Game Programming Patterns

impress top gear

Game Programming Patterns

ゲーム プログラミング パターンズ

ソフトウェア開発の問題解決メニュー

Robert Nystrom = 著
武舎広幸 = 監訳
阿部和也＋上西昌弘 = 訳

パターンの背景／エッセンス／適用条件／サンプルコードを解説

デザインパターンの極意！

著者10年の経験と考察を集約

インプレス

※本文中に登場する会社名、製品名、サービス名は、各社の登録商標または商標です。
※本書の内容は原著執筆時点のものです。本書で紹介した製品／サービスなどの名前や内容は変更される可能性があります。
※本書の内容に基づく実施・運用において発生したいかなる損害も、著者、訳者、ならびに株式会社インプレスは一切の責任を負いません。
※本文中では ®、TM、© マークは明記しておりません。

■正誤表の Web ページ
正誤表を掲載した場合、以下の URL のページに表示されます。
http://book.impress.co.jp/books/1114101121_3

Copyright© 2014 by Robert Nystrom.
Japanese traslation rights arranged with Robert Nystrom through Japan UNI Agency,Inc.

2つの必須要素である信頼と時間を
与えてくれたミーガンに捧げる。

To Megan, for faith and time, the two essential ingredients.

謝　辞

本を書くとはどういうことか。私は本を書いたことがある人の話しか聞いたことがありませんでした。しかし、本を書くことの負担を本当に知っている人が、実は他にもいるのです――物書きと暮らす不運を背負っている人です。私がこの本を書き上げるのに使った時間は、私の妻ミーガンが非常に多忙な生活の中から必死の思いで作り出してくれたものです。食事の後片付けや子どもたちを風呂に入れることは「執筆」とは言えないかもしれません。でも、妻がそれをしてくれなければこの本は誕生しなかったのです。

私がこの本を構想し始めたのはエレクトロニック・アーツ社のプログラマーだったときです。会社は本の構想をどう扱ったらよいのかわからなかったようです。その一方でマイケル・マローン、オリビア・ナレット、リチャード・ウィフォールの各氏は、構想を支援し、最初の数章に対して詳細な、洞察に富んだフィードバックを与えてくれました。感謝しています。

本を半分ほど書いたところで、私は昔ながらの出版形態をとらないことに決めました。そうすれば編集者から内容に関する助言が得られなくなることはわかっていたのですが、10人以上の読者からのメールで、この本に望んでいる内容が伝えられていたのです。校正者も得られなくなりましたが、文章を改良するのに助けとなるバグレポートが250以上も寄せられていました[※1]。従うべき執筆計画はなくなってしまいましたが、章を1つ書き終わるごとに読者が背中を押してくれたので、書く気が失せることはありませんでした。

この本（原著）は「自費出版（self publishing）」の形をとっていますが、「集団出版（crowd publishing）」と呼ぶほうが当たっているでしょう。執筆は孤独な作業になりがちですが、私が孤独を感じることはありませんでした。この本を2年も放っておいた間でさえ、励ましは続いていました。後続の章を待っている人々がいるのだということを私が忘れてしまう――それを何十人もの人が許さなかったからこそ、執筆を再開して完成にまで至ることができたのです[※2]。

私にメールやコメントをくれた人、「よいね」をしてくれた人、お気に入りにしてくれた人、ツイートやリツイートをしてくれた人、私に連絡をくれた人、この本のことを友だちに話してくれた人、バグレポートを送ってくれた人、私の心はあなた方への感謝で一杯です。この本を書き上げることは私の人生の最大の目標の1つでした。それが達成されたのも皆さんのおかげです。ありがとう！

2014年9月6日　ロバート・ナイストロム

※1　原稿を整理してくれるコピー・エディターは得ることができました。まさに必要としていたときに現れたのがローレン・ブリーズで、素晴らしい仕事をしてくれました。

※2　コルム・スローンはこの本のすべての章を2回も注意深く読んで、すばらしいフィードバックを山のようにくれました。まったく善意でしてくれたことです。特別な感謝を捧げます。ビールを1杯（それとも20杯？）おごりましょう。

CONTENTS

▶Part I イントロダクション 1

0.1 既存の本について—— 4
0.2 デザインパターンとのつながり—— 5
0.3 この本の読み方—— 6
0.4 サンプルコードについて—— 7
0.5 これから進む道—— 8

第1章 アーキテクチャ、実行速度、ゲーム 9

1.1 ソフトウェアアーキテクチャとは何か—— 10
1.2 分離のコスト—— 13
1.3 実行効率と実行速度—— 14
1.4 悪いコードの良い点—— 15
1.5 バランスの保持—— 16
1.6 単純さ—— 17
1.7 うまくやるヒント—— 18

▶Part II デザインパターン再訪 19

第2章 コマンド 21

2.1 入力の設定—— 23
2.2 アクターへの指示—— 25
2.3 「取り消し」と「再実行」—— 28
2.4 クラスが良くて関数はダメ?—— 32
2.5 参考情報—— 33

第3章 フライウェイト 35

3.1 木があってこその森—— 36
3.2 1,000 個のインスタンス—— 39
3.3 フライウェイトパターン—— 39
3.4 木を生やす場所—— 40
3.5 パフォーマンス—— 45
3.6 参考情報—— 46

CONTENTS

第4章　オブザーバ　47

4.1　達成の認定── 48
4.2　実装── 50
4.3　「処理速度が遅すぎる」── 54
4.4　「動的メモリ確保の処理が多すぎる」── 55
4.5　残った課題── 60
4.6　現在のオブザーバ── 63
4.7　オブザーバの今後── 64

第5章　プロトタイプ　65

5.1　デザインパターン「プロトタイプ」── 66
5.2　言語の理論的枠組みとしてのプロトタイプ── 72
5.3　プロトタイプとデータモデリング── 77

第6章　シングルトン　81

6.1　シングルトンパターン── 82
6.2　使用する理由── 84
6.3　使用をためらう理由── 86
6.4　代替手段── 90
6.5　シングルトンに残されたもの── 96

第7章　ステート　97

7.1　みんなそうだった── 98
7.2　解決手段としての有限状態機械── 102
7.3　列挙と分岐── 103
7.4　「状態」のパターン── 106
7.5　各状態のクラス── 106
7.6　状態への委任── 107
7.7　状態のオブジェクトの存在場所── 108
7.8　入口処理と出口処理── 110
7.9　うますぎた話── 112
7.10　階層的状態機械── 114

7.11　プッシュダウン・オートマトン―― 116
7.12　どれほど有用なのか―― 118

▶Part Ⅲ シーケンスのパターン 119

第8章　ダブルバッファ　121
8.1　パターン誕生の背景―― 122
8.2　パターン―― 125
8.3　パターンの適用条件―― 125
8.4　パターン使用上の注意―― 126
8.5　パターンのサンプルコード―― 126
8.6　検討項目―― 135
8.7　参考情報―― 138

第9章　ゲームループ　139
9.1　パターン誕生の背景―― 140
9.2　パターン―― 143
9.3　パターンの適用条件―― 143
9.4　パターン使用上の注意―― 143
9.5　パターンのサンプルコード―― 144
9.6　検討項目―― 151
9.7　参考情報―― 154

第10章　更新メソッド　155
10.1　パターン誕生の背景―― 156
10.2　パターン―― 159
10.3　パターンの適用条件―― 159
10.4　パターン使用上の注意―― 160
10.5　パターンのサンプルコード―― 163
10.6　検討項目―― 168
10.7　参考情報―― 170

CONTENTS

▶Part IV ビヘイビアのパターン 171

第11章 バイトコード 173
11.1 パターン誕生の背景── 174
11.2 パターン── 179
11.3 パターンの適用条件── 180
11.4 パターン使用上の注意── 180
11.5 パターンのサンプルコード── 182
11.6 検討項目── 193
11.7 参考情報── 198

第12章 サブクラスサンドボックス 199
12.1 パターン誕生の背景── 200
12.2 パターン── 202
12.3 パターンの適用条件── 202
12.4 パターン使用上の注意── 202
12.5 パターンのサンプルコード── 203
12.6 検討項目── 206
12.7 参考情報── 212

第13章 型オブジェクト 213
13.1 パターン誕生の背景── 214
13.2 パターン── 218
13.3 パターンの適用条件── 218
13.4 パターン使用上の注意── 218
13.5 パターンのサンプルコード── 220
13.6 検討項目── 226
13.7 参考情報── 230

▶Part V 分離のパターン 231

第14章 コンポーネント 233
14.1 パターン誕生の背景── 234
14.2 パターン── 238
14.3 パターンの適用条件── 238
14.4 パターン使用上の注意── 238
14.5 パターンのサンプルコード── 239
14.6 検討項目── 249
14.7 参考情報── 254

第15章 イベントキュー 255
15.1 パターン誕生の背景── 256
15.2 パターン── 260
15.3 パターンの適用条件── 260
15.4 パターン使用上の注意── 261
15.5 パターンのサンプルコード── 263
15.6 検討項目── 271
15.7 参考情報── 274

第16章 サービスロケータ 275
16.1 パターン誕生の背景── 276
16.2 パターン── 277
16.3 パターンの適用条件── 277
16.4 パターン使用上の注意── 278
16.5 パターンのサンプルコード── 279
16.6 検討項目── 284
16.7 参考情報── 290

CONTENTS

▶Part Ⅵ 最適化のパターン 291

第17章 データ局所化 293
17.1 パターン誕生の背景── 294
17.2 パターン── 299
17.3 パターンの適用条件── 299
17.4 パターン使用上の注意── 300
17.5 パターンのサンプルコード── 300
17.6 検討項目── 311
17.7 参考情報── 316

第18章 ダーティフラグ 317
18.1 パターン誕生の背景── 318
18.2 パターン── 322
18.3 パターンの適用条件── 323
18.4 パターン使用上の注意── 324
18.5 パターンのサンプルコード── 326
18.6 検討項目── 330
18.7 参考情報── 332

第19章 オブジェクトプール 333
19.1 パターン誕生の背景── 334
19.2 パターン── 336
19.3 パターンの適用条件── 336
19.4 パターン使用上の注意── 337
19.5 パターンのサンプルコード── 340
19.6 検討項目── 346
19.7 参考情報── 350

第20章 空間分割　351
20.1　パターン誕生の背景―― 352
20.2　パターン―― 354
20.3　パターンの適用条件―― 354
20.4　パターン使用上の注意―― 354
20.5　パターンのサンプルコード―― 355
20.6　検討項目―― 365
20.7　参考情報―― 368

索引　369

Part I

イントロダクション

Introduction

・Chapter 1　アーキテクチャ、実行速度、ゲーム
Architecture, Performance, and Games

2台の中古TRS-80が空き教室に置いてありました。私と友人は小学校5年生のときに、そのマイクロコンピュータを使っていいと先生から言われました。さらに先生は私たちが興味を持つようにと、練習に使えるような簡単なBASICプログラムを印刷した紙の束もどこからか見つけてきて、渡してくれたのです。

カセットドライブが壊れていたので、コードを実行する場合にはいつも最初から慎重に入力しなければなりませんでした。

ですから、実行してみるのは主に次のような2、3行のプログラム[※1]でした。

```
10 PRINT "BOBBY IS RADICAL!!!"
20 GOTO 10
```

そんな短いプログラムでも誤りはつきものでした。プログラミングを習ったわけではないので、ちょっとした文法の誤りでも私たちにとっては理解の範囲を超えていました。プログラムが動かないことはしょっちゅうでしたが、そのたびに最初から入力し直したものです。

渡された紙束の最後にはすごいプログラムが付いていました。数ページがコードでびっしり埋まっているのです。入力してみようとするだけで勇気を振り絞る必要がありました。でも誘惑には勝てませんでした——リストの上に印刷してあるタイトルが「トンネルとトロール（北欧の伝説に登場する妖精）」だったのですから。どんなプログラムか見当もつきませんでしたが、ゲームだろうと思いました。自分でプログラムしたコンピュータゲームをするなんて、最高にかっこいいと思いませんか?

そのプログラムを動かすことはついにできないまま、1年後にはその教室が使えなくなりました（そのプログラムが単なるキャラクタージェネレータで、ゲーム本体ではないことに気づいたのは、後年私が実際にBASICがわかるようになってからのことです）。でも、賽は投げられたのです——これ以降、私はゲームプログラマーになりたいと思い続けたのですから。

10代のとき、私の家にQuickBASICを搭載したMacintoshがやって来ました。後からTHINK Cも来ました。その年の夏休みは、ほとんどゲームを作って過ごしました。独学なので進歩も遅く、作業も大変でしたから、なるべく簡単に動かせるもの——地図の中を移動するものやちょっとしたパズルなど——をやってみましたが、プログラムが大きくなるにつれ、どんどん難しくなりました[※2]。

最初は、プログラムをただ動かすだけでも大変です。そのうち、自分の頭の中に収まりきらないプログラムをどう書くかが問題になりました。『C++プログラムの書き方』といった本を読む代わりに、私はプログラムの構成法について書いた本を探し始めます。

数年後、ある友人が私に1冊の本を渡してくれました。それが『オブジェクト指向における再利用のためのデザインパターン』（以降、『…のためのデザインパターン』と略す場合あり）だったのです。私が10代のときから探していた本にとうとう出会ったのです。最初から最後まで一気に読み通しました。私は依

※1　コンピュータで願い事を何度も出力すれば、本当になるかもしれません（Bobbyは著者の名前Robertの愛称）。

※2　夏休みには、ルイジアナ州南部の沼地でヘビやカメを捕まえるのにも熱中しました。子どもの頃の夏があんなに暑いものでなかったなら、この本はプログラミングの本ではなく爬虫両生類学の本になっていたかもしれません。

然として自分のプログラム作りに悪戦苦闘していましたが、他にも苦しんでいる人々がいて、対策を見つけたのだと知っただけで大きな安心につながりました。とうとう道具が手に入った、素手で作業しなくてもよくなった、と思えました[※3]。

2001年になって、私は夢に見た職業に就きました。エレクトロニック・アーツのソフトウェアエンジニアです。本物のゲームのソースコードを見てプロがどのような仕事をしているのかを知ることができると思うとうれしくてたまりませんでした。「Madden Football」のような巨大なゲームのアーキテクチャはどのようなものだろう。異なるシステム間の相互作用はどのようになっているのか。1つのコードベースで複数のプラットフォームに対応するにはどのようにするのだろう。知りたいことはいくらでもあります。

ソースコードを見てみると、落胆と驚愕が私を同時に襲いました。グラフィックス、AI、アニメーション、視覚効果と素晴らしいコードばかりです。CPUの能力を極限まで引き出し、その引き出したものを利用し尽くす能力を持った人々がいることがわかりました。私が思い付きもしないようなことが、いとも簡単に実現されているのです。

でも、この素晴らしいコードのアーキテクチャはといえば、あらかじめきちんと設計されたものではありませんでした。機能を重視するあまり、組織化はそっちのけになっていたのです。モジュール間の結合は頻繁にみられました。新しい機能は、コードベースのどこにでも、入れられる場所なら、ところかまわず突っ込んでありました。私は落胆しました。『…のためのデザインパターン』を読んでいるとしても原書70ページ辺りの「シングルトン」より先は絶対読んでいないだろうというプログラマーが多いことがわかったのです。

もちろん現実はそんなに悪いものではありません。私が想像していたゲームプログラマーというのは、ホワイトボードに囲まれた象牙の塔の中で何週間も続けてアーキテクチャの詳細を物静かに討論する人々でした。それに対して私が目にしたコードは、厳しい納期に合わせて作業する人たちが書いた、現実世界のものなのです。皆最善を尽くしており、その「最善」は非常に良いものであることが私にも徐々にわかってきました。ゲームのコードを書く作業が長くなるにつれ、表面下に隠れた素晴らしいものに気付くことが多くなりました。

残念なことに「隠れた」というのは少し足りない表現かもしれません。コードの中に宝石が埋まっているのですが、みんなはその上を歩き回っているのに気が付かないのです。同僚がうまい方策を求めて苦心惨憺している一方で、実はその同じコードの中にぴったりの解決策の例が埋まっているというのを何度も経験しました。

この本は、そんな状況を打開するために書いた本です。ゲームのコードの中から私が探し出した最善のパターンを洗練して公開してあります。自分の時間を使うなら、同じことを繰り返すのではなく、新しいことを発明したいものです。

※3　その友人とは初対面でした。自己紹介が終わって5分経つか経たないかのうちに、私は彼のソファーに座って、その後の2時間あまり彼のことを無視して本を読みふけりました。現在の私の人付き合いのスキルは、そのときより少しはましになったと思いたいものです。

0.1 既存の本について

書店に行けば、ゲームプログラミングの本が10冊以上並んでいます。そこにまた1冊、この本を追加する意味は何でしょうか。私が見たところ、ゲームプログラミングの本は次の2つに分類されます。

- 特定の領域に特化した本——ゲーム開発の特定の領域に絞って、深く解説した本です。3Dグラフィックス、リアルタイムレンダリング、物理シミュレーション、人工知能、音声処理などの本があります。ゲームプログラマーがキャリアを重ねるにつれて特化していくような領域です。
- ゲームエンジンを扱う本——こちらの本はゲームエンジン全体にわたり、さまざまな構成要素をすべてカバーしようとする本です。特定のジャンルのゲームについて、完全なゲームエンジンを作成することを主眼としています。一番多いのが3Dシューティングゲームです。

私はどちらの本も好きですが、どちらにもカバーされていない領域があると思うのです。特定の領域を扱う本では、その領域のコードがゲーム全体のコードとどのように連携するのかがあまり書かれていません。物理シミュレーションやレンダリングの専門家にはなれるかもしれませんが、両方のコードをどのようにしたらうまく連携させられるのかは、本を読んでもわかりません。

その点、ゲームエンジン全体を扱う本では連携が扱われています。しかし、連携が強固すぎたり、特定の領域に偏りすぎたりしていることが多いのです。モバイル機器が普及し、気軽にゲームが楽しめるようになったために、多種多様なジャンルのゲームが作成されるようになりました。人気ゲームを移植していればすむ時代は終わったのです。1つのゲームエンジンだけを説明するような本では、自分の作りたいゲームがそのエンジンに適合しない場合には役に立ちません。

それに対し、私がこの本で提供したいのは、解決策のメニューのようなものです。この本の各々の章では、単独でコードに適応可能なアイデアを解説しています。ですから、作りたいゲームに役立つものをメニューから選んで組み合わせることができます※4。

※4　人気シリーズの『Game Programming Gems』も気に入ったものを組み合わせて適用できる点が似ています。

0.2　デザインパターンとのつながり

題名に「パターン」という言葉の入った本はどれも、「四人組（GoF:Gang of Four）」と呼ばれたエリック・ガンマ、リチャード・ヘルム、ラルフ・ジョンソン、ジョン・ブリシディーズの『オブジェクト指向における再利用のためのデザインパターン』とはっきりした関連を持っています[※5]。

この本の題名を『ゲームプログラミングパターン』としたからといって、四人組の本がゲームの世界では役に立たないと言うつもりはありません。まったく逆です。「Part II　デザインパターン再訪」では『・・・のためのデザインパターン』で解説されているパターンを数多く取り上げます。ただし、ゲームプログラミングにどうすれば応用可能かに力点を置いています。

その一方で、私はこの本がゲーム以外のソフトウェアにも応用可能だと思っています。本の題名を『続・デザインパターン』としてもよかったのですが、ゲームを扱ったほうが魅力のある例が示せると考えました。社員台帳や銀行口座についての本を書いても面白くありませんから。

そのようなわけで、この本で説明するパターンはどのようなソフトウェアにも有用ですが、とりわけゲームプログラミングで突き当たる次のような問題を解決するのに役立つと思います。

- ゲームのアーキテクチャにおいてはタイミングと順序が重要です。正しい順序かつ正しいタイミングですべてが進行しなければなりません。
- 開発のサイクルは非常に短くなっており、多くのプログラマーが多種多様なモジュールを次から次へと素早く仕上げていかなければなりません。しかも他のプログラマーの邪魔をしたり、コードベースのあちこちを汚し回ったりしてはいけません。
- すべてのモジュールの仕様が決まったら、相互作用を組み込まなければなりません。モンスターは主人公に噛み付きます。魔法の薬は混合が可能です。爆弾は敵も味方も同じように吹き飛ばします。こういった相互作用を実現するために、コードベースをスパゲッティにしてはなりません。
- そして、何と言ってもゲームで最も重要なのが実行速度です。開発対象とするプラットフォームの性能を誰が最大限引き出すか、ゲーム開発者たちは常に競争しているようなものです。マシンサイクルを１つでも２つでも短くできることが、高評価を得て何百万本も売れるゲームと、アニメーションがフレーム落ちしてレビュー担当者を怒らせるゲームの違いを生むのです。

※5　デザインパターンという考え自体は、すでに発刊されている本から得られたものです。問題に対する応用可能な解決法を説明するために、パターンを扱う言語を作成するという考えは、クリストファー・アレグザンダー（サラ・イシカワ、マレー・シルバースタインとの共著）による『パタン・ランゲージ』から来ています。この本は、「アーキテクチャ」といってもビルや住宅といった「建築（アーキテクチャ）」についての本ですが、著者たちは別の分野の解決策を表現するのにも同じ手法を用いてほしいと考えました。四人組がそれをソフトウェアで実現しようと試みたのが『・・・のためのデザインパターン』です。

0.3 この本の読み方

この本は、大きく3つの部分に分かれています。最初は導入と本の全体像の紹介です。この章と次の第1章がこれに当たります。

2つめの部分（第2章から第7章）は「デザインパターン再訪」で、四人組の本に掲載されたパターンをいくつか説明します。どの章も、パターンに私なりの工夫を加え、ゲームプログラミングとどのような関連があるのかを説明しています。

最後の部分（第8章から第20章）はこの本の核となる部分です。私が有用だと考えた13のデザインパターンを解説しています。パターンは「シーケンスのパターン」「ビヘイビアのパターン」「分離のパターン」「最適化のパターン」の4つのカテゴリに分けてあります。それぞれのパターンの説明は同じような構成になっており、この本をリファレンスとして使ったときに必要なものをすぐに見つけられるようにしてあります。

- 「パターン誕生の背景」セクションは、そのパターンが解決しようとしている問題と絡めてパターンの働きを簡単に説明します。このセクションを先頭にもってきたのは、取り組んでいる問題の解決の助けになるパターンを本の中から素早く見つけられるようにするためです。また、パターンを適用しようとする問題の例も説明します。「アルゴリズム」は細部まで決まったがっちりしたものですが、「パターン」は普通、具体的な形がなく、何か特定の問題に適用してはじめて形が現れます。例題なしでパターンを教えようとするのは、パン生地についての説明なしにパン焼きを教えようとするようなものです。このセクションでは後で美味しいパンを焼き上げるための生地を扱います。
- 「パターン」セクションは、前のセクションの説明を受けて、そこからパターンのエッセンスを抽出します。パターンの教科書的な説明が必要な場合はここを見てください。パターンについてよく知っていて、細かいことを確認したい場合には、このセクションが役に立ちます。
- パターンを説明するのに例題はたった1つしかありません。自分が抱えている問題にパターンが適用できるかどうかを知るにはどうすればよいのでしょうか。「パターンの適用条件」のセクションはどのような場合にそのパターンが役に立つか、どのような場合には使用を避けたほうが賢明かについて、ある種の指針を示しています。「パターン使用上の注意」のセクションは、パターンを適用したときに生ずる副作用やリスクについて指摘しています。
- 私もそうでしたが、きちんとした使用例から学びたいと思っている場合は「パターンのサンプルコード」セクションを参照してください。パターンを完全に実装したコードについて順に見ていきます。パターンの効果が的確にわかるはずです。
- パターンはアルゴリズムとは異なり、具体的なコードがありません。パターンを使うたびに異なるコードとして実装されます。「検討項目」では、実装法のバリエーションを検討し、パターンを適用する際に考慮すべき別解を示します。
- まとめとして、「参考情報」セクションではこの章で取り上げたパターンと他のパターンとの関係を示し、実際のオープンソースなどのコードでこのパターンを使っているものを挙げます。

0.4 サンプルコードについて

この本のサンプルコードはC++で記述してありますが、C++でなければパターンが役に立たないとか、C++が他の言語より優れているなどという気はまったくありません。どの言語でも同じように役立ちますが、一部のパターンはオブジェクトやクラスが使える言語のほうが実装しやすいでしょう。

私がC++を選んだ理由はいくつかありますが、第1の理由は商品として市場に出ているゲームはC++で書かれたものが最も多いためです。業界の標準語なのです。おまけに、C++の元になったC言語の文法はJava、C#、JavaScriptを始めとした数多くのコンピュータ言語の基礎となっています。仮にC++を知らなくても、ちょっと努力するだけでここにあるサンプルコードが理解できるでしょう。

この本はC++を学ぶ本ではありません。サンプルコードは可能なかぎり単純化してあり、C++の書き方の手本となるようなコードではありません。コードからは、そのコードの元になっている考え方を読み取るようにしてください。コードそのものを真似しないように。

特に、コードが「モダンな」(つまりC++11以降の)形式で書かれていないことに注意してください。標準ライブラリも使っていませんし、テンプレートもめったに出てきません。つまり、C++としては「悪い」コードなのです。単純化して、C、Objective-C、Javaといった他の言語を使っているプログラマーにも利用しやすいようにしようと思ったのです。

同じコードを繰り返したり、パターンに直接関係しないコードで紙幅を無駄にしたりしないために、サンプルコードは適宜省略してあります。省略する際には「…」を置いて、省略があることを示すようにしました。

何かの作業をして値を返す関数を例にとります。説明したいパターンに関連するのは戻り値だけであり、作業には関係がないとしましょう。その場合、サンプルコードは次のようになります。

```
bool update()
{
    // 何かの作業…
    return isDone();
}
```

0.5 これから進む道

パターンというのはソフトウェア開発の中でも常に変化し拡大し続けている分野です。四人組は自分たちが見い出したソフトウェアのパターンを記録に残し、広く伝えようとしました。この本は、彼らが始めたこのプロセスを継続するものであり、プロセスはこの本の出版後も続いていきます。読者の皆さんがプロセスの中心になるのです。自分で新たなパターンを開発したり、この本のパターンを改良したり（あるいは重大な欠陥を見つけたり）したときは、ソフトウェアのコミュニティに貢献する絶好の機会です。提案、修正など何でも結構です。この本の内容について、何かあったらぜひご連絡ください。

第１章

アーキテクチャ、実行速度、ゲーム
Architecture, Performance, and Games

デザインパターンについて解説を始める前に、ソフトウェアのアーキテクチャについて、またそれがゲームにどのように応用できるかについて、私がどのように考えているかを述べておいたほうがよいだろうと思います。そのほうがこの本についての理解が進むでしょう。少なくとも、デザインパターンとソフトウェアアーキテクチャがどれほどひどいものか（あるいは素晴らしいものか）という議論に巻き込まれた際に議論に加わるための材料は手に入ると思います[※1]。

1.1　ソフトウェアアーキテクチャとは何か

この本を最初から最後まで読み通しても、3Dグラフィックスの背後にある線形代数について、あるいはゲームの物理シミュレーションの背後にある微積分について、知ることはできません。AIの探索枝でα-β枝刈りを実行する方法や、音の再生時に室内の残響をシミュレートする方法も載っていません[※2]。

この本は、そういったことのどれにも属さないコードに関する本です。しかも、コードを書くことではなく、コードをまとめ上げること（構造・組織化）について解説しています。どのようなプログラムでも、たとえ「何でもかんでもmainメソッドの中に突っ込んで、後は実行あるのみ」といったものでも、コードは何らかの意味の構造があり組織化されています。ですから、どうすればコードがうまく組織化されるかについて話すほうが面白いと思うのです——良い構造と悪い構造はどのようにして見分けられるのでしょうか。

私はこの問題を5年も考え続けてきました。皆さんも直感でデザインの良し悪しを見分けることができるでしょう。あまりにひどいコードベースに悩まされて、「もうこのコードは安楽死させてやるしかない」と思ったこともあるのではないでしょうか[※3]。

なかには運が良い人も少しはいて、逆の経験をしています。非常に良いデザインが施されたコードを扱う機会に恵まれたのです。設備の完璧に整った豪華ホテルで、どんな気まぐれにも喜んで対応してくれるコンシェルジュに囲まれたようなコードベースです。両方のコードベースはどこが違うのでしょうか。

▶1.1.1　良いソフトウェアアーキテクチャとはどのようなものか

私が思うに、良いデザインとは、何かを変更したときにまるでプログラム全体がその変更を予想して作られていたかのように思えるデザインです。良いデザインならば、抱えていた課題が、2つ3つの関数呼び出しを加えるだけでものの見事に解決してしまい、コード全体の調和をほんのわずかでも乱すことはありません。

こう書くと聞こえは良いのですが、そのようなコードが自然にできるわけではありません。「部分的な変更があっても全体に影響が及ばないようなコードを書くようにするとよい」ということなのです。

※1　私は読者がどちらの側につくかを想定しているわけではありません。私は武器商人のようなもので、どちらの側の戦闘員にも売れる武器を持っているのです。

※2　この段落は、この本の宣伝としては最悪ですね。

※3　部分的にせよそのコードを書いたのは私たちなのですから、私たちのほとんどがその問題の一部に加担しているということを認めないわけにはいきません。

もう少し具体的に言いましょう。一番重要な点は、アーキテクチャとは変更を容易にするものだということです。コードには変更がつきものなのです。誰もコードをいじらないというのなら——コードが完璧だからということでも、あまりに劣悪なのでファイルを開くとエディタがけがれるということでも同じです——デザインの良し悪しに意味はありません。デザインの価値は、どのようにして容易に変更を受け付けるかにあるのです。変更されないプログラムは、スタートラインから一歩も踏み出さない走者のようなものです。

▶1.1.2　変更の実施法

新しい機能の追加、不具合の修正など、さまざまな理由で変更が必要になります。エディタを起動してコードを変更する前に、私たちはまず既存のコードが何をしているのかを理解する必要があります。もちろん全体を理解する必要はありませんが、変更に関連する部分については理解したうえで作業を進める必要があります[※4]。

この作業は無視されがちなのですが、プログラミングの中で最も時間を要する過程となるのが普通です。ディスクからRAMへの読み出しに時間がかかるといってもたかが知れています。2本の視神経を経由して大脳にデータを読み込むには非常に時間がかかるのです。

さまざまな要素の関連性を脳に叩き込んでから、少し考えて解法をひねり出します。試行錯誤を重ねることもあるでしょうが、比較的簡単にできることが多いでしょう。問題が理解できて、コード内で関連する部分がわかれば、実際に修正するコードがごくわずかで済むことは多いのです。

しかし、キーボード上にしばらく指を走らせてはEnterキーを叩き、画面にOKが表示されるまで繰り返せば完了……というわけにはいきません。

テスト[※5]を作成して実行し、変更したコードをレビューに送り出す前に、後片付けが残っています。

コードベースにいくらかのコードを追加したのなら、あちこちに影響が出て、関連した修正も行っているはずです。今回の変更が、次にコードを修正する人にとって支障になるという事態は避けたいものです。変更が非常に小さい場合を除いて、追加したコードがプログラム全体とスムーズにつながるように、何らかの整理と再構成を施しているのが普通です。再構成がきちんとできていればコードはきれいにつながっており、次にコードを見た人はどこを修正したかわからないはずです。

つまり、フローチャートで表すと、プログラミングは次の図のようになります（図1.1[※6]）。

※4　理解が必要なのですから、単に文字をOCRで読み込むのとは違います。

※5　そうです。「テスト」をするのです。ゲームのコードに対してユニットテストを書くのが大変なのは事実ですが、コードベースの中の大部分のコードがテスト可能であるのも事実です。ここで持論を力説するつもりはありませんが、もしテストの自動化をまだしていないなら、自動化することを考えてみてください。何度も繰り返して手作業で検証するよりも良い方法ではないでしょうか。

※6　このループに出口がないという事実は、ちょっと考えてみると危ないことにも思えます。

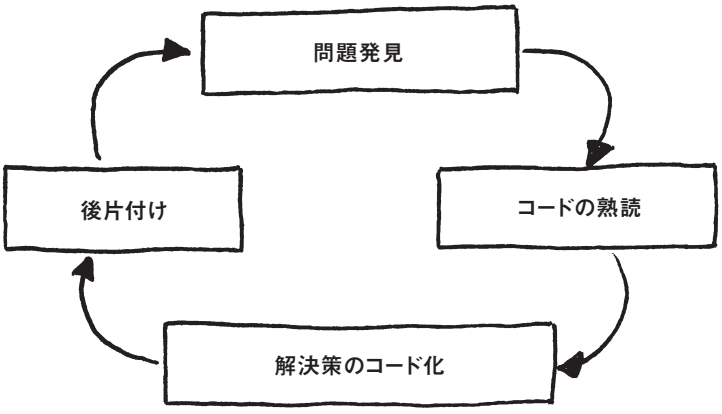

図 1.1　仕事の流れの概略

▶1.1.3　分離の効用

少しわかりにくいかもしれませんが、私はよいソフトウェアアーキテクチャの本領が発揮されるのは、コードを読んで理解するときだと思っています。コードを脳に読み込むのは非常に時間のかかる処理なので、読み込む量を減らす方策を見つけることに意義があります。この本では、Part V全体（第14章から第16章）を「分離のパターン」に充てています。『…のためのデザインパターン』もかなりの部分が分離についての説明になっています。

「分離」の定義はさまざまですが、2つのプログラムが「分離できていない（結合している）」というのは、一方を理解しないともう一方も理解できない状態だと私は考えています。2つが分離できれば、2つをそれぞれ独立して検討することができます。自分の抱えている問題に関連するのが一方のプログラムだけの場合、そちらだけ脳に読み込めばよく、もう1つは読み込まなくてよいのですから、これほどありがたいことはありません。

私にとって、ソフトウェアアーキテクチャを考える主要な目的は、この「次に進む前に頭に入れておかなければならない知識を最小にすること」です。

もちろんその後のことも問題になります。「分離」は「一方のプログラムを変更してももう一方は変更する必要がない状態」とも定義できます。何らかの変更が必要な場合について考えているわけですが、分離が進んでいればいるほど、変更がゲーム全体に及ぼす影響は小さくなるということです。

1.2 分離のコスト

とてもうまい話に聞こえますね。すべてを分離すれば何も悩まずにコードが書けるのです。どんな変更であっても1つか2つのメソッドを少し触るだけで済みます。そしてコードベースの深いところには触る必要もなく、変更を済ませれば、どこを変えたのかまったくわからないほどです。

そう思えばこそ、プログラマーは抽象化、モジュール化、デザインパターン、ソフトウェアアーキテクチャといったものに目の色を変えるのです。アーキテクチャの良くできたプログラムを扱う場合は、仕事をしていて楽しくなります。誰でも生産性が向上して嬉しくなります。良いアーキテクチャは生産性に大きな違いを生みます。どのように大きな影響を及ぼすか、強調してもしきれません。

とはいえ、世の中すべてそうなのですが、ただで手に入るものはありません。良いアーキテクチャを実現するにはきちんと学んだうえで真に努力することが要求されます。何か機能を変更したり実装したりする際にはいつも、それを既存のプログラムの中にしっくりと納めるための努力をしながら作業することが必要です。コードの構造を改善する努力を怠らず、開発サイクルの中で生ずる何千という小さな変更を行ってもその構造が乱れないように気を配らなければなりません[※7]。

このような観点からプログラムのどの部分を分離すべきかを考えたうえで、適切な場所に抽象化レイヤを導入しなければなりません。同様に、拡張性をどこに組み込むかも、将来的に変更がしやすくなることを考慮して決定しなければなりません。

こういった話は誰もが大好きです。未来の開発者の姿（あるいは未来の自分の姿）として思い描くのは、束縛がなく、強力で、まるで拡張を待ち受けているようなコードベースを扱っている姿です。「ゲームエンジン1つですべてが足りる」世界を想像しています。

しかし、ここから話がややこしくなります。コードに抽象化レイヤを導入したり、拡張性をサポートしたりするということは、後でその柔軟性を利用する必要が生じると考えているからにほかなりません。ゲームにコードを追加し、複雑化させれば、コード作成、デバッグ、保守（メンテナンス）にそれだけ時間がかかるのですから。

かけた手間は、予想どおりに後でそのコードを変更するのであれば、元が取れます。しかし、未来を予想することは困難であり、せっかくモジュール化したのに利用しないのであれば、それは直ちにお荷物になります。世話をしなければならないコードの量が増えてしまいます[※8]。

コードの構造化に凝りすぎると、コードベースのアーキテクチャはどんどん複雑化していきます。やがてインタフェースと抽象化レイヤだらけになり、プラグインシステム、抽象基底クラス、仮想メソッドがあふれかえり、あらゆる種類の機能拡張用コードポイントが埋め込まれます。

※7　この文の後段の「構造が乱れないようにする」という点には特別な注意が必要です。というのも、出だしは非常に美しいプログラムだったのが、「ちょっと直すだけ」というのを何度も繰り返しているうちに満身創痍になって命を落としたプログラムをたくさん見てきたからです。庭仕事と同じで、新しい苗木を植えるだけではだめなのです。雑草取りや剪定が欠かせません。

※8　「YAGNI」という言葉があります。「必要になるときなど来ない（You aren't gonna need it）」の頭文字です。将来何が必要になるかを考えなければならないときに、「要るときが来るかもしれない」という気持ちに対抗するための呪文として使われます。

抽象化の階層の中を探し回って実際に仕事をしているコードにたどり着くまでに果てしなく時間がかかります。変更をしなければならない場合には、もちろん助けになるインタフェースはあるのでしょうが、それを見つけられるかが問題です。理論的には、拡張作業を行う前に理解すべきコードの量は分離によって減少するはずなのですが、抽象化の階層が増えているので、それを理解するだけで脳の処理能力を使い切ってしまいます。

このようなコードベースを見た人は、ソフトウェアアーキテクチャ、特にデザインパターンに近づくことを避けようとします。下手に手を出すと、コードを書くことにとらわれすぎて、ゲームを世に出そうとしているのだという事実を見失ってしまうからです。世の中には拡張性の魔力に取り憑かれ、何のためのゲームエンジンなのかを忘れたまま、その作成と改良に何年も費やしている開発者が無数にいるのです。

1.3 実行効率と実行速度

ソフトウェアアーキテクチャと抽象化に対して、よく耳にするもう1つの批判は、実行速度が低下するというものです。特にゲームについてしばしば問題とされます。コードに柔軟性を持たせるためのパターンの多くは、仮想ディスパッチ、インタフェース、ポインタ、メッセージなどを利用していますが、すべて実行時にある程度のコストがかかります※9。

これには理由があります。ソフトウェアアーキテクチャの多くは、プログラムに柔軟性を持たせるためにあると書きました。その柔軟性により、変更の作業がより楽になります。そのために、プログラムの前提となる固定条件をなるべく少なくしています。ここで、あるコードが1つのクラスを対象にして動作しているとしましょう。そこにインタフェースを導入するのは、そのインタフェースを実装しているクラスであれば、どのようなクラスであっても、そのコードの対象にできるようにするためです。オブザーバ（Observer）やメッセージ（Message）といったデザインパターンを導入するのは、現在データをやり取りしている2つのクラスを、将来3つや4つに簡単に増やせるようにするためです。

それに対して、実行速度は固定条件と深く関わっています。最適化は固定条件を利用して進められます。敵が256個以上はいないと仮定して問題なければ、IDを1バイトで済ませることができます。1種類の具体的な型を対象とするメソッドを呼び出すのであれば、メソッドを固定できますし、いっそのことインラインに展開してもよいでしょう。すべてのキャラクターが同じクラスであれば、配列で処理できます。

だからといって、別に柔軟性を求めることそのものが悪いというわけではありません。柔軟性があればゲームの変更に時間を要しません。面白いゲームを仕上げるのに開発スピードが速いことは絶対条件

※9　この話が当てはまらない反例の1つにC++のテンプレートがあります。テンプレートを用いてプログラミングを行うと、実行時の性能低下なしにインタフェースの抽象化が可能です。柔軟性にも種々の段階があることがわかります。あるクラスの実メソッドを呼び出すコードを書けば、その時点でクラスを割り当てたことになります。どのクラスを呼び出すかについてコードに書き込んだのと同じです。仮想メソッドやインタフェースを経由すると、実行時まで呼び出すクラスが特定できません。柔軟性はずっと高くなりますが、実行時のオーバーヘッドが生じることになります。テンプレートを用いたプログラムは柔軟性の面でちょうど中間になります。どのクラスを呼び出すかは、コンパイル時にテンプレートがインスタンス化される時点で決定されます。

ですから、これは重要な点です。たとえどんなに優秀なゲームプログラマーであっても、バランスのとれたゲームを紙の上だけで作り上げることはできません。変更しては動かし、実験してみることがどうしても必要です。

アイデアを試してみて、どのような感じになるかを確かめるという作業が短時間でできればできるほど、いろいろ試してみることができ、ことによると素晴らしい発見ができるかもしれません。さらに、うまい仕組みが見つかった後でも時間をかけてチューニングする必要があり、そのたびに変更が必要になります。バランスが少し悪いだけでゲームの面白さが半減してしまうかもしれませんからチューニングは重要です。

答えは簡単には出ません。ゲームのプロトタイプを素早く更新できるようにプログラムに柔軟性を持たせれば、実行時のコストは何らかの形で増加します。逆に、コードを最適化すると柔軟性が減少します。

私の経験から述べると、面白いゲームを速くすることのほうが、速いゲームを面白くするより簡単です。ですから妥協点としては、デザインが落ち着くまではコードに柔軟性を持たせておき、最終的にデザインが決まったらゲームの実行効率を上げるために抽象化のコードをいくらか削除する、というのが無難でしょう。

1.4 悪いコードの良い点

少し話を変えましょう。良いコードにも悪いコードにも、それぞれふさわしい時と場合があるのでは、という話です。この本で扱うのは、保守しやすい、きれいなコードをいかに書くかという方法論が大半です。ですから、どのように書くのが「正しい」かという立場で解説していきます。しかし、なぐり書きのコードにだって価値があるのです。

優れたアーキテクチャを持つコードを書くには十分な配慮が必要であり、それには時間がかかります。さらに、プロジェクトが進行している間中、優れたアーキテクチャを維持し続けるには、多大な努力を必要とします。キャンプ場の標語にたとえれば「帰るときは来たときより綺麗になっているようにしよう」という心がけを保ちつつコードベースを扱う必要があります。

これは、そのコードベースで長期間にわたって仕事をするなら良い態度です。しかし、先ほども述べたように、ゲームデザインでは実験や探求が非常に多く要求されます。特に初期段階では、最終的に使えないとわかっているコードを書くことも頻繁に行われています。

あるゲームのアイデアが使い物になるかどうか知りたい段階で、アーキテクチャの整ったコードを書くのは、一種の賭けになってしまいます。そのアイデアが実際に画面に展開されて評価を受けるまでに時間がかかるだけでなく、もしもそのアイデアが使い物にならないと結論されたら、そのコードを構造化するためにかけた時間はまったくの無駄になってしまうのです。

プロトタイプの作成——デザイン上の疑問に答えを出すためにとりあえず動くコードをなぐり書きすること——も、正統なプログラミング作業です。ところが、非常に大きな落とし穴があります。一時しのぎのコー

ドを書いた場合、用が済んだらそのコードを破棄できることが確実でなければなりません。次のような悪い上司を私は何度も見てきました。

上司：このアイデアを試してみたいんだが。プロトタイプでいいんだ。ちゃんとしたものでなくていい。サクサクっとやってどのくらいで作れるかな?

担当：そうですね、面倒な処理を省いて、テストを省略して、ドキュメントも書かず、バグも盛りだくさんといったところで、2、3日で仮のものがお見せできますかね。

上司：いいじゃない!　頼むよ!

～2、3日後～

上司：オイ、あのプロトタイプ、いいじゃないか。今日中に不具合を修正して製品版に組み込めるようにしてくれるか?

なぐり書きの一時しのぎコードは、動いているように見えても実はそうではなく、保守は不可能であり、実際に使うには書き直しが必要だということを周囲に理解してもらわなければなりません。使い続けなければならない可能性が少しでもあるなら、きちんと書いたほうが自分の身のためです[※10]。

1.5　バランスの保持

目標は3つあります[※11]。

- プロジェクトが生きている間はコードが読みやすくなければならないので、整ったアーキテクチャが維持される。
- 実行速度が速い。
- 今日見い出された問題にできるだけ短期間で答えを出せる。

3つの目標は少なくとも部分的には対立しています。アーキテクチャが良いということは長期的には生産性を向上させますが、保守においては修正を行うたびに形を整えるための配慮が必要になります。

最も速く書ける実装法が、最も速く実行される実装法であることはまれです。逆に、最適化を行おうとすればかなりの時間を割かなければなりません。最適化してしまうと、コードベースは変更が難しくなります。高度に最適化されたコードは融通が利かなくなり、修正に困難を伴うのです。

現場では常に「今日の仕事は今日片付けて、何かあったらそれはまた明日」というプレッシャーが働き

※10　プロトタイプのコードが実用に供されるのを確実に防ぐ技の1つとして、ゲームエンジンで使用されている言語とは別の言語でプロトタイプを作成するという技があります。そうすれば、実際のゲームで使用するには書き直さざるをえなくなります。

※11　考えてみれば、どの目標もある種の速さに関するものだというのは興味深いことです。この3つはそれぞれ長期的な開発速度、ゲームの実行速度、短期的な開発速度です。

ます。しかし、今日取り上げられた問題を解決するコードはとりあえずすぐ実装する——これでは、コードベースはつぎはぎとバグと矛盾の塊になってしまい、将来の生産性が大きく下がります。

ここにも単純な答えはありません。あるのはトレードオフのみです。私が読者から受け取ったメールを見ると、みなさんはこうした言葉でがっかりするようです。特にゲームを作りたいと思っただけの、プログラミング経験の少ない方々は「正解はない。さまざまな誤答があるだけだ」と聞くと縮み上がってしまうのかもしれません。

でも、私には逆にそれが面白いのです。どの職業を考えても、人々が習得しようと懸命に努力している職業の中心には複雑に絡み合った制限があります。どのような仕事にせよ、簡単にできる方法があれば、誰でもそれを実行します。1週間でマスターできる仕事では最終的には飽きてしまいます。穴掘りのプロとして有名になった人はいないわけです※12。

この面白さはゲームそのものの面白さと共通しています。チェスのようなゲームは、必勝の手を考え出すことができません。それは、各駒の働きが微妙なバランスを保っているからで、一生かかってもすべての戦略を研究し尽くすことはできません。それとは逆にゲームのデザインが悪ければ、必勝の手ができてしまうので、何度も同じ手で勝っているうちに飽きられてしまいます。

1.6 単純さ

最近、私が感じているのは、そのような複雑に絡み合った条件を緩和するものがあるとすれば、それは単純さではないかということです。私が書く最近のコードでは、なるべくすっきりと、問題を直接解決するコードになるよう努力しています。コードを読んだときに何をしているのかが正確に把握でき、他の解決策と間違えないようなコードです。

データ構造とアルゴリズムを（ほぼこの順序で）正しく理解するように努め、そこを出発点とします。物事を単純に進めることができれば、最終的にはコード量が減少します。そうすれば変更するときにも脳に読み込むべきコード量が減るということです。また、オーバーヘッドも少なくなり、実行するコード自体も減少するわけですから、プログラムの実行も高速になるのが普通です（とはいえ、必ずそうだとは言えません。小さいコードに繰り返しや再帰呼び出しを詰め込めば遅くなります）。

ただし、「単純なコードは、書く時間も短くなる」とは言ってはいないことに注意してください。全体のコードの量が減れば、それを書く時間も短くて済むと考えるかもしれませんが、良いコードは蓄積によって得られるのではなく、精製によって得られるのです※13。

※12　もしかしたらいるかもしれません。私はきちんと調べていません。知っていることといえば、熱心なアマチュアの穴掘り人がいるかもしれないこと、穴掘りには流儀があること、いろいろな趣味の世界があることなどでしょうか。

※13　ブレーズ・パスカルが手紙の最後に「もっと短い手紙にできたのですが、その時間がありませんでした」と書いたのは有名な話です。アントワーヌ・ド・サン＝テグジュペリの次の言葉も有名です。「完璧だと言えるのは、何も加えるものがないときではなくて、削れるものが何も残っていないときだ」。もっと身近な例を挙げると、この本の各章に手を入れるたびに原稿が短くなるのに気付きました。章によっては初稿より20％短くなりました。

私たちに与えられる問題自体が「きれいな問題」であることはまれです。普通はただの事例の塊です。XはYの場合にはZを実行するが、Wの場合はAを実行すること、といったユースケースの寄せ集めです。言い方を変えれば、ゲームプログラムがさまざまな状況でどのように振る舞うかという事例の長大なリストです。あまり頭を使わずにプログラムを書くと、ユースケースを1つずつコードに落とし込むようなものになるでしょう。新米のプログラマーを見ていると、実際にそのようにすることが数多く見受けられます。ユースケースを見て思いついた条件文を山ほど書いていくのです。

しかし、その解決策には美しさがまったくありません。そのようなコードは、作成者が考慮した条件にしか対応できず、そこから少しでも外れた入力を与えただけで不具合が生じるのです。美しい解決策とは一般的な解のことです。短いロジックで広い範囲のユースケースを正しく処理できる解法です。

美しい解決策を見つけるのはパターンマッチングの作業やパズルを解く作業に少し似ています。一見ばらばらに見えるユースケースの例を見て、そのすべてに共通するような隠れた秩序を探るのです。苦労して見つけたときの満足感は何とも言えません。

1.7 うまくやるヒント

導入部の章は読まない人が多いので、ここまで読んでくださった読者の方々にはお礼を申し上げます。辛抱強く付き合っていただいたお返しとしては不十分かもしれませんが、以下にきっと役に立つと思われるヒントをお伝えすることにします。

- 抽象化と分離により、プログラムの改良は早く簡単にできるようになりますが、取り組んでいるコードがそのような柔軟性を必要としていると確信できるまでは、抽象化と分離に無駄な時間を割くべきではありません。
- 開発サイクルのすべての段階で実行速度を考慮しつつデザインを考えるべきですが※14、低レベルの究極的な最適化は、コードに前提条件を持ち込んでしまうので、できるだけ後に回すべきです。
- ゲームのデザインを変更してみるときに、その作業が素早いのは良いことです。しかし、変更した後には片付けをしっかりしましょう。そのコードとは長い付き合いになるのですから。
- 破棄することが決まっているコードをきれいに書くのは時間の無駄です。ロックスターたちはホテルに泊まったとき、翌日にはチェックアウトするとわかっているので部屋を掃除したりしません。
- しかし何と言っても、何か面白いと思ったことがあるなら、それをやって楽しむのが最上です。

※14　納期があと2ヶ月に迫るまで「ゲームの実行速度が1フレーム/秒しか出ない」という重大問題を放置すべきではありません。

Game Programming Patterns

Part Ⅱ

デザインパターン再訪

Design Patterns Revisited

・Chapter 2　コマンド……Command
・Chapter 3　フライウェイト……Flyweight
・Chapter 4　オブザーバ……Observer
・Chapter 5　プロトタイプ……Prototype
・Chapter 6　シングルトン……Singleton
・Chapter 7　ステート……State

『オブジェクト指向における再利用のためのデザインパターン』（「四人組」による執筆）の英語版が出版されてから20年以上の年月がたちました。変化の激しいソフトウェア業界においては、「古典」と呼ぶにふさわしい本かもしれません。しかし、この本の人気はいまだに衰えておらず、良いデザインはフレームワークや方法論を超えて存在しうるものだということを示しているように思えます。しかし一方で、ここ20年で我々が学んだことが多いこともまた事実です。

Part II（第2章から第7章）では、「四人組」のデザインパターンの中から、この本のPart III以降の議論に特に有用と思われる、6つを取り上げて紹介します。シングルトン（第6章）のように使われすぎのものもあるように思いますし、コマンド（第2章）のように高い評価を得ていないものもあります。また、フライウェイト（第3章）とオブザーバ（第4章）は、ゲーム開発との関係でぜひとも取り上げたいと思いました。そして、プロトタイプ（第5章）と状態（ステート）（第7章）は、その応用範囲の広さを考慮してここで紹介したいと考えました。

第 2 章

コマンド
Command

リクエストをオブジェクトとしてカプセル化する。
これによって、異なるリクエスト / キュー / ログリクエストをもったクライアントのパラメータ化、および取り消し可能なオペレーションが実装できるようになる。

Encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.

「コマンド」は私の好きなパターンの1つです。私の書く大きなプログラムでは、それがゲームであれ何であれ、そのほとんどのどこかでこれを使っています。うまく使うと厄介なコードをすっきりとまとめることができます。そんな優れものパターンに四人組は、案の定、難解な説明文を付けています。

この章の冒頭の文がわかりにくいというのには、誰もが同意してくれると思います。第1に比喩（メタファー）がどれもわかりにくい。単語がどんな意味でも持ちうる不思議なソフトウェアの世界ならいざ知らず、普通の世界では「クライアント」は「人間」であり、ビジネスの相手のことです。私の知るかぎり、人間はパラメータ化できません。

そして残りの部分は、このパターンを使うとよさそうな場合を羅列しているだけで、うまく当てはまればよいのですが、それ以外の場合についてはわかりません。私なりにコマンドパターンに簡潔な謳い文句をつけてみると次のようなものになります。

コマンドとは、メソッド呼び出しを具現化（reify）したものである[※1]。

もっとも、簡潔にしたためにかえって言葉足らずになってしまえば、オリジナルと比べてたいして良くはなりません。少し補足しましょう。「具現化」とは実体のあるものにするということです。一部のプログラミング言語にある「リフレクション」は、実行時にプログラム中の型を命令として扱うことができる機能です。他のオブジェクトのクラスを表すオブジェクトを作り、それを使ってその型で何ができるかを知ることができます。つまり、リフレクションとは「型システムを具現化したもの」です。なお、コンピュータ用語のreifyには、あるものを「第一級（ファーストクラス）」にするという意味もあります 。

どちらの意味でも、何らかの「概念」をひと塊の「データ」つまりオブジェクトに変換して、変数に入れたり、関数に渡したりできるようにすることを指します。ですから、コマンドパターンが「メソッド呼び出しを具現化したもの」だということは、「オブジェクトにラップされたメソッド呼び出し」だという意味です。

さまざまなプログラミング言語に備わっている「コールバック」「ファーストクラス関数」「関数ポインタ」「クロージャ」あるいは「関数の部分適用」などといった概念と類似するものです。実際、どれも大きく外れてはいません。四人組も後に「コマンドパターンは、コールバックのオブジェクト指向における代替品である」と語っています。

こちらのほうが、彼らがこのパターンに付けたものよりも良い説明文になるでしょう。とはいえ、やはり抽象的でボヤッとした感じがします。私は具体的な物事から章を始めるのが好みなのですが、しくじってしまいました。その代わりと言っては何ですが、ここからは全部、コマンドパターンが見事に決まる例で説明します。

※1　「reify」は、「thing（モノ）」を意味するラテン語の「res」と英語の接尾辞である「-ify」からできた単語です。したがって、thingify（モノ化する）という意味になります。thingifyのほうが意味はわかりやすいでしょう。

2.1 入力の設定

どんなゲームのプログラムにも、ボタンのクリック、キーボード関連のイベント、マウスのクリックなどのユーザーからの入力を処理する部分があり、ゲーム内で何らかの意味のある動作に入力を変換します。

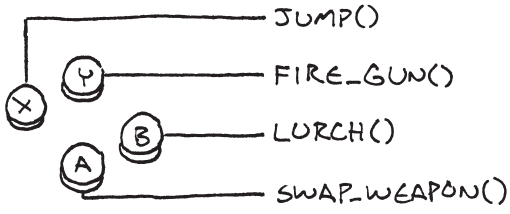

図2.1　ボタンと動作の対応

これ以上ないという単純な実装は、次のようになります※2。

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();  //  ジャンプ
  else if (isPressed(BUTTON_Y)) fireGun();  // 発砲
  else if (isPressed(BUTTON_A)) swapWeapon();  // 武器の交換
  else if (isPressed(BUTTON_B)) lurchIneffectively(); // 意味もなくふらつく
}
```

この関数は典型的には、1フレームごとに1回、ゲームループ（第9章）から呼び出されるものですが、何をしているかはわかるでしょう。ユーザーからの入力に対してゲーム内の動作を固定的に結び付けるならこのコードでよいのですが、多くのゲームではボタンと動作の対応をユーザーが設定できるようになっています。

その設定を可能にするには、`jump()`や`fireGun()`の直接の呼び出しを、入れ替えができるものに変更しなければなりません。「入れ替え」と言うと「変数への代入」を意味している感じがしますが、「入れ替え」にはゲーム内の動作を表すオブジェクトが必要です。ここでコマンドパターンが登場します。

ゲーム内で出せる指示（コマンド）を表す基底クラスを次のように定義します※3。

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
```

※2　プロのコツ――［B］ボタンはあまり押さない。

※3　何も返さないメソッドが1つだけ記述されているインタフェースがあるときは、コマンドパターンを使うよいチャンスです。

```
};
```

ゲームでの各動作に対応するサブクラスを作成します。

```
class JumpCommand : public Command
{
public:
  virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
  virtual void execute() { fireGun(); }
};

 // 以下、同様…
```

入力ハンドラの中では、各ボタンに対するコマンドオブジェクトへのポインタを格納します。

```
class InputHandler
{
public:
  void handleInput();

  // 以下、コマンドをボタンに割り当てるメソッドなど

private:
  Command* buttonX_;
  Command* buttonY_;
  Command* buttonA_;
  Command* buttonB_;
};
```

これで入力ハンドラの処理は、これらのオブジェクトに依頼するだけです[※4]。

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();

  else if (isPressed(BUTTON_B)) buttonB_->execute();
```

※4　ここでNULLかどうかの確認をせずに済ませている点に気付きましたか。このメソッドでは、各ボタンには必ず何らかのコマンドが割り当てられていると想定しています。明示的にNULLの確認をせずに、何もしないボタンにも対応したい場合は、何もしないメソッドexecute()をもったコマンドクラスを定義すればよいのです。そして、ボタンの処理を行う部分をNULLに設定せずにそのオブジェクトを指すようにします。これは「Null Object」というパターンです。

```
}
```

各入力に応じて関数を直接呼び出していたところに、新たに間接参照の層ができました。

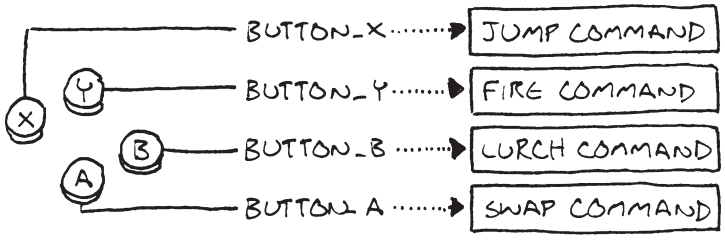

図2.2　ボタンと割り当て可能なコマンドの対応

これがコマンドパターンの概略です。その利点がわかるならば、この章の残りはおまけと考えてください。

2.2　アクターへの指示

今、定義したばかりの各コマンドクラスは、上の例ではうまくいきますが、使える場合がかなり限られています。問題は、暗黙のうちにプレーヤーのキャラクターを見つけて動かす方法を知っている`jump()`、`fireGun()`などのトップレベルの関数があることを前提としていることです。

この結合を前提としているために、こうしたコマンドの用途が限定されてしまうのです。`JumpCommand`でジャンプさせられるのは、そのプレーヤーだけです。この制限をゆるめましょう。関数を呼び出すとその関数が自分でコマンドの対象を見つけるのではなく、関数を呼び出すときに対象としたいオブジェクトを渡すようにします。

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute(GameActor& actor) = 0;
};
```

ここで`GameActor`はゲーム内のキャラクターを表す「ゲーム要素」のクラスです。コマンドの派生クラスが任意のアクターに対してメソッドを呼び出せるように、アクターを`execute()`に渡します。

```
class JumpCommand : public Command
{
public:
```

```
  virtual void execute(GameActor& actor)
  {
    actor.jump();
  }
};
```

こうすれば、このクラスを1つ使うだけでゲーム内の任意のキャラクターを跳び回らせることができます。あとは入力ハンドラとコマンドの間に、コマンドを受け取って、そのコマンドに適切なオブジェクトに対して呼び出す部分を追加するだけです。まず、`handleInput()`がコマンドを返すように変更します。

```
Command* InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) return buttonX_;
  if (isPressed(BUTTON_Y)) return buttonY_;
  if (isPressed(BUTTON_A)) return buttonA_;
  if (isPressed(BUTTON_B)) return buttonB_;

  // 何も押されていないので何もしない
  return NULL;
}
```

このメソッド内では渡すべきアクターがわからないので、このコマンドを実行することはできません。ここで「コマンドは呼び出しを具象化したもの」であることが活きてきます。つまり、呼び出しの実行を遅らせることができるので、ここで実行しなくても後で実行できるのです。

そのためには、コマンドを受け取って、プレーヤーを表すアクターに対してそれを実行するコードが必要です。次のようにします。

```
Command* command = inputHandler.handleInput();
if (command)
{
  command->execute(actor);
}
```

`actor`がプレーヤーのキャラクターへの参照だとすれば、ユーザーの入力に応じてそのキャラクターを間違いなく動かすので、最初の例と同じ動作をするようになりました。さらに、コマンドとそれを実行するアクターの間に間接参照の層を追加したことによって、ちょっとしたことが可能になっています。つまり、コマンドを適用する対象のアクターを切り替えることによって、ゲーム内の任意のアクターをプレーヤーがコントロールするようにできるのです。

実際には、これは一般的な機能ではありませんが、似た使用例は現実によくあります。ここまではプレーヤーが操作するキャラクターについてばかり考えてきましたが、ゲーム世界の他のいろいろなアクターについてはどうでしょうか？ こうしたアクターはゲームのＡＩエンジンによって動かされます。この同じコマ

ンドパターンは、AIエンジンとアクターのインタフェースとして使うことが可能です。AIのコードが単にCommandオブジェクトを出力するだけでよいのです。

コマンドを選択するAIと、コマンドを実行するアクターのコードをここで切り離したことによって、柔軟性が得られます。アクターごとに異なるAIモジュールを使うこともできます。あるいは、AIをうまく組み合わせて異なるタイプの振る舞いを作り出すことも可能です。より攻撃的な敵が欲しければ、そのような攻撃的なAIを組み込んで専用コマンドを生成させるだけです。もちろんプレーヤーのキャラクターにAIを割り当てることもできます。たとえばゲームを自動操作で実行させる必要があるデモモードなどに有効です。

アクターを制御するコマンドを第一級（ファーストクラス）オブジェクトにすることによって、直接のメソッド呼び出しという強い結合をなくしました。その代わりに、図のようなコマンドのキュー※5またはストリームとして考えます※6。

図2.3　ストリームのイメージ

一部のコード（入力ハンドラやAI）はコマンドを生成して、それをストリームに入れます。他のコード（ディスパッチャやアクター自身）は、それを取り出してその処理を呼び出します。間にこのキューを入れることによって、コマンドの生成側と消費側を分離しています※7。

※5　キューにすることの利点については、「イベントキュー」（第15章）を参照してください。

※6　私が「ストリーム」の絵を描く必要を感じたのはなぜでしょうか。またそれがチューブのように見えるのはなぜでしょうか。

※7　これらのコマンドを受け取って直列化すれば、それらのストリームをネットワーク越しに送信できます。プレーヤーの入力を受け取り、ネットワークを経由して他のマシンに送り込み、それをそこで再現することができます。この手法を利用することで、ネットワークを経由したマルチプレーヤーゲームが実現できます。