

/THEORY/IN/PRACTICE

The Art of Readable Code

Simple and Practical Techniques for Writing Better Code

O'REILLY®

Dustin Boswell
Trevor Foucher

The Art of Readable Code

“Being aware of how the code you create affects those who look at it later is an important part of developing software. The authors did a great job in taking you through the different aspects of this challenge, explaining the details with instructive examples.”

—Michael Hunger, Software Developer, Neo Technology

As programmers, we've all seen source code that's so ugly and buggy it makes our brain ache. Over the last five years, authors Dustin Boswell and Trevor Foucher have analyzed hundreds of examples of “bad code” (much of it their own) to determine why they're bad and how they could be improved. Their conclusion? You need to write code that minimizes the time it would take someone else to understand it—even if that someone else is you.

This book focuses on basic principles and practical techniques you can apply every time you write code. Using easy-to-digest code examples from different languages, each chapter dives into a different aspect of coding, and demonstrates how you can make your code easy to understand.

- Simplify naming, commenting, and formatting with tips that apply to every line of code
- Refine your program's loops, logic, and variables to reduce complexity and confusion
- Attack problems at the function level, such as reorganizing blocks of code to do one task at a time
- Write effective test code that is thorough and concise—as well as readable

Dustin Boswell graduated from CalTech, and worked at Google for five years on web crawling infrastructure and ad programs. He's built several websites and enjoys working on big data and machine learning.

Trevor Foucher has spent the past decade at Microsoft as an engineer, manager, and tech lead on Windows and security products, and at Google working on ad programs and search infrastructure.

US \$34.99

CAN \$36.99

ISBN: 978-0-596-80229-5



5 3 4 9 9



Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY
oreilly.com

読みやすいコードの技術

ダスティン・ボズウェルとトレヴァー・フーシエ

O'REILLY®

北京・ケンブリッジ・ファーナム・ケルン・セバストポル・東京

読みやすいコードの技術

ダステイン・ボズウェルとトレヴァー・フーシエ著

著作権 © 2012 ダステイン・ボズウェルおよびトレバー・フーチャー。無断転載を禁じます。アメリカ合衆国で印刷されています。

O'Reilly Media, Inc. (1005 Gravenstein Highway North, Sebastopol, CA 95472) によって発行されています。

オライリーの書籍は、教育、ビジネス、または販売促進目的で購入できます。ほとんどのタイトルではオンライン版も利用できます (<http://my.safaribooksonline.com>)。詳細については、当社の法人/機関営業部門までお問い合わせください: (800) 998-9938 または company@oreilly.com。

編集者: メアリー・トレセラー

インデクサ: ボトマック インデックス LLC

制作編集者: テレサ・エルシー

表紙デザイナー: スーザン・トンプソン

コピーエディター: ナンシー・ウルフ・コタリー

インテリアデザイナー: デビッド・フタト

校正者: テレサ・エルシー

イラストレーター: デイブ・アルレッドとロバート・ロマーノ

2011年11月: 初版。

初版の改訂履歴:

2011-11-01 最初のリリース

見る <http://oreilly.com/catalog/errata.csp?isbn=9780596802295> リリースの詳細については、

Nutshell Handbook、Nutshell Handbook のロゴ、および O'Reilly のロゴは、O'Reilly Media, Inc. の登録商標です。読みやすいコードの技術、楽譜の画像、および関連するトレードドレスは、O'Reilly Media, Inc. の商標です。

メーカーと販売者が自社製品を区別するために使用する名称の多くは、商標として主張されています。これらの名称が本書に登場し、O'Reilly Media, Inc. が商標権の主張を認識していた場合、その名称は大文字または頭文字で印刷されています。

本書の作成にはあらゆる予防措置が講じられていますが、出版社および著者は、本書に含まれる情報の使用によって生じた誤りや脱落、または損害に対して一切の責任を負いません。

ISBN: 978-0-596-80229-5

【LSI】

1320175254

コンテンツ

序文	vii
1	1
コードは理解しやすいものでなければなりません コードを「よ り良く」するには何か必要ですか?	2
可読性の基本定理は小さいほど常に優れ ているのでしょうか?	3
理解までの時間は他の目標と矛盾しますか? 難し い部分	3
	4
	4
<hr/>	
パート 1 表面レベルの改善	
2	7
名前に情報を詰め込む 特定の単語 を選択する	8
tmp や retval などの一般的な名前を避ける 抽象的 な名前よりも具体的な名前を好む 名前に追加情報 を付加する 名前の長さはどのくらいにする必要が ありますか?	10
名前の書式設定を使用して意味の概要を 伝える	13
	15
	18
名前の書式設定を使用して意味の概要を 伝える	20
	21
3	23
誤解されない名前 例: <i>Filter()</i>	24
例: クリップ(テキスト、長さ)	24
(包括的な) 制限の最小値と最大値を優先します。	25
包括的な範囲の最初と最後を優先します。	26
包含/排他範囲の名前付けブール値の開始と終 了を優先する	26
ユーザーの期待に応える	27
例: 複数の名前候補の評価の概要	27
	29
	31
4	33
美学	34
なぜ美学が重要なのでしょうか?	34
改行を再配置して一貫性を保ち、不規則性を解消するた めのメソッドをコンパクトに使用する	35
役立つ場合は列の配置を使用する	37
意味のある順序を選択し、一貫して使用する	38
宣言をブロックに整理する	39
コードを「段落」に分割する	40
個人的なスタイルと一貫性のま とめ	41
	42
	43

5	何をコメントすべきかを知る コメントしてはいけないこと あなたの考えを記録する 読者の立場になって考えてみよう 最終的な考え方 – ライターの行き詰まりを乗り越えるまとめ	45 47 49 51 56 57
6	正確かつコンパクトなコメントを作成する コメントをコンパクトに保つ あいまいな代名詞を避ける すさんな文章を洗練する 関数の動作を正確に説明する 特殊なケースを示す入出力例を使用する コードの意図を説明する 「名前付き関数パラメータ」のコメントに情報量の多い単語が使用されている まとめ	59 60 60 61 61 61 61 62 63 64 65
<hr/>		
パート2 ループとロジックの簡素化		
7	制御フローを読みやすくする 条件文の引数の順序 if/else ブロックの順序 ?: 条件式 (別名「三項演算子」) do/while ループを避ける 関数から早く戻る悪名高い goto ネストを最小限に抑える 実行の流れを理解できますか? まとめ	69 70 71 73 74 75 76 77 79 80
8	巨大な表現を分解する 変数の説明 要約変数 短絡論理を悪用したド・モルガンの法則の使用 例: 巨大なステートメントを分解する 複雑なロジックとの格闘 式を簡素化する別の創造的な方法の概要	83 84 84 85 86 86 89 90 90
9	変数と可読性 変数の削除 変数のスコープを縮小する ライブ・ツォンス変数を優先する 最後の例 まとめ	93 94 97 103 104 106

パート3 コードの再構成

10	無関係な部分問題の抽出 <i>導入例: findClosestLocation()</i> 純粋なユーティリティコード	109
	その他の汎用コード 多数の汎用コードを作成するプロジェクト	110
	固有の機能	111
	既存のインターフェースの簡素化	112
	ニーズに合わせてインターフェースを再構築する	114
	まとめ	115
11	一度に1つのタスク タスクは小規模でも構いません オブジェクトからの値の抽出より大きな例	116
	まとめ	117
12	思考をコードに変える ロジックを明確に説明する ライブラリを理解することは、この方法をより大きな問題に適用するのに役立ちます	118
13	書くコードの削減 その機能をわざわざ実装する必要はありません - 必要ありません 質問して要件を細分化します コードベースを小さく保つ 身の回りのライブラリについてよく知る例: コーディングの代わりに Unix ツールを使用する	121
	まとめ	123
		124
		128
		130
		131
		132
		133
		134
		137
		139
		140
		140
		142
		143
		144
		145

第4部 嶄選されたトピック

14	テストと可読性 テストを読みやすく維持しやすくする このテストの何が問題になっているのでしょうか? このテストを読みやすくするエラー メッセージを読みやすくする適切な テスト入力を選択する テスト関数に名前を付ける そのテストの何が間違っていたのでしょうか? 行き過ぎたテストフレンドリーな 開発 まとめ	149
15	「分/時間カウンター」の設計と実装 問題 クラスインターフェイスの定義	150
		150
		151
		154
		156
		158
		159
		160
		162
		162
15	「分/時間カウンター」の設計と実装 問題 クラスインターフェイスの定義	165
		166
		166

試み 1: 単純なソリューション 試み	169
2: コンベヤベルトの設計 試み 3:	171
時間を使やした設計 3 つのソ	174
リューションの比較 まとめ	179
	179
あ 参考文献	181
索引	185

序文



私たちは優秀なエンジニアを擁し、大きな成功を収めているソフトウェア会社で働いてきましたが、遭遇したコードにはまだ改善の余地がたくさんあります。実際、私たちは非常に醜いコードをいくつか見てきましたし、おそらくあなたもそうしているでしょう。

しかし、美しく書かれたコードを見ると感動します。優れたコードは、何が起こっているかをすぐに知ることができます。使うのが楽しくて、自分のコードをより良くする動機になります。

この本の目的は、コードを改善するのに役立つことです。「コード」というときは、文字通り、エディターで見つめているコード行を意味します。私たちはプロジェクトの全体的なアーキテクチャや、デザインパターンの選択について話しているのではありません。これらは確かに重要ですが、私たちの経験では、プログラマーとしての日常生活のほとんどは、変数の名前付け、ループの作成、関数レベルでの問題の解決などの「基本的な」ことに費やされています。そして、この作業の大部分は、すでに存在するコードを読んで編集することです。この本が日常のプログラミングに非常に役立ち、チームの全員に勧めていただけることを願っています。

この本の内容

この本は可読性の高いコードの書き方について書かれています。この本の重要な考え方は、**コードは理解しやすいものでなければなりません**。具体的には、他の人がコードを理解するのにかかる時間を最小限に抑えることを目標にする必要があります。

この本では、この考え方を説明し、C++、Python、JavaScript、Javaなどのさまざまな言語の例を多数示します。高度な言語機能は避けているので、これらの言語をすべて知らなくても、簡単に理解できるはずです。(私たちの経験では、いずれにしても、可読性の概念はほとんど言語に依存しません。)

各章では、コーディングのさまざまな側面と、コーディングを「理解しやすく」する方法について詳しく説明します。この本は4つの部分に分かれています。

表面レベルの改善

命名、コメント、美学 - コードベースのすべての行に適用されるシンプルなヒント

ループとロジックの簡素化

プログラム内のループ、ロジック、変数を改良して理解しやすくする方法

コードを再編成する

コードの大きなブロックを整理し、関数レベルで問題に対処するためのより高度な方法

選択されたトピック

「理解しやすさ」をテストとより大規模なデータ構造のコーディング例に適用する

この本の読み方

私たちの本は、楽しくカジュアルに読んでいただくことを目的としています。ほとんどの読者が1～2週間でこの本をすべて読み終えることを願っています。

章は「難易度」順に並べられており、基本的なトピックが最初にあり、より高度なトピックが最後にあります。ただし、各章は独立しているため、単独で読むこともできます。必要に応じてスキップしてください。

コード例の使用

この本はあなたの仕事を成し遂げるのに役立ちます。一般に、本書のコードをプログラムやドキュメントで使用できます。コードの重要な部分を複製しない限り、許可を得るために当社に連絡する必要はありません。たとえば、本書のコードのいくつかのチャンクを使用するプログラムを作成する場合、許可は必要ありません。オンライン書籍の例を収録した CD-ROM を販売または配布するには許可が必要です。本書を引用したりサンプルコードを引用して質問に回答する場合、許可は必要ありません。本書の大量のサンプルコードを製品のドキュメントに組み込むには許可が必要です。

帰属を明示していただきますが、必須ではありません。属性には通常、タイトル、著者、出版社、ISBN が含まれます。例えば："読みやすいコードの技術ダスティン・ボズウェルとトレヴァー・フーシエ著。著作権 2012 Dustin Boswell および Trevor Foucher、978-0-596-80229-5。

コード例の使用がフェアユースまたは上記で与えられた許可の範囲外であると思われる場合は、お気軽に次のアドレスまでご連絡ください。[許可@oreilly.com](mailto:许可@oreilly.com)。

Safari® ブック オンライン

 Safari Books Online は、7,500 冊以上のテクノロジーおよびクリエイティブな参考書やビデオを簡単に検索して、必要な答えをすぐに見つけることができるオンデマンドのデジタルライブラリです。

定期購入すると、オンラインライブラリのどのページでも読んだり、ビデオを視聴したりできます。携帯電話やモバイル デバイスで本を読みましょう。印刷可能になる前に新しいタイトルにアクセスしたり、開発中の原稿に独占的にアクセスしたり、著者にフィードバックを投稿したりできます。コードサンプルのコピー アンド ペースト、お気に入りの整理、章のダウンロード、重要なセクションのブックマーク、メモの作成、ページの印刷など、多くの時間節約機能を利用できます。

O'Reilly Media は、この本を Safari Books Online サービスにアップロードしました。この本や、オンラインやその他の出版社が発行する同様のトピックに関する書籍に完全にデジタルでアクセスするには、次のサイトで無料でサインアップしてください。
<http://my.safaribooksonline.com>。

お問い合わせ方法

この本に関するコメントや質問は出版社に宛ててください。

オライリー・メディア株式会社
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (米国またはカナダ) 707-829-0515
(国際または国内)
707-829-0104 (ファックス)

この本の Web ページには、正誤表、例、その他の追加情報が記載されています。このページには次の場所からアクセスできます。

<http://shop.oreilly.com/product/9780596802301.do>

この本についてコメントしたり、技術的な質問をしたりするには、次の宛先に電子メールを送信してください。

bookquestions@oreilly.com

当社の書籍、コース、カンファレンス、ニュースの詳細については、次の Web サイトを参照してください。<http://www.oreilly.com>。

Facebook で私たちを見つけてください:<http://facebook.com/oreilly>

Twitter でフォローします：<http://twitter.com/oreillymedia>

YouTube で私たちをご覧ください:<http://www.youtube.com/oreillymedia>

謝辞

Alan Davidson、Josh Ehrlich、Rob Konigsberg、Archie Russell、Gabe W.、Asaph Zemach など、原稿全体のレビューに時間を割いてくださった同僚に感謝いたします。この本の間違いはすべて彼らのせいです（冗談です）。

Michael Hunger、George Heineman、Chuck Hudson をはじめ、本書のさまざまな草稿について詳細なフィードバックをくださった多くの査読者に感謝します。

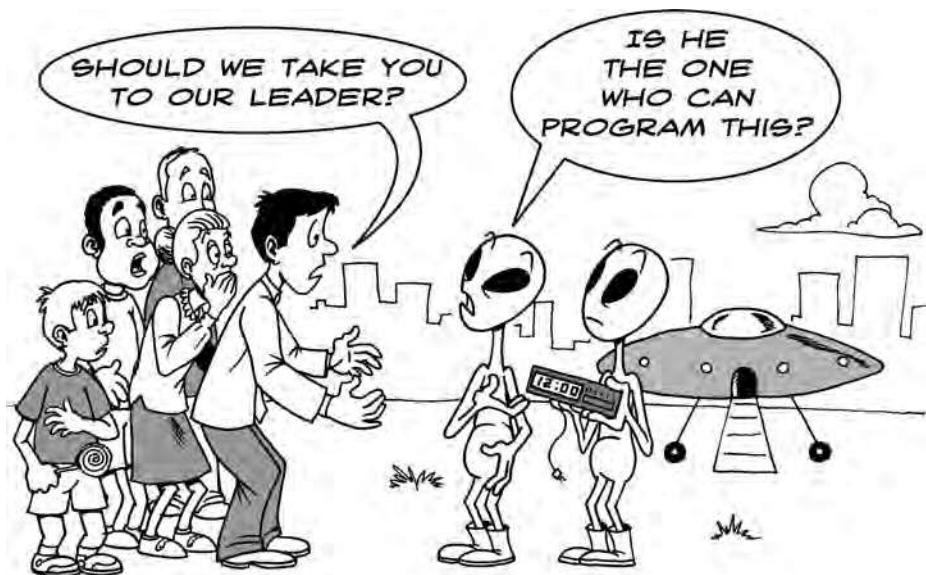
また、John Blackburn、Tim Dasilva、Dennis Geels、Steve Gerding、Chris Harris、Josh Hyman、Joel Ingram、Erik Mavrinac、Greg Miller、Anatole Paine、Nick White からも数多くのアイデアやフィードバックを得ました。オライリーの OFPS システムに関する私たちの草案をレビューしてくださった多数のオンラインコメント投稿者に感謝します。

終わりのない忍耐とサポートをしてくださったオライリーのチーム、特に Mary Treseler (編集者)、Teresa Elsey (制作編集者)、Nancy Kotary (コピー編集者)、Rob Romano (イラストレーター)、Jessica Hosman ('ツール)、および Abby Fox に感謝します。 ('ツール)。そして、私たちのクレイジーな漫画のアイデアを実現してくれた漫画家のデイブ・オールレッドにも感謝します。

最後に、途中で私たちを励まし、絶え間ないプログラミングの会話に耐えてくれた Melissa と Suzanne に感謝します。

第一章

コードは理解しやすいものでなければなりません



過去5年間にわたり、私たちは何百もの「悪いコード」の例(そのほとんどは私たち自身のもの)を収集し、何がそのコードを悪くしているのか、そしてそれを改善するためにどのような原則や技術が使用されたのかを分析してきました。私たちが気づいたのは、すべての原則が1つのテーマから生じているということです。

キー・アイデア

コードは理解しやすいものでなければなりません。

私たちは、これがコードの記述方法を決定する際に使用できる最も重要な指針であると信じています。この本全体を通じて、この原則を日常のコーディングのさまざまな側面に適用する方法を示します。ただし、始める前に、この原則について詳しく説明し、それがなぜそれほど重要であるかを説明します。

コードを「より良く」するには何が必要ですか?

ほとんどのプログラマー(作成者を含む)は直感と直観に基づいてプログラミングの決定を行います。誰もが次のようなコードを知っています。

```
for (ノード* ノード = リスト->先頭; ノード != NULL; ノード = ノード->次)
    Print(ノード->データ);
```

次のようなコードよりも優れています。

```
ノード* ノード = リスト->ヘッ
ド; if(ノード == NULL) を返す;

while (node->next != NULL) {
    Print(ノード->データ);
    ノード = ノード->次;
}
if(ノード != NULL) Print(ノード->データ);
```

(両方の例はまったく同じ動作をしますが)。

しかし多くの場合、それは難しい選択です。たとえば、次のコードは次のとおりです。

```
戻り指数 >= 0 ? 仮数 * (1 << 指数) : 仮数 / (1 << -指数);
```

以下よりも良いか悪いか:

```
if (指数 >= 0) {
    仮数 * (1 << 指数) を返します。} それ以
外 {
    仮数 / (1 << -指数) を返します。
}
```

最初のバージョンはよりコンパクトですが、2番目のバージョンはそれほど威圧的ではありません。どちらの基準がより重要ですか? 一般に、何かをコーディングする方法をどのように決定しますか?

可読性の基本定理

このような多くのコード例を研究した結果、読みやすさに関しては他のどの指標よりも重要な指標が1つあるという結論に達しました。これは非常に重要であるため、私たちはこれを「可読性の基本定理」と呼んでいます。

キーアイデア

コードは、他の人が理解するのにかかる時間を最小限に抑えるように作成する必要があります。

これはどういう意味でしょうか？文字通り、あなたの典型的な同僚を例に挙げ、コードを読んで理解するまでにどれくらいの時間がかかったかを測定した場合、この「理解までの時間」が最小化したい理論的指標になります。

そして、私たちが「理解する」と言うとき、この言葉には非常に高いハードルがあります。誰かのために全部わかるコードを変更したり、バグを見つけたり、コードの残りの部分とどのように相互作用するかを理解できる必要があります。

さて、あなたは次のように考えているかもしれません。他の人がそれを理解できるかどうかを誰が気にしますか？コードを使用しているのは私だけでしたとえ一人で取り組んでいるプロジェクトであっても、この目標を追求する価値はあります。その「別の誰か」はもしかしたらあなた6か月後、自分のコードが見慣れないものになったとき。そして、誰かがあなたのプロジェクトに参加するかもしれないし、あなたの「使い捨てコード」が別のプロジェクトに再利用されるかもしれないのです。

小さい方が常に良いのでしょうか？

一般的に言えば、問題を解決するために記述するコードが少ないほど、より良い結果が得られます（「[第13章 コードの記述量を減らす](#)」）。おそらく、5000行のクラスよりも2000行のクラスを理解するのにかかる時間は短くなります。

ただし、行数が少ないほど良いとは限りません。次のような1行の式が使用される場合がよくあります。

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

2行の場合よりも理解するのに時間がかかります。

```
バケット = FindBucket(キー);  
if (バケット != NULL) assert(!bucket->IsOccupied());
```

同様に、コメントを使用すると、ファイルに「コードを追加」する場合でも、コードをより早く理解できます。

```
// 「hash = (65539 * hash) + c」の高速バージョン  
hash = (hash << 6) + (hash << 16) - hash + c;
```

したがって、コードの行数を減らすことは良い目標ですが、理解するまでの時間を最小限に抑えることはさらに良い目標です。

理解までの時間は他の目標と矛盾しますか?

あなたはこう思っているかもしれません、コードを効率的にする、適切に設計する、テストを容易にするなどの他の制約についてはどうでしょうか?これらは、コードを理解しやすくするという欲求と矛盾することがあるのではないか?

これらの他の目標はまったく干渉しないことがわかりました。高度に最適化されたコードの領域であっても、コードを可読性の高いものにする方法はまだあります。そして、コードを理解しやすくすると、多くの場合、コードが適切に設計され、テストしやすくなります。

この本の残りの部分では、さまざまな状況で「読みやすさ」を適用する方法について説明します。ただし、疑問がある場合は、可読性の基本定理が本書の他のルールや原則よりも優先されることを思い出してください。また、一部のプログラマは、完全に因数分解されていないコードを修正する必要に迫られることがあります。一歩下がって自問することが常に重要です。このコードは理解しやすいですか?そうであれば、おそらく他のコードに移っても問題ありません。

難しい部分

はい、想像上の部外者があなたのコードを理解しやすいと感じるかどうかを常に考えるには、余分な作業が必要です。そうするためにには、これまでコーディング中にオンになっていなかったかもしれない脳の一部をオンにする必要があります。

しかし、(私たちがそうしてきたように)この目標を採用すれば、あなたはより優れたプログラマーになり、バグが減り、自分の仕事にもっと誇りを持ち、周囲の誰もが使いたがるコードを作成できるようになると確信しています。それでは始めましょう!

パート

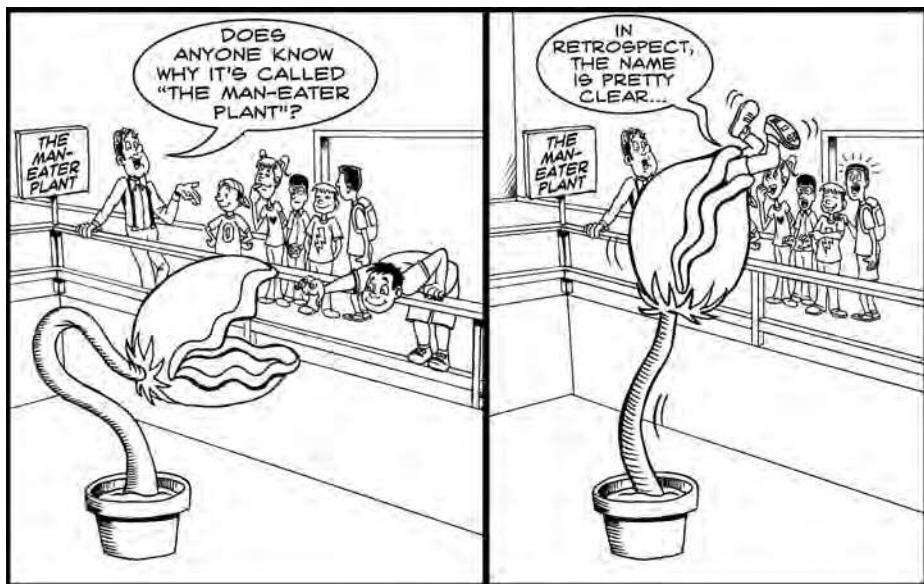
表面レベルの改善

読みやすさのツアーハンズは、適切な名前を選択し、適切なコメントを書き、コードをきちんとフォーマットするなど、「表面レベル」の改善と考えられることから始まります。このような種類の変更は簡単に適用できます。コードをリファクタリングしたり、プログラムの実行方法を変更したりすることなく、それらを「適切な場所」で作成できます。多大な時間をかけずに段階的に作成することもできます。

これらのトピックは非常に重要です。コードベース内のコードのすべての行に影響を与えます。それらの変更は小さいように見えますが、合計するとコードベースに大きな改善をもたらすことができます。コードに素晴らしい名前があり、よく書かれたコメントがあり、空白がきれいに使用されていれば、コードは次のようになります。多くの読みやすくなります。

もちろん、読みやすさに関しては、表面レベルではさらに多くのことがあります（これについては、本書の後半で説明します）。ただし、この部分の内容は、少しの労力で非常に広く応用できるため、最初に取り上げる価値があります。

情報を名前に詰め込む



変数、関数、クラスのいずれに名前を付ける場合でも、多くの同じ原則が適用されます。私たちは名前を小さなコメントとして考えるのが好きです。スペースがあまりなくとも、適切な名前を選択することで多くの情報を伝えることができます。

キーイデア

名前に情報を詰め込みます。

プログラムで目にする名前の多くはあいまいです。一時。一見合理的と思われる言葉であっても、サイズまたは得る、あまり情報を詰め込まないでください。この章では、そのような名前を選択する方法を説明します。

この章は、6つの特定のトピックで構成されています。

- 特定の単語を選択する
- 一般的な名前を避ける（または、それをいつ使用するかを知る）
- 抽象的な名前ではなく具体的な名前を使用する
- サフィックスまたはプレフィックスを使用して名前に追加情報を付加する
- 名前の長さを決める
- 名前の書式設定を使用して追加情報を詰め込む

特定の単語を選択する

「名前に情報を詰め込む」ことの一環として、非常に具体的な単語を選択し、「空の」単語を避けることが挙げられます。

たとえば、次の例にあるように、「get」という単語は非常に具体的ではありません。

確かにページの取得(URL):

◦◦◦

「得る」という言葉はあまり多くを言いません。このメソッドはローカルキャッシュ、データベース、またはインターネットからページを取得しますか？インターネットからのものであれば、より具体的な名前は次のようにになります。

FetchPage()またはダウンロードページ()。

以下にその例を示しますバイナリツリークラス：

```
クラス BinaryTree {  
    整数サイズ();  
    ◦◦◦  
};
```

何を期待しますか？サイズ()返す方法は？ツリーの高さ、ノードの数、それともツリーのメモリフットプリントでしょうか？

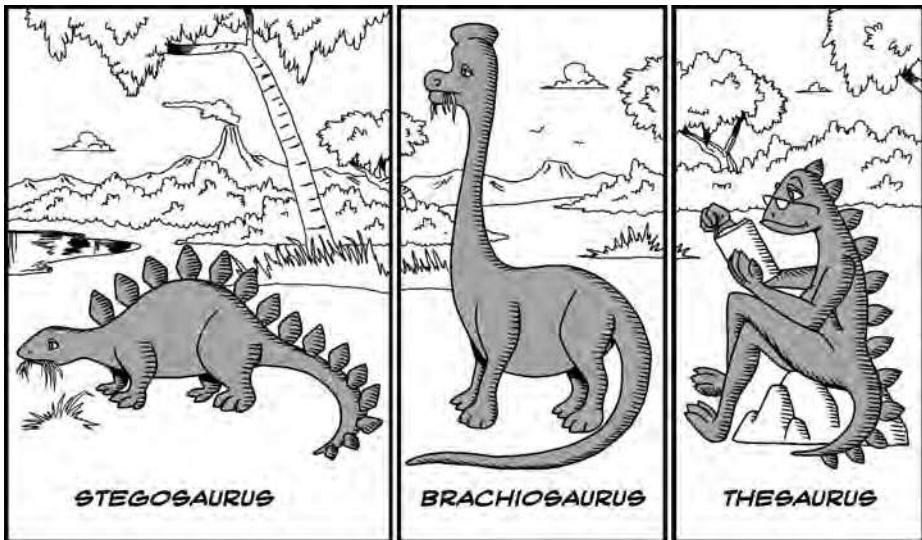
問題はそれですサイズ()あまり情報を伝えません。より具体的な名前は次のようにになります
高さ()、ノード数()、またはメモリバイト()。

別の例として、ある種のものがあるとします。系クラス：

```
クラススレッド{  
    空所停止();  
    ...  
};
```

名前停止() これは問題ありませんが、正確に何を行うかによっては、より具体的な名前が存在する可能性があります。たとえば、次のように呼ぶことができます。殺す() 代わりに、元に戻すことができない重い操作の場合に使用します。あるいは、そう呼ぶかもしれません一時停止()、方法があるなら再開する() それ。

もっと「カラフル」な言葉を探して



ためらわずに類語辞典を使用したり、友人にもっと良い名前の提案を聞いたりしてください。英語は豊かな言語であり、選択できる単語がたくさんあります。

以下に単語の例と、あなたの状況に当てはまりそうな、より「カラフルな」バージョンをいくつか示します。

言葉	代替案
送信	届ける、発送する、発表する、配布する、ルートする
探す	検索、抽出、特定、回復
始める	起動、作成、開始、開く
作る	作成、セットアップ、ビルド、生成、構成、追加、新規

ただし、調子に乗らないでください。PHPには、次のような関数があります。爆発()文字列。カラフルな名前で、何かを粉々に碎く様子をよく表していますが、どう違うのでしょうか？

からスプリット () ?(2つの機能は異なりますが、名前からその違いを推測するのは困難です)。

キー・アイデア

かわいらしいことよりも、はっきりしていて正確なことのほうがいいのです。

tmp や retval などの一般的な名前を避ける

のような名前tmp、retval、そしてふー通常、「名前が思いつきません」という意味の警察です。このように空の名前を使用する代わりに、**エンティティの値**または**目的を説明する名前**を選択します。

たとえば、次の JavaScript 関数を使用します。戻り値:

```
var euclidean_norm = 関数 (v) {  
    变数戻り値=0.0;  
    for (var i = 0; i < v.length; i += 1)  
        戻り値+=v[i] * v[i]; return  
    Math.sqrt(戻り値);  
};
```

使いたくなる戻り値戻り値にもっと良い名前が思いつかないとき。しかし

戻り値には、「戻り値である」以外の情報はほとんど含まれていません(これは通常、明らかです)。

より適切な名前は、変数の目的または変数に含まれる値を説明するものになります。この場合、変数は次の二乗和を累積しています。v.したがって、より良い名前はsum_squares。これにより、変数の目的が事前に通知され、バグの発見に役立つ可能性があります。

たとえば、ループ内が誤って次のようになった場合を想像してください。

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

このバグは、名前が次のとおりであるとより明白になります。平方和:

```
和二乗+=v[i]; // 合計する「平方」はどこですか? バグ!
```

アドバイス

名前戻り値あまり情報が詰め込まれていません。代わりに、変数の値を説明する名前を使用してください。

ただし、一般的な名前が意味を持つ場合もあります。どのような場合にそれらを使用するのが理にかなっているかを見てみましょう。

一時

2つの変数を交換する典型的なケースを考えてみましょう。

```
if(右<左){  
    一時=右;  
    右=左;  
    左=一時;  
}
```

このような場合の名前は、一時全く問題ありません。この変数の唯一の目的は一時記憶域であり、有効期間はわずか数行です。名前一時この変数には他の役割がないという特定の意味が読者に伝わります。他の関数に渡されたり、リセットされたり、複数回再利用されたりすることはありません。

しかし、ここで次のようなケースがあります。一時単なる怠惰から使用されています：

```
弦一時=ユーザー名();一時+=" "+  
user.phone_number();一時+=" "+  
user.email();。。。  
  
template.set("ユーザー情報",一時);
```

この変数の有効期間は短いですが、一時的な保存場所であることは、この変数に関して最も重要なことではありません。代わりに、次のような名前ユーザー情報のほうがわかりやすいでしょう。

次の場合、一時名前に含める必要がありますが、一部それの：

```
tmp_file=tempfile.NamedTemporaryFile()。。。  
  
セーブデータ(tmp_file, ...)
```

変数に名前を付けたことに注目してくださいtmp_fileそれだけではありません一時、ファイルオブジェクトだからです。ちょうどそれを呼び出した場合を想像してください一時：

```
セーブデータ(一時, ...)
```

この1行のコードだけを見ても、次のことは明らかではありません。一時ファイル、ファイル名、あるいは書き込まれるデータの場合もあります。

アドバイス

名前一時は、有効期間が短く一時的であることがその変数に関する最も重要な事実である場合にのみ使用する必要があります。

ループ反復子

のような名前i、j、イター、そしてそれインデックスおよびループ反復子として一般的に使用されます。これらの名前は一般的なものです、「私は反復子です」という意味であると理解されています。(実際、これらの名前のいずれかをいくつかの目的で使用した場合、他の混乱を招くことになるので、やめてください。)

しかし、場合によっては、それよりも優れたイテレータ名が存在します。私、j、そしてk。たとえば、次のループでは、どのユーザーがどのクラブに所属しているかを検索します。

```
for (int私=0;私<クラブのサイズ();私++)
    for (intj=0;j<club[i].members.size();j++)
        for (intk=0;k<ユーザーのサイズ();k++)
            if (クラブ[私].メンバー[k] == ユーザー[j])
                cout << "user[" << j << "]クラブにいます[" << 私 << "]"
                << 終わり;
```

の中にもし声明、メンバー[]そしてユーザー[]間違ったインデックスを使用しています。このようなバグは、そのコード行だけでは問題ないように見えるため、発見するのが困難です。

```
if (クラブ[i].メンバー[k] == ユーザー[j])
```

この場合、より正確な名前を使用することで解決した可能性があります。ループインデックスに名前を付ける代わりに(i、j、k)、別の選択肢は(クラブ_i、メンバー_i、ユーザー_i)または、より簡潔に言うと(シ、ミ、ワイ)。このアプローチは、バグをより目立たせるのに役立ちます。

```
if (クラブ[ci].メンバー[ワイ] == ユーザー[私]) # バグ！最初の文字が一致しません。
```

正しく使用すると、インデックスの最初の文字が配列の最初の文字と一致します。

```
if (clubs[ci].members[mi] == users[ui]) # OK。最初の文字が一致します。
```

一般名に関する判決

これまで見てきたように、総称名が役立つ状況がいくつかあります。

アドバイス

のような一般的な名前を使用する場合は、tmp、それ、または戻り値、そうするのには十分な理由があります。

多くの場合、それらは純粋な怠惰から過剰に使用されます。これは当然のことです。他に良いものが思い浮かばない場合は、次のような意味のない名前を使用する方が簡単です。ふーそして次に進みます。しかし、良い名前を考えるためにさらに数秒かかる習慣を身につければ、「ネーミングの筋肉」がすぐに鍛えられることに気づくでしょう。

抽象的な名前より具体的な名前を好む



変数、関数、その他の要素に名前を付ける場合は、抽象的ではなく具体的に説明します。

たとえば、という名前の内部メソッドがあるとします。ServerCanStart()、これは、サーバーが特定のTCP/IP ポートでリッスンできるかどうかをテストします。名前ServerCanStart()やや抽象的ですが。より具体的な名前は次のとおりですCanListenOnPort()。この名前は、メソッドが何を実行するかを直接説明します。

次の 2 つの例では、この概念をさらに詳しく説明します。

例: DISALLOW_EVIL_CONSTRUCTORS

以下は Google のコードベースの例です。C++ では、クラスにコピー コンストラクターまたは代入演算子を定義しない場合、デフォルトが提供されます。これらの方は便利ではありますが、

これらは、ユーザーが気づいていない場所で「舞台裏」で実行されるため、メモリリークやその他の事故を簡単に引き起こす可能性があります。

その結果、Googleはマクロを使用して、これらの「邪悪な」コンストラクターを禁止する規約を設けています。

```
クラス クラス名 {  
    プライベート：  
        DISALLOW_EVIL_CONSTRUCTORS (クラス名) ;  
  
    公共：  
        . . .  
};
```

このマクロは次のように定義されました。

```
# 定義するDISALLOW_EVIL_CONSTRUCTORS(クラス名)  
    \ クラス名(const クラス名&); \  
    void 演算子=(const クラス名&);
```

このマクロをプライベート：クラスのセクションに追加すると、これら2つのメソッドはプライベートになり、たとえ誤ってでも使用できなくなります。

名前DISALLOW_EVIL_CONSTRUCTORSあまり良くないけど。「悪」という言葉の使用は、議論の余地のある問題に対する過度に強い姿勢を伝えます。さらに重要なのは、そのマクロが何を禁止しているのかが明らかでないことです。を禁止します。演算子=()メソッド、そしてそれは「コンストラクター」ですらない!

この名前は何年にもわたって使用されましたが、最終的にはそれほど挑発的ではなく、より具体的な名前に置き換えられました。

```
# 定義するDISALLOW_COPY_AND_ASSIGN (クラス名) ...
```

例: --run_locally

私たちのプログラムの1つは--という名前のオプションのコマンドラインフラグを持っていました。ローカルで実行します。このフラグにより、プログラムは追加のデバッグ情報を出力しますが、実行速度は遅くなります。このフラグは通常、ラップトップなどのローカルマシンでテストするときに使用されます。ただし、プログラムがリモートサーバーで実行されている場合はパフォーマンスが重要であるため、フラグは使用されませんでした。

名前の由来がわかります--run_locallyが誕生しましたが、いくつかの問題があります。

- チームの新しいメンバーは、それが何をするのか知りませんでした。彼はローカルで実行するときにそれを使用していましたが(想像してください)、なぜそれが必要なのかはわかりませんでした。
- プログラムがリモートで実行されているときに、デバッグ情報を出力する必要がある場合がありました。通り過ぎる--run_locallyリモートで実行されているプログラムはおかしく見えますが、混乱を招くだけです。
- パフォーマンステストをローカルで実行することがありますが、ロギングによってテストの速度が低下することを望まないため、--を使用しませんでした。ローカルで実行します。

問題はそれです--run_locally通常使用される状況にちなんで名付けられました。代わりに、--のようなフラグ名を使用します。追加のロギングより直接的かつ明確になります。

しかし、もしとしたら --run_locally 追加のロギング以上の作業が必要ですか? たとえば、特別なローカルデータベースをセットアップして使用する必要があるとします。さて名前は --run_locally これらの両方を同時に制御できるため、より魅力的に思えます。

しかし、その目的でそれを使うと、名前を選ぶことになりますなぜならそれは曖昧で間接的であり、おそらく良いアイデアではありません。より良い解決策は、という名前の 2 番目のフラグを作成することです。

--use_local_database。ここでは 2 つのフラグを使用する必要がありますが、これらのフラグはより明示的です。2 つの直交するアイデアを 1 つにまとめようとするものではなく、一方のみを使用し、他方は使用しないというオプションを提供します。

名前に追加情報を付加する



前に述べたように、変数の名前は小さなコメントのようなものです。スペースがあまりない場合でも、名前に詰め込んだ追加情報は、変数が表示されるたびに表示されます。

したがって、変数に関して読者が知っておくべき非常に重要な点がある場合は、名前に追加の「単語」を追加する価値があります。たとえば、16進文字列を含む変数があるとします。

文字列ID; // 例: "af84ef845cd8"

名前を付けてもいいかもしれませんhex_id代わりに、読者がIDの形式を覚えておくことが重要な場合に使用します。

単位付きの値

変数が測定値(時間やバイト数など)である場合、変数名に単位をエンコードすると便利です。

たとえば、Webページの読み込み時間を測定するJavaScriptコードを次に示します。

```
変数始める= (new Date()).getTime(); // ページの先頭。。。
```

```
変数経過= (new Date()).getTime() -始める; // ページの下部
document.writeln("読み込み時間: " +経過+ "秒");
```

このコードには明らかに問題はありませんが、機能しません。時間をもらう()秒ではなくミリ秒を返します。

_を追加することでMS変数に追加すると、すべてをより明示的にすることができます。

```
変数start_ms= (new Date()).getTime(); // ページの先頭。。。
```

```
変数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、プログラミングで登場する単位はたくさんあります。以下は、単位のない関数パラメーターの表と、単位を含むより良いバージョンです。

関数パラメータ	単位をエンコードするためのパラメータの名前変更
Start(int遅れ)	遅延→遅延秒数
CreateCache(intサイズ)	サイズ→サイズ_mb
スロットルダウンロード(フロート)限界	限界→max_kbps
回転(フロート)角度	角度→度_CW

その他の重要な属性のエンコード

名前に追加情報を付加するこの手法は、単位を含む値に限定されません。変数に何か危険な点や驚くべき点がある場合は常にこれを実行する必要があります。

たとえば、多くのセキュリティエクスプロイトは、プログラムが受け取るデータの一部がまだ安全な状態ではないことに気づかないことから発生します。このためには、次のような変数名を使用するとよいでしょう。

信頼できないURLまたはunsafeメッセージボディ。安全でない入力をクレンジングする関数を呼び出した後、結果の変数は次のようにになります。信頼できるURLまたは安全なメッセージ本文。

次の表は、名前に追加情報をエンコードする必要がある場合の追加の例を示しています。

状況	変数名	もっと良い名前
あパスワード「平文」であり、さらに処理する前に暗号化する必要があります	パスワード	平文_パスワード
ユーザーが提供するコメント表示される前にエスケープする必要がある	コメント	エスケープされていない_コメント
のバイトhtmlUTF-8に変換されています	html	html_utf8
着信データ「URLエンコード」されている	データ	データ_urlenc

のような属性は使用しないでください。エスケープされていない_または_utf8のために毎プログラム内の変数。これらは、誰かが変数の内容を間違えた場合にバグが簡単に侵入する可能性がある場所、特にセキュリティバグのように悲惨な結果が生じる場合に最も重要です。基本的に、理解することが重要なことである場合は、それを名前に含めます。

これはハンガリー語表記ですか?

[ハンガリー語表記](#)は、Microsoft 社内で広く使用されている命名システムです。すべての変数の「型」を名前のプレフィックスにエンコードします。ここではいくつかの例を示します。

名前	意味
最後	何らかのデータ構造の最後の要素へのポインタ (p)
pszバッファ	ゼロで終わる (z) 文字列 (s) バッファへのポインタ (p)
ち	文字数 (c) (ch)
mpcpx	色へのポインタ (pco) から x 軸の長さ (px) へのポインタへのマップ (m)

まさに「名前に属性を付与する」例です。ただし、これは、特定の属性セットのエンコードに重点を置いた、より正式で厳密なシステムです。

このセクションで私たちが主張しているのは、より広範で非公式なシステムです。つまり、変数の重要な属性を特定し、必要であればそれらを読みやすくエンコードするというものです。「英語表記」と呼んでもいいかもしれません。

名前の長さはどれくらいにすべきですか?



適切な名前を選択する場合、名前が長すぎてはいけないという暗黙の制約があります。次のような識別子を扱いたくない人はいません。

`newNavigationControllerWrappingViewControllerForDataSourceOfClass`

名前が長ければ長いほど覚えにくくなり、画面上でより多くのスペースを消費し、余分な行が折り返される可能性があります。

一方、プログラマはこのアドバイスを行き過ぎて、単一単語 (または単一文字) の名前のみを使用する可能性があります。では、このトレードオフをどのように管理すべきでしょうか? ネーミングはどうやって決めますか
変数d、日、またはlast_update からの日数?

この決定は判断に基づくものであり、その最良の答えは、その変数がどのように使用されているかに正確に依存します。ただし、決定に役立つガイドラインがいくつかあります。

スコープが短い場合は、短い名前を使用しても問題ありません

短い休暇に行くときは、通常、長い休暇に行くときよりも荷物が少なくなります。同様に、「スコープ」 (この名前を「認識」できる他のコード行数) が小さい識別子には、それほど多くの情報を取り扱う必要はありません。つまり、すべての情報 (変数の型、初期値、破棄方法) がわかりやすいため、短い名前を使用することができます。

```

if (デバッグ) {
    マップ<文字列,整数>メートル;
    LookUpNamesNumbers(&メートル); 印
    刷(メートル);
}

```

それでもメートル情報は何も詰め込まれていませんが、読者はこのコードを理解するために必要な情報をすべてすでに持っているため、問題にはなりません。

ただし、次のように仮定します。メートルがクラスメンバーまたはグローバル変数であった場合、次のコードスニペットが表示されました。

```

LookUpNamesNumbers(&メートル); 印
刷(メートル);

```

このコードは、コードの種類や目的が不明瞭であるため、非常に読みにくくなっています。メートルは。

したがって、識別子の有効範囲が広い場合、名前にはそれを明確にするのに十分な情報が含まれている必要があります。

長い名前の入力 - もう問題ありません

長い名前を避ける正当な理由はたくさんありますが、「入力するのが難しい」という理由は、もはやその1つではありません。私たちがこれまで見てきたすべてのプログラミングテキストエディタには、「単語補完」が組み込まれています。驚くべきことに、ほとんどのプログラマーはこの機能に気づいていません。エディタでこの機能をまだ試していない場合は、今すぐこの本を置いて試してみてください。

1. 名前の最初の数文字を入力します。
2. 単語補完コマンドをトリガーします(以下を参照)。
3. 完成した単語が正しくない場合は、正しい名前が表示されるまでコマンドをトリガーし続けます。

驚くほど正確です。あらゆる種類のファイル、あらゆる言語で動作します。また、コメントを入力している場合でも、どのトークンでも機能します。

編集者	指示
ヴィ	Ctrl+P
Emacs	メタ/(打つESC、それから /)
日食	Alt- /
インテリJアイデア	Alt- /
テキストメイト	ESC

頭字語と略語

プログラマーは、名前を小さくするために頭字語や略語を使用することがあります(たとえば、クラスに名前を付けるなど)。BEMマネージャーの代わりにバックエンドマネージャー。この縮小は潜在的な混乱を招く価値があるのでしょうか?

私たちの経験では、プロジェクト固有の略語は通常悪い考えです。プロジェクトに初めて参加する人にとって、それらは不可解で威圧的に見えます。十分な時間が経つと、それらは著者にとって不可解で威圧的なものにさえ見え始めます。

したがって、私たちの経験則は次のとおりです。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？ そうであれば、おそらく大丈夫です。

たとえば、プログラマーが使用するのはかなり一般的です。評価の代わりに評価、ドキュメントの代わりにドキュメント、文字列の代わりに弦。それで、新しいチームメイトが見ていますFormatStr()おそらくそれが何を意味するか理解するでしょう。しかし、おそらく彼または彼女は、それが何なのか理解できないでしょう。

BEMマネージャーは。

不要な言葉を捨てる

名前内の単語は、情報をまったく失わずに削除できる場合があります。たとえば、代わりにConvertToString()、名前ToString()より小さくなり、実際の情報が失われることはありません。同様に、代わりにDoServeLoop()、名前サーブループ()同様に明らかです。

意味を伝えるために名前の書式を使用する

アンダースコア、ダッシュ、大文字の使用方法によっても、名前にさらに多くの情報を詰め込むことができます。たとえば、次のような C++ コードを次に示します。Google オープンソース プロジェクトで使用される書式設定規則：

```
静态定数整数kMaxOpenFiles=100;

クラスログリーダー{
    公共：
        空所ファイルを開く (弦ローカルファイル);

    プライベート：
        整数オフセット_;
        DISALLOW_COPY_AND_ASSIGN(ログリーダー);
};
```

エンティティごとに異なる形式を使用することは、一種の構文強調表示のようなもので、コードを読みやすくするのに役立ちます。

この例の書式設定のほとんどは非常に一般的なものです。キャメルケースクラス名に使用し、下部分離変数名の場合。しかし、その他の規則には驚かれた方もいるかもしれません。

たとえば、定数値は次の形式になります。k定数名の代わりにCONSTANT_NAME。このスタイルには # と簡単に区別できるという利点があります。定義するマクロ、つまりMACRO_NAME慣例により。

クラスメンバー変数は通常の変数と似ていますが、次のようにアンダースコアで終わる必要があります。オフセット_。最初は、この規則は奇妙に思えるかもしれません、瞬時に区別することができます。

他の変数のメンバーは非常に便利です。たとえば、大規模なメソッドのコードをざっと眺めていて、次の行があるとします。

```
統計。クリア () ;
```

普通は不思議に思うかも知れませんが、する統計このクラスに属しますか？このコードはクラスの内部状態を変更していますか？もしメンバー_慣例が使用されているため、すぐに結論付けることができます。いいえ、統計ローカル変数でなければなりません。そうでなければ、名前が付けられます統計_。

その他の書式設定規則

プロジェクトまたは言語のコンテキストによっては、名前にさらに多くの情報を含めるために使用できる他の書式規則がある場合があります。

たとえば、[JavaScript: 良い部分](#)(Douglas Crockford、O'Reilly、2008年)、著者は「コンストラクター」(関数を使用して呼び出すことを意図したもの)を提案しています。新しい)は大文字にする必要があり、通常の関数は小文字で始める必要があります。

```
var x = 新しい日付ピッカー(); // DatePicker() は「コンストラクター」関数です //  
変数y = ページの高さ(); pageHeight() は通常の関数です
```

別のJavaScriptの例を次に示します。jQueryライブラリ関数(名前は1文字の\$)を呼び出すとき、便利な規則は、jQueryの結果の先頭にも\$を付けることです。

```
変数$すべての画像 = $("img"); // $all_images はjQueryオブジェクトです // 高さは  
var height = 250; そうではありません
```

コード全体を通して、\$が次のことであることは明らかです。すべての画像jQueryの結果オブジェクトです。

これが最後の例です。今回はHTML/CSSに関するものです。HTMLタグにIDまたはクラス属性では、アンダースコアとダッシュの両方が値に使用できる有効な文字です。考えられる規則の1つは、ID内の単語を区切るにはアンダースコアを使用し、クラス内の単語を区切るにはダッシュを使用することです。

```
<div id=中列「クラス=」メインコンテンツ> ...
```

このような規則を使用するかどうかは、あなたとあなたのチーム次第です。ただし、どのシステムを使用する場合でも、プロジェクト全体で一貫性を保つ必要があります。

まとめ

この章の単一のテーマは次のとおりです。自分の名前に情報を詰め込む。これは、読者が名前を読むだけで多くの情報を抽出できることを意味します。

ここで取り上げた具体的なヒントをいくつか紹介します。

- **具体的な言葉を使う**—たとえば、代わりに得る、のような言葉フェッチまたはダウンロード文脈によっては、その方が良いかもしれません。
- **一般的な名前を避ける**のように一時そして戻り値、特別に使用する理由がない限り。

- **具体的な名前を使用する**物事をより詳細に説明するもの、つまり名前ServerCanStart()は比べて曖昧CanListenOnPort()。
- **重要な詳細情報を添付してください**い変数名にーたとえば、_を追加しますMS値がミリ秒単位の変数に代入するか、先頭に追加します生_エスケープが必要な未処理の変数に。
- **より大きなスコープには長い名前を使用します**-複数の画面にまたがる変数に、不可解な1文字または2文字の名前を使用しないでください。数行しか含まれない変数には、短い名前の方が適しています。
- **大文字やアンダースコアなどを意味のある方法で使用する**ーたとえば、クラス メンバーに「_」を追加して、ローカル変数と区別できます。

誤解されない名前



前の章では、名前に多くの情報を含める方法について説明しました。この章では、別のトピック、つまり誤解される可能性のある名前に注意することに焦点を当てます。

キーアイデア

「誰かがこの名前から他にどのような意味を解釈できるだろうか？」と自問して、自分の名前を積極的に精査してください。

ここでは、積極的に「間違った解釈」を模索するなど、創造性を發揮するよう努めてください。この手順は、曖昧な名前を見つけて変更できるようにするのに役立ちます。

この章の例では、目にするそれぞれの名前の誤解について議論しながら「声に出して考え」、より良い名前を選択します。

例: Filter()

一連のデータベース結果を操作するコードを作成しているとします。

結果=データベース.all_objects。フィルター(「年 <= 2011」)

どういうことですか結果今は含まれていますか？

- 年が 2011 年以下のオブジェクト？
- 年が次のオブジェクトない<=2011年？

問題はそれですフィルターは曖昧な言葉です。「選ぶ」という意味なのか「取り除く」という意味なのかは不明です。名前は避けたほうがいいよフィルターとても誤解されやすいからです。

「選び出したい」場合は、次の名前が適しています。選択する ()。「排除したい」場合は、より適切な名前を付けますは除外()。

例: クリップ(テキスト、長さ)

段落の内容をクリップする関数があるとします。

```
# テキストの末尾を切り落とし、「...」を追加します def ク  
リップ(テキスト、長さ):  
    ...
```

想像できる方法は 2 つありますクリップ () 動作します:

- 削除します長さ端から
- 最大値まで切り捨てられます。長さ

2 番目の方法(切り捨て)が考えられますが、確実なことはわかりません。読者にやっかいな疑問を抱かせるよりは、関数に名前を付けたほうがよいでしょう。Truncate(テキスト、長さ)。

ただし、パラメータ名は長さも責任がある。もしそうだったら最大長、そうすればさらに明確になります。

しかし、まだ終わっていません。名前最大長まだ複数の解釈が残っています：

- バイト数
- 文字数
- 単語の数

前の章で見たように、これは名前に単位を付ける必要がある場合です。この場合は「文字数」を意味するので、代わりに最大長、そのはず

max_chars。

(包括的な) 制限の最小値と最大値を優先する

ショッピング カート アプリケーションで、ユーザーが一度に 10 個以上の商品を購入できないようにする必要があるとします。

```
CART_TOO_BIG_LIMIT = 10

shopping_cart.num_items() >= CART_TOO_BIG_LIMIT の場合:
    Error(「カート内の商品が多すぎます。」)
```

このコードには古典的な off-by-one バグがあります。`>=` を `>` に変更することで簡単に修正できます。

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

(または再定義することでCART_TOO_BIG_LIMITに11)。しかし、根本的な問題は、CART_TOO_BIG_LIMITは曖昧な名前です。「～まで」を意味するのか、「～まで」を意味するのかが明確ではありません。

アドバイス

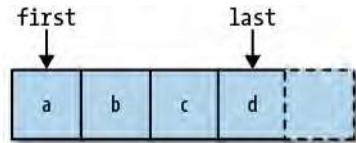
制限に名前を付ける最も明確な方法は、最大_またはmin_物事が制限される前に。

この場合、名前は次のようにする必要があります。MAX_ITEMS_IN_CART。新しいコードはシンプルで明確です。

```
MAX_ITEMS_IN_CART = 10

if shopping_cart.num_items() > MAX_ITEMS_IN_CART:
    Error(「カート内の商品が多すぎます。」)
```

包含範囲の最初と最後を優先する



以下は、「～まで」なのか「～まで」なのか、「～まで」なのかがわからない別の例です。

```
print integer_range(start=2,停止=4)  
# これは [2,3] または [2,3,4] (またはその他) を出力しますか?
```

それでも始めるは適切なパラメータ名です。停止ここでは複数の方法で解釈できます。

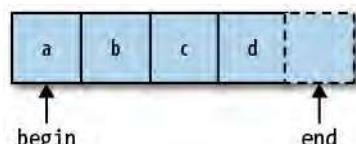
のために包括的このような範囲(範囲には両方の終点が含まれる必要があります)、良い選択は次のとおりです。最初の最後。例えば：

```
set.PrintKeys(最初="パート"、最後="マギー")
```

とは異なり停止、言葉最後明らかに包括的です。

に加えて最初の最後、名最小/最大また、そのコンテキストで「適切に聞こえる」と仮定すると、包括的な範囲でも機能する可能性があります。

包含/排他範囲の開始と終了を優先する



実際には、多くの場合、包含/排他範囲を使用する方が便利です。たとえば、10月16日に発生したすべてのイベントを出力したい場合は、次のように記述する方が簡単です。

```
PrintEventsInRange("10月16日前12時",「10月17日前12時」)
```

次のように書くよりも、

```
PrintEventsInRange("10月16日前12時",「10月16日 11:59:59.9999pm」)
```

では、これらのパラメータに適切な名前のペアはでどうか? 包括的/排他的範囲に名前を付けるための一般的なプログラミング規則は次のとおりです。始まり/終わり。

しかし、その言葉は終わり少し曖昧です。たとえば、「本の終わりにいます」という文では、「終わり」は包括的です。残念ながら、英語には「最後の値を過ぎたところ」を表す簡潔な言葉がありません。

なぜなら始まり/終わりは非常に慣用的であるため (少なくとも、C++ の標準ライブラリではこの方法で使用されており、配列をこの方法で「スライス」する必要があるほとんどの場所で使用されています)、これが最良のオプションです。

ブール値の命名

ブール変数またはブール値を返す関数の名前を選択するときは、それが何であるかを明確にしてください。真実そして間違い本当に意味。

以下に危険な例を示します。

ブール読み取りパスワード=真実;

読み方によっては (ダジャレではありません)、大きく異なる 2 つの解釈があります。

- 私たちは必要パスワードを読むには
- パスワードはすでに設定されていますされた読む

この場合、「read」という単語を避け、名前を付けるのが最善です。パスワードが必要または user_is_authenticated その代わり。

一般に、次のような単語を追加すると、ある、持っている、できる、またはすべきブール値をより明確にすることができます。

たとえば、次のような関数スペース左()数値を返す可能性があるようです。ブール値を返すことを意図している場合、より良い名前は次のようにになります。SpaceLeft() があります。

最後に、避けたほうが良いのは、否定された名前に含まれる用語。たとえば、次の代わりに:

ブール無効化_SSL=間違い;

次のように言うと読みやすくなります (そしてよりコンパクトになります)。

ブールuse_ssl=真実;

ユーザーの期待に応える

名前によっては、たとえ別の意味を込めていたとしても、ユーザーがその名前の意味について先入観を持っているため、誤解を招くものもあります。このような場合は、誤解を招かないように「あきらめて」名前を変更するのが最善です。

例: get^{*}()

多くのプログラマは、メソッドが次で始まるという慣例に慣れています。得る単に内部メンバーを返す「軽量アクセサー」です。この慣例に反すると、ユーザーに誤解を与える可能性があります。

Java での例を次に示します。ないすること:

```
パブリック クラス StatisticsCollector {  
    public void addSample(double x) { ... }  
  
    パブリックダブル平均値を取得する(){  
        // すべてのサンプルを反復処理し、合計 / num_samples を返します。  
    }  
    ...  
}
```

この場合、実装はgetMean()過去のデータを反復処理し、その場で平均を計算することです。データが大量にある場合、このステップは非常にコストがかかる可能性があります。しかし、何の疑いも持たないプログラマーは次のように呼びかけるかもしれません。getMean()安価な通話だと思い込んで、うっかりしていました。

代わりに、メソッドの名前を次のように変更する必要があります。computeMean()、それはむしろ高価な操作のように思えます。(あるいは、実際に軽量な操作になるように再実装する必要があります。)

例: list::size()

C++ 標準ライブラリの例を次に示します。次のコードは、サーバーの 1 つを最高速度まで低下させる、非常に見つけにくいバグの原因でした。

```
void ShrinkList(list<Node*>& list, int max_size) {  
    その間 (リスト.サイズ()>最大サイズ){  
        FreeNode(list.back());  
        list.pop_back();  
    }  
}
```

「バグ」とは、作者がそれを知らなかったということですリスト.サイズ()ですの上) 操作 - 事前に計算されたカウントを返すのではなく、リンク リストをノードごとにカウントします。シュリンクリスト()のの上)手術。

コードは技術的には「正しく」、実際にすべての単体テストに合格しました。でもいつシュリンクリスト()100 万個の要素を含むリストで呼び出されたため、完了するまでに 1 時間以上かかりました。

もしかしたら、あなたは「それは発信者のせいだ。もっとドキュメントを注意深く読むべきだった。」と考えているかもしれません。それは事実ですが、この場合、リスト.サイズ()定数時間の操作ではありません驚くべき。C++ の他のすべてのコンテナーには定数時間があります。

サイズ () 方法。

持っていたサイズ () 指名されましたcountSize()またはcountElements()、同じ間違いをする可能性は低くなります。C++ 標準ライブラリの作成者は、おそらくメソッドに名前を付けたかったのでしょう。サイズ () 他のすべてのコンテナーと一致させるにはベクターそして地図。しかし、実際に実行したため、他のコンテナーと同様に、プログラマはそれが高速な操作であると誤解しやすくなります。ありがたいことに、最新の C++ 標準では、サイズ () することが $\mathcal{O}(1)$ 。

魔法使いは誰ですか?

少し前に、作成者の1人がOpenBSDオペレーティングシステムをインストールしていました。ディスクのフォーマット手順中に、ディスクのパラメータを要求する複雑なメニューが表示されました。選択肢の1つは、「」に行くことでした。ウィザードモード。"彼は、この使いやすいオプションを見つけて安心し、選択しました。残念なことに、インストーラは手動ディスクフォーマットコマンドを待つ低レベルプロンプトに落とされ、そこから抜け出す明確な方法はありませんでした。明らかに「魔法使い」という意味だったあなた魔法使いだった!

例: 複数の名前候補の評価

良い名前を決めるとき、複数の候補が検討されるかもしれません。最終的な選択を決める前に、それぞれの名前の利点について頭の中で議論するのが一般的です。次の例は、この批評プロセスを示しています。

トラフィックの多いWebサイトでは、Webサイトへの変更によってビジネスが改善されるかどうかをテストするために「実験」がよく行われます。以下は、いくつかの実験を制御する構成ファイルの例です。

実験ID:100

説明:」フォントサイズを14ptに増やす

トランザクションの割合:5%

。。。

各実験は、約15個の属性と値のペアによって定義されます。残念ながら、非常によく似た別の実験を定義する場合は、これらの行のほとんどをコピーして貼り付ける必要があります。

実験ID:101

説明:」フォントサイズを大きくして13pt」

[他の行はexperiment_id 100と同一]

ある実験で別の実験のプロパティを再利用する方法を導入することで、この状況を解決したいとします。(これは「プロトタイプ継承」パターンです。)最終的には、次のように入力することになります。

実験ID: 101

the_other_experiment_id_I_want_to_reuse: 100

[必要に応じてプロパティを変更します]

問題は、何をすべきですか。the_other_experiment_id_I_want_to_reuse本当に指名されるの?

考慮すべき4つの名前は次のとおりです。

- 1.テンプレート
- 2.再利用
- 3.コピー
- 4.継承する

これらの名前はどれも意味があり、私たちなぜなら、この新しい機能を設定言語に追加しているのは私たちだからです。ただし、コードに遭遇し、この機能について知らない人にその名前がどのように聞こえるかを想像する必要があります。それでは、それぞれの名前を分析して、誰かがどのように誤解するかを考えてみましょう。

1. 名前を使って想像してみましょうテンプレート：

実験ID: 101
テンプレート：100
。。。

テンプレートにはいくつかの問題があります。まず、「私はテンプレートです」と言っているのか、「私はこの他のテンプレートを使用しています」と言っているのかが明確ではありません。第二に、「テンプレート」とは多くの場合、抽象的なそれはその前に「埋める」必要がありますコンクリート。テンプレート化された実験は「実際の」実験ではないと考える人もいるかもしれません。全体、テンプレートこの状況では曖昧すぎます。

2. どうでしょうか再利用：

実験ID: 101
再利用：100
。。。

再利用「」という言葉は問題ありませんが、このように書かれていると、「この実験は最大100回再利用できます」と言っているように思われるかもしれません。名前を変更すると、再利用ID役立つだろう。しかし、混乱した読者はこう思うかもしれません。再利用ID: 100「私の再利用用IDは100です」という意味です。

3. 考えてみましょうコピー。

実験ID: 101
コピー：100
。。。

コピー良い言葉です。しかし、それ自体では、コピー: 100「この実験を100回コピーしてください」または「これは何かの100番目のコピーです」と言っているように見えます。この用語が指すことを明確にするために、別の実験として、名前を次のように変更できます。コピー実験。これはおそらくこれまでで最高の名前です。

4. しかし、ここで考えてみましょう継承:

実験ID: 101
継承:100
。。。

「継承」という言葉はほとんどのプログラマーに馴染みがあり、継承後にさらに変更が行われることは理解されています。クラス継承を使用すると、別のクラスのすべてのメソッドとメンバーを取得して、それらを変更したり追加したりできます。実生活においても、親戚から所有物を相続した場合、それを売却したり、自分で他のものを所有したりする可能性があることは理解されています。

ただし、もう一度、別の実験から継承していることを明確にしましょう。改善できる
の名前継承元あるいは継承元実験ID。

全体、コピー実験そして継承からの実験ID何が起こっているかを最も明確に説明し、誤解される可能性が最も低いため、これらの名前が最適です。

まとめ

最良の名前は、誤解のない名前です。コードを読む人は、意図したとおりにコードを理解します。それ以外の方法はありません。残念なことに、プログラミングに関しては、多くの英単語があいまいです。フィルター、長さ、そして限界。

名前を決める前に、悪魔の代弁者になって、自分の名前がどのように誤解されるかを想像してください。最良の名前は誤解されにくいものです。

値の上限または下限を定義する場合、最大_そしてmin_を使用するのに適したプレフィックスです。包含範囲の場合、初めそして最後良いです。包含範囲/排他範囲については、始めるそして終わり最も慣用的なので最高です。

ブール値に名前を付けるときは、次のような単語を使用します。はそしてもっているそれがブール値であることを明確にするためです。否定語は避けてください (例:disable_ssl)。

特定の単語に対するユーザーの期待に注意してください。たとえば、ユーザーは次のことを期待するかもしれません。得る () または サイズ () 軽量メソッドであること。

第四章

美学



雑誌のレイアウトには、段落の長さ、段組の幅、記事の順序、表紙に何を掲載するかなど、多くのことが考慮されます。優れた雑誌は、ページからページへスキップするのが簡単ですが、すぐに読むのも簡単です。

優れたソースコードは、同様に「目に優しい」ものである必要があります。この章では、スペース、配置、順序をうまく利用することでコードがどのように読みやすくなるかを説明します。

具体的には、次の3つの原則を使用します。

- 読者が慣れることができるパターンを使用して、一貫したレイアウトを使用します。
- 類似したコードを類似したものにします。
- 関連するコード行をブロックにグループ化します。

美学 VS. デザイン

この章では、コードに加えることができる単純な「美的」改善のみを取り上げます。このような種類の変更は簡単に行うことができ、多くの場合、可読性が大幅に向上します。コードの大規模なリファクタリング(新しい関数やクラスの分割など)がさらに役立つ場合があります。私たちの見解は、優れた美学と優れたデザインは独立したアイデアです。理想的には両方を目指すべきです。

なぜ美学が重要なのでしょうか?



このクラスを使用する必要がある場合を想像してください。

```
クラス StatsKeeper{
    公共：
    // 一連の double を追跡するためのクラス
    void Add(double d); // およびそれらに関する簡単な統計を取得するメソッド
    プライベート： int カウント; /* ここまで何件か
    */ 公共：
    double Average();

    プライベート： 二重最小値。
    リスト<double>
    過去の項目
    ;最大値の 2 倍;
};
```

代わりにこのクリーンなバージョンを使用する場合よりも、それを理解するのにはるかに時間がかかるでしょう。

```
// 一連の double を追跡するためのクラス // およびそれらに
    関する迅速な統計を取得するメソッド。クラス StatsKeeper
{
    公共：
    void Add(double d);
    double Average();

    プライベート：
    list<double> past_items; int カウン
    ト; // ここまで何個

    二重最小値。
    最大2倍。
};
```

明らかにそれは見た目が美しいコードでの作業が容易になる。考えてみると、プログラミングに費やす時間のほとんどはコードを調べるために費やされています。コードを素早くざっと確認できるほど、誰でも簡単にコードを使用できるようになります。

一貫性とコンパクトになるように改行を再配置する

さまざまなネットワーク接続速度の下でプログラムがどのように動作するかを評価する Java コードを作成しているとします。あなたが持っている TcpConnection シミュレータ これはコンストラクターで 4 つのパラメーターを受け取ります。

1. 接続速度 (Kbps)
2. 平均レイテンシー (ミリ秒)
3. レイテンシーの「ジッター」 (ミリ秒)
4. パケットロス (割合)

コードには3つの異なるものが必要でしたTcpConnectionシミュレータインスタンス:

```
パブリック クラス PerformanceTester {
    パブリック 静的最終 TcpConnectionSimulator wifi = 新しいTcpConnectionシミュレータ(
        500, /* Kbps */
        80, /* ミリ秒の遅延 */
        200, /* ジッター */
        1 /* パケット損失 % */);

    パブリック 静的最終 TcpConnectionSimulator t3_fiber =
        新しいTcpConnectionシミュレータ(
            45000, /* Kbps */
            10, /* ミリ秒の遅延 */ 0, /* ジッ
                ター */
            0 /* パケット損失 % */);

    パブリック 静的最終 TcpConnectionSimulator セル = 新しいTcpConnectionシミュレータ(
        100, /* Kbps */
        400, /* ミリ秒の遅延 */
        250, /* ジッター */
        5 /* パケット損失 % */);
}
```

このサンプルコードでは、80文字の制限内に収めるために多くの追加の改行が必要でした(これがあなたの会社のコーディング標準でした)。残念ながら、それによって次のような定義がなされました。t3_ファイバ
隣の人とは違うように見えます。このコードの「シルエット」は奇妙であり、注目を集めます。t3_
ファイバ理由もなく。これは、「似たコードは似ているはずだ」という原則に従っていません。

コードの一貫性を高めるために、追加の改行を導入できます(途中でコメントを並べます)。

```
パブリック クラス PerformanceTester {
    パブリック 静的最終 TcpConnectionSimulator wifi =
        新しいTcpConnectionシミュレータ(
            500, /* Kbps */
            80, /* ミリ秒の遅延 */
            200, /* ジッター */ 1
            /* パケットロス % */);

    パブリック 静的最終 TcpConnectionSimulator t3_fiber =
        新しいTcpConnectionシミュレータ(
            45000, /* Kbps */ 10, /* ミリ秒の遅
                延 */ 0,
            /* ジッター */
            0 /* パケットロス % */);

    パブリック 静的最終 TcpConnectionSimulator セル =
        新しいTcpConnectionシミュレータ(
            100, /* Kbps */
            400, /* ミリ秒の遅延 */
            250, /* ジッター */ 5
            /* パケットロス % */);
}
```

このコードには一貫したパターンがあり、読み進めるのが簡単です。しかし、残念ながら、垂直方向のスペースを多く使用します。また、各コメントが3回複製されます。

クラスをよりコンパクトに記述する方法は次のとおりです。

```
パブリック クラス PerformanceTester {
    // TcpConnectionSimulator(スループット、レイテンシー、ジッター、パケットロス) //
    // [Kbps] [ms] [ms] [パーセント]

    パブリック 静的最終 TcpConnectionSimulator wifi =
        新しいTcpConnectionシミュレータ(500、80、200, 1);

    パブリック 静的最終 TcpConnectionSimulator t3_fiber =
        新しいTcpConnectionシミュレータ(45000、10、0, 0);

    パブリック 静的最終 TcpConnectionSimulator セル =
        新しいTcpConnectionシミュレータ(100、400、250, 5);
}
```

コメントを一番上に移動し、すべてのパラメータを1行に配置しました。コメントは各数値のすぐ隣にありませんが、「データ」はよりコンパクトな表に並べられています。

メソッドを使用して不規則性を解消する

次の機能を提供する人事データベースがあるとします。

```
// 「Doug Adams」のようなpartial_nameを「Mr. Douglas Adams」に変換します。// 不可能な場合は、「エラー」に説明が入ります。
弦フルネームを展開(DatabaseConnection dc、文字列部分名、文字列* エラー);
```

この関数は一連の例でテストされました。

```
DatabaseConnection データベース接続; 文字
列エラー;
主張する (フルネームを展開(database_connection, "Doug Adams", &error)
    == 「ダグラス・アダムス氏」);
アサート(エラー == "");
主張する (フルネームを展開(データベース接続、「ジェイク・ブラウン」、&エラー)
    == "ジェイコブ・ブラウン三世氏");
アサート(エラー == "");
主張する (フルネームを展開(database_connection, "そんな奴はいない", &error) == ""); assert(error == "-
致するものが見つかりませんでした");
主張する (フルネームを展開(database_connection, "John", &error) == "");
assert(error == "複数の結果");
```

このコードは見た目が美しくありません。一部の行は長すぎるために、次の行に折り返されます。このコードのシルエットは醜く、一貫したパターンがありません。

ただし、これは改行を再配置してもできることは限られている場合です。もっと大きな問題はそれは、「」のような繰り返し文字列がたくさんあることです。assert(ExpandFullName(データベース接続...,"そして"エラー"邪魔になっているもの。このコードを実際に改善するには、コードを次のようにするためのヘルパー メソッドが必要です。

フルネームを確認する(「ダグ・アダムス」、「ダグラス・アダムス氏」、「」); フルネームを確認する(「ジェイク・ブラウン」、「ミスター・ジェイク・ブラウン三世」、「」); フルネームを確認する(「そのような男はいません」、「」、「一致するものが見つかりませんでした」); フルネームを確認する("ジョン", "", "複数の結果");

ここで、それぞれ異なるパラメーターを使用した4つのテストが実行されていることがより明確になります。たとえすべての「汚れ仕事」が内側にあるとしてもCheckFullName()、この機能もそれほど悪くありません。

```
void CheckFullName(string 部分名,
    文字列 Expected_full_name,
    string Expected_error) {
    //database_connection はクラス メンバー文字列エラーに
    なりました。
    string full_name = ExpandFullName(データベース接続,部分的な名前、 & エラー) ; アサー
    ト(エラー==予想されるエラー); アサート(フルネーム ==期待されるフルネーム);
}
```

私たちの目標はコードをより美しくすることだけでしたが、この変更には他にも多くの副次的な利点があります。

- 以前の重複コードの多くが削除され、コードがよりコンパクトになります。
- 各テストケースの重要な部分(名前とエラー文字列)が単独で一目瞭然になりました。以前は、これらの文字列には次のようなトークンが散在していました。
データベース接続そしてエラー、そのため、コードを一目で「理解する」のは困難でした。
- 新しいテストの追加がはるかに簡単になりました。

この話の教訓は、コードを「見た目に美しく」すると、多くの場合、表面的な改善以上の効果が得られ、コードの構造を改善できる可能性があるということです。

役立つ場合は列の配置を使用する

直線的なエッジとコラムにより、読者はテキストを読みやすくなります。

コードを読みやすくするために「列の配置」を導入できる場合があります。たとえば、前のセクションでは、引数を間隔をあけて並べることができます。

CheckFullName():

```
CheckFullName("ダグ・アダムス"    , 「ダグラス・アダムスさん」, 「」) ;
CheckFullName(" ジェイク ブラウン", "ジェイク ブラウン 3世氏", "");
CheckFullName(「そんな奴はない」, 「」           , 「一致するものが見つかりませんでした」);
CheckFullName("ジョン"           , 「」           , 「複数の結果」);
```

このコードでは、2番目と3番目の引数を区別するのが簡単です。CheckFullName()。

以下に、変数定義の大規模なグループを含む簡単な例を示します。

```
# POST パラメータをローカル変数に抽出します詳
細 = request.POST.get('details') location =
request.POST.get('location') Phone =
request.POST.get('phone')
```

```

Eメール = request.POST.get('メール')=
URL      request.POST.get('url')

```

お気づきかと思いますが、3番目の定義にはタイプミスがあります（クエストの代わりにリクエスト）。すべてがきれいに並んでいると、このような間違いがより顕著になります。

の中にウィゲットコードベースでは、使用可能なコマンド ライン オプション（100 個以上）が次のようにリストされています。

```

コマンド[] ={
    . . .
    {"タイムアウト",           ヌル,           cmd_spec_timeout},
    {"タイムスタンプ",         &opt.タイムスタンプ, cmd_boolean},
    {"「試してみます」,         &opt.ntry,        cmd_number_inf},
    {"プロキシを使う",          &opt.use_proxy,   cmd_boolean},
    {"ユーザーエージェント",    ヌル,           cmd_spec_useragent},
    . . .
};

```

このアプローチにより、リストをざっと読んだり、ある列から次の列に移動したりすることが非常に簡単になりました。

列の配置を使用する必要がありますか？

列の端には「視覚的な手すり」があり、全体を読みやすくなります。これは「似たコードを似たものにする」という良い例です。

しかし、プログラマーの中にはそれを好みない人もいます。理由の1つは、アライメントの設定と維持に多くの労力がかかることです。もう1つの理由は、変更を行うときに大きな「差分」が作成されることです。1行の変更により、他の5行が変更される可能性があります（ほとんどは空白のみ）。

私たちのアドバイスは、それを試してみることです。私たちの経験では、プログラマーが懸念しているほど多くの作業は必要ありません。そして、そうなった場合は、単に停止することができます。

意味のある順序を選択し、一貫して使用する

コードの順序が正確さに影響しない場合も多くあります。たとえば、次の5つの変数定義は任意の順序で記述できます。

```

詳細 = request.POST.get('詳細') 場所 =
request.POST.get('location') 電話
                    = request.POST.get('電話') =
Eメール request.POST.get('メール')=
URL      request.POST.get('url')

```

このような状況では、単にランダムに並べるのではなく、意味のある順序で並べると便利です。以下にいくつかのアイデアを示します。

- 変数の順序を < の順序と一致させます。入力>対応する HTML フォームのフィールド。

- ・「最も重要なものの」から「最も重要でないもの」の順に並べます。
- ・アルファベット順に並べます。

どのような順序であっても、コード全体で同じ順序を使用する必要があります。後で順序を変更すると混乱を招く可能性があります。

詳細の場合:	録画詳細	= 詳細	
電話の場合:	録音電話	= 電話	# ねえ、「場所」はどこに行ったの？
電子メールの場合:	受信メール	= 電子メール	
URLの場合:	録画URL	= URL	
場所の場合:rec.location = 場所			# なぜ「場所」が今ここにあるのですか？

宣言をブロックに整理する

脳は自然にグループと階層の観点から考えるため、そのようにコードを整理することで、読者がコードをすばやく理解できるようになります。

たとえば、フロントエンド サーバーの C++ クラスとそのすべてのメソッド宣言を次に示します。

```
クラス フロントエンドサーバー{
    公共:
        フロントエンドサーバー();
        void ViewProfile(HttpRequest* リクエスト);
        void OpenDatabase(string location, string user); void
        SaveProfile(HttpRequest* リクエスト);
        string ExtractQueryParam(HttpRequest* リクエスト, string param);
        void ReplyOK(HttpRequest* request, string html);
        void FindFriends(HttpRequest* リクエスト);
        void ReplyNotFound(HttpRequest* リクエスト、文字列エラー);
        void CloseDatabase(文字列の場所);
        ~フロントエンドサーバー();
};
```

このコードはひどいものではありませんが、レイアウトは読者がこれらすべてのメソッドを理解するのに役立つものではありません。すべてのメソッドを1つの巨大なブロックにリストするのではなく、次のように論理的にグループに編成する必要があります。

```
クラス フロントエンドサーバー{
    公共:
        フロントエンドサーバー();
        ~フロントエンドサーバー();

        // ハンドラー
        void ViewProfile(HttpRequest* リクエスト);
        void SaveProfile(HttpRequest* リクエスト);
        void FindFriends(HttpRequest* リクエスト);

        // リクエスト/返信ユーティリティ
        string ExtractQueryParam(HttpRequest* リクエスト, string param);
        void ReplyOK(HttpRequest* request, string html);
        void ReplyNotFound(HttpRequest* リクエスト、文字列エラー);
```

```
// データベースヘルパー
void OpenDatabase(string location, string user); void
CloseDatabase(文字列の場所);
};
```

このバージョンは非常に理解しやすいです。また、コード行数が増えて読みやすくなっています。その理由は、4つの高レベルのセクションをすぐに理解し、必要なときに各セクションの詳細を読むことができるからです。

コードを「段落」に分割する

書かれたテキストはさまざまな理由で段落に分割されます。

- ・類似したアイデアをグループ化し、他のアイデアと区別する方法です。
- ・視覚的な「踏み台」を提供します。これがないと、ページ上で自分の位置を失いやすくなります。
- ・ある段落から別の段落へのナビゲーションが容易になります。

同じ理由で、コードは「段落」に分割する必要があります。たとえば、次のような巨大なコードの塊を読みたがる人はいません。

```
# ユーザーの電子メール連絡先をインポートし、システム内のユーザーと照合します。
# 次に、まだ友達になっていないユーザーのリストを表示します。def
assign_new_friends(ユーザー、電子メールのパスワード):
    友達 = user.friends()
    friends_emails = set(友達の f には f.email) contacts =
    import_contacts(user.email, email_password) contact_emails = set(連絡先の c
    には c.email) non_friend_emails = contact_emails - friends_emails
    requested_friends = User.objects.select(email__in=non_friend_emails) 表示
    ['ユーザー'] = ユーザー

    display['friends'] = 友達 display['suggested_friends'] =
    示唆された_友達 return
    render("suggested_friends.html", display)
```

明らかではないかもしれません、この関数はいくつかの異なるステップを経ます。したがって、これらのコード行を段落に分割すると特に便利です。

```
def assign_new_friends(ユーザー、電子メールのパスワード):
    # ユーザーの友人の電子メール アドレスを取得し
    # ます。 友達 = user.friends()
    friends_emails = set(友達内の f の f.email)

    # このユーザーの電子メール アカウントからすべての電子メール アドレスを
    # インポートします。 contacts = import_contacts(user.email,
    email_password) contact_emails = set(連絡先の c の c.email)

    # まだ友達になっていない一致するユーザーを検索します。 non_friend_emails =
    contact_emails - friends_emails requested_friends =
    User.objects.select(email__in=non_friend_emails)
```

```
# これらのリストをページに表示します。
表示['ユーザー'] = ユーザー
表示['フレンズ'] = フレンズ表示['サジェスト_フレンズ'] =
サジェスト_フレンズ

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

各段落に概要コメントも追加していることに注意してください。これは、読者がコードをざっと読むのにも役立ちます。（見る[第5章 何をコメントすべきかを知る。](#)）

書かれたテキストと同様に、コードを分割する方法は複数あり、プログラマはより長い段落またはより短い段落を好む場合があります。

個人のスタイルと一貫性

突き詰めれば個人のスタイルに応じた美的な選択がいくつかあります。たとえば、クラス定義の開き中括弧は次のように配置されます。

```
クラスロガー{
    . . .
};
```

または

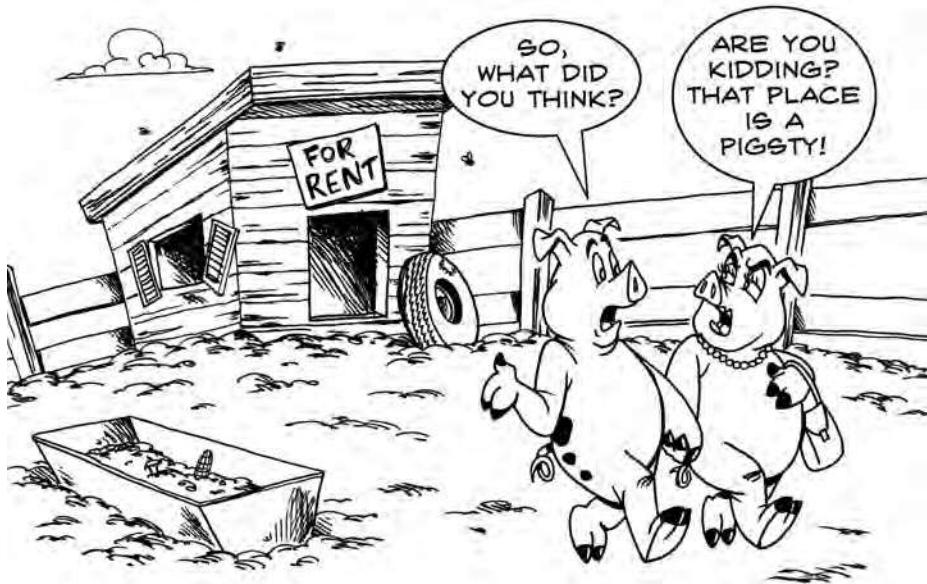
```
クラスロガー
{
    . . .
};
```

これらのスタイルのいずれかを選択しても、コードベースの読みやすさには実質的な影響はありません。ただし、これら2つのスタイルがコード全体で混在している場合は、読みやすさに影響します。

私たちは、チームが「間違った」スタイルを使用していると感じた多くのプロジェクトに取り組んできましたが、一貫性がはるかに重要であることを知っていたため、プロジェクトの慣例に従いました。

キー・アイデア

一貫したスタイルは、「正しい」スタイルよりも重要です。



まとめ

誰もが見た目が美しいコードを読みたいと考えています。一貫性のある意味のある方法でコードを「フォーマット」すると、読みやすくなり、速くなります。

ここで説明した具体的なテクニックは次のとおりです。

- ・複数のコード ブロックが同様のことを実行している場合は、それらのブロックに同じシルエットを与えるようにしてください。
- ・コードの一部を「列」に配置すると、コードをざっと読みやすくなります。
- ・コード内で1つの場所でA、B、およびCが言及されている場合、別の場所ではB、C、およびAと言及しないでください。意味のある順序を選択し、それを守ります。
- ・空行を使用して、大きなブロックを論理的な「段落」に分割します。

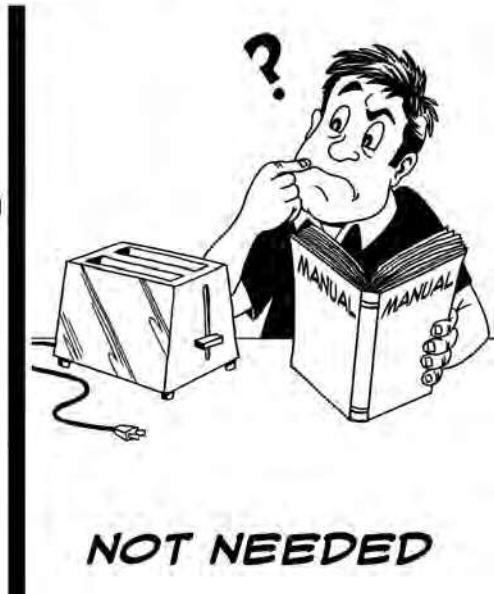
第五章

何をコメントすべきかを知る

INSTRUCTION MANUALS



NEEDED!!



NOT NEEDED

この章の目的は、何をコメントすべきかを理解できるようにすることです。コメントの目的は「コードの動作を説明する」ことだと思うかもしれません、それはほんの一部にすぎません。

キー・アイデア

コメントの目的は、書き手と同じように読者にも多くのことを知ってもらうことです。

コードを書いているとき、頭の中には貴重な情報がたくさんあります。他の人があなたのコードを読むと、その情報は失われ、目の前にあるコードだけが残ります。

この章では、頭の中にある情報をいつ書き留めるかについて、多くの例を示します。コメントに関する日常的な点は省略しました。代わりに、コメントのより興味深い、そして「十分にサービスが受けられていない」側面に焦点を当ててきました。

この章は次の領域に分かれています。

- 何を知るかしないコメントします
- コーディングしながら考えを記録する
- 読者の立場に立って、読者が知る必要があることを想像する

コメントしてはいけないこと



コメントを読むと実際のコードを読むよりも時間がかかり、各コメントが画面上のスペースを占有します。つまり、それだけの価値があるということです。では、価値のないコメントと良いコメントの境界線はどこにあるのでしょうか？

このコード内のコメントはすべて無価値です。

```
// Account のクラス定義 クラスアカウン
ト{
    公共：
        // コンストラクター
        アカウント () ;

    // 利益メンバーを新しい値に設定します void
    SetProfit(二重利益);

    // このアカウントからの利益を返します ダブ
    ル GetProfit();
}
```

これらのコメントは、新しい情報を提供したり、読者がコードをよりよく理解するのに役立つわけではないため、価値がありません。

キー・アイデア

コード自体からすぐに導き出せる事実についてはコメントしないでください。

ただし、「迅速に」という言葉は重要な違いです。このPythonコードのコメントを考えてみましょう。

```
# 2番目の '*' 以降をすべて削除します
名前 =
'*'.join(line.split('*')[2])
```

厳密に言えば、このコメントは「新しい情報」を提示するものではありません。コード自体を見れば、最終的にはそれが何をしているのかがわかります。しかし、ほとんどのプログラマにとって、コメント付きコードを読むほうが、コメントなしでコードを理解するよりもはるかに速くなります。

コメントのためだけにコメントしないでください



教授の中には、学生に宿題コードの各関数にコメントを付けるよう要求する人もいます。その結果、一部のプログラマは関数をコメントなしで裸のままにしておくことに罪悪感を感じ、関数の名前と引数を文形式で書き直すことになります。

```
// 指定されたサブツリー内で、指定された名前、指定された深さを使用してノードを検索します。
Node* FindNodeInSubtree(Node* サブツリー, 文字列名, int 深さ);
```

これは「価値のないコメント」のカテゴリに分類されます。関数の宣言とコメントは実質的に同じです。このコメントは削除するか改善する必要があります。

ここにコメントしたい場合は、より重要な詳細について詳しく説明することもできます。

```
// 指定された「名前」を持つノードを検索するか、NULL を返します。  
// 深さ <= 0 の場合、「サブツリー」のみが検査されます。  
// 深さ == N の場合、「サブツリー」とその下の N レベルのみが検査されます。  
Node* FindNodeInSubtree(Node* サブツリー, 文字列名, int 深さ);
```

不適切な名前にはコメントしないでください - 代わりに名前を修正してください

コメントで悪名を補う必要はありません。たとえば、これは、という名前の関数に対する無害に見えるコメントです。CleanReply():

```
// リクエストに記載されているように、返信に制限を適用します。  
// 返されるアイテムの数や合計バイト サイズなど。 void CleanReply(リクエスト、リプライ返信);
```

コメントのほとんどは、「クリーン」の意味を単に説明しているだけです。代わりに、「制限を強制する」というフレーズを関数名に移動する必要があります。

```
// 'reply' がカウント/バイトなどを満たしていることを確認してください。「リクエスト」  
無効による制限EnforceLimitsFromRequest(要求要求、応答応答);
```

この関数名は、より「自己文書化された」ものです。適切な名前は、関数が使用されるあらゆる場所で表示されるため、適切なコメントよりも優れています。

以下は、不適切な名前の関数に対するコメントの別の例です。

```
// このキーのハンドルを解放します。これは実際のレジストリを変更しません。 void  
DeleteRegistry(RegistryKey* キー);
```

名前レジストリの削除()危険な機能のように聞こえます（削除しますレジストリ?!）。「これは実際のレジストリを変更するものではありません」というコメントは、混乱を解消しようとしています。

代わりに、次のようなより自己文書化された名前を使用できます。

```
空所レジストリハンドルのリリース(RegistryKey* キー);
```

一般に、コードの読みにくさを補おうとする「要点コメント」は望ましくありません。プログラマーはよくこのルールを次のように述べます。良いコード > 悪いコード + 良いコメント。

自分の考えを記録する

さて、何を知ったかというと、ないコメントするには、何を話し合おうかすべきコメントされることもあります(ただし、コメントされないこともあります)。

単純に「考えを記録する」、つまりコードを書いているときに考えた重要な考えを記録するだけで、多くの良いコメントが得られます。

「監督コメント」を収録

映画には、映画製作者が洞察を示し、映画がどのように作られたかを理解するのに役立つストーリーを語る「監督解説」トラックがよくあります。同様に、コードに関する貴重な洞察を記録するには、コメントを含める必要があります。

以下に例を示します。

```
// 驚くべきことに、このデータの場合、バイナリツリーはハッシュテーブルよりも 40% 高速でした。// ハッシュを計算するコストは、左右の比較よりも高かった。
```

このコメントは読者に何かを教え、オプティマイザーを目指す人が時間を無駄にするのを防ぎます。

別の例を次に示します。

```
// このヒューリスティックはいくつかの単語を見逃している可能性があります。それで大丈夫です；これを100%解決するのは難しいです。
```

このコメントがないと、読者はバグがあると考え、失敗するテストケースを考え出すのに時間を無駄にするか、バグを修正しようとしてしまう可能性があります。

コメントは、コードの状態が良くない理由を説明することもできます。

```
// このクラスは乱雑になってきています。// 物事を整理しやすくするために、'ResourceNode' サブクラスを作成する必要があるかもしれません。
```

このコメントは、コードが乱雑であることを認めていますが、次の人がそれを修正することを奨励しています(具体的な開始方法も含めて)。コメントがなければ、多くの読者はその乱雑なコードに怖気づいて、コードに触れることを恐れるでしょう。

コードの欠陥をコメントする

コードは常に進化しており、その過程で必ず欠陥が発生します。それらの欠陥を文書化することを恥ずかしがらないでください。たとえば、いつ改善を行う必要があるかを指摘します。

```
// TODO: より高速なアルゴリズムを使用する
```

またはコードが不完全な場合:

```
// TODO(dustin): JPEG 以外の画像形式を処理します
```

プログラマーの間で人気のあるマーカーが多数あります。

マーカー	代表的な意味
TODO:	まだ手に入れていないこと
修正:	既知の壊れたコードはこちら
ハック:	明らかに洗練されていない問題の解決策
XXX:	危険！ここで大きな問題が発生

チームには、これらのマーカーをいつ使用するか、使用するかについて特定の規則がある場合があります。例えば、TODO:重大な問題のために予約されている可能性があります。もしそうなら、より小さな欠陥に対しては次のようなものを使用できますやること:(小文字) またはおそらく後で:その代わり。

重要なことは、コードが将来どのように変更されるべきかについての考えについて、いつでも遠慮なくコメントする必要があるということです。このようなコメントは、コードの品質と状態についての貴重な洞察を読者に与え、コードを改善する方法についての指示を与えることもあります。

定数についてコメントする

定数を定義する場合、多くの場合、その定数が何をするのか、またはなぜその特定の値を持つのかについての「ストーリー」が存在します。たとえば、コード内に次の定数が含まれる場合があります。

```
NUM_THREADS = 8
```

この行にはコメントが必要ないように見えるかもしれません、この行を選択したプログラマはそれについて詳しく知っている可能性があります。

```
NUM_THREADS = 8 #>= 2 * num_processors であれば、それで十分です。
```

コードを読む人は、その値を調整する方法についてのガイダンスを得ることができます(たとえば、1に設定すると低すぎる可能性があり、50に設定するとやりすぎです)。

あるいは、定数の正確な値がまったく重要ではない場合もあります。この趣旨のコメントは依然として役に立ちます。

```
// 合理的な制限を課します - いずれにせよ、人間はそれほど多くの量を読むことはできません。  
#n。 const int MAX_RSS_SUBSCRIPTIONS = 1000;
```

場合によっては、高度に調整された値であるため、あまり調整すべきではない場合があります。

```
画像品質 = 0.72; // ユーザーは 0.72 がサイズと品質のトレードオフとして最適であると考えました
```

これらすべての例で、コメントを追加することは考えられなかったかもしれません、コメントは非常に役立ちます。

名前が十分に明確であるため、コメントを必要としない定数がいくつかあります(例:SECONDS_PER_DAY)。しかし、私たちの経験では、ほとんどの定数はコメントを追加することで改善できます。その定数の値を決めるときに何を考えたかを書き留めるだけです。

読者の立場になって考えてみよう

本書で使用する一般的なテクニックは次のとおりです。**あなたのコードが部外者にどのように見えるかを想像してください**—あなたほどあなたのプロジェクトに詳しくない人。このテクニックは、コメントが必要なものを認識するのに特に役立ちます。

よくある質問を予想する



**“ARE THERE ANY QUESTIONS?...
...THAT AREN’T COVERED BY THAT SIGN.”**

他の人があなたのコードを読むと、次のように思われる部分があります。はあ？これは一体どういうことなのでしょうか？あなたの仕事は、それらの部分にコメントを付けることです。

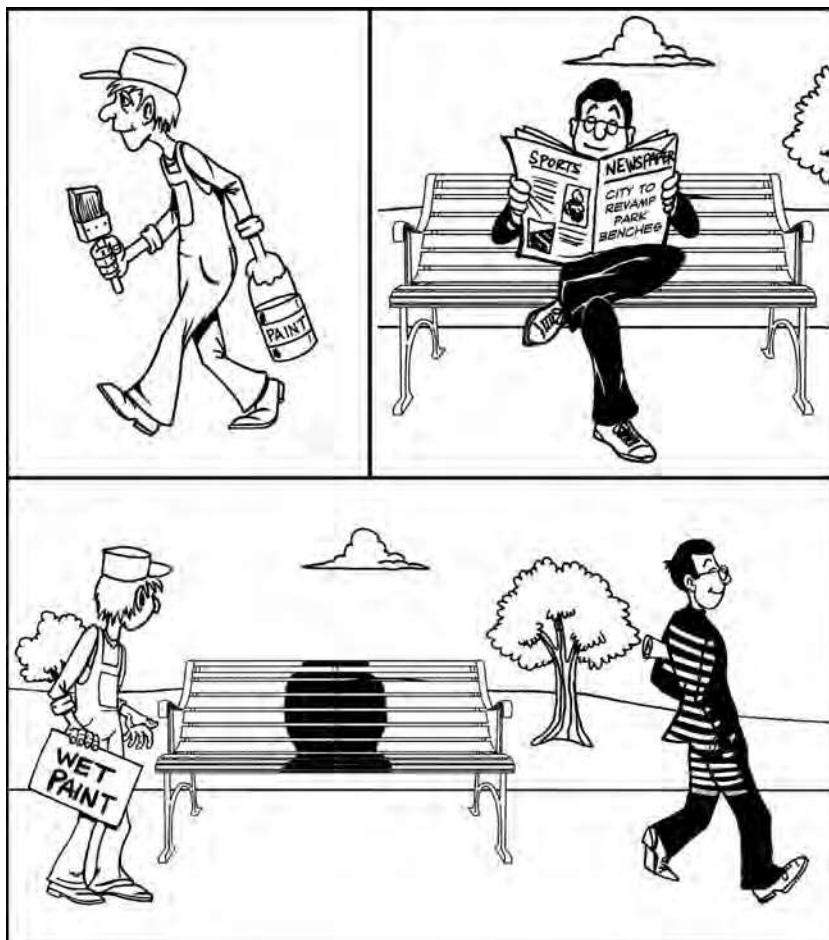
たとえば、次の定義を見てみましょう。クリア () :

```
構造体レコーダー {
    ベクトル<float>データ;
    ...
    void Clear() {
        ベクトル<float>().swap(data); // はあ？なぜ data.clear() だけではだめなのでしょうか?
    }
};
```

ほとんどの C++ プログラムはこのコードを見て次のように考えます。なぜ彼はそうしなかったのか data.clear() 空のベクトルと交換する代わりに?さて、これが強制的に実行する唯一の方法であることがわかりました。ベクターメモリを本当にメモリアロケータに放棄します。これはあまり知られていない C++ の詳細です。肝心なのは、コメントする必要があるということです。

```
// ベクトルにメモリを強制的に放棄させます (「STL スワップ トリック」を調べてください)
ベクトル<float>().swap(data);
```

広告に起こりやすい落とし穴



関数またはクラスを文書化するときは、次のことを自問するとよいでしょう。このコードの何が驚くべきでしょうか? どのように悪用される可能性がありますか? 基本的に、コードを使用するとき人々が遭遇する可能性のある問題を「事前に考えて」予測する必要があります。

たとえば、特定のユーザーに電子メールを送信する関数を作成したとします。

```
void SendEmail(宛先文字列, 件名文字列, 本文文字列);
```

この機能の実装には外部電子メール サービスへの接続が含まれます。これには最大で 1 秒、場合によってはそれ以上かかる場合があります。Web アプリケーションを作成している人は、これに気づかず、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可能性があります。(これを行うと、電子メール サービスがダウンしている場合に Web アプリケーションが「ハング」します。)

この起こりそうな事故を防ぐには、この「実装の詳細」についてコメントする必要があります。

```
// 外部サービスを呼び出して電子メールを配信します。(1分後にタイムアウトします。)
void SendEmail(宛先文字列, 件名文字列, 本文文字列);
```

ここに別の例があります:FixBrokenHtml()不足している終了タグなどを挿入することで壊れたHTMLを書き換えようとする関数:

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

この関数は、深くネストされた一致しないタグがある場合に実行時間が大幅に長くなるという注意点を除いて、うまく機能します。厄介なHTML入力の場合、この関数は次のようにになります。分実行する。

ユーザーが後で自分でこれを発見できるようにするのではなく、事前に通知する方が良いでしょう。

```
// 実行時間は O(number_tags * Average_tag_Depth) なので、不適切にネストされた入力に注意してください。
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理解するのが最も難しいことの1つは、「全体像」、つまりクラスがどのように相互作用するか、データがシステム全体をどのように流れるか、そしてエントリポイントがどこにあるかということです。システムを設計した人は、システムに深く関わっているため、このことについてコメントすることを忘れることがよくあります。

次の思考実験を考えてみましょう。新しい人があなたのチームに加わったばかりで、彼女はあなたの隣に座っています。あなたは彼女にコードベースに慣れてもらう必要があります。

彼女にコードベースを案内しながら、特定のファイルやクラスを指摘して、次のようなことを言うかもしれません。

- ・「これは、ビジネスロジックとデータベースの間の接着コードです。どのアプリケーションコードもこれを直接使用すべきではありません。」
- ・「このクラスは複雑に見えますが、実際は単なるスマートキャッシュです。システムの残りの部分については何も知りません。」

1分間のカジュアルな会話の後、新しいチームメンバーは、自分でソースを読むよりもはるかに多くのことを知ることができます。

これはまさに、高レベルのコメントとして含めるべき種類の情報です。

ファイルレベルのコメントの簡単な例を次に示します。

```
// このファイルには、ファイルシステムへの // より便利なインターフェイスを提供するヘルパー関数  
が含まれています。ファイルのアクセス許可やその他の詳細な詳細を処理します。
```

広範で正式な文書を作成しなければならないという考えに圧倒されないでください。いくつかの適切に選択された文は、まったく何もしないよりは優れています。

要約コメント

関数の奥深くであっても、「全体像」についてコメントすることをお勧めします。以下は、その下の低レベルコードをきちんと要約したコメントの例です。

```
# 顧客が自分で購入したすべてのアイテムを検索します。all_customers  
の customer_id の場合:  
    all_sales[customer_id].sales で販売中:  
        sale.recipient == customer_id の場合:  
            . . .
```

このコメントがないと、コードの各行を読むのは少し謎になります。(「次のことを繰り返しているようですすべての顧客…でも何のために？」)

内部にいくつかの大きな「チャンク」がある長い関数にこれらの要約コメントを含めると特に便利です。

```
def GenerateUserReport():  
    # このユーザーのロックを取得します  
    . . .  
  
    # データベースからユーザー情報を読み取る  
    . . .  
  
    # 情報をファイルに書き込む  
    . . .  
  
    # このユーザーのロックを解除します
```

これらのコメントは関数の動作の箇条書きの概要としても機能するため、読者は詳細に入る前に関数の動作の要点を把握できます。(これらのチャンクが簡単に分離できる場合は、それらを独自の関数にすることもできます。前に述べたように、良いコードは、良いコメントが付いた悪いコードよりも優れています。)

内容、理由、または方法についてコメントする必要がありますか?

次のようなアドバイスを聞いたことがあるかもしれません。なぜではなく、何(またはその方法)」キャッチャーではありますが、これらのステートメントは単純すぎるため、人によって意味が異なると思われます。

私たちのアドバイスは、読者がコードをより簡単に理解できるようにすることをすべて行うことです。これには、コメントすることが含まれる場合があります。何、どうやって、またはなぜ(または3つすべて)。

最終的な考え方—ライターズブロックを乗り越える

多くのプログラマーは、良いコメントを書くのは大変な作業のように感じられるため、コメントを書きたがらません。ライターがこの種の「ライターズブロック」に陥った場合、最善の解決策は、ただ書き始めることです。したがって、次回コメントを書くのをためらっているときは、たとえ中途半端であっても、思いついたことをそのままコメントしてください。

たとえば、関数に取り組んでいて、次のように考えたとします。ああ、このリストに重複があると、この作業は難しくなります。それを書き留めてください:

// ああ、くそー、このリストに重複があると、この作業は難しくなります。

ほら、そんなに大変だったの? 実際、これはそれほど悪いコメントではありません。確かに、何もしないよりはマシです。ただし、言語は少し曖昧です。これを修正するには、各フレーズを調べて、より具体的なものに置き換えるだけです。

- ・「ああ、くだらない」というのは、実際には「注意してください。これは注意すべきことです」という意味です。
- ・「これ」とは、「この入力を処理するコード」を意味します。
- ・「扱いが難くなる」とは、「実装が困難になる」という意味です。

新しいコメントは次のようにになります。

// 注意: このコードはリスト内の重複を処理しません(処理が難しいため)

コメントを書くタスクが次の簡単な手順に分割されていることに注目してください。

1. 心に浮かんだコメントを書き留めます。
2. コメントを読んで、何を改善する必要があるかを確認します。
3. 改善を行う。

コメントを頻繁に行うと、ステップ1のコメントの品質がどんどん向上し、最終的には修正がまったく必要なくなる可能性があることがわかります。また、早めに頻繁にコメントすることで、最後に大量のコメントを書かなければならぬという不快な状況を避けることができます。

まとめ

コメントの目的は、コードを作成したときに作成者が知っていたことを読者が理解できるようにすることです。この章全体は、コードに関するそれほど明白ではない情報のすべてを認識し、それらを書き留めることについてです。

何ないコメントします：

- ・コード自体からすぐに導き出せる事実。
- ・不正なコード(不正な関数名など)を補う「要点コメント」-代わりにコードを修正します。

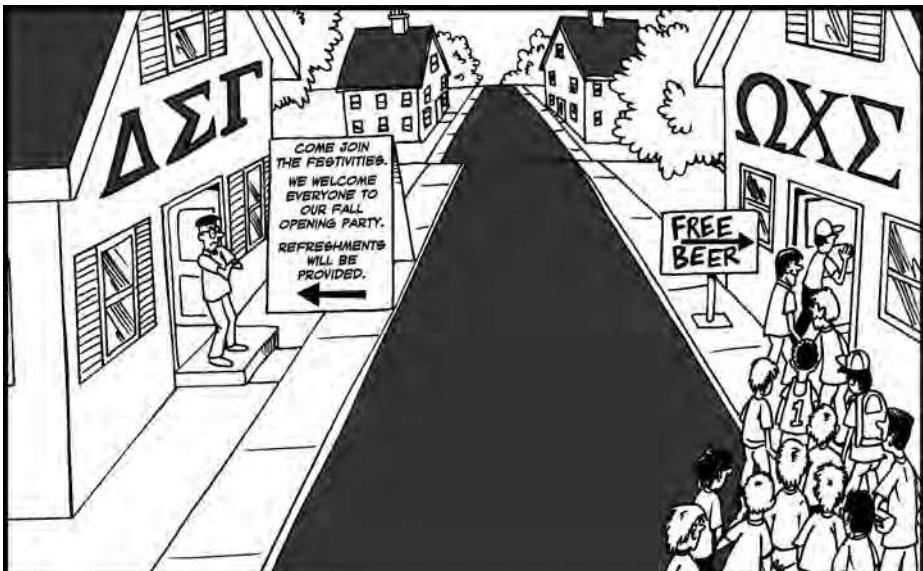
記録すべき考えは次のとおりです。

- ・コードが一方通行であり、別の方向ではない理由に関する洞察(「ディレクターの解説」)。
- ・TODO: や XXX: などのマーカーを使用したコード内の欠陥。
- ・定数がその値を取得する方法に関する「ストーリー」。

読者の立場になって考えてみましょう。

- ・コードのどの部分が読者に「え?」と思われるかを予測します。そしてそれらにコメントします。
- ・平均的な読者が予期しない驚くべき動作を文書化します。
- ・ファイル/クラス レベルで「全体像」コメントを使用して、すべての部分がどのように組み合わされるかを説明します。
- ・読者が詳細に迷わないように、コードのブロックをコメントで要約します。

コメントを正確かつコンパクトに作成する



前の章は実現についてでした何あなたはコメントするべきです。この章の内容は、どうやって正確かつコンパクトなコメントを書くこと。

コメントを書くつもりなら、そうするのもいいかもしれません正確な—できるだけ具体的かつ詳細に。一方、コメントは画面上で余分なスペースを占有し、読むのに余分な時間がかかります。したがって、コメントも次のようにする必要がありますコンパクト。

キー・アイデア

コメントは情報とスペースの比率を高くする必要があります。

この章の残りの部分では、これを行う方法の例を示します。

コメントをコンパクトに保つ

C++型定義のコメントの例を次に示します。

```
// intはカテゴリタイプです。  
// 内側のペアの最初の浮動小数点は「スコア」、// 2番目は  
「重み」です。  
typedef hash_map<int,ペア<float, float>>スコアマップ;
```

しかし、たった1行で説明できるのに、なぜ3行も使って説明するのでしょうか?

```
// カテゴリタイプ->(スコア、重み)  
typedef hash_map<int,ペア<float, float>>スコアマップ;
```

一部のコメントには3行のスペースが必要ですが、これはその中にはありません。

曖昧な代名詞を避ける

古典的な「誰が一番ですか?」スキットのイラストでは、代名詞が物事を非常に混乱させる可能性があります。

読者が代名詞を「解決」するには余分な作業が必要です。また、場合によっては、「それ」または「これ」が何を指しているのかが不明瞭です。以下に例を示します。

```
// データをキャッシュに挿入しますが、それまず大きすぎます。
```

このコメントでは、「それ」はデータまたはキャッシュを指す場合があります。コードの残りの部分を読めば、おそらくそれが理解できるでしょう。しかし、もしそうしなければならないとしたら、そのコメントには何の意味があるのでしょうか?

最も安全なのは、混乱の可能性がある場合に代名詞を「埋める」ことです。前の例では、「それ」が「データ」とあると仮定しましょう。

```
// データをキャッシュに挿入しますが、データまず大きすぎます。
```

これが最も簡単な変更です。「それ」を完全に明確にするために文を再構成することもできます。

```
// データが十分に小さい場合は、キャッシュに挿入します。
```

ポーランド語のずさんな文章

多くの場合、コメントをより正確にすることは、コメントをよりコンパクトにすることと並行して行われます。

Web クローラーの例を次に示します。

```
# この URL を以前にクロールしたことがあるかどうかに応じて、異なる優先順位を与えます。
```

この文は問題ないように思えるかもしれません、次のバージョンと比較してください。

```
# これまでにクロールしたことのない URL を優先します。
```

この文はより単純で、短く、より直接的です。ということも説明されていますより高いクロールされていない URL が優先されます。前のコメントにはその情報が含まれていませんでした。

関数の動作を正確に説明する

ファイル内の行数をカウントする関数を作成したところを想像してください。

```
// このファイルの行数を返します。int  
CountLines(文字列ファイル名) { ... }
```

このコメントはあまり正確ではありません。「線」を定義する方法はたくさんあります。考慮すべきいくつかの例外的なケースを次に示します。

- """ (空のファイル) - 0 行か 1 行か?
- 」こんにちは"-0行ですか1行ですか?
- 」こんにちは\n"-1行か2行でしょうか?
- 」こんにちは\n 世界"-1行か2行でしょうか?
- 」こんにちは\n\r 残酷な\n 世界\r"-2行、3行、それとも4行でしょうか?

最も単純な実装は、改行の数をカウントすることです (\n) 文字。(これは Unix コマンドの方法です) トイレの実装に一致する、より適切なコメントを次に示します。

```
// ファイル内の改行バイト ('\n') の数をカウントします。int  
CountLines(文字列ファイル名) { ... }
```

このコメントは最初のバージョンよりもそれほど長くはありませんが、より多くの情報が含まれています。これは、改行がない場合に関数が 0 を返す可能性があることを読者に伝えます。また、読者にキャリッジリターン (\r) 無視されます。

特殊なケースを示す入出力例を使用する

コメントに関しては、慎重に選択された入力/出力の例は 1000 の言葉に匹敵します。

たとえば、文字列の一部を削除する一般的な関数を次に示します。

```
// 入力 'src' から 'chars' のサフィックス/プレフィックスを削除します。 String Strip(String src, String chars) { ... }
```

このコメントは次のような質問には答えられないため、あまり正確ではありません。

- は文字削除するのは部分文字列全体でしょうか、それとも事実上、順序付けされていない文字のセットだけでしょうか？

- 複数ある場合はどうなるか文字の終わりにソース？

代わりに、適切に選ばれた例が次の質問に答えることができます。

```
// ...  
// 例: Strip("abba/a/ba", "ab") は "/a/" を返します String  
Strip(String src, String chars) { ... }
```

この例では、次の完全な機能を「披露」しています。ストリップ()。これらの質問に答えられない場合、より単純な例は役に立たないことに注意してください。

```
// 例: Strip("ab", "a") は "b" を返します
```

イラストを使用できる関数の別の例を次に示します。

```
// 要素 <pivot が>= pivot の前に来るよう 'v' を再配置します。  
// 次に、v[i] < ピボット (ピボット未満の場合は -1) となる最大の 'i' を返します。 int  
Partition(vector<int>* v, int pivot);
```

このコメントは実際には非常に正確ですが、視覚化するのが少し難しいです。さらに詳しく説明するために含めることができる例を次に示します。

```
// ...  
// 例: Partition([8 5 9 8 2], 8) の結果は [5 2 | 8 9 8] と 1 を返します int Partition(vector<int>*  
v, int pivot);
```

私たちが選択した具体的な入力/出力の例については、言及すべき点がいくつかあります。

- ピボットは、そのエッジケースを示すベクトル内の要素に等しい。
- ベクトルに重複を入れます (8) これが許容可能な入力であることを示します。
- 結果のベクトルはソートされていません。ソートされていると、読者は誤った考えを抱く可能性があります。
- 戻り値が1、私たちは確認しました1ベクトル内の値でもありませんでした。それは混乱を招くでしょう。

コードの意図を述べる

前の章で述べたように、コメントは多くの場合、コードを書いたときに何を考えていたかを読者に伝えることを目的としています。残念ながら、多くのコメントは、多くの新しい情報を追加することなく、コードの動作を文字通りに説明するだけで終わります。

そのようなコメントの例を次に示します。

```
void DisplayProducts(list<Product> 製品) {  
    products.sort(製品を価格で比較);  
  
    // 逆の順序でリストを反復処理します  
    for (list<Product>::reverse_iterator it = products.rbegin(); it != products.rend();  
         ++それ)  
        DisplayPrice(it->price);  
  
    }      。 。 。
```

このコメントはその下の行を説明するだけです。代わりに、次のより良いコメントを検討してください。

```
// 各価格を最高値から最低値まで表示します  
for (list<Product>::reverse_iterator it = products.rbegin(); ... )
```

このコメントは、プログラムがより高いレベルで何を行っているかを説明します。これは、プログラマーがコードを書いたときに考えていたこととより一致しています。

興味深いことに、このプログラムにはバグがあります。の製品を価格で比較関数 (図示せず) はすでに、より高価な商品を最初に並べ替えています。コードは作成者の意図とは逆のことを行っています。

これが、2番目のコメントの方が優れている理由です。バグにもかかわらず、最初のコメントは技術的には正しいです (ループは逆の順序で反復されます)。しかし、2番目のコメントでは、読者は作成者の意図 (高価格の商品を最初に表示する) がコードの実際の動作と矛盾していることに気づく可能性が高くなります。実際、コメントは次のように機能します。冗長性チェック。

結局のところ、最良の冗長性チェックは単体テストです (「[第14章 テストと可読性](#)」)。しかし、プログラムの意図を説明するこのようなコメントを持つことには依然として価値があります。

「名前付き関数パラメータ」のコメント

次のような関数呼び出しを見たとしましょう。

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

この関数呼び出しは、整数リテラルと布尔値リテラルが渡されるため、少し謎めいています。

Pythonなどの言語では、引数を名前で割り当てることができます。

```
def Connect(タイムアウト、use_encryption): ...  
  
# 名前付きパラメータを使用して関数を呼び出す  
Connect(タイムアウト =10、use_encryption =間違い)
```

C++ や Java などの言語では、これを行うことはできません。ただし、インラインコメントを使用しても同じ効果が得られます。

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

// コメント化されたパラメーターを使用して関数を呼び出します
接続する (/* タイムアウト_ms = */10, /*use_encryption = */間違い) ;
```

最初のパラメータに「名前を付けた」ことに注意してくださいtimeout_msの代わりにタイムアウト。理想的には、関数の実引数は次のようにになります。タイムアウト_ms、しかし、何らかの理由でこの変更ができない場合、これは名前を「改善」する便利な方法です。

ブール引数に関しては、/* を入れることが特に重要です。名前 = */でフロント値の。コメントを入れる後ろにこの値は非常にわかりにくいです：

```
// これはやめてください!
Connect( ... , false/*use_encryption */);

// これもやめてください!
Connect( ... , false/* =use_encryption */);
```

これらの例では、間違い「暗号化を使用する」または「暗号化を使用しない」を意味します。

ほとんどの関数呼び出しではこのようなコメントは必要ありませんが、謎に見える引数を説明するのに便利な(そしてコンパクトな)方法です。

情報量の多い単語を使用する

何年もプログラミングをしていると、同じ一般的な問題と解決策が繰り返し発生することに気づきます。多くの場合、これらのパターンやイディオムを説明するために開発された特定の単語やフレーズが存在します。これらの単語を使用すると、コメントをよりコンパクトにすることができます。

たとえば、次のようなコメントがあったとします。

```
// このクラスには、データベースと同じ情報を格納するメンバーが多数含まれていますが、// 速度を上げるためにここに格納されています。このクラスが後で読み取られるとき、// それらのメンバーが最初にチェックされ、存在するかどうかが確認され、存在する場合は返されます。それ以外の場合は、// データベースが読み取られ、そのデータは次回のためにそれらのフィールドに保存されます。
```

代わりに、次のように言うこともできます。

```
// このクラスはキャッシュ層データベースに。
```

別の例として、次のようなコメントがあります。

```
// 番地から余分な空白を削除し、「Avenue」を「Ave」に変えるなど、他の多くのクリーンアップを// 実行します。このようにして、少し異なる方法で入力された2つの異なる番地がある場合、// それらは同じクリーンアップされたバージョンになり、// これらが等しいことを検出できます。
```

代わりに次のようにすることもできます。

//正規化住所(余分なスペースを削除、「Avenue」->「Ave.」など)

「ヒューリスティック」、「ブルートフォース」、「素朴な解決策」など、多くの意味を詰め込んだ単語やフレーズがたくさんあります。少し長々と感じられるコメントがある場合は、それが典型的なプログラミング状況として説明できるかどうかを確認してください。

まとめ

この章では、できるだけ小さなスペースにできるだけ多くの情報を詰め込むコメントの書き方について説明します。具体的なヒントは次のとおりです。

- ・「it」や「this」などの代名詞が複数のものを指す場合は避けてください。
- ・関数の動作を可能な限り正確に記述します。
- ・慎重に選択した入力/出力例を使用してコメントを説明します。
- ・明白な詳細ではなく、コードの高レベルの意図を述べます。
- ・インラインコメントを使用します(例:Function(* arg = */ ...))謎の関数の引数を説明します。
- ・多くの意味を詰め込んだ言葉を使用して、コメントを簡潔にしましょう。

パート II

ループとロジックの簡素化

でパート I では、表面レベルの改善について説明しました。これは、一度に 1 行ずつコードの読みやすさを改善する簡単な方法で、大きなリスクや労力をかけずに適用できます。

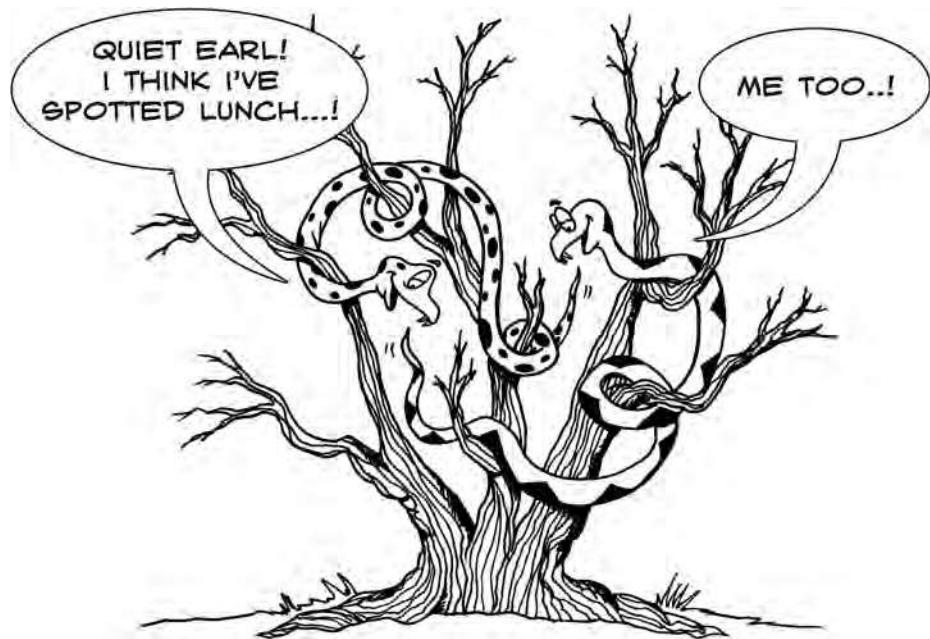
次のパートでは、プログラムの「ループとロジック」、つまりコードを機能させるための制御フロー、論理式、変数についてさらに詳しく説明します。いつものように、私たちの目標は、コードのこれらの部分を理解しやすくすることです。

これは、コードの「精神的な荷物」を最小限に抑えることによって実現されます。複雑なループ、巨大な式、または多数の変数を見るたびに、頭の中の精神的な荷物が増えます。もっと真剣に考えて、より多くのことを覚える必要があります。これは「わかりやすい」とは真逆です。

コードに多くの精神的負担がかかると、バグが見逃される可能性が高く、コードの変更が難しくなり、作業が楽しくなくなります。

第七章

制御フローを読みやすくする



コードに条件文、ループ、その他の要素がなかった場合制御フローステートメントは非常に読みやすいでしょう。これらのジャンプと分岐は難しい作業であり、コードがすぐに混乱する可能性があります。この章では、コード内の制御フローを読みやすくすることについて説明します。

キー・アイデア

フローを制御するためのすべての条件文、ループ、その他の変更は、できるだけ「自然」なものにしてください。つまり、読者がコードを止めて読み直すことのない方法で記述してください。

条件文の引数の順序

次の 2 つのコードのうち、読みやすいのはどれですか。

if (長さ >= 10)

または

if (10 <= 長さ)

ほとんどのプログラマーにとって、最初の方がはるかに読みやすいです。しかし、次の 2 つはどうでしょうか。

while (受信バイト数 < 期待バイト数)

または

while (期待バイト数 > 受信バイト数)

繰り返しますが、最初のバージョンの方が読みやすいです。しかし、なぜ？一般的なルールは何ですか？書いたほうが良いかどうかはどうやって判断しますか $a < b$ または $b > a$ ？

私たちが役立つと判断したガイドラインは次のとおりです。

左側	右側
「尋問されている」という表現は、その価値がさらに流動的です。	値がより一定である、比較対象の式。

このガイドラインは英語の用法と一致しており、「年間少なくとも 10 万ドル稼いでいる場合」または「少なくとも 18 歳である場合」と言うのは非常に自然です。「18 歳があなたの年齢以下の場合」というのは不自然です。

これはその理由を説明します while (受信バイト数 < 期待バイト数) のほうが読みやすいです。受信バイト数はチェックしている値であり、ループの実行につれて増加します。bytes_expected 比較されるより「安定した」値です。

「ヨーダ記法」：まだ役に立ちますか？

一部の言語 (C および C++ を含むが Java は除く) では、代入をもし状態：

```
if (オブジェクト==ヌル) ...
```

おそらくこれはバグであり、プログラマーは実際には次のことを意図していました。

```
if (オブジェクト==NULL) ...
```

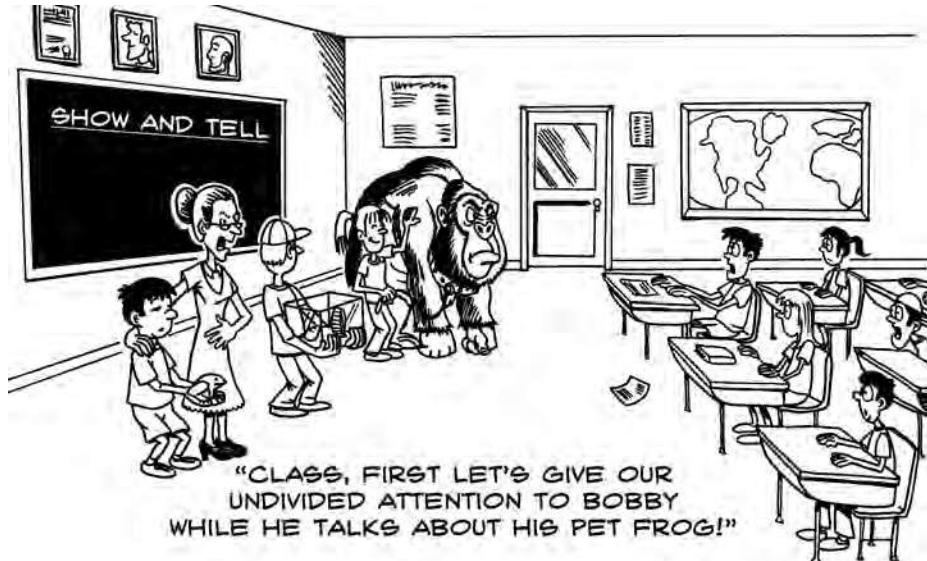
このようなバグを防ぐために、多くのプログラマーは引数の順序を入れ替えます。

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

このように、== が誤って = として書かれた場合、式は if (NULL = オブジェクト) コンパイルすらしません。

残念ながら、順序を入れ替えるとコードが少し不自然になってしまいます。(ヨーダが言うように、「それについて私が言うことは何もありません。」) ありがたいことに、最新のコンパイラは次のようなコードに対して警告します。(obj=NULL) の場合、そのため、「ヨーダ記法」は過去のものになりつつあります。

if/else ブロックの順序



if/else ステートメントを作成するときは、通常、ブロックの順序を自由に入れ替えることができます。たとえば、次のように書くこともできます。

```
if (a == b) {
    // ケース 1 ...
} それ以外 {
    // ケース 2 ...
}
```

または次のようにします。

```
もし (a != b) {
    // ケース 2 ...
} それ以外 {
    // ケース 1 ...
}
```

これについてはこれまであまり考えたこともなかったかもしれません、場合によっては、一方の順序を他方の順序よりも優先する正当な理由がある場合があります。

- 対処することを好みます。ポジティブ否定ではなく大文字を最初に置きます—例:if (デバッグ) その代わりのif (!デバッグ)。
- 対処することを好みます。より単純なままで最初にそれを邪魔にならないようにします。このアプローチでは、次の両方が可能になる可能性もあります。もしそしてそのそれ以外同時に画面上に表示されるので便利です。
- より多くのことに対処することを好む面白いまたは目立つケースを最初に。

場合によっては、これらの設定が矛盾する場合があり、判断を下さなければなりません。しかし多くの場合、明確な勝者が存在します。

たとえば、次の Web サーバーを構築しているとします。応答 URL にクエリ パラメータが含まれているかどうかに基づいてすべて展開：

```
if (!url.HasQueryParameter("expand_all")) {
    応答.Render(アイテ
    ム);。。。
} それ以外 {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    。。。
}
```

読者が最初の行を一目見たとき、彼女の脳はすぐに次のことを考えます。すべて展開場合。それは誰かが「ピンクの象のことを考えるな」と言ったときのようなものです。それについて考えずにはいられません。「してはいけないこと」は、より珍しい「ピンクの象」によってかき消されています。

ここ、すべて展開私たちのピンクの象です。これはより興味深いケースであるため(ポジティブなケースであります)、最初に対処しましょう。

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

```

    。 。 。
} それ以外{
    応答.Render(アイテ
    ム);。 。 。
}

```

一方、否定的な場合の状況は次のとおりです。はより単純で、より興味深い/危険なものなので、最初に扱います。

```

ファイルでない場合:
# エラーをログに記録します ...
それ以外:
#。。。。

```

繰り返しになりますが、詳細によっては判断が下される可能性があります。

要約すると、私たちのアドバイスは、これらの要素に注意を払い、次のようなケースに注意することです。もしそうでなければおかしな順番になっています。

?: 条件式 (別名「三項演算子」)

C系言語では、条件式を次のように記述できます。条件は ? a:b、これは本質的にコンパクトな書き方です if (cond) { a } else { b }。

可読性に対するその影響については議論の余地があります。支持者は、通常は複数行が必要な内容を1行で書くことができる良い方法だと考えています。反対派は、読みにくく、デバッガでのステップ実行が難しい可能性があると主張しています。

以下は、三項演算子が読みやすくコンパクトな例です。

```
time_str += (時間 >= 12) ? "午後" : "午前";
```

三項演算子を避けて、次のように書くこともできます。

```

if (時間 >= 12) {
    time_str += "午後";
} それ以外{
    time_str += "午前";
}

```

これは少し冗長で冗長です。この場合、条件式は合理的であると思われます。

ただし、これらの式はすぐに読みにくくなる可能性があります。

```
戻り指数 >= 0 ? 仮数 * (1 << 指数) : 仮数 / (1 << -指数);
```

ここで、三項演算子はもはや2つの単純な値の間で選択するだけではありません。このようなコードを記述する動機は通常、「すべてを1行に詰め込む」ことです。

キーイデア

行数を最小限に抑えるのではなく、理解するのに必要な時間を最小限に抑えることがより良い指標となります。

ロジックをもし/そうでなければこのステートメントにより、コードがより自然になります。

```
if(指数 >= 0) {  
    仮数 * (1 << 指数) を返します。} それ以  
外 {  
    仮数 / (1 << -指数) を返します。  
}
```

アドバイス

デフォルトでは、もし/そうでなければ。3項?: は、最も単純な場合にのみ使用してください。

do/while ループを避ける



Perl と同様に、多くの評判の高いプログラミング言語には、do {式} while (条件) ループ。の表現少なくとも 1 回は実行されます。以下に例を示します。

```
// 'node' から始めて、指定された 'name' をリスト内で検索します。//  
'max_length' を超えるノードは考慮しないでください。  
public boolean ListHasNode(ノードノード, 文字列名, int max_length) {  
    する {  
        if (node.name().equals(name))  
            true を返します。  
        ノード = ノード.next();  
        while (node != null && --max_length > 0);  
        false を返します。  
    }
```

何がおかしいのか一方を行いますループとは、コードのブロックが条件に基づいて再実行される場合とされない場合があります。下にそれ。通常、論理条件は次のとおりです。その上彼らが守るコード - これがコードの動作方法ですもし、その間に、そしてのために発言。通常、コードは上から下に読むので、一方を行います少し不自然です。多くの読者はコードを2回読んでしまいます。

その間ループ内のコードブロックを読み取る前にすべての反復の条件がわかるため、ループが読みやすくなります。しかし、コードを削除するためだけにコードを複製するのは愚かです。一方を行います：

```
// do/while の真似 — これはやめてください! 体
```

```
その間 (状態){  
    本体 (もう一度)  
}
```

幸いなことに、実際にはほとんどの場合、一方を行いますループは次のように書くこともできます
その間最初のループ：

```
public boolean ListHasNode(ノードノード, 文字列名, int max_length){  
    while (node != null && max_length-- > 0){  
        if (node.name().equals(name)) が true を返します。  
        ノード = ノード.next();  
    }  
    false を返します。  
}
```

このバージョンには、次の場合でも動作するという利点もあります。最大長が0であるか、ノードはヌル。

避けるべきもう一つの理由一方を行いますそれは続くステートメント内では混乱を招く可能性があります。たとえば、このコードは何をするのでしょうか？

```
する{  
    続く;  
    while (偽);
```

永久にループするのか、それとも1回だけループするのか？ほとんどのプログラマーは立ち止まって考えなければなりません。（ループは1回だけ行う必要があります。）

全体として、C++の作成者であるBjarne Stroustrupが最もよく言っています（C++プログラミング言語）：

私の経験では、doステートメントはエラーと混乱の原因です。…私は「目の前で見える」状態が好きです。そのため、私はdoステートメントを避ける傾向があります。

関数から早めに戻る

プログラマの中には、関数に複数の要素を含めるべきではないと信じている人もいます。戻る発言。これはナンセンスです。関数から早めに戻ることはまったく問題なく、多くの場合望ましいことです。例えば：

```
public boolean Contains(String str, String substr){  
    if (str == null || substr == null) false を返します。 if  
    (substr.equals("")) true を返します。
```

```
    . . .  
}
```

これらの「ガード句」なしでこの機能を実装するのは非常に不自然です。

単一の終了ポイントが必要な動機の1つは、関数の下部にあるすべてのクリーンアップコードが確実に呼び出されるようにするためです。しかし、現代の言語では、この保証を達成するためのより洗練された方法が提供されています。

言語	クリーンアップコードの構造化イディオム
C++	デストラクタ
Java、Python	最後に試してみてください
パイソン	と
C#	を使用して

純粋な C には、関数の終了時に特定のコードをトリガーするメカニズムがありません。したがって、大量のクリーンアップコードを含む大規模な関数がある場合、早期に返すことを正しく行うのは困難になる可能性があります。この場合、他のオプションには、関数のリファクタリングや、クリーンアップに移動します。

悪名高き後藤

C 以外の言語では、ほとんど必要ありません。後藤仕事を成し遂げるためのもっと良い方法がたくさんあるからです。後藤また、すぐに手に負えなくなり、コードを理解しにくくすることでも知られています。

しかし、まだ見ることができます後藤さまざまな C プロジェクト (特に Linux カーネル) で使用されます。すべての使用を拒否する前に後藤冒頭として、なぜいくつかの使用法があるのか を詳しく分析することは有益です。後藤他の人よりも優れています。

最も単純で最も無害な使用法後藤シングルと一緒にです出口関数の最後に:

```
if (p == NULL) 出口へ進む;
```

```
    . . .
```

出口:

```
fclose(ファイル1);  
fclose(ファイル2);  
    . . .
```

戻る;

もしこれが唯一の形式だったら後藤許可された、後藤あまり問題ないでしょう。

問題が発生する可能性があるのは、複数後藤ターゲット、特にパスが交差するとき。特に、後藤それは行きます上向き本物のスペゲッティ コードを作成でき、確実に構造化ループに置き換えることができます。ほとんどの時間、後藤避けるべきです。

ネストを最小限に抑える

深くネストされたコードは理解しにくいです。ネストの各レベルは、読者の「精神的スタック」に追加の条件を押し付けます。読者が閉じ中括弧 {} を見ると、スタックを「ポップ」してその下にある条件を思い出すのが難しい場合があります。

この比較的単純な例を次に示します。自分がどのブロック状態にあるかを再確認するために振り返ってみてください。

```
if (user_result == 成功) {
    if (permission_result != SUCCESS) {
        Reply.WriteErrors("読み取り権限エラー");
        Reply.Done();
        戻る;
    }
    Reply.WriteErrors("");
} それ以外 {
    Reply.WriteErrors(user_result);
}

}
Reply.Done();
```

最初の閉じ中括弧を見たとき、次のように考えなければなりません。おお、許可結果 != 成功 終わつたばかりなので今は許可結果 == 成功、これはまだブロック内にあります。

user_result == 成功。

全体として、次の値を維持する必要があります。ユーザー結果そして許可_結果常に頭の中になります。そしてそれぞれのようにもし {} ブロックが閉じると、対応する値を頭の中で切り替える必要があります。

この特定のコードは、成功そして非成功状況。

ネスティングが蓄積される仕組み

前のサンプルコードを修正する前に、どのようにしてそのような結果になったのかについて話しましょう。もともと、コードは単純でした。

```
if (user_result == 成功) {
    Reply.WriteErrors("");
} それ以外 {
    Reply.WriteErrors(user_result);
}
Reply.Done();
```

このコードは完全に理解できます。どのようなエラー文字列を書き込むかを判断し、次の処理を実行します。返事。

しかし、プログラマは2番目の操作を追加しました。

```
if (user_result == 成功) {
    if (permission_result != SUCCESS) {
        Reply.WriteErrors("読み取り権限エラー");
```

```
    Reply.Done();
    戻る;
}
Reply.WriteErrors("");

. . .
```

この変更は理にかなっています。プログラマーは挿入する新しいコードの塊を持っていて、それを挿入する最も簡単な場所を見つけました。この新しいコードは彼女の心の中で新鮮で、精神的に「大胆」でした。そして、この変更の「差分」は非常にきれいで、単純な変更のように見えます。

しかし、後で他の人がそのコードに遭遇すると、そのコンテキストはすべて失われます。これは、このセクションの冒頭で最初にコードを読んだときの状況であり、一度にすべてを理解する必要がありました。

キー・アイデア

変更を加えるときは、新しい視点からコードを見てください。一歩下がって全体を見てみましょう。

早期復帰による巣の撤去

さて、コードを改善しましょう。このようなネストは、「失敗ケース」をできるだけ早く処理し、関数から早めに戻ることで削除できます。

```
if (user_result != SUCCESS) {
    Reply.WriteErrors(user_result);
    Reply.Done();
    戻る;
}

if (permission_result != SUCCESS) {
    Reply.WriteErrors(permission_result);
    Reply.Done();
    戻る;
}

Reply.WriteErrors("");
Reply.Done();
```

このコードには、2 レベルではなく 1 レベルのネストしかありません。しかし、もっと重要なことは、読者は決して自分の心のスタックから何かを「ポップ」する必要がないということです。もしブロックは戻る。

ループ内のネストの削除

早く戻るというテクニックは、常に適用できるとは限りません。たとえば、ループ内にネストされたコードの例を次に示します。

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

```

if (results[i]->name != "") {
    cout << "候補を検討しています..." << endl;
    . . .
}
}
}

```

ループ内で、早期に戻ると同様のテクニックは次のとおりです。続く：

```

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

    if (結果[i]->名前 == "") 続く; cout << "候補を検討し
    ています..." << endl;
    . . .
}

```

それと同じように、if (...) が戻る; 関数のガード句として機能します。(...) 繼続する場合; ステートメントはループのガード句として機能します。

一般に、続くこのステートメントは読者を混乱させる可能性があります。後藤ループの中。ただし、この場合、ループの各繰り返しは独立しているため (ループは「for each」です)、読者は次のことを簡単に理解できます。続く単に「この項目をスキップする」という意味です。

実行の流れを理解できますか?



この章では、ループ、条件、その他のジャンプを読みやすくする方法など、低レベルの制御フローについて説明してきました。ただし、プログラムの「フロー」についても高いレベルで考える必要があります。理想的には、プログラムの実行パス全体を簡単にたどることができます。

主要 () そして、プログラムが終了するまで、ある関数が別の関数を呼び出しながら、コードを頭の中でステップ実行していきます。

ただし、実際には、プログラミング言語やライブラリには、コードを「舞台裏」で実行したり、コードを追跡するのを困難にしたりする構造が含まれています。ここではいくつかの例を示します。

プログラミング構造	高レベルのプログラムフローがどのように曖昧になるか
ねじ切り	どのコードがいつ実行されるかは不明です。
シグナル/割り込みハンドラー	特定のコードはいつも実行される可能性があります。
例外	複数の関数呼び出しを通じて実行がバブル化する可能性があります。
関数ポインタと無名関数	コンパイル時には分からぬいため、どのコードが実行されるかを正確に知ることは困難です。
仮想メソッド	object.virtualMethod()未知のサブクラスのコードを呼び出す可能性があります。

これらの構造の一部は非常に便利で、コードを読みやすくし、冗長性を減らすこともできます。しかし、プログラマとして、私たちは時々、読者が後でコードを理解するのがどれほど難しいかを理解せずに、夢中になってそれらを過度に使用してしまうことがあります。また、これらの構造により、バグの追跡が非常に困難になります。

重要なのは、コードの大部分でこれらの構造を使用しないようにすることです。これらの機能を悪用すると、(漫画のように)スリーカードモンテのゲームのようにコードをトレースする可能性があります。

まとめ

コードの制御フローを読みやすくするためにできることはたくさんあります。

比較を書くとき(while (期待バイト数 > 受信バイト数))、変化する値を左側に、より安定した値を右側に置くことをお勧めします (while (受信バイト数 < bytes_expected))。

ブロックの順序を変更することもできます。もしそうでなければ声明。一般に、最初にポジティブな/簡単な/興味深いケースを処理するようにしてください。これらの基準が矛盾する場合もありますが、矛盾しない場合は、これに従うのが良い経験則です。

三項演算子 (?:) などの特定のプログラミング構造体一方を行いますループ、そして後藤多くの場合、コードが読めなくなります。より明確な代替手段がほとんどの場合存在するため、通常は使用しないことが最善です。

ネストされたコード ブロックを理解するには、さらに集中力が必要です。新しいネストごとに、リーダーの「スタックにプッシュ」するために、より多くのコンテキストが必要になります。代わりに、深いネストを避けるために、より「線形」なコードを選択してください。

早期に復帰すると、ネストが削除され、コード全体がクリーンアップされます。「ガード ステートメント」(関数の先頭で単純なケースを処理する)は特に便利です。

巨大な式を分解する



ダイオウイカは驚くべき知的な動物ですが、ほぼ完璧に近い体の設計には致命的な欠陥が1つあります。それは、食道の周りを包み込むドーナツ型の脳を持っていることです。そのため、一度にたくさんの食べ物を飲み込むと、脳に損傷を与えます。

これはコードと何の関係があるのでしょうか? そうですね、大きすぎる「チャンク」に分割されたコードも同様の影響を与える可能性があります。最近の研究によると、私たちのほとんどは一度に3つまたは4つの「こと」しか考えることができません。^{*簡単に言えば、コードの表現が大きくなるほど、理解するのが難しくなります。}

キー・アイデア

巨大な表現をより理解しやすい部分に分割します。

この章では、コードを飲み込みやすくするためにコードを操作および分解するさまざまな方法を説明します。

変数の説明

式を分解する最も簡単な方法は、より小さな部分式をキャプチャする追加の変数を導入することです。この追加の変数は、部分式の意味を説明するのに役立つため、「説明変数」と呼ばれることもあります。

以下に例を示します。

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

以下は同じコードに説明変数を加えたものです。

```
ユーザー名=line.split(':')[0].strip() ifユーザー  
名=="根":  
    . . .
```

要約変数

たとえそうでない表現でも必要説明する場合でも(それが何を意味するのか理解できるため)、その式を新しい変数に取り込むと便利な場合があります。私たちはこれをと呼んでいますサマリー変数その目的が単に、より大きなコードの塊を、より簡単に管理および検討できる小さな名前に置き換えることである場合。

たとえば、次のコードの式を考えてみましょう。

```
もし (request.user.id == document.owner_id) {  
    // ユーザーはこのドキュメントを編集できます...  
}
```

* コーワン、N. (2001)。短期記憶における魔法の数字4:精神的記憶能力の再考。行動科学と脳科学、24、97-185。

。 。 。

```
もし (request.user.id != document.owner_id) {  
    // ドキュメントは読み取り専用です...  
}
```

表現`request.user.id == document.owner_id`それほど大きくないようと思えるかもしれませんか、変数が5つあるため、考えるのに少し時間がかかります。

このコードの主な概念は、「ユーザーはドキュメントを所有しているか?」です。この概念は、要約変数を追加することにより明確に述べることができます。

```
最終ブール値user_owns_document = (request.user.id == document.owner_id);
```

```
もし (ユーザー所有のドキュメント) {  
    // ユーザーはこのドキュメントを編集できます...  
}
```

。 。 。

```
もし (!ユーザー所有のドキュメント) {  
    // ドキュメントは読み取り専用です...  
}
```

大したことではないようと思えるかもしれないが、この声明は、`if(ユーザー所有のドキュメント)`は少し考えやすくなります。また、ユーザー所有のドキュメント先頭で定義されているものは、「これがこの関数全体で参照する概念である」ことを読者に事前に伝えます。

ド・モルガンの法則の使用

回路や論理のコースを受講したことがあれば、ド・モルガンの法則を覚えているかもしれません。これらは、ブール式を同等の式に書き換える2つの方法です。

- 1) (a または b または c) ではない ... (a ではない)、(b ではない)、および (c ではない) ... (a ではない) または (b ではない) または (c ではない)
- 2) (a と b と c) ではない

これらの法律を覚えるのが難しい場合は、簡単に要約すると次のようになります。ないそして切り替えますおよび「または。」(あるいは、その逆に、「因数分解する」ことになります。ない。")

これらの法則を使用して、ブール式を読みやすくすることができる場合があります。たとえば、コードが次の場合:

```
もし (!!(ファイルが存在します && !is_protected)) Error("申し訳ありません、ファイルを読み取れませんでした。");
```

これは次のように書き換えることができます。

```
もし (!file_exists || is_protected) Error("申し訳ありません、ファイルを読み取れませんでした。");
```

短絡ロジックの悪用

ほとんどのプログラミング言語では、布尔演算子は短絡評価を実行します。たとえば、次のようなステートメント `if (a || b)` 評価しません `b` もあるそれは本当です。この動作は非常に便利ですが、複雑なロジックを実行するために悪用される場合があります。

著者の一人がかつて書いたステートメントの例を次に示します。

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

このコードは英語で「このキーのバケットを取得します。」と言っています。バケットが `null` でない場合は、バケットが占有されていないことを確認してください。」

たった1行のコードですが、ほとんどのプログラマーは立ち止まって考えてしまします。次に、このコードと比較してください。

```
バケット = FindBucket(キー);
if (バケット != NULL) assert(!bucket->IsOccupied());
```

まったく同じことを行い、2行のコードですが、はるかに理解しやすくなっています。

では、そもそもなぜコードは単一の巨大な式として記述されたのでしょうか? 当時はそれがとても賢いと感じました。ロジックを簡潔なコードの塊にまで切り詰めるのには、ある種の喜びがあります。それは当然です。それはミニチュアパズルを解くようなもので、私たちは皆、仕事を楽しみたいと思っています。問題は、コードを読み進める人にとって、このコードが精神的なスピードを高めることだったということです。

キーイデア

コードの「賢い」ナゲットには注意してください。後で他の人がコードを読むと、混乱を招くことがよくあります。

これは、短絡的な動作の利用を避けるべきだという意味ですか? いいえ、きれいに使用できるケースはたくさんあります。たとえば、次のとおりです。

```
if (オブジェクト && オブジェクト->メソッド()) ...
```

言及する価値のある新しいイディオムもあります。Python、JavaScript、Rubyなどの言語では、「`or`」演算子は引数の1つを返す(布尔値に変換されない)ため、次のようなコードになります。

```
x = a || b || c
```

最初の「真実の」値を選択するために使用できます。a、b、またはc。

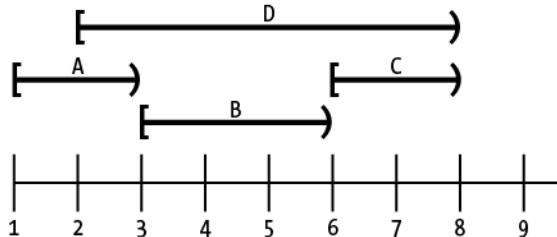
例: 複雑なロジックとの格闘

次を実装していると仮定します範囲クラス:

```
構造体の範囲{
    int の開始;
    意図する;
```

```
// たとえば、[0,5) は [3,8) と重複します bool
OverlapsWith(Range other);
};
```

次の図は、範囲の例をいくつか示しています。

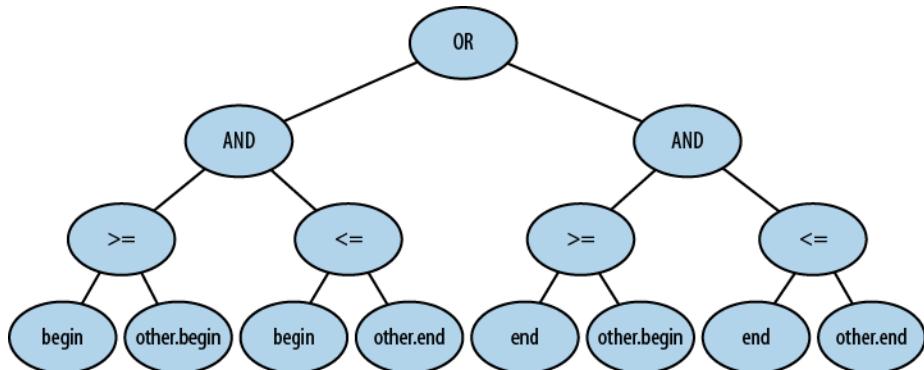


ご了承ください終わりは非包括的です。つまり、A、B、C は互いに重なりませんが、D はそれらのすべてと重なります。

これを実装する試みの 1 つを次に示します。OverlapsWith()—範囲のいずれかの端点が範囲内にあるかどうかを確認します。他の'範囲':

```
bool Range::OverlapsWith(Range other) {
    // 'begin' または 'end' が 'other' 内にあるかどうかを確認します。return (begin >= other.begin && begin <= other.end) ||
    (end >= other.begin && end <= other.end);
}
```

コードの長さはわずか 2 行ですが、多くの処理が行われています。次の図は、関連するすべてのロジックを示しています。



考るべきケースや条件が非常に多いため、バグがすり抜けてしまいがちです。

そういえば、そこにはバグです。前のコードは次のように主張します。範囲 [0,2)[と重複します 2,4)実際にはそうではないのに。

問題は、比較するときに注意しなければならないことです始まり/終わり`<=`または`<`のみを使用して値を指定します。この問題の解決策は次のとおりです。

```
return (begin >= other.begin && begin <その他。終了) ||
       (終わり>other.begin && end <= other.end);
```

今ではそれが正しいですよね？実は、あるんです別のバグ。このコードは、次の場合を無視しています。始まり/終わり完全に取り囲む他の。

このケースを処理する修正も次のとおりです。

```
return (begin >= other.begin && begin < other.end) ||
       (終了 > other.begin && end <= other.end) || (開始
       <= その他.開始 && 終了 >= その他.終了);
```

やあ、このコードは複雑すぎます。このコードを読んで、それが正しいと自信を持ってわかる人はいないでしょう。どうしようか？この巨大な表現をどのように分解できるでしょうか？

よりエレガントなアプローチを見つける

これは、立ち止まって、まったく別のアプローチを検討する必要があるときの1つです。単純な問題(2つの範囲が重なっているかどうかを確認する)として始まったものは、驚くほど複雑なロジックになりました。これは多くの場合、次のような兆候ですもっと簡単な方法があるはずです。

しかし、より洗練されたソリューションを見つけるには創造力が必要です。どうやってやるのですか？テクニックの1つは、「逆の」方法で問題を解決できるかどうかを確認することです。置かれている状況に応じて、配列を逆方向に反復処理したり、データ構造を前方ではなく後方に埋めたりすることを意味する場合があります。

ここでは、その逆に、`OverlapsWith()`「重ならない」です。範囲が2つあるかどうかの判断しないでください可能性が2つしかないため、オーバーラップははるかに単純な問題であることがわかります。

1. 他の範囲は、この範囲が始まる前に終了します。
2. この範囲が終了した後に、他の範囲が始まります。

これを非常に簡単にコードに変換できます。

```
bool Range::OverlapsWith(Range other) {
    if (other.end <= begin) は false を返します。if // 始める前にそれらは終わりま
    (other.begin >= end) は false を返します。    す // 終わった後に始まります

    true を返します。// 残された唯一の可能性: 重複している
}
```

ここでのコードの各行は非常に単純で、比較が1回だけ行われます。これにより、読者は`<=`が正しいかどうかに集中するのに十分な知力が残ります。

巨大なステートメントを分解する

この章では個々の式の分解について説明しますが、同じテクニックがより大きなステートメントの分解にも適用されます。たとえば、次の JavaScript コードには、一度に多くのことを理解する必要があります。

```
var update_highlight = 関数 (message_num) {
    if ($("#vote_value" + message_num).html() === "上へ") {
        $("#thumbs_up" + message_num).addClass("強調表示"); $
        ("#thumbs_down" + message_num).removeClass("強調表示"); } else
    if ($("#vote_value" + message_num).html() === "Down") {
        $("#thumbs_up" + message_num).removeClass("強調表示"); $
        ("#thumbs_down" + message_num).addClass("強調表示"); } それ
    以外 {
        $("#thumbs_up" + message_num).removeClass("highlighted"); $
        ("#thumbs_down" + message_num).removeClass("強調表示");
    }
};
```

このコード内の個々の式はそれほど大きくありませんが、すべてを組み合わせると、一度に衝撃を与える1つの巨大なステートメントになります。

幸いなことに、式の多くは同じであるため、関数の先頭で要約変数として式を抽出できます(これは、DRY(「繰り返しを繰り返さない」)原則のインスタンスもあります)。

```
var update_highlight = 関数 (message_num) {
    変数いいぞ=$("#サムズアップ"+メッセージ番号); var拒绝=$("##
    サムズダウン"+メッセージ番号); var投票値=$("#投票値"+メッ
    セージ番号).html(); varこんにちは="強調表示されました";

   もし (投票値==="上") {
    いいぞ.addClass(こんにちは); 拒絶
    .removeClass(こんにちは); } else if (
    投票値==="下") {
    いいぞ.removeClass(こんにちは
    ); 拒絶.addClass(こんにちは); } そ
    れ以外 {
    いいぞ.removeClass(こんにちは);
    拒絶.removeClass(こんにちは);
    }
};
```

の創造var hi = "強調表示"厳密には必要ありませんが、あったように六つのコピーには、次のような魅力的な利点がありました。

- 入力ミスを防ぐのに役立ちます。(実際、最初の例では、文字列のスペルが"と間違っていたことに気づきましたか?強調された」5番目の場合は?)
- 線幅がさらに縮小され、コードをスキャンしやすくなります。
- クラス名を変更する必要がある場合、変更できる場所は1つだけです。

式を簡素化する別の創造的な方法

各式で多くのことが起こっている別の例を次に示します。今回は C++ です。

```
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());。。。
}
```

もう一度、あなたの目には長くて似たようなコードが目の前にありますか、まったく同じではありません。10 秒ほど注意深く調べてみると、各行が毎回異なるフィールドに対して同じことを行っていることに気づくかもしれません。

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

C++ では、これを実装するマクロを定義できます。

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

    ADD_FIELD(合計メモリ);
    ADD_FIELD(空きメモリ);
    ADD_FIELD(スワップメモリ );
    ADD_FIELD(ステータス文字列);
    ADD_FIELD(プロセス数);。。。

    # undef ADD_FIELD
}
```

煩雑な要素をすべて取り除いたので、コードを見ると、何が起こっているかの本質をすぐに理解できます。各行が同じことを行っていることは非常に明らかです。

私たちはマクロの使用をあまり推奨しているわけではないことに注意してください。実際、マクロを使用するとコードが混乱し、微妙なバグが発生する可能性があるため、通常はマクロを避けます。しかし、場合によっては、このケースのように、単純であり、読みやすさに明らかな利点をもたらす可能性があります。

まとめ

巨大な表現は考えるのが難しいです。この章では、読者が内容を部分的に理解できるように、内容を細分化するいくつかの方法を示しました。

簡単な手法の1つは、大きな部分式の値を取得する「説明変数」を導入することです。このアプローチには次の3つの利点があります。

- ・巨大な式を細かく分解します。
- ・部分式を簡潔な名前で記述することにより、コードを文書化します。
- ・読者がコード内の主な「概念」を識別するのに役立ちます。

もう1つの手法は、ドモルガンの法則を使用してロジックを操作することです。この手法では、ブール式をよりクリーンな方法で書き換えることができます (例:if (!(a && !b)) になるもし (!a || b))。

複雑な論理条件が次のような小さなステートメントに分解される例を示しました。if (a < b) ...」。実際には、全てこの章の改良されたコード例には、

もし以下のステートメント二その中にある価値観。このセットアップは理想的です。これは常に可能であるとは限りません。場合によっては、問題を「否定」したり、目標とは逆のことを考慮したりする必要があります。

最後に、この章は個々の式の分解について説明していますが、これらと同じテクニックがより大きなコードブロックにも適用されることがよくあります。したがって、複雑なロジックがどこであっても、積極的に分解してください。

第九章

変数と可読性



この章では、変数プログラムを理解しにくくします。

具体的には、次の3つの問題に取り組む必要があります。

1. 変数が増えるほど、それらすべてを追跡するのが難しくなります。
2. 変数のスコープが大きくなるほど、それを追跡する必要がある時間が長くなります。
3. 変数が頻繁に変更されると、その現在の値を追跡することが難しくなります。

次の3つのセクションでは、これらの問題に対処する方法について説明します。

変数の削除

で第8章、巨大な式を分解するでは、「説明」または「概要」変数を導入することでコードが読みやすくなる方法を説明しました。これらの変数は、巨大な式を分割し、ドキュメントの形式として機能するため役立ちました。

このセクションでは、次のような変数を排除することに興味があります。しないでください可読性を向上させます。このような変数が削除されると、新しいコードはより簡潔になり、理解しやすくなります。

次のセクションでは、これらの不要な変数がどのように表示されるかを示すいくつかの例を示します。

役に立たない一時変数

次のPythonコードのスニペットで、次の点を考慮してください。今変数：

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

は今保持する価値のある変数ですか？ いいえ、その理由は次のとおりです。

- 複雑な式を分解していない。
- 説明を明確にするものではありません - 表現datetime.datetime.now()十分明らかです。
- 1回だけ使用されるため、冗長なコードは圧縮されません。

それなし今、コードも同様に理解しやすいです。

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

のような変数は通常、コードが編集された後に残る「残り物」です。変数

今元々は複数の場所で使用されていた可能性があります。あるいは、プログラマーが次のような使用を想定していたのかもしれません。今何度も使用しましたが、実際には必要ありませんでした。

中間結果の削除



配列から値を削除する JavaScript 関数の例を次に示します。

```
var move_one = function (array, value_to_remove) {
    变数削除するインデックス=ヌル;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === 削除する値) {
            削除するインデックス=私;
            壊す;
        }
    }
    もし (削除するインデックス!==ヌル) {
        配列.スプライス(削除するインデックス, 1);
    }
};
```

変数削除するインデックスを保持するためにのみ使用されます 中間結果。このような変数は、結果を取得したらすぐに処理することで削除できる場合があります。

```
var move_one = function (array, value_to_remove) {
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === 削除する値) {
            配列.splice(i, 1);
            戻る;
        }
    }
};
```

コードが早期に返せるようにすることで、削除するインデックスコードが大幅に簡素化されました。

一般に、これは良い戦略です。できるだけ早くタスクを完了する。

制御フロー変数の排除

場合によっては、ループ内に次のコード パターンが表示されることがあります。

```
ブルル値終わり=間違い;  
while /* 条件 */ && ! 終わり {  
 。。。  
 もし (...) {  
  終わり=真実;  
  続く;  
}  
}
```

変数終わりに設定されることもあります真実ループ全体の複数の場所にあります。

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のルールを満たそうとすることがよくあります。そんなルールはないよ！

のような変数終わりこれらは「制御フロー変数」と呼ばれるものです。それらの唯一の目的は、プログラムの実行を制御することです。実際のプログラム データは含まれません。私たちの経験では、構造化プログラミングをうまく活用することで制御フロー変数を削除できることがよくあります。

```
while /* 条件 */ {  
 。。。  
 もし (...) {  
  壊す;  
}  
}
```

このケースは非常に簡単に修正できましたが、もし問題があった場合はどうなるでしょうか。複数単純なネストされたループ 壊す十分ではないでしょうか？このようなより複雑な場合、解決策には多くの場合、コードを新しい関数 (ループ内のコード、またはループ自体) に移動することが含まれます。

同僚にそう思われたいですか 彼らはずっと面接中ですか？

Microsoft の Eric Brechner 氏は、優れた面接の質問には少なくとも 3 つの変数が含まれている必要があると語っています。^{*}おそらく、3 つの変数を同時に扱うと、一生懸命考える必要があるからでしょう。これは、候補者を限界まで押し込むとする面接では当然のことです。しかし、同僚がコードを読んでいる間、まるで面接を受けているように感じてもらいたいでしょうか？

^{*}エリック・ブレクナーのIM ライトの「ハードコード」(Microsoft Press、2007)、p. 166.

変数のスコープを縮小する

「グローバル変数は避けてください」というアドバイスは誰もが聞いたことがあるでしょう。これらすべてのグローバル変数がどこでどのように使用されているかを追跡するのは難しいため、これは良いアドバイスです。また、「名前空間を汚染する」(ローカル変数と競合する可能性のある名前を大量にそこに置く)ことによって、コードがローカル変数を使用するつもりで誤ってグローバル変数を変更したり、その逆の可能性があります。

実際、「範囲を縮小する」のは良い考えです。全てグローバル変数だけでなく、変数も含めます。

キーアイデア

できるだけ少ないコード行で変数を可視にします。

多くのプログラミング言語は、モジュール、クラス、関数、ブロックスコープなど、複数のスコープ/アクセスレベルを提供します。より制限されたアクセスを使用することは、より少ないコード行で変数を「認識」できることを意味するため、一般に優れています。

なぜこれを行うのでしょうか? それは、読者が同時に考えなければならない変数の数を効果的に減らすことができるからです。すべての変数のスコープを2分の1に縮小すると、一度にスコープ内に含まれる変数の数は平均して半分になります。

たとえば、次のように2つのメソッドのみで使用されるメンバー変数を持つ非常に大規模なクラスがあるとします。

```
クラス LargeClass {
    弦str_;

    void メソッド1() {
        str_= ...;
        メソッド2();
    }

    void Method2() {
        // 用途str_
    }

    // 使用しない他のメソッドがたくさんありますstr_...
};
```

ある意味、クラスメンバー変数は、クラスの領域内の「ミニグローバル」のようなものです。特に大規模なクラスの場合、すべてのメンバー変数と各変数を変更するメソッドを追跡するのは困難です。ミニグローバルの数は少ないほど良いです。

この場合、「降格」するのが合理的かもしれません。str_ローカル変数にするには:

```
クラス LargeClass {
    void メソッド1() {
        弦str= ...; 方法2(str
    );
}
```

```
void メソッド 2(文字列str) {  
    // 用途str  
}  
  
};  
// これで他のメソッドは認識できなくなりますstr。
```

クラスメンバーへのアクセスを制限する別 の方法は、次のとおりです。できるだけ多くのメソッドを静的にする。静的メソッドは、「これらのコード行がそれらの変数から分離されている」ことを読者に知らせる優れた方法です。

または、別のアプローチは次のとおりです**大人数のクラスを小さなクラスに分ける**。このアプローチは、小規模なクラスが実際に相互に分離されている場合にのみ役立ちます。相互にメンバーにアクセスする2つのクラスを作成したとしても、実際には何も達成したことにはなりません。

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場合も同様です。 そうする大きな動機は、データ(つまり、変数)を分離することです。

ただし、言語が異なれば、スコープを正確に構成するものについてのルールも異なります。変数のスコープに関する興味深いルールをいくつか指摘したいと思います。

C++ の if ステートメントのスコープ

次の C++ コードがあるとします。

```
支払い情報*情報=データベース.ReadPaymentInfo(); も  
し (情報){  
    cout << "ユーザーの支払い: " <<情報->amount() << endl;  
}
```

```
// 以下にさらに多くのコード行があります ...
```

変数情報関数の残りの部分のスコープ内に残るため、このコードを読んでいる人はそのままにしておくことができます。情報再び使用されるかどうか、またどのように使用されるかが念頭にあります。

しかし、この場合、情報内部でのみ使用されますもし声明。C++ では、実際に次のように定義できます。

情報条件式では次のようにになります。

```
もし (PaymentInfo* 情報 =データベース.ReadPaymentInfo()) {  
    cout << "ユーザーが支払った: " << info->amount() << endl;  
}
```

今、読者は簡単に忘れてしまます情報範囲外になった後。

JavaScript での「プライベート」変数の作成

1 つの関数のみで使用される永続変数があるとします。

提出された=間違い; // 注: グローバル変数

```
var submit_form = 関数(フォーム名){  
    もし (提出された){  
        戻る; // フォームを二重送信しないでください  
    }  
    . . .  
    提出された=真実;  
};
```

のようなグローバル変数提出されたこのコードを読む人に大きな不安を与える可能性があります。どうやら submit_form() を使用する唯一の関数です提出された、しかし、確実に知ることはできません。実際、別の JavaScript ファイルが、という名前のグローバル変数を使用している可能性があります。提出された別の目的でも！

ラップすることでこの問題を防ぐことができます提出されたの中に閉鎖:

```
var submit_form = (関数(){  
    変数提出された=間違い; // 注: 以下の関数によってのみアクセスできます  
  
    戻り関数(フォーム名){  
        もし (提出された){  
            戻る; // フォームを二重送信しないでください  
        }  
        . . .  
        提出された=真実;  
    };  
}());
```

最後の行のかっこに注意してください。匿名の外部関数がすぐに実行され、内部関数が返されます。

このテクニックをこれまで見たことがない場合、最初は奇妙に見えるかもしれません。これには、内部関数のみがアクセスできる「プライベート」スコープを作成する効果があります。これで読者は不思議に思う必要はなくなりました他にどこがやるのか提出された慣れる？または、同じ名前の他のグローバルとの競合を心配する必要があります。（見る [JavaScript: 良い部分](#) このようなテクニックの詳細については、Douglas Crockford 著 [O'Reilly, 2008] を参照してください。）

JavaScript のグローバル スコープ

JavaScriptでキーワードを省略した場合変数定義から (例:x=1の代わりにvar x=1)、変数はグローバルスコープに入れられます。毎JavaScript ファイルと<スクリプト>ブロックはそれにアクセスできます。以下に例を示します。

```
<スクリプト>
var f=関数 () {
    // 危険: 'i' は 'var' で宣言されていません! のために
    (i=0;私は10未満です。i+=1) ...
};

f();
</script>
```

このコードは誤って私グローバルスコープに追加すると、後続のブロックでもそれを参照できるようになります。

```
<スクリプト>
アラート(私); // アラート「10」。「i」はグローバル変数です。</
script>
```

多くのプログラマーはこのスコープ規則を認識していないため、この驚くべき動作によって奇妙なバグが発生する可能性があります。このバグの一般的な症状は、2つの関数が両方とも同じ名前のローカル変数を作成するのに、使用することを忘れた場合です。変数これらの関数は無意識のうちに「クロストーク」を起こし、下手なプログラマーは自分のコンピュータが憑依されているか、RAM が故障していると結論付けるかもしれません。

JavaScript の一般的な「ベスト プラクティス」は次のとおりです。変数を常に使用して定義します。変数キーワード(例えば、var x=1)。この方法では、変数のスコープが、変数が定義されている(最も内側の)関数に制限されます。

Python と JavaScript にはネストされたスコープはありません

C++ や Java などの言語には、ブロックスコープ、ここで変数は内部で定義されていますもし、試してみたら、または同様の構造は、そのブロックのネストされたスコープに限定されます。

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

しかし、Python と JavaScript では、ブロック内で定義された変数が関数全体に「流出」します。たとえば、次の使用に注目してください。例の値この完全に有効な Python コードでは次のようになります。

```
# この時点まで、example_value は使用されていません。リク
エストの場合:
request.values の値の場合:
    値 > 0 の場合:
        例の値=価値
        壊す

debug.loggers のロガーの場合:
    logger.log("例:", 例の値)
```

このスコープ規則は多くのプログラマーにとって驚くべきことであり、このようなコードは読みにくくなります。他の言語では、どこにあるかを見つけるのが簡単になります。例の値最初に定義されたものです。内部にある関数の「左端」に沿って見ていきます。

前の例にもバグがあります。例の値コードの最初の部分で設定されていない場合、2番目の部分で例外が発生します。名前エラー: 'example_value' が定義されていません。」以下を定義することで、この問題を修正し、コードをより読みやすくすることができます。例の値それが使用されている場所に(ネストの観点から)「最も近い共通の祖先」:

例の値=なし

リクエストの場合:

request.values の値の場合:

値 > 0 の場合:

例の値=値

壊す

もし例の値:

debug.loggers のロガーの場合:

logger.log("例:", 例の値)

ただし、これは次の場合です。例の値完全に排除することができます。例の値は中間結果を保持しているだけです。[95 ページの「中間結果の削除」](#)、このような変数は、多くの場合、「できるだけ早くタスクを完了する」ことで排除できます。この場合、サンプル値が見つかったらすぐにログに記録することを意味します。

新しいコードは次のようになります。

```
def LogExample(値):
    debug.loggers のロガーの場合:
        logger.log("例:", 値)
```

リクエストの場合:

request.values の値の場合:

値 > 0 の場合:

LogExample(value) # 'value' をすぐに処理する 壊す

定義を下に移動する

元の C プログラミング言語では、すべての変数定義を関数またはブロックの先頭に置く必要がありました。この要件は残念でした。多くの変数を含む長い関数の場合、たとえそれがずっと後になるまで使用されなかったとしても、読者はそれらすべての変数についてすぐに考える必要があったからです。(C99 と C++ ではこの要件が削除されました。)

次の例では、すべての変数が関数の先頭で無害に定義されています。

```
def ViewFilteredReplies(original_id):
    フィルターされた返信= []
    ルートメッセージ=Messages.objects.get(original_id) すべての返信=
    Messages.objects.select(root_id=original_id)
```

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

```
all_replies の返信の場合:
Reply.spam_votes <= MAX_SPAM_VOTES の場合:
    filtered_replies.append(返信)
```

filtered_replies を返す

このコード例の問題は、読者が一度に 3 つの変数について考え、それらの間でギアを切り替える必要があることです。

読者は後になるまでそれらすべてについて知る必要がないため、最初に使用する直前に各定義を移動するだけで簡単です。

```
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 = []
    all_replies の返信の場合:
        Reply.spam_votes <= MAX_SPAM_VOTES の場合:
            filtered_replies.append(返信)
```

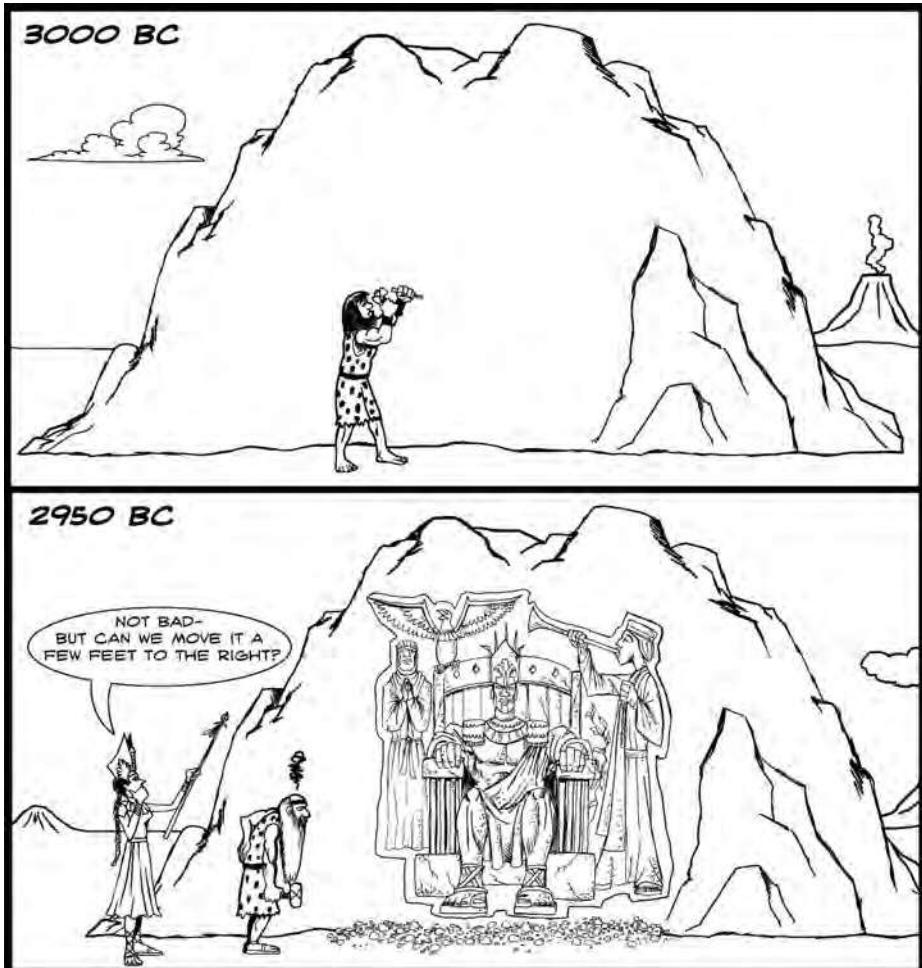
filtered_replies を返す

と疑問に思われるかもしれません。すべての返信が必要な変数であるか、次のようにして削除できるかどうか。

```
返信用Messages.objects.select(root_id=original_id):
    .
    .
    .
```

この場合、すべての返信は非常に優れた説明変数なので、これを保持することにしました。

ライトワーンス変数を優先する



この章ではこれまで、多くの変数が「実行中」であるプログラムを理解するのがいかに難しいかについて説明してきました。そうですね、常に変化する変数について考えるのはさらに困難です。それらの値を追跡することはさらに困難になります。

この問題に対処するために、少し奇妙に聞こえるかもしれない提案があります。 **ライトワーンス変数を好む。**

「永続的な固定要素」である変数は考えやすくなります。確かに、次のような定数があります。

```
静的 const int NUM_THREADS = 10;
```

読者側はあまり考える必要はありません。そして同じ理由で、定数C++では(そして最後のJavaでは)を強くお勧めします。

実際、多くの言語(PythonやJavaを含む)では、次のようないくつかの組み込み型が使用されます。弦は不変。James Gosling(Javaの作成者)が言ったように、「[Immutable]は問題がないことが多い傾向にあります。」

ただし、変数を一度だけ書き込むことができない場合でも、変数が変更される場所が少なくなれば、それでも役立ちます。

キー・アイデア

変数を操作する場所が増えるほど、その現在の値を推測するのが難しくなります。

それで、どうやってやるのですか？変数を追記型に変更するにはどうすればよいでしょうか？次の例でわかるように、多くの場合、コードを少し再構築する必要があります。

最後の例

この章の最後の例では、これまで説明してきた原則の多くを示す例を示したいと思います。

次のように配置された多数の入力テキストフィールドを持つWebページがあるとします。

```
<input type="テキスト" id="入力1" value="ダスティ  
ン"> <input type="text" id="入力2" value="Trevor">  
<input type="text" id="入力3" value=""> <input  
type="text" id="入力4" value="メリッサ">。。。
```

ご覧のとおり、IDから始まります入力1そしてそこから増加します。

あなたの仕事は、という名前の関数を書くことです。setFirstEmptyInput()これは文字列を受け取り、最初の空の<に入れます。入力>ページ上(示されている例では、「入力3」)。関数は更新されたDOM要素を返す必要があります(またはヌル空の入力が残っていない場合)。これを行うためのコードを次に示します。しませんこの章の原則を適用します。

```
var setFirstEmptyInput = function (new_value) {  
    見つかった変数 = false;  
    変数 i = 1;  
    var elem = document.getElementById('input' + i); while  
(elem !== null) {  
        if (elem.value === '') {  
            見つかった = true;  
            壊す;  
        }  
        i++;  
        elem = document.getElementById('input' + i);  
    }  
}
```

```

    if (見つかった) elem.value = new_value; 要素
    を返します。
};


```

このコードは仕事を完了させますが、見た目は良くありません。何が問題なのでしょうか?どうすれば改善できますか?

このコードを改善するにはさまざまな方法がありますが、ここでは使用する変数の観点から検討していきます。

- 変数が見つかりました
- 変数i
- 変数要素

これら3つの変数はすべて関数全体に存在し、複数回書き込まれます。それぞれの使い方を改善してみましょう。

この章の前半で説明したように、次のような中間変数見つかった多くの場合、早めに戻ることで解消できます。その改善点は次のとおりです。

```

var setFirstEmptyInput = function (new_value) {
    変数 i = 1;
    var elem = document.getElementById('input' + i); while
    (elem !== null) {
        if (elem.value === '') {
            elem.value = 新しい値; 要素
            を返します。
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    null を返します。
};


```

次に、見てください。エレム。これはコード全体で非常に「ループ状」な方法で複数回使用されており、その値を追跡するのは困難です。コードは次のように見えます要素実際には単にインクリメントしているだけの場合に、反復処理している値です。私。それでは、再構築しましょう

その間ループに入るるためにループオーバー私：

```

var setFirstEmptyInput = function (new_value) {
    for (var i = 1; true; i++) {
        var elem = document.getElementById('input' + i); if
        (elem === null)
        null を返します。// 検索に失敗しました。空の入力が見つかりません。

        if (elem.value === '') {
            elem.value = 新しい値; 要素
            を返します。
        }
    }
};


```

特に注意してください。要素ライフスパンがループ内に含まれる1回限りの変数として機能します。の用法真実としてのためにループ条件は珍しいですが、その代わりに、の定義と変更を確認できます。私一行で。(伝統的なその間(本当)それも合理的でしょう。)

まとめ

この章では、プログラム内の変数がどのようにして急速に蓄積され、追跡できなくなる可能性があるかについて説明します。変数の数を減らし、変数をできるだけ「軽量」にすることで、コードを読みやすくすることができます。具体的には：

- **変数を削除する**それは邪魔になるだけです。特に、結果を即座に処理することで「中間結果」変数を削除する方法の例をいくつか示しました。
- **各変数のスコープを縮小する**できるだけ小さくすること。各変数をコード行数が最も少ない場所に移動します。視界の外は気の外です。
- **追記型変数を推奨します**。一度だけ設定される変数(または定数、最終、または不变)コードを理解しやすくなります。

パート III

コードを再編成する

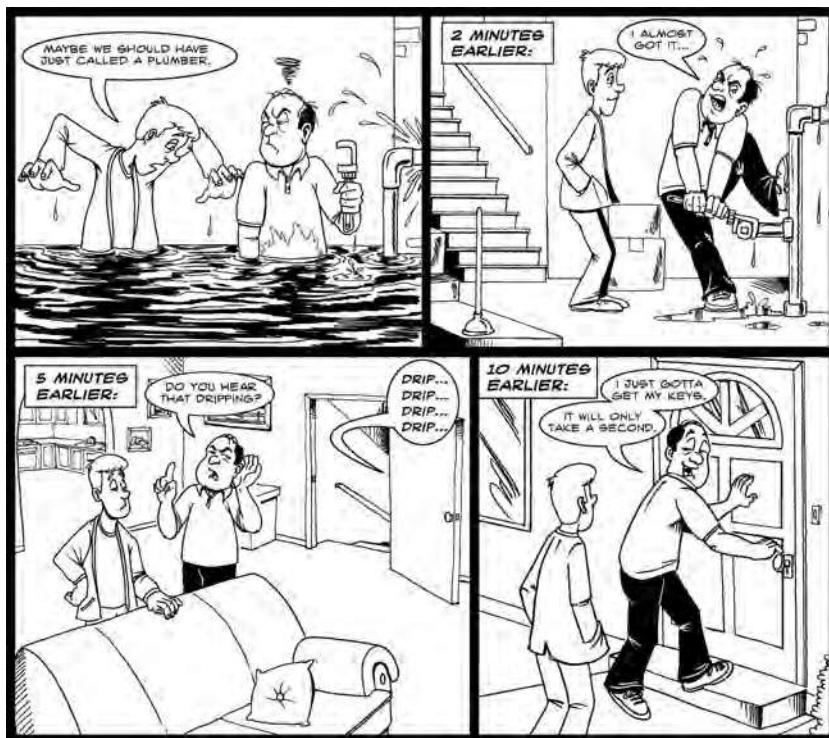
でパート II では、プログラムの「ループとロジック」を変更してコードを読みやすくする方法について説明しました。プログラムの構造を少し変更する必要があるいくつかのテクニックについて説明しました。

このパートでは、関数レベルでコードに加えることができるより大きな変更について説明します。具体的には、コードを再編成する 3 つの方法について説明します。

- プログラムの主な目的に関係のない「無関係なサブ問題」を抽出します。
- 一度に 1 つのタスクのみを実行するようにコードを再配置します。
- 最初にコードを言葉で説明し、この説明を使用して、より明確なソリューションを導き出します。

最後に、コードを完全に削除するか、最初からコードを書かないようにすることができる状況、つまりコードを理解しやすくする唯一の最善の方法について説明します。

無関係な部分問題の抽出



エンジニアリングとは、大きな問題を小さな問題に分解し、それらの問題の解決策を元に戻すことです。この原則をコードに適用すると、コードがより堅牢になり、読みやすくなります。

この章のアドバイスは、**無関係な部分問題を積極的に特定して抽出します**。意味は次のとおりです。

1. 特定の関数またはコード ブロックを見て、「このコードの高レベルの目標は何ですか?」と自問してください。
2. コードの各行について、「動作していますか?」と尋ねます。直接その目標に向けて?それとも問題を解決していますか**無関係な副問題**それを満たす必要があるのか?」
3. 十分な行数で無関係な部分問題を解決している場合は、そのコードを別の関数に抽出します。

コードを個別の関数に抽出することは、おそらく毎日行うことです。しかし、この章では、抽出の特定のケースに焦点を当てることにしました。**無関係な部分問題**、抽出されたコードは、幸いなことに、なぜそれが呼び出されているかを知りません。

ご覧のとおり、これは適用するのが簡単なテクニックですが、コードを大幅に改善できます。しかし、何らかの理由で、多くのプログラマーはこのテクニックを十分に活用していません。重要なのは、これらの無関係な部分問題を積極的に探すことです。

この章では、遭遇する可能性のあるさまざまな状況に応じてこのテクニックを説明するさまざまな例を説明します。

導入例: `findClosestLocation()`

次の JavaScript コードの大まかな目標は次のとおりです。**指定された点に最も近い場所を見つける**(斜体で示した高度なジオメトリにとらわれないでください)。

```
// 'array' のどの要素が指定された緯度/経度に最も近いかを返します。// 地球を完全な球体としてモデル化します。
var findClosestLocation = 関数 (緯度、経度、配列) {
  最も近い変数;
  varmost_dist = Number.MAX_VALUE; for (var i = 0;
  i < array.length; i += 1) {
    // 両方の点をラジアンに変換します。
    var lat_rad = ラジアン(緯度);
    var lng_rad = ラジアン(経度);
    var lat2_rad = ラジアン(array[i].latitude); var
    lng2_rad = ラジアン(配列[i].経度);

    // 「余弦の球面法則」公式を使用します。
    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 (距離 < 最も近い距離) {
      最も近い = 配列[i];
```

```

        最も近い距離 = 距離;
    }
}

最も近いものを返します。
};

```

ループ内のコードのほとんどは、無関係なサブ問題に取り組んでいます。2つの緯度/経度点間の球面距離を計算します。そのコードは非常に多くあるため、別のファイルに抽出するのが合理的です。球状距離()関数：

```

変数球状距離=関数 (lat1, lng1, lat2, lng2) {
    var lat1_rad = ラジアン(lat1); var
    lng1_rad = ラジアン(lng1); var
    lat2_rad = ラジアン(lat2); var
    lng2_rad = ラジアン(lng2);

    // 「余弦の球面法則」公式を使用します。
    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));
};

```

残りのコードは次のようにになります。

```

var findClosestLocation = 関数 (緯度、経度、配列) {
    最も近い変数;
    varmost_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 (距離 < 最
        も近い距離) {
            最も近い = 配列[i];
            最も近い距離 = 距離;
        }
    }
    最も近いものを返します。
};

```

このコードは、読者が複雑な幾何学方程式に気を取られることなく、高レベルの目標に集中できるため、はるかに読みやすくなっています。

追加のボーナスとして、球状距離()単独でテストする方が簡単になります。そして球状距離()将来的に再利用できるタイプの関数です。これが、これが「無関係な」副問題である理由です。完全に自己完結型であり、アプリケーションがそれをどのように使用しているかを認識しません。

純粋なユーティリティコード

文字列の操作、ハッシュテーブルの使用、ファイルの読み取り/書き込みなど、ほとんどのプログラムで実行される基本的なタスクのコアセットがあります。

多くの場合、これらの「基本ユーティリティ」は、プログラミング言語の組み込みライブラリによって実装されます。たとえば、ファイルの内容全体を読み取りたい場合、PHP では次のように呼び出すことができます。

```
file_get_contents("ファイル名"), または Python では、次のことができます open("ファイル名").read()。
```

しかし、場合によっては自分でギャップを埋めなければならないこともあります。たとえば、C++ では、ファイル全体を読み取る簡潔な方法はありません。代わりに、必然的に次のようなコードを書くことになります。

```
ifstream ファイル(ファイル名);  
  
// ファイルのサイズを計算し、そのサイズのバッファを割り当てます。  
file.seekg(0, ios::end);  
const int file_size = file.tellg(); char* file_buf  
= 新しい文字 [file_size];  
  
// ファイル全体をバッファに読み込みます。  
file.seekg(0, ios::beg);  
file.read(file_buf, file_size);  
file.close();  
  
. . .
```

これは、次のような新しい関数に抽出する必要がある、無関係な部分問題の典型的な例です。ReadFileToString()。これで、コードベースの残りの部分は C++ のように動作できるようになります。した持っています ReadFileToString() 関数。

一般に、「うちの図書館にこんな本があればいいのに」と思っているなら、XYZ()関数」を書いてみましょう。(まだ存在しないものとします。) 時間が経つにつれて、プロジェクト間で使用できる優れたユーティリティコードのコレクションが構築されます。

その他の汎用コード

JavaScript をデバッグするとき、プログラマーはよく使用します。アラート()プログラマに何らかの情報を表示するメッセージボックスをポップアップする、Web バージョンの「printf()」デバッグ中。」たとえば、次の関数呼び出しへ、Ajax を使用してサーバーにデータを送信し、サーバーから返された辞書を表示します。

```
ajax_post({  
  URL: 'http://example.com/submit'、データ: データ、  
  on_success: 関数(応答データ) {  
    var str = "{}\n";  
    for (response_data の var キー) {  
      str += " " + キー + " = " + 応答データ[キー] + "\n";  
    }  
    アラート(str + "}");  
  
    // 'response_data' の処理を 続行します ...  
  }  
});
```

このコードの大まかな目標は次のとおりです。サーバーに対してAjax呼び出しを行い、応答を処理します。しかし、コードの多くは無関係な副問題を解決しています。辞書をきれいに印刷します。そのコードを次のような関数に抽出するのは簡単です。format_pretty(obj):

```
var format_pretty = 関数 (obj) {
  var str = "{\n"; for (obj
  の var キー) {
    str += " " + key + " = " + obj[key] + "\n";
  }
  str + "}";
};

str + "}";
} を返します。
```

予期せぬメリット

抽出する理由はたくさんありますformat_pretty()は良いアイデアです。呼び出し側のコードが簡素化され、format_pretty()あると便利な機能です。

しかし、それほど明白ではない別の大きな理由があります。改善するのは簡単ですformat_pretty()コードが単独の場合。小さな機能を個別に開発していると、機能の追加、信頼性の向上、エッジケースへの対応などが簡単に感じられます。

いくつかのケースをご紹介しますformat_pretty(obj)扱いません:

- 期待するオブジェクトオブジェクトになること。代わりにブレーン文字列(または未定義)、現在のコードは例外をスローします。
- 各値を期待します。オブジェクトシンプルなタイプになります。代わりにネストされたオブジェクトが含まれている場合、現在のコードではそれらが[オブジェクト オブジェクト]、それはあまり美しくありません。

別れる前にformat_pretty()独自の機能に組み込むと、これらすべての改善を行うのは大変な作業のように感じられたでしょう。(実際、ネストされたオブジェクトを再帰的に出力することは、別の関数がなければ非常に困難です。)

しかし今では、この機能を追加するのは簡単です。改善されたコードは次のようにになります。

```
変数フォーマット_プリティ=関数 (obj, インデント){
  // null、未定義、文字列、非オブジェクトを処理します。if
  (obj === null) は「null」を返します。
  if (obj === 未定義) は「未定義」を返します。
  if (typeof obj === "string") return "" + obj + ""; if (typeof obj !
  == "object") return String(obj);

  if (インデント === 未定義) インデント = "";

  // (null 以外の) オブジェクトを処理しま
  す。var str = "{\n";
  for (obj の var キー) {
    str += インデント + " " + キー + " = "; str += フォー
    メット_プリティ(obj[キー], インデント + " " );
  }
  str + インデント + "}";
};

str + "}";
} を返します。
```

これにより、前述の欠点がカバーされ、次のような出力が生成されます。

```
{  
    キー1=1  
    key2 = true  
    key3 = 未定義  
    キー4 = null  
    キー5 = {  
        key5a = {  
            key5a1 = 「こんにちは」  
        }  
    }  
}
```

汎用コードを大量に作成する

機能ReadFileToString()そしてformat_pretty()これらは、無関係な部分問題の良い例です。これらは非常に基本的で広く適用できるため、プロジェクト全体で再利用できる可能性があります。コードベースには、多くの場合、このようなコード用の特別なディレクトリがあります(例:ユーティリティ/)簡単に共有できるようにします。

汎用コードは優れています。プロジェクトの残りの部分から完全に切り離されています。このようなコードは、開発、テスト、および理解が容易です。すべてのコードがこのようになればいいのに!

SQLデータベース、JavaScriptライブラリ、HTMLテンプレートシステムなど、使用する多くの強力なライブラリやシステムについて考えてみましょう。これらのコードベースはプロジェクトから完全に分離されているため、内部について心配する必要はありません。その結果、プロジェクトのコードベースは小さいままになります。

プロジェクトのより多くの部分を独立したライブラリとして分離できれば、コードの残りの部分が小さくなり、考えやすくなるため、より良い結果が得られます。

これはトップダウンプログラミングですか、それともボトムアッププログラミングですか?

トップダウンプログラミングは、最上位のモジュールと関数が最初に設計され、それらをサポートするために必要に応じて下位レベルの関数が実装されるスタイルです。

ボトムアッププログラミングでは、最初にすべてのサブ問題を予測して解決し、次にこれらの部分を使用して上位レベルのコンポーネントを構築しようとします。

この章は、どちらかの方法を推奨するものではありません。ほとんどのプログラミングには、両方の組み合せが含まれます。重要なのは最終結果です。部分的な問題は削除され、個別に対処されます。

プロジェクト固有の機能

理想的には、抽出する部分問題はプロジェクトに完全に依存しないものになります。しかし、そうでなくとも大丈夫です。サブ問題を解消することは依然として驚異的な効果を発揮します。

これはビジネス レビュー ウェブサイトの例です。この Python コードは、新しい

仕事オブジェクトを作成し、そのオブジェクトを設定します名前、URL、そして作成日:

```
ビジネス = ビジネス()
仕事。名前=request.POST["名前"]

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/パス名) URL/パス名 = URL/パス名.strip("-")
```

仕事。URL= "/biz/" + URL/パス名

```
仕事。作成日=datetime.datetime.utcnow()
business.save_to_database()
```

のURLの「クリーン」バージョンであるはずです名前。たとえば、名前は「ACジョーズ」
タイヤ&スモッグ社」URL「/」になりますbiz/ac-joes-tire-smog-inc」。

このコード内の無関係な副問題は次のとおりです。名前を有効な URL に変換します。このコードは非常に簡単に抽出できます。この作業中に、正規表現をプリコンパイルすることもできます(そして、読みやすい名前を付けます)。

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

```
確かにメイク_url_フレンドリー (文章) :
    テキスト = text.lower()
    text = CHARS_TO_REMOVE.sub("", text)
    text = CHARS_TO_DASH.sub('-', text)
    return text.strip("-")
```

元のコードには、より「規則的な」パターンが含まれています。

```
ビジネス = ビジネス()
仕事。名前=request.POST["名前"]
仕事。URL= "/biz/" +メイク_url_フレンドリー(business.name)
ビジネス。作成日=datetime.datetime.utcnow()
business.save_to_database()
```

このコードは、正規表現や複雑な文字列操作に気をとられることがないため、読むのに必要な労力がはるかに少なくなります。

コードをどこに置くべきかmake_url_フレンドリー()?かなり一般的な関数のように見えるので、別個に置くのが合理的かもしれませんユーティリティ/ディレクトリ。一方、これらの正規表現は米国の企業名を念頭に置いて設計されているため、おそらくコードは使用されるのと同じファイル内に留まるはずです。実際にはそれほど重要ではなく、定義は後で簡単に移動できます。さらに重要なことは、make_url_フレンドリー()全然抽出されてた。

既存のインターフェースの簡素化

ライブラリがクリーンなインターフェイスを提供することは、誰もが気に入ります。つまり、引数が少なく、多くのセットアップが必要なく、通常は使用するのにはほとんど労力を必要としないインターフェイスです。これにより、コードがエレガントに見え、シンプルかつ強力になります。

ただし、使用しているインターフェイスがクリーンでない場合でも、クリーンな独自の「ラッパー」関数を作成できます。

たとえば、JavaScript でブラウザの Cookie を処理することは理想とは程遠いです。概念的には、Cookie は名前と値のペアのセットです。しかし、ブラウザが提供するインターフェイスには、単一のドキュメント.クッキー構文は次のとおりです。

名前1=値1; 名前2=値2; ...

必要な Cookie を見つけるには、この巨大な文字列を自分で解析する必要があります。以下は、「」という名前の Cookie の値を読み取るコードの例です。最大結果":

```
var max_results;
var cookies = document.cookie.split(''); for (var i
= 0; i < cookies.length; i++) {
    var c = クッキー[i];
    c = c.replace(/^\s+/,""); // 先頭のスペースを削除 if
    (c.indexOf("max_results=") === 0)
    max_results = Number(c.substring(12, c.length));
}
```

うわー、見にくいコードですね。明らかに、`get_cookie()`関数が作成されるのを待っているので、次のように書くだけです。

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

Cookie の値を作成または変更することはさらに奇妙なことです。設定する必要がありますドキュメント.クッキー正確な構文で値に変換します。

```
document.cookie = "max_results=50; 期限切れ=2020年1月1日水曜日 20:53:47 UTC; パス=/";
```

このステートメントは他の既存の Cookie をすべて上書きするように見えますが、(魔法のように) そうではありません。

Cookie を設定するためのより理想的なインターフェイスは次のようなものになります。

```
set_cookie(名前、値、有効期限までの日数);
```

Cookie の消去も直感的ではありません。Cookie の有効期限が過去に切れるように設定する必要があります。代わりに、理想的なインターフェイスは次のようになります。

```
クッキーの削除 (名前);
```

ここでの教訓は、**理想的とは言えないインターフェイスに妥協する必要はありません**。独自のラッパー関数をいつでも作成して、行き詰まっているインターフェイスの見苦しい詳細を隠すことができます。

ニーズに合わせてインターフェイスを再構築する

プログラム内のコードの多くは、関数への入力の設定や出力の後処理など、他のコードをサポートするためだけに存在します。この「接着」コードは、多くの場合、プログラムの実際のロジックとは何の関係もありません。このような平凡なコードは、個別の関数に抽出するのに最適な候補です。

たとえば、次のような機密性の高いユーザー情報を含む Python 辞書があるとします。

```
{"ユーザー名": "...", "パスワード": "..."}  
そして、そのすべての情報を URL に入れる必要があります。機密性が高いため、最初に辞書を暗号化することにします。暗号クラス。
```

しかし暗号入力として辞書ではなくバイト文字列を期待します。そして暗号バイト文字列を返しますが、URL セーフなものが必要があります。暗号また、多くの追加パラメータも必要となるため、使用するのがかなり面倒です。

単純なタスクとして始まったものが、大量の接着コードに変わります。

```
user_info = {"ユーザー名": "...", "パスワード": "..."}  
user_str = json.dumps(user_info)  
暗号 = 暗号("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)  
encrypted_bytes = 暗号.更新(user_str)  
暗号化バイト += 暗号.最終() # 現在の 128 ビット ブロックをフラッシュします  
URL = "http://example.com/?user_info=" +  
Base64.urlsafe_b64encode(encrypted_bytes)。。。
```

私たちが取り組んでいる問題はユーザーの情報を URL に暗号化します。このコードの大部分は単に実行しているだけですこの Python オブジェクトを URL に適した文字列に暗号化します。その部分問題を抽出するのは簡単です。

```
確かにurl_safe_encrypt(オブジェクト):  
    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します  
    return base64.urlsafe_b64encode(encrypted_bytes)
```

次に、結果として得られるコードを実行します。本物プログラムのロジックは単純です。

```
user_info = {"ユーザー名": "...", "パスワード": "..."}  
URL = "http://example.com/?user_info=" + url_safe_encrypt(ユーザー情報)
```

物事を行き過ぎてしまう

この章の冒頭で述べたように、私たちの目標は次のとおりです。積極的に無関係な部分問題を特定して抽出します。」「積極的に」と言ったのは、ほとんどのプログラマーが十分に積極的ではないためです。しかし、興奮しすぎて物事を行き過ぎてしまう可能性があります。

たとえば、前のセクションのコードは、次のようにさらに細分化することもできます。

```
user_info = { "ユーザー名": "...", "パスワード": "..." }
URL = "http://example.com/?user_info=" +url_safe_encrypt_obj (ユーザー情報)

確かにurl_safe_encrypt_obj(オブジェクト):
    obj_str = json.dumps(obj) 戻り値
    url_safe_encrypt_str(obj_str)

確かにurl_safe_encrypt_str (データ) :
    暗号化バイト =暗号化する (データ)
    base64.urlsafe_b64encode(encrypted_bytes)を返します

確かに暗号化する (データ) :
    暗号 =make_cipher() encrypted_bytes =
        cipher.update(データ)
        encrypted_bytes += cipher.final() # 残りのバイトをフラッシュします return
        encrypted_bytes

確かにmake_cipher():
    return Cipher("aes_128_cbc"、key=PRIVATE_KEY、init_vector=INIT_VECTOR、op=ENCODE)
```

これらの小さな関数をすべて導入すると、実際には可読性が損なわれます。読者が追跡しなければならないことがさらに増え、実行パスをたどるには飛び回らなければならないからです。

コードに新しい関数を追加すると、読みやすさに多少の(しかし明らかな)コストがかかります。前のケースでは、このコストを相殺するために何も得られません。プロジェクトの他の部分で必要な場合は、これらの小さな関数を追加するのが合理的かもしれません。しかし、それまではその必要はありません。

まとめ

この章について簡単に考えると、次のようにになります。**汎用コードをプロジェクト固有のコードから分離します。**結局のところ、ほとんどのコードは汎用です。一般的な問題を解決するためにライブラリとヘルパー関数の大規模なセットを構築すると、プログラムを独自のものにする小さなコアが残ります。

この手法が役立つ主な理由は、プログラマーがプロジェクトの他の部分から切り離された、より小さく明確に定義された問題に集中できるためです。結果として、これらの副問題に対する解決策はより徹底的で正確になる傾向があります。後で再利用できる場合もあります。

参考文献

マーティン・ファウラーのリファクタリング:既存のコードの設計を改善する(Fowler et al.、Addison-Wesley Professional, 1999)では、リファクタリングの「抽出メソッド」について説明し、コードをリファクタリングする他の多くの方法をカタログ化しています。

ケント・ベックス*Smalltalk* のベスト プラクティス パターン(Prentice Hall, 1996)では、コードを多数の小さな関数に分割するための多数の原則をリストした「合成メソッド パターン」について説明しています。特に、原則の1つは、「單一メソッド内のすべての操作を同じ抽象レベルに保つ」です。

これらのアイデアは、「無関係な部分問題を抽出する」というアドバイスに似ています。この章で説明したのは、メソッドを抽出する場合の単純かつ特殊なケースです。

一度に1つのタスク

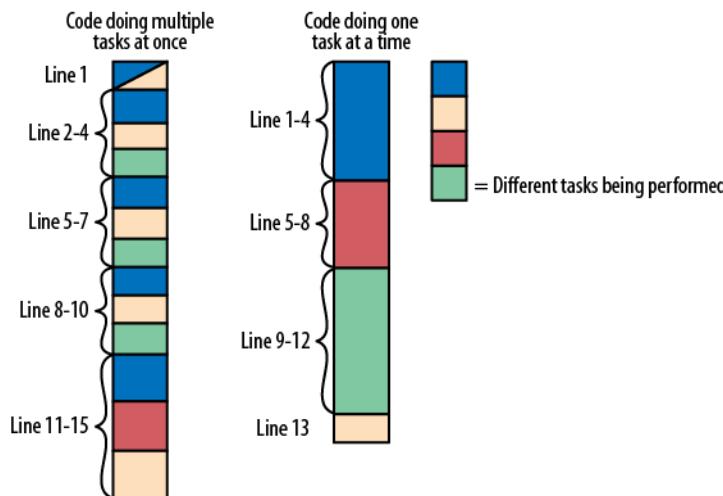


複数のことを同時に実行するコードは理解しにくくなります。単一のコード ブロックで、新しいオブジェクトの初期化、データのクレンジング、入力の解析、ビジネス ロジックの適用をすべて同時に実行することができます。すべてのコードが織り込まれていると、それぞれの「タスク」が単独で開始され完了する場合よりも理解しにくくなります。

キーアイデア

コードは一度に1つのタスクのみを実行するように編成する必要があります。

別の言い方をすれば、この章はコードの「デフラグ」についてです。次の図は、このプロセスを示しています。左側は、コードの一部が実行しているさまざまなタスクを示し、右側は、一度に1つのタスクを実行するように編成された後の同じコードを示しています。



「関数は1つのことだけを行うべきである」というアドバイスを聞いたことがあるかもしれません。私たちのアドバイスも同様ですが、必ずしも関数の境界に関するものではありません。確かに、大きな関数を複数の小さな関数に分割することは良い場合があります。ただし、これを行わなくとも、その大きな関数内のコードを編成することができるため、独立した論理セクションがあるように感じられます。

コードに「一度に1つのタスク」を実行させるために使用するプロセスは次のとおりです。

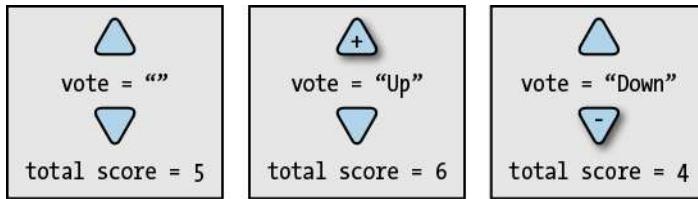
1. コードが実行しているすべての「タスク」をリストします。私たちは「タスク」という言葉を非常に大まかに使用します。「このオブジェクトが有効であることを確認する」というような小さなことであれば、「ツリー内のすべてのノードを反復処理する」というような漠然としたものにすることもできます。
2. これらのタスクをできる限り異なる関数、または少なくともコードの異なるセクションに分割するようにしてください。

この章では、これを行う方法の例をいくつか示します。

タスクは小さくてもよい

ブログに、ユーザーがコメントの「賛成」または「反対」に投票できる投票ウィジェットがあるとします。合計スコアコメントの合計はすべての投票の合計です。「賛成」投票ごとに+1、「反対」投票ごとに-1です。

ユーザーの投票が取り得る3つの状態と、それが投票総数にどのような影響を与えるかは次のとおりです。スコア：



ユーザーがいずれかのボタンをクリックすると(投票を行う/変更するため)、次のJavaScriptが呼び出されます。

```
投票変更(古い投票, 新しい投票); // 各投票は「賛成」、「反対」、「」のいずれかです
```

この関数は合計を更新しますスコアのすべての組み合わせで機能します古い投票/新しい投票:

```
var vote_changed = 関数 (old_vote, new_vote) {
    var スコア = get_score();

    if (new_vote !== old_vote) {
        if (new_vote === '上') {
            スコア += (old_vote === '下' ? 2 : 1); } else if
        (new_vote === 'Down') {
            スコア -= (old_vote === '上' ? 2 : 1); } else if
        (new_vote === '') {
            スコア += (old_vote === '上' ? -1 : 1);
        }
    }

    set_score(スコア);
};
```

コードはかなり短いですが、多くのことを実行します。多くの複雑な詳細があり、オフバイワンエラー、タイプミス、その他のバグがあるかどうかを一目で判断するのは困難です。

このコードは1つのこと(スコアの更新)だけを実行しているように見えますが、実際には次のことが行われます。二同時に実行されるタスク:

- 1.古い投票そして新しい投票数値に「解析」されます。
- 2.スコア更新中です。

各タスクを個別に解決することで、コードを読みやすくすることができます。次のコードは、投票を数値に解析するという最初のタスクを解決します。

```
var vote_value = 関数 (投票) {
    if (投票 === '賛成') {
```

```

        +1 を返します。
    }
    if (投票 === '反対') {
        -1 を返します。
    }
    0を返します。
};


```

これで、コードの残りの部分で 2 番目のタスクを解決できるようになります。スコア：

```

var vote_changed = 関数 (old_vote, new_vote) {
    var スコア = get_score();

    スコア -= 投票値(古い投票); スコア // 古い投票を削除 // 新し
    += 投票値(新しい投票);           い投票を追加

    set_score(スコア);
};


```

ご覧のとおり、このバージョンのコードは、それが機能することを確信するのに必要な精神的な労力がはるかに少なくなります。これはコードを「理解しやすい」ものにする大きな部分です。

オブジェクトからの値の抽出

かつて、ユーザーの位置情報をわかりやすい文字列にフォーマットする JavaScript がありました。「都市、田舎」「アメリカ、サンタモニカ」や「フランス、パリ」など。私たちに与えられたのは、位置情報構造化された情報が豊富に含まれる辞書。私たちがしなければならなかったのは、「市」と「国」をすべてのフィールドから抽出し、それらを連結します。

次の図は、入力/出力の例を示しています。

`location_info`

LocalityName	"Santa Monica"
SubAdministrativeAreaName	"Los Angeles"
AdministrativeAreaName	"California"
CountryName	"USA"



"Santa Monica, USA"

ここまで簡単そうに見えますが、難しいのは、これら 4 つの値のいずれかまたはすべてが欠落している可能性があります。これに私たちがどのように対処したかは次のとおりです。

- 「City」を選択するときは、可能な場合は「LocalityName」（都市/町）を使用し、次に「SubAdministrativeAreaName」（より大きな都市/郡）、次に「AdministrativeAreaName」（州/準州）を使用することを好みました。

- ・3つすべてが欠けている場合、「City」には愛情を込めてデフォルトの「Middle-of-Nowhere」が与えられました。
- ・「CountryName」が欠落している場合は、「Planet Earth」がデフォルトとして使用されます。

次の図は、欠損値を処理する2つの例を示しています。

location_info

LocalityName	(undefined)
SubAdministrativeAreaName	(undefined)
AdministrativeAreaName	(undefined)
CountryName	"Canada"



"Middle-of-Nowhere, Canada"

location_info

LocalityName	(undefined)
SubAdministrativeAreaName	"Washington, DC"
AdministrativeAreaName	(undefined)
CountryName	"USA"



"Washington, DC, USA"

このタスクを実装するために作成したコードは次のとおりです。

```
var place = location_info["地域名"]; // 例: 「サンタモニカ」 if (!place) {
    place = location_info["サブ行政エリア名"]; // 例: 「ロサンゼルス」
}
if (!place) {
    place = location_info["行政区域名"]; // 例: 「カリフォルニア」
}
if (!place) {
    place = "辺鄙な場所";
}
if (location_info["国名"]){
    place += ", " + location_info["国名"]; // 例: "アメリカ" } else {
    place += "、惑星地球";
}
```

返却場所。

確かに、少し面倒ですが、仕事は完了しました。

しかし、数日後、機能を改善する必要がありました。米国内の場所については、州（可能であれば）国の代わりに。したがって、「米国サンタモニカ」の代わりに「カリフォルニア州サンタモニカ」が返されます。

この機能を前のコードに追加すると、コードはさらに見苦しくなるでしょう。

「一度に1つのタスク」を適用する

このコードを思いどおりに曲げるのではなく、立ち止まって、コードがすでに複数のタスクを同時に実行していることに気づきました。

1. 辞書から値を抽出する位置情報
2. 「都市」の優先順位を調べ、何も見つからなかった場合はデフォルトで「人里離れた場所」を選択します。
3. 「国」を取得し、存在しない場合は「プラネットアース」を使用する
4. アップデート場所

そこで、代わりに、これらのタスクをそれぞれ独立して解決できるように元のコードを書き直しました。

最初のタスク(値を抽出する)位置情報)単独で簡単に解決できました。

```
変数町 = location_info["地域名"]; // 例: "サンタモニカ" // 例:
変数市 = location_info["サブ管理エリア名"]; = "ロサンゼルス" // 例: "CA"
変数州 = location_info["行政区域名"];
変数国=location_info["国名"]; // 例: "アメリカ"
```

この時点で、使用は完了しました位置情報長くて直感的でないキーを覚える必要もありませんでした。代わりに、操作する4つの単純な変数がありました。

次に、戻り値の「後半」が何になるかを把握する必要がありました。

```
// デフォルトから始めて、最も具体的な値で上書きし続けます。変数後半= "地球"; if (国) {
    後半=国;
}
if (州 && 国 === "アメリカ") {
    後半=州;
}
```

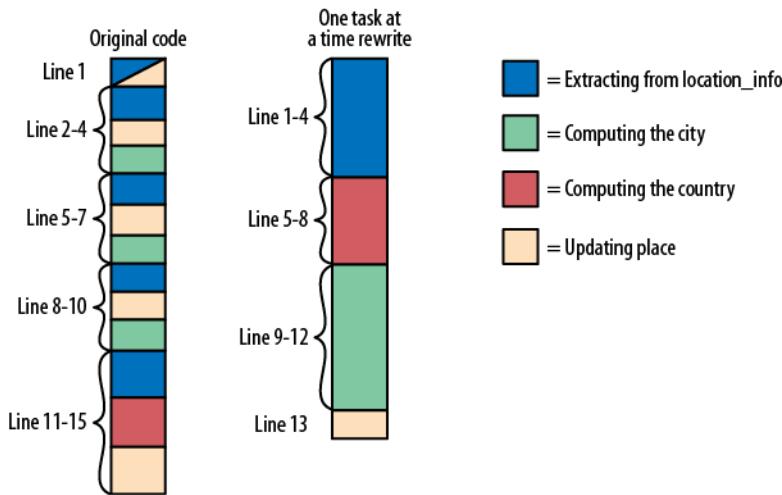
同様に、「前半」を把握できます。

```
変数前半= "辺鄙な場所"; if (州 && 国 !== "米
国") {
    前半=州;
}
if (都市) {
    前半=市;
}
if (町) {
    前半=町;
}
```

最後に、情報をまとめました。

```
戻る前半+ "、 " +後半;
```

この章の冒頭にある「最適化」の図は、実際には元のソリューションとこの新しいバージョンを表したものです。同じ図にさらに詳しい情報を記入したものを次に示します。



ご覧のとおり、2番目のソリューションの4つのタスクは個別の領域に最適化されています。

別のアプローチ

コードをリファクタリングする場合、複数の方法があることがよくあります。このケースも例外ではありません。いくつかのタスクを分離すると、コードを考えるのが容易になります。コードをリファクタリングするためのさらに良い方法を思いつくかもしれません。

たとえば、以前のシリーズでは、もしそうの場合は正しく機能するかどうかを確認するには、コメントを注意深く読む必要があります。実際、そのコードでは2つのサブタスクが同時に実行されています。

1. 変数のリストを確認し、使用可能な中で最も好みしいものを選択します。
2. 国が「米国」かどうかに応じて、異なるリストを使用します。

振り返ってみると、以前のコードには「if USA」ロジックが残りのロジックと織り込まれていることがわかります。代わりに、米国のケースと米国以外のケースを個別に処理できます。

```

var 前半、後半;

if(国 === "アメリカ") {
    前半 = 町 || 市 || "人里離れた"; 後半 = 状態 || "アメリカ合衆
    国";
} それ以外{
    前半 = 町 || 市 || 状態 || "人里離れた"; 後半 = 国 || "地球";
}

前半 + "," + 後半を返します。

```

JavaScriptに詳しくない場合は、`|| b || c`は慣用的であり、最初の「真実の」値（この場合は、定義された空でない文字列）に評価されます。このコードには非常に優れた利点があります。

設定リストを簡単に調べて更新できます。ほとんどもコメントが削除され、ビジネスロジックがより少ないコード行で表現されています。

より大きな例

私たちが構築した Web クローリング システムでは、UpdateCounts()各 Web ページがダウンロードされた後、さまざまな統計を増分するために呼び出されます。

```
void UpdateCounts(HttpDownload hd) {
    counts["終了状態"][[hd.exit_state()]]++;
    // 例: "成功" または "失敗" // 例:
    counts["HTTP 応答"][[hd.http_response()]]++;
    // "404 NOT FOUND"
    counts["コンテンツタイプ"][[hd.content_type()]]++;
    // 例: "text/html"
}
```

そうですね、私たちはこうして頑張ったコードは見えました！

実際には、httpダウンロードオブジェクトには、ここで示されているメソッドがありませんでした。その代わり、

httpダウンロードこれは非常に大規模で複雑なクラスであり、多くの入れ子になったクラスがあったため、それらの値を自分で見つけ出す必要がありました。さらに悪いことに、これらの値が完全に欠落している場合がありました。その場合は、単に「」を使用しました。未知"デフォルト値として。

これらすべてのせいで、実際のコードは非常に混乱したものになりました。

```
// 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) {
    // 利用可能な場合は、終了状態を計算します。
    if (!hd.has_event_log() || !hd.event_log().has_exit_state()) {
        counts["終了状態"]["不明"]++;
    } それ
    以外{
        string state_str = ExitStateTypeName(hd.event_log().exit_state());
        counts["終了状態"][[state_str]]++;
    }

    // HTTP ヘッダーがまったくない場合は、残りの要素に「unknown」を使用します。if (!
    hd.has_http_headers()){
        counts["HTTP Response"]["unknown"]++;
        counts["Content-Type"]["unknown"]++;
    } 戻
    る;
}

HttpHeaders ヘッダー = hd.http_headers();

// 既知の場合は HTTP 応答をログに記録し、そうでない場合は「不明」をログに記録し
ます if (!headers.has_response_code()) {
    counts["HTTP Response"]["unknown"]++;
} そ
れ以外{
    文字列コード = StringPrintf("%d", headers.response_code());
    counts["HTTP 応答"][[コード]]++;
}

// 既知の場合は Content-Type をログに記録し、そうでない場合は「unknown」をログに記
録します if (!headers.has_content_type()){
    counts["Content-Type"]["unknown"]++;
}
```

```

} それ以外{
    文字列 content_type = ContentTypeMime(headers.content_type());
    counts["Content-Type"][[content_type]]++;
}
}

```

ご覧のとおり、多くのコードとロジックがあり、コード行が数行繰り返されている場合もあります。このコードは読んでいて面白くありません。

特に、このコードは異なるタスク間を行き来します。コード全体に組み込まれたさまざまなタスクを次に示します。

1. 「」を使用する未知"各キーのデフォルト値として
2. のメンバーかどうかの検出httpダウンロード行方不明です
3. 値を抽出して文字列に変換する
4. アップデートカウント[]

これらのタスクの一部をコード内の個別の領域に分割することで、コードを改善できます。

```

void UpdateCounts(HttpDownload hd) {
    // タスク: 抽出する各値のデフォルト値を定義します 文字列 exit_state = "不明"; 文字
    列 http_response = "不明"; 文字列コンテンツタイプ = "不明";
}

```

```

// タスク: HttpDownload から各値を 1 つずつ抽出してみます 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: カウントを更新[] counts["終了状態"]
[exit_state]++;
counts["HTTP レスポンス"]
[http_response]++;
counts["Content-Type"]
[content_type]++;
}

```

ご覧のとおり、コードには次の目的を持った 3 つの個別の領域があります。

1. 対象となる 3 つのキーのデフォルトを定義します。
2. 利用可能な場合は、これらの各キーの値を抽出し、文字列に変換します。
3. アップデートカウント[]キー/値ごとに。

これらの地域の良いところは、孤立した相互に参照することができます。あるリージョンを読んでいる間、他のリージョンについて考える必要はありません。

4つのタスクをリストしましたが、そのうち3つだけを分離できたことに注意してください。それはまったく問題ありません。最初にリストしたタスクは出発点にすぎません。別れてもいくつかのここでそうであったように、そのうちのいくつかは物事を大いに助けることができます。

さらなる改善

この新しいバージョンのコードは、元の怪物から大幅に改善されています。そして、このクリーンアップを実行するために他の関数を作成する必要さえなかったことに注目してください。前に述べたように、「一度に1つのタスク」という考え方には、関数の境界に関係なくコードをクリーンアップするのに役立ちます。

ただし、次の3つのヘルパー関数を導入することで、このコードを別の方法で改善することもできました。

```
void UpdateCounts(HttpDownload hd) {
    counts["終了状態"][[終了状態(HD)]]++; counts["HTTP応答"][[HTTPレスポンス(HD)]]++; counts["コンテンツタイプ"][[コンテンツタイプ(HD)]]++;
}
```

これらの関数は、対応する値を抽出するか、「不明」を返します。例えば：

```
弦終了状態(httpダウンロードhd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state()); } それ
    以外 {
        「不明」を返します。
    }
}
```

この代替ソリューションでは変数も定義されていないことに注意してください。で述べたように、[第9章 変数と可読性](#)、中間結果を保持する変数は、多くの場合完全に削除できます。

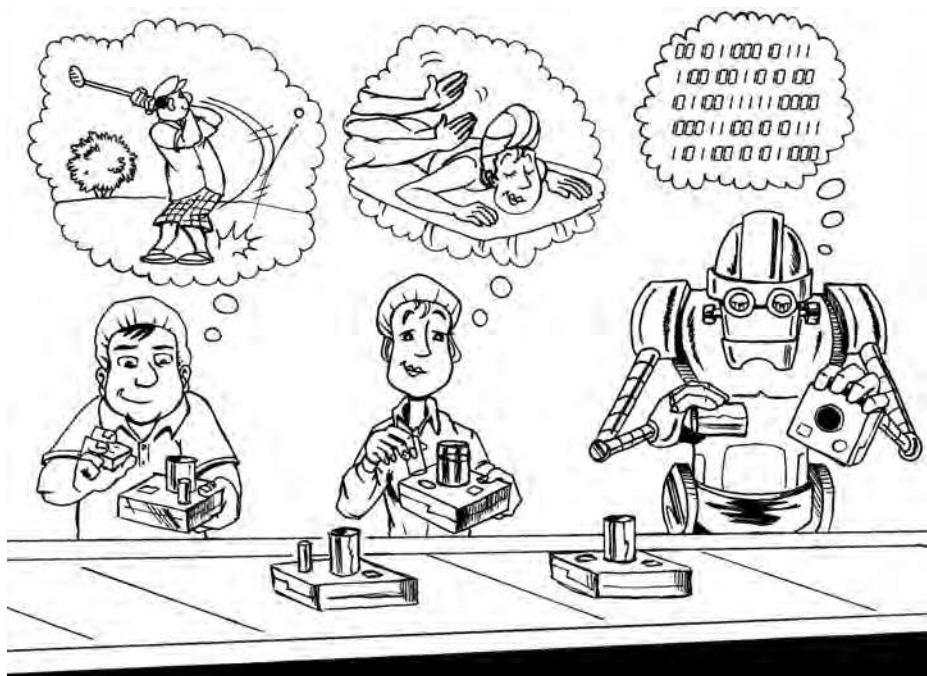
このソリューションでは、問題を別の方向に「スライス」しただけです。どちらのソリューションも、読者が一度に1つのタスクについてのみ考える必要があるため、非常に読みやすくなっています。

まとめ

この章では、コードを整理するための簡単なテクニックを説明します。**一度に1つのタスクだけを実行する。**

コードが読みにくい場合は、コードが実行しているタスクをすべてリストしてみてください。これらのタスクの一部は、簡単に別個の関数（またはクラス）になる可能性があります。他のものは、単一の関数内の論理的な「段落」になる可能性があります。これらのタスクをどのように分離するかの正確な詳細は、それらが分離されているという事実ほど重要ではありません。難しいのは、プログラムが行っている小さな動作をすべて正確に記述することです。

思考をコードに変える



おばあちゃんに説明できない限り、何かを本当に理解することはできません。

- アルバート・aigneshwan

複雑なアイデアを誰かに説明するとき、細かい部分のせいで混乱してしまいがちです。自分よりも知識のない人でも理解できるように、アイデアを「平易な英語で」説明できることは貴重なスキルです。アイデアを最も重要な概念にまで絞り込む必要があります。こうすることで、相手の理解が深まるだけでなく、自分の考えをより明確に考えることができます。

読者にコードを「提示」するときも、同じスキルを使用する必要があります。私たちは、ソースコードがプログラムの動作を説明する主な方法であると考えています。したがって、コードは「平易な英語で」記述する必要があります。

この章では、コードをより明確に記述するのに役立つ簡単なプロセスを使用します。

1. 同僚に話すときと同じように、コードが何をする必要があるかを平易な英語で説明します。
2. この説明で使用されているキーワードやフレーズに注意してください。
3. この説明に一致するコードを作成します。

ロジックを明確に説明する

以下は、PHPのWebページのコードのスニペットです。このコードは、保護されたページの上部にあります。ユーザーがページを表示する権限を持っているかどうかを確認し、そうでない場合は、権限がないことをユーザーに伝えるページをすぐに返します。

```
$is_admin = is_admin_request(); if  
($document) {  
    if (!$is_admin && ($document['ユーザー名'] != $_SESSION['ユーザー名'])) {  
        not_authorized() を返します。  
    }  
} それ以外 {  
    if (!$is_admin) {  
        not_authorized() を返します。  
    }  
}  
// ページのレンダリングを続行します ...
```

このコードにはかなりのロジックが含まれています。で見たようにパートII、ループとロジックの簡素化、このような大きなロジックツリーは理解するのが簡単ではありません。このコードのロジックはおそらく簡素化できますが、どうすればよいでしょうか? ロジックをわかりやすい英語で説明することから始めましょう。

認証するには2つの方法があります: 1) あなたは管理者です。
2) 現在のドキュメントを所有している場合(存在する場合)
それ以外の場合、権限がありません。

この説明に基づいた代替ソリューションを次に示します。

```
if (is_admin_request()) {  
    // 許可された  
elseif ($document && ($document['ユーザー名'] == $_SESSION['ユーザー名'])) {  
    // 許可された  
} それ以外 {  
    not_authorized() を返します。  
}  
  
// ページのレンダリングを続行します ...
```

このバージョンは空のボディが2つあるため、少し特殊です。ただし、否定がないため、コードは小さくなり、ロジックは単純になります。(前の解決策には3つの「否定事項」がありました。) 肝心のは、理解しやすいということです。

ライブラリを知ると役立つ

かつて、ユーザーに役立つ次のような提案を表示する「ヒントボックス」を備えたWebサイトがありました。

ヒント: 過去のクエリを確認するには、ログインしてください。[別のヒントを見せてください！]

数十のヒントがあり、それらはすべてHTML内に隠されていました。

```
<div id="tip-1" class="tip">ヒント: 過去のクエリを表示するには、ログインしてください。</div> <div id="tip-2" class="tip">ヒント: 画像をクリックしてください。</div> を近くで見ることができます。。。
```

ユーザーがページにアクセスすると、次のいずれかが表示されます。ディビジョンsはランダムに表示され、残りは非表示のままでした。

「別のヒントを表示！」リンクをクリックすると、次のヒントに循環します。jQuery JavaScriptライブラリを使用してその機能を実装するコードを次に示します。

```
var show_next_tip = function () {  
    var num_tips = $('.tip').size(); var  
    show_tip = $('.tip:visible');  
  
    var show_tip_num = Number(shown_tip.attr('id').slice(4)); if  
    (show_tip_num === num_tips) {  
        $('#tip-1').show();  
    } それ以外 {  
        $('#tip-' + (show_tip_num + 1)).show();  
    }  
    表示された_ヒント.非表示();  
};
```

このコードは大丈夫です。しかし、それをより良くすることは可能です。このコードが何をしようとしているのかを言葉で説明するから始めましょう。

現在表示されているヒントを見つけて非表示にします。次に、その次のヒントを見つけてそれを示します。
ヒントがなくなった場合は、最初のヒントに戻ります。

この説明に基づいて、別の解決策を次に示します。

```
var show_next_tip = function () {
    var cur_tip = $('.tip:visible').hide(); var
    next_tip = cur_tip.next('.tip'); if
    (next_tip.size() === 0) {
        next_tip = $('.tip:first');
    }
    next_tip.show();
};
```

// 現在表示されているヒントを見つけて非表示にしま
す // その次のヒントを見つけて
// ヒントがなくなったら、 // 最初のヒン
トに戻ります
// 新しいヒントを表示する

このソリューションに含まれるコード行は少なく、整数を直接操作する必要がありません。それは、人間がコードについてどのように考えるかにより一致します。

この場合、jQuery にがあることが役に立ちました。次 () 私たちが使える方法。簡潔なコードを書くには、ライブラリが何を提供するかを認識することが必要です。

この方法をより大きな問題に適用する

前の例では、プロセスを小さなコード ブロックに適用しました。次の例では、これをより大きな関数に適用します。ご覧のとおり、この方法は、切り離せる部分を特定するのに役立ち、コードを分割するのに役立ちます。

株式の購入を記録するシステムがあると想像してください。各トランザクションには次の 4 つのデータがあります。

- 時間 (購入の正確な日時)
- ティッカーシンボル (例: GOOG)
- 価格 (例: 600 ドル)
- 株数 (例: 100)

ここに示すように、何らかの奇妙な理由で、データは 3 つの別々のデータベース テーブルに分散されています。各データベースでは、時間は一意の主キーです。

time	ticker_symbol	time	price	time	number_of_shares
3:45	IBM	3:45	\$120	3:45	50
3:59	IBM	4:30	\$600	3:59	200
4:30	GOOG	5:00	\$25	4:10	75
5:20	AAPL	5:20	\$200	4:30	100
6:00	MSFT	6:00	\$25	5:20	80

ここで、3 つのテーブルを再び結合するプログラムを (SQL として) 記述する必要があります。参加する操作でも構いません)。行はすべて次の基準で並べ替えられているため、この手順は簡単です。時間、しかし、残念ながら一部の行が欠落しています。3つすべてが含まれるすべての行を検索したいとします。時間前の図に示すように、一致し、整列できない行は無視されます。

一致する行をすべて検索する Python コードを次に示します。

```
def PrintStockTransactions():
    Stock_iter=db_read("SELECT 時間、ティッカーシンボル FROM ...")
    価格_イター= ...
    num_shares_iter= ...

    # 3つのテーブルのすべての行を並行して繰り返します。その間
    Stock_iterそして価格_イターそしてnum_shares_iter:
        在庫時間 =Stock_iter.time 価格_時間 =価格_
        イター.ctime num_shares_time =
        num_shares_iter。時間

        # 3行すべてが同じ時刻でない場合は、最も古い行をスキップします
        # 注: 同点で古いものが2つある場合、以下の「<」を単に「<」にすることはできません。
        Stock_time != Price_time または Stock_time != num_shares_time の場合:
            在庫時間 <= 価格時間および在庫時間 <= 株数数時間の場合:
                Stock_iter.NextRow()
            elif 価格_時間 <= 株式時間および価格_時間 <= 株数_時間:
                価格_イター.NextRow()
            elif num_shares_time <= Stock_time および num_shares_time <= Price_time:
                num_shares_iter.NextRow() それ以外
            の場合:
                False をアサート # 不可能 続く

        株式時間 == 価格_時間 == 株数_時間のアサート

        # 整列した行を印刷します。
        印刷 "@"、stock_time、
        印刷する Stock_iter.ticker_symbol、
        印刷価格_イター。価格、
        印刷する num_shares_iter.nu mber_of_shares

        Stock_iter.NextRow()
        価格_イター.NextRow()
        num_shares_iter.NextRow()
```

このサンプル コードは機能しますが、ループが一致しない行をスキップする方法に関しては多くの処理が行われます。あなたの頭の中でいくつかの警告フラグが立っているかもしれません。これで行が欠落する可能性はありますか? いずれかの反復子のストリームの終わりを超えて読み取られる可能性がありますか?

では、どうすればもっと読みやすくできるのでしょうか?

ソリューションの英語の説明

もう一度、一步下がって、私たちがやろうとしていることを平易な英語で説明しましょう。

3つの行反復子を並行して読み取っています。
列の時間が一致しない場合は、列を進めて列を並べます。次に、整列した行を印刷し、再度行を進めます。一致する行がなくなるまでこれを続けます。

元のコードを振り返ってみると、最も厄介な部分は「行を進めて整列させる」を処理するブロックでした。コードをより明確に示すために、煩雑なロジックをすべて抽出することができます。という名前の新しい関数にAdvanceToMatchingTime()。

この新しい関数を利用した新しいバージョンのコードを次に示します。

```
def PrintStockTransactions():
    Stock_iter= ...
    価格_イター= ...
    num_shares_iter= ...

    True の場合:
        時間 =AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter) 時間が
        「なし」の場合:
            戻る

        # 整列した行を印刷します。「@」、
        時刻、を印刷します。
        Stock_iter.ticker_symbol を印刷、
        price_iter.price を印刷、
        print num_shares_iter.number_of_shares

        Stock_iter.NextRow()
        価格_イター.NextRow()
        num_shares_iter.NextRow()
```

ご覧のとおり、行を並べる際の汚い詳細がすべて隠されているため、このコードは非常に理解しやすくなっています。

メソッドを再帰的に適用する

どのように書くか想像しやすいAdvanceToMatchingTime()—最悪の場合、最初のバージョンの醜いコード ブロックと非常によく似たものになるでしょう。

```
def AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter):
    # 3つのテーブルのすべての行を並行して繰り返します。その間
    Stock_iterそして価格_イターそしてnum_shares_iter:
        在庫時間 =Stock_iter.time 価格_時間 =価格_
        イター.time num_shares_time =
        num_shares_iter。時間

        # 3行すべてが同じ時刻ではない場合、stock_time != Price_time または
        Stock_time != num_shares_time の場合、最も古い行をスキップします。
        在庫時間 <= 価格時間および在庫時間 <= 株数数時間の場合:
            Stock_iter.NextRow()
        elif 価格_時間 <= 株式時間および価格_時間 <= 株数_時間:
            価格_イター.NextRow()
        elif num_shares_time <= Stock_time および num_shares_time <= Price_time:
            num_shares_iter.NextRow() それ以外
        の場合:
            False をアサート # 不可能 続く
```

```
アサート stock_time == price_time == num_shares_time
return stock_time
```

しかし、私たちのメソッドを適用してコードを改善しましょう。AdvanceToMatchingTime()同じように。この関数が何を行う必要があるかについては、次のとおりです。

現在の各行の時間を確認します。それらが揃っていれば完了です。それ以外の場合は、「後ろ」の行を進めます。
行が整列するまで(またはイテレータの1つが終了するまで)、これを繰り返します。

この説明は、前のコードよりもはるかに明確で洗練されています。注意すべき点の1つは、説明にはまったく言及されていないことです。Stock_iterまたは私たちの問題に特有のその他の詳細。これは、変数の名前をより単純かつ一般的なものに変更することもできることを意味します。結果のコードは次のとおりです。

```
def AdvanceToMatchingTime(row_iter1, row_iter2, row_iter3):
    一方、row_iter1、row_iter2、およびrow_iter3は次のようにになります。
    t1 = row_iter1.time
    t2 = row_iter2.time
    t3 = row_iter3.time

    t1 == t2 == t3の場合:
        t1 を返す

    tmax = 最大(t1, t2, t3)

    # 行が「後ろ」にある場合は、その行を進めます。
    # 最終的には、この while ループによってすべてが調整されます。t1
    < tmax の場合: row_iter1.NextRow()
    t2 < tmax の場合: row_iter2.NextRow() t3
    < tmax の場合: row_iter3.NextRow()

    return None # アライメントが見つかりませんでした
```

ご覧のとおり、このコードは以前よりもはるかに明確になっています。アルゴリズムがよりシンプルになり、難しい比較が少なくなりました。そして、次のような短い名前を使用しましたt1また、関係する特定のデータベース列について考える必要もなくなりました。

まとめ

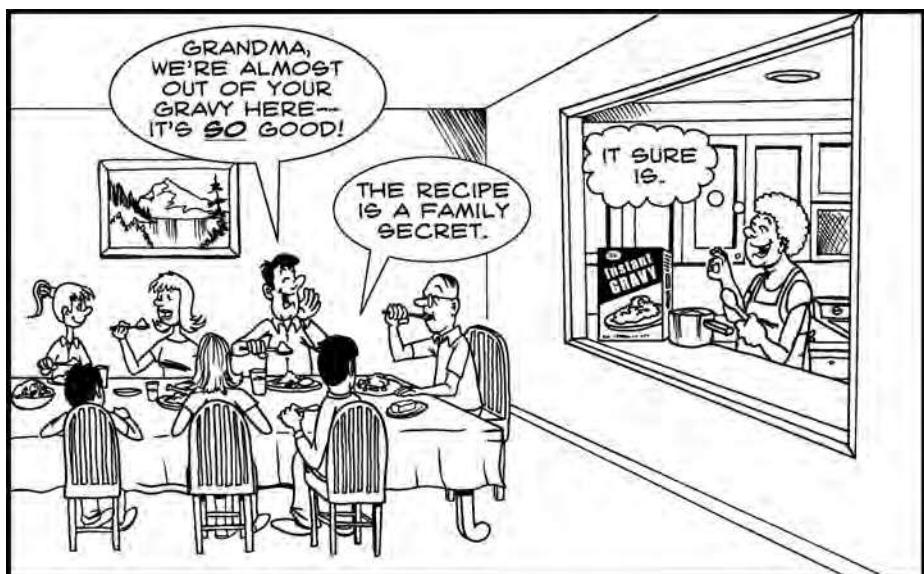
この章では、プログラムを平易な英語で記述し、その記述を使用してより自然なコードを作成するための簡単なテクニックについて説明しました。このテクニックは一見シンプルですが、非常に強力です。説明に使用されている単語やフレーズを確認することも、どのサブ問題を解決するかを特定するのに役立ちます。

しかし、この「平易な英語で物事を言う」というプロセスは、コードを書く以外にも応用できます。たとえば、ある大学のコンピュータラボのポリシーでは、学生がプログラムのデバッグに手助けが必要な場合、まず部屋の隅にいる専用のティベアに問題を説明する必要があると規定されています。驚くべきことに、問題を声に出して説明するだけで、生徒が解決策を見つけるのに役立つことがあります。この技術は「ラバーダッキング」と呼ばれます。

別の見方としては、問題や設計を言葉で説明できない場合は、何かが欠落しているか未定義である可能性があります。プログラム（またはアイデア）を言葉にすると、それを強制的に形にすることができます。

第十三章

コードの記述量を減らす



いつのまにか知らないコードを書くことは、おそらくプログラマーが学ぶことができる最も重要なスキルです。作成するコードの各行は、テストおよび保守が必要な行です。ライブラリを再利用したり、機能を削除したりすることで、時間を節約し、コードベースを無駄のない効率的なものに保つことができます。

キー・アイデア

最も読みやすいコードは、コードがまったく存在しないことです。

その機能はわざわざ実装する必要はありません - 必要ありません

プロジェクトを開始するとき、興奮して実装したいすべての素晴らしい機能について考えるのは自然なことです。

しかし、プログラマーは機能の数を過大評価する傾向があります。本当に必要不可欠な彼らのプロジェクトに。多くの機能は未完成または未使用のままであり、あるいは単にアプリケーションを複雑にするだけです。

プログラマーはまた、機能の実装にかかる労力を過小評価する傾向があります。私たちは、大まかなプロトタイプの実装にかかる時間を楽観的に見積もっていますが、将来のメンテナンス、文書化、およびコードベースへの追加の「重量」にどれだけの余分な時間がかかるかは忘れていません。

質問して要件を細分化する

すべてのプログラムが高速で、100% 正確で、あらゆる入力を処理できる必要があるわけではありません。要件を徹底的に精査すると、必要なコードが少なく、より単純な問題を切り出すことができる場合があります。この例をいくつか見てみましょう。

例: 店舗検索

ビジネス向けの「店舗検索」を作成しているとします。あなたは次のような要件があると考えています。

特定のユーザーの緯度/経度に対して、最も近い緯度/経度のストアを検索します。

これを 100% 正しく実装するには、以下を処理する必要があります。

- 日付変更線の両側にある場合
- 場所が北極または南極に近い場合
- 「1 マイルあたりの経度」の変化に応じて地球の曲率を調整する

これらすべてのケースを処理するには、かなりの量のコードが必要です。

ただし、あなたのアプリケーションの場合、テキサス州には 30 店舗しかありません。この小さな地域では、リストにある 3 つの問題はそれほど重要ではありません。その結果、要件を次のように減らすことができます。

テキサス近郊のユーザーの場合、テキサスで（おおよそ）最も近い店舗を検索します。

この問題を解決するのは簡単です。各ストアを反復処理して、緯度/経度間のユークリッド距離を計算するだけで済むからです。

例: キャッシュの追加

かつて、ディスクからオブジェクトを頻繁に読み取る Java アプリケーションがありました。アプリケーションの速度はこれらの読み取りによって制限されるため、何らかのキャッシュを実装したいと考えました。一般的な読み取りシーケンスは次のようになります。

```
オブジェクトAを読み取る
オブジェクトAを読み取る
オブジェクトAを読み取る
オブジェクトBを読み取る
オブジェクトBを読み取る
オブジェクトCを読み取る
オブジェクトDを読み取る
オブジェクトDを読み取る
```

ご覧のとおり、同じオブジェクトへの繰り返しアクセスが多数あったため、キャッシュが確実に役立つはずです。

この問題に直面したとき、私たちが最初に直感したのは、最も最近使用されていないアイテムを破棄するキャッシュを使用することでした。私たちのライブラリには利用可能なものがなかったため、自分で実装する必要がありました。ただし、そのようなデータ構造は以前に実装したことがあったため、これは問題ではありませんでした(これには、ハッシュテーブルと単一リンクリストの両方が含まれており、おそらく合計 100 行のコードになります)。

しかし、アクセスが常に連続していることに気付きました。したがって、LRU キャッシュを実装する代わりに、1 つのアイテムのキャッシュを実装しました。

```
最後に使用されたディスクオブジェクト; // クラスメンバー
```

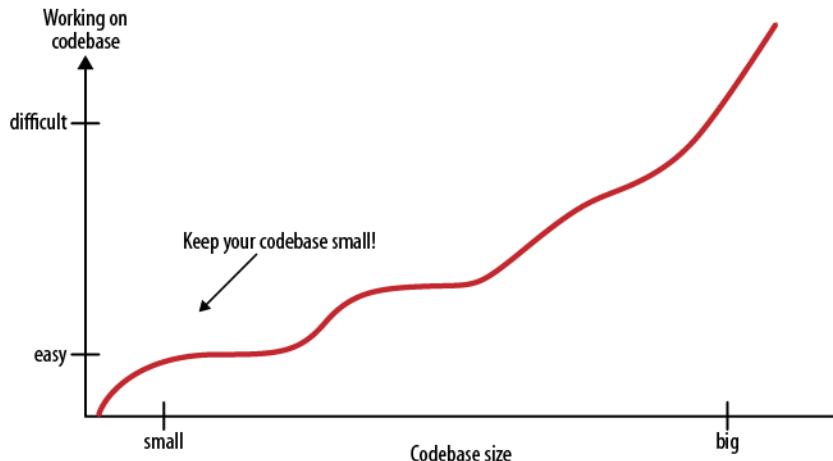
```
DiskObject lookUp(文字列キー) {
    if (lastused == null || !lastused.key().equals(key)) {
        lastused = loadDiskObject(キー);
    }
}
```

```
最後に使用したものを返します。
}
```

これにより、多くのコーディングを行わずに 90% のメリットが得られ、プログラムのメモリ使用量も小さくなりました。

「要件の削除」と「より単純な問題の解決」のメリットは、どれだけ誇張してもしすぎることはありません。要件は微妙な方法で相互に干渉することがよくあります。これは、問題の半分を解決するには、コーディングの労力が 4 分の 1 で済む可能性があることを意味します。

コードベースを小さく保つ



初めてソフトウェアプロジェクトを開始し、ソースファイルが1つか2つしかないときは、問題はありません。コードのコンパイルと実行は簡単で、変更も簡単に行え、各関数やクラスがどこで定義されているかを覚えておくのも簡単です。

その後、プロジェクトが成長するにつれて、ディレクトリはますます多くのソースファイルでいっぱいになります。すぐに、それらをすべて整理するために複数のディレクトリが必要になります。どの関数が他のどの関数を呼び出しているかを覚えるのは難しく、バグを追跡するにはもう少し手間がかかります。

最終的には、多くのソースコードがさまざまなディレクトリに分散されることになります。このプロジェクトは巨大であり、すべてを理解している人は一人もいません。新しい機能を追加するのは苦痛であり、コードを扱うのは面倒で不快です。

私たちが説明したことは、宇宙の自然法則です。調整されたシステムが成長するにつれて、それを結合し続けるために必要な複雑さはさらに速く増加します。

対処する最善の方法は、**コードベースを可能な限り小さく軽量に保つ**プロジェクトが成長しても。したがって、次のことを行う必要があります。

- 可能な限り多くの汎用「ユーティリティ」コードを作成し、重複したコードを削除します。
(見る [第10章無関係な部分問題の抽出](#)。)
- 未使用のコードまたは不要な機能を削除します。(次の補足記事を参照してください。)
- プロジェクトを、切り離されたサブプロジェクトに分割しておきます。
- 一般に、コードベースの「重み」を意識してください。軽くて機敏な動きを保ちます。

未使用のコードの削除

庭師は、植物を生かして成長させるために植物を剪定することがよくあります。同様に、邪魔になる未使用的コードを削除することをお勧めします。

一度コードが記述されると、多くの実際の作業が行われるため、プログラマーはそのコードを削除することに消極的になることがよくあります。それを削除するということは、それに費やした時間が無駄だったと認めることがあります。まあ、乗り越えてください！これはクリエイティブな分野です。写真家、作家、映画制作者も、すべての作品を保管しているわけではありません。

孤立した関数を削除するのは簡単ですが、気づかぬうちに「未使用のコード」が実際にプロジェクト全体に織り込まれていることがあります。ここではいくつかの例を示します。

- ・もともと国際的なファイル名を処理するようにシステムを設計していましたが、現在はコードに変換コードが散りばめられています。ただし、そのコードは完全に機能するわけではなく、アプリが国際的なファイル名で使用されることはありません。

なぜこの機能を削除しないのでしょうか？

- ・システムのメモリが不足してもプログラムが動作するようにしたいため、メモリ不足の状況から回復しようとする賢いロジックがたくさんあります。これは良いアイデアではありましたか、実際にシステムのメモリが不足すると、プログラムはとにかく不安定なゾンビになります。すべてのコア機能が使用できなくなり、マウス1クリックで終了してしまいます。

「システムのメモリが不足しています。申し訳ありません」という単純なメッセージでプログラムを終了し、このメモリ不足のコードをすべて削除してはどうでしょうか？

あなたの周りの図書館をよく知る

多くの場合、プログラマは既存のライブラリで問題を解決できることに気づいていません。あるいは、図書館で何ができるかを忘れてしまうこともあります。ライブラリコードを利用できるように、その機能を知ることが重要です。

ここにささやかな提案があります:時々、15分間かけて、標準ライブラリ内のすべての関数、モジュール、型の名前を読んでください。これらには、C++ 標準テンプレート ライブラリ (STL)、Java API、組み込み Python モジュールなどが含まれます。

目標はライブラリ全体を暗記することではありません。これは、利用可能なものを把握するためのものです。そうすれば、次回新しいコードに取り組むときに、「待って、これはAPIで見たものと似ているような…」と思うでしょう。私たちは、この作業を事前に行うと、最初からこれらのライブラリを使用する傾向が高まるため、すぐに効果が得られると考えています。

例: Python のリストとセット

Python でリストがあるとします ([のような] 2, 1, 2]) 一意の要素のリストが必要です (この場合、[2,1])。このタスクは、一意であることが保証されているキーのリストを含む辞書を使用して実装できます。

```
def unique(要素):
    温度 = {}
    要素内の要素の場合:
        temp[element] = None # 値は重要ではありません。
    temp.keys()を返す

unique_elements = unique([2,1,2])
```

ただし、代わりに、あまり知られていないものを使用することもできますセットタイプ：

```
unique_elements = セット([2,1,2]) # 重複を削除
```

このオブジェクトは通常のオブジェクトと同様に反復可能です。リスト。本当に欲しいならリスト再びオブジェクトを使用する場合は、次のようにするだけです。

```
unique_elements = リスト(set([2,1,2])) # 重複を削除
```

明らかに、セットここでの仕事に最適なツールです。しかし、もしあなたがそのことに気づいていなかったら、セットと入力すると、次のようなコードが生成されます。個性的 () その上。

ライブラリの再利用が大きなメリットとなる理由

よく引用される統計として、平均的なソフトウェアエンジニアは 1 日に出荷可能なコードを 10 行作成するというものがあります。プログラマがこれを初めて聞くと、信じられないというようにためらいます。「コードが 10 行? すぐに書けますよ!」

キーワードは出荷可能。成熟したライブラリのコードの各行は、かなりの量の設計、デバッグ、書き換え、文書化、最適化、テストを表します。このダーウィンのプロセスを生き延びたコード行は非常に貴重です。このため、ライブラリを再利用すると、時間を節約し、記述するコードが少なくなるという点で非常に効果的です。

例: コーディングの代わりに Unix ツールを使用する

Web サーバーが 4xx または 5xx HTTP 応答コードを頻繁に返す場合、それは潜在的な問題の兆候です (4xx はクライアントエラー、5xx はサーバー エラー)。そこで私たちは、Web サーバーのアクセス ログを解析し、どの URL が最も多くのエラーを引き起こしているかを判断するプログラムを作成したいと考えました。

アクセス ログは通常次のようにになります。

```
1.2.3.4 example.com [24/Aug/2010:01:08:34] "GET /index.html HTTP/1.1" 200 ...
2.3.4.5 example.com [24/Aug/2010:01:14:27] "GET /help?topic=8 HTTP/1.1" 500 ...
3.4.5.6 example.com [24/Aug/2010:01:15:54] "GET /favicon.ico HTTP/1.1" 404 ...
. . .
```

通常、これらには次の形式の行が含まれます。

ブラウザ IP ホスト [日付] "GET /url-path HTTP/1.1" HTTP 応答コード ...

最も一般的なを見つけるプログラムを作成するURLパス4xx または 5xx 応答コードを含む s は、C++ や Java などの言語で簡単に 20 行のコードを必要とする可能性があります。

代わりに、Unix では、次のコマンド ラインを入力できます。

```
猫のアクセスログ | awk '{ print $5 "" $7 }' | egrep "[45].$" \| 並べ替え |  
ユニーク -c | 並べ替え -nr
```

これにより、次のような出力が生成されます。

```
95 /favicon.ico 404  
13 /help?topic=8 500  
11 /ログイン 403  
。。。  
<カウント></パス><http 応答コード>
```

このコマンド ラインの優れた点は、「実際の」コードを記述したり、ソース管理に何かをチェックしたりする必要がないことです。



まとめ

冒険、興奮——ジェダイが切望するものはこれらではない。

—ヨーダ

この章では、新しいコードをできるだけ少なく書くことについて説明します。新しいコード行ごとにテスト、文書化、保守する必要があります。さらに、コードベース内のコードが増えるほど「重くなり」、開発が難しくなります。

次のようにすることで、新しいコード行の記述を避けることができます。

- 製品から不要な機能を削除し、過剰なエンジニアリングを行わない
- 要件を再考して、仕事を遂行できる最も簡単なバージョンの問題を解決する
- 定期的に API 全体を読んで標準ライブラリに精通する

PARTIV

選択されたトピック

前の3つのパートでは、コードを理解しやすくするための幅広いテクニックを取り上げました。このパートでは、これらのテクニックのいくつかを、選択した2つのトピックに適用します。

まず、テストについて説明します。つまり、効果的でありながら同時に読みやすいテストを作成する方法です。

次に、特別な目的のデータ構造（「分/時間カウンタ」）の設計と実装を詳しく見て、パフォーマンス、優れた設計、読みやすさが相互作用する例を見ていきます。

第十四章

テストと可読性



この章では、きちんとした効果的なテストを作成するための簡単なテクニックを紹介します。

テストは人によって意味が異なります。この章では、「テスト」という言葉を次の意味で使用します。どれでも別の(「実際の」)コード部分の動作をチェックすることを唯一の目的とするコード。ここではテストの読みやすさの側面に焦点を当て、実際のコードを書く前にテストコードを書くべきか(「テスト駆動開発」)、またはテスト開発のその他の哲学的側面については触れません。

テストを読みやすく維持しやすくする

テストコードが読みやすいことは、テスト以外のコードと同様に重要です。他のプログラマーは、テストコードを、実際のコードがどのように機能し、使用されるべきかに関する非公式のドキュメントとして見ることがよくあります。したがって、テストが読みやすければ、ユーザーは実際のコードがどのように動作するかをよりよく理解できるようになります。

キーアイデア

他のプログラマーがテストを変更したり追加したりできるように、テストコードは可読である必要があります。

テストコードが大きくて恐ろしい場合、次のようなことが起こります。

- プログラマーは実際のコードを変更することを恐れます。ああ、すべてのテストを更新するのは悪夢のようなコードをいじりたくありません。
- プログラマーは新しいテストを追加しない新しいコードを追加するとき。時間が経つにつれて、テストされるモジュールの数が減り、すべてが機能するという確信が持てなくなります。

代わりに、コードのユーザー(特にあなた!)がテストコードに慣れるよう奨励したいと考えています。新しい変更が既存のテストを破壊する理由を診断でき、新しいテストを追加するの簡単だと感じるはずです。

このテストの何が間違っているのでしょうか?

私たちのコードベースには、スコア付けされた検索結果のリストを並べ替えてフィルターする関数がありました。関数の宣言は次のとおりです。

```
// 'docs' をスコア順に並べ替え(高い順)、マイナススコアのドキュメントを削除します。  
void SortAndFilterDocs(vector<ScoredDocument>* docs);
```

この関数のテストはもともと次のようにになっていました。

```
void Test1(){  
    Vector<ScoredDocument> ドキュメント;  
    ドキュメント docs.resize(5);  
    docs[0].url = "http://example.com";  
    docs[0].score = -5.0;  
    docs[1].url = "http://example.com";  
    docs[1].score = 1;
```

```

docs[2].url = "http://example.com";
docs[2].score = 4;
docs[3].url = "http://example.com";
docs[3].score = -99998.7;
docs[4].url = "http://example.com";
docs[4].score = 3.0;

SortAndFilterDocs(&docs);

アサート(docs.size() == 3);
assert(docs[0].score == 4);
assert(docs[1].score == 3.0);
assert(docs[2].score == 1);
}

```

少なくともあります八このテストコードにはさまざまな問題があります。この章の終わりまでに、それらをすべて特定して修正できるようになります。

このテストをさらに読みやすくする

一般的な設計原則として、次のことを行う必要があります。**重要度の低い詳細をユーザーから隠し、より重要な詳細が最も目立つようにします。**

前のセクションのテストコードは、明らかにこの原則に違反しています。テストのセットアップにおける重要でない細部など、テストのあらゆる詳細が中心にあります。ベクトル<スコア付きドキュメント>。サンプルコードのほとんどには、URL、スコア、そしてドキュメント[]、これは、基礎となる C++ オブジェクトがどのように設定されるかについての単なる詳細であり、このテストが高レベルで何を行っているかについては説明しません。

これをクリーンアップするための最初のステップとして、次のようなヘルパー関数を作成できます。

```

void MakeScoredDoc(ScoredDocument* sd, ダブルスコア, 文字列 URL) {
    sd->スコア = スコア;
    sd->url = URL;
}

```

この関数を使用すると、テストコードがわずかにコンパクトになります。

```

void Test1() {
    Vector<ScoredDocument> ドキュメ
    ント; docs.resize(5);
    MakeScoredDoc(&docs[0], -5.0, "http://example.com");
    MakeScoredDoc(&docs[1], 1, "http://example.com");
    MakeScoredDoc(&docs[2], 4, "http://example.com");
    MakeScoredDoc(&docs[3], -99998.7, "http://
    example.com");。。。
}

```

しかし、これだけでは十分ではありません。重要でない詳細がまだ目の前にあります。たとえば、パラメータ「http://example.com」目障りなだけです。これは常に同じであり、正確な URL は重要ではありません。有効な URL を入力する必要があるだけです。スコア付きドキュメント。

私たちが見なければならないもう1つの重要ではない詳細は、docs.resize(5)そして &ドキュメント[0]、&ドキュメント[1]、等々。ヘルパー関数を変更して、より多くの作業を実行できるようにして呼び出してみましょう。AddScoredDoc():

```
void AddScoredDoc(vector<ScoredDocument>& ドキュメント, ダブル スコア) {
    スコア付きドキュメント sd;
    sd.score = スコア;
    sd.url = "http://example.com";
    docs.push_back(sd);
}
```

この関数を使用すると、テストコードはさらにコンパクトになります。

```
void Test1() {
    Vector<ScoredDocument> ドキュメント; AddScoredDoc(docs, -5.0);
    AddScoredDoc(docs, 1);
    AddScoredDoc(docs, 4);
    AddScoredDoc(docs,
    -99998.7); . . .
}
```

このコードはより優れていますが、「読み書き可能性が高い」テストにはまだ合格していません。新しいスコア付きドキュメントのセットを使用して別のテストを追加したい場合は、大量のコピー アンド ペーストが必要になります。では、さらに改善するにはどうすればよいでしょうか?

最小限のテストステートメントの作成

このテストコードを改善するには、次のテクニックを使用してみましょう。[第12章 思考をコードに変える](#)。私たちのテストが何をしようとしているのかをわかりやすい英語で説明してみましょう。

スコアが [-5, 1, 4, -99998.7, 3]。後 SortAndFilterDocs()、残りのドキュメントのスコアは [4, 3, 1]、その順番で。

ご覧のとおり、その説明のどこにも、ベクトル<スコア付きドキュメント>。ここで最も重要なのはスコアの配列です。理想的には、テストコードは次のようにになります。

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");
```

このテストの本質を1行のコードに要約することができました。

ただし、これは珍しいことではありません。ほとんどのテストの本質は次のとおりです。この入力/状況では、この動作/出力が予想されます。そして多くの場合、この目標はたった1行で表現できます。コードを非常にコンパクトで読みやすくすることに加えて、テストステートメントを短くしておくと、テストケースをさらに簡単に追加できます。

カスタム「ミニ言語」の実装

注目してくださいCheckScoresBeforeAfter()スコアの配列を記述する2つの文字列引数を取ります。C++の新しいバージョンでは、次のように配列リテラルを渡すことができます。

```
CheckScoresBeforeAfter({-5, 1, 4, -99998.7, 3}, {4, 3, 1});
```

当時はこれを行うことができなかつたため、スコアをカンマで区切って文字列の中に入れました。このアプローチが機能するためには、CheckScoresBeforeAfter()これらの文字列引数を解析する必要があります。

一般に、カスタム ミニ言語の定義は、少ないスペースで多くの情報を表現する強力な方法となります。他の例としては、printf()および正規表現ライブラリ。

この場合、カンマ区切りの数値リストを解析するためのヘルパー関数を作成するのは、それほど難しいことではありません。これが何ですかCheckScoresBeforeAfter()次のようにになります:

```
void CheckScoresBeforeAfter(string input, string Expected_output) {
    Vector<ScoredDocument> docs = ScoredDocsFromString(input);
    SortAndFilterDocs(&docs);
    文字列出力 = ScoredDocsToString(docs); アサート(出力 == 期待される出力);
}
```

完全を期すために、以下に変換するヘルパー関数を示します。弦そしてベクトル<スコア付きドキュメント>:

```
Vector<ScoredDocument> ScoredDocsFromString(string スコア) {
    Vector<ScoredDocument> ドキュメント;
    replace(scores.begin(),scores.end(),' ','');

    // スペースで区切られたスコアの文字列から 'docs' を入力します。
    isstringstream ストリーム(スコア);
    ダブルスコア。
    while (ストリーム >> スコア) {
        AddScoredDoc(ドキュメント,スコア);
    }

    書類を返却する。
}

string ScoredDocsToString(vector<ScoredDocument> docs) {
    ostringstream ストリーム;
    for (int i = 0; i < docs.size(); i++) {
        if (i > 0) ストリーム << ", ";
        ストリーム << docs[i].score;
    }

    stream.str() を返します。
}
```

これは一見すると大量のコードのように思えるかもしれません、これによってできることは信じられないほど強力です。1回の呼び出しでテスト全体を作成できるため、CheckScoresBeforeAfter()、さらにテストを追加したくなるでしょう(この章の後半で行います)。

エラーメッセージを読みやすくする



前のコードは素晴らしいましたが、それが起こるとどうなるでしょうかアサート(出力 == 期待される出力)ラインが失敗しますか？次のようなエラーメッセージが表示されます。

アサーションが失敗しました: (output == Expect_output),
関数 CheckScoresBeforeAfter、ファイル test.cc、37 行目。

このエラーを見たことがあれば、明らかに次のように疑問に思うでしょう。の価値観は何でしたか出力そして期待される出力？

より良いバージョンのassert()の使用

幸いなことに、ほとんどの言語とライブラリには、より洗練されたバージョンの主張する () 使用できます。したがって、次のように書く代わりに:

アサート(出力 == 期待される出力);

Boost C++ ライブラリを使用できます。

BOOST_REQUIRE_EQUAL(出力、期待される出力)

テストが失敗すると、次のような詳細なメッセージが表示されます。

test.cc(37): 「CheckScoresBeforeAfter」の致命的なエラー: 重要なチェック
出力 == 期待された出力が失敗しました [「1、3、4」 != 「4、3、1」]

それははるかに役に立ちます。

これらのより便利なアサーション メソッドが利用可能な場合は、それを使用する必要があります。テストが失敗するたびに報われます。

他の言語でのより良い Assert()

Python では、組み込みステートメント `a==b` をアサートします 次のような単純なエラー メッセージが生成されます。

```
ファイル「file.py」、<module> の X 行目
a==b をアサートします
アサーションエラー
```

代わりに、アサート `Equal()` のメソッド 単体テスト モジュール:

単体テストのインポート

```
クラス MyTestCase(unittest.TestCase):
    def testFunction(self):
        a = 1
        b = 2
        self.assertEqual(a, b)
```

`__name__ == '__main__'` の場合:
ユニットテスト.メイン()

これにより、次のようなエラー メッセージが生成されます。

```
ファイル「MyTestCase.py」、testFunction の 7 行目
self.assertEqual(a, b)
アサーションエラー:1 != 2
```

どの言語を使用している場合でも、役立つライブラリ/フレームワーク (XUnit など) がおそらく存在します。
ライブラリを知ることは有益です!

手作りのエラーメッセージ

使用する `BOOST_REQUIRE_EQUAL()`、よりわかりやすいエラー メッセージを取得することができました。

出力 == 期待された出力が失敗しました [「1、3、4」 != 「4、3、1」]

ただし、このメッセージはさらに改善される可能性があります。たとえば、この失敗を引き起こした元の入力を確認すると便利です。理想的なエラー メッセージは次のようなものです。

```
CheckScoresBeforeAfter() が失敗しました。
入力 :          「-5、1、4、-99998.7、3」
期待される出力: "4、3、1" 実
際の出力: "1、3、4"
```

これがあなたが望むものであれば、先に進んで書いてください！

```

void CheckScoresBeforeAfter(...) {
    . . .

    if (出力 != 期待される出力) {
        cerr << "CheckScoresBeforeAfter() が失敗しました。" << endl;
        cerr << "入力: " << input << endl;
        cerr << "期待される出力: " << Expected_output << endl; cerr << "実
        際の出力: " << 出力 << endl; cerr << endl; アボート () ;

    }
}

```

この話の教訓は、エラーメッセージはできる限り役に立つべきだということです。場合によっては、「カスタムアサーション」を構築して独自のメッセージを出力することが最善の方法です。

適切なテスト入力の選択

テストに適切な入力値を選択するにはコツがあります。私たちが現在持っているものは少し無計画に思えます:

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3", "4, 3, 1");
```

適切な入力値を選択するにはどうすればよいでしょうか? 適切な入力では、コードを徹底的にテストする必要があります。ただし、読みやすいようにシンプルである必要もあります。

キーアイデア

一般に、コードを完全に実行する最も単純な入力セットを選択する必要があります。

たとえば、次のように書いたとします。

```
CheckScoresBeforeAfter("1, 2, 3", "3, 2, 1");
```

このテストは単純ですが、「負のスコアをフィルターする」動作はテストしません。

SortAndFilterDocs()。コードのその部分にバグがある場合、この入力はバグをトリガーしません。

反対の極端な例として、テストを次のように書いたとします。

```
CheckScoresBeforeAfter("123014, -1082342, 823423, 234205, -235235",
                      "823423, 234205, 123014");
```

これらの値は必要に複雑です。(そして、コードを徹底的にテストすることさえしません。)

入力値の簡略化

では、これらの入力値を改善するにはどうすればよいでしょうか?

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",           「4, 3, 1」);
```

おそらく最初に気づくのは、非常に「うるさい」値であることです。99998.7。この値は単に「任意の負の数」を意味しているため、より単純な値は次のようにになります。1. (もし -99998.7 は「非常に負の数」を意味していましたが、より良い値は次のような鮮明な値でした。1e100。)

キーアイデア

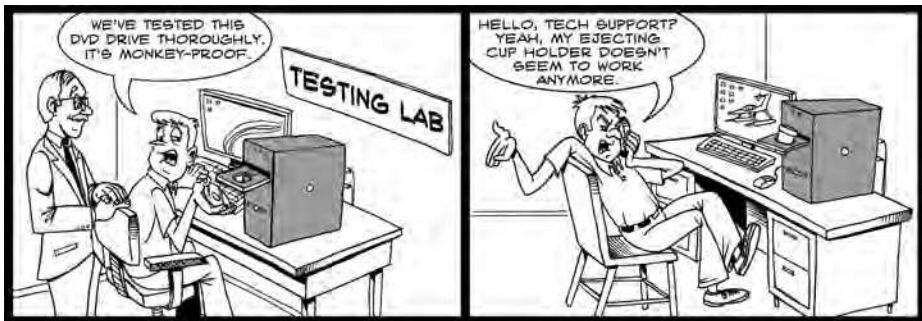
作業を完了できるクリーンでシンプルなテスト値を好みます。

テストの他の値はそれほど悪くありませんが、ここでそれらを可能な限り単純な整数に減らすことができます。また、負の値が削除されるかどうかをテストするには、負の値が1つだけ必要です。テストの新しいバージョンは次のとおりです。

```
CheckScoresBeforeAfter("1, 2, -1, 3", "3, 2, 1");
```

効果を損なうことなくテスト値を簡素化しました。

大規模な「スマッシャー」テスト



大規模でクレイジーな入力に対してコードをテストすることには間違いなく価値があります。たとえば、次のようなテストを含めたくなるかもしれません。

```
CheckScoresBeforeAfter("100, 38, 19, -25, 4, 84, [たくさんの価値観] ...",  
"100, 99, 98, 97, 96, 95, 94, 93, ...");
```

このような大規模な入力は、バッファオーバーランやその他の予期しないバグなどのバグを明らかにするのに役立ちます。

しかし、このようなコードは大きくて見るのも恐ろしく、コードのストレステストには完全には効果的ではありません。代わりに、プログラムで大きな入力を構築し、(たとえば) 100,000 個の値の大きな入力を構築する方が効率的です。

複数の機能テスト

コードを徹底的に実行するために単一の「完璧な」入力を作成するよりも、複数の小さなテストを作成する方が簡単で効果的で読みやすい場合がよくあります。

各テストでは、コードを特定の方向に進めて、特定のバグを見つけようとします。たとえば、次の4つのテストがあります。SortAndFilterDocs():

```
CheckScoresBeforeAfter(「2、1、3」、「3、2、1」);           // 基本的な並べ替え
CheckScoresBeforeAfter(「0、-0.1、-10」、「0」);           // 0未満のすべての値が削除され
CheckScoresBeforeAfter(「1、-2、1、-2」、「1、1」);           ました // 重複は問題ありません //
CheckScoresBeforeAfter(「」、「」);                         空の入力はOK
```

非常に徹底的にテストしたい場合は、さらに多くのテストを作成できます。個別のテストケースがあると、次の人がコードに取り組むのが容易になります。誰かが誤ってバグを導入した場合、テストの失敗により、失敗した特定のテストが特定されます。

テスト関数の名前付け

テストコードは通常、テストするメソッドや状況ごとに1つずつ関数に編成されます。たとえば、コードのテストSortAndFilterDocs()という名前の関数内にありましたテスト1():

```
空所テスト1(){
  ...
}
```

テスト関数に適切な名前を付けるのは面倒で無関係に思えるかもしれません、次のような意味のない名前に頼らないでください。テスト1()、テスト2()、など。

代わりに、テストの詳細を説明するために名前を使用する必要があります。特に、テストコードを読む人が次のことをすぐに理解できると便利です。

- テスト対象のクラス (存在する場合)
- テスト対象の機能
- テストされている状況またはバグ

適切なテスト関数名を作成する簡単な方法は、その情報を、場合によっては「Test_」プレフィックスを付けて連結することです。

たとえば、名前を付ける代わりに、テスト1()、を使用できますTest_<関数名>()フォーマット：

```
空所Test_SortAndFilterDocs(){
  ...
}
```

このテストがどの程度洗練されているかに応じて、テストする状況ごとに個別のテスト関数を検討することもできます。を使用できますTest_<関数名>_<状況>()フォーマット：

```
空所Test_SortAndFilterDocs_BasicSorting(){
  ...
}
```

```
空所Test_SortAndFilterDocs_NegativeValues(){
    . . .
}
```

. . .

ここでは、長い名前やぎこちない名前を付けることを恐れないとください。これはコードベース全体で呼び出される関数ではないため、長い関数名を避ける理由は当てはまりません。テスト関数名は事実上コメントのように機能します。また、そのテストが失敗した場合、ほとんどのテストフレームワークはアサーションが失敗した関数の名前を出力するため、わかりやすい名前が特に役立ちます。

テストフレームワークを使用している場合は、メソッドの名前付けに関するルールや規則がすでに存在する可能性があることに注意してください。たとえば、Python単体テストモジュールは、テストメソッド名が「test」で始まるのを期待します。

ネーミングに関して言えば、ヘルパー・テストコード内の関数を使用する場合、その関数がそれ自体で何らかのアサーションを実行するのか、それとも単なる通常の「テストを意識しない」ヘルパーであるのかを強調すると便利です。たとえば、この章では、次のようなヘルパー関数を呼び出します。主張する()名前が付けられていますチェック...()。しかし、機能はAddScoredDoc()通常のヘルパー関数と同じように名前が付けられました。

そのテストの何が間違っていたのでしょうか?

この章の冒頭で、このテストには少なくとも8つの問題があると主張しました。

```
void Test1() {
    Vector<ScoredDocument> ドキュメント;
    ドキュメント docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;

    SortAndFilterDocs(&docs);

    アサート(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

より良いテストを作成するためのいくつかのテクニックを学習したので、それらを特定してみましょう。

1. テストは非常に長く、重要でない詳細がたくさんあります。このテストが何を行っているかを 1 つの文で説明できるため、テストステートメントはそれほど長くする必要はありません。
2. 別のテストを追加するのは簡単ではありません。コピー/貼り付け/変更したくなるでしょうが、そうするとコードがさらに長くなり、重複が多くなります。
3. テスト失敗メッセージはあまり役に立ちません。このテストが失敗すると、次のように表示されます。アサーションが失敗しました: `docs.size() == 3`、これでは、さらにデバッグするのに十分な情報が得られません。
4. テストでは、すべてを一度にテストしようとします。ネガティブ フィルタリングと並べ替え機能の両方をテストしようとしています。これを複数のテストに分割した方が読みやすくなります。
5. テスト入力は単純ではありません。特に、スコア例は `-99998.7` は「うるさく」、その特定の値に重要性がないにもかかわらず注意を引きます。単純な負の値で十分です。
6. テスト入力はコードを完全には実行しません。たとえば、スコアが次の場合はテストされません。0. (その文書はフィルタリングされるでしょうか?)
7. 空の入力ベクトル、非常に大きなベクトル、重複したスコアを持つベクトルなど、他の極端な入力はテストしません。
8. 名前 `test1()` は意味がありません。名前はテストされる機能または状況を説明する必要があります。

テストフレンドリーな開発

一部のコードは他のコードよりもテストが簡単です。テストに理想的なコードは、明確に定義されたインターフェイスを持ち、状態やその他の「セットアップ」があまりなく、検査する必要のある隠しデータがあまりないものです。

後でテストを作成することを承知してコードを作成すると、面白いことが起こります。 **テストしやすいようにコードの設計を開始します**。幸いなことに、この方法でコーディングすることは、一般的により良いコードを作成することも意味します。テストしやすい設計では、多くの場合、別々の処理を実行する別々の部分を備えた、よく整理されたコードが自然に生成されます。

テスト駆動開発

テスト駆動開発 (TDD) は、テストを作成するプログラミングスタイルです。前に実際のコードを書くのはあなたです。TDD 支持者は、このプロセスにより、コードを作成した後にテストを作成する場合よりも、テスト以外のコードの品質が大幅に向上升すると信じています。

これは激しく議論されているトピックなので、ここでは立ち入りません。少なくとも、コードを書くときにテストを念頭に置くだけで、コードの改善に役立つことがわかりました。

ただし、TDD を採用するかどうかに関係なく、最終的には他のコードをテストするコードが作成されます。この章の目的は、テストの読み書きを容易にすることです。

プログラムをクラスとメソッドに分割するすべての方法の中で、通常、最も分離された方法がテストしやすくなります。一方、プログラムが非常に相互接続されており、クラス間で多くのメソッド呼び出しがあり、すべてのメソッドに多くのパラメーターがあるとします。そのプログラムのコードが理解しにくいだけでなく、テストコードも同様に醜く、読み書きが困難になります。

多数の「外部」コンポーネント(初期化が必要なグローバル変数、ロードが必要なライブラリまたは構成ファイルなど)があると、テストを書くのも面倒になります。

一般に、コードを設計しているときに次のように認識すると、うーん、これは悪夢のようなテストになりそうだ、それは立ち止まってデザインを再考する良い理由になります。[表14-1](#)典型的なテストと設計の問題をいくつか示します。

表14-1. テストしにくいコードの特徴と、それがどのように設計上の問題につながるか

特性	テスト容易性の問題	設計上の問題
グローバル変数の使用	すべてのグローバル状態はテストごとにリセットする必要があります(そうしないと、異なるテストが相互に干渉する可能性があります)。	どの関数にどのような副作用があるのかを理解するのは困難です。各機能を切り離して考えることはできません。すべてが機能するかどうかを理解するには、プログラム全体を考慮する必要があります。
コードは多くの外部コンポーネントに依存します	最初に設定する足場が多すぎるため、テストを書くのが難しくなります。テストを書くのは楽しくないので、人々はテストを書くことを避けます。	依存関係の1つが失敗すると、システムが失敗する可能性が高くなります。特定の変更がどのような影響を与えるかを理解するのはさらに困難です。クラスをリファクタリングするにはさらに困難です。システムには、考慮すべき障害モードと回復パスがさらに多くあります。
コードには非決定的な動作がある	テストは不安定で信頼性が低いです。時々失敗するテストは無視されてしまいます。	プログラムには競合状態やその他の状態が発生する可能性が高くなります。再現不可能なバグ。このプログラムを推論するのはさらに困難です。本番環境のバグを追跡して修正するのは非常に困難です。

一方、テストを書きやすい設計がある場合、それは良い兆候です。[表14-2](#)に、いくつかの有益なテストおよび設計特性を示します。

表14-2. よりテストしやすいコードの特徴と、それがどのように優れた設計につながるか

特性	テスト容易性の利点	デザイン上のメリット
クラスには内部状態がほとんどまたはまったくありません	メソッドのテストに必要な設定が少なくなり、検査する必要のある隠し状態も少なくなるため、テストの作成が容易になります。	状態が少ないクラスは、より単純で理解しやすくなります。
クラス/関数は1のことだけを行います	完全にテストするために必要なテストケースは少なくなります。	より小型/単純なコンポーネントはよりモジュール化されており、システムは一般により分離されています。

クラスは他のいくつかのクラスに依存します。高いデカッピング	各クラスは独立してテストできます(複数のクラスを一度にテストするよりもはるかに簡単です)。	システムは並行して開発可能です。クラスは、システムの残りの部分を中断することなく簡単に変更または削除できます。
関数にはシンプルで明確に定義されたインターフェイスがあります	テストのための明確に定義された動作があります。シンプルなインターフェイスでは、テストにかかる作業が少なくなります。	インターフェイスはプログラマーにとって学びやすく、再利用される可能性が高くなります。

行き過ぎ

テストに集中しすぎる可能性もあります。ここではいくつかの例を示します。

- **テストを可能にするために、実際のコードの読みやすさを犠牲にします。** 実際のコードをテスト可能に設計することは、win-winの状況である必要があります。実際のコードはよりシンプルでより分離され、テストは簡単に作成できます。しかし、テストするために実際のコードに多くの醜い配管を挿入する必要がある場合は、何かが間違っています。
- **テストカバレッジを100%にすることに執着する。** コードの最初の90%をテストすることは、多くの場合、最後の10%をテストするよりも作業が少なくなります。残りの10%には、ユーザーインターフェイスや愚かなエラーのケースが含まれる可能性があります。これらの場合、バグのコストは実際にはそれほど高くなく、テストの労力はそれだけの価値がありません。

実際は、いずれにしても100%のカバレッジを得ることができないということです。見逃したバグではない場合は、見逃した機能であるか、仕様を変更する必要があることに気づいていない可能性があります。

バグのコストに応じて、テストコードにどれだけの開発時間を費やす価値があるかが決まります。Webサイトのプロトタイプを構築している場合、テストコードを作成する価値はまったくないかもしれません。一方、宇宙船や医療機器のコントローラーを作成している場合は、おそらくテストが主な焦点となります。

- **テストが製品開発の邪魔になる。** プロジェクトの1つの側面にすぎないはずのテストがプロジェクト全体を支配する状況を私たちは目にしてきました。テストはなだめられるある種の神のようになります。プログラマーは貴重なエンジニアリング時間を別のことに費やしたほうが良いかもしれないということに気付かず、ただ儀式や動作をこなすだけです。

まとめ

テストコードでは、可読性が依然として非常に重要です。テストが非常に読みやすい場合は、非常に書き込みやすいため、テストをさらに追加する人が増えます。また、実際のコードをテストしやすいように設計すると、コード全体の設計が向上します。

テストを改善するための具体的なポイントは次のとおりです。

- 各テストの最上位は可能な限り簡潔にする必要があります。理想的には、各テストの入出力を1行のコードで記述できます。

- テストが失敗した場合は、バグの追跡と修正が容易になるエラー メッセージを出力する必要があります。

- コードを完全に実行する最も単純なテスト入力を使用します。

- テスト関数に完全に説明的な名前を付けて、それぞれが何をテストしているのかが明確になるようにします。その代わりのテスト1()、のような名前を使用しますテスト_<関数名>_<状況>。

そして何よりも、新しいテストの変更と追加が簡単になります。

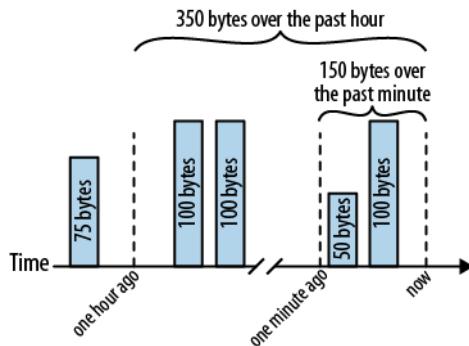
設計と実装 「分/時カウンター」



実際の運用コードで使用されるデータ構造、「分/時間カウンター」を見てみましょう。最初にこの問題を解決し、次にパフォーマンスを向上させ、機能を追加するという、エンジニアが経験するであろう自然な思考プロセスを説明します。最も重要なことは、本書全体の原則を使用して、コードを読みやすくするよう努めることです。途中で道を間違えたり、他の間違いを犯したりするかもしれません。追いかけて捕まえられるかどうかを確認してください。

問題

過去 1 分間および過去 1 時間に Web サーバーが転送したバイト数を追跡する必要があります。これらの合計がどのように維持されるかを示す図は次のとおりです。



これは非常に単純な問題ですが、ご覧のとおり、これを効率的に解決することは興味深い課題です。まずはクラスインターフェイスを定義しましょう。

クラスインターフェイスの定義

C++ のクラスインターフェイスの最初のバージョンは次のとおりです。

```
クラス分時カウンター{
    公共：
        // カウントを追加します
        空所カウント(int num_bytes);

        // この分のカウントを返します int分カウント();
        // この時間のカウントを返します int時間数();
};
```

このクラスを実装する前に、名前とコメントを調べて、変更する必要があるものがないか確認してみましょう。

名前の改善

クラス名分時カウンターかなり良いです。とても具体的で、具体的で、言いやすいです。

クラス名を指定すると、メソッド名がMinuteCount()そして時間数()も合理的です。あなたは彼らに電話したかもしれませんGetMinuteCount()そしてGetHourCount()、しかし、これは何の役にも立ちません。で述べたように、[第3章、誤解されない名前](#)、「get」は多くの人にとって「軽量アクセサ」を意味します。ご覧のとおり、実装は軽量ではないため、「get」を省略したままにするのが最善です。

メソッド名カウント()には問題がありますが。私たちは同僚にどう思うかを尋ねましたカウント()で十分ですが、これが「すべての期間の合計カウント数を返す」ことを意味すると考える人もいます。この名前は少し直観に反しています（しゃれではありません）。問題はそれですカウントは名詞と動詞の両方であり、「あなたが見たサンプルの数を数えてほしい」または「このサンプルを数えてほしい」のいずれかを意味します。

代わりに検討すべき代替名は次のとおりです。カウント()：

- インクリメント()
- 観察する()
- 記録()
- 追加()

インクリメント()増加するだけの値があることを暗示しているため、誤解を招きます。（この場合、時間数は時間の経過とともに変動します。）

観察する()大丈夫ですが、少し曖昧です。

記録()名詞・動詞の問題もあるのでダメです。

追加()興味深いのは、「これを数値的に追加する」または「データのリストに追加する」のいずれかを意味するためです。この場合、両方の要素が少し含まれているため、機能します。そこで、メソッドの名前を次のように変更します。void Add(int num_bytes)。

しかし、引数名はnum_bytesは具体的すぎます。はい、主な使用例はバイト数をカウントすることですが、分時カウンターこれを知る必要はありません。他の人がこのクラスを使用してクエリやデータベーストランザクションをカウントする可能性があります。次のようなより一般的な名前を使用することもできます。デルタ、しかし、用語デルタは、値が負になる可能性がある場所でよく使用されますが、それは望ましくありません。名前カウントこれは単純で一般的であり、「非否定的」であることを意味します。また、曖昧さの少ない文脈で「カウント」という単語をこっそり入れることもできます。

コメントの改善

これまでに作成したクラスインターフェイスは次のとおりです。

```
クラス MinuteHourCounter {
    公共:
```

```
// カウントを追加します
void Add(int count);

// この 1 分間のカウントを返します int
MinuteCount();

// この 1 時間のカウントを返します int
HourCount();
};
```

これらのメソッドのコメントをそれぞれ確認して改善してみましょう。最初のものについて考えてみましょう。

```
// カウントを追加します
void Add(int count);
```

このコメントは完全に不要になったので、削除するか改善する必要があります。改良版は次のとおりです。

```
// 新しいデータ ポイントを追加します (カウント >= 0)。
// 次の 1 分間、MinuteCount() は +count だけ大きくなります。// 次の 1
時間、HourCount() は +count だけ大きくなります。 void Add(int count);
```

次に、次のコメントについて考えてみましょう。MinuteCount():

```
// この 1 分間のカウントを返します int
MinuteCount();
```

このコメントの意味を同僚に尋ねたところ、次の 2 つの矛盾する解釈がありました。

1. 午後 12 時 13 分など、現在の分の間のカウントを返します。
2. 時間と分の境界に関係なく、過去 60 秒間のカウントを返します。

2 番目の解釈は、それが実際にどのように機能するかです。そこで、より正確かつ詳細な表現でこの混乱を解消しましょう。

```
// 過去 60 秒間の累積カウントを返します。 int MinuteCount();
```

(同様に、次のコメントも改善する必要があります)HourCount().)

これまでのすべての変更とクラスレベルのコメントを含むクラス定義を次に示します。

```
// 過去 1 分間および過去 1 時間の累積カウントを追跡します。// 最近の帯域幅使用状況を
追跡する場合などに役立ちます。
クラス MinuteHourCounter {
    // 新しいデータ ポイントを追加します (カウント >= 0)。
    // 次の 1 分間、MinuteCount() は +count だけ大きくなります。// 次の 1
時間、HourCount() は +count だけ大きくなります。 void Add(int count);

    // 過去 60 秒間の累積カウントを返します。 int MinuteCount();
```

```
// 過去 3600 秒間の累積カウントを返します。int HourCount();  
};  
(簡潔にするために、今後はコードリストからコメントを省略します。)
```

外部の視点を得る

お気づきかもしれません、同僚によって実行されたケースがすでにいくつかありました。外部の視点を求めるこ
とは、コードが「ユーザーフレンドリー」かどうかをテストする優れた方法です。他の人も同じ結論に達する可能性があ
るため、第一印象にオープンになるようにしてください。そして、その「他の人々」には次のような人が含まれる可
能性があります。あなた6か月以内に。

試み 1: 素朴な解決策

問題の解決に進みましょう。まずは簡単な解決策から始めます。

リストタイムスタンプ付きの「イベント」：

```
クラス MinuteHourCounter {  
    構造体イベント {  
        Event(int count, time_t time) : count(count), time(time) {}  
        int count;  
        time_t time;  
    };  
  
    リスト<イベント>イベント;  
  
    公共：  
        void Add(int count) {  
            イベント.push_back(イベント(カウント, 時間()));  
        }  
  
        . . .  
    };
```

その後、必要に応じて最新のイベントを数えることができます。

```
クラス MinuteHourCounter {  
    . . .  
  
    整数分カウント() {  
        int カウント = 0;  
        const time_t now_secs = time();  
        for (list<イベント>::reverse_iterator i = イベント.rbegin();  
            私!=イベント.rend() && i->time > now_secs - 60; ++i) { カウン  
ト += i->カウント;  
    }  
    戻り数;  
}
```

```

整数時間数(){
    int カウント = 0;
    const time_t now_secs = time();
    for (list<イベント>::reverse_iterator i = イベント.rbegin();
        私!=イベント.rend() && i->time > now_secs - 3600; ++i) { カウン
        ト += i->カウント;
    }
    戻り数;
}
};
```

コードは理解しやすいですか？

この解決策は「正しい」ですが、読みやすさの問題がいくつかあります。

- **ののためにループは少し長めです。**ほとんどの読者は、コードのこの部分を読んでいる間、速度が大幅に低下します(少なくとも、バグがないことを確認している場合は、速度が大幅に低下します)。
- **MinuteCount()そして時間数()はほぼ同一です。**重複したコードを共有できればコードは小さくなります。冗長コードは比較的複雑であるため、この詳細は特に重要です。(難しいコードはすべて1か所に限定した方がよいでしょう。)

読みやすいバージョン

のコードMinuteCount()そして時間数()単一の定数だけが異なります(60対3600)。明らかなリファクタリングは、両方のケースを処理するヘルパー メソッドを導入することです。

```

クラス MinuteHourCounter {
    list<Event> イベント;

    整数カウント開始(time_t カットオフ) {
        int カウント = 0;
        for (list<Event>::reverse_iterator rit = events.rbegin();
            rit != events.rend(); ++rit) { if (rit->
                時間 <= カットオフ) {
                    壊す;
                }
                カウント += rit->カウント;
            }
        戻り数;
    }
}
```

公共：

```

void Add(int count) {
    events.push_back(イベント(カウント, 時間()));
}

int MinuteCount() {
    戻るカウント開始(時間() - 60);
```

```

    }

    int HourCount() {
        戻るカウント開始(時間() - 3600);
    }
};

```

この新しいコードについては、指摘する価値のあることがいくつかあります。

まず、それに注目してくださいCountssince()絶対的なものを取る切り落とす相対ではなくパラメータ秒前価値（60または3600）。どちらの方法でもうまくいきましたが、この方法Countssince()もう少し簡単な仕事があります。

次に、イテレータの名前を次のように変更しました。私にりった。名前私整数インデックスによく使用されます。私たちはその名前を使用することを検討しましたそれ、これはイテレータでは一般的です。しかし、この場合、逆行するこの事実は、コードの正確性にとって非常に重要です。変数名に接頭辞を付けることにより、次のようなステートメントに心地よい対称性を追加します。rit != events.render()。

最後に条件を抽出しましたrit->時間 <= カットオフの外へのためにループして別個にしましたもし声明。なぜ？なぜなら、次の形式のforループは「伝統的」だからです。for(開始; 終了; 前進)が最も読みやすいです。読者は「すべての要素を確認する」ということをすぐに理解でき、それ以上考える必要はありません。

パフォーマンスの問題

コードの見た目は改善されました、この設計には2つの重大なパフォーマンス上の問題があります。

1. それはただ成長し続けます。

クラスはこれまでに見たすべてのイベントを保持します。無制限の量のメモリを使用します。理想的には、分時カウンター1時間以上経過したイベントは不要になったので、自動的に削除する必要があります。

2. MinuteCount()そして時間数()遅すぎます。

方法Countssince()かかりますの上) 時間、場所n関連する時間枠内のデータ ポイントの数です。を呼び出す高性能サーバーを想像してください。追加 () 1秒間に何百回も。すべての電話時間数()100万ものデータ ポイントをカウントする必要があります。理想的には、分時カウンター分けておいたほうがいい分_カウントそして時間数を呼び出すたびに最新の状態に保たれる変数追加 ()。

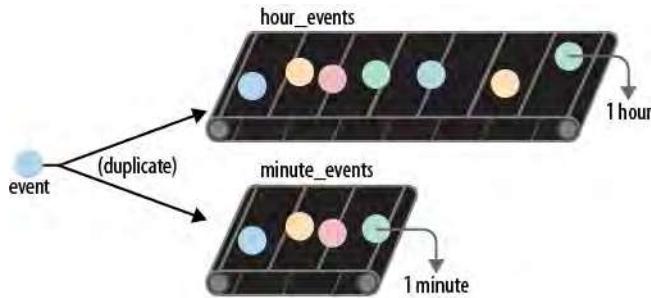
試み 2: コンベアベルトの設計

前述の問題の両方を解決する設計が必要です。

1. 不要になったデータを削除します。
2. 事前計算を維持する分_カウントそして時間数最新の合計。

その方法は次のとおりです。リストベルトコンベアのように。新しいデータが一方の端に到着すると、合計に追加されます。そして、データが古すぎると、もう一方の端から「外れて」しまい、合計から差し引かれます。

このコンベアベルトの設計を実装するには、いくつかの方法があります。1つの方法は、2つの独立した状態を維持することですリストs、1つは過去1分間のイベント、もう1つは過去1時間のイベントです。新しいイベントが入ったら、両方のリストにコピーを追加します。



この方法は非常に簡単ですが、すべてのイベントのコピーが2つ作成されるため、非効率的です。

別のある方法は、2つを維持することですリストs、イベントは最初に最初の部分に入ります。リスト(「直前のイベント」)、これが2番目のイベントに反映されます。リスト(「過去1時間のイベント(直前ではない)」)。



この「2段階」のコンベアベルト設計の方が効率的だと思われる所以、これを実装してみましょう。

2段コンベアベルト設計の実装

クラスのメンバーをリストすることから始めましょう。

```
クラス MinuteHourCounter {
    list<イベント> 分_イベント;
    list<イベント> 時間イベント; // minutes_events に含まれない要素のみが含まれます

    int 分_カウント;
    int 時間数; // 過去1分を含む、過去1時間のすべてのイベントをカウントします
};
```

このコンベアベルト設計の核心は、時間の経過に応じてイベントを「シフト」できることです。イベントはから移動します分_イベントに時間イベント、そして分_カウントそして時間数更新される

それに応じて。これを行うには、という名前のヘルパー メソッドを作成します。ShiftOldEvents()。このメソッドを取得したら、クラスの残りの部分は非常に簡単に実装できます。

```
void Add(int count) {
    const time_t now_secs = time(); シフ
    トオールドイベント(今_秒);

    // 分のリストにフィードします (時間のリストではなく、後で行われます)
    minutes_events.push_back(Event(count, now_secs));

    分カウント += カウント;
    時間カウント += カウント;
}

int MinuteCount() {
    シフトオールドイベント (時間
    () ); 分数を返します;
}

int HourCount() {
    シフトオールドイベント (時間
    () ); 時間数を返します;
}
```

明らかに、私たちは汚い仕事をすべて延期しました。ShiftOldEvents():

```
// 古いイベントを検索して削除し、それに応じて、hour_count と minutes_count を減らします。
void ShiftOldEvents(time_t now_secs) {
    const int minutes_ago = now_secs - 60;
    const int 時間前 = 今秒 - 3600;

    // 1 分以上経過したイベントを 'minute_events' から 'hour_events' に移動します // (1 時間以
    上経過したイベントは 2 番目のループで削除されます。)
    その間 (! 分_イベント。空の () && 分_イベント.front().time <= minutes_ago) {
        時間イベント.push_back(分_イベント。フロント ());
        分数 -= 分_イベント.front().count; 分_イベント
        .pop_front();
    }

    // 1 時間以上前のイベントを 'hour_events' から削除します
    その間 (! 時間イベント。空の () && 時間イベント.front().time <= 時間前) {
        時間数 -= 時間イベント.front().count; 時間イベ
        ント.pop_front();
    }
}
```

もう終わりですか？

先ほど述べた2つのパフォーマンスの問題は解決されており、ソリューションは機能しています。多くのアプリケーションでは、このソリューションで十分です。しかし、欠点も数多くあります。

まず、設計が非常に柔軟性に欠けます。過去24時間のカウントを維持したいとします。そのためには、コードに多くの変更を加える必要があります。そして、おそらくお気づきかと思いますが、ShiftOldEvents()は、分と時間のデータ間の微妙な相互作用を伴う、かなり高密度の関数です。

次に、このクラスのメモリ使用量はかなり大きくなります。高トラフィックのサーバー呼び出しがあったとします。追加()1秒あたり100回。過去1時間のすべてのデータを保持しているため、このコードは最終的に約5MBのメモリを必要とします。

一般に、頻度が高いほど、追加()と呼ばれるほど、より多くのメモリが使用されます。実稼働環境では、予測不可能な大量のメモリを使用するライブラリは適切ではありません。理想的には、分時カウンター頻度に関係なく一定量のメモリを使用します追加()と呼ばれます。

試み3: 時間をかけた設計

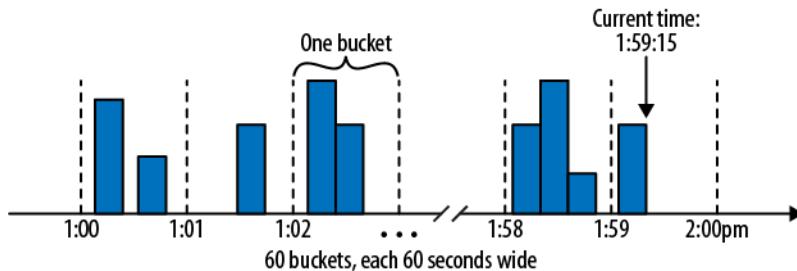
気づいていないかもしれません、以前の実装には両方とも小さなバグがありました。私たちが使用したtime_t整数の秒数を保存するタイムスタンプを保存します。この丸めのせいで、MinuteCount()実際には、呼び出したタイミングに応じて、59～60秒相当のデータが返されます。

たとえば、イベントが発生した場合、時間=0.99秒、その時間は四捨五入されますt=0秒。そして、あなたが電話した場合MinuteCount()で時間=60.1秒の場合、イベントの合計が返されます。t=1、2、3、...60。そのため、厳密には1分以内であっても、最初のイベントは見逃されます。

平均して、MinuteCount()59.5秒相当のデータが返されます。そして時間数()3599.5秒相当のデータが返されます(無視できるエラー)。

1秒未満の粒度の時間を使用することで、これらすべてを解決できます。しかし興味深いことに、分時カウンターそもそもそこまでの精度は必要ありません。私たちはこの事実を利用して新しいものを設計します。分時カウンターその方がはるかに高速で、使用するスペースも少なくなります。精度とパフォーマンスのトレードオフですが、それだけの価値は十分にあります。

重要なアイデアは、バケツ小さな時間枠内のすべてのイベントをまとめて、それらのイベントを1つの合計で要約します。たとえば、過去1分間のイベントを、それぞれ1秒幅の60個の個別のバケットに挿入できます。過去1時間のイベントを、それぞれ1分幅の60個の個別のバケットに挿入することもできます。



図に示すバケットを使用したメソッドMinuteCount()そして時間数()精度は60あたり1パーツであります、これは妥当な値です。*

より高い精度が必要な場合は、メモリ使用量を増やす代わりに、より多くのバケットを使用できます。しかし重要なことは、この設計ではメモリ使用量が固定されており、予測可能であるということです。

時間を割いた設計の実装

この設計を1つのクラスだけで実装すると、理解するのが難しい複雑なコードが大量に作成されることになります。代わりに、からのアドバイスに従います。[第11章、一度に1つのタスクを作成し、この問題のさまざまな部分を処理するための別のクラスを作成します。](#)

まず、単一の期間(過去1時間など)のカウントを追跡するための別のクラスを作成しましょう。それをと呼びますTrailingBucketCounter。基本的には以下のジェネリック版です
分時カウンター1つの期間のみを処理します。インターフェースは次のとおりです。

```
// 過去 N バケットの時間のカウントを保持するクラス。クラス
TrailingBucketCounter {
    公共：
        // 例: TrailingBucketCounter(30, 60) は、最後の 30 分間の時間を追跡します。トレーリングバケット
        トカウンター(int num_buckets, int secs_per_bucket);

        空所追加(int カウント、time_t now);

        // 最後の num_buckets 相当の時間の合計カウントを返します int トレーリン
        グカウント(今の時間);
    };
}
```

なぜだろうと不思議に思うかもしれません追加()そしてTrailingCount()現在時刻が必要です(今の時間_t)引数としてこれらのメソッドが現在の値を計算するだけであればもっと簡単ではないでしょうか。時間()彼ら自身？

*前のソリューションと同様に、最後のバケットは平均して半分しか埋まりません。この設計では、60個ではなく61個のバケットを保持し、現在の「進行中」バケットを無視することで、過小評価を修正できます。しかし、これによりデータが部分的に「古く」なります。より良い修正は、進行中のバケットを最も古いバケットの補足部分と組み合わせて、偏りのない最新のカウントを取得することです。この実装は読者のための演習として残されています。

奇妙に思えるかもしれません、現在の時刻で通過することにはいくつかの利点があります。まず、それはトレーリングバケットカウンター「クロックレス」クラス。一般にテストが容易で、バグが発生しにくいです。次に、すべての呼び出しを保持します。時間 () 内部分時間カウンター。時間に敏感なシステムでは、時刻を取得するためのすべての呼び出しを 1 か所にまとめられれば便利です。

仮定するとトレーリングバケットカウンターすでに実装されており、分時カウンター実装は簡単です：

```
クラス MinuteHourCounter {
    TrailingBucketCounter minutes_counts;
    TrailingBucketCounts 時間カウント;

    公共：
        MinuteHourCounter():
            minutes_counts(/* num_buckets = */ 60, /* secs_per_bucket = */
                           1), hour_counts(/* num_buckets = */ 60, /* secs_per_bucket = */ 60) {
        }

        void Add(int count) {
            time_t now = time(); 分_カウント。
            追加 (数えて、今) ; 時間数。追加
            (数えて、今) ;
        }

        int MinuteCount() {
            time_t now = time();
            分カウントを返します。トレーリングカウント (今) ;
        }

        int HourCount() {
            time_t now = time();
            時数を返します。トレーリングカウント (今) ;
        }
};
```

このコードは、はるかに読みやすく、柔軟性も高くなります。バケットの数を増やしたい場合 (精度は向上しますが、メモリ使用量も増加するため)、それは簡単です。

TrailingBucketCounter の実装

あとは実装するだけですトレーリングバケットカウンタークラス。もう一度、この問題をさらに細分化するためにヘルパークラスを作成します。

というデータ構造を作成します。コンベアキューその仕事は、基礎となるカウントとその合計を処理することです。のトレーリングバケットカウンタークラスは、オブジェクトを移動するタスクに集中できます。

コンベアキュー経過時間に応じて。

ここにありますコンベアキューインターフェース：

```
// 最大数のスロットを備えたキュー。古いデータが最後に「落ちる」。クラス
ConveyorQueue{
    コンベアキュー(int max_items);
```

```

// キューの後ろの値をインクリメントします。空所戻る
に追加(整数カウント);

// キュー内の各値は 'num_shifted' だけ前方にシフトされます。// 新しい
項目は 0 に初期化されます。
// 最も古いアイテムが削除されるため、<= max_items になります。
空所シフト(int num_shifted);

// 現在キューにあるすべてのアイテムの合計値を返します。整数合計();
};


```

このクラスが実装されると仮定すると、どれだけ簡単かを見てください。トレーリングバケットカウンター以下を実装することです:

```

クラス TrailingBucketCounter {
    ConveyorQueue バケット。
    const int secs_per_bucket;
    time_t last_update_time; // Update() が最後に呼び出された時刻

    // 経過した時間のバケット数を計算し、それに応じて Shift() を実行します。void
    Update(time_t now) {
        int current_bucket = 今 / 秒ごとのバケット;
        int last_update_bucket = last_update_time / secs_per_bucket;

        バケット.Shift(current_bucket - last_update_bucket);
        last_update_time = 今;
    }

    公共：
    TrailingBucketCounter(int num_buckets, int secs_per_bucket) :
        バケット(num_buckets),
        secs_per_bucket(secs_per_bucket) {
    }

    void Add(int count, time_t now) {
        今すぐアップデート();
        バケット.AddToBack(カウント);
    }

    int TrailingCount(time_t now) {
        今すぐアップデート();
        戻るバケット.TotalSum();
    }
};


```

これは 2 つのクラスに分類されます (トレーリングバケットカウンターそしてコンベアキュー) で議論したことの別の例です [第11章、一度に 1 つのタスク](#)。なしでもできたでしょう
コンベアキューすべてを内部に直接実装しました TrailingBucketCounter。ただし、この方法では、コードを理解しやすくなります。

ConveyorQueueの実装

あとは実装するだけですコンペアキュークラス：

```
// 最大数のスロットを持つキュー。古いデータは末尾からシフトされます。クラスConveyorQueue
{
    queue<int> q;
    int max_items;
    int total_sum; // q のすべての項目の合計

    公共：
    コンペアキュー(int max_items) : max_items(max_items), total_sum(0) {}

    整数合計() {
        total_sum を返します。
    }

    空所シフト(int num_shifted) {
        // シフトされた項目が多すぎる場合は、キューをクリアしてください。
        if (num_shifted >= max_items) {
            q = queue<int>(); // キューをクリア
            total_sum = 0;
            戻る;
        }

        // 必要なゼロをすべてpushします
        while (num_shifted > 0) {
            q.push(0);
            num_shifted--;
        }

        // 余分なアイテムをすべて削除します。
        while (q.size() > max_items) {
            total_sum -= q.front();
            q.pop();
        }
    }

    空所戻るに追加(整数カウント) {
        if (q.empty()) Shift(1);           // q に少なくとも 1 つの項目があることを確認してください。
        q.back() += カウント;
        total_sum += カウント;
    }
};
```

これで完了です。私たちには、分時カウンターこれは高速でメモリ効率が高く、さらに柔軟性が高くなります。トレーリングバケットカウンターそれは簡単に再利用できます。たとえば、より汎用性の高いものを作成するのは非常に簡単です。最近のカウンター最終日や過去 10 分など、幅広い間隔をカウントできます。

3つのソリューションの比較

この章で検討したソリューションを比較してみましょう。次の表は、コード サイズとパフォーマンス統計を示しています(高トラフィックのユースケースを 100 と仮定)追加 () /秒):

解決	コード行	時間あたりのコストCount()	メモリ使用量	HourCount() でのエラー
素朴な解決策	33	O(1 時間あたりのイベント数) (約360万)	無限の	3600 あたり 1 パーツ
コンベアベルトの設計	55	O(1)	O(1 時間あたりのイベント数) (~5MB)	3600 あたり 1 パーツ
時間を重視した設計 (パケット60個)	98	O(1)	O(パケット数) (~500バイト)	60個につき1部

最終的な 3 クラスのソリューションのコードの総量は、他のどの試みよりも多いことに注意してください。ただし、パフォーマンスははるかに優れており、設計はより柔軟です。また、各クラスを個別に記述すると、非常に読みやすくなります。これは常に前向きな変化です。読みやすい 100 行の方が、読みにくい 50 行よりも優れています。

場合によっては、問題を複数のクラスに分割すると、(1 クラスのソリューションでは発生しない) クラス間の複雑さが発生する可能性があります。ただし、この場合、あるクラスから次のクラスへの単純な「線形」使用 チェーンがあり、エンド ユーザーに公開されるクラスは 1 つだけです。全体として、この問題を解決することで得られる利点により、これは成功です。

まとめ

決勝に至るまでの手順を振り返ってみましょう分時カウンターデザイン。このプロセスは、他のコード部分がどのように進化するかの典型的なものです。

まず、単純なソリューションのコーディングから始めました。これにより、速度とメモリ使用量という 2 つの設計課題を認識することができました。

次に試したのは「ベルトコンベア」のデザイン。この設計により速度とメモリ使用量は改善されましたが、高性能アプリケーションにはまだ十分ではありませんでした。また、この設計は非常に柔軟性がありませんでした。他の時間間隔を処理できるようにコードを調整するのは多大な作業が必要でした。

私たちの最終設計では、サブ問題に分割することで以前の問題を解決しました。以下に、作成した3つのクラスをボトムアップ順に示し、それぞれが解決した副問題を示します。

コンペアキュー

「シフト」可能で合計を維持できる最大長のキュー

トレーリングバケットカウンター

を移動しますコンペアキュー経過時間に応じて、単一(最新)の時間間隔のカウントを指定された精度で維持します

分時カウンター

単純に2つ入ってるトレーリングバケットカウンターや、1つは分カウント用、もう1つは時間カウント用

付録

参考文献



私たちは、実際に何が機能するかを理解するために、製品コードからの何百ものコード例を分析してこの本を作成しました。しかし、私たちはこの追求に役立つ多くの本や記事も読みました。

さらに詳しく知りたい場合は、次のリソースを参照してください。以下のリストは決して完全ではありませんが、始めるには適しています。

高品質のコードの書き方に関する書籍

コードコンプリート: ソフトウェア構築の実践ハンドブック、第2版、Steve McConnell著 (Microsoft Press、2004)

コード品質などを含むソフトウェア構築のあらゆる側面について、厳密でよく研究された本。

リファクタリング: 既存のコードの設計を改善する、Martin Fowlerらによる。(アディソン・ウェスリー・プロフェッショナル、1999)

コードの段階的改善の哲学について説明し、さまざまリファクタリングの詳細なカタログと、問題を壊す可能性を減らしてこれらの変更を行うための手順が記載されている素晴らしい本です。

プログラミングの実践、ブライアン・カーニハンとロブ・パイク著 (Addison-Wesley Professional、1999年)

デバッグ、テスト、移植性、パフォーマンスなどのプログラミングのさまざまな側面を、さまざまなコーディング例とともに説明します。

実践的なプログラマー: ジャーニーマンからマスターへ、アンドリュー・ハントとデイビッド・トーマス著 (Addison-Wesley Professional、1999年)

多くの優れたプログラミングとエンジニアリングの原則を短いディスカッションにまとめたコレクション。

クリーンなコード: アジャイルソフトウェアクラフトマンシップのハンドブック、ロバート・C・マーティン著 (ブレンティス・ホール、2008年)

私たちの本に似た本(ただしJavaに特化した本)。エラー処理や同時実行性などの他のトピックについても説明します。

さまざまなプログラミングトピックに関する書籍

JavaScript: 良い部分、ダグラス・クロックフォード著 (オライリー、2008年)

この本は、明確に読みやすさを重視しているわけではありませんが、私たちの本と同様の精神を持ついると信じています。それは、エラーが発生しにくく、推論が容易なJavaScript言語のクリーンなサブセットを使用することです。

効果的なJava、第2版、ジョシュア・ブロック著 (ブレンティス・ホール、2008年)

Javaプログラムを読みやすくし、バグをなくすことについての驚異的な本。これはJavaに関するものですが、原則の多くはすべての言語に当てはまります。強くお勧めします。

デザインパターン: 再利用可能なオブジェクト指向ソフトウェアの要素、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 著 (Addison-Wesley Professional、1994 年)

ソフトウェアエンジニアがオブジェクト指向プログラミングを語るための共通言語「パターン」についての原書。一般的で便利なパターンのカタログとして、プログラマーが難しい問題を初めて自分で解決しようとするときに起こりがちな落とし穴を回避するのに役立ちます。

プログラミングパール、第 2 版、Jon Bentley 著 (Addison-Wesley Professional、1999 年)

実際のソフトウェアの問題に関する一連の記事。各章には、現実世界の問題を解決するための優れた洞察が含まれています。

高パフォーマンスの Web サイト、スティーブ・サウザーズ著 (オライリー、2007)

この本はプログラミングに関する本ではありませんが、コードをあまり書かずに Web サイトを最適化するさまざまな方法が説明されているため、注目に値します ([第13章 コードの記述量を減らす](#))。

ジョエルがソフトウェアについて語る: そして多様性について、そして…、ジョエル・スボルスキー著

からの最高の記事の一部 <http://www.joelonsoftware.com/>。Spolsky はソフトウェアエンジニアリングのさまざまな側面について執筆し、多くの関連トピックについて洞察力に富んだ見解を示しています。必ずお読みください。絶対にやってはいけないこと、パート I、"そして "Joel テスト: より良いコードを作成するための 12 ステップ"

歴史的メモの書籍

しっかりしたコードを書く、スティーブ・マグワイア著 (マイクロソフト プレス、1993 年)

この本は残念ながら少し古くなってしまいましたが、コードをよりバグのないものにする方法に関する素晴らしいアドバイスで私たちに間違いなく影響を与えました。これを読むと、私たちが推奨している内容と多くの重複があることに気づくでしょう。

Smalltalk のベスト プラクティスパターン、ケント・ベック著 (ブレンティス・ホール、1996 年)

例は Smalltalk にありますが、この本には多くの優れたプログラミング原則が記載されています。

プログラミングスタイルの要素、Brian Kernighan と PJ Plauger 著 (Computing McGraw-Hill、1978 年)

「ものを書くための最も明確な方法」の問題を扱った最も古い本の 1 つ。例のほとんどは Fortran と PL1 にあります。

リテラシーのあるプログラミング、ドナルド E. クヌース著 (言語情報研究センター、1992 年)

私たちは、クヌースの次の声明に心から同意します。コンピューター何をすべきか、むしろ説明することに集中しましょう。人間私たちがコンピュータに何をさせたいのか」(p. 99)。ただし、注意してください。この本の大半はクヌースのことについて書かれています。ウェブキュメント用のプログラミング環境。ウェブは事実上、コードを脇に置いて、プログラムを文学作品として作成するための言語です。

を使用した後、ウェブ-派生システムを使用している私たち自身も、コードが常に変更されている場合(通常はそうなります)、いわゆる「読み書き可能なプログラム」を最新の状態に保つことは、私たちが推奨する方法を使用してコードを最新の状態に保つことよりも難しいと考えています。

記号

4xx HTTP レスポンス コード、144
5xx HTTP レスポンス コード、
144 ?: 条件式、73 ~ 74

あ

略語、使用した名前、19 抽象的な名前と具体的な名前、13 ~ 15 の頭字語、使用した名前、19
美学、34-43
コードを段落に分割、41 ~ 42 列の配置、38 ~ 39
ブロックにまとめられた宣言、40-41 の重要性、35
一貫性とコンパクト性のための改行、35-37
不規則性を解消する方法、37 ~ 38 コードの順序、39 ~ 40
個人のスタイル vs. 一貫性、42 vs. デザイン、34
Ajax、サーバーにデータを送信する、
112 `alert()` (JavaScript)、112
曖昧な名前、24
曖昧な代名詞、コメント付き、60 匿名関数、80
引数
名前による割り当て、63
条件文での順序、70
配列、値を削除する JavaScript 関数、95
`assert()` メソッド、154-155 `assertEqual()` メソッド (Python)、155 代入、if ステートメント 内、71 属性、名前のエンコーディング、16-17
Web ページ ユーザーの承認、PHP 用、132

B

ベック、ケント、Smalltalk ベスト プラクティス パターン、119

開始と終了、包含/排他範囲の使用、26-27
全般的なコメント、55 ブロックスコープ、100
コードのブロック、宣言は 40 ~ 41 個のブール値にまとめられています
の名前、27
式の書き換え、85 Boost C++
ライブリ、154 ボトムアップ プログラミング、114 Brechner, Eric、96
小さな時間枠でイベントをバケット化、174 ~ 178 個のバグ
コメントと、50
1 ずつオフ、25

C

C プログラミング言語、変数定義
場所、101-102
C#、クリーンアップ コードの構造化イディオム、76
C++
ブロックスコープ、100
ファイルを読み取るためのコード、112
if ステートメントのスコープ、98
名前付き関数/パラメータのインライン コメント、64
マクロ、90
式の単純化、90 標準ライブラリ、28
クリーンアップ コードの構造化イディオム、76
キャッシュ、追加、141
大文字化、名前の使用、20 暗号
クラス (Python)、117
クラスインターフェイス、分/時カウンター用、166 ~ 169
クラス メンバー変数、97
クラスメンバー、アクセスを制限、98
クラス
複数の 179 の名前、8 からのクラス間の複雑さ
クリーンアップ コード、構造化イディオム、76

インデックスを改善するためのご提案をお待ちしております。に電子メールを送信します Index@oreilly.com。

巧妙なコード、混乱による、86

Clip() 関数、24

JavaScript のクロージャー、99

コード、viii、150

(テストコードも参照) 重複

を排除、38 個の領域を分

離、129 個が少ない vs. 多

い、3 個

複数のタスクと単一のタスク、122 ~

130 の良い点、2

冗長、170

未使用の削除、143

テストしやすい開発、160 思考を
変換する、132~138 理解できる、

2

書く量を減らす、140~145

コードベース

汎用コード用のディレクトリ、114 小さく
保つ、142

列の配置、38 ~ 39 コマンドライ

ン フラグ、名前、14 コメント、

3、46 ~ 57、60 ~ 65

あいまいな代名詞、60 全体

像、54

コードの欠陥の説明、50~51 コ

ンパクトさ、60

定数の説明、関数の動作の説明 51 個、情報

量の多い単語 61 個、コーナーを説明するた
めの 64 個の入出力例

ケース、61~62

コードインに関する洞察、コードのインテ
ント ステートメント 50、62 ~ 63 の並び、
36 ~ 37

分/時カウンタの改善、167 ~ 169 名前付き関数

パラメータ、63 ~ 64

名前と、49

精度、60、61

目的、46

読者の視点、51 概要、42、

55

何を、なぜ、どのように、56 使用

すべきではない場合、47~49 ライ

ターズブロック、56

複雑なアイデア、説明する能力、132

複雑さ、142

複雑なロジック、分解、86 ~ 88 具体名と

抽象名、13 ~ 15 条件式 (?:)、73 ~ 74 条

件、引数の順序、70 一貫したレイアウト、

34

改行、35 ~ 37 個人ス

タイル vs.、42 定数、

103

説明するコメント、51

コンストラクター、名前の書式設定、21

continue ステートメント、75

制御フロー、70~81

?: 条件式、73 ~ 74 関数からの早期

復帰、75 ~ 76 変数の削除、96

以下の実行フロー、80 goto ス

テートメント、76

ネスト、77~79

ConveyorQueue インターフェース、176

実装、178

JavaScript の Cookie、116 コピー コ

ンストラクター、デフォルト、13

特殊なケース、コメントの入出力例

図解、61~62

松葉杖コメント、49

D

ダッシュ、名前付き、20

データベース テーブル、結合するプログラム、134~
137 ドモルガンの法則、85

ブロックにまとめられた宣言、40 ~ 41 デフ
ラグ コード、122

未使用のコードの削除、143 デザイン、美
学との比較、34 開発時間、スイートス
ポット、162 Python の辞書、144

機密情報、117

DISALLOW_COPY_AND_ASSIGN マクロ、14

DISALLOW_EVIL_CONSTRUCTOR マクロ、13

do-while ループ、回避、74~75

DRY (Don't Reply Yourself) 原則、89 重複
コードの排除、38

E

Eclipse、単語補完コマンド、19 Emacs、単語
補完コマンド、19 終了、包含/排他範囲の使
用、26 ~ 27 エラー メッセージ

手作り、155~156

可読性、154~156

例外、80

実行フロー、以下、80 ユーザーの期待、
一致、27 ~ 28 変数の説明、84

表現

ブレイクダウン、84~91

複雑なロジック入力、86~88

単線と複数回線、3 短絡口

ジックの乱用、86 簡素化、90

外部コンポーネント

テストの問題、161

抽出、110

(サブ問題のコード抽出も参照) オブジェクトからの値、124～128

F

偽、27
機能、実装しない決定、140 ファイルの内容、読み取り、112
Filter() 関数、24
findClosestLocation() の例、110～111 の最初と最後、使用する範囲を含む、26
FIXME: マーカー、50
実行の流れ、以下、80 for ループ、170、171
内部のネストの削除、78-79 名前のフォーマット、意味、20-21
format_pretty() 関数、113
ファウラー、マーティン、リファクタリング: 設計の改善既存のコード、119
関数ポインタ、80
機能、プロジェクト固有、115 関数
匿名、80
行動の説明に関するコメント、61 からの早期復帰、75～76、78
コードを個別の 110～118 個の名前に抽出する、8
ラッパー、116
可読性の基礎理論、3

G

汎用コード、112～114
作成、114
一般名、10～12
get() メソッド、ユーザーの期待、27 グローバル
スコープ、JavaScript、100
グローバル変数
避ける、97
テスト容易性、161
グーグル
DISALLOW_EVIL_CONSTRUCTOR マクロ、オープンソースの 14 の書式設定規則
プロジェクト、20～21
ゲームズ・ゴズリング、104
goto ステートメント、76

H

ハック: マーカー、50
ヘルパー・メソッド、37, 130
テストコード内の名前、159
分/時カウンターの ShiftOldEvents()、173 テスト
コードのクリーンアップ、151
高レベルのコメント、55
HTML タグ、id または class 属性名、21
HttpDownload オブジェクト、128

ハンガリー語表記、17

私

if ステートメント
内部の割り当て、71
個別の処理、127-128 イン
デックスの名前、12
引数の順序、70 C++ の
スコープ、98
if/else ブロック、順序、72～73 個の不变データ
型、104 個の実装機能、そうでない決定、140 個の包含範囲、最初と最後の for、26 個の包含/排他的範
囲、開始と終了、26～

27

インデックス、名前、12 個の情報量の多い単語、コメ
ント付き、64 個のインライン コメント、名前付き関数
パラメータ、
64
入力値、テストに適したものを選択、156～158
の入力/出力コメントの例、説明
コーナーケース、61-62
IntelliJ IDEA、単語補完コマンド、19 インタ
フェイス
再形成、117
既存の簡素化、116
中間結果変数、消去、95、101、
105
コード領域の分離、129

J

ジャワ
ブロックスコープ、100-101
名前付き関数パラメータのインライン コメント、
64
クリーンアップ コードの構造化イディオム、76
JavaScript
アラート()、112
クッキー、116
findClosestLocation() の例、110-111 名前の書式設定、21
配列から値を削除する関数、95 グローバル
スコープ、100
ネストされたスコープなし、100～
101 または演算子、86
プライベート変数、99 jQuery
JavaScript ライブリ、133
jQuery ライブリ関数、名前の書式設定、21

L

最後、包含範囲を使用、26 ライ
ブリ、116
正規表現の知識、133-134、143-
144 正規表現、153

制限、名前、コード内の 25
の改行、35～37
コード行数の最小化、時間要件の最小化、
73
list::size() メソッド、ユーザーの期待、28
Python のリスト、144
論理
複雑な内容を分解する、86～88 明確
な説明、132
ループ反復子、12
ループ、内部のネストを削除、78～79

M

マクロ (C++)、90
一致するデータベース行、検索する Python コード、135～
137
最大、包括的な制限の場合、25
メモリリーク、14
記憶要件、174 精神的な荷
物、67
乱雑なコード、コメント、50 分、包括的な制限、
25 ミニ言語、カスタム実装、152～153 分/時カウ
ンター、166～180

クラスインターフェイス、166～
169 コメント、167～169
ソリューションの比較、179 コンペア
ベルトの設計、171～174 単純なソ
リューション、169～171
パフォーマンスの問題、171 タイムバケット化
された設計、174～178 TrailingBucketCounter
の実装、176～
177

N

名前付き関数パラメータのコメント、63～64
の名前
の頭字語または略語、19 誤解を避け
るため、24～31 ブール値、27

コメントと、49
具体と抽象、13～15 エンコード属性、
16～17 複数の候補の評価、29～31 意
味の書式設定、20～21 汎用、10～12

8、16～17 の情報、18～20
の長さ
制限、25
ループ反復子のオプション、12 測定単位、16
MinuteHourCounter クラスの改善、167 Python
引数の割り当て、63

単語の特異性、および 8～10

テスト関数の場合、158～159 if/else の負の場
合と正の場合、72～73 ネスティング、77～79

蓄積、77
早期復帰による削除、78 ルー
プ内削除、78～79 非決定的動
作、161

O

1 つずつずれるバグ、25
OpenBSD オペレーティングシステム、ウィザード モード、29 またはオ
ペレーター、86
コードの順序、39～40

P

段落、コードの分割、41～42 パフォーマ
ンスと精度、174 個人のスタイルと一貫
性、42 他人の視点、169

PHP
ファイル内容の読み取り、112
Web ページのユーザー認証、132 の落と
し穴、コメントによる予測、53～54 の平易
な英語
コードの説明 132 テストの説明 152 プレー
ンテキスト、名前のインジケーター 17 if/
else の肯定的なケースと否定的なケース 72～73
精度とパフォーマンスの比較 174

printf()、153
プライベート変数、JavaScript、99
の問題
コメント付きの予想、テストコード
の 53～54、150
製品開発、制限としてのテスト、162 プロジェク
ト固有の機能、115
プロトタイプ継承パターン、名前の評価
、29～31
エンティティの目的、名前の選択、および 10～12、10
Python
名前による引数の割り当て、63 Assert
ステートメント、155
一致するデータベース行を検索するコード、135～
137 ユーザーの機密情報を含む辞書、117 リスト
とセット、144
ネストされたスコープなし、
100 または演算子、86
ファイル内容の読み取り、112 クリーンアップ コードの構
造化イディオム、76 単体テスト モジュール名とテスト ×
ソッド名、159

Q

質問、コメント付きの予想、52

R

範囲

包括的、最初と最後の、26 包括的/排他的、開始と終了、26～27 可読性

エラー メッセージと、154～156
の基本理論、3つのテスト コード、
および、150～153
変数と、94～106

ファイル内容の読み取り、112 冗長性
チェック、コメントとして、63 冗長
コード、170

リファクタリング: 既存のコードの設計を改善する
(ファウラー)、119

正規表現

図書館、153

プリコンパイル、115

未使用のコードの削除、143

要件、質問および内訳、140～141 戻り値、名前、10

職務から早期に復帰、75～76

ネストの削除、78 逆反復

子、171

Ruby、または演算子、86

-- ローカルで実行するコマンドライン フラグ、14～15

S

範囲

グローバル、JavaScript の場合、C+
+ の if ステートメント 100、名前の
長さ 98、および、18

変数の縮小、97～102 セキュリ

ティ バグ、名前、および 17 Python の
セット、144

ShiftOldEvents() メソッド、173

短絡ロジックの悪用、86 シグナ

ル/割り込みハンドラー、80 コー

ドのシルエット、36

Smalltalk ベスト プラクティス パターン (Beck),
119 単語の特異性、名前の選択、および 8～10 のス
テートメント、内訳、89

静的メソッド、98

統計、増分、128～130 株式購入、記
録、134～137 ビジネス向け店舗検索、
140～141 Stroustrup、Bjarne、75

副問題コード抽出、110～118

findClosestLocation() の例、110～111 汎
用コード、112～114

プロジェクト固有の機能、115 既存
のインターフェースの簡素化、116

やりすぎ、117

ユーティリティコード、111～112

要約コメント、42、55

要約変数、84～85、89 「表面レ
ベル」の改善、5

T

タスク

オブジェクトからの値の抽出、124～
128 様式対單一、122～130
サイズ、123～124
UpdateCounts() 関数の例、128～130 一時変
数、94

三項演算子、73～74 テス

ト コード

最小限のステートメントの作成,
152 ヘルパー メソッド名、159 問題

の特定、150

可読性、150～153

テスト駆動開発 (TDD)、160 テスト、
150～163

CheckScoresBeforeAfter() 関数、153 適切な
入力値の選択、156～158 コード開発、およ
び、160
行き過ぎ、162 良い設計、161 問題
の特定、159～160 大きなインプッ
ト、157

複数の機能テスト、158 テスト機能
の名前、158～159 Web サイトの変
更、29

テキストエディター、単語補完コマンド、19

TextMate、単語補完コマンド、19 スレッド、
80

時間、コードを理解するための要件、時間に敏
感な 3 つのシステム、176

tmp 変数、代替、11 TODO:

マーカー、50

トップダウン プログラミング、114

TrailingBucketCounter クラス、176～177

true、27

タイプミス、検索する列の配置、39

U

アンダースコア、名前付き、20

Unix ツール、144

UpdateCounts() 関数、128～130

Web ページのユーザー認証、PHP 用、132 のユー
ザー情報、機密性の高い Python 辞書、

117

ユーザー、期待の一一致、27～28 ユー
ティリティ コード、抽出、111～112

V

値、オブジェクトから抽出、124～128 var

キーワード (JavaScript)、100

変数

クラスメンバー、97
消去、94–96
中間結果の削除、95、101、105 説明、84

グローバル、テスト容易性、161 可
読性への影響、94 ~ 106 の測定単位
の名前、16 定義の下への移動、101
~ 102 の名前、8

定義の順序、39 ~ 40 プライ
ペート、JavaScript で、99 ス
コーブの縮小、97 ~ 102 概
要、84 ~ 85
スワッピング、名前の選択時の選択、
11 一時的、94
ライトワنس、103–104、106
Vi、単語補完コマンド、19 仮想×
ソッド、80

W

Web ページ、ユーザー認証用の PHP、132 Web サー
バー、転送されたバイト数の追跡 (分/を参照)
時カウンター)
Web サイト、変更をテストするための実験、29
while ループ
引数の順序、70 対 do-
while ループ、75
単語補完コマンド、長い名前、および 19 ラッパー
関数、116
ライトワنس変数、103 ~ 104、106 ライ
ターズブロック、コメント、および 56

バツ

XXX: マーカー、50

著者について

サークルで育てられたものの、**ダスティン・ボズウェル**自身はアーチャーよりもコンピューターの方が得意であることに早くから気づいていた。ダスティンはカリフォルニア工科大学で学士号を取得し、そこでコンピューター サイエンスに夢中になり、修士号を取得するためにカリフォルニア大学サンディエゴ校に進学しました。彼は Google に 5 年間勤務し、Web クローリング インフラストラクチャを含むさまざまなプロジェクトに取り組みました。彼は数多くの Web サイトを構築しており、「ビッグ データ」と機械学習に取り組むことを楽しんでいます。ダスティンは現在、インターネットスタートアップ中毒で、サンタモニカの山々をハイキングしたり、新しい父親になったりして自由時間を過ごしています。

トレバー・フーシエは、Microsoft と Google で 10 年以上大規模なソフトウェア開発に取り組んできました。彼は現在、Google で検索インフラストラクチャのエンジニアを務めています。余暇には、ゲーム 大会に参加したり、SF を読んだり、妻が経営するファッショングループの新興企業の COO を務めています。Trevor は、カリフォルニア大学バークレー校で電気工学とコンピューター サイエンスの学士号を取得して卒業しました。

奥付

表紙画像はゲッティイメージズより引用。表紙のフォントはAkzidenz GroteskとOratorです。テキストのフォントはAdobeのMeridienです。見出しのフォントはITC Baileyです。