいだろう。でも、以下の式だと読みにくい。

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

これは、単純な2つの値から1つを選ぶようなものではない。それなのにこんな

コードを書くというのは、「何でも1行に収めたい」と思っているからだ。

鍵となる考え方

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。

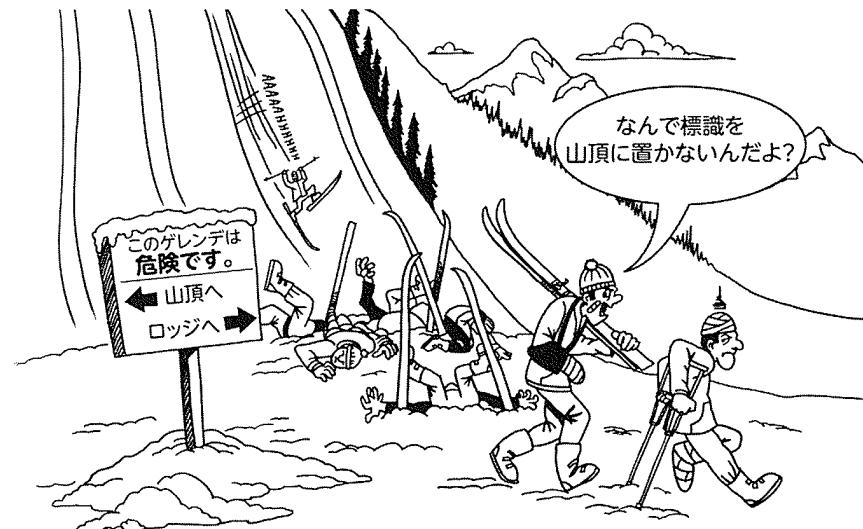
下手に省略せずにきちんと if/else 文を使えば、コードがより自然になる。

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

アドバイス

基本的に if/else を使おう。三項演算子はそれによって簡潔になるときにだけ使おう。

7.4 do/while ループを避ける



Perlなどの評判のいいプログラミング言語には、do { 式 } while (条件) ループがある。式の部分は最低1回は実行される。以下に例を挙げよう。

```
// 'name' に合致するものを 'node' のリストから探索する。
// 'max_length' を超えたノードは考えない。
public boolean ListHasNode(Node node, String name, int max_length) {
    do {
        if (node.name().equals(name))
            return true;
        node = node.next();
    } while (node != null && --max_length > 0);

    return false;
}
```

do/while ループが変わっているのは、コードブロックを再実行する条件が下にあることだ。if 文・while 文・for 文などの条件は、コードブロックの上にある。コードは上から下に読んでいくので、do/while は少し不自然だ。コードを 2 回読むことになってしまう。

while ループにすれば、コードブロックを読む前に繰り返しの条件がわかるので、読みやすくなる。でも、コードを重複させてまで do/while を削除するのはバカらしい。

```
// 模似 do/while (やってはダメ！)
本体

while (condition) {
    本体（2 度め）
}
```

do/while ループは、while ループで書き直せることが多い。

```
public boolean ListHasNode(Node node, String name, int max_length) {
    while (node != null && max_length-- > 0) {
        if (node.name().equals(name)) return true;
        node = node.next();
    }
    return false;
}
```

こうすれば、max_length が 0 でも、node が null でも動くようになる。

do/while を避ける理由は他にもある。それは、内部にある continue 文がまぎらわしいからだ。例えば、以下のコードは何をしているのだろう？

```
do {
    continue;
} while (false);
```

ループは永久に続くのだろうか？ それとも 1 回だけなのだろうか？ 多くのプログラマは、立ち止まって考えることになる（答えは、ループは 1 回だけになる）。

ビャーネ・ストロヴェルトupp (C++ の作者) は、著書『C++ Programming Language[†]』でこう言っている。

私の経験では、do-statement は、エラーや混乱の原因になることが多い。（中略）私は条件が「前もって」書かれている方が好きだ。そのため、私は do-statement を避けることが多い。

7.5 関数から早く返す

関数で複数の return 文を使ってはいけないと思っている人がいる。アホくさ。関数から早く返すのはいいことだ。むしろ望ましいときもある。例えば、

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;

    ...
}
```

このような「ガード節」を使わずに実装するとすごく不自然な実装になる。

関数の出口を 1 つにしたいというのは、何らかのクリーンアップコードを確実に実行したいからだろう。現代の言語では、こうした仕組みがより洗練された形で提供されている。

[†] 訳注 「プログラミング言語 C++ 第 3 版」(Bjarne Stroustrup 著、長尾高弘 訳、アジソンウェスレイパブリッシャーズジャパン)、p.178

言語	クリーンアップコードのイディオム
C++	デストラクタ
Java・Python	try ... finally
Python	with
C#	using

純粋な C 言語には、関数の終了時に特定のコードを呼び出すトリガー機能は存在しない。したがって、クリーンアップコードが大量に存在する大きな関数では、早めに返すのをうまくやるのは難しいかもしれない。ただし、関数をリファクタリングしたり、goto クリーンアップを慎重に使ったりするなどの方法がないわけでもない。

7.6 悪名高き goto

C 言語以外では、goto はほとんど必要ない。同じことをする方法が他にあるからだ。goto を使うとすぐに手に負えなくなったり、コードについていくのが難しくなったりするので、すごく評判が悪い。

でも、今でも C 言語のさまざまなプロジェクトで goto は使われている。例えば、Linux カーネルがそうだ。神への冒とくだと goto をはねつけるよりも、goto を使うべき理由を分析するほうがいいだろう。

最も単純で害のない goto というのは、関数の最下部に置いた exit と一緒に使うものだ。

```
if (p == NULL) goto exit;

...
exit:
fclose(file1);
fclose(file2);
...
return;
```

goto が唯一許されるのがこれだ。これなら goto は大した問題にならない。でも、goto の飛び先が複数になると問題だ。経路が交差していたらなおさらである。特に、goto から上に飛ぶのは本物のスパゲティコードになる。これは普通のル

ープに置き換える可能だ。基本的には goto は使わないほうがいいだろう。

7.7 ネストを浅くする

ネストの深いコードは理解しにくい。ネストが深くなると、読み手は「精神的スタック」に条件をブッシュしなければいけない。閉じ括弧 (}) を見てスタックからポップしようとしても、その条件が何だったのかうまく思い出せない。

以下は、比較的単純な例だ。自分がどの条件ブロックにいるのかを常に確認しているのがわかるだろうか。

```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();
```

最初の閉じ括弧を見つけると、「permission_result != SUCCESS が終わるんだな。次は、permission_result == SUCCESS か。でも、これはまだ user_result == SUCCESS の内部だな」と考える。

つまり、user_result と permission_result の値を常に覚えておかなければいけないということだ。それに、if {} ブロックが終了するたびに、覚えておいた値を反対にしなければいけない。

それから、このコードは SUCCESS と SUCCESS の否定が交互に登場しているので、もっとタチが悪い。

ネストが増える仕組み

先ほどのサンプルコードを修正する前に、どうしてこうなったかについて話しておこう。最初は単純なコードだった。

```

if (user_result == SUCCESS) {
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();

```

このコードなら完璧に理解できる。エラー文字列を決めてから、`reply` を終了しているだけだ。

ここに新しいコードが追加された。

```

if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
}

...

```

この変更には納得がいく。最も簡単にコードを挿入できる場所がここだったのだ。これを書いたプログラマにとって、新しく追加するコードは新鮮で「関心を引く」コードだった。そして、この変更の「差分」は何も間違っていない。簡潔な変更である。

でも、あとで誰かがこのコードを見たときには、こうした文脈はすべて失われてしまう。こうして、最初に見たコードができあがっていった。これを瞬時に把握できるだろうか。

鍵となる考え方

変更するときにはコードを新鮮な目で見る。一步下がって全体を見る。

早めに返してネストを削除する

それじゃあ、コードを改善していこう。ネストを削除するには「失敗ケース」をできるだけ早めに関数から返せばいい。

```

if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}

if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}

reply.WriteErrors("");
reply.Done();

```

これでネストの深さが2レベルから1レベルになった。もっと大切なのは、精神的なスタックから「ポップ」する必要がなくなったことだ。すべての `if` ブロックは `return` で終わっている。

ループ内部のネストを削除する

早めに返す技法はいつでも使えるわけではない。例えば、ループ内部でネストしたコードを見てみよう。

```

for (int i = 0; i < results.size(); i++) {
    if (results[i] != NULL) {
        non_null_count++;

        if (results[i]->name != "") {
            cout << "Considering candidate..." << endl;
            ...
        }
    }
}

```

早めに返すのと同じようなことをループ内部で行うには、`continue` を使う。

```

for (int i = 0; i < results.size(); i++) {
    if (results[i] == NULL) continue;
    non_null_count++;
}

```

```
if (results[i]->name == "") continue;
cout << "Considering candidate..." << endl;
```

```
}

...
```

`if (...) return;` が関数のガード節になるのと同じように、`if (...) continue;` 文がループのガード節になっている。

ただし、`continue` はわかりにくくなることが多い。`goto` と同じようにループを行ったり来たりするからだ。でも、このループは 1 つだけなので、`continue` が「この項目は飛ばす」意味だとすぐにわかる。

7.8 実行の流れを追えるかい？

スリーカードモンテ



本章では、低レベルの制御フローについて説明してきた。ループや条件などのジャンプを簡単に読めるようにする方法だ。でも、プログラムの高レベルの「流れ」についても考えなくてはいけない。できることならプログラムのすべての実行パスを簡単に追えるようになるといい。`main()` から出発して、心のなかでコードを追っていく。関数を次々に呼び出していく。それをプログラムが終了するまで続けるのだ。

ただし、プログラミング言語やライブラリには、コードを「舞台裏」で実行する構

成要素がある。こうした構成要素を使っていると、コードを追うのが難しくなる。いくつか例を挙げよう。

構成要素	高レベルの流れが不明瞭になる理由
スレッド	どのコードがいつ実行されるのかよくわからない。
シグナル／割り込みハンドラ	他のコードが実行される可能性がある。
例外	いろんな関数呼び出しが終了しようとする。
関数ポインタと無名関数	コンパイル時に判別できないので、どのコードが実行されるのかわからない。
仮想メソッド	<code>object.virtualMethod()</code> は未知のサブクラスのコードを呼び出す可能性がある。

これらの構成要素を使うことで、コードが読みやすくなったり、冗長性が低くなったりすることもある。でも、あとで理解しにくくなることを考えずに、調子に乗って使いすぎてしまうプログラマもいる。そうなると、バグを見つけるのが難しくなる。

コード全体に占める割合を大きくしないことが大切だ。こうした構成要素はうまく使わないと、(先ほどの漫画の) スリーカードモンテのように、コードの行方を見失ってしまう。

7.9 まとめ

コードの制御フローを読みやすくするために君ができることはいくつもある。

比較 (`while (bytes_expected > bytes_received)`) を書くときには、変化する値を左に、より安定した値を右に配置する (`while (bytes_received < bytes_expected)`)。

`if/else` 文のブロックは適切に並び替える。一般的には、肯定形・単純・目立つものを先に処理する。こうした基準は衝突することもあるけど、衝突がなければ基準を守っておこう。

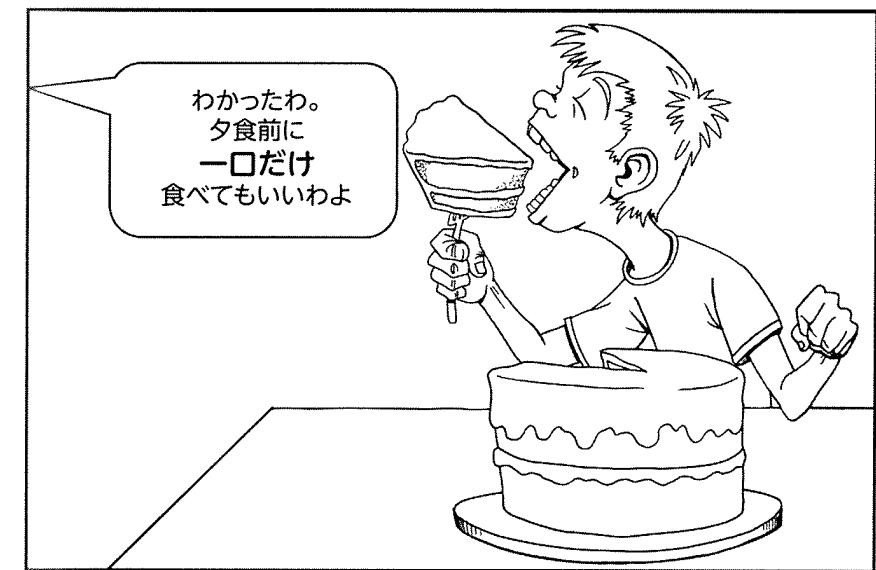
三項演算子 (`(?:)`)・`do/while` ループ・`goto` などのプログラミング構成要素を使うと、コードが読みにくくなることが多い。代替となるものが必ずあるので、これらは使わないほうがいいだろう。

ネストしているとコードを追うのに集中力が必要になる。ネストが増えるたびに「スタックにプッシュ」することが増える。深いネストを避けるには「直線的」な

コードを選択する。

早めに返してあげると、ネストを削除したりコードをクリーンにしたりできる。特に「ガード節」（関数の上部で単純な条件を先に処理するもの）が便利だ。

8章 巨大な式を分割する



ダイオウイカは驚くほど知的な生物である。しかし、その完璧に近い身体の構造には、1つの致命的な欠点がある。ドーナツ状の脳が食道を取り囲んでいるために、大量の食料を一度に摂取すると脳に損傷を受けてしまうのだ。

これをコードに置き換えるとどうなるだろう？ コードの「塊」が大きすぎると、周囲に悪影響を及ぼすことになる。最近の研究では、人間は一度に3～4の「もの」しか考えられないそうだ[†]。つまり、コードの式が大きくなれば、それだけ理解が難しくなるのである。

鍵となる考え方

巨大な式は飲み込みやすい大きさに分割する。

本章では、コードを飲み込みやすくするための処理や分割の方法を紹介する。

8.1 説明変数

式を簡単に分割するには、式を表す変数を使えばいい。この変数を「説明変数」と呼ぶこともある。式の意味を説明してくれるからだ。

例えば、以下のようなコードがあったとする。

```
if line.split(':')[0].strip() == "root":  
    ...
```

説明変数を使えば、以下のようになる。

```
username = line.split(':')[0].strip()  
if username == "root":  
    ...
```

8.2 要約変数

式を説明する必要がない場合でも、式を変数に代入しておくと便利だ。大きなコードの塊を小さな名前に置き換えて、管理や把握を簡単にする変数のことを要約変数と呼ぶ。

例えば、以下のコードの式を考えてみよう。

[†] Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24, 97-185.

```
if (request.user.id == document.owner_id) {  
    // ユーザはこの文書を編集できる  
}
```

...

```
if (request.user.id != document.owner_id) {  
    // 文書は読み取り専用  
}
```

request.user.id == document.owner_id はそれほど大きな式ではない。でも、変数が5つも入っているから、考えるのにちょっと時間がかかる。

このコードが言いたいのは「ユーザは文書を所持しているか？」だ。要約変数を追加すれば、この概念をもっと明確に表現できる。

```
final boolean user_owns_document = (request.user.id == document.owner_id);  
  
if (user_owns_document) {  
    // ユーザはこの文書を編集できる  
}
```

...

```
if (!user_owns_document) {  
    // 文書は読み取り専用  
}
```

if (user_owns_document) にしたら少しは考えやすくなった。また、user_owns_document を最上部に定義したこと、「この関数で参照する概念」を事前に伝えることができるようになった。

8.3 ド・モルガンの法則を使う

電気回路か論理学を学んだことがあれば、ド・モルガンの法則を覚えているかもしれない。論理式を等価な式に置き換える方法は2つある。

- 1) $\text{not } (a \text{ or } b \text{ or } c) \Leftrightarrow (\text{not } a) \text{ and } (\text{not } b) \text{ and } (\text{not } c)$
- 2) $\text{not } (a \text{ and } b \text{ and } c) \Leftrightarrow (\text{not } a) \text{ or } (\text{not } b) \text{ or } (\text{not } c)$

この法則が覚えてくれば、「not を分配して and/or を反転する」(逆方向は「not をくくりだす」と覚えればいい)。

この法則を使えば、論理式を読みやすくできる。例えば、以下のようなコードは、

```
if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");
```

以下のように書き直せる。

```
if (!file_exists || is_protected) Error("Sorry, could not read file.");
```

8.4 短絡評価の悪用

ブール演算子は短絡評価を行うものが多い。例えば、`if (a || b)` の `a` が `true` なら、`b` は評価されない。この動作はすごく便利だけど、悪用すると複雑なロジックになってしまう。

以下は、著者のひとりが過去に書いたコードの例だ。

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

これは「このキーのバケットを取得する。もしバケットが `null` じゃなかったら、中身が入っていないかを確認する」という意味だ。

たった1行のコードだけど、しばらく立ち止まって考えなきゃいけない感じだ。以下のコードと比較してみよう。

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

全く同じものだけど、こちらは2行になっている。コードは長くなつたけど、ずっと理解しやすくなった。

どうして1行で書こうとしたのだろう？ そのときは「オレは頭がいい」と思っていたのだ。ロジックを簡潔なコードに落とし込むことに一種の喜びを感じていた。この気持ちはみんなにも理解してもらえると思う。まるでパズルを解いているような感じだ。仕事は楽しくやりたいからね。問題は、これがコードのスピードバンプ[†]になっていたことだ。

[†] 訳注 車を減速させる路面の隆起のこと。ここでは、コードを読む速度を遅くさせるものという意味。

鍵となる考え方

「頭がいい」コードに気を付ける。あとで他の人がコードを読むときにわかりにくくなる。

これは短絡評価を避けろということだろうか？ それは違う。短絡評価は簡潔に使えることが多い。例えば、こんな感じだ。

```
if (object && object->method()) ...
```

他にも言っておきたいイディオムがある。Python・JavaScript・Rubyなどの言語では、複数の引数のなかから1つを返す「OR」演算子が使える（値がブール値になるわけではない）。例えば、以下のようなコードは、

x = a || b || c

`a · b · c` のなかから最初に「真」と評価できるものを選ぶ。

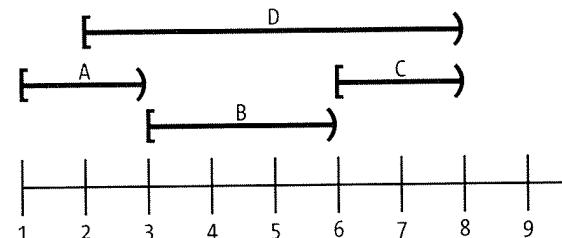
8.5 例：複雑なロジックと格闘する

Range クラスを実装しているとしよう。

```
struct Range {
    int begin;
    int end;

    // 例えば、[0,5] は [3,8] と重なっている。
    bool OverlapsWith(Range other);
};
```

以下の図は、範囲の例だ。

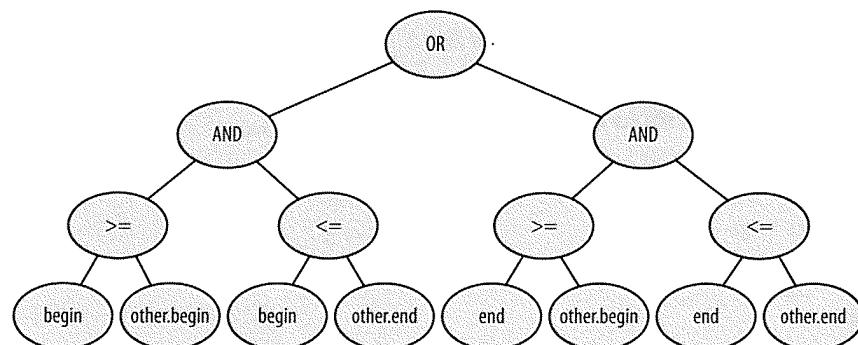


`end` は値を含まない。ここでは、`A`・`B`・`C` はお互いに重なっていないけど、`D` はすべてと重なっている。

いずれかの端が他の範囲と重なっているかを確認する `OverlapsWith()` を実装しているとしよう。

```
bool Range::OverlapsWith(Range other) {
    // 'begin' または 'end' が 'other' のなかにあるかを確認する。
    return (begin >= other.begin && begin <= other.end) ||
           (end >= other.begin && end <= other.end);
}
```

まだ 2 行しか書いていないけど、これからもっと増えていく。以下の図は、すべてのロジックを示したものだ。



場合分けや条件が多すぎて、バグを見逃しやすい。

というわけで、もうバグがある。先のコードでは、本当は重なっていない `Range[0,2)` と `[2,4)` が重なってしまうのだ。

`begin/end` の値を比較するときには、`<=` と `<` の使い分けに注意しなければいけない。この問題を修正したものがこちらだ。

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end);
```

これで正しくなったよね？ いやいや、まだ他にもバグがある。`begin/end` が他の範囲を取り囲んでいるケースが抜けている。

この処理を加えたコードがこちらだ。

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end) ||
       (begin <= other.begin && end >= other.end);
```

おっと、コードが複雑になってしまった。これでは誰もコードを読んでくれないし、これが絶対に正しいと自信を持って言うこともできない。それじゃあ、何をすればいいだろうか？ どうすればこの巨大な式を分割できるだろうか？

より優雅な手法を見つける

ここは立ち止まって、全く違った手法を考えてみるべきだ。最初は簡単な問題（2つの範囲の重なりを確認する）だったのに、驚くほど複雑なロジックのコードになってしまった。こういう場合には、もっと簡単な方法があるのだ。

そんな解決策を見つけるには創造性が必要だ。でも、どうすればいいのだろう？ 「反対」から問題を解決してみると手がある。例えば、配列を逆順にイテレートしてみる。データ後ろから挿入してみる。とにかくいつも反対のことをやってみるのだと。

`OverlapsWith()` の「反対」を考えると「重ならない」になる。2つの範囲が重ならないのは簡単だ。2つの場合しかない。

1. 一方の範囲の終点が、ある範囲の始点よりも前にある場合。

2. 一方の範囲の始点が、ある範囲の終点よりも後にある場合。

これをコードに置き換えるのは簡単だ。

```
bool Range::OverlapsWith(Range other) {
    if (other.end <= begin) return false; // 一方の終点が、この始点よりも前にある
    if (other.begin >= end) return false; // 一方の始点が、この終点よりも後にある
    return true; // 残ったものは重なっている
}
```

一度に 1 つずつしか比較していないので、コードがずっと単純になった。これで

<= が正しいかどうかを集中して見れるようになった。

8.6 巨大な文を分割する

本章では、式の分割について説明している。同じ技法は文の分割にも使える。例えば、以下の JavaScript のコードは、すぐには理解できない。

```
var update_highlight = function (message_num) {
    if ($("#vote_value" + message_num).html() === "Up") {
        $("#thumbs_up" + message_num).addClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    } else if ($("#vote_value" + message_num).html() === "Down") {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).addClass("highlighted");
    } else {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    }
};
```

それぞれの式はそれほど大きなものではないけど、それらがすべて一箇所に集まる巨大な文となって一齊に襲いかかってくる。

幸いなことに同じ式がいくつかあるので、それらを要約変数として関数の最上部に抽出すればいい（これは DRY 原則の実例である）。

```
var update_highlight = function (message_num) {
    var thumbs_up = $("#thumbs_up" + message_num);
    var thumbs_down = $("#thumbs_down" + message_num);
    var vote_value = $("#vote_value" + message_num).html();
    var hi = "highlighted";

    if (vote_value === "Up") {
        thumbs_up.addClass(hi);
        thumbs_down.removeClass(hi);
    } else if (vote_value === "Down") {
        thumbs_up.removeClass(hi);
        thumbs_down.addClass(hi);
    } else {
        thumbs_up.removeClass(hi);
```

```
        thumbs_down.removeClass(hi);
    }
};
```

var hi = "highlighted" の部分は厳密には不要だ。でも、同じものが 6 回も登場しているし、こうしたほうが便利なこともある。

- タイプミスを減らすのに役立つ（最初の例の 5 番めの文字列が "highlighted" とスペルミスしていたことに気づいていただろうか？）。
- 横幅が縮まるのでコードが読みやすくなる。
- クラス名を変更することになれば、一箇所を変更すればいい。

8.7 式を簡潔にするもう 1 つの創造的な方法

以下のコードの式も長い。今回は C++ のコードである。

```
void AddStats(const Stats& add_from, Stats* add_to) {
    add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
    add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
    add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
    add_to->set_status_string(add_from.status_string() + add_to->status_string());
    add_to->set_num_processes(add_from.num_processes() + add_to->num_processes());
    ...
}
```

どの式も長くてよく似ているけれど、同じものではなさそうだ。でも、10 秒ほどよく見ていれば、フィールド名が違うだけで、どの式も同じことをしていることに気づくだろう。

```
add_to->set_XXX(add_from.XXX() + add_to->XXX());
```

こういうときは、C++ ではマクロを定義すればいい。

```
void AddStats(const Stats& add_from, Stats* add_to) {
#define ADD_FIELD(field) add_to->set_##field(add_from.field() + add_to->field())
```

```
ADD_FIELD(total_memory);
ADD_FIELD(free_memory);
ADD_FIELD(swap_memory);
ADD_FIELD(status_string);
ADD_FIELD(num_processes);
...
#undef ADD_FIELD
}
```

これですべてのゴミをはぎ取ることができた。コードを見てすぐに本質が理解できる。それぞれの行が同じことをしているのも明確だ。

何もマクロを頻繁に使えと言っているわけじゃない。コードがわかりにくくなるし、見つけにくいバグが潜り込んでしまうこともある。ぼくたちはマクロを使うのを避けているほどだ。でも、今回のような場面では、簡潔で読みやすくなるという明確な利点がもたらされている。

8.8 まとめ

巨大な式を一度に理解しようと思うと難しい。本章では、巨大な式を分割して、読み手が1つずつ飲み込めるようにする方法を説明した。

最も簡単な方法は「説明変数」を導入することだ。大きな式の値を保持する説明変数には、3つの利点がある。

- 巨大な式を分割できる。
- 簡潔な名前で式を説明することで、コードを文書化できる。
- コードの主要な「概念」を読み手が認識しやすくなる。

その他には、ド・モルガンの法則を使ってロジックを操作する方法がある。これは論理式をキレイに書き直すことにも使える（例えば、`if (!(a && !b))` は、`if (!a || b)` になる）。

複雑な論理条件は「`if (a < b) ...`」のような小さな文に分割した。本章で取り上げたすべての改善コードには、`if` 文の中身が2行以上含まれていない。これは理想的な状況だ。同じことが常にできるとは限らない。そんなときは、問題を「否定」したり、反対のことを考えてみたりすることが必要になる。

最後に、本章の分割は式を対象にしたものだったけど、巨大なコードブロックにも同じ技法が使える。複雑なロジックを見かけたら、積極的に分割して欲しい。

本章では、変数を適当に使うとプログラムが理解しにくくなるという話をしよう。

具体的には、以下の3つの問題に取り組むことになる。

1. 変数が多いと変数を追跡するのが難しくなる。
2. 変数のスコープが大きいとスコープを把握する時間が長くなる。
3. 変数が頻繁に変更されると現在の値を把握するのが難しくなる。

これらの問題にどう対処するかを議論していこう。

9.1 変数を削除する

「8章 巨大な式を分割する」では、「説明変数」や「要約変数」を使ってコードを読みやすくした。なぜ読みやすくなったのかというと、変数が巨大な式を分割して、説明文のようになったからである。

本節では、コードが読みやすくならない変数を削除する。こうした変数を削除すれば、コードは簡潔で理解しやすいものになる。

このような不要な変数が使われている例をこれから見ていく。

役に立たない一時変数

以下のPythonコードにある変数nowを考えてみよう。

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

このnowを使う意味はあるだろうか？ 意味がない理由を以下に挙げよう。

- 複雑な式を分割していない。
- より明確になっていない。datetime.datetime.now()のままでも十分に明確だ。
- 一度しか使っていないので、重複コードの削除になっていない。

nowがなくても楽に理解できる。

```
root_message.last_view_time = datetime.datetime.now()
```

nowのような変数は、コードを編集した「残骸」だ。変数nowも最初は複数の場所で使われていたのだろう。あるいは、何度もnowを使おうと思っていたのに、一度しか使う機会がなかったのかもしれない。

中間結果を削除する



以下は、配列から値を削除するJavaScriptの関数の例だ。

```
var remove_one = function (array, value_to_remove) {
  var index_to_remove = null;
  for (var i = 0; i < array.length; i += 1) {
    if (array[i] === value_to_remove) {
      index_to_remove = i;
      break;
    }
  }
  if (index_to_remove !== null) {
    array.splice(index_to_remove, 1);
  }
};
```

変数index_to_removeは、中間結果を保持するためだけに使っている。結果そのまま使えば、このような変数は削除できる。

```
var remove_one = function (array, value_to_remove) {
  for (var i = 0; i < array.length; i += 1) {
```

```

    if (array[i] === value_to_remove) {
        array.splice(i, 1);
        return;
    }
};


```

関数から早く返すことで、`index_to_remove` をすべて削除できた。これでコードがずっと簡潔になった。

タスクはできるだけ早く完了するほうがいい。

制御フロー変数を削除する

ループでこんなコードを見かけることがある。

```

boolean done = false;

while /* 条件 */ && !done) {
    ...

    if (...) {
        done = true;
        continue;
    }
}

```

この変数 `done` は、ループのいろんなところで `true` に設定されている。

このようなコードは、「ループの途中から抜け出してはいけない」という暗黙的なルールを守ろうとしているのだろう。だが、そんなルールなど存在しない！

`done` のような変数のことをぼくたちは「制御フロー変数」と呼んでいる。これはプログラムの実行を制御するためだけの変数であり、実際のプログラムに関係のあるデータは含まれていない。ぼくたちの経験からすると、うまくプログラミングすれば、制御フロー変数は削除できる。

```

while /* 条件 */ {
    ...
    if (...) {
        break;
    }
}

```

```

    }
}

```

これなら修正は簡単だ。でも、`break` が使えないようなネストが何段階もあるループはどうすればいいのだろう？ そうした複雑な場合には、コード（ループの内部のコードやループ全体）を新しい関数に移動するといい。

四六時中、面接を受けるっていうのはどういう気持ち？

Microsoft 社のエリック・ブレックナーは、面接の質問には少なくとも 3 つの変数がなければいけないと正在している[†]。3 つの変数を同時に処理するには、しっかりと考えなければいけないからだろう！ これが面接ならいい。候補者を追い込むのが目的だからだ。でも、君のコードを同僚が読んでいるときに、面接を受けているような気分になるとしたらどうだろう。そんな気持ちにさせたいだろうか？

9.2 変数のスコープを縮める

「グローバル変数は避ける」というアドバイスは、誰もが一度は耳にしたことがあるはずだ。これは素晴らしいアドバイスである。グローバル変数というのは、どこでどのように使われるのかを追跡するのが難しい。また、「名前空間を汚染する」（ローカル変数と衝突する可能性がある）ことから、ローカル変数を使っているつもりでグローバル変数を修正したり、グローバル変数を使っているつもりでローカル変数を修正したりしてしまう。

グローバル変数に限らず、すべての変数の「スコープを縮める」のはいい考えだ。

鍵となる考え方

変数のことが見えるコード行数をできるだけ減らす。

多くのプログラミング言語には、スコープやアクセスのレベルが複数用意されている。例えば、モジュール・クラス・関数・ブロックスコープなどがそうだ。アクセスはできるだけ制限して、変数のことが「見えてしまう」コードを減らすのがいいとされている。

[†] Eric Brechner's I. M. Wright's "Hard Code" (Microsoft Press, 2007), p. 166.

では、なぜそうするのがいいとされているのだろう？ それは、一度に考えなければならない変数を減らせるからだ。すべての変数のスコープを1/2に縮めることができれば、スコープに存在する変数の数は平均して1/2になる。

例えば、巨大なクラスがあるとしよう。メンバ変数は2つのメソッドから使用されている。

```
class LargeClass {
    string str_;

    void Method1() {
        str_ = ...;
        Method2();
    }

    void Method2() {
        // str_ を使っている
    }

    // str_ を使っていないメソッドがたくさんある
};
```

メンバ変数というのは、クラスのなかで「ミニグローバル」になっているとも言える。大きなクラスでは、すべてのメンバ変数を追跡したり、どのメソッドが変数を変更しているかを把握したりするのは難しい。したがって、ミニグローバルはできるだけ減らしたほうがいい。

この場合は、`str_`をローカル変数に「格下げ」するといいかもしれない。

```
class LargeClass {
    void Method1() {
        string str = ...;
        Method2(str);
    }

    void Method2(string str) {
        // str を使っている
    }

    // その他のメソッドは str が見えない。
};
```

クラスのメンバへのアクセスを制限するもう1つの方法は、メソッドができるだけ `static` にすることだ。`static` メソッドを使えば「メンバ変数とは関係ない」とことがよくわかる。

もう1つの方法は、大きなクラスを小さなクラスに分割することだ。ただし、分割後のクラスが独立していれば問題ないけど、クラスで相互にメンバを参照し合うようならやっても意味がない。

このことは、大きなファイルを小さなファイルに分割したり、大きな関数を小さな関数に分割したりするときも同じだ。分割したいのはデータ（つまり、変数）なのである。

ただし、言語によってスコープの決め方は違う。変数などのスコープに関する興味深い規則をいくつか紹介しよう。

C++ の if 文のスコープ

以下のような C++ のコードがあるとしよう。

```
PaymentInfo* info = database.ReadPaymentInfo();
if (info) {
    cout << "User paid: " << info->amount() << endl;
}
```

// 以下、コードが続く

変数 `info` は関数のスコープ内にあるので、またいつどのように使われるかを考えながらコードを読まなければいけない。

変数 `info` が必要なのは、`if` 文のなかだけである。このような場合には、C++ の条件式で変数 `info` を定義すればいい。

```
if (PaymentInfo* info = database.ReadPaymentInfo()) {
    cout << "User paid: " << info->amount() << endl;
}
```

こうしておけば、スコープをすぎたら `info` を忘れることができる。

JavaScript で「プライベート」変数を作る

承認的な変数があるとしよう。これは1つの関数でしか使われていない。

```
submitted = false; // 注意: グローバル変数

var submit_form = function (form_name) {
    if (submitted) {
        return; // 二重投稿禁止
    }
    ...
    submitted = true;
};
```

`submitted`のようなグローバル変数は、コードを読む人を不安にさせる。`submitted`を使っているのは `submit_form()` 関数だけのように見えるけど、本当にそういうのかはよくわからない。他のファイルからグローバル変数 `submitted`を使っていける可能性だってある。しかも、別の目的で！

この問題を回避するには、変数 `submitted` をクロージャで包んであげればいい。

```
var submit_form = (function () {
    var submitted = false; // 注意: 以下の関数からしかアクセスされない

    return function (form_name) {
        if (submitted) {
            return; // 二重投稿禁止
        }
        ...
        submitted = true;
    };
}());
```

最終行にある括弧に注目して欲しい。外側の無名関数がすぐに実行されて、内側の関数を返している。

この技法をはじめて見たのであれば、最初は奇妙に見えるかもしれない。これは、内側の関数だけがアクセスできる「プライベート」スコープを作る効果がある。これで読み手は「`submitted`はいつ使われるの？」と気になったり、同じ名前のグローバル変数と衝突しないかと不安になったりすることはない（他にも同様の技法が『JavaScript: The Good Parts』（Douglas Crockford, O'Reilly, 2008）[†]に載っている）。

[†] 訳注：『JavaScript: The Good Parts——「良いパート』によるベストプラクティス』（ダグラス・クロックフォード著、水野貴明訳、オライリー・ジャパン）

JavaScript のグローバルスコープ

JavaScript では、変数の定義に `var` をつけるないと（例えば、`var x = 1`じゃなくて `x = 1` になると）、その変数はグローバルスコープに入ってしまう。グローバルスコープに入ると、すべての JavaScript ファイルや `<script>` ブロックからアクセスできてしまう。以下に例を挙げよう。

```
<script>
    var f = function () {
        // 危険: 'i' は 'var' で宣言されていない!
        for (i = 0; i < 10; i += 1) ...
    };

    f();
</script>
```

このコードでは、意図せずに変数 `i` をグローバルスコープに入れている。したがって、他のブロックからも変数が見えてしまう。

```
<script>
    alert(i); // '10' が表示される。'i' はグローバル変数なのだ!
</script>
```

このスコープの規則のことを知らないプログラマが多い。そして、この驚くべき振る舞いによって、奇妙なバグが生み出されている。よくあるのは、2つの関数で `var` のない同じ名前のローカル変数を作ってしまうことだ。これだと2つの関数が「混線」してしまう。未熟なプログラマであれば、コンピュータのプロセッサやメモリが壊れたと思い込んでしまうだろう。

JavaScript の「ベストプラクティス」は、「変数を定義するときには常に `var` キーワードをつける（例：`var x = 1`）」だ。このプラクティスを使えば、変数のスコープをその変数が定義された（最も内側の）関数に制限してくれる。

Python と JavaScript のネストしないスコープ

C++ や Java のような言語にはブロックスコープがある。`if`・`for`・`try`などのブロックで定義された変数は、スコープがそのブロックに制限されるのだ。

```

if (...) {
    int x = 1;
}
x++; // コンパイルエラー！'x' は未定義です。

```

Python や JavaScript では、ブロックで定義された変数はその関数全体に「こぼれる」。例えば、以下の Python のコードにある変数 `example_value` に注目しよう。

```

# ここまで example_value を使っていない。
if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break

for logger in debug.loggers:
    logger.log("Example:", example_value)

```

このスコープ規則に驚くプログラマは多い。それに、このようなコードは読みにくい。その他の言語では、`example_value` が定義された場所はすぐに見つかる。関数の「左端」を見ていいからだ。

先ほどの例にはバグもある。`example_value` に値が設定されていなければ、最後のところで「`NameError: 'example_value' is not defined`」という例外が発生する。`example_value` を使っている場所の「最も近い共通の祖先（ネスト的な意味で）」で変数を定義すれば、この問題を解決できるし、コードも読みやすくなる。

```

example_value = None

if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break

if example_value
    for logger in debug.loggers:
        logger.log("Example:", example_value)

```

ただし、この `example_value` は完全に削除することもできる。`example_value` は中間結果を保持しているだけなので、「中間結果を削除する」で見たように、「タスクができるだけ早く完了」すればいい。この場合は、値を見つけたらすぐにログに書き込むようにする。

新しいコードは以下のようになる。

```

def LogExample(value):
    for logger in debug.loggers:
        logger.log("Example:", value)

if request:
    for value in request.values:
        if value > 0:
            LogExample(value) # すぐに 'value' を使う
            break

```

定義の位置を下げる

元々の C 言語では、関数やブロックの先頭で変数を定義する必要があった。長い関数の中で変数をたくさん使っていると、ずっとあとにしか使わないとしても、すべての変数を先頭で定義しなければいけなかったのだ（C99 と C++ にこの制約はない）。

以下の例では、すべての変数が関数の先頭にだらだらと定義されている。

```

def ViewFilteredReplies(original_id):
    filtered_replies = []
    root_message = Messages.objects.get(original_id)
    all_replies = Messages.objects.select(root_id=original_id)

    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()

    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)

    return filtered_replies

```

このコードの問題点は、常に3つの変数のことを切り替えて考えなければいけないことだ。

最初からすべての変数を知る必要はないのだから、変数の定義は変数を使う直前に移動すればいい。

```
def ViewFilteredReplies(original_id):
    root_message = Messages.objects.get(original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()

    all_replies = Messages.objects.select(root_id=original_id)
    filtered_replies = []
    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)

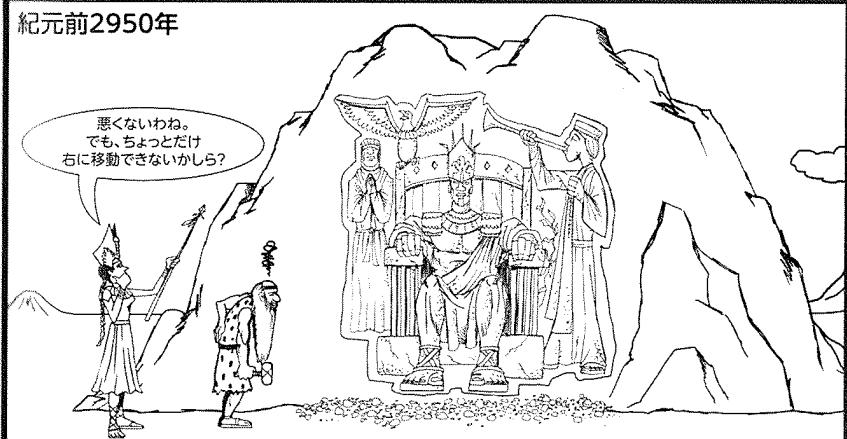
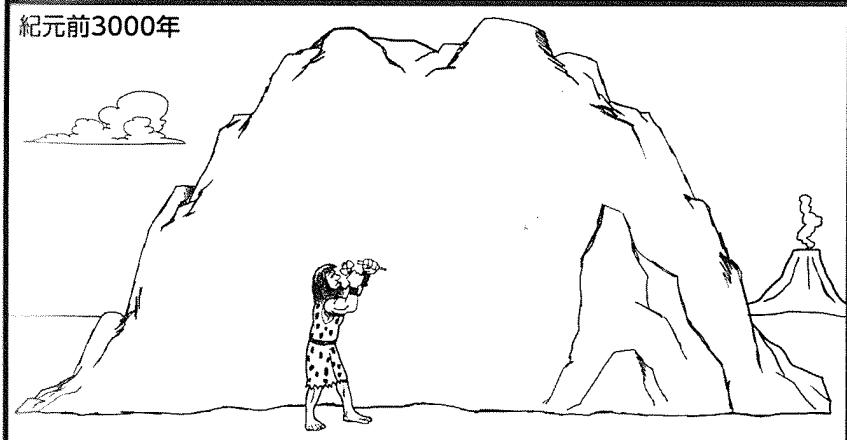
    return filtered_replies
```

変数 `all_replies` は必要ないので、以下のように削除できると思ったかもしれない。

```
for reply in Messages.objects.select(root_id=original_id):
```

この場合は、`all_replies` が優れた説明変数になっているので、そのままにしておこう。

9.3 変数は一度だけ書き込む



本章では、「生きている」変数が多いとコードが理解しにくくなることを説明した。でも、もっと理解しにくいのは、変数が絶えず変更され続けることだ。値を追跡する難易度が格段に上がってしまう。

この問題と戦うために、ちょっと変わったものを提案したい。それは、変数は一度だけ書き込むというものだ。

「永続的に変更されない」変数は扱いやすい。例えば、以下のような定数は、

```
static const int NUM_THREADS = 10;
```

多くのことを考える必要がない。これと同じ理由で、C++ の `const` (や Java の `final`) は是非とも使うべきだ。

実際、多くの言語 (Python や Java など) では、`String` などの組み込み型はイミュータブルになっている。ジェームズ・ゴスリング (Java の作者) が言うには、「(イミュータブルは) トラブルになる傾向が少ない」そうだ。

変数を一度だけ書き込む手法が使えないとしても、変数の変更箇所はできるだけ少なくしたほうがいいだろう。

鍵となる考え方

変数を操作する場所が増えると、現在値の判断が難しくなる。

それでは、変数を一度だけ書き込むにはどうすればいいだろう？ 以下の例のように、少しづつ何度もコードを再構築することになるだろう。

9.4 最後の例

本章の最後の例として、これまでに紹介した原則を使ったものを示そう。以下のように配置された入力テキストフィールドがウェブページにあるとする。

```
<input type="text" id="input1" value="Dustin">
<input type="text" id="input2" value="Trevor">
<input type="text" id="input3" value="">
<input type="text" id="input4" value="Melissa">
...

```

ご覧のように、`id` は `input1` から始まって 1 ずつ増加している。

君の仕事は、`setFirstEmptyInput()` という関数を書くことだ。この関数は、文字列を受け取って、ウェブページにある最初の空の `<input>` に入力するというものだ (この例では「`input3`」)。関数の戻り値は、更新した DOM 要素 (空の入力フィールドがなければ `null`) になる。以下に、本章の原則を適用していないコードがある。

```
var setFirstEmptyInput = function (new_value) {
    var found = false;
    var i = 1;
```

```
var elem = document.getElementById('input' + i);
while (elem !== null) {
    if (elem.value === '') {
        found = true;
        break;
    }
    i++;
    elem = document.getElementById('input' + i);
}
if (found) elem.value = new_value;
return elem;
};
```

このコードは動くけど、あまりキレイじゃない。どこがダメで、どうすれば改善できるだろうか？ 考え方はいろいろあると思うけど、ここでは変数から考えていこう。

- var found
- var i
- var elem

この 3 つの変数は関数のなかで何度も書き換えられている。これらの変数の使い方を改善してみよう。

本章の最初で説明したように、`found` などの中間変数は早めに返せば削除できる。以下が改善結果だ。

```
var setFirstEmptyInput = function (new_value) {
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    return null;
};
```

次に、elemを見ていこう。いろんなところで「ループ状」に何度も使用されていて、値を追いかけるのが難しくなっている。elemはiに合わせてイテレートしていくように見える。だから、whileをforループに書き換えてみよう。

```
var setFirstEmptyInput = function (new_value) {
    for (var i = 1; true; i++) {
        var elem = document.getElementById('input' + i);
        if (elem === null)
            return null; // 検索失敗。空の入力フィールドは見つからなかった。

        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
    }
};
```

変数elemがループ内で一度だけ書き込まれていることに注目して欲しい。forループの条件にtrueを使うことはあまりないかもしれないけど、これによってiの定義と増加が1行で書けるようになった（伝統的なwhile(true)でも大丈夫だ）。

9.5 まとめ

本章では、プログラムの変数はすぐに増えるので、いずれ追跡できなくなるという話をした。変数を減らして、できるだけ「軽量」にすれば、コードは読みやすくなる。具体的には、

- 邪魔な変数を削除する。——本章では、結果をすぐに使って、「中間結果」の変数を削除する例を示した。
- 変数のスコープをできるだけ小さくする。——変数を数行のコードからしか見えない位置に移動する。「去る者は日々に疎し」って言うからね（いい意味で）。
- 一度だけ書き込む変数を使う。——変数に一度だけ値を設定すれば（あるいは、constやfinalなどのイミュータブルにする方法を使えば）、コードが理解しやすくなる。

第 III 部 コードの再構成

第Ⅱ部では、コードを読みやすくするために、プログラムの「ループとロジック」を変更する方法を説明した。そして、プログラムの構造を少しだけ変更する技法をいくつか紹介した。

第Ⅲ部では、コードを大きく変更する技法を紹介する。具体的には、コードを再構成する3つの方法について説明する。

- プログラムの主目的と関係のない「無関係の下位問題」を抽出する。
- コードを再構成して、一度に1つのことをやるようにする。
- 最初にコードを言葉で説明する。その説明を元にしてキレイな解決策を作る。

最後に、コードを完全に削除できる状況について説明する。また、コードを書かずには済ませる状況についても説明する（コードを理解しやすくするには、コードを書かないのがいちばんだ）。

エンジニアリングとは、大きな問題を小さな問題に分割して、それぞれの解決策を組み立てるに他ならない。この原則をコードに当てはめれば、堅ろうで読みやすいコードになる。

本章のアドバイスは、無関係の下位問題を積極的に見つけて抽出することだ。ぼくたちは以下のことを考えている。

1. 関数やコードブロックを見て「このコードの高レベルの目標は何か？」と自問する。
2. コードの各行に対して「高レベルの目標に直接的に効果があるのか？ あるいは、無関係の下位問題を解決しているのか？」と自問する。
3. 無関係の下位問題を解決しているコードが相当量あれば、それらを抽出して別の関数にする。

コードを抽出して別の関数にするなんて毎日やっていることかもしれないけど、本章では無関係の下位問題を抽出する場合に限定している。このようにして抽出されたコードは、自分がアプリケーションからどのように呼び出されるのかわかっていない。

この技法は簡単に使えるのに、コードを大幅に改善できる。だけど、なぜだか知らないけど、多くのプログラマがうまく使いこなせていない。うまく使うコツは、無関係の下位問題を積極的に探し出すことだ。

本章では、君がこれから直面するさまざまな状況に対して、この技法を適用した例をいくつか紹介していく。

10.1 入門的な例：findClosestLocation()

以下に JavaScript のコードがある。このコードの高レベルの目標は「与えられた地点から最も近い場所を見つける」ことだ（斜体のところで難しい幾何学の計算ができるけど怖がらないで）。

```
// 与えられた緯度経度に最も近い 'array' の要素を返す。
// 地球が完全な球体であることを前提としている。
var findClosestLocation = function (lat, lng, array) {
  var closest;
```

```
var closest_dist = Number.MAX_VALUE;
for (var i = 0; i < array.length; i += 1) {
  // 2つの地点をラジアンに変換する。
  var lat_rad = radians(lat);
  var lng_rad = radians(lng);
  var lat2_rad = radians(array[i].latitude);
  var lng2_rad = radians(array[i].longitude);

  // 「球面三角法の第二余弦定理」の公式を使う。
  var dist = Math.acos(Math.sin(lat_rad) * Math.sin(lat2_rad) +
    Math.cos(lat_rad) * Math.cos(lat2_rad) *
    Math.cos(lng2_rad - lng_rad));
  if (dist < closest_dist) {
    closest = array[i];
    closest_dist = dist;
  }
}
return closest;
```

ループ内のコードは無関係の下位問題を扱っている。それは「2つの地点（緯度経度）の球面距離を算出する」だ。コード量が多いので、新しい関数 spherical_distance() に抽出するといいだろう。

```
var spherical_distance = function (lat1, lng1, lat2, lng2) {
  var lat1_rad = radians(lat1);
  var lng1_rad = radians(lng1);
  var lat2_rad = radians(lat2);
  var lng2_rad = radians(lng2);

  // 「球面三角法の第二余弦定理」の公式を使う。
  return Math.acos(Math.sin(lat1_rad) * Math.sin(lat2_rad) +
    Math.cos(lat1_rad) * Math.cos(lat2_rad) *
    Math.cos(lng2_rad - lng1_rad));
};
```

残ったコードは以下のようになる。

```
var findClosestLocation = function (lat, lng, array) {
  var closest;
```

```

var closest_dist = Number.MAX_VALUE;
for (var i = 0; i < array.length; i += 1) {
    var dist = spherical_distance(lat, lng, array[i].latitude, array[i].longitude);
    if (dist < closest_dist) {
        closest = array[i];
        closest_dist = dist;
    }
}
return closest;
};

```

コードがずっと読みやすくなった。難しそうな幾何学の計算に心を奪われることなく、高レベルの目標に集中できるようになった。

さらに言うと、`spherical_distance()` は個別にテストができる関数だ。`spherical_distance()` は将来的に再利用可能な関数だ。これが「無関係の」下位問題と呼ばれる理由だ。完全に自己完結しているので、自分がアプリケーションにどのように使われるかを知らないのだ。

10.2 純粹なユーティリティコード

プログラムの核となる基本的なタスクというものがある。例えば、文字列の操作・ハッシュテーブルの使用・ファイルの読み書きなどがそうだ。

こうした「基本的なユーティリティ」は、プログラミング言語の組み込みライブラリとして実装されている。例えば、ファイルの中身をすべて読み込んだければ、PHP なら `file_get_contents("filename")` が、Python なら `open("filename").read()` が使える。

でも、たまに自分でこの溝を埋めなきゃいけないことがある。例えば C++ では、ファイルの中身をすべて読み込む方法が用意されていない。したがって、以下のようなコードを自分で書くことになる。

```

ifstream file(file_name);

// ファイルサイズを計算して、バッファにそのサイズを割り当てる。
file.seekg(0, ios::end);
const int file_size = file.tellg();
char* file_buf = new char [file_size];

```

```

// ファイルをバッファに読み込む。
file.seekg(0, ios::beg);
file.read(file_buf, file_size);
file.close();

...

```

これは「無関係の下位問題」の古典的な例だ。この部分は、`ReadFileToString()`などの新しい関数に置き換えるべきだろう。そうすれば、C++ が `ReadFileToString()` という関数をあらかじめ用意しているかのようなコードになる。

「このライブラリに XYZ() 関数があればなあ」と思ったら、その関数を自分で書けばいいのだ！（ただし、既存の関数がない場合に限る）そうしたコードは複数のプロジェクトで使えるユーティリティコードになっていくだろう。

10.3 その他の汎用コード

JavaScript をデバッグするときには、`alert()` でメッセージボックスをポップアップして、何らかの情報を表示させことが多い。これはウェブ版の「printf() デバッグ」だ。以下の関数呼び出しでは、Ajax でデータをサーバに送信して、返ってきたディクショナリを表示している。

```

ajax_post({
    url: 'http://example.com/submit',
    data: data,
    on_success: function (response_data) {
        var str = "{\n";
        for (var key in response_data) {
            str += " " + key + " = " + response_data[key] + "\n";
        }
        alert(str + "}");
    }
});

```

このコードの高レベルの目標は「サーバを Ajax で呼び出してレスポンスを処理する」である。でも、このコードの大部分は「ディクショナリをキレイに印字 (pretty print) する」という「無関係の下位問題」を解決しようとしている。このコードを

抽出して、`format_pretty(obj)` のような関数にするのは簡単だ。

```
var format_pretty = function (obj) {
  var str = "\n";
  for (var key in obj) {
    str += " " + key + " = " + obj[key] + "\n";
  }
  return str + "}";
};
```

思いも寄らない恩恵

`format_pretty()` を抽出する理由はたくさんある。呼び出し側のコードは簡潔になるし、`format_pretty()` ならあとから再利用できる。

でも、もう1つ大きな理由がある。気づきにくいかもしれないけど、「コードが独立していれば、`format_pretty()` の改善が楽になるから」だ。関数というのは、小さくして独立したものになっていれば、機能追加・読みやすさの向上・エッジケースの処理などが楽にできる。

以下は、`format_pretty(obj)` が処理できないケースだ。

- `obj` にはオブジェクトを期待している。普通の文字列（や `undefined`）だと例外が発生する。
- `obj` には単純な型を期待している。ネストしたオブジェクトだと `object Object` のように表示されるので、プリティとは言えない。

`format_pretty()` が独立した関数になっていなければ、これらを改善するのは大変だったと思う（特にネストしたオブジェクトを再帰的に印字するのは難しい）。

でも、関数になっていれば、こうした機能は簡単に追加できる。改善したコードは以下のようになる。

```
var format_pretty = function (obj, indent) {
  // null・undefined・文字列・非オブジェクトを処理する。
  if (obj === null) return "null";
  if (obj === undefined) return "undefined";
  if (typeof obj === "string") return '"' + obj + '"';
  if (typeof obj !== "object") return String(obj);
```

```
if (indent === undefined) indent = "";
// (非null) オブジェクトを処理する。
var str = "{\n";
for (var key in obj) {
  str += indent + " " + key + " = ";
  str += format_pretty(obj[key], indent + " ") + "\n";
}
return str + indent + "}";
};
```

これは先ほどの欠点を解決している。出力は以下のようになる。

```
{
  key1 = 1
  key2 = true
  key3 = undefined
  key4 = null
  key5 = {
    key5a = {
      key5a1 = "hello world"
    }
  }
}
```

10.4 汎用コードをたくさん作る

`ReadFileToString()` や `format_pretty()` は「無関係の下位問題」のいい例だ。いずれも基本的で広く適用可能なので、複数のプロジェクトで再利用できる。このようなコードには、簡単に共有できるように特別なディレクトリ（例：`util/`）を用意する。

汎用コードは素晴らしい。プロジェクトから完全に切り離されているからだ。このようなコードは開発もテストも理解も楽だ。すべてのコードがこうなればいいのに！

便利なライブラリやシステムのことを見てみよう。SQLデータベース・JavaScriptのライブラリ・HTMLのテンプレートシステム。いずれも内部のことを考えずに使っている。つまり、君のプロジェクトから完全に切り離されているわけだ。結果として、君のプロジェクトのコードは小さく保たれている。

君のプロジェクトもできるだけ独立したライブラリに分離したほうがいい。そうす

れば、残りのコードは小さくて考えやすいものになる。

このプログラミングはトップダウン？ ボトムアップ？

トップダウンプログラミングとは、先に高レベルのモジュールや関数を設計してから、それらをサポートする低レベルの関数を実装していく方式だ。

ボトムアッププログラミングとは、先にすべての下位問題を解決してから、それらを利用する高レベルのコンポーネントを実装していく方式だ。

本章はいずれかの手法を推奨するものではない。プログラミングとは両方の側面を持つものだと思う。大切なのは、下位問題を分離して処理できるかどうかだ。

10.5 プロジェクトに特化した機能

抽出する下位問題というのは、プロジェクトから完全に独立したものであるほうがいい。ただし、完全に独立していなくても、それはそれで問題ない。下位問題を取り除くだけでも効果がある。

ビジネスレビューサイトから例を持ってきた。このPythonコードでは、新しいBusinessオブジェクトを作り、name・url・date_createdを設定している。

```
business = Business()
business.name = request.POST["name"]

url_path_name = business.name.lower()
url_path_name = re.sub(r"\.", "", url_path_name)
url_path_name = re.sub(r"[^a-z0-9]+", "-", url_path_name)
url_path_name = url_path_name.strip("-")
business.url = "/biz/" + url_path_name

business.date_created = datetime.datetime.utcnow()
business.save_to_database()

business.date_created = datetime.datetime.utcnow()
```

urlはnameの「クリーン」バージョンだ。例えば、nameが「A.C. Joe's Tire & Smog, Inc.」であれば、urlは「/biz/ac-joes-tire-smog-inc」になる。

このコードの「無関係の下位問題」は、「名前を有効なURLに変換する」だ。これは楽に抽出できる。ついでに、正規表現もプリコンパイルしておこう（読みやすい

名前もつけておこう）。

```
CHARS_TO_REMOVE = re.compile(r"\.]+")
CHARS_TO_DASH = re.compile(r"[^a-z0-9]+")

def make_url_friendly(text):
    text = text.lower()
    text = CHARS_TO_REMOVE.sub('', text)
    text = CHARS_TO_DASH.sub('-', text)
    return text.strip("-")
```

元のコードに「規則性」のあるパターンができた。

```
business = Business()
business.name = request.POST["name"]
business.url = "/biz/" + make_url_friendly(business.name)
business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

コードが読みやすくなった。これで正規表現や文字列処理に心を奪われずに済む。make_url_friendly()はどこに置けばいいのだろう？汎用的な関数なので、util/ディレクトリに入れてもいいと思う。でも、この正規表現はアメリカのビジネス名だけを対象にしているので、元のファイルと同じ場所に置いたほうがいいかもしれない。大切なことじゃないので、あとで決めてもいいだろう。大切なのは、make_url_friendly()を抽出するということだ。

10.6 既存のインターフェースを簡潔にする

誰もがキレイなインターフェースを提供するライブラリが好きだ。引数は少なくて、事前設定も必要なくて、面倒なことをしなくても使えるライブラリ。インターフェースがキレイだとコードが優雅に見える。簡潔で、しかも強力だ。

インターフェースがキレイじゃなくても、自分で「ラップ」関数を作ることができる。

例えば、JavaScriptでブラウザのクッキーを扱うのはすごく残念な感じだ。クッキーは名前と値のペアになっているはずなのに、ブラウザが提供するインターフェースには、以下のような構文のdocument.cookieという文字列しかない。

```
name1=value1; name2=value2; ...
```

必要なクッキーを探すには、この巨大な文字列を自分でパースしなければいけない。以下は、`max_results` という名前のクッキーを読み込むコードだ。

```
var max_results;
var cookies = document.cookie.split(';");
for (var i = 0; i < cookies.length; i++) {
  var c = cookies[i];
  c = c.replace(/\s+/g, ''); // 先頭の空白を削除
  if (c.indexOf("max_results=") === 0)
    max_results = Number(c.substring(12, c.length));
}
```

うへえ。汚いコードだ。これは関数 `get_cookie()` を作ってから、以下のように書きたい。

```
var max_results = Number(get_cookie("max_results"));
```

クッキーの値の作成や変更もおかしなことになっている。`document.cookie` に正しい構文で値を設定する必要があるのだ。

```
document.cookie = "max_results=50; expires=Wed, 1 Jan 2020 20:53:47 UTC; path=/";
```

この文はすべてのクッキーを書き換えるように思えるけど、（不思議なことに）書き換えないものである！[†]

クッキーを設定する理想的なインターフェースはこうだ。

```
set_cookie(name, value, days_to_expire);
```

クッキーの削除も直感的ではない。有効期限を過去にして設定しなければいけないのだ。理想的なインターフェースはもっと簡潔だ。

```
delete_cookie(name);
```

ここでの教訓は「理想とは程遠いインターフェースに妥協することはない」というこ

[†] 訳注：書き換えずに追記する。

とだ。自分でラッパー関数を用意して、手も足も出ない汚いインターフェースを覆い隠すのだ。

10.7 必要に応じてインターフェースを整える

プログラムの多くのコードは、その他のコードを支援するためだけに存在する。例えば、関数の事前処理や事後処理などがそうだ。こうした「グルー」コードは、プログラムの本質的なロジックとは関係ないことが多い。別の関数に分離するのはこういうコードになる。

例えば、`{ "username": "...", "password": "..." }` のようにユーザの機密情報を含んだ Python のディクショナリがあるとする。これらの情報を URL に使いたい。でも、機密情報なので、Cipher クラスで暗号化したい。

ただし、Cipher クラスはディクショナリではなく文字列を受け取る。こちらが欲しいのは URL セーフな文字列だけど、Cipher クラスは単なる文字列を返すようになっている。また、Cipher クラスは、他にも引数が必要なので使いにくい。

最初は単純なタスクだったのに、多くのグルーコードが必要になった。

```
user_info = { "username": "...", "password": "..." }
user_str = json.dumps(user_info)
cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
encrypted_bytes = cipher.update(user_str)
encrypted_bytes += cipher.final() # 現在の 128 ビットブロックをフラッシュする
url = "http://example.com/?user_info=" + base64.urlsafe_b64encode(encrypted_bytes)
...
```

ここで取り組んでいるのは「ユーザの情報を暗号化して URL に含める」だけど、コードの大部分が「Python のオブジェクトを URL セーフな文字列にする」ものになっている。こうした下位問題は抽出しておこう。

```
def url_safe_encrypt(obj):
  obj_str = json.dumps(obj)
  cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
  encrypted_bytes = cipher.update(obj_str)
  encrypted_bytes += cipher.final() # 現在の 128 ビットブロックをフラッシュする
  return base64.urlsafe_b64encode(encrypted_bytes)
```

その結果、プログラムの本質的なロジックを扱うコードは簡潔になる。

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + urlsafe_encrypt(user_info)
```

10.8 やりすぎ

本章の最初に言ったけど、ぼくたちの目標は「無関係の下位問題を積極的に見つけて抽出する」ことだ。「積極的に」と言ったのは、みんなの積極性が足りないからだ。でも、やりすぎたり、度を越したりする可能性もある。

例えば、前節のコードをさらに分割すると、以下のようなになる。

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + urlsafe_encrypt_obj(user_info)

def urlsafe_encrypt_obj(obj):
    obj_str = json.dumps(obj)
    return urlsafe_encrypt_str(obj_str)

def urlsafe_encrypt_str(data):
    encrypted_bytes = encrypt(data)
    return base64.urlsafe_b64encode(encrypted_bytes)

def encrypt(data):
    cipher = make_cipher()
    encrypted_bytes = cipher.update(data)
    encrypted_bytes += cipher.final() # 残りをフラッシュする
    return encrypted_bytes

def make_cipher():
    return Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
```

小さな関数を作りすぎると、逆に読みにくくなってしまう。あちこちに飛び回る実行パスを追いかけることになるからだ。

新しい関数をコードに追加すると、ごくわずかに（でも確実に）読みにくさのコストが発生する。上記の例は、このコストを相殺できるようなものではない。プロジェクトの他の部分から再利用できるのであれば、小さな関数を追加するのも意味のあることかもしれない。でも、それまでは必要ない。

10.9 まとめ

本章を簡単にまとめると、プロジェクト固有のコードから汎用コードを分離することだ。ほとんどのコードは汎用化できる。一般的な問題を解決するライブラリやヘルパー関数を作っていてけば、プログラムに固有の小さな核だけが残る。

この技法が役に立つの、プロジェクトの他の部分から分離された、境界線の明確な小さな問題に集中できるからだ。こうした下位問題に対する解決策は、より緻密で正確なものになる。それに、あとでコードを再利用できるかもしれない。

あわせて読みたい

マーチン・ファウラーの『Refactoring: Improving the Design of Existing Code』(Fowler et al., Addison-Wesley Professional, 1999)[†]では、「メソッドの抽出」というリファクタリング手法が紹介されている。また、さまざまなリファクタリングカタログも掲載されている。

ケント・ベックの『Smalltalk Best Practice Patterns』(Prentice Hall, 1996)[‡]では、「Composed Method」パターンが紹介されている。これは、小さな関数に分割する原則をまとめたものだ。特に「メソッド内部の処理は同じ抽象度のレベルになるようにします」の原則が大切だ。

これらの考えは、ぼくたちの「無関係の下位問題を抽出する」というアドバイスとよく似ている。本章で説明したのは、メソッドを抽出する手法の一例だ。

[†] 訳注 「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・ファウラー著、児玉公信、平澤章、友野晶夫、梅沢真史 訳、ピアソンエデュケーション)

[‡] 訳注 『ケント・ベックの Smalltalk ベストプラクティス・パターン——シンプル・デザインへの宝石集』(ケント・ベック著、梅沢真史、皆川誠、小黒直樹、森島みどり 訳、ピアソンエデュケーション)

[§] 訳注 訳書では「メソッド内部のメッセージは同じ抽象度のレベルになるようにします」になっているけど、「メッセージ」ではなく「処理」または「操作」が妥当と思われる。原文は「Keep all of the operations in a single method at the same level of abstraction.」。

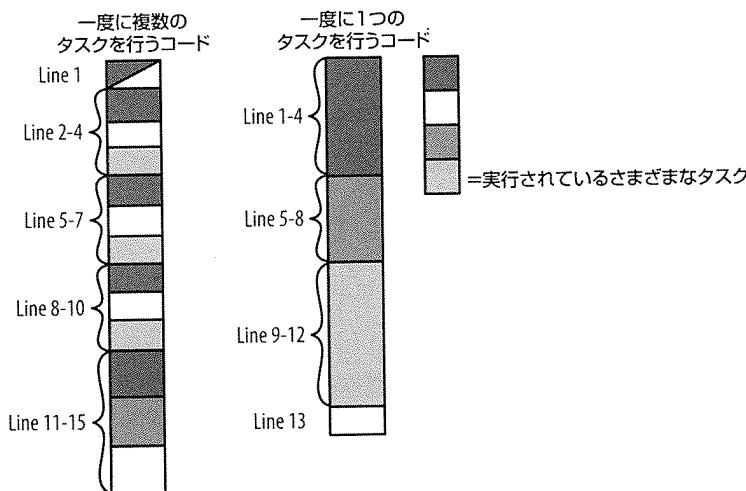
一度に複数のことをするコードは理解しにくい。例えば、オブジェクトを生成して、データをキレイにして、入力をパースして、ビジネスロジックを適用しているようなコードだ。これらのコードがすべて絡み合っていると、「タスク」が個別に完結しているコードよりも理解するのが難しい。

鍵となる考え方



コードは1つずつタスクを行うようにしなければいけない。

別の言い方をすれば、本章はコードの「デフラグ」について説明している。以下の図は、デフラグの手順を示したものだ。左側には、さまざまなタスクを行なうコードを描いている。右側には、一度に1つのタスクを行うように再構成したコードを描いている。



「関数は一度に1つのことを行うべきだ」というアドバイスを聞いたことがあるかもしれない。ぼくたちのアドバイスもこれと似ている。でも、関数に限った話じゃない。もちろん、大きな関数は小さな複数の関数に分割したほうがいい。でも、関数のなかでコードを小さく構成することもできる。例えば、論理的な区分に分けてあげるのだ。

「一度に1つのタスクをする」ためにはぼくたちが使っている手順を紹介しよう。

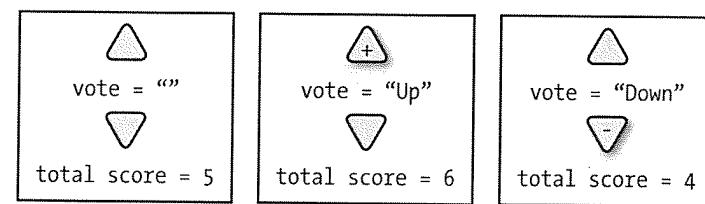
1. コードが行っている「タスク」をすべて列挙する。この「タスク」という言葉はユルく使っている。「オブジェクトが妥当かどうかを確認する」のように小さなこともあれば、「ツリーのすべてのノードをイテレートする」のようにあいまいなこともある。
2. タスクをできるだけ異なる関数に分割する。少なくとも異なる領域に分割する。

本章では、このための方法をいくつか紹介する。

11.1 タスクは小さくできる

ブログに設置する投票用のウィジットがあるとする。ユーザは「アップ（賛成）」と「ダウン（反対）」を投票できる。scoreは、すべての投票を合計したものである。「アップ」は1点で、「ダウン」は-1点だ。

ユーザの投票には3つの状態がある。これらがscoreに与える影響を見てみよう。



ユーザが（投票や更新のために）ボタンをクリックしたら、以下のJavaScriptが呼び出される。

```
vote_changed(old_vote, new_vote); // 投票は "Up"・"Down"・"" のいずれか
```

以下の関数は、old_voteとnew_voteのすべての組み合わせを考慮して、scoreを更新している。

```
var vote_changed = function (old_vote, new_vote) {
  var score = get_score();

  if (new_vote !== old_vote) {
```

```

if (new_vote === 'Up') {
  score += (old_vote === 'Down' ? 2 : 1);
} else if (new_vote === 'Down') {
  score -= (old_vote === 'Up' ? 2 : 1);
} else if (new_vote === '') {
  score += (old_vote === 'Up' ? -1 : 1);
}
}

set_score(score);
};

```

短いコードだけど、いろんなことをやっている。難しいところがたくさんあるので、エラーやタイプオшибкаがあるっても一目見ただけではわからない。

このコードはたった1つのこと（スコアを更新すること）をしているように見え、実際に一度に2つのタスクを行っている。

1. `old_vote` と `new_vote` を数値に「パース」する。
2. `score` を更新する。

2つのタスクを別々に解決すれば、読みやすいコードになる。以下のコードは、最初のタスク（投票を数値にパースする）を解決したものである。

```

var vote_value = function (vote) {
  if (vote === 'Up') {
    return +1;
  }
  if (vote === 'Down') {
    return -1;
  }
  return 0;
};

```

残りのコードは、2つめのタスク（スコアの更新）を解決している。

```

var vote_changed = function (old_vote, new_vote) {
  var score = get_score();

```

```

  score -= vote_value(old_vote); // 古い値を削除する
  score += vote_value(new_vote); // 新しい値を追加する

  set_score(score);
};

```

新しいコードは、一目見ただけで正しく動きそうだ。これがコードを「楽に理解できる」ようにすることである。

11.2 オブジェクトから値を抽出する

以前、ユーザの所在地を読みやすい文字列（「都市、国」）に整形する JavaScript のコードを書いたことがある。例えば、「Santa Monica, USA」や「Paris, France」のような感じだ。`location_info` ディクショナリに構造化された情報があって、そこからすべてのフィールドの「都市」と「国」を選んで連結するのである。

以下は、入出力の例を示したものだ。

location_info	
LocalityName	"Santa Monica"
SubAdministrativeAreaName	"Los Angeles"
AdministrativeAreaName	"California"
CountryName	"USA"



"Santa Monica, USA"

ここまで簡単だ。ややこしいのは、この4つの値のいずれかが（あるいはすべてが）存在しない可能性があることだ。以下に対応策を示そう。

- 「都市」を選ぶときには、「LocalityName」（市や町）→「SubAdministrativeAreaName」（都市や郡）→「AdministrativeAreaName」（州や地域）の順番で使用可能なものを使う。
- 以上の3つすべてが使えない場合は、「都市」に「Middle-of-Nowhere」（何でもない場所）というデフォルト値を設定する。

- 「国」に「CountryName」(国名)が使えない場合は、「Planet Earth」(地球)というデフォルト値を設定する。

以下の図は、値がないときの対処法の例だ。

location_info	
LocalityName	(undefined)
SubAdministrativeAreaName	(undefined)
AdministrativeAreaName	(undefined)
CountryName	"Canada"

location_info	
LocalityName	(undefined)
SubAdministrativeAreaName	"Washington, DC"
AdministrativeAreaName	(undefined)
CountryName	"USA"

↓ ↓

"Middle-of-Nowhere, Canada"

"Washington, DC, USA"

このタスクを実装したコードが以下になる。

```
var place = location_info["LocalityName"]; // 例: "Santa Monica"
if (!place) {
    place = location_info["SubAdministrativeAreaName"]; // 例: "Los Angeles"
}
if (!place) {
    place = location_info["AdministrativeAreaName"]; // 例: "California"
}
if (!place) {
    place = "Middle-of-Nowhere";
}
if (location_info["CountryName"]) {
    place += ", " + location_info["CountryName"]; // 例: "USA"
} else {
    place += ", Planet Earth";
}

return place;
```

ひどいコードなのはわかってるけど、これでもちゃんと動く。数日後、機能を追加することになった。アメリカの場合は、国名ではなく（可能であれば）州名を表示するそうだ。例えば、「Santa Monica, USA」は「Santa Monica,

California」になるわけだ。

先ほどのコードにこの機能を追加したら、もっとひどいコードになるだろう。

「一度に1つのタスク」を適用する

コードを変更する前に、このコードが一度に複数のタスクを行っていることに気づいただろう。

1. location_info ディクショナリから値を抽出する。
2. 「都市」の優先順位を調べる。何も見つからなかったら、デフォルトで「Middle-of-Nowhere」にする。
3. 「国」を取得する。なければ「Planet Earth」にする。
4. place を更新する。

これらのタスクを個別に解決するようなコードに書き換えよう。

最初のタスク (location_info から値を抽出) は、そのままでいいだろう。

```
var town  = location_info["LocalityName"]; // 例: "Santa Monica"
var city  = location_info["SubAdministrativeAreaName"]; // 例: "Los Angeles"
var state = location_info["AdministrativeAreaName"]; // 例: "CA"
var country = location_info["CountryName"]; // 例: "USA"
```

これで location_info がなくなって、長くて直感的ではないキーを覚える必要がなくなった。これからは4つの簡潔な変数を扱えばいい。

次に、戻り値の「後半」を見つけなければいけない。

```
// 先にデフォルト値を設定して、値が見つかったら書き換える。
var second_half = "Planet Earth";
if (country) {
    second_half = country;
}
if (state && country === "USA") {
    second_half = state;
}
```

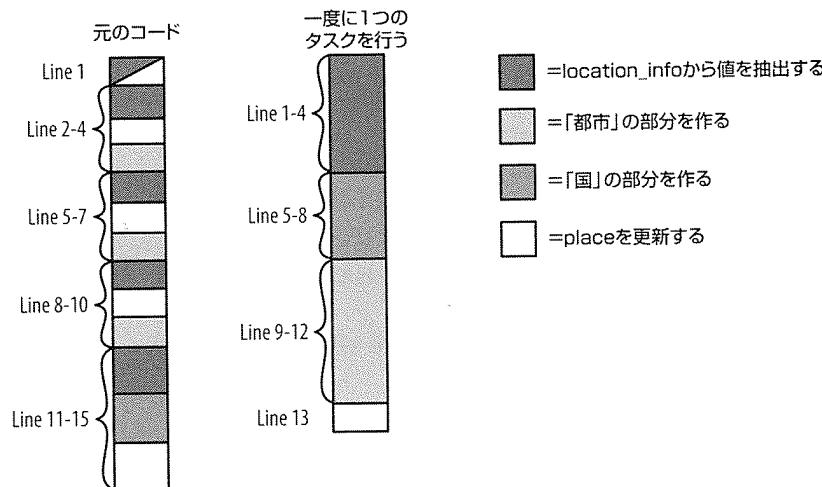
同じようにして「前半」を見つけることができる。

```
var first_half = "Middle-of-Nowhere";
if (state && country !== "USA") {
    first_half = state;
}
if (city) {
    first_half = city;
}
if (town) {
    first_half = town;
}
```

最後に情報をつなぎ合わせる。

```
return first_half + ", " + second_half;
```

最初に示した「デフラグ」の図は、実はこのコードを表したものだった。以下に、もっと詳細なものを示そう。



ご覧のように、解決策のコードでは、4つのタスクが別々の領域にデフラグされている。

その他の手法

コードをリファクタリングするときには、複数の手法が使えることが多い。今回も例外ではない。タスクを分割すると、コードのことを考えやすくなる。その結果、もっとうまくリファクタリングできる方法を思いつく可能性がある。

例えば、if文が連続していると、すべての場合分けを注意深く読まなければいけない。このコードには2つのサブタスクがある。

複数の変数を調べて、最も適したものを選択する。

1. 国が「USA」ならば、別の変数を使う。
2. 先ほどのコードには「if USA」のロジックが他のロジックと混ざっていた。「USA」と「非 USA」は別々に処理できる。

```
var first_half, second_half;
```

```
if (country === "USA") {
    first_half = town || city || "Middle-of-Nowhere";
    second_half = state || "USA";
} else {
    first_half = town || city || state || "Middle-of-Nowhere";
    second_half = country || "Planet Earth";
}

return first_half + ", " + second_half;
```

JavaScriptに詳しくない人のために説明すると、`a || b || c`は、最初に「真」と評価できる値（ここでは定義された空ではない文字列）を取り出すイディオムだ。これで検査や更新の処理が簡潔になった。if文が少なくて、ビジネスロジックを扱うコードも短くなった。

11.3 もっと大きな例

ぼくたちが作ったウェブクローリングシステムがある。`UpdateCounts()` はさまざまな統計値を更新する関数だ。ウェブページをダウンロードしたあとに毎回呼ばれていることになっている。

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][[hd.exit_state()]]++; // 例: "SUCCESS" or "FAILURE"
    counts["Http Response"][[hd.http_response()]]++; // 例: "404 NOT FOUND"
    counts["Content-Type"][[hd.content_type()]]++; // 例: "text/html"
}
```

まだ、こんなふうにならいいなと思っている段階だ！

実際の HttpDownload オブジェクトには、上記のようなメソッドはない。HttpDownload は巨大で複雑なクラスになっていて、ネストしたクラスもたくさん入っている。必要な値は自分で探索なければいけない。しかも、値がどこかへ消えてしまっていることもある。そういう場合は、デフォルト値に「unknown」を使うことにしよう。

というわけで、すごく汚いコードになった。

// 警告：このコードを長時間直視しないでください。

```
void UpdateCounts(HttpDownload hd) {
    // 可能であれば Exit State を見つける。
    if (!hd.has_event_log() || !hd.event_log().has_exit_state()) {
        counts["Exit State"][[ "unknown" ]]++;
    } else {
        string state_str = ExitStateTypeName(hd.event_log().exit_state());
        counts["Exit State"][[ state_str ]]++;
    }

    // HTTP ヘッダがなければ、残りの要素に "unknown" を設定する。
    if (!hd.has_http_headers()) {
        counts["Http Response"][[ "unknown" ]]++;
        counts["Content-Type"][[ "unknown" ]]++;
        return;
    }

    HttpHeaders headers = hd.http_headers();

    // HTTP レスポンスをログに記録する。なければ "unknown" と記録する。
    if (!headers.has_response_code()) {
        counts["Http Response"][[ "unknown" ]]++;
    } else {
        string code = StringPrintf("%d", headers.response_code());
        counts["Http Response"][[ code ]]++;
    }
}
```

// Content-Type をログに記録する。なければ "unknown" と記録する。

```
if (!headers.has_content_type()) {
    counts["Content-Type"][[ "unknown" ]]++;
} else {
    string content_type = ContentTypeMime(headers.content_type());
    counts["Content-Type"][[ content_type ]]++;
}
```

大量のコードだ。ロジックもいっぱい。重複コードだってある。読んでて楽しくない。

このコードは、複数のタスクを交互に切り替えている。タスクには以下のようなものがある。

1. キーのデフォルト値に「unknown」を使う。
2. HttpDownload のメンバがあるかどうかを確認する。
3. 値を抽出して文字列に変換する。
4. counts[] を更新する。

このコードを改善するには、タスクを別々の領域に分割すればいい。

```
void UpdateCounts(HttpDownload hd) {
    // タスク：抽出したい値にデフォルト値を設定する。
    string exit_state = "unknown";
    string http_response = "unknown";
    string content_type = "unknown";

    // タスク：HttpDownload から値を1つずつ抽出する。
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        exit_state = ExitStateTypeName(hd.event_log().exit_state());
    }
    if (hd.has_http_headers() && hd.http_headers().has_response_code()) {
        http_response = StringPrintf("%d", hd.http_headers().response_code());
    }
    if (hd.has_http_headers() && hd.http_headers().has_content_type()) {
        content_type = ContentTypeMime(hd.http_headers().content_type());
    }
}
```

```
// タスク : counts[] を更新する。
counts["Exit State"][exit_state]++;
counts["Http Response"][http_response]++;
counts["Content-Type"][content_type]++;
}
```

このコードには目的を持った3つの領域がある。

1. 3つのキーのデフォルト値を定義する。
2. 可能であれば、キーの値を抽出して文字列に変換する。
3. キーの counts[] を更新する。

領域に分けておくといいのは、それぞれが分離されているからだ。ある領域にいるときは、他の領域のことは考えなくて済む。

タスクは4つあったのに、領域は3つに分かれている。でも、これでいい。最初に挙げたタスクは出発点すぎない。ここでやったように、そこからいくつかに分割できれば、それでいいのだ。

さらなる改善

新しいバージョンのコードは、元の巨大怪物から大きく改善されたものになった。でも、新しい関数は作らなかった。前にも言ったように、「一度に1つのこと」は関数に限った話ではない。

他にもこのコードを改善する方法はある。3つのヘルパー関数を導入するのだ。

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][ExitState(hd)]++;
    counts["Http Response"][HttpResponse(hd)]++;
    counts["Content-Type"][ContentType(hd)]++;
}
```

これらの関数は、対応する値を抽出して、もしなければ「unknown」を返すものだ。

```
string ExitState(HttpDownload hd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state());
    } else {
        return "unknown";
    }
}
```

この解決策では、変数の定義すらしていない！「9章 変数と読みやすさ」で書いたように、中間結果を保持する変数は削除できることが多い。

2つの解決策は、問題を異なる方向に「スライス」している。どちらの解決策も読みやすく、一度に1つのタスクのことだけを考えられるようになっている。

11.4 まとめ

本章では、コードを構成する簡単な技法「一度に1つのタスクを行う」を紹介した。

読みにくいコードがあれば、そこで行われているタスクをすべて列挙する。そこには別の関数（やクラス）に分割できるタスクがあるだろう。それ以外は、関数の論理的な「段落」になる。タスクをどのように分割するよりも、分割するということが大切なのだ。いちばん難しいのは、プログラムが行っていることを正確に説明することである。

