

# 大きなゲームの為の プログラミング手法

---

ゲームプログラマのためのデザインパターン *for XNA/C#*

野村 周平



# 目次

第 0 章 - はじめに.....	5
この本の立ち位置・読者対象 .....	6
第 1 章 - まずはゲームを作つてみよう.....	7
何を作ろうか考えよう .....	8
そのゲーム、何日で作れる? .....	9
大きなゲームは、小さな機能の複合体である.....	10
仕様書を作る癖を付けよう .....	12
360° 弾避けゲームを作つてみよう .....	13
ゲーム仕様.....	13
まずは何も考えずに組んでみる .....	15
完成したら動かしてみよう.....	23
第 2 章 - 問題点を探そう .....	24
大きなゲームと小さなゲームの違い .....	25
第 3 章 - プログラムの改良をしよう .....	28
ステップ 1: 機能ごとに分けよう .....	29
コメントを付ける .....	30
サブルーチンで分割する .....	35
マジックナンバーの定数化 .....	41
ステップ 2: データごとにまとめよう .....	44
構造体 .....	44
あるものは使おう .....	57
オブジェクト指向への第一歩 .....	65
カプセル化 .....	68
ステップ 3: 「動け!」だけで勝手に動くようにしよう .....	87
ポリモーフィズム .....	88
シーン管理とタスク管理 .....	109
名前空間を使って整理する .....	136
Singleton パターン .....	149
ステップ 4: パフォーマンスを改善しよう .....	151
メモリ管理とガーベージ コレクション .....	152
敢えてガーベージ コレクションを呼んでしまう .....	156
Flyweight パターンを使う .....	157
State パターンを使う .....	164
ステップ 5: もっとスマートに美しく書こう .....	181

コーディングスタイルを統一する	182
ネーミングルールを決めよう	183
入口 1 つに出口 1 つ	183
たとえば、弾を作る処理	183
バグ対策	183
テストをしよう	183
手動より自動	183
ユニットテストツール紹介	183
第 4 章 - あとがき	184
ゲームプログラミングの心得	184
できるだけ小さく作れ	184
小さなゲームでも大きなゲームの作り方を	184
バージョン管理ツールを使おう	184
プログラムに限った話ではない	184
バシバシ公開して叩かれろ	184
パフォーマンス改善は最後に	184
付録	185
UML の解説	185
サンプルプログラム紹介	185
開発環境の設定	185
索引	186
筆者自己紹介	187

## 第0章 - はじめに

皆さんこんにちは。野村 周平です。この度はこの本にご興味を持って戴き、誠にありがとうございます。あなたはこの本に興味を持って戴いたということは、多少はゲーム製作に興味があることと存じます。これから実際にゲームを作りながら教えていきたいと思いますので、宜しくお願ひします。

私はサラリーマンしながら副業として専門学校でゲームプログラミングの講師をやっています。講師の方は始めてからたったの一年間ですが、教える側としても色々なことがわかりました。知り合いから専門学校の学生はバカばかり、特にここ暫くはゆとりの程度が酷い、と散々なことを吹き込まれて、正直なところ自分もそういう先入観を持っていましたが、いざ蓋を開けてみると実は彼らはレベルが高いんです。例えばクラスとは何者か一応それなりに判っているみたいですし、そちらの入門サイトレベルの内容なら判るし組めるのです。ゲームもしょうもないワンキーゲーム程度のものでしたらちよこちよこと作れるようです。でも、その学生さんは決まってちょっと大きなゲーム（ここではマリオブラザーズは「最も小さい大きなゲーム」と見て良いでしょう）を作らせようとすると、あつという間に破綻してしまうのです。

私はその時学生たちのソースコードを見て、なぜ作れないか一目で理解できました。彼らは小さなゲームの作り方しか知らないのです。それも入門サイトのサンプルコードから推察したような作りばかり。そこで私は書店へ行き、大きなゲームの作り方を解説している書籍を探しました。確かに幾つかの書籍は、私の教える内容に極めて近いものがありました。それらは総じて深く考えずに「ゲーム作るぞー！」と意気だっている人が、興味を持って手に取るようなものではありませんでした。そこで、私は筆を執り、彼ら学生さんと独学でゲームを作ろうと勉強している方々のための、大きなゲームプログラムの教科書を作ろうと決意しました。

少々前置きが長くなってしまいましたね。ここから本編に入っていきますので、皆さんどうか宜しくお願ひします。最後に、本書を書くきっかけとなりました、東京デザイナー学院ゲームクリエイター科の皆様に感謝します。

野村 周平

## この本の立ち位置・読者対象

本書は「小さなゲームしか作れない人のための、大きなゲームプログラミング手法」の入門書です。今回少ないおまじないでゲームが作れるように、と C# 言語及び Microsoft XNA Framework 3.1 を選択しましたが、本書はこれらの入門書ではありません。プログラムが本当に初めてである場合は、まず C# 言語の入門書と XNA の入門書をお探しの上で、本書を読むことを推奨します。下記に改めて読者対象を要約します。

- 「猫でもわかるプログラミング<sup>1</sup>」など入門サイト程度のレベルのプログラムの基本が理解できる。(変数・配列・ポインタ・構造体・制御構文。オブジェクト指向の知識も少しはあると尚良い)
- C# と XNA の基本をある程度理解している。  
(キー入力で 2D グラフィックを動かす程度の XNA アプリが自力で作れれば十分です)
- でも、これがどう組み合わせればゲームになるのかわからない。あるいは本当に小さなゲームならどうにか作れるが、少しでも大きなゲームを作ろうとすると、あっという間に破綻してしまう。

もしあなたが上記のレベルに満たない場合、C#、または XNA の入門書と併読すると大変わかりやすいでしょう。一方、上記のレベルとほぼ同じくらいの方でしたら、きっとこの本は一番あなたのため役立つものと思います。

また、本書は「大きなゲームのためのプログラミング手法」を効率よく、分かりやすく説明するために画面演出などを最小限に抑えています。具体的には XNA に標準でついてくる GameThumbnail を使った 2D 描画のみで、3D 描画など画面演出上の凝った処理などは原則扱いません。3D 描画周りはどの XNA 入門書でも載っていますので、そちらを参考にしていただけると幸いです。

---

<sup>1</sup> 初心者向けプログラミング解説サイトとして有名。 [http://www.kumei.ne.jp/c\\_lang/](http://www.kumei.ne.jp/c_lang/)

## 第1章 - まずはゲームを作ってみよう

本書はゲームプログラミング手法の入門書ですので、早速何かゲームを作ってみましょう。しかし、何を作るかを全く決めずに、またはそれが曖昧なままでいきなりプログラムを組み始めてしまうと、あっという間に破綻してしまいます。そうしているうちにゲーム完成前にモチベーションが下がってしまい、断念してしまう一因にもなります。そこで、まずは一旦プログラムから目を逸らして、「何を作るか」を決めましょう。

## 何を作ろうか考えよう

あなたはどんなゲームを作るか決めていますか？もし決めていない場合、まずは何を作るか考えてみましょう。パズルでしょうか？アクションでしょうか？クイズゲームも良いでしょうし、シューティングも悪くありません。ここで一番好きなジャンルを選ぶと、後々モチベーションを維持しやすいです。ただし、RPG は少々おすすめできません。アドベンチャーも脚本と原画が既に出来上がっているとかでなければ、やめた方が良いでしょう。理由は次節で説明します。

作るゲームを決めたら、その製作ブック専用としてノートを一冊買って、メモを取っておきましょう。この際 Word や Excel を使うことはあまりお勧めできません。画面構造や簡単なラフをこれらのツールで描くのは、手書きよりもはるかに多くの時間を浪費してしまい、その間に折角のインスピレーションを逃がしてしまう場合もあります。どうしても Word や Excel でまとめたい場合、一旦ノートに取ってから、本決まりした項目だけをそれらツールで清書していくと良いでしょう。

## そのゲーム、何日で作れる？

あなたは本書を見ているということは、少なくとも家庭用ビデオゲームを一度くらいは遊んだことがありますよね？一作品でも良いので全解きしたことのある方でしたら、最後のスタッフロールを見たことがあるかと思います。普段はぼけ一つと読み飛ばしてしまうかと思いますが、ここには実は製作に関わる重要な情報が隠れているのです。ゲーム雑誌とか見ると「製作期間〇〇カ月」とかたまに載っていますが、もし製作期間（専門用語では、「工期」と言います）とスタッフロールの人数、両方知っている場合はその二つをかけ合わせてみましょう。一つの数値が出てきましたね。これがゲームの規模「工数」です。

少々判りにくいかと思いますので、一つ例題を出しましょ。開発メンバーが7人いて、彼らが5日間でゲームを仕上げる場合、 $7 \times 5 = 35$ となります。この35が工数です。単位は「人日」と言います。人数と日数をかけているため人日です。月数でかければ人月です。例えばこれと同じゲームを7人ではなく5人で作る場合、逆に35から5で割った数値、7日間が工期、すなわち製作時間です。1人だけでしたらもっと単純ですね。35を1で割るので、そのまま35日間かかることになります。このような計算手法を「人月計算<sup>2</sup>」と言います。

さて、それではあなたの今考えたゲームは、どの位の製作時間を要するのでしょうか？予想してみましょう。もし初めてのゲーム製作で何カ月もかかりそうだと思う場合、それは企画を改めた方が良いでしょう。最初に作るゲームは1か月程度の工数が望ましいです。そのくらいなら途中で萎えずにモチベーションが高いままで完成を迎えられるでしょう。ゲーム製作は、あなたの想像以上に飽きやすいものです。簡単なゲームなら1か月で行けるかもしれません、RPGなどは1か月では少々厳しいですね。ちなみに、このようにゲーム製作にかかる工数を製作開始前に予想、あるいは計算することを「工数見積もり」と言います。これはプロのプログラマになるためには避けては通れない知識<sup>3</sup>なので、できれば今のうちに覚えておいて、何か作る前に見積もりをしておくと練習になるかもしれません。

人月計算の公式：

$$\text{工数} = \text{工期} \times \text{人数}$$

<sup>2</sup> 実際はメンバーの能力差もありますし、お互い認識の誤差もあるでしょう。それを埋めるために打ち合わせが要りますし、そうすればその分の時間のロスが生じます。人月計算はメンバーが少ないほど、また規模が小さいほど正確に出やすくなります。

<sup>3</sup> 教えておきながらなんですが、私個人的には人月計算は大嫌いです。  
もしメンバーが2100人いたら1分でゲームが作れますか？

大きなゲームは、小さな機能の複合体である

さて、では実際に予想してみるといきなり言われても、「そんなの大きすぎて予想できないよ！」と言う方が多いのではないでしょうか？では一旦この話は置いといて、別のソフトウェアを見積もってみましょう。

あなたは「Hello, world」をご存知でしょうか？これは画面に「Hello, world!」という文字を表示するだけのソフトウェアで、プログラム言語の習得のために最初に作るコードです。この程度のソフトウェアなら簡単に予想できますよね？早い人なら 1 分程度かもしれませんし、逆にいくら遅い人でも、恐らく 1 時間はかかるでしょう。

そろそろ話を戻しましょうか。例えばあなたが、マリオブラザーズのクローンゲームを作りたいとしましょう。でも「そんなの大きすぎて予想できないよ！」となってしまった場合、予想できる範囲までゲームを分割してしまえばよいのです。

分割するためには、まずそのゲームにどんなキャラクタがいるかをリストアップして、ノートに書き込んでいきます。マリオ以外にも色々いますよね、ハエとかカメとか、あと床や POW<sup>4</sup>など当たり判定のある物体もキャラクタに含みます。次にそれぞれのキャラクタの機能をリストアップします。例えばマリオの場合、左右へ走る、ジャンプする、床や POW を突き上げる、倒れている敵に体当たりして敵を倒す、動いている敵に体当たりして死ぬなどの機能がありますね。おっとキャラクタの描画も忘れてはいけません。この調子でほかのキャラクタもすべてリストアップしてみましょう。面倒だと投げ出す人もいるかもしれませんですが、このリストは後でも使用するので、今のうちにやっておくとよいでしょう。それにゲームプログラミングはこの数倍面倒なので、この程度で面倒だと投げ出すと後々大変ですよ？さて、話を戻してキャラクタ以外の機能についても、忘れずにリストアップしましょう。例えばスコア計算や残機情報の管理及び表示、ゲームオーバー判定もありますよね？

ここまで分割できたら、そろそろ一機能単位の工数が予想できるのではないか？まだ予想が難しいようなら分割が足りない証拠でしょう。全項目の工数が予想できたら、それを全部足し合わせます<sup>5</sup>。そうして出てきた数字が、そのゲームの「工数<sup>6</sup>」となります。出た工数は忘れないようにノートに書

<sup>4</sup> 下から突き上げると画面中の敵が全員ひっくり返る特殊な床です。

<sup>5</sup> 実際仕事でやる場合は、もう少し水増ししたりします。それぞれの機能を繋げるのにも時間がかかり

いておきましょう。合計値だけではなく、各機能単体の工数も全て書き記しておきます。そして、一つの機能を完成させたときにどのくらいの時間がかかったのか（「実績」と言います）、また実績と予想との誤差を別の色のペンで書き記します。これを専門用語で「予実管理」と言いますが、これを繰り返すうちにどの程度の誤差が出るか判るようになりますし、より正確な見積もりが出せるようになります。正確な見積もりが出せると、いつリリースできるかがはっきりしますし、期限に間に合うかどうかが製作開始前に判るため、機能の調整なども簡単にできます。コミックマーケット直前で間に合わなくなって、落としてしまうなんてこともなくなります！

---

ますし、複数人数でやる場合は打ち合わせによるロスもありますし。経験上一人で作る場合は2倍、数人で作る場合は3倍くらいになると後々言うこともなく快適に開発が進められます。

<sup>6</sup> ここでは実装だけの工数だけを説明しています。実際にはグラフィック製作やサウンド製作、テストプレイやマニュアル作成、果ては特設ウェブサイト製作も工数に含みます。

## 仕様書を作る癖を付けよう

前回の機能分割の項目までで、おのずとそのゲームの大まかな機能一覧が見えてきたと思いますので、それをさらに事細かく書き足して、「仕様書」を作ります。また面倒だと思う人もいるかもしれません、仕様書がないと、集団で作るときに各個人の脳内イメージが大きくずれて大変困ります。また、一人で作るときでも仕様書があると、何を以て完成するかが把握しやすいため、面倒でも作っておきましょう。

## 360° 弾避けゲームを作ってみよう

本書ではサンプル 1 として 360° 弾避けゲームを作ることにします。以下にこれを選んでみた理由を簡単に示しておきます。

1. 「小さなゲーム」でありながら「大きなゲーム」の要素を含んでいること。  
(「大きなゲームの作り方」を知らなくても、辛うじて作れる程度である)
2. 全くプログラムを知らないとかでなければ、1か月以内には作れるはずであること。
3. 後述するが、プログラムを習得したばかりの人が陥りやすいミスを再現した、問題点のあるソースコードを書きやすいこと<sup>7</sup>。またその際に恐ろしく長くならないこと。

### ゲーム仕様

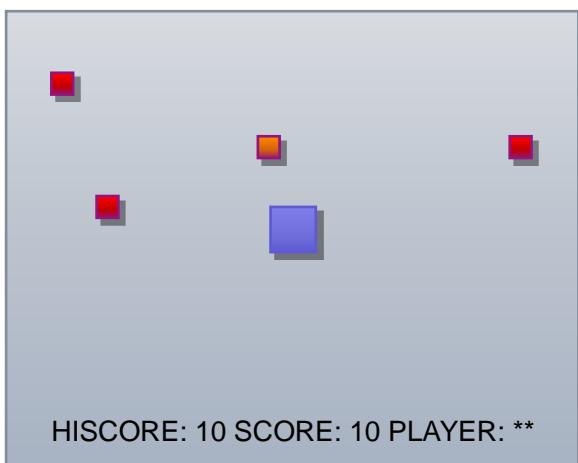
#### ● シーン

- タイトル画面
  - ✧ ゲーム名は「SAMPLE 1」
  - ✧ タイトルとハイスコアを表示。
  - ✧ キーボードのスペースキーを押すとゲームシーンへ移行、ESCキーでゲーム終了。
- ゲーム画面
  - ✧ ゲームオーバー時にタイトルシーンへ移行。



#### ● キャラクタ

- 主人公
  - ✧ 初期状態で画面中心にいる。
  - ✧ 大きさは 64 ピクセル。
  - ✧ キーボードの十字キーで上下左右に動く。画面外に移動することはできない。
  - ✧ 弾に接触するとミス。画面中の弾はクリアされる。
  - ✧ 主人公に初期状態で 3 回ミス猶予があり、猶予がなくなるとゲームオーバー。



<sup>7</sup> あまり大きなゲームでこれをやると、筆者ですら発狂しそうで怖い。

- 敵機
  - ✧ 全方位の画面端の任意の位置(ランダム)から敵機が出る。
  - ✧ 大きさは主人公の半分の 32 ピクセル。
  - ✧ ランダム等速度の自機狙い。最初は遅いが徐々に高速化。
  - ✧ 直進タイプと発射後 1 秒間ホーミングするタイプの 2 種類<sup>8</sup>。比率は 8:2。
  - ✧ 直進タイプは赤色、ホーミングタイプは橙色。
  - ✧ 1 発発射ごとにスコア+10 点。
  - ✧ 最初は毎秒 1 発だが徐々に激化する。プレイ開始 50 秒後には倍の毎秒 2 発、1 分後で 2 秒に 5 発、おおよそ 1 分 40 秒後には秒間 60 発。
- スコア
  - ゲーム開始時は 0 点。
  - 500 点ごとにミス猶予が 1 回追加される。
  - 最高スコアを更新した場合、ハイスコアとしてゲーム終了時まで保持される。
- HUD
  - スコア・ハイスコア表示。
  - ミス猶予数(残機)を \* (アスタリスク)の数で表示。

---

<sup>8</sup> 初時は精度の悪い自機狙い直進の赤紫色もありましたが、コード量が増えるためここでは削減しました。もし余力のある方は付け加えてみてください。ちなみに第 3 章ステップ 3 の最後の辺り (Sample1\_11) で、この敵機も実装しています。

まずは何も考えずに組んでみる

そろそろ「いつになつたらプログラムが組めるんだ」、と突っ込みが来そうですね。でも、もう我慢する必要はありません、皆さんお待ちかねのプログラミングの時間が来ました。XNA ゲームのプロジェクトを作成し、思うがままソースコードを VisualStudio へ書き殴ってください。

今回のサンプル(Sample1\_01)のコードはすべて XNA プロジェクト作成時に自動的に作成される Game1.cs をベースに書き足しています。また、コンテンツとして GameThumbnail(サムネイル画像をコピー)と SpriteFont(デフォルト状態のスプライトフォント:アスキーワードのみ)を追加しています。

```
// Program.cs

using System;

namespace Sample1_01
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

```
// Game.cs

using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Sample1_01
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1
        : Game
    {

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D gameThumbnail;
        SpriteFont spriteFont;
        bool game;
        int counter;
        int score;
        int prevScore;
        int hiScore;
        int playerAmount;
        float playerX;
        float playerY;
        float[] enemyX = new float[100];
        float[] enemyY = new float[100];
        float[] enemySpeed = new float[100];
        double[] enemyAngle = new double[100];
        int[] enemyHomingAmount = new int[100];
        bool[] enemyHoming = new bool[100];

        public Game1()
```

```

{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    gameThumbnail = Content.Load<Texture2D>("GameThumbnail");
    spriteFont = Content.Load<SpriteFont>("SpriteFont");
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    KeyboardState keyState = Keyboard.GetState();
    if (game)
    {
        if (keyState.IsKeyDown(Keys.Left))
        {
            playerX -= 3;
        }
        if (keyState.IsKeyDown(Keys.Right))
        {
            playerX += 3;
        }
        if (keyState.IsKeyDown(Keys.Up))
    }
}

```

```

{
    playerY -= 3;
}

if (keyState.IsKeyDown(Keys.Down))
{
    playerY += 3;
}

if (playerX < 0)
{
    playerX = 0;
}

if (playerX > 800)
{
    playerX = 800;
}

if (playerY < 0)
{
    playerY = 0;
}

if (playerY > 600)
{
    playerY = 600;
}

if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0)
{
    for (int i = 0; i < enemyX.Length; i++)
    {
        if(enemyX[i] > 800 || enemyX[i] < 0 &&
            enemyY[i] > 600 || enemyY[i] < 0)
        {
            Random rnd = new Random();
            int p = rnd.Next((800 + 600) * 2);
            if (p < 800 || p >= 1400 && p < 2200)
            {
                enemyX[i] = p % 800;
            }
        }
    }
}

```

```

        enemyY[i] = p < 1400 ? 0 : 600;
    }
    else
    {
        enemyX[i] = p < 1400 ? 0 : 800;
        enemyY[i] = p % 600;
    }
    enemySpeed[i] = rnd.Next(1, 3) + counter * 0.001f;
    enemyAngle[i] = Math.Atan2(
        playerY - enemyY[i], playerX - enemyX[i]);
    enemyHoming[i] = rnd.Next(100) >= 80;
    enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;
    score += 10;
    if (score % 500 < prevScore % 500)
    {
        playerAmount++;
    }
    prevScore = score;
    if (hiScore < score)
    {
        hiScore = score;
    }
    break;
}
}

bool hit = false;
for (int i = 0; i < enemyX.Length; i++)
{
    if (Math.Abs(playerX - enemyX[i]) < 48 &&
        Math.Abs(playerY - enemyY[i]) < 48)
    {
        hit = true;
        game = --playerAmount >= 0;
        break;
    }
}

```

```

    }

    if (--enemyHomingAmount[i] > 0)
    {
        enemyAngle[i] = Math.Atan2(
            playerY - enemyY[i], playerX - enemyX[i]);
    }

    enemyX[i] += (float)Math.Cos(enemyAngle[i]) * enemySpeed[i];
    enemyY[i] += (float)Math.Sin(enemyAngle[i]) * enemySpeed[i];
}

if (hit)
{
    for (int i = 0; i < enemyX.Length; i++)
    {
        enemyX[i] = -32;
        enemyY[i] = -32;
        enemySpeed[i] = 0;
    }
}

counter++;

}

else
{
    if (keyState.IsKeyDown(Keys.Escape))
    {
        Exit();
    }

    if (keyState.IsKeyDown(Keys.Space))
    {
        game = true;
        playerX = 400;
        playerY = 300;
        counter = 0;
        score = 0;
        prevScore = 0;
        playerAmount = 2;
    }
}

```

```

        for (int i = 0; i < enemyX.Length; i++)
    {
        enemyX[i] = -32;
        enemyY[i] = -32;
        enemySpeed[i] = 0;
    }
}

base.Update(gameTime);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    if (game)
    {
        spriteBatch.Draw(
            gameThumbnail, new Vector2(playerX, playerY), null,
            Color.White, 0f, new Vector2(32, 32), 1f,
            SpriteEffects.None, 0f);
        for (int i = 0; i < enemyX.Length; i++)
        {
            spriteBatch.Draw(
                gameThumbnail, new Vector2(enemyX[i], enemyY[i]), null,
                enemyHoming[i] ? Color.Orange : Color.Red, 0f,
                new Vector2(32, 32), 0.5f, SpriteEffects.None, 0f);
        }
        spriteBatch.DrawString(spriteFont, "SCORE: " + score.ToString(),
            new Vector2(300, 560), Color.Black);
    }
}

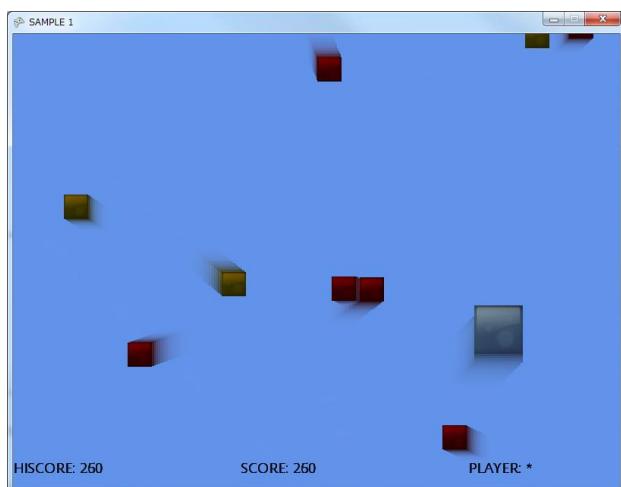
```

```
        spriteBatch.DrawString(spriteFont,
            "PLAYER: " + new string('*', playerAmount),
            new Vector2(600, 560), Color.Black);
    }
    else
    {
        spriteBatch.DrawString(spriteFont, "SAMPLE 1", new
            Vector2(200, 100), Color.Black, 0f, Vector2.Zero, 5f,
            SpriteEffects.None, 0f);
        spriteBatch.DrawString(spriteFont, "PUSH SPACE KEY.",
            new Vector2(340, 400), Color.Black);
    }
    spriteBatch.DrawString(spriteFont, "HISCORE: " + hiScore.ToString(),
        new Vector2(0, 560), Color.Black);
    spriteBatch.End();
    base.Draw(gameTime);
}
}
```

完成したら動かしてみよう

さてゲームが完成しましたので、早速ですが動かしてみましょう。

どうやら、何の問題もなくちゃんと動いているようですね<sup>9</sup>。仕様通り自機は十字キーで動き、赤弾は直進、橙弾はホーミングで自機を目指して飛んできます。残機が 500 点ごとに増えていき、逆に残機がない状態でミスをするとゲームオーバーとなり、タイトル画面に戻ります。ハイスコアはソフトウェア終了時までちゃんと保持されますね。



では、ここまでできたところで、早速このゲームをさらに改造してやりこみ要素や追加ステージ、はたまた協力プレー機能——などを組み込んでいくと、あっという間にソースコードが破綻してしまいます。ゲームの機能を改良する前に、現在のソースコードを改良する必要があります。それを確認するために、次章から書いたソースコードを確認していきましょう。

<sup>9</sup> 参考までに、筆者は 1,430 点までいけました。

## 第2章 - 問題点を探そう

前章で一つゲームのソースコードを書きましたが、ここではその問題点を探して、修正してみましょう。ここから先を読む前に、皆さんも何がまずいのか、予め予想してみて、メモしましょう。

## 大きなゲームと小さなゲームの違い

前章のソースコードはいくつかの問題点を孕んでいますが、中でも最も大きいのが、「**小さいゲーム向けのソースコードの組み方である**」ことです。大きなゲームを作るためには、大きなゲームに適したプログラミングをすることが必要です。小さいゲーム向けのプログラミング手法のままで大きなゲームを書くと、大抵途中で破綻します<sup>10</sup>。では、大きなゲーム向けのソースコードとは一体どのようなものなのでしょうか？

実際には、そんな一言で言えるほど簡単な正解ではありませんが、誤解を恐れず極論を言うと、「**小さいソースコードを書くこと**」が正解です。「なぜ大きなゲームを作るのに小さいソースコードを書かなきゃいけないんだ？ どう考へても逆じゃないか？ これは禅問答か？」と、疑問に思う方も多いのではないかと思いますが、謎かけでも頓智でも何でもなく、大きいゲームを作るためには小さいソースコードを書いて、その小さいソースコード同士を繋げていくことにより大きくしていくのが、破綻せずに完成させる最大の近道なのです。

それでは、大きいソースコードと小さいソースコードの違いとは何なのでしょうか？ 正解を挙げたところで、前章のソースコードをもう一度見てみましょう。そして、最初に挙げた「Hello, world」も見て、両者のソースコードの違いをよく考へてみましょう。「Hello, world」の方が圧倒的に見やすいことに、恐らく皆さんは気付くはずです。そうです、小さいソースコードを書くと、その分見やすくなるのです。実際に前章のソースコードにある、各メソッドの行数を数えると、下記のようになります。

- Game1 コンストラクタ 2 行
- LoadContent メソッド 4 行
- Update メソッド 125 行
- Draw メソッド 34 行

Game1 コンストラクタと LoadContent メソッドは良いとしても、Update メソッドは多すぎですね。1 メソッドだけで 125 行と、相当な行数を消費しています。Draw メソッドもやや膨れていますね。このゲームを選定した理由として、コードが長くなり過ぎないことと言いましたが、それでも何も考えずに書くと、ここまで膨れ上がってしまいました。私個人的には 1 メソッド内のコード量は多くても 30 行～40 行、できるなら 20 行弱に抑えるべきだと思います。それ以上増えると、ソースコードの可読性が大きく落ちます。可読性が悪いと改修するのも大変です。下手すると右を叩けば左が出て、左を叩けば右が出て、と目も当てられない状況に陥ることもあります。ちなみに、そのような可読性の低下を補うために、コメ

---

<sup>10</sup> 実際、これを書くだけでも結構疲れました。

ントやドキュメンテーション<sup>11</sup>がありますが、これに頼りすぎてコードの短縮化を怠ることは、正直な話推奨しません。

少々私の経験談となります。2006 年の晩夏、私がサラリーマンとしてプログラマを始めたばかりの頃、ある現場に派遣されて、そこで Web アプリケーションの改修作業に従事させられました。その時使用していた言語は C#ではなく Java (J2EE1.3)でしたが、1 メソッド辺り数千行はあるソースコードが平気で出てくるのです。そしてメソッド内のコードも、お世辞にも洗練されているとは言えず、私は一目見て、そのコードを読むのをやめました。そのメソッドのドキュメンテーション、即ち上っ面だけを見て、そういう機能なのだろうと思ってそのメソッドを利用していたのですが、どうも想定通りの動作をしてくれないのです。最初自分のところのコードの書き方、あるいは呼び出し方がおかしいのかと思っていたのですが、そのメソッドの代わりにタブを呼んでみたりしたところ、どうやらそうでもないのです。まさかなあ……と思いそのメソッドを作った人に事情を聴いたら、なんと今は内部構造が変わって全く別のメソッドになっていたとのことです。つまりそのコメントは良く言えば古いバージョン用の解説、悪く言えば全くの嘘っぽちだったのです。このときはすぐ近くに当事者がいたため助かりましたが、もしさうでもなかつたことを考えると、ぞっとなります。

ここまで読んで、「コメントとソースコードの同期を、徹底化すればいいじゃないか」と思う方が恐らく多いのではないかでしょうか。事実それが正解です。しかし実際のところ、ドキュメンテーション、あるいはコメントとソースコードを同期させ続けるのは、かなりの体力と時間の浪費が必要です。私は今までして無理してコメントを同期させるより、コメントはシンプルに書き(それこそ「〇〇します」の一言程度で十分)、余った作業時間を内部ソースコードの短縮化・洗練化に充てた方が有意義だと思います。あなたがゲームライブラリやエンジンを作っているわけでなければ、大量のコメントとドキュメンテーションで説明されるより、とても短くてユーザが 3 秒そのメソッドの中身を見れば、その機能がなんであるかを理解できるのが私にとっての理想です<sup>12</sup>。

そう言うと稀に、行数を縮めるために横に伸ばす人も見かけますが、これは一層可読性を落としてしまい本末転倒です。桁数も 100~120 桁程度<sup>13</sup>に抑えるのが望ましいでしょう。

どうでもいい話ですが、私が鼻水垂らして小学校通っていた頃は、ソースコードの単価は印刷した紙のグラム単位で、金額を水増しするために、ループなどをすべてインライン展開してコード量を増やしていました。そんな話を年配の方々からよく聞きますが、私はきっと都市伝説だろう、と信じています。

---

<sup>11</sup> コメントの一種だが、専用ツールを使用してメソッドやフィールドと紐づけたコメントを文書化できる特殊なコメントのこと。Javadoc や POD など、言語ごとに実装は異なる。

<sup>12</sup> 本当に理想であり、夢物語なんですけどね。1 分ならまだ望みはあっても、3 秒は流石に……。

<sup>13</sup> 以前は 1 行 80 桁まで、とよく言われていましたが、流石に今時では短すぎるでしょう。ちなみに MS-DOS や N-88 BASIC の画面が横 80 桁だったところから由来しています。

それでは少々話がそれてしましましたが、次のページからは「では小さいソースコードにするために  
はどのようにすればよいのか」を解説していきたいと思います。

## 第3章 - プログラムの改良をしよう

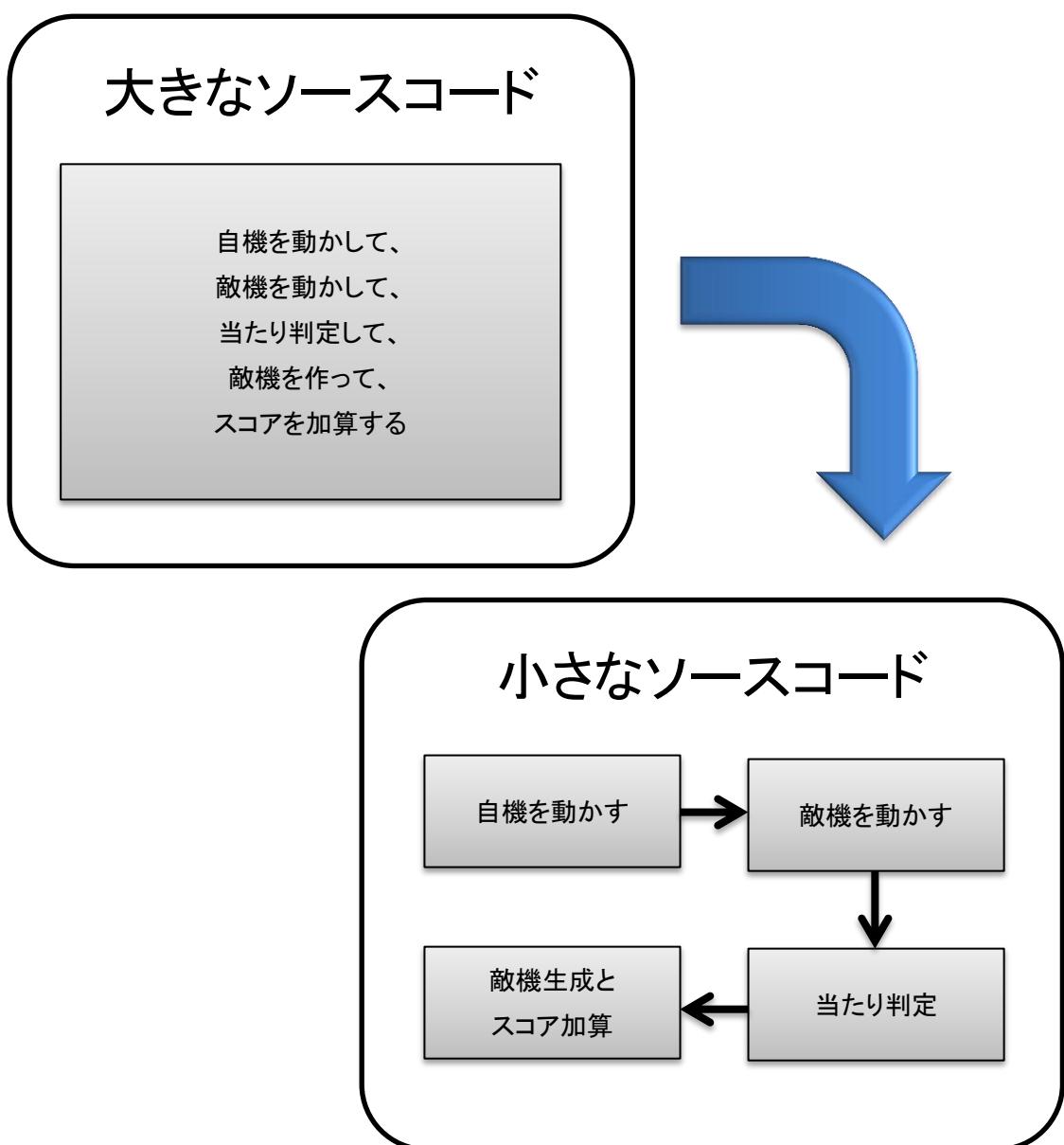
本章では、サンプル1「360°弾避けゲーム」について、前章で挙げた問題点を修正していきます。とは言っても、いきなり大きく改変したりすることはありません。そんなことしても、きっと皆さんには混乱してしまうでしょう<sup>14</sup>。みなさんが理解しやすいよう、少しずつ改変していきます。

---

<sup>14</sup> 勇気ある方は完成版をいきなり見てみるのも良いでしょう。

## ステップ 1: 機能ごとに分けよう

前章では「大きなゲームを作るためには、小さいソースコードを書く」と述べましたね。では、ゲームの大きさを維持したままソースコードを小さくするにはどうすればよいでしょう<sup>15</sup>? 何も、ソースコード全体を縮める必要はありません。ゲームを機能ごとに分割すれば、一つの機能としては小さく見えますよね?



<sup>15</sup> 「フォントサイズを小さくすれば良い」と答えた方がいました。折角なのでユーモア賞としてここに掲載しておきましょう。

## コメントを付ける

私は前章でも述べたとおり、あまり几帳面にコメントを書きすぎることは、あまり望ましくないと考えています。しかし、自分用の一時的なメモ書き程度なら、積極的に活用するのも問題ないでしょう。一方ドキュメンテーションは `private` なメソッドやフィールドであっても、一応書いておくべきです。但し、やはり内容はゲームライブラリ(ゲームエンジン)などを作っているわけでない場合、「〇〇をします」程度の至極簡潔なもので十分でしょう。

下記に示したコードでは、`Game1` クラスの全フィールドにドキュメンテーションを追加し、また `Update` メソッドや `Draw` メソッドの大まかな機能の区切りの部分にコメントを加えました。また、一部不要な変数を削除しています。(Sample1\_02)

```
// フィールド変数にドキュメンテーションを付ける例（抜粋）

/// <summary>スプライト バッチ。</summary>
SpriteBatch spriteBatch;

/// <summary>キャラクタ用画像。</summary>
Texture2D gameThumbnail;

/// <summary>フォント画像。</summary>
SpriteFont spriteFont;

/// <summary>ゲーム中かどうか。</summary>
bool game;

/// <summary>ゲームの進行カウンタ。</summary>
int counter;
```

```
// 機能の大まかな区切りとしてコメントを挿入する例(Update() メソッドより抜粋)
```

```
KeyboardState keyState = Keyboard.GetState();
if (game)
{
    // ゲーム画面
    // プレイヤー移動処理
    if (keyState.IsKeyDown(Keys.Left))
    {
        playerX -= 3;
    }
    if (keyState.IsKeyDown(Keys.Right))
    {
        playerX += 3;
    }
    if (keyState.IsKeyDown(Keys.Up))
    {
        playerY -= 3;
    }
    if (keyState.IsKeyDown(Keys.Down))
    {
        playerY += 3;
    }
    if (playerX < 0)
    {
        playerX = 0;
    }
    if (playerX > 800)
    {
        playerX = 800;
    }
    if (playerY < 0)
    {
        playerY = 0;
    }
    if (playerY > 600)
```

```

{
    playerY = 600;
}

// 弾の生成
if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0)
{
    for (int i = 0; i < enemyX.Length; i++)
    {
        if (enemyX[i] > 800 || enemyX[i] < 0 &&
            enemyY[i] > 600 || enemyY[i] < 0)
        {
            Random rnd = new Random();
            int p = rnd.Next((800 + 600) * 2);
            if (p < 800 || p >= 1400 && p < 2200)
            {
                enemyX[i] = p % 800;
                enemyY[i] = p < 1400 ? 0 : 600;
            }
            else
            {
                enemyX[i] = p < 1400 ? 0 : 800;
                enemyY[i] = p % 600;
            }
            enemySpeed[i] = rnd.Next(1, 3) + counter * 0.001f;
            enemyAngle[i] = Math.Atan2(
                playerY - enemyY[i], playerX - enemyX[i]);
            enemyHoming[i] = rnd.Next(100) >= 80;
            enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;
            score += 10;
            if (score % 500 < prevScore % 500)
            {
                playerAmount++;
            }
            prevScore = score;
            if (hiScore < score)

```

```

    {
        hiScore = score;
    }
    break;
}
}

// 弾の移動、及び接触判定
bool hit = false;
for (int i = 0; i < enemyX.Length; i++)
{
    if (Math.Abs(playerX - enemyX[i]) < 48 &&
        Math.Abs(playerY - enemyY[i]) < 48)
    {
        hit = true;
        game = --playerAmount >= 0;
        break;
    }
    if (--enemyHomingAmount[i] > 0)
    {
        enemyAngle[i] = Math.Atan2(
            playerY - enemyY[i], playerX - enemyX[i]);
    }
    enemyX[i] += (float)Math.Cos(enemyAngle[i]) * enemySpeed[i];
    enemyY[i] += (float)Math.Sin(enemyAngle[i]) * enemySpeed[i];
}
if (hit)
{
    for (int i = 0; i < enemyX.Length; i++)
    {
        enemyX[i] = -32;
        enemyY[i] = -32;
        enemySpeed[i] = 0;
    }
}

```

```
    counter++;
}

else
{
    // タイトル画面
    if (keyState.IsKeyDown(Keys.Escape))
    {
        Exit();
    }

    if (keyState.IsKeyDown(Keys.Space))
    {
        // ゲーム開始
        game = true;
        playerX = 400;
        playerY = 300;
        counter = 0;
        score = 0;
        prevScore = 0;
        playerAmount = 2;
        for (int i = 0; i < enemyX.Length; i++)
        {
            enemyX[i] = -32;
            enemyY[i] = -32;
            enemySpeed[i] = 0;
        }
    }
}

base.Update(gameTime);
```

### サブルーチンで分割する

サブルーチンとは、端的に述べるとプログラム上における、一定の機能の集合体のことです。サブルーチンというキーワードは、最近では耳にしないですね。今風にいうとメソッドとか言います。サブルーチンで分割することは、特に難しいことではありません。すでに Update メソッドと Draw メソッドに分かれているので、それをさらに細分化してやるだけです。

さて、前回機能ごとの大まかな区切りでコメントを付けましたので、今度はそれを目印にサブルーチンで分割していきます(Sample1\_03)。

```
// Update メソッド内のロジックをサブルーチンで分割する例

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    KeyboardState keyState = Keyboard.GetState();

    if (game)
    {
        movePlayer(keyState);
        createEnemy();
        if (enemyMoveAndHitTest())
        {
            enemyReset();
        }
        counter++;
    }
    else
    {
        updateTitle(keyState);
    }
}
```

```
base.Update(gameTime);  
}  
  
/// <summary>自機を移動します。</summary>  
/// <param name="keyState">現在のキー入力状態。</param>  
private void movePlayer(KeyboardState keyState)  
{  
    if (keyState.IsKeyDown(Keys.Left))  
    {  
        playerX -= 3;  
    }  
    if (keyState.IsKeyDown(Keys.Right))  
    {  
        playerX += 3;  
    }  
    if (keyState.IsKeyDown(Keys.Up))  
    {  
        playerY -= 3;  
    }  
    if (keyState.IsKeyDown(Keys.Down))  
    {  
        playerY += 3;  
    }  
    if (playerX < 0)  
    {  
        playerX = 0;  
    }  
    if (playerX > 800)  
    {  
        playerX = 800;  
    }  
    if (playerY < 0)  
    {  
        playerY = 0;  
    }  
    if (playerY > 600)
```

```

{
    playerY = 600;
}
}

/// <summary>敵機を作成します。</summary>
private void createEnemy()
{
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0)
    {
        for (int i = 0; i < enemyX.Length; i++)
        {
            if (enemyX[i] > 800 || enemyX[i] < 0 &&
                enemyY[i] > 600 || enemyY[i] < 0)
            {
                Random rnd = new Random();
                int p = rnd.Next((800 + 600) * 2);
                if (p < 800 || p >= 1400 && p < 2200)
                {
                    enemyX[i] = p % 800;
                    enemyY[i] = p < 1400 ? 0 : 600;
                }
                else
                {
                    enemyX[i] = p < 1400 ? 0 : 800;
                    enemyY[i] = p % 600;
                }
                enemySpeed[i] = rnd.Next(1, 3) + counter * 0.001f;
                enemyAngle[i] = Math.Atan2(
                    playerY - enemyY[i], playerX - enemyX[i]);
                enemyHoming[i] = rnd.Next(100) >= 80;
                enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;
                score += 10;
                if (score % 500 < prevScore % 500)
                {
                    playerAmount++;
                }
            }
        }
    }
}

```

```

        }

        prevScore = score;
        if (hiScore < score)
        {
            hiScore = score;
        }
        break;
    }
}

}

/// <summary>敵機の移動、及び接触判定をします。</summary>
/// <returns>接触した場合、true。</returns>
private bool enemyMoveAndHitTest()
{
    bool hit = false;
    for (int i = 0; i < enemyX.Length; i++)
    {
        if (Math.Abs(playerX - enemyX[i]) < 48 &&
            Math.Abs(playerY - enemyY[i]) < 48)
        {
            hit = true;
            game = --playerAmount >= 0;
            break;
        }
        if (--enemyHomingAmount[i] > 0)
        {
            enemyAngle[i] = Math.Atan2(
                playerY - enemyY[i], playerX - enemyX[i]);
        }
        enemyX[i] += (float)Math.Cos(enemyAngle[i]) * enemySpeed[i];
        enemyY[i] += (float)Math.Sin(enemyAngle[i]) * enemySpeed[i];
    }
    return hit;
}

```

```

/// <summary>敵機を初期状態にリセットします。</summary>
private void enemyReset()
{
    for (int i = 0; i < enemyX.Length; i++)
    {
        enemyX[i] = -32;
        enemyY[i] = -32;
        enemySpeed[i] = 0;
    }
}

/// <summary>タイトル画面を更新します。</summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void updateTitle(KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Escape))
    {
        Exit();
    }
    if (keyState.IsKeyDown(Keys.Space))
    {
        // ゲーム開始
        game = true;
        playerX = 400;
        playerY = 300;
        counter = 0;
        score = 0;
        prevScore = 0;
        playerAmount = 2;
        enemyReset();
    }
}

```

これだけでも Update メソッドと Draw メソッドは相當にすっきりしてきましたね。サブルーチン化することによって、以前のサンプルでは複数個所にあった敵の位置初期化処理も、一つに共通化されています。

す。次節からこのコードをさらに整理していきます。

## マジックナンバーの定数化

XNA はプロジェクトを自動生成すると、初期設定では画面サイズが SVGA(800×600)となっています。今回これに合わせて作っていきましたが、もし例えればここで「やっぱ VGA(640×480)の方が昔からのゲームの標準だから、そっちに合わせて」などと言われたら、あなたならどのように対応しますか？このゲームの場合、座標を決め打ちで入れている箇所は、どう少なく見積もっても 10 箇所は上回ります。1 個 1 個手入力で直していきますか？もし仮にそれで対応できたとしても、後になって「やっぱ XBOX360 でワイド表示に対応させるために 720p(1280×720)にして」とか言われたら、まさに発狂ものでしょう。今の話は極端な例えでしたが、実際にそうでなくとも一つ一つ手入力での修正は、修正漏れや誤入力などのミスの大きな原因となります。また、このように何の脈絡もなく現れる数値のことを「マジックナンバー」と呼び、可読性の低下を招きます。

このようなマジックナンバーには、人間に理解しやすい別名を与えるべきです。これを一般的に「定数化」と言います。例えばいきなり「800」と言う数値が出てきても、そのコードの作者でない誰かが見た場合、もしかしたら前後の文脈から判断して、画面サイズの横幅と気付く人もいるかもしれません、あなたがトリッキーな呪文を書くことを好む場合、前後の文脈から判断しきれず、それは謎の数値として見做されてしまうでしょう。そこで「SCREEN\_WIDTH」などと別名を与えてやると、そこだけ見てこれは画面の横幅か、と誰が見ても即座に認識できるようになります。

下記のサンプル(Sample1\_04)では、マジックナンバーを const 定数、または readonly 変数で定義していますが、場合によっては列挙体を使用したり、Singleton パターン<sup>16</sup>を応用した静的なインスタンスを使ったりした方がスマートな場合もあるでしょう。また、その箇所でしか使用しないことが明白なマジックナンバーは、定数をローカルスコープで定義する方が良いでしょう。また、このサンプルは前節のもの(Sample1\_03)をベースとしているので、どこが異なるかよく比較してみましょう。

---

<sup>16</sup> 本書の中盤～後半で詳しく解説しますが、そのクラスのインスタンスが、一つないし固定数存在することが保障されるアルゴリズムのことです。

```
// プレイヤーの移動範囲を定数化した例

/// <summary>自機の移動速度。</summary>
const float PLAYER_SPEED = 3;

/// <summary>画面横幅。</summary>
const float SCREEN_WIDTH = 800;

/// <summary>画面縦幅。</summary>
const float SCREEN_HEIGHT = 600;

/// <summary>画面左端。</summary>
const float SCREEN_LEFT = 0;

/// <summary>画面上端。</summary>
const float SCREEN_TOP = 0;

/// <summary>画面右端。</summary>
const float SCREEN_RIGHT = SCREEN_LEFT + SCREEN_WIDTH;

/// <summary>画面下端。</summary>
const float SCREEN_BOTTOM = SCREEN_TOP + SCREEN_HEIGHT;

/// <summary>自機を移動します。</summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void movePlayer(KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Left))
    {
        playerX -= PLAYER_SPEED;
    }
    if (keyState.IsKeyDown(Keys.Right))
    {
        playerX += PLAYER_SPEED;
    }
    if (keyState.IsKeyDown(Keys.Up))
```

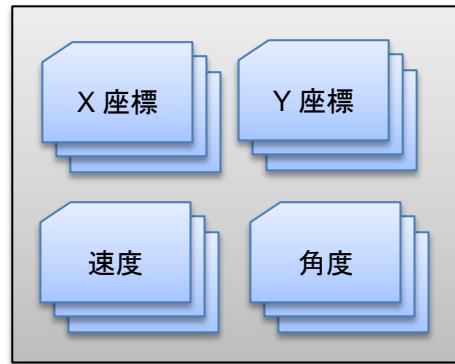
```
{  
    playerY -= PLAYER_SPEED;  
}  
  
if (keyState.IsKeyDown(Keys.Down))  
{  
    playerY += PLAYER_SPEED;  
}  
  
if (playerX < SCREEN_LEFT)  
{  
    playerX = SCREEN_LEFT;  
}  
  
if (playerX > SCREEN_RIGHT)  
{  
    playerX = SCREEN_RIGHT;  
}  
  
if (playerY < SCREEN_TOP)  
{  
    playerY = SCREEN_TOP;  
}  
  
if (playerY < SCREEN_BOTTOM)  
{  
    playerY = SCREEN_BOTTOM;  
}  
  
base.Update(gameTime);  
}
```

## ステップ 2: データごとにまとめよう

### 構造体

このゲームでは最大 100 個の弾が自機目がけて襲い掛かってきます。100 個の弾を同時に動かすためには、100 個分の位置、100 個分の速度、100 個分の角度が必要となります。さらにこのゲームではホーミング弾かどうかと、ホーミングの残り持続時間もそれぞれ 100 個分必要となりますね。ところで、これらのデータにアクセスするためには、どのようにやっていたか思い出してください。例えばここでは 58 番目の弾のデータを取り出すと仮定しましょう。この処理の流れを日本語で示すと大体下記のような感じだったはずです。

1. X 座標一覧のうち 58 番目の値を取り出す。
2. Y 座標一覧のうち 58 番目の値を取り出す。
3. 速度一覧のうち 58 番目の値を取り出す。
4. 角度一覧のうち 58 番目の値を取り出す。
5. ホーミングフラグ一覧のうち 58 番目の値を取り出す。
6. ホーミング持続時間残数一覧のうち 58 番目の値を取り出す。



これでもまあ、悪くはないと思いますが、下記のやり方の方がスマートに見えませんか？

1. 弾情報ファイル一覧のうち、58 番目の弾情報データ（以下データ A とする）を取り出す。
2. データ A から X 座標を取り出す。
3. データ A から Y 座標を取り出す。
4. データ A から速度を取り出す。
5. データ A から角度を取り出す。
6. データ A からホーミングフラグを取り出す。
7. データ A からホーミング持続時間残数を取り出す。



一つのデータに複数の値を入れるにはどうすればよいでしょうか？プログラミングの基本をある程度勉強してきた方なら気付くかもしれません、構造体を使えばよいのです。早速 Enemy.cs と言うファイルを作成してプロジェクトに追加し、その中に構造体 Enemy を作ってみましょう。なお、C#では 1 ファイル 1 構造体などとの縛りはないため、Game1.cs の中に Enemy 構造体を書いてしまっても問題ありません。しかし、今後構造体（そして、クラス）が大量に増えることが予想出来たため、そうすると

Game1.cs が膨れ上がってしまうのを少しでも防ぐためにも、原則として 1 ファイルに 1 構造体に心掛けた方が良いでしょう。

今までゲームクラスでは弾の X 座標・Y 座標・速度・角度などを個別に管理していましたが、このサンプル (Sample1\_05) では弾情報の構造体のみを管理すればよいようになります。

構造体の宣言は Game1.cs とは別ファイルにするとよいでしょう。そのクラスでしか使わず、外部に公開したくないなど、よほど強い理由がない限りは極力 1 ファイル 1 クラス、または 1 構造体にすることを心がけましょう。この場合、ファイル名はクラス名、または構造体名に合わせます。また、もしフォルダを刻みたい場合、名前空間名に合わせるとよいでしょう。

ではこの調子で残りのコードを書いていきます。敵以外にもいくつかのデータを構造体化してみました。こうすると今まで散乱していたデータ一覧が、ある程度まとまって見えるでしょう。

```
/// <summary>敵機の情報。</summary>
struct Enemy
{
    /// <summary>大きさ。</summary>
    public const float SIZE = 32;

    /// <summary>最大数。</summary>
    public const int MAX = 100;

    /// <summary>ホーミング確率。</summary>
    public const int HOMING_PERCENTAGE = 20;

    /// <summary>ホーミング時間。</summary>
    public const int HOMING_LIMIT = 60;

    /// <summary>X 座標。</summary>
    public float x;

    /// <summary>Y 座標。</summary>
    public float y;

    /// <summary>移動速度。</summary>
    public float speed;

    /// <summary>移動角度。</summary>
    public double angle;

    /// <summary>ホーミング対応かどうか。</summary>
    public bool homing;

    /// <summary>ホーミング有効時間。</summary>
    public int homingAmount;
}
```

```
// Graphics.cs

/// <summary>スプライト バッチやコンテンツなど描画周りのデータ一覧。</summary>
struct Graphics
{
    /// <summary>スプライト バッチ。</summary>
    public SpriteBatch spriteBatch;

    /// <summary>キャラクタ用画像。</summary>
    public Texture2D gameThumbnail;

    /// <summary>フォント画像。</summary>
    public SpriteFont spriteFont;
}
```

```
// Player.cs

/// <summary>自機の情報。</summary>
struct Player
{
    /// <summary>大きさ。</summary>
    public const float SIZE = 64;

    /// <summary>移動速度。</summary>
    public const float SPEED = 3;

    /// <summary>自機の初期残機。</summary>
    public const int DEFAULT_AMOUNT = 2;

    /// <summary>ミス猶予(残機)数。</summary>
    public int amount;

    /// <summary>X 座標。</summary>
    public float x;
```

```
/// <summary>Y 座標。</summary>
public float y;
}
```

```
// Score.cs

/// <summary>スコア情報。</summary>
struct Score
{
    /// <summary>エクステンドの閾値。</summary>
    public const int EXTEND_THRESHOLD = 500;

    /// <summary>現在のスコア。</summary>
    public int now;

    /// <summary>前フレームのスコア。</summary>
    public int prev;

    /// <summary>ハイスコア。</summary>
    public int highest;
}
```

```
/// <summary>
/// This is the main type for your game
/// </summary>
public class Game1
    : Game
{
    /// <summary>画像サイズ。</summary>
    const float RECT = 64;

    /// <summary>画面横幅。</summary>
    const float SCREEN_WIDTH = 800;

    /// <summary>画面縦幅。</summary>
```

```
const float SCREEN_HEIGHT = 600;

/// <summary>画面左端。</summary>
const float SCREEN_LEFT = 0;

/// <summary>画面上端。</summary>
const float SCREEN_TOP = 0;

/// <summary>画面右端。</summary>
const float SCREEN_RIGHT = SCREEN_LEFT + SCREEN_WIDTH;

/// <summary>画面下端。</summary>
const float SCREEN_BOTTOM = SCREEN_TOP + SCREEN_HEIGHT;

/// <summary>ゲーム中かどうか。</summary>
bool game;

/// <summary>ゲームの進行カウンタ。</summary>
int counter;

/// <summary>描画周りデータ。</summary>
Graphics graphics = new Graphics();

/// <summary>スコア データ。</summary>
Score score = new Score();

/// <summary>自機 データ。</summary>
Player player = new Player();

/// <summary>敵機一覧データ。</summary>
Enemy[] enemies = new Enemy[Enemy.MAX];

/// <summary>Constructor.</summary>
public Game1()
{
    new GraphicsDeviceManager(this);
```

```

Content.RootDirectory = "Content";
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    graphics.spriteBatch = new SpriteBatch(GraphicsDevice);

    graphics.gameThumbnail = Content.Load<Texture2D>("GameThumbnail");
    graphics.spriteFont = Content.Load<SpriteFont>("SpriteFont");
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    KeyboardState keyState = Keyboard.GetState();
    if (game)
    {
        movePlayer(keyState);
        createEnemy();
        if (enemyMoveAndHitTest())
        {
            enemyReset();
        }
        counter++;
    }
    else
    {
}

```

```
        updateTitle(keyState);
    }

    base.Update(gameTime);
}

/// <summary>自機を移動します。</summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void movePlayer(KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Left))
    {
        player.x -= Player.SPEED;
    }
    if (keyState.IsKeyDown(Keys.Right))
    {
        player.x += Player.SPEED;
    }
    if (keyState.IsKeyDown(Keys.Up))
    {
        player.y -= Player.SPEED;
    }
    if (keyState.IsKeyDown(Keys.Down))
    {
        player.y += Player.SPEED;
    }
    if (player.x < SCREEN_LEFT)
    {
        player.x = SCREEN_LEFT;
    }
    if (player.x > SCREEN_RIGHT)
    {
        player.x = SCREEN_RIGHT;
    }
    if (player.y < SCREEN_TOP)
    {
        player.y = SCREEN_TOP;
    }
}
```

```

}

if (player.y > SCREEN_BOTTOM)
{
    player.y = SCREEN_BOTTOM;
}
}

/// <summary>敵機を作成します。</summary>
private void createEnemy()
{
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0)
    {
        const float AROUND_HALF = SCREEN_WIDTH + SCREEN_HEIGHT;
        const float AROUND_HALF_QUARTER = SCREEN_WIDTH * 2 + SCREEN_HEIGHT;
        const int AROUND = (int)AROUND_HALF * 2;
        for (int i = 0; i < Enemy.MAX; i++)
        {
            if ((enemies[i].x > SCREEN_RIGHT || enemies[i].x < SCREEN_LEFT) &&
                (enemies[i].y > SCREEN_BOTTOM || enemies[i].y < SCREEN_TOP))
            {
                Random rnd = new Random();
                int p = rnd.Next(AROUND);
                if (p < SCREEN_WIDTH || p >= AROUND_HALF &&
                    p < AROUND_HALF_QUARTER)
                {
                    enemies[i].x = p % SCREEN_WIDTH;
                    enemies[i].y = p < AROUND_HALF ? 0 : SCREEN_HEIGHT;
                }
                else
                {
                    enemies[i].x = p < AROUND_HALF ? 0 : SCREEN_WIDTH;
                    enemies[i].y = p % SCREEN_HEIGHT;
                }
                enemies[i].speed = rnd.Next(1, 3) + counter * 0.001f;
                enemies[i].angle = Math.Atan2(
                    player.y - enemies[i].y, player.x - enemies[i].x);
            }
        }
    }
}

```

```

        enemies[i].homing = rnd.Next(100) < Enemy.HOMING_PERCENTAGE;
        enemies[i].homingAmount =
            enemies[i].homing ? Enemy.HOMING_LIMIT : 0;
        score.now += 10;
        if (score.now % Score.EXTEND_THRESHOLD <
            score.prev % Score.EXTEND_THRESHOLD)
        {
            player.amount++;
        }
        score.prev = score.now;
        if (score.highest < score.now)
        {
            score.highest = score.now;
        }
        break;
    }
}
}

/// <summary>敵機の移動、及び接触判定をします。</summary>
/// <returns>接触した場合、true。</returns>
private bool enemyMoveAndHitTest()
{
    bool hit = false;
    const float HITAREA = Player.SIZE * 0.5f + Enemy.SIZE * 0.5f;
    for (int i = 0; i < Enemy.MAX; i++)
    {
        if (Math.Abs(player.x - enemies[i].x) < HITAREA &&
            Math.Abs(player.y - enemies[i].y) < HITAREA)
        {
            hit = true;
            game = --player.amount >= 0;
            break;
        }
        if (--enemies[i].homingAmount > 0)

```

```

    {
        enemies[i].angle = Math.Atan2(
            player.y - enemies[i].y, player.x - enemies[i].x);
    }

    enemies[i].x += (float) Math.Cos(enemies[i].angle) * enemies[i].speed;
    enemies[i].y += (float) Math.Sin(enemies[i].angle) * enemies[i].speed;
}

return hit;
}

/// <summary>敵機を初期状態にリセットします。</summary>
private void enemyReset()
{
    const float FIRST_POSITION = -Enemy.SIZE;
    for (int i = 0; i < Enemy.MAX; i++)
    {
        enemies[i].x = FIRST_POSITION;
        enemies[i].y = FIRST_POSITION;
        enemies[i].speed = 0;
    }
}

/// <summary>タイトル画面を更新します。</summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void updateTitle(KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Escape))
    {
        Exit();
    }

    if (keyState.IsKeyDown(Keys.Space))
    {
        // ゲーム開始
        game = true;
        player.x = SCREEN_LEFT + SCREEN_WIDTH * 0.5f;
        player.y = SCREEN_TOP + SCREEN_HEIGHT * 0.5f;
    }
}

```

```

        counter = 0;
        score.now = 0;
        score.prev = 0;
        player.amount = Player.DEFAULT_AMOUNT;
        enemyReset();
    }
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    graphics.spriteBatch.Begin();
    if (game)
    {
        drawGame();
    }
    else
    {
        drawTitle();
    }
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "HISCORE: " + score.highest.ToString(),
        new Vector2(0, 560), Color.Black);
    graphics.spriteBatch.End();
    base.Draw(gameTime);
}

/// <summary>タイトル画面を描画します。</summary>
private void drawTitle()
{

```

```

graphics.spriteBatch.DrawString(
    graphics.spriteFont, "SAMPLE 1", new Vector2(200, 100),
    Color.Black, 0f, Vector2.Zero, 5f, SpriteEffects.None, 0f);
graphics.spriteBatch.DrawString(graphics.spriteFont,
    "PUSH SPACE KEY.", new Vector2(340, 400), Color.Black);
}

/// <summary>ゲーム画面を描画します。</summary>
private void drawGame()
{
    drawPlayer();
    drawEnemy();
    drawHUD();
}

/// <summary>自機を描画します。</summary>
private void drawPlayer()
{
    graphics.spriteBatch.Draw(graphics.gameThumbnail,
        new Vector2(player.x, player.y), null, Color.White, 0f,
        new Vector2(RECT * 0.5f), Player.SIZE / RECT, SpriteEffects.None, 0f);
}

/// <summary>敵機を描画します。</summary>
private void drawEnemy()
{
    const float SCALE = Enemy.SIZE / RECT;
    Vector2 origin = new Vector2(RECT * 0.5f);
    for (int i = 0; i < Enemy.MAX; i++)
    {
        graphics.spriteBatch.Draw(
            graphics.gameThumbnail, new Vector2(enemies[i].x, enemies[i].y),
            null, enemies[i].homing ? Color.Orange : Color.Red,
            0f, origin, SCALE, SpriteEffects.None, 0f);
    }
}

```

```
/// <summary>HUD を描画します。</summary>
private void drawHUD()
{
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "SCORE: " + score.now.ToString(),
        new Vector2(300, 560), Color.Black);
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "PLAYER: " + new string('*', player.amount),
        new Vector2(600, 560), Color.Black);
}
}
```

## あるものは使おう

「車輪の再発明」と言う慣用句をご存知ですか？その名の通り、すでに存在しているものをもう一度作ってしまうことを指します。例えばパソコンで動作するOSと聞くと何を思い浮かべますか？読者の皆さんなら、恐らく真っ先にWindowsが出てくるかと思います。他にも挙げてみましょう。Macの標準OS「OS X」があります。他にもLinuxやBSDなどUNIXベースのOSもあります(OS XもBSDの仲間です)。もうちょっとマイナーどころとなると、BeOSやMonaOSなんかもありますね。車輪の再発明とは、これらのOSが信用できない、あるいはほかの理由で自家製OSを作ってしまうことです。これがもっとひどくなると、社内では自家製・または自社製ソフトウェアしか価値を認めず、存在を認めず、当然運用も認めない<sup>17</sup>などとなり、「自前主義」とか「NIH症候群」などと呼ばれるようになります。

車輪の再発明は、学習・研究目的で行う場合は、ブラックボックスの内部構造が理解できるなど、あなたにとってメリットとなることが多いでしょう。但し、それ以外の目的で行うことはデメリットの方が大きく、お勧めできません。例えば「使いたいソフトウェアの品質が目に余るほど低く、またはそれが望んでいるものの形とかなりかけ離れているため、その代替版を作る」などの明確な目的がある場合を除き、再発明版の方がクオリティの低いソフトウェアとなってしまう場合が殆どです<sup>18</sup>。私は、そのような車輪の再発明に労力を割くくらいなら、類似しているソフトウェアを探し、仕様を把握してそれに合わせた構造を作る方が、有意義な時間を使えると考えます。

少々脱線しかけたので話を戻しましょう。前項では弾の情報を管理するために、構造体を作りましたね。そこではX座標とY座標、速度と角度、そしてホーミング制御用のデータが2つの合計6つのデータがありました。しかしXNAに標準で入っているVector2型を使用すると、これが4つのデータで済みます。Vector2型にはXとYの二つの変数があり、それを使い位置情報を保持できるだけでなく、速度や方角を保持するためにも活用することができます。型名を見る限り、むしろこちらの使い方が本来のものであることに気づきますね。

下記のソースコード(Sample1\_06)では、自機と弾の座標と速度角度管理にVector2型を使用しています。速度角度計算に三角関数を使わず、ベクトル計算で代用していたり、さらに処理を分割したりするところなど、細部でいろいろな変更を施していますので、その辺もよく比較してみてください。

---

<sup>17</sup> 2008年頃にそういう現場の経験もあります。ちなみにOSはWindows2000でした。Word97やExcel97を使っていて、96年頃のLotus Notesを使っていました。堂々とNIHを謳っていた割に、この辺は別にいいのかなあ。どうでもいい雑学でした。

<sup>18</sup> これを「四角い車輪の再発明」という人も、少なからずいるようです。

```
// Player.cs

/// <summary>自機の情報。</summary>
struct Player
{
    /// <summary>大きさ。</summary>
    public const float SIZE = 64;

    /// <summary>移動速度。</summary>
    public const float SPEED = 3;

    /// <summary>自機の初期残機。</summary>
    public const int DEFAULT_AMOUNT = 2;

    /// <summary>入力を受け付けるキー一覧。</summary>
    public Keys[] acceptInputKeyList;

    /// <summary>キー入力に対応した移動方向。</summary>
    public Dictionary<Keys, Vector2> velocity;

    /// <summary>ミス猶予(残機)数。</summary>
    public int amount;

    /// <summary>現在座標。</summary>
    public Vector2 position;
}
```

```
//Enemy.cs

/// <summary>敵機の情報。</summary>
struct Enemy
{
    /// <summary>大きさ。</summary>
    public const float SIZE = 32;
```

```
/// <summary>最大数。</summary>
public const int MAX = 100;

/// <summary>ホーミング確率。</summary>
public const int HOMING_PERCENTAGE = 20;

/// <summary>ホーミング時間。</summary>
public const int HOMING_LIMIT = 60;

/// <summary>現在座標。</summary>
public Vector2 position;

/// <summary>移動速度と方角。</summary>
public Vector2 velocity;

/// <summary>ホーミング対応かどうか。</summary>
public bool homing;

/// <summary>ホーミング有効時間。</summary>
public int homingAmount;
}
```

```
// Game1.cs

/// <summary>
/// This is the main type for your game
/// </summary>
public class Game1
: Game
{
    // .....(中略).....
}
```

```

/// <summary>
/// Allows the game to perform any initialization it needs to before starting to run.
/// This is where it can query for any required services and load any non-graphic
/// related content. Calling base.Initialize will enumerate through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    player.acceptInputKeyList =
        new Keys[] { Keys.Up, Keys.Down, Keys.Left, Keys.Right };
    player.velocity = new Dictionary<Keys, Vector2>();
    player.velocity.Add(Keys.Up, new Vector2(0, -Player.SPEED));
    player.velocity.Add(Keys.Down, new Vector2(0, Player.SPEED));
    player.velocity.Add(Keys.Left, new Vector2(-Player.SPEED, 0));
    player.velocity.Add(Keys.Right, new Vector2(Player.SPEED, 0));
    base.Initialize();
}

// .....(中略)......

/// <summary>自機を移動します。</summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void movePlayer(KeyboardState keyState)
{
    Vector2 prev = player.position;
    for (int i = 0; i < player.acceptInputKeyList.Length; i++)
    {
        Keys key = player.acceptInputKeyList[i];
        if (keyState.IsKeyDown(key))
        {
            player.position += player.velocity[key];
        }
    }
    if (!SCREEN.Contains((int)player.position.X, (int)player.position.Y))
    {
        player.position = prev;
    }
}

```

```

        }

    }

// .....(中略).....  

/// <summary>
/// 敵機を作成します。
/// </summary>
private void createEnemy()
{
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0)
    {
        float AROUND_HALF = SCREEN.Width + SCREEN.Height;
        float AROUND_HALF_QUARTER = SCREEN.Width * 2 + SCREEN.Height;
        int AROUND = (int)AROUND_HALF * 2;
        for (int i = 0; i < Enemy.MAX; i++)
        {
            if (!SCREEN.Contains(
                (int)enemies[i].position.X, (int)enemies[i].position.Y))
            {
                Random rnd = new Random();
                int p = rnd.Next(AROUND);
                if (p < SCREEN.Width || p >= AROUND_HALF &&
                    p < AROUND_HALF_QUARTER)
                {
                    enemies[i].position.X = p % SCREEN.Width;
                    enemies[i].position.Y = p < AROUND_HALF ? 0 : SCREEN.Height;
                }
                else
                {
                    enemies[i].position.X = p < AROUND_HALF ? 0 : SCREEN.Width;
                    enemies[i].position.Y = p % SCREEN.Height;
                }
                enemies[i].velocity = createVelocity(
                    enemies[i].position, rnd.Next(1, 3) + counter * 0.001f);
                enemies[i].homing = rnd.Next(100) < Enemy.HOMING_PERCENTAGE;
            }
        }
    }
}

```

```

        enemies[i].homingAmount = Enemy.HOMING_LIMIT;
        addScore(10);
        break;
    }
}
}

}

// .....(中略).....



/// <summary>敵機の移動、及び接触判定をします。</summary>
/// <returns>接触した場合、true。</returns>
private bool enemyMoveAndHitTest()
{
    bool hit = false;

    const float HITAREA = Player.SIZE * 0.5f + Enemy.SIZE * 0.5f;
    const float HITAREA_SQUARED = HITAREA * HITAREA;

    for (int i = 0; i < Enemy.MAX; i++)
    {
        if (Vector2.DistanceSquared(enemies[i].position, player.position) <
            HITAREA_SQUARED)
        {
            hit = true;
            game = --player.amount >= 0;
            break;
        }
        if (enemies[i].homing && --enemies[i].homingAmount > 0)
        {
            enemies[i].velocity =
                createVelocity(enemies[i].position, enemies[i].velocity.Length());
        }
        enemies[i].position += enemies[i].velocity;
    }
    return hit;
}

```

```

/// <summary>敵機の移動速度と方角を計算します。</summary>
/// <param name="position">位置。</param>
/// <param name="speed">速度。</param>
/// <returns>計算された敵機の新しい移動速度と方角。</returns>
private Vector2 createVelocity(Vector2 position, float speed)
{
    Vector2 velocity = player.position - position;
    if (velocity == Vector2.Zero)
    {
        // 長さが0だと単位ベクトル計算時にNaNが出るため対策
        velocity = Vector2.UnitX;
    }
    velocity.Normalize();
    return (velocity * speed);
}

/// <summary>敵機を初期状態にリセットします。</summary>
private void enemyReset()
{
    Vector2 firstPosition = new Vector2(-Enemy.SIZE);
    for (int i = 0; i < Enemy.MAX; i++)
    {
        enemies[i].position = firstPosition;
        enemies[i].velocity = Vector2.Zero;
    }
}

// .....(中略)......

/// <summary>自機を描画します。</summary>
private void drawPlayer()
{
    graphics.spriteBatch.Draw(graphics.gameThumbnail, player.position,
        null, Color.White, 0f, new Vector2(RECT * 0.5f),
        Player.SIZE / RECT, SpriteEffects.None, 0f);
}

```

```
/// <summary>敵機を描画します。</summary>
private void drawEnemy()
{
    const float SCALE = Enemy.SIZE / RECT;
    Vector2 origin = new Vector2(RECT * 0.5f);
    for (int i = 0; i < Enemy.MAX; i++)
    {
        graphics.spriteBatch.Draw(graphics.gameThumbnail, enemies[i].position,
            null, enemies[i].homing ? Color.Orange : Color.Red,
            0f, origin, SCALE, SpriteEffects.None, 0f);
    }
}

// .....(中略).....
}
```

## オブジェクト指向への第一歩

前々回で座標や速度などの弾情報を構造体という形で 1 つのデータとしてまとめました。ところで、構造体にはどのようなものが入れられるのでしょうか？列挙してみましょう。

- 変数
- 関数(メソッド)
- プロパティやインデクサ(アクセサ)
- イベント

構造体には上記のように変数以外のもの、例えば関数なども入れることができます。事実上クラスとほぼ同じものが入れられるのです。そうすると、前章で分割したサブルーチンの一部は、構造体の中に埋め込んでしまった方が自然に見えますよね？このように構造体の中に埋め込むことは、他のメリット（カプセル化やポリモーフィズムなど）もあるのですが、それは次節で紹介しますので、ここでは早速コードを書き直してみましょう。以下に Score.cs の修正したコードを掲載します。

```
// Score.cs

/// <summary>スコア情報。</summary>
struct Score
{
    /// <summary>エクステンドの閾値。</summary>
    public const int EXTEND_THRESHOLD = 500;

    /// <summary>現在のスコア。</summary>
    public int now;

    /// <summary>前フレームのスコア。</summary>
    public int prev;

    /// <summary>ハイスコア。</summary>
    public int highest;

    /// <summary>スコアをリセットします。</summary>
    public void reset()
    {
        now = 0;
        prev = 0;
    }

    /// <summary>スコアを加算します。</summary>
    /// <param name="score">加算されるスコア値。</param>
    public bool add(int score)
    {
        now += score;
        bool extend =
            now % EXTEND_THRESHOLD < prev % EXTEND_THRESHOLD;
        prev = now;
        if (highest < now)
        {
            highest = now;
        }
    }
}
```

```
    }

    return extend;
}

}

}
```

これでスコア情報に加算するメソッドと、スコアをリセットするメソッドが追加されました。使い方は今まで構造体にアクセスしていたのと全く同じ方法で実行することができます。

なお、Sample1\_07 では、一部サブルーチンを各構造体へ埋め込んだだけでなく、同時に敵以外のすべての構造体をクラスに置き換えた上、敵クラスと敵一覧クラスに分割しています。C#における構造体とクラスの違いや、置き換えることによりどのようなメリット・デメリットがあるかなど、各自で挙げてみましょう。

```
// クラス/構造体メソッドを
// 呼び出す方法

Score score = new Score();

// .....(中略)......

// スコアに 10 加算します。
score.add(10);
```

## カプセル化

前節では、変数のほかにメソッドなども、各クラス（または構造体）の中へ入れてしまいました。その結果、一部の変数などはそのクラスの中でしか使用しないものなども出てきました。そのような変数は積極的に隠してしまいましょう。また、クラス外でも使われる変数についても、極力直接書き換えられないようにすべきです。

なぜこのような、不便になりかねないようなことをすべきなのでしょうか？たとえばプレイヤーのクラス（以下 Player クラス）には残機情報「amount」が入っていますね。これをインクリメント<sup>19</sup>することによりエクステンドし、デクリメントすることでミス扱いにしています。この制御はメインである Game1 クラス内で行っています。また、残機情報は完全に公開されていて、Player クラスのオブジェクトにアクセスできれば誰でも残機情報を改変することができます。極端に言ってしまうと、Game1 クラスの気分次第で、プレイヤーの残機を 1 個減らすべきところで 2 個減らしたり、強制的に 0 にしたりも出来るわけです。Player クラスさんにとって、これは迷惑な話ですよね？開発者にとっても、余りに何でも出来すぎてしまうと思わぬバグの原因となってしまう可能性があります。

例えば二人のプログラマが協力して一つのゲームを作っているところを想像してください。私が前節で作った Player クラスを操作する機能を誰かが作ると仮定しましょう。Player クラスには現在位置を示す position という変数があり、その変数の値は好き勝手に変更できるようです。するとそのプログラマは、position は禁止されていない以上は好き勝手に変更できるものだろうと考えて、直接値を挿入してしまうものです。これは困りますね。折角 Player クラス内で position を制御するコードを書いたのに、これではその意味がなくなってしまいます。こうなら

```
/// <summary>
/// 残機を減らします。
/// </summary>
/// <returns>ゲームが続行可能である場合、true。</returns>
public bool miss()
{
    resetPosition();
    return --amount >= 0;
}

/// <summary>
/// 現在位置を初期化します。
/// </summary>
private void resetPosition()
{
    Point center = Game1.SCREEN.Center;
    position = new Vector2(center.X, center.Y);
}
```

<sup>19</sup> 整数型の値を 1 増やすことをインクリメントと言います。逆に 1 減らすことをデクリメントと言います。

ないようにするためにには、position の書き換えをコードレベルで禁止すればよいのです。アクセス指定子 private を使用すれば、position は Player クラスの中からしか見ることができなくなります。もしどうしても書き換えたい、たとえば「ミスと判定されたらプレイヤーを画面中央に戻したい」場合、その処理を Player クラスの中に追加してあげましょう。このようにデータを隠蔽したり、場合によってはメソッドなどを隠蔽したりすることを「カプセル化」と言います。全面的にデータ隠蔽を施したサンプルを用意しましたので、前節のサンプルとの違いをよく比較してみてください。(Sample1\_08)

```
// Game1.cs

/// <summary>
/// This is the main type for your game
/// </summary>

public class Game1
    : Game
{
    /// <summary>画面矩形情報。</summary>
    public static readonly Rectangle SCREEN = new Rectangle(0, 0, 800, 600);

    /// <summary>ゲーム中かどうか。</summary>
    private bool game;

    /// <summary>ゲームの進行カウンタ。</summary>
    private int counter;

    /// <summary>描画周りデータ。</summary>
    private Graphics graphics;

    /// <summary>スコア データ。</summary>
    private readonly Score score = new Score();

    /// <summary>敵機一覧データ。</summary>
    private readonly Enemies enemies = new Enemies();

    /// <summary>自機データ。</summary>
    private readonly Player player = new Player();
```

```

/// <summary>
/// Constructor.
/// </summary>
public Game1()
{
    new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    graphics = new Graphics(this);
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    KeyboardState keyState = Keyboard.GetState();
    if (game)
    {
        player.move(keyState);
        createEnemy();
        if (enemies.moveAndHitTest(player.position))
        {
            game = player.miss();
        }
        counter++;
    }
}

```

```

else
{
    updateTitle(keyState);
}

base.Update(gameTime);
}

/// <summary>
/// 敵機を作成します。
/// </summary>

private void createEnemy()
{
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0 &&
        enemies.create(player.position, counter * 0.001f) &&
        score.add(10))
    {
        player.extend();
    }
}

/// <summary>
/// タイトル画面を更新します。
/// </summary>

/// <param name="keyState">現在のキー入力状態。</param>

private void updateTitle(KeyboardState keyState)
{
    if (keyState.IsKeyDown(Keys.Escape))
    {
        Exit();
    }

    if (keyState.IsKeyDown(Keys.Space))
    {
        // ゲーム開始
        game = true;
        counter = 0;
        score.reset();
    }
}

```

```
player.reset();
enemies.reset();
}

}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    graphics.spriteBatch.Begin();
    if (game)
    {
        drawGame();
    }
    else
    {
        drawTitle();
    }
    drawHUD(game);
    graphics.spriteBatch.End();
    base.Draw(gameTime);
}

/// <summary>
/// HUD を描画します。
/// </summary>
/// <param name="all">全情報を描画するかどうか。</param>
private void drawHUD(bool all)
{
    if (all)
    {
```

```

        graphics.spriteBatch.DrawString(graphics.spriteFont,
            "PLAYER: " + player.amountString,
            new Vector2(600, 560), Color.Black);
    }

    score.draw(graphics, all);
}

/// <summary>
/// タイトル画面を描画します。
/// </summary>
private void drawTitle()
{
    graphics.spriteBatch.DrawString(
        graphics.spriteFont, "SAMPLE 1", new Vector2(200, 100),
        Color.Black, 0f, Vector2.Zero, 5f, SpriteEffects.None, 0f);
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "PUSH SPACE KEY.", new Vector2(340, 400), Color.Black);
}

/// <summary>
/// ゲーム画面を描画します。
/// </summary>
private void drawGame()
{
    player.draw(graphics);
    enemies.draw(graphics);
}
}

```

```

// Graphics.cs

/// <summary>
/// スプライト バッチやコンテンツなど描画周りのデータ一覧。
/// </summary>
class Graphics
{

```

```
/// <summary>画像サイズ。</summary>
public const float RECT = 64;

/// <summary>スプライト バッチ。</summary>
public readonly SpriteBatch spriteBatch;

/// <summary>キャラクタ用画像。</summary>
public readonly Texture2D gameThumbnail;

/// <summary>フォント画像。</summary>
public readonly SpriteFont spriteFont;

/// <summary>
/// コンストラクタ。
/// コンテンツを読み込みます。
/// </summary>
/// <param name="game">ゲーム メイン オブジェクト。</param>
public Graphics(Game game)
{
    spriteBatch = new SpriteBatch(game.GraphicsDevice);
    gameThumbnail = game.Content.Load<Texture2D>("GameThumbnail");
    spriteFont = game.Content.Load<SpriteFont>("SpriteFont");
}

}
```

```
// Score.cs

/// <summary>
/// スコア情報。
/// </summary>
class Score
{

    /// <summary>エクステンドの閾値。</summary>
    private const int EXTEND_THRESHOLD = 500;
```

```
/// <summary>前フレームのスコア。</summary>
private int prev;

/// <summary>現在のスコア。</summary>
public int now
{
    get;
    private set;
}

/// <summary>ハイスコア。</summary>
public int highest
{
    get;
    private set;
}

/// <summary>
/// スコアをリセットします。
/// </summary>
public void reset()
{
    now = 0;
    prev = 0;
}

/// <summary>
/// スコアを加算します。
/// </summary>
/// <param name="score">加算されるスコア値。</param>
/// <returns>エクステンド該当となる場合、true。</returns>
public bool add(int score)
{
    now += score;
    bool extend = now % EXTEND_THRESHOLD < prev % EXTEND_THRESHOLD;
    prev = now;
}
```

```
if (highest < now)
{
    highest = now;
}
return extend;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
/// <param name="nowScore">現在のスコアも描画するかどうか。</param>
public void draw(Graphics graphics, bool nowScore)
{
    if (nowScore)
    {
        graphics.spriteBatch.DrawString(graphics.spriteFont,
            "SCORE: " + now.ToString(),
            new Vector2(300, 560), Color.Black);
    }
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "HISCORE: " + highest.ToString(), new Vector2(0, 560), Color.Black);
}
```

```
// Player.cs

/// <summary>
/// 自機の情報。
/// </summary>
class Player
{

    /// <summary>大きさ。</summary>
    public const float SIZE = 64;
```

```
/// <summary>移動速度。</summary>
private const float SPEED = 3;

/// <summary>自機の初期残機。</summary>
private const int DEFAULT_AMOUNT = 2;

/// <summary>入力を受け付けるキー一覧。</summary>
private readonly Keys[] acceptInputKeyList;

/// <summary>キー入力に対応した移動方向。</summary>
private readonly Dictionary<Keys, Vector2> velocity;

/// <summary>ミス猶予(残機)数。</summary>
private int m_amount;

/// <summary>現在座標。</summary>
public Vector2 position
{
    get;
    private set;
}

/// <summary>ミス猶予(残機)数。</summary>
public int amount
{
    get
    {
        return m_amount;
    }
    private set
    {
        m_amount = value;
        amountString = value < 0 ? string.Empty : new string('*', value);
    }
}
```

```

/// <summary>ミス猶予(残機)数の文字列による表現。</summary>
public string amountString
{
    get;
    private set;
}

/// <summary>
/// 各種値を初期化します。
/// </summary>
public Player()
{
    acceptInputKeyList =
        new Keys[] { Keys.Up, Keys.Down, Keys.Left, Keys.Right };
    velocity = new Dictionary<Keys, Vector2>();
    velocity.Add(Keys.Up, new Vector2(0, -Player.SPEED));
    velocity.Add(Keys.Down, new Vector2(0, Player.SPEED));
    velocity.Add(Keys.Left, new Vector2(-Player.SPEED, 0));
    velocity.Add(Keys.Right, new Vector2(Player.SPEED, 0));
}

/// <summary>
/// 残機を増やします。
/// </summary>
public void extend()
{
    amount++;
}

/// <summary>
/// 残機を減らします。
/// </summary>
/// <returns>ゲームが続行可能である場合、true。</returns>
public bool miss()
{
    // ミスするとプレイヤーは元の座標へと戻る
}

```

```
    resetPosition();

    return --amount >= 0;
}

/// <summary>
/// 現在位置を初期化します。
/// </summary>

private void resetPosition()
{
    Point center = Game1.SCREEN.Center;
    position = new Vector2(center.X, center.Y);
}

/// <summary>
/// 座標や残機情報を初期化します。
/// </summary>

public void reset()
{
    resetPosition();
    amount = DEFAULT_AMOUNT;
}

/// <summary>
/// キー入力に応じて移動します。
/// </summary>
/// <param name="keyState">現在のキー入力状態。</param>

public void move(KeyboardState keyState)
{
    Vector2 prev = position;
    for (int i = 0; i < acceptInputKeyList.Length; i++)
    {
        Keys key = acceptInputKeyList[i];
        if (keyState.IsKeyDown(key))
        {
            position += velocity[key];
        }
    }
}
```

```
if (!Game1.SCREEN.Contains((int)position.X, (int)position.Y))
{
    position = prev;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position,
        null, Color.White, 0f, new Vector2(Graphics.RECT * 0.5f),
        Player.SIZE / Graphics.RECT, SpriteEffects.None, 0f);
}
```

```
// Enemies.cs

/// <summary>
/// 敵機の情報。
/// </summary>
class Enemies
{

    /// <summary>最大数。</summary>
private const int MAX = 100;

    /// <summary>敵機一覧データ。</summary>
private Enemy[] list = new Enemy[MAX];
```

```
/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
/// <returns>敵機を作成できた場合、true。</returns>
public bool create(Vector2 playerPosition, float speed)
{
    bool result = false;
    for (int i = 0; !result && i < MAX; i++)
    {
        result = list[i].start(playerPosition, speed);
    }
    return result;
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool moveAndHitTest(Vector2 playerPosition)
{
    bool hit = false;
    for (int i = 0; !hit && i < MAX; i++)
    {
        hit = list[i].moveAndHitTest(playerPosition);
    }
    if (hit)
    {
        reset();
    }
    return hit;
}
```

```
/// <summary>
/// 敵機を初期状態にリセットします。
/// </summary>
public void reset()
{
    Vector2 firstPosition = new Vector2(-Enemy.SIZE);
    for (int i = 0; i < MAX; i++)
    {
        list[i].sleep();
    }
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    for (int i = 0; i < MAX; i++)
    {
        list[i].draw(graphics);
    }
}
```

```
// Enemy.cs

/// <summary>
/// 敵機の情報。
/// </summary>
struct Enemy
{

    /// <summary>大きさ。</summary>
    public const float SIZE = 32;
```

```
/// <summary>ホーミング確率。</summary>
private const int HOMING_PERCENTAGE = 20;

/// <summary>ホーミング時間。</summary>
private const int HOMING_LIMIT = 60;

/// <summary>初期位置。</summary>
private static readonly Vector2 firstPosition = new Vector2(-SIZE);

/// <summary>移動速度と方角。</summary>
private Vector2 velocity;

/// <summary>ホーミング対応かどうか。</summary>
private bool homing;

/// <summary>ホーミング有効時間。</summary>
private int homingAmount;

/// <summary>現在座標。</summary>
public Vector2 position
{
    get;
    private set;
}

/// <summary>色。</summary>
public Color color
{
    get;
    private set;
}
```

```

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool moveAndHitTest(Vector2 playerPosition)
{
    const float HITAREA = Player.SIZE * 0.5f + SIZE * 0.5f;
    const float HITAREA_SQUARED = HITAREA * HITAREA;
    bool hit = (HITAREA_SQUARED > Vector2.DistanceSquared(position, playerPosition));
    if (homing && --homingAmount > 0)
    {
        initVelocity(playerPosition, velocity.Length());
    }
    position += velocity;
    return hit;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    const float SCALE = SIZE / Graphics.RECT;
    Vector2 origin = new Vector2(Graphics.RECT * 0.5f);
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position, null, color,
        Of, new Vector2(Graphics.RECT * 0.5f), SCALE, SpriteEffects.None, 0f);
}

/// <summary>
/// 敵機をアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
/// <returns>敵機をアクティブにできた場合、true。</returns>

```

```

public bool start(Vector2 playerPosition, float speed)
{
    bool result = !Game1.SCREEN.Contains((int)position.X, (int)position.Y);
    if (result)
    {
        startForce(playerPosition, speed);
    }
    return result;
}

/// <summary>
/// 敵機を強制的にスリープにします。
/// </summary>
public void sleep()
{
    position = firstPosition;
    velocity = Vector2.Zero;
}

/// <summary>
/// 敵機を強制的にアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
private void startForce(Vector2 playerPosition, float speed)
{
    Random rnd = new Random();
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
}

```

```

pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
}

else
{
    pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
    pos.Y = p % Game1.SCREEN.Height;
}

position = pos;
initVelocity(playerPosition, rnd.Next(1, 3) + speed);
homing = rnd.Next(100) < Enemy.HOMING_PERCENTAGE;
color = homing ? Color.Orange : Color.Red;
homingAmount = Enemy.HOMING_LIMIT;
}

/// <summary>
/// 敵機の移動速度と方角を初期化します。
/// </summary>
/// <param name="playerPosition">プレイヤーの位置。</param>
/// <param name="speed">速度。</param>
private void initVelocity(Vector2 playerPosition, float speed)
{
    Vector2 v = playerPosition - position;
    if (v == Vector2.Zero)
    {
        // 長さが0だと単位ベクトル計算時にNaNが出るため対策
        v = Vector2.UnitX;
    }
    v.Normalize();
    velocity = v * speed;
}
}

```

### ステップ3:「動け！」だけで勝手に動くようにしよう

ステップ2までは、恐らくこれを見ている皆さんも、ある程度自力で実践できていた、という人も多いのではないのでしょうか。実際に、大きなゲームが作れる人とすぐ破綻する人の差は、ここステップ3が理解できているか否か<sup>20</sup>によるものと思います。

余談ですが、本書の中で一番教えたかった場所はここだったりします。今までのステップは、実はステップ3で改良するための地盤づくりにすぎません。極端な話、ステップ3と4さえマスターてしまえば、大抵のゲームの基礎部分は作れるようになると言っても過言ではないでしょう。

---

<sup>20</sup> 尤も、さらに大きなゲームになると、プログラミング以外の要素も多くなってくるのですが……。

## ポリモーフィズム

ところで、オブジェクト指向を構成する性質には、以下の 3 つがあります。

- カプセル化
- インヘリタンス(継承)
- ポリモーフィズム(多態性)

カプセル化は前章でも触れましたね。オブジェクト指向プログラミングの入門本などを見ていると、継承(と、少しだけカプセル化)だけが犬や猫の例えを出してしつこく解説され、ポリモーフィズムはリモコンの電源ボタンなどに例えた概念をちょっとだけ紹介しておしまい、な傾向があつた<sup>21</sup>気がします。でも、オブジェクト指向の一番凄いところはポリモーフィズムだったりするのです。

例えば、前章までに作ってきたサンプルゲームにおいて、ゲームプレイ中に毎フレーム行わなければならない処理は何があるでしょうか？列挙してみましょう。

1. キーの入力状態を更新する。
2. そのキーの入力状態をもとに、自機を移動する。
3. 一定間隔で敵機を作成する。
4. 敵機を一斉に移動する。
5. 敵機と自機との接触判定を行う。
6. その結果接触と判定された場合、自機を一機減らし、ゲーム続行可能か判定する。
7. 画面をクリアする
8. 自機を描画する。
9. 敵機を描画する。
10. 残機数を描画する。
11. スコアを描画する。

こうして改めて列挙してみると、たった 1 フレームの間だけでも、結構やるべきことが多いことに気付きますね。コードの可読性を維持し、思わぬバグを生み出さないためにもちょっと処理量を減らす必要がありそうです。とは言え、コードを見る限り削減できそうな無駄な処理は見当たりません。そうすると次はメソッドの分割が思いつきますが、ここではもう一つの方法を用います。前頁にも挙がっていたポリモーフィズムを使います。

---

<sup>21</sup> 少なくとも、4~5 年前はそうでした。今時の入門本はどうなのでしょう？

実践に先立って軽くポリモーフィズムのおさらいをしましょう。よくある例えですが、テレビとパソコンが目の前にあったとします。一度でも分解したことのある人なら理解できると思いますが、両者の内部構造は全く異なります。しかしその本体に「通電した状態で電源ボタンを押す」と言う手続きを経ることにより、どちらも動作を始めます。つまり、パソコンやテレビの動作原理を知らなくても、「通電する」と「電源ボタンを押す」ことだけさえ理解していればパソコンやテレビは動作を始めることができるのです。

このようにボタンを押すなどの要素、を中身の異なる者に対して共通化することをポリモーフィズムと言います。しかし、これをどうこのサンプルゲームに適用すればよいでしょうか？先ほどまとめた毎フレーム行う処理をデータごとに区切ると、以下のようにまとめられそうです。

- キー入力
- 自機
- 敵機
- スコア

これらのクラスに共通する基底クラス、またはインターフェイスを作り、そこに「更新する」「描画する」の二つのメソッドが存在することを保証しまえばよさそうですね。クラス名は「タスク」辺りがよさそうですね。

ここでは基底クラスではなくインターフェイスを使用し、また更新・描画に加えて「初期化」機能も追加しました。ところでインターフェイスを使用したことはありますか？本書を読んでいるレベルと言うことは恐らく「勉強したものの、使ったことがないから実際どういう者かよくわからない」と言う方が多いのではないの

```
/// <summary>
/// タスク インターフェイス。
/// </summary>
interface ITask
{
    /// <summary>
    /// 初期化を行います。
    /// </summary>
    void reset();

    /// <summary>
    /// 1 フレーム分の更新を行います。
    /// </summary>
    /// <param name="keyState">現在のキー入力状態。</param>
    void update(KeyboardState keyState);

    /// <summary>
    /// 1 フレーム分の描画を行います。
    /// </summary>
    /// <param name="graphics">グラフィック データ。</param>
    void draw(Graphics graphics);
}
```

でしょうか？軽くおさらいをしておきましょう。

インターフェイスとは、異なる者同士が通信するために使う取り決めを指します。例えば、人同士が通信しあうために自然言語があります。つまり自然言語は人対人のインターフェイスとなるわけです。しかし、お互いに話せる言語が一致していないと、たとえ人種や特徴がそっくりでも、その人とはまともに通信できません。一方で、相手は日本語が通じるということが予め判ってさえいれば、人種が違っていると、またどんな容姿が異なっているようと<sup>22</sup>、通信は簡単に行うことができます。これはプログラミングでも同じです。ITask というインターフェイスは先ほどの自然言語に例えると初期化、更新、そして描画と言う三つの命令だけを受け付ける言語の一種である、と言えるでしょう。たとえそこに謎のオブジェクト A があったとしても、もしそれが ITask インターフェイスを実装していると判ってしまえば、中身はよくわからなくても初期化、更新、そして描画を命令することができるのです。つまり、インターフェイスは使う側で、そのオブジェクトはどんなやつだろうと三つの命令を持っていることを保障する証明書のようなものなのです。その一方で、そのインターフェイスを組み込む側は、それに記述された命令、メソッドをすべて実装しないとエラーとなってしまい、プログラムは動作しません。このように使う側に機能の保証、組み込む側に機能の義務をそれぞれ与えるのがプログラミングにおけるインターフェイスの役割です。

少々話が長引いてしまいましたね。では、この ITask インターフェイスを前頁で挙げた四つのクラスに組み込んで、一斉に動かしてみましょう。(Sample1\_09)

---

<sup>22</sup> 極端な話、日本語が通じると判っていれば相手は人間である必要ですらないのです。ドラえもんやアトム、初音ミク(有志が絶対に作ってくれるはずだ！)でも構わないので。この例えはポリモーフィズムを理解するうえでとても重要です。

```

/// <summary>ゲームプレイ画面のタスク一覧。</summary>
private readonly ITask[] taskGame;

/// <summary>
/// Constructor.
/// </summary>
public Game1()
{
    new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    taskTitle = new ITask[] { score, mgrInput };
    taskGame = new ITask[] { enemies, player, score, mgrInput };
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    graphics = new Graphics(this);
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    KeyboardState keyState = mgrInput.keyboardState;
    ITask[] tasks = game ? taskGame : taskTitle;
    for (int i = 0; i < tasks.Length; i++)
    {
        tasks[i].update(keyState);
    }
}

```

```

if (game)
{
    createEnemy();
    if (enemies.hitTest(player.position))
    {
        game = player.miss();
        score.drawNowScore = game;
    }
    counter++;
}
else
{
    updateTitle(keyState);
}
base.Update(gameTime);
}

/// <summary>
/// 敵機を作成します。
/// </summary>
private void createEnemy()
{
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0 &&
        enemies.create(player.position, counter * 0.001f) &&
        score.add(10))
    {
        player.extend();
    }
}

/// <summary>
/// タイトル画面を更新します。
/// </summary>
/// <param name="keyState">現在のキー入力状態。</param>
private void updateTitle(KeyboardState keyState)
{
}

```

```

if (keyState.IsKeyDown(Keys.Escape))
{
    Exit();
}

if (keyState.IsKeyDown(Keys.Space))
{
    // ゲーム開始
    game = true;
    counter = 0;
    for (int i = 0; i < taskGame.Length; i++)
    {
        taskGame[i].reset();
    }
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    graphics.spriteBatch.Begin();
    ITask[] tasks = game ? taskGame : taskTitle;
    for (int i = 0; i < tasks.Length; i++)
    {
        tasks[i].draw(graphics);
    }
    if (!game)
    {
        drawTitle();
    }
    graphics.spriteBatch.End();
    base.Draw(gameTime);
}

```

```
/// <summary>
/// タイトル画面を描画します。
/// </summary>
private void drawTitle()
{
    graphics.spriteBatch.DrawString(
        graphics.spriteFont, "SAMPLE 1", new Vector2(200, 100),
        Color.Black, 0f, Vector2.Zero, 5f, SpriteEffects.None, 0f);
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "PUSH SPACE KEY.", new Vector2(340, 400), Color.Black);
}
```

```
// KeyState.cs

/// <summary>
/// キー入力管理クラス。
/// </summary>
class KeyState
: ITask
{

    /// <summary>キーボードの入力状態。</summary>
    public KeyboardState keyboardState
    {
        get;
        private set;
    }

    /// <summary>
    /// タスクを開始します。
    /// </summary>
    public void reset()
    {
        // 特にすることはない。
    }
}
```

```
/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update(KeyboardState keyState)
{
    keyboardState = Keyboard.GetState();
}

/// <summary>
/// 1 フレーム分の描画を行います。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    // このクラスでは別段何かを描画する必要はない。
}
}
```

```
// Enemy.cs

/// <summary>
/// 敵機の情報。
/// </summary>
struct Enemy
: ITask
{

    /// <summary>大きさ。</summary>
    public const float SIZE = 32;

    /// <summary>ホーミング確率。</summary>
    private const int HOMING_PERCENTAGE = 20;

    /// <summary>ホーミング時間。</summary>
    private const int HOMING_LIMIT = 60;
```

```
/// <summary>初期位置。</summary>
private static readonly Vector2 firstPosition = new Vector2(-SIZE);

/// <summary>移動速度と方角。</summary>
private Vector2 velocity;

/// <summary>ホーミング対応かどうか。</summary>
private bool homing;

/// <summary>ホーミング有効時間。</summary>
private int homingAmount;

/// <summary>現在座標。</summary>
public Vector2 position
{
    get;
    private set;
}

/// <summary>色。</summary>
public Color color
{
    get;
    private set;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
/// <param name="keyState">現在のキー入力状態。</param>
public void update(KeyboardState keyState)
{
    position += velocity;
}
```

```

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool hitTest(Vector2 playerPosition)
{
    const float HITAREA = Player.SIZE * 0.5f + SIZE * 0.5f;
    const float HITAREA_SQUARED = HITAREA * HITAREA;
    bool hit = (HITAREA_SQUARED > Vector2.DistanceSquared(position, playerPosition));
    if (homing && --homingAmount > 0)
    {
        initVelocity(playerPosition, velocity.Length());
    }
    return hit;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    const float SCALE = SIZE / Graphics.RECT;
    Vector2 origin = new Vector2(Graphics.RECT * 0.5f);
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position, null, color,
        0f, new Vector2(Graphics.RECT * 0.5f), SCALE, SpriteEffects.None, 0f);
}

/// <summary>
/// 敵機をアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
/// <returns>敵機をアクティブにできた場合、true。</returns>
public bool start(Vector2 playerPosition, float speed)

```

```

{
    bool result = !Game1.SCREEN.Contains((int)position.X, (int)position.Y);
    if (result)
    {
        startForce(playerPosition, speed);
    }
    return result;
}

/// <summary>
/// 敵機を強制的にスリープにします。
/// </summary>
public void reset()
{
    position = firstPosition;
    velocity = Vector2.Zero;
}

/// <summary>
/// 敵機を強制的にアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
private void startForce(Vector2 playerPosition, float speed)
{
    Random rnd = new Random();
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
        pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
    }
}

```

```

    }

    else
    {
        pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
        pos.Y = p % Game1.SCREEN.Height;
    }

    position = pos;
    initVelocity(playerPosition, rnd.Next(1, 3) + speed);
    homing = rnd.Next(100) < Enemy.HOMING_PERCENTAGE;
    color = homing ? Color.Orange : Color.Red;
    homingAmount = Enemy.HOMING_LIMIT;
}

/// <summary>
/// 敵機の移動速度と方角を初期化します。
/// </summary>
/// <param name="playerPosition">プレイヤーの位置。</param>
/// <param name="speed">速度。</param>
private void initVelocity(Vector2 playerPosition, float speed)
{
    Vector2 v = playerPosition - position;
    if (v == Vector2.Zero)
    {
        // 長さが0だと単位ベクトル計算時にNaNが出るため対策
        v = Vector2.UnitX;
    }
    v.Normalize();
    velocity = v * speed;
}
}

```

```
// Enemies.cs
```

```

/// <summary>
/// 敵機の情報。
/// </summary>
```

```

class Enemies
    : ITask
{
    /// <summary>最大数。</summary>
    private const int MAX = 100;

    /// <summary>敵機一覧データ。</summary>
    private Enemy[] list = new Enemy[MAX];

    /// <summary>
    /// 敵機を作成します。
    /// </summary>
    /// <param name="playerPosition">自機の座標。</param>
    /// <param name="speed">基準速度。</param>
    /// <returns>敵機を作成できた場合、true。</returns>
    public bool create(Vector2 playerPosition, float speed)
    {
        bool result = false;
        for (int i = 0; !result && i < MAX; i++)
        {
            result = list[i].start(playerPosition, speed);
        }
        return result;
    }

    /// <summary>
    /// 1フレーム分の更新を行います。
    /// </summary>
    /// <param name="keyState">現在のキー入力状態。</param>
    public void update(KeyboardState keyState)
    {
        for (int i = 0; i < MAX; i++)
        {
            list[i].update(keyState);
        }
    }
}

```

```
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool hitTest(Vector2 playerPosition)
{
    bool hit = false;
    for (int i = 0; !hit && i < MAX; i++)
    {
        hit = list[i].hitTest(playerPosition);
    }
    if (hit)
    {
        reset();
    }
    return hit;
}

/// <summary>
/// 敵機を初期状態にリセットします。
/// </summary>
public void reset()
{
    Vector2 firstPosition = new Vector2(-Enemy.SIZE);
    for (int i = 0; i < MAX; i++)
    {
        list[i].reset();
    }
}
```

```
/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    for (int i = 0; i < MAX; i++)
    {
        list[i].draw(graphics);
    }
}
```

```
// Player.cs

/// <summary>
/// 自機の情報。
/// </summary>
class Player
: ITask
{

    /// <summary>大きさ。</summary>
    public const float SIZE = 64;

    /// <summary>移動速度。</summary>
    private const float SPEED = 3;

    /// <summary>自機の初期残機。</summary>
    private const int DEFAULT_AMOUNT = 2;

    /// <summary>入力を受け付けるキー一覧。</summary>
    private readonly Keys[] acceptInputKeyList;

    /// <summary>キー入力に対応した移動方向。</summary>
    private readonly Dictionary<Keys, Vector2> velocity;
```

```
/// <summary>ミス猶予(残機)数。</summary>
private int m_amount;

/// <summary>現在座標。</summary>
public Vector2 position
{
    get;
    private set;
}

/// <summary>ミス猶予(残機)数。</summary>
public int amount
{
    get
    {
        return m_amount;
    }
    private set
    {
        m_amount = value;
        amountString = value < 0 ? string.Empty : new string('*', value);
    }
}

/// <summary>ミス猶予(残機)数の文字列による表現。</summary>
public string amountString
{
    get;
    private set;
}

/// <summary>
/// 各種値を初期化します。
/// </summary>
public Player()
{
```

```

acceptInputKeyList =
    new Keys[] { Keys.Up, Keys.Down, Keys.Left, Keys.Right };
velocity = new Dictionary<Keys, Vector2>();
velocity.Add(Keys.Up, new Vector2(0, -Player.SPEED));
velocity.Add(Keys.Down, new Vector2(0, Player.SPEED));
velocity.Add(Keys.Left, new Vector2(-Player.SPEED, 0));
velocity.Add(Keys.Right, new Vector2(Player.SPEED, 0));
}

/// <summary>
/// 残機を増やします。
/// </summary>
public void extend()
{
    amount++;
}

/// <summary>
/// 残機を減らします。
/// </summary>
/// <returns>ゲームが続行可能である場合、true。</returns>
public bool miss()
{
    // ミスするとプレイヤーは元の座標へと戻る
    resetPosition();
    return --amount >= 0;
}

/// <summary>
/// 現在位置を初期化します。
/// </summary>
private void resetPosition()
{
    Point center = Game1.SCREEN.Center;
    position = new Vector2(center.X, center.Y);
}

```

```

/// <summary>
/// 座標や残機情報を初期化します。
/// </summary>
public void reset()
{
    resetPosition();
    amount = DEFAULT_AMOUNT;
}

/// <summary>
/// キー入力に応じて移動します。
/// </summary>
/// <param name="keyState">現在のキー入力状態。</param>
public void update(KeyboardState keyState)
{
    Vector2 prev = position;
    for (int i = 0; i < acceptInputKeyList.Length; i++)
    {
        Keys key = acceptInputKeyList[i];
        if (keyState.IsKeyDown(key))
        {
            position += velocity[key];
        }
    }
    if (!Game1.SCREEN.Contains((int)position.X, (int)position.Y))
    {
        position = prev;
    }
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{

```

```
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "PLAYER: " + amountString, new Vector2(600, 560), Color.Black);
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position,
        null, Color.White, 0f, new Vector2(Graphics.RECT * 0.5f),
        Player.SIZE / Graphics.RECT, SpriteEffects.None, 0f);
}
}
```

```
// Score.cs

/// <summary>
/// スコア情報。
/// </summary>
class Score
: ITask
{

    /// <summary>エクステンドの閾値。</summary>
private const int EXTEND_THRESHOLD = 500;

    /// <summary>現在のスコアを描画するかどうか。</summary>
public bool drawNowScore = false;

    /// <summary>前フレームのスコア。</summary>
private int prev;

    /// <summary>現在のスコア。</summary>
public int now
{
    get;
    private set;
}

    /// <summary>ハイスコア。</summary>
public int highest
{
```

```

    get;
    private set;
}

/// <summary>
/// スコアをリセットします。
/// </summary>
public void reset()
{
    now = 0;
    prev = 0;
    drawNowScore = true;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
/// <param name="keyState">現在のキー入力状態。</param>
public void update(KeyboardState keyState)
{
    // スコアクラスは別段毎フレーム更新するようなものはない。
}

/// <summary>
/// スコアを加算します。
/// </summary>
/// <param name="score">加算されるスコア値。</param>
/// <returns>エクステンド該当となる場合、true。</returns>
public bool add(int score)
{
    now += score;
    bool extend = now % EXTEND_THRESHOLD < prev % EXTEND_THRESHOLD;
    prev = now;
    if (highest < now)
    {
        highest = now;
    }
}

```

```
        }

        return extend;
    }

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    if (drawNowScore)
    {
        graphics.spriteBatch.DrawString(graphics.spriteFont,
            "SCORE: " + now.ToString(),
            new Vector2(300, 560), Color.Black);
    }

    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "HISCORE: " + highest.ToString(), new Vector2(0, 560), Color.Black);
}
}
```

## シーン管理とタスク管理

前節でポリモーフィズムの実践例を提示しました。しかし、実はまだほかにも改良できる部分があります。たとえばこのゲームにはいくつのシーンがありますか？シーンは画面とも読み替えてかまいません。

- タイトルシーン
- ゲームフィールドシーン

以上の二つですね。現在シーンはこの二つしかないため、if 文を使用してゲーム フィールドの状態か否か(前節に挙げたコードで言うところの bool game 変数ですね)で分岐しています。しかし、ここでもしランキングシーンとクレジット表示シーン、あとゲームオーバー演出シーンを加えたと仮定した場合、どのように実装しましょうか？まず真っ先に思いつく方法として、switch ステートメントでシーン番号によって分岐する手段がありますね。例を示してみます。

これはシーンが増えると一つのメソッドが長くなりそうで嫌ですね。ほかの方法がないかどうか、考えてみましょう。ここでもポリモーフィズムが活用できそうです。さっそく書き換えてみましょう。

```
// switch を使用した分岐の一例

public int scene;
public const int SCENE_TITLE = 0;
public const int SCENE_GAME = 1;
public const int SCENE_CREDIT = 2;

// .....(中略)......

switch (scene)
{
    case SCENE_TITLE:
        updateTitle();
        break;
    case SCENE_GAME:
        updateGame();
        break;
    case SCENE_CREDIT:
        updateCredit();
        break;
    // .....(中略)......

}
```

```
// ITask.cs

/// <summary>
/// タスク インターフェイス。
/// </summary>
interface ITask
{
    /// <summary>
    /// タスクを開始します。
    /// </summary>
    void setup();

    /// <summary>
    /// 1 フレーム分の更新を行います。
    /// </summary>
    void update();

    /// <summary>
    /// 1 フレーム分の描画を行います。
    /// </summary>
    /// <param name="graphics">グラフィック データ。</param>
    void draw(Graphics graphics);
}
```

```
// IScene.cs

/// <summary>
/// シーン インターフェイス。
/// </summary>
```

```
interface IScene
  : ITask
{
  /// <summary>次に遷移するシーン。</summary>
  IScene next
  {
    get;
  }
}
```

```
// Title.cs

/// <summary>
/// タイトル画面。
/// </summary>
class Title
  : IScene
{
  /// <summary>クラス オブジェクト。</summary>
  public static readonly IScene instance = new Title();

  /// <summary>
  /// コンストラクタ。
  /// </summary>
  private Title()
  {
    next = this;
  }

  /// <summary>次に遷移するシーン。</summary>
  public IScene next
  {
    get;
    private set;
  }
```

```
}

/// <summary>
/// ゲーム シーンの初期化を行います。
/// </summary>
public void setup()
{
    Score.instance.drawNowScore = false;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update()
{
    next = this;
    KeyboardState keyState = KeyState.instance.keyboardState;
    if (keyState.IsKeyDown(Keys.Escape))
    {
        next = null;
    }
    if (keyState.IsKeyDown(Keys.Space))
    {
        // ゲーム開始
        next = GamePlay.instance;
    }
}

/// <summary>
/// 1 フレーム分の描画を行います。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
```

```
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "PUSH SPACE KEY.", new Vector2(340, 400), Color.Black);
    Score.instance.draw(graphics);
}
}
```

```
// GamePlay.cs

/// <summary>
/// ゲームプレイ画面。
/// </summary>
class GamePlay
: IScene
{

    /// <summary>クラス オブジェクト。</summary>
    public static readonly IScene instance = new GamePlay();

    /// <summary>敵機一覧データ。</summary>
    private readonly Enemies enemies = new Enemies();

    /// <summary>自機データ。</summary>
    private readonly Player player = new Player();

    /// <summary>タスク管理クラス。</summary>
    private readonly TaskManager mgrTask;

    /// <summary>ゲームの進行カウンタ。</summary>
    private int counter;

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    private GamePlay()
    {
        next = this;
    }
}
```

```
    mgrTask = new TaskManager(new ITask[] { enemies, player, Score.instance });
}

/// <summary>次に遷移するシーン。</summary>
public IScene next
{
    get;
    private set;
}

/// <summary>
/// ゲーム シーンの初期化を行います。
/// </summary>
public void setup()
{
    mgrTask.setup();
    counter = 0;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update()
{
    next = this;
    mgrTask.update();
    createEnemy();
    next = (enemies.hitTest(player.position) && !player.miss()) ?
        Title.instance : this;
    counter++;
}

/// <summary>
/// 敵機を作成します。
/// </summary>
private void createEnemy()
```

```
{  
    if (counter % (int)MathHelper.Max(60 - counter * 0.01f, 1) == 0 &&  
        enemies.create(player.position, counter * 0.001f) &&  
        Score.instance.add(10))  
    {  
        player.extend();  
    }  
}  
  
/// <summary>  
/// 1 フレーム分の描画を行います。  
/// </summary>  
/// <param name="graphics">グラフィック データ。</param>  
public void draw(Graphics graphics)  
{  
    mgrTask.draw(graphics);  
}  
}
```

```
// SceneManager.cs  
  
/// <summary>  
/// シーン管理クラス。  
/// </summary>  
class SceneManager  
    : ITask  
{  
  
    /// <summary>現在のシーン。</summary>  
    public IScene nowScene  
    {  
        get;  
        private set;  
    }
```

```
/// <summary>
/// コンストラクタ。
/// </summary>
/// <param name="first">最初のシーン。</param>
public SceneManager(IScene first)
{
    changeScene(first);
}

/// <summary>
/// タスクを開始します。
/// </summary>
public void setup()
{
    // 特にすることはない。
}

/// <summary>
/// 1フレーム分の更新を行います。
/// </summary>
public void update()
{
    nowScene.update();
    changeScene(nowScene.next());
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    nowScene.draw(graphics);
}
```

```
/// <summary>
/// シーンを切り替えます。
/// </summary>
/// <param name="next">次のシーン。</param>
private void changeScene(IScene next)
{
    if (nowScene != next)
    {
        nowScene = next;
        if (next != null)
        {
            next.setup();
        }
    }
}
```

```
// TaskManager.cs

/// <summary>
/// タスク管理クラス。
/// </summary>
class TaskManager
    : ITask
{
    /// <summary>タスク一覧。</summary>
    private readonly ITask[] tasks;

    /// <summary>タスク数。</summary>
    private readonly int length;

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    /// <param name="tasks">タスク一覧。</param>
```

```
public TaskManager(ITask[] tasks)
{
    this.tasks = tasks;
    length = tasks.Length;
}

/// <summary>
/// タスクを開始します。
/// </summary>
public void setup()
{
    for (int i = 0; i < length; i++)
    {
        tasks[i].setup();
    }
}

/// <summary>
/// 1フレーム分の更新を行います。
/// </summary>
public void update()
{
    for (int i = 0; i < length; i++)
    {
        tasks[i].update();
    }
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    for (int i = 0; i < length; i++)
    {
```

```
    tasks[i].draw(graphics);
}
}
}
```

```
// Game1.cs

/// <summary>
/// This is the main type for your game
/// </summary>
public class Game1
: Game
{

/// <summary>画面矩形情報。</summary>
public static readonly Rectangle SCREEN = new Rectangle(0, 0, 800, 600);

/// <summary>描画周りデータ。</summary>
private Graphics graphics;

/// <summary>シーン管理クラス。</summary>
private SceneManager mgrScene = new SceneManager(Title.instance);

/// <summary>タスク管理クラス。</summary>
private readonly TaskManager mgrTask;

/// <summary>
/// Constructor.
/// </summary>
public Game1()
{
    new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    mgrTask = new TaskManager(new ITask[] { KeyState.instance, mgrScene });
}
```

```

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    graphics = new Graphics(this);
    mgrTask.setup();
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    mgrTask.update();
    if (mgrScene.nowScene == null)
    {
        Exit();
    }
    base.Update(gameTime);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    graphics.spriteBatch.Begin();
    mgrTask.draw(graphics);
    graphics.spriteBatch.End();
}

```

```
}
```

```
// KeyState.cs

/// <summary>
/// キー入力管理クラス。
/// </summary>
class KeyState
    : ITask
{

    /// <summary>クラス オブジェクト。</summary>
    public static readonly KeyState instance = new KeyState();

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    private KeyState()
    {
    }

    /// <summary>キーボードの入力状態。</summary>
    public KeyboardState keyboardState
    {
        get;
        private set;
    }

    /// <summary>
    /// タスクを開始します。
    /// </summary>
    public void setup()
    {
        // 特にすることはない。
    }
}
```

```
/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update()
{
    keyboardState = Keyboard.GetState();
}

/// <summary>
/// 1 フレーム分の描画を行います。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    // このクラスでは別段何かを描画する必要はない。
}
}
```

```
// Score.cs

/// <summary>
/// スコア情報。
/// </summary>
class Score
: ITask
{

    /// <summary>クラス オブジェクト。</summary>
    public static readonly Score instance = new Score();

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    private Score()
    {
    }
}
```

```
/// <summary>エクステンドの閾値。</summary>
private const int EXTEND_THRESHOLD = 500;

/// <summary>現在のスコアを描画するかどうか。</summary>
public bool drawNowScore = false;

/// <summary>前フレームのスコア。</summary>
private int prev;

/// <summary>現在のスコア。</summary>
public int now
{
    get;
    private set;
}

/// <summary>ハイスコア。</summary>
public int highest
{
    get;
    private set;
}

/// <summary>
/// スコアをリセットします。
/// </summary>
public void setup()
{
    now = 0;
    prev = 0;
    drawNowScore = true;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
```

```

public void update()
{
    // スコアクラスは別段毎フレーム更新するようなものはない。
}

/// <summary>
/// スコアを加算します。
/// </summary>
/// <param name="score">加算されるスコア値。</param>
/// <returns>エクステンド該当となる場合、true。</returns>
public bool add(int score)
{
    now += score;
    bool extend = now % EXTEND_THRESHOLD < prev % EXTEND_THRESHOLD;
    prev = now;
    if (highest < now)
    {
        highest = now;
    }
    return extend;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    if (drawNowScore)
    {
        graphics.spriteBatch.DrawString(graphics.spriteFont,
            "SCORE: " + now.ToString(),
            new Vector2(300, 560), Color.Black);
    }
    graphics.spriteBatch.DrawString(graphics.spriteFont,
        "HISCORE: " + highest.ToString(), new Vector2(0, 560), Color.Black);
}

```

```
}
```

```
// Player.cs

/// <summary>
/// 自機の情報。
/// </summary>
class Player
: ITask
{

    /// <summary>大きさ。</summary>
    public const float SIZE = 64;

    /// <summary>移動速度。</summary>
    private const float SPEED = 3;

    /// <summary>自機の初期残機。</summary>
    private const int DEFAULT_AMOUNT = 2;

    /// <summary>入力を受け付けるキー一覧。</summary>
    private readonly Keys[] acceptInputKeyList;

    /// <summary>キー入力に対応した移動方向。</summary>
    private readonly Dictionary<Keys, Vector2> velocity;

    /// <summary>ミス猶予(残機)数。</summary>
    private int m_amount;

    /// <summary>現在座標。</summary>
    public Vector2 position
{
    get;
```

```

    private set;
}

/// <summary>ミス猶予(残機)数。</summary>
public int amount
{
    get
    {
        return m_amount;
    }
    private set
    {
        m_amount = value;
        amountString = value < 0 ? string.Empty : new string('*', value);
    }
}

/// <summary>ミス猶予(残機)数の文字列による表現。</summary>
public string amountString
{
    get;
    private set;
}

/// <summary>
/// 各種値を初期化します。
/// </summary>
public Player()
{
    acceptInputKeyList =
        new Keys[] { Keys.Up, Keys.Down, Keys.Left, Keys.Right };
    velocity = new Dictionary<Keys, Vector2>();
    velocity.Add(Keys.Up, new Vector2(0, -Player.SPEED));
    velocity.Add(Keys.Down, new Vector2(0, Player.SPEED));
    velocity.Add(Keys.Left, new Vector2(-Player.SPEED, 0));
    velocity.Add(Keys.Right, new Vector2(Player.SPEED, 0));
}

```

```
}

/// <summary>
/// 残機を増やします。
/// </summary>
public void extend()
{
    amount++;
}

/// <summary>
/// 残機を減らします。
/// </summary>
/// <returns>ゲームが続行可能である場合、true。</returns>
public bool miss()
{
    // ミスするとプレイヤーは元の座標へと戻る
    resetPosition();
    return --amount >= 0;
}

/// <summary>
/// 現在位置を初期化します。
/// </summary>
private void resetPosition()
{
    Point center = Game1.SCREEN.Center;
    position = new Vector2(center.X, center.Y);
}

/// <summary>
/// 座標や残機情報を初期化します。
/// </summary>
public void setup()
{
    resetPosition();
```

```

        amount = DEFAULT_AMOUNT;
    }

    /// <summary>
    /// キー入力に応じて移動します。
    /// </summary>
    public void update()
    {
        KeyboardState keyState = KeyState.instance.keyboardState;
        Vector2 prev = position;
        for (int i = 0; i < acceptInputKeyList.Length; i++)
        {
            Keys key = acceptInputKeyList[i];
            if (keyState.IsKeyDown(key))
            {
                position += velocity[key];
            }
        }
        if (!Game1.SCREEN.Contains((int)position.X, (int)position.Y))
        {
            position = prev;
        }
    }

    /// <summary>
    /// 描画します。
    /// </summary>
    /// <param name="graphics">グラフィック データ。</param>
    public void draw(Graphics graphics)
    {
        graphics.spriteBatch.DrawString(graphics.spriteFont,
            "PLAYER: " + amountString, new Vector2(600, 560), Color.Black);
        graphics.spriteBatch.Draw(graphics.gameThumbnail, position,
            null, Color.White, 0f, new Vector2(Graphics.RECT * 0.5f),
            Player.SIZE / Graphics.RECT, SpriteEffects.None, 0f);
    }
}

```

```
}
```

```
// Enemies.cs

/// <summary>
/// 敵機の情報。
/// </summary>
class Enemies
: ITask
{

/// <summary>最大数。</summary>
private const int MAX = 100;

/// <summary>敵機一覧データ。</summary>
private Enemy[] list = new Enemy[MAX];

/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
/// <returns>敵機を作成できた場合、true。</returns>
public bool create(Vector2 playerPosition, float speed)
{
    bool result = false;
    for (int i = 0; !result && i < MAX; i++)
    {
        result = list[i].start(playerPosition, speed);
    }
    return result;
}

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
```

```
public void update()
{
    for (int i = 0; i < MAX; i++)
    {
        list[i].update();
    }
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool hitTest(Vector2 playerPosition)
{
    bool hit = false;
    for (int i = 0; !hit && i < MAX; i++)
    {
        hit = list[i].hitTest(playerPosition);
    }
    if (hit)
    {
        setup();
    }
    return hit;
}

/// <summary>
/// 敵機を初期状態にリセットします。
/// </summary>
public void setup()
{
    for (int i = 0; i < MAX; i++)
    {
        list[i].setup();
    }
}
```

```
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    for (int i = 0; i < MAX; i++)
    {
        list[i].draw(graphics);
    }
}
```

```
// Enemy.cs

/// <summary>
/// 敵機の情報。
/// </summary>
struct Enemy
: ITask
{

    /// <summary>大きさ。</summary>
private const float SIZE = 32;

    /// <summary>ホーミング確率。</summary>
private const int HOMING_PERCENTAGE = 20;

    /// <summary>ホーミング時間。</summary>
private const int HOMING_LIMIT = 60;

    /// <summary>初期位置。</summary>
private static readonly Vector2 firstPosition = new Vector2(-SIZE);
```

```
/// <summary>移動速度と方角。</summary>
private Vector2 velocity;

/// <summary>ホーミング対応かどうか。</summary>
private bool homing;

/// <summary>ホーミング有効時間。</summary>
private int homingAmount;

/// <summary>現在座標。</summary>
private Vector2 position;

/// <summary>色。</summary>
private Color color;

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update()
{
    position += velocity;
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <returns>接触した場合、true。</returns>
public bool hitTest(Vector2 playerPosition)
{
    const float HITAREA = Player.SIZE * 0.5f + SIZE * 0.5f;
    const float HITAREA_SQUARED = HITAREA * HITAREA;
    bool hit = (HITAREA_SQUARED > Vector2.DistanceSquared(position, playerPosition));
    if (homing && --homingAmount > 0)
    {
        initVelocity(playerPosition, velocity.Length());
    }
}
```

```

    }

    return hit;
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    const float SCALE = SIZE / Graphics.RECT;
    Vector2 origin = new Vector2(Graphics.RECT * 0.5f);
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position, null, color,
        Of, new Vector2(Graphics.RECT * 0.5f), SCALE, SpriteEffects.None, 0f);
}

/// <summary>
/// 敵機をアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
/// <returns>敵機をアクティブにできた場合、true。</returns>
public bool start(Vector2 playerPosition, float speed)
{
    bool result = !Game1.SCREEN.Contains((int)position.X, (int)position.Y);
    if (result)
    {
        startForce(playerPosition, speed);
    }
    return result;
}

/// <summary>
/// 敵機を強制的にスリープにします。
/// </summary>
public void setup()

```

```

{
    position = firstPosition;
    velocity = Vector2.Zero;
}

/// <summary>
/// 敵機を強制的にアクティブにします。
/// </summary>
/// <param name="playerPosition">自機の座標。</param>
/// <param name="speed">基準速度。</param>
private void startForce(Vector2 playerPosition, float speed)
{
    Random rnd = new Random();
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
        pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
    }
    else
    {
        pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
        pos.Y = p % Game1.SCREEN.Height;
    }
    position = pos;
    initVelocity(playerPosition, rnd.Next(1, 3) + speed);
    homing = rnd.Next(100) < Enemy.HOMING_PERCENTAGE;
    color = homing ? Color.Orange : Color.Red;
    homingAmount = Enemy.HOMING_LIMIT;
}

```

```
/// <summary>
/// 敵機の移動速度と方角を初期化します。
/// </summary>
/// <param name="playerPosition">プレイヤーの位置。</param>
/// <param name="speed">速度。</param>
private void initVelocity(Vector2 playerPosition, float speed)
{
    Vector2 v = playerPosition - position;
    if (v == Vector2.Zero)
    {
        // 長さが0だと単位ベクトル計算時にNaNが出るため対策
        v = Vector2.UnitX;
    }
    v.Normalize();
    velocity = v * speed;
}
```

## 名前空間を使って整理する

`switch` によるコード分岐を使用すると一つのメソッドが長くなる欠点があるように、ポリモーフィズムを使った場合にも、必然的にクラス数が増えていくという別の欠点があります。C#では原則 1 クラス 1 ファイル<sup>23</sup>なので、クラス名が増えるということは、ファイル数も増えていくことになります。ファイルごとにそれぞれ解りやすい名前がついているためまだましに思えるかもしれません、それでも増えていくと大変見づらくなると思います。また、ポリモーフィズムを使うとコードのコピー＆ペースト(以下コピペ)が増えてしまう懸念があります。コピペが増えてくると、前章で紹介したマジックナンバーと同じこと、すなわち誤修正などのミスが起きる大きな要因となってしまいます。

まずは増えすぎたクラス、即ち増えすぎたファイルを整理しましょう。これには、名前空間を使用するのが得策です。C#では原則 1 名前空間 1 フォルダ<sup>24</sup>なので、見た目はかなりすっきりしますね。今回は一階層のみですが、名前空間は二階層、三階層ともっと深く刻むことができます。(例えば `Microsoft.Xna.Framework.Graphics` 名前空間は、`Microsoft/Xna/Framework/Graphics/` なので四階層ですね)



次にコピペ問題も解決しましょう。例えば前節のサンプルでは `TaskManager` クラスと `Enemies` クラスとの内容が半分被っていました。そこで今回被っている部分を `TaskManager` に統合し、`Enemies` 固有の部分は `TaskManager` を継承する形で実装して、名前も `EnemyManager` として改めてみました。

<sup>23</sup> Java などと違い強制ではありません。私でも関連するクラスはまとめてしまうことがあります。

<sup>24</sup> こちらも Java などと違い強制ではありません。

```
// TaskManager.cs

/// <summary>
/// タスク管理クラス。
/// </summary>
class TaskManager<T>
    : ITask
    where T : ITask
{
    /// <summary>タスク一覧。</summary>
    public readonly List<T> tasks = new List<T>();

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    public TaskManager()
    {
    }

    /// <summary>
    /// タスクを開始します。
    /// </summary>
    public void setup()
    {
        int length = tasks.Count;
        for (int i = 0; i < length; i++)
        {
            tasks[i].setup();
        }
    }
}
```

```
/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public void update()
{
    int length = tasks.Count;
    for (int i = 0; i < length; i++)
    {
        tasks[i].update();
    }
}

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    int length = tasks.Count;
    for (int i = 0; i < length; i++)
    {
        tasks[i].draw(graphics);
    }
}
```

```
// ITask.cs

/// <summary>
/// タスク インターフェイス。
/// </summary>
interface ITask
{
    /// <summary>
    /// タスクを開始します。
    /// </summary>
```

```
void setup();

/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
void update();

/// <summary>
/// 1 フレーム分の描画を行います。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
void draw(Graphics graphics);
}
```

```
// EnemyManager.cs

/// <summary>
/// 敵機の情報。
/// </summary>
class EnemyManager
: TaskManager<Enemy>
{

    /// <summary>ホーミング敵機の確率。</summary>
    private const int HOMING_PERCENTAGE = 20;

    /// <summary>粗悪精度敵機の確率。</summary>
    private const int INFERIORITY_PERCENTAGE = 30;

    /// <summary>疑似乱数ジェネレータ。</summary>
    private readonly Random rnd = new Random();

    /// <summary>
    /// 敵機を作成します。
    /// </summary>
    /// <param name="speed">基準速度。</param>
```

```

public void create(float speed)
{
    Enemy result = null;
    int percentage = rnd.Next(100);
    if (percentage - HOMING_PERCENTAGE < 0)
    {
        result = new EnemyHoming(speed);
    }
    else if (percentage - INFERIORITY_PERCENTAGE < 0)
    {
        result = new EnemyInferiority(speed);
    }
    else
    {
        result = new EnemyStraight(speed);
    }
    if (result != null)
    {
        tasks.Add(result);
    }
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <returns>接触した場合、true。</returns>
public bool hitTest()
{
    bool hit = false;
    int length = tasks.Count;
    for (int i = 0; !hit && i < length; i++)
    {
        hit = tasks[i].hitTest();
    }
    if (hit)
    {

```

```
        tasks.Clear();
    }

    return hit;
}

}
```

もしかしたら初めて見る記述かもしれません、今回から TaskManager クラスも若干改造を加え、総称型(ジェネリックスとも言います)<sup>25</sup>を使用しています。そして中を見ると、T と言う型をクラス内のあるところで使用しているように見受けられますが、これが総称型です。これは継承先や使用先でクラス名が変化する特殊な型で、たとえば「new TaskManager<Player>();」と記述して作られたオブジェクトでは T 型はすべて Player 型に置き換えられますし、EnemyManager は TaskManager<Enemy>を継承しているので、スーパークラスの T 型はすべて Enemy 型に置き換えられます。EnemyManager 内にてキャストを使用せずとも hitTest() が使用できるのはそのためです。

少々話がそれましたので戻しましょう。EnemyManager クラスは TaskManager クラスの機能を積んだ上で、敵管理に特化した独自機能を積んでいます。このように継承を使って機能を分割することにより、大半の処理がの処理が被っている場合にロジックのコピペをしなくて済むようになります。

継承を使った機能分割をもう少し練習してみましょう。そこで今回のサンプルからは敵機の種類を増やし、従来の自機狙い直線とホーミングに加え、ちょっと精度の悪い自機狙いの直線タイプ(赤紫色)を追加し、赤の確率を 50% に減らして残った 30% の確率で発射されるようにしました。(Sample1\_11)

```
// Enemy.cs

/// <summary>
/// 敵機の情報。
/// </summary>
abstract class Enemy
: ITask
{
```

<sup>25</sup> 参考までに、C++ではテンプレートと呼びます。機能はほぼ一緒です。  
(内部的な話になってくるとかなり違ってくるのですが、本書では割愛します)

```

/// <summary>大きさ。</summary>
private const float SIZE = 32;

/// <summary>疑似乱数ジェネレータ。</summary>
protected static readonly Random rnd = new Random();

/// <summary>移動速度と方角。</summary>
protected Vector2 velocity;

/// <summary>現在座標。</summary>
private Vector2 position;

/// <summary>色。</summary>
private Color color;

/// <summary>
/// コンストラクタ。
/// </summary>
/// <param name="speed">基準速度。</param>
/// <param name="color">色。</param>
public Enemy(float speed, Color color)
{
    this.color = color;
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
        pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
    }
    else
    {

```

```

pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
pos.Y = p % Game1.SCREEN.Height;
}

position = pos;
initVelocity(rnd.Next(1, 3) + speed);
}

/// <summary>
/// タスクを開始します。
/// </summary>
public void setup()
{
    // 特にすることはない。
}

/// <summary>
/// 1フレーム分の更新を行います。
/// </summary>
public virtual void update()
{
    position += velocity;
}

/// <summary>
/// 敵機の移動、及び接触判定をします。
/// </summary>
/// <returns>接触した場合、true。</returns>
public bool hitTest()
{
    const float HITAREA = Player.SIZE * 0.5f + SIZE * 0.5f;
    const float HITAREA_SQUARED = HITAREA * HITAREA;
    bool hit = (HITAREA_SQUARED > Vector2.DistanceSquared(position,
Player.instance.position));
    return hit;
}

```

```

/// <summary>
/// 描画します。
/// </summary>
/// <param name="graphics">グラフィック データ。</param>
public void draw(Graphics graphics)
{
    const float SCALE = SIZE / Graphics.RECT;
    Vector2 origin = new Vector2(Graphics.RECT * 0.5f);
    graphics.spriteBatch.Draw(graphics.gameThumbnail, position, null, color,
        0f, new Vector2(Graphics.RECT * 0.5f), SCALE, SpriteEffects.None, 0f);
}

/// <summary>
/// 敵機の移動速度と方角を初期化します。
/// </summary>
/// <param name="speed">速度。</param>
protected virtual void initVelocity(float speed)
{
    Vector2 v = Player.instance.position - position;
    if (v == Vector2.Zero)
    {
        // 長さが 0 だと単位ベクトル計算時に NaN が出るため対策
        v = Vector2.UnitX;
    }
    v.Normalize();
    velocity = v * speed;
}
}

```

```

// EnemyStraight.cs

/// <summary>
/// 自機めがけて直進する敵機の情報。
/// </summary>
class EnemyStraight
: Enemy

```

```
{  
  
/// <summary>  
/// コンストラクタ。  
/// </summary>  
/// <param name="speed">基準速度。</param>  
public EnemyStraight(float speed) :  
    base(speed, Color.Red)  
{  
}  
}
```

```
// EnemyHoming.cs  
  
/// <summary>  
/// 自機めがけてホーミングする敵機の情報。  
/// </summary>  
class EnemyHoming  
: Enemy  
{  
  
/// <summary>ホーミング時間。</summary>  
private const int HOMING_LIMIT = 60;  
  
/// <summary>ホーミング有効時間。</summary>  
private int homingAmount;  
  
/// <summary>  
/// コンストラクタ。  
/// </summary>  
/// <param name="speed">基準速度。</param>  
public EnemyHoming(float speed) :  
    base(speed, Color.Orange)  
{  
    homingAmount = HOMING_LIMIT;  
}
```

```
/// <summary>
/// 1 フレーム分の更新を行います。
/// </summary>
public override void update()
{
    base.update();
    if (--homingAmount > 0)
    {
        initVelocity(velocity.Length());
    }
}
```

```
// EnemyInferiority.cs

/// <summary>
/// 粗悪な精度で自機めがけて直進する敵機の情報。
/// </summary>
class EnemyInferiority
: Enemy
{

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    /// <param name="speed">基準速度。</param>
    public EnemyInferiority(float speed) :
        base(speed, Color.Magenta)
    {

    }

    /// <summary>
    /// 敵機の移動速度と方角を初期化します。
    /// </summary>
    /// <param name="speed">速度。</param>
    protected override void initVelocity(float speed)
```

```

{
    base.initVelocity(speed);

    // ここでベクトルをわざと乱して、精度を落とす
    Quaternion q = Quaternion.CreateFromAxisAngle(
        Vector3.UnitZ, (float)rnd.NextDouble() - 0.5f);
    velocity = Vector2.Transform(velocity, q);
}
}

```

今までのサンプルでは各敵機固有の機能は if 文を使った分岐で解消していましたが、今回は継承を使ったサブクラスへの委譲によって各敵機の固有機能を実装しています。このように、極力分岐を使用せずに委譲を用いることは、今後スマートなプログラムを組んでいくうえで非常に重要な考え方ですので、ぜひとも覚えておいてください。

ところで、大きなゲームを作るにはまず不可欠だろうと言えるタスク管理クラスですが、こんな便利な機能がゲーム開発に特化しているはずの XNA で提供されていないのでしょうか？結論から言うと、存在します。Game クラスの中を(または MSDN の API マニュアルを眺めていると)Component と言うプロパティが出てくることに気付きますが、これがまさしくタスク管理クラスとほぼ同等のものです。ここではタスクのことをゲーム コンポーネントと呼び、DrawableGameComponent クラス<sup>26</sup>、またはそれを継承したクラスのオブジェクトを Game.Component に食わせてやると、そのクラスの Update() と Draw() の二つのメソッドが毎フレーム呼ばれるようになります。実際に使ってみると気付くかと思いますが、構造が Game クラスそっくりなので、あたかも複数の Game クラスが同時に動いているかのように見えるでしょう。

こんな便利な機能があるのに、なぜ使用しないのでしょうか？まず第一の理由として中身の構造を理解してもらうためです。ゲーム制作において、車輪の再発明はあまり良いことではないと前章で説明しましたね。ただし、それにはいくつか例外があり、学習目的であえて車輪の再発明を行うことは決して悪いことではありません。また、既存のものが非常に遅いとか、その他致命的な欠点がある場合も作り直したほうが手っ取り早い場合もあります。今回のゲーム コンポーネントは両方に当てはまります。ゲーム コンポーネントは敵機など、大量のコンポーネントを登録するとパフォーマンスに大きく影響してしまいます。これが第二の理由となるのですが、その詳しい理由などは次章で説明しましょう。

次章へ続く前に、最後にちょっとだけ脱線します。今回練習用に新しいタイプの敵機を追加しました

---

<sup>26</sup> 描画機能を削った GameComponent クラスもあります。

が、むやみな新キャラクタや武器などの追加は、ゲームバランス崩壊の原因<sup>27</sup>になりますので、やりすぎには注意しましょう。

---

<sup>27</sup> 今回のバージョンでは、筆者は 1,730 点取ることができました。

## Singleton パターン

次章に続く前に、もう少しだけ前節のサンプルについて解説します。前節のサンプルからは、Player など一部のクラスにおいて Singleton パターン（シングルトン、と読みます）を使っています。これはその読みから「一匹の豚」などと半分冗談めいた例え方をされますが、豚……ここで言うインスタンスが、実行中終了するまで常に一

```
// Singleton パターンの最も単純な実装例
sealed class Foo
{
    public static readonly Foo instance = new Foo();
    private Foo() {}
    ...
}
```

定数（厳密には起動から終了ではなく、「最初に呼び出されてから」プログラムが終了するまでです）存在していて、絶対に増えたり減ったりしないことを保証するロジックのことを指します。

例えばこの Player クラスには instance と言う静的な変数にこのクラスのオブジェクトが入っています。では、この Player クラスを弄らずにもう一つオブジェクトを作つてみることを想定してみてください。そんなの素直に new Player()すれば良いのでは、と思われるかもしれません、実はこれはうまくいきません。クラスの new 可能な範囲は、コンストラクタのアクセス修飾子によって決定されます。例えばこれが public ならどこからでも new できますが、private だとそのクラスの中でしか new できません。では Player クラスにおけるコンストラクタのアクセス修飾子はどうなっているかといいますと、private となっています。よって Player クラスの外部からは絶対に new できません。他にも継承して使う方法もありますが、これもだめです。継承するためにはコンストラクタが public か、protected でなければいけません。

Singleton パターンを使用すると、例えばこのサンプルの場合はプログラム中のどこからでも Player クラスのオブジェクトを参照できます。オブジェクトを参照できると言うことは、Player 内のパラメータ、すなわち座標や残気数などの情報にアクセスできることになります。これは応用次第ではゲーム全体で共有する情報、例えばハイスクア情報やゲームの起動時間などを記録するのに大変役立ちそうです。でも、ゲーム全体で値を共有する方法は他にももう一つありますよね？ そうです、静的クラスを使用する方法です。実際、見比べてみると確かに両者の使い道や使い方はかなり似通っています。しかし、Singleton パターンには静的クラスにないメリットがいくつもあります。

まず、Singleton パターンではほかのクラスからの継承や、インターフェイスの実装が可能です。使い方こそ静的クラスでも実態はインスタンスなので、通常のクラスと同様に継承が行えます。また、実態はインスタンスなので、特定のインターフェイスを実装したいくつかの Singleton クラスを別のクラスの

変数にアップキャストする形で登録して、その値を入れ替えることで全く違うクラスにアクセスさせたりすることもできます（これは次章で出てくる State パターンを理解するために大変重要です）。

他のメリットとして、静的クラスは変数一つにつき絶対に一つしか値を登録できませんが、Singleton パターンでは、変数一つにつきオブジェクトを用意した数だけの数の値を登録することもできます。Singleton、と言う名前に惑わされそうですが、オブジェクトは必ず一つでなければならぬというルールはありません。

```
// Singleton パターンで 1 つの変数に複数の値を登録する例
sealed class Score
{
    public static readonly Score playerA = new Score();
    public static readonly Score playerB = new Score();
    private Score() {}

    public int hiScore;
    ...
}
```

このように静的クラスに比べていいとこづくめの Singleton パターンですが、逆に短所はあるのでしょうか？実は短所はあるにはあるのですが、それと言って大した短所ではありません。

1. 少しおまじないコードが増える
2. 値を参照するためのコードがやや長くなる
3. オブジェクト 1 個分（複数個の場合、その分）のメモリ占有量が増える

	Static	Singleton
継承/実装	×	○
複数	×	○
参照渡し	×	○
容量	Field の数	Field の数+1

思いつく限り挙げてもこの程度です。もしかしたら 3 のメモリ占有量が増える、と言う短所が気になる方もいるかもしれません、これも結論から言うと大きな問題ではありません。オブジェクト一つで使われるメモリの量は、C# 言語の場合 精々数十バイトです。たとえ百箇所で Singleton パターンを使っても余計に使われるメモリは数キロバイト。確かに昔の開発現場は「一バイトは血の一滴」などとよく言われていましたが、今時の環境でしたら全く気にする必要はないと言って良いでしょう。これを気にするくらいなら、まずは画像サイズ<sup>28</sup>を削ることから検討したほうが有意義でしょう。次章で述べますが、ゲーム プログラムにとっては、メモリ内に常に居座り続けるよりも、頻繁に確保したり解放したりするほうが、パフォーマンスに大きな影響を与えるのです。

<sup>28</sup> 例えば GameThumbnail はゲーム用に圧縮をかけても 4.6 キロバイトになります。これはオブジェクト約百個分にも相当します。

#### ステップ 4:パフォーマンスを改善しよう

前章でゲーム コンポーネントの極端な多用や頻繁なメモリ確保・解放が、パフォーマンスに悪影響を及ぼすと説明しましたね。ここでは、その理由と解決策を説明していきます。

## メモリ管理とガーベージ コレクション

C#では一般的にメモリ確保は new 命令で行い、解放はどこからも参照されなくなったときに、定期的に起動されるガーベージ コレクション(以下 GC)によって解放されます。しかし、動的なメモリ確保、及び解放はとても遅いのです。

遅い理由を説明する前に、まず、プログラム上で扱うメモリの種類について説明します。プログラムから扱うメモリにはスタックとヒープの二種類があり、使われる値の種類によって自動的に使い分けられます<sup>29</sup>。スタックはローカル変数を初期化するときに使われます。例えば int i = 1; とローカル変数 i を作ったとき、i の記憶領域はスタックに確保されます。

一方ヒープは静的クラスの中身や new 命令を使ってクラスのオブジェクトを作るときに使われます<sup>30 31</sup>。例えば右記のようなコードで new Foo(); を実行してクラス Foo のオブジェクトを作るとき、オブジェクトの中身を格納するのに必要な記憶領域がヒープに確保されます。

では、ローカル変数で Foo f = null; としたらどうなのでしょうか？実はこの段階ではヒープは全く確保されません。ローカル変数 f の中身は Foo オブジェクトへのアドレス(住所)が入っていて、そのアドレスはスタックに入っています。ちなみに null は「住所不定」と言う意味だと思えばよいでしょう。そして次に、f = new Foo(); とすると Foo オブジェクトがヒープ内に確保されて、そのアドレスが返されて f に格納されます。このようにアドレスだけ入っていて、実体は別のところにある変数のことを C# では「参照」と呼びます。

```
// 実体と参照の違いの実証コード

class Foo
{
    public int i;
}

struct Bar
{
    public int i;
}

static void Main(string[] args)
{
    int i;           // Stack
    Foo foo = new Foo(); // Heap
    Bar bar = new Bar(); // Stack

    i = 1;
    foo.i = 1;
    bar.i = 1;

    int j = i;      // Deep copy
    Foo qux = foo; // Shallow copy
    Bar baz = bar; // Deep copy

    j = 2;          // i == 1
    qux.i = 2;     // foo.i == 2
    baz.i = 2;     // bar.i == 1
}
```

話を整理すると、スタックはプログラムから直接見ることができる領域で、ヒープはスタックに格納さ

<sup>29</sup> C や C++ 言語の場合、手動で使い分けることもできます。

<sup>30</sup> 例外として、配列変数と文字列、そしてデリゲートの中身もスタックではなくヒープに確保されます。

<sup>31</sup> さらに例外として、構造体(struct)はスタックに確保されます。これは C# 特有の仕様です。

れたアドレスを辿ることによって見ることができる領域である、と言うことですね。しかし、ここまで二種類のメモリを解説しましたが、それがどのように確保・解放が遅いことに関係してくるのでしょうか？結論から言ってしまうと、スタックの確保は早く、一方ヒープの確保は遅いのです。と言うよりも、実はスタックは最初から確保されていて、後からいちいち確保したり解放したりしません。

スタックはローカル変数のために使われる、と言いましたよね？C#では書いたソースコードをコンピュータが把握できる機械語に変換するために「コンパイル<sup>32</sup>」と言う作業を行いますが、その時に一定量（大抵は1メガバイト程度）のサイズをスタックとして確保する命令を勝手に入れてしまい、そしてメソッドごとにローカル変数の数を数えて、確保したスタックからどれだけのメモリを割り当てるかを予め計算してしまいます。ローカル変数の数はソースコードを弄らない限り増えたり減ったりしないので、実行時ではなくコンパイル時に計算しても辻褄が合うのです。

一方、ヒープはそういうわけにはいきません。例えば配列変数の中身なんかはヒープに確保されますが、たとえば配列の要素数が「現在動いている敵の数だけ作られる」とすると、とてもコンパイル中に決定できるものではありませんね。そのため、ヒープは実行時に必要に応じて「nバイト分確保しろ」とOSに伝え、OSが空いているメモリを持ってくる、と言う作業が出てくるのでヒープの確保・解放は遅くなるのです。ついでに言ってしまうと、ヒープからの読み出しもスタックに比べるとほんの少し遅いです。これは常にスタックに置かれているアドレスから参照しているからです。

このように確保に手間のかかるヒープですが、C#においては特に解放が問題になってきます。CやC++では解放も自前で行う必要があるためさほど問題にはなりません<sup>33</sup>が、C#では使われなくなつた……即ち、どこからも参照されなくなってしまった時<sup>34</sup>に、定期的に動くGCによって該当するヒープを一斉に解放していきます。このGCが曲者なのです。たださえ遅いメモリ解放を一気に行うため、瞬間に物凄い負荷となります。今時のPCの性能では余程頻繁にGCが起きるような書き方でもしない限り、パフォーマンスに目に見えるような影響は出ませんが、ちょっと古いPCとか、XNAの場合ソースレベル互換のあるXBOX360やWindowsPhone7で動かすと、チフリーズと言う形ではっきりと目に見えるようになります。

前章で述べたもう一つの重くなる原因、ゲームコンポーネントの大量の登録についても説明します。実はここでも似た現象が発生します。XNAのゲームコンポーネントがGameクラスに登録された瞬間に、GameComponentCollection.ComponentAddedと言うイベントが発生して、そこでGameComponentCollectionEventArgsと言うオブジェクトが作成されます。例えば1万個のゲームコンポーネントを登録すると、このオブジェクトも1万個作られるのです。GCと違い今回は解放ではなく

<sup>32</sup> VisualStudioではコンパイルを含めた一連の作業を「ビルド」と呼ばれています。

<sup>33</sup> その代り、解放し忘れ（メモリリーク）と言う別の問題が出てきます。

<sup>34</sup> 東方Projectを知っている人でしたら、誰からも忘れ去られたものが幻想入りする光景を連想すると解りやすいかもしれません。

確保ですが、確保も大量に行うとやはり GC に相当する負荷がかかります。また性質の悪いことに、この直後に GC が動いてしまう可能性が非常に高くなります。GC は定期的に発生すると述べましたが、発生するタイミングは一定時間ごとではなく、「最後に GC が動いてから一定量以上新しくヒープが確保されたら」発生するのです。今回の例で言うと、大量のオブジェクトが作られることにより、大量のヒープが確保されます。それを GC は感知して前回起動時より一定量以上のヒープ確保があったと判断し、GC が動作します。よって、ゲーム コンポーネントを大量に登録するとメモリ確保の負荷と GC が連動することによる解放の負荷と言うダブル負荷に襲われるのです。

また、ゲーム コンポーネントには、Initialize と言うメソッドがあります。これは一見ゲーム コンポーネントが Game クラスに登録されたときに呼ばれるように見えますが、実際には Game.Initialize メソッドに連動して呼ばれるだけなのです。つまりゲーム コンポーネントの Initialize メソッドは実質起動直後にしか呼び出されず、後から登録されたゲーム コンポーネントは Initialize メソッドは呼ばれない、と言うことになります。

これが前章のサンプルにおける、敵機をゲーム コンポーネント化しなかった理由です。大量に登録すると猛烈な負荷がかかり、後から登録したものは明示的な処理をしないと初期化されないゲーム コンポーネントは、敵機に使用するのは不適であると判断したためです。では、どのようなものがゲーム コンポーネントに適しているかと言うと、最初から最後までずっと動き続けるタスクなんかは適していると思います。例えば、前章のサンプルではタスク管理クラスの仕組みを説明するため敢えてしましたが、キー入力クラスやスコアクラスはゲームの起動時から終了時までずっと居座り続けるため、これをゲーム コンポーネント化するのは悪くない選択でしょう。

いずれにしても、GC と言うのはゲームにとってはメモリリークを解消してくれる便利な機能であると同時に、ゲームのパフォーマンスに悪影響を与えててしまう、言ってしまえば鈍間な黒子（黒衣）のような存在であることが解りました。パフォーマンスの改善には彼になるべく引っ込んでもらうのが得策でしょう。GC を抑制するためには、まず現時点でどれだけ GC が動いているかを把握する必要があります。

即ち現状を数値化するわけです。現状が数値化されれば、目標も数値化されて、見えない目標に盲目的に突っ走り貴重な時間を浪費することもなくなります。

話を戻して、どれだけ GC が動いているかを把握するためには、右記のようなタスクを作って、ゲーム コンポーネントなどに常駐させると良いでしょう。そして、update メソッドを毎フレームはやりすぎなので 1 秒間に 1 ~

```
// ヒープ使用量を監視するタスク

public long heap;
public long delta;
public void update()
{
    long use = System.GC.GetTotalMemory(false);
    delta = use - heap;
    heap = use;
}
```

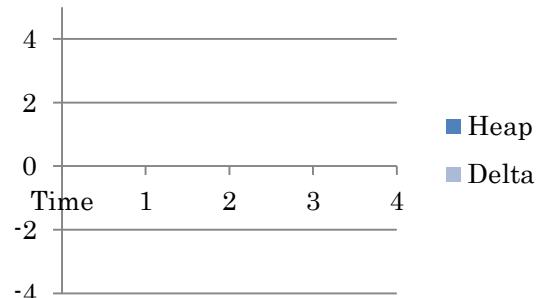
3回程度呼んでやると、ヒープ使用量を把握することができます。フィールド変数 `heap` が現在のヒープの使用量、`delta` が前回 `update` メソッドを呼んでからの変化量です。この時、`delta` が負数を示したら GC が発生したことを意味します。

では、GC の活躍の現状が把握できたところで、どのようにして GC を抑制すればよいのでしょうか？最も理想のパターンはヒープを一切使用しない方法ですが、これをゲームで再現するのは、少々非現実的ですね。そもそも Game オブジェクトを作成している時点でヒープを消費しているため、XNA 上でそれを行うのはまず不可能です。そうなると次に理想のパターンとして、ゲームの出来るだけ最初に必要量を全て確保しておいて、それをひたすら使いまわす方法です。これをプログラム上で実装したものを Flyweight パターンと言いますが、具体的な実装方法は次節以降で解説します。

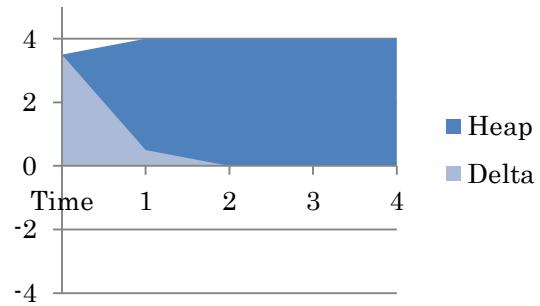
一方、常に一定量を新たに確保し続けるのは、ゲームにとっては余り良くない兆候です。常に確保し続ける、即ち `delta` が一定量を示し続けているということは、言い換えるとヒープの総量が増え続けていることを意味します。前のページでも述べましたが、GC が発生する条件は「最後に GC が動いてから一定量以上新しくヒープが確保されること」です。よって、常に確保を続けていると、いずれ確保された量が一定以上となり、GC が発生してしまうのです。

では、次節以降でどのようなロジックを組めばヒープの確保が抑えられるかを説明します。

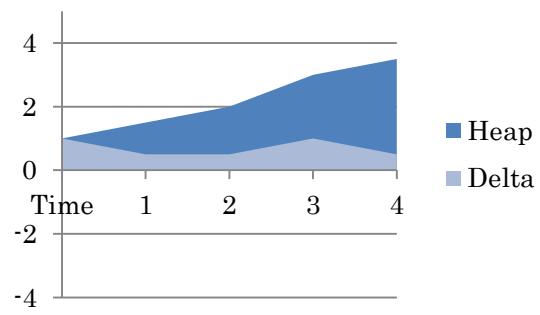
## 最も理想の推移



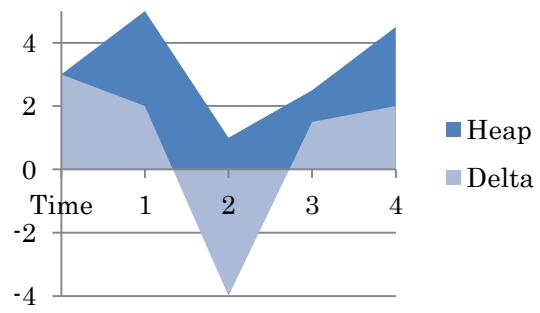
## 現実的な理想的な推移



## 余り良くない推移



## GC発生！



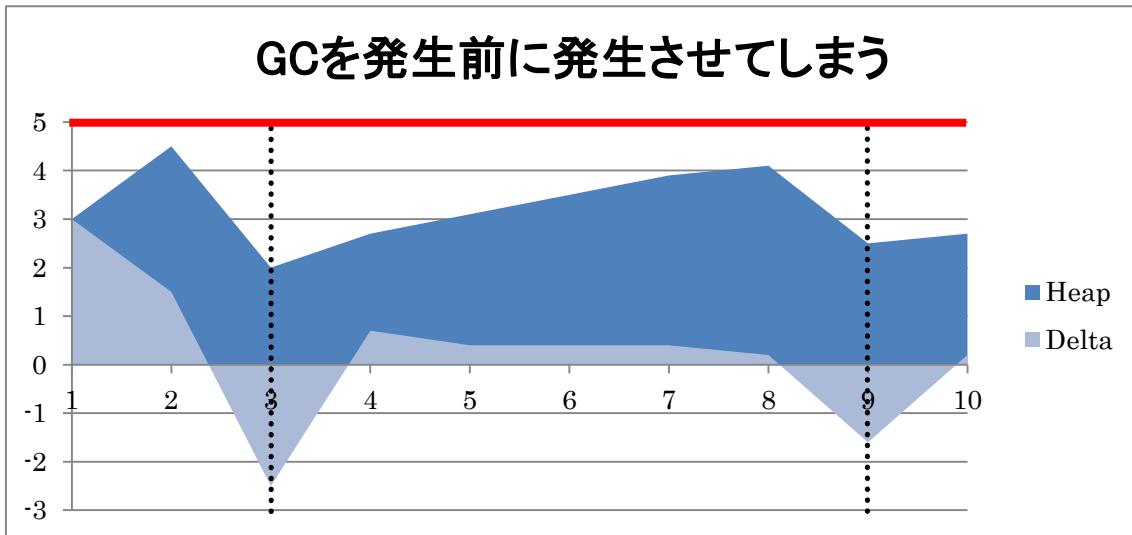
## 敢えてガーベージ コレクションを呼んでしまう

GCを抑えるための方法として最も簡単な方法が、GCを強制的に呼び出してしまう方法です。GCは前節で説明した発生条件に加えて、「手動で発生させる」ことが可能です。

「GC を抑えるために GC を呼ぶなんて、馬鹿じやないのか」と、もしかしたら思われるかもしれません、「発生タイミングをずらす」と言い換えた方がこの場合は解りやすいでしょうか。つまり画面が遷移する(切り替わる)瞬間など、別にプチフリーズが起きても構わないような状態の時に合わせて意図的に GC を発生させると、次に GC が発生するタイミングを遅らせることができるのです。

```
// 手動で GC を発生させる  
System.GC.Collect();
```

この方法は、常に新しく確保される量(前節のコードで出てきた `delta` のことを指します。以下  $\Delta$  と示します)が少なく、かつ画面遷移が頻繁に行われる場合において、特に有効です。 $\Delta$  が少なければ、その分次に発生する GC までだいぶ時間があります。その前に画面遷移してしまえばそこで GC を呼ぶため、また時間の猶予が生まれます。うまくいけば、これだけで GC が勝手に呼ばれないようにすることも可能でしょう。



しかし、この方法は所詮発生を遅らせているだけに過ぎず、根本的にヒープの確保を抑制できているわけではありません。そのため、確保量が多い場合や、連続したアクションシーンが多く、プチフリーズが許容できるチャンスがないなどの場合には焼け石に水程度の効果しか現れません。

そこで根本的な確保を解消するため、前節で名前だけ挙げた Flyweight パターンを紹介していきます。

## Flyweight パターンを使う

ところで Flyweight パターンの説明に入る前に、前章にも似たような名前のものが出てきたことに気付きましたか？ そう、 Singleton パターンと言うのがありましたね。 実はこれらを総称して「GoF による 23 のデザインパターン」と呼ばれ、その名の通り Singleton や Flyweight を含めて 23 パターンのロジックがあります。 本書では、その中でも特にゲーム製作に役立ついくつかのデザインパターンのみに絞って解説していますが、 残りのデザインパターンも今後のプログラミングにおいて大きなヒントとなるようなものばかりなので、覚えておいて損はありません。もし余力があれば、ネットで調べるなり、 デザインパターンの入門書を紐解くなりするのも良いでしょう。

話がそれましたね。 Flyweight パターンとは、複数あるオブジェクトを再利用させることでメモリ使用量を文字通り「フライ級」に減量するロジックのことを指します。 前章のサンプルでは複数生成されるオブジェクトと言えば何があるでしょうか？ 言うまでもなく敵機オブジェクトだけですね。 では前章のサンプルでは敵機オブジェクトはどのように扱われていたでしょうか？ 流れを追ってみましょう。

1. `EnemyManager.create` により `Enemy` クラスを継承したオブジェクトが生成される。
2. 生成したオブジェクトは `EnemyManager.tasks` (`TaskManager` より継承) に格納される。
3. 自機との接触判定が行われ、接觸した瞬間 `EnemyManager.tasks` はクリアされる。

まず 1 にてオブジェクト生成のためヒープが確保されます。 そして 3 にてタスク一覧がクリアされることにより、 1 で生成されたオブジェクトが入っているヒープへのアドレスを知っている変数がプログラム中に誰もいなくなりますね。 その瞬間その部分のヒープは GC による解放の対象となるのです( 実際に GC が発生するのはこの対象が一定以上蓄積された時です )。 それでは、これを下記のようにすればどうなるでしょうか？

1. `EnemyManager.create` にて休眠中の `Enemy` クラスを継承したオブジェクトを探し、 あればそれを呼び起こして 3 へ進む。
2. `Enemy` クラスを継承したオブジェクトを生成し、 `EnemyManager.tasks` に格納する。
3. 自機との接触判定が行われ、接觸した瞬間 `EnemyManager.tasks` に登録された `Enemy` オブジェクトはすべて休眠状態に入る。
4. 画面外に出た敵機も休眠扱いとする。

この場合、最初の頃は休眠中のオブジェクトがないため前者の例と変わりありませんが、時間がたつほど新しくオブジェクトが作られる可能性は減っていきます。 もしさらに GC の抑制を徹底させなければ、 敵機の上限数を仕様で固めた上で `EnemyManager.tasks` の配列を完全固定長にして、起動時

に全数確保の上休眠状態にし、そこから使いまわすと良いでしょう<sup>35</sup>。こうすることで前のページで例示したヒープ確保の「現実的な理想の推移」の状態に近づいたのではないでしょうか？このように、要らなくなってしまったオブジェクトを捨てるのではなく眠らせて、次に別のところで必要になったらそれを使いまわしてやるロジックのことを Flyweight パターンと言います。

Flyweight パターンを使うとゲーム全体のヒープ確保・解放の量が劇的に減りますが、一つ注意点があります。Flyweight パターンによって使いまわされるオブジェクトは中古品です。あちこち設定が変わってしまっているため、しっかりと再初期化をしてあげないと、思わぬバグの原因となります。

それでは、早速前章のサンプルコードに手を加えて、Flyweight パターンを実装してみましょう。変更点は以下の通りです。(Sample1\_12)

- EnemyManager
  - Enemy オブジェクトを作る部分にワンクッシュョンかませて、休眠中オブジェクトを探す処理を加える。
  - 接触時、Enemy のリストを消さずに全数休眠状態にする。
- Enemy 及びその継承クラス
  - 初期化する処理をコンストラクタから分離し、外部から何度も呼び出し可能にする。

---

<sup>35</sup> 実は Sample1\_10までは Flyweight パターンそのもののコードを書いていました。初心者がやらかしそうなミスを再現したコードを書いていたつもりが……慣れって恐ろしい。

```

// EnemyManager.cs

/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="speed">基準速度。</param>
public bool create(float speed)
{
    bool result = false;
    int percentage = rnd.Next(100);
    if (percentage - HOMING_PERCENTAGE < 0)
    {
        result = create<EnemyHoming>(speed);
    }
    else if (percentage - INFERIORITY_PERCENTAGE < 0)
    {
        result = create<EnemyInferiority>(speed);
    }
    else
    {
        result = create<EnemyStraight>(speed);
    }
    return result;
}

/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="speed">基準速度。</param>
/// <returns>敵機を作成できた場合、true。</returns>
public bool create<T>(float speed)
    where T : Enemy, new()
{
    bool result = false;
    int length = tasks.Count;
    for (int i = 0; !result && i < length; i++)

```

```
{  
    if (tasks[i] is T)  
    {  
        result = tasks[i].start(speed);  
    }  
}  
if (!result)  
{  
    T enemy = new T();  
    enemy.start(speed);  
    tasks.Add(enemy);  
    result = true;  
}  
return result;  
}  
  
/// <summary>  
/// 敵機の移動、及び接触判定をします。  
/// </summary>  
/// <returns>接触した場合、true。</returns>  
public bool hitTest()  
{  
    bool hit = false;  
    int length = tasks.Count;  
    for (int i = 0; !hit && i < length; i++)  
    {  
        hit = tasks[i].hitTest();  
    }  
    if (hit)  
    {  
        setup();  
    }  
    return hit;  
}
```

```
// Enemy.cs

/// <summary>初期位置。</summary>
private static readonly Vector2 firstPosition = new Vector2(-SIZE);

/// <summary>
/// コンストラクタ。
/// </summary>
/// <param name="color">色。</param>
public Enemy(Color color)
{
    this.color = color;
    setup();
}

/// <summary>
/// 敵機を強制的にスリープにします。
/// </summary>
public void setup()
{
    position = firstPosition;
    velocity = Vector2.Zero;
}

/// <summary>
/// 敵機をアクティブにします。
/// </summary>
/// <param name="speed">基準速度。</param>
/// <returns>敵機をアクティブにできた場合、true。</returns>
public virtual bool start(float speed)
{
    bool result = !Game1.SCREEN.Contains((int)position.X, (int)position.Y);
    if (result)
    {
        startForce(speed);
    }
}
```

```

        return result;
    }

/// <summary>
/// 敵機を強制的にアクティブにします。
/// </summary>
/// <param name="speed">基準速度。</param>
private void startForce(float speed)
{
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
        pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
    }
    else
    {
        pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
        pos.Y = p % Game1.SCREEN.Height;
    }
    position = pos;
    initVelocity(rnd.Next(1, 3) + speed);
}

```

```

// EnemyHoming.cs

/// <summary>
/// コンストラクタ。
/// </summary>
public EnemyHoming() :
    base(Color.Orange)

```

```
{  
}  
  
/// <summary>  
/// 敵機をアクティブにします。  
/// </summary>  
/// <param name="speed">基準速度。</param>  
/// <returns>敵機をアクティブにできた場合、true。</returns>  
public override bool start(float speed)  
{  
    bool result = base.start(speed);  
    if (result)  
    {  
        homingAmount = HOMING_LIMIT;  
    }  
    return result;  
}
```

## State パターンを使う

さて、前節にて Flyweight パターンを使うことで、ヒープの無駄な確保と解放を抑え、結果として GC の発生を抑えることができました。これでメモリ周りのパフォーマンス改善は完了かと思いつつ、実はこれには非常に大きな欠陥が残っています。

前項で挙げたサンプルには、敵機は三種類いましたよね？そして、それぞれの敵機の固有機能は Enemy から継承することで分割していましたよね？改めて全種類を下に列挙しましょう。

1. 自機に正確に狙いをつけて直進するタイプ。赤色。EnemyStraight クラス。
2. 自機に正確に狙いをつけてホーミング追跡するタイプ。黄色。EnemyHoming クラス。
3. 自機に大難把に狙いをつけて直進するタイプ。赤紫色。EnemyInferiority クラス。

この 3 つのクラスはすべて Enemy 型から継承されているため、そのまま Enemy 型の配列にアップキャストして混同されて格納していました。そして、敵機を生成する際は Flyweight パターンに従い、配列の中から休眠しているオブジェクトを探し、それを望む敵機としてアップキャスト可能かどうかを確認して、出来るならそれを返し、出来ないなら次を探す、と言うロジックでした。実はここに問題が隠れているのです。

前項のサンプルでは敵機は三種類だったので特にさほど大きな問題にはなりませんが、もしこれが数百種類いたとなるとどうでしょう？しかも大抵のゲームは後半戦になるほど、序盤に出てくる雑魚は出てこなくなる傾向がありますよね？そうなると、序盤に出てくる雑魚のクラスは敵管理クラスに「お前敵種類 Z じゃないから使えない」と言われ続けて、ずっと休眠したまま最後まで居残り続けることになるでしょう。さらに敵機が増えてくるともっと深刻なことになります、オブジェクトを使いまわしているつもりが、使いまわしているのはほんの一握りにすぎず、ほとんどのオブジェクトはずっと休眠しているというとんでもないメモリの無駄遣いになってしまいます。ヒープの確保・解放を抑えるどころか、使い回しが不完全なため逆にどんどん新しいオブジェクトを作ってしまい、その結果大量のヒープ確保が生じてしまいます。そしてそれに連動して GC が発生するばかりでなく、休眠中のオブジェクトは GC で解放されないためどんどんヒープが蓄積されて、いずれはスワップ<sup>36</sup>が発生したり、最悪の場合メモリ不足でヒープ確保に失敗<sup>37</sup>して強制終了したりしてしまうケースも考えられます。

---

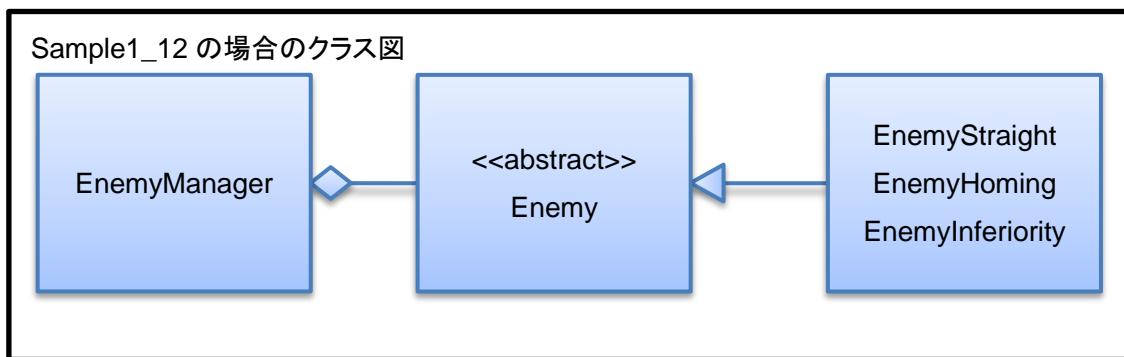
<sup>36</sup> HDDなどの補助記憶装置の一部を仮想的なメモリと見なし、実際にそこにアクセスすることをスワップと呼びます。HDD はメインメモリ(主記憶装置)に比べると圧倒的に遅いため、パフォーマンスが大幅に低下します。最近 SSD など高速な補助記憶装置が出ていますが、結局メインメモリよりバス幅が細いため、そこでネックとなっています。

<sup>37</sup> C#の場合、OutOfMemoryException 例外が飛んできます。

それでは、この状態を解消するためにはどうすればよいのでしょうか？やはり Flyweight パターンは欠陥パターンだったのでしょうか？それとも固有機能を別クラスに分離したのが悪かったのでしょうか？

実は、別に Flyweight パターンが欠陥パターンと言うわけではなく、固有機能を別クラスに分離したのが悪かったのでもなく、固有機能を別クラスに分離する「やり方」が悪かった、と言うのが正解です。では、どの部分が悪かったのか見ていきましょう。

敵機管理クラスに対して、指定したタイプ……ここでは EnemyStraight としましょう……の敵機を作れとゲームロジックが命令したと想定します。ここで敵機管理クラスが抱えている休眠した敵機がなければそのままオブジェクトを生成しておしまいなのですが、休眠している敵機がある場合はリサイクルのためその敵機を使おうとします。でも、休眠している敵機が Enemy を継承した型であることは間違いないのですが<sup>38</sup>、EnemyStraight 型かどうかはそのままでは判りません。判定するためには、is 演算子でその型かどうかを取得するか、または as 演算子で実際に変換してやること<sup>39</sup>で実現できます。そのため、敵機管理クラスはすべての休眠している敵機に対して判定を行ない、EnemyStraight 型の敵機を探し当て、それを使おうとします。勿論見つからなかった時の処理は休眠している敵機がない時と同じです。

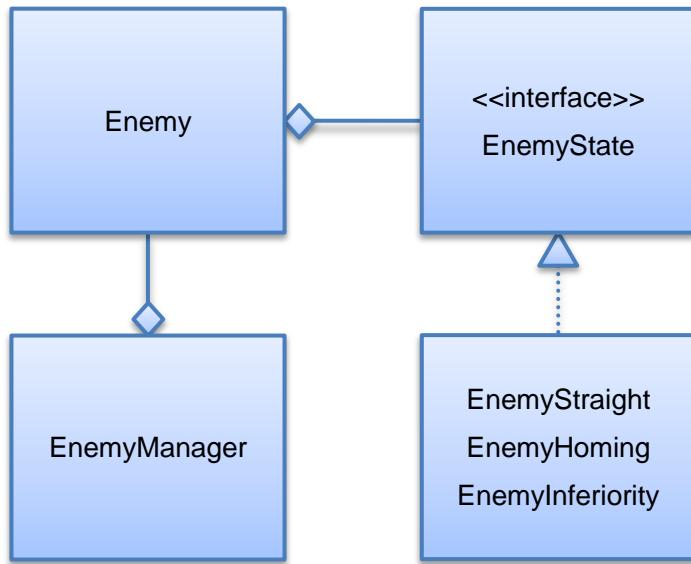


この時、どうしても使い回せないオブジェクトは、Enemy 型であっても EnemyStraight 型でないオブジェクトですね。どうやらこの継承が使い回しの邪魔をしてしまい、無駄遣いに繋がる原因となってしまっているようです。しかし、もし継承をやめるとしたら、固有機能を一体どこに書けばよいのでしょうか？そこで、前章で学習したポリモーフィズムがもう一度出てきます。前回使った時は update 及び draw と言うメソッドを ITask と名付けたインターフェイスにまとめて、それを実装したクラスは「何だか良く解らないけれど、とにかく更新できて描画できる」オブジェクトと見なし、全部一緒にまとめて使っていったことを思い出してください。

<sup>38</sup> Enemy 型は abstract 指定、即ち抽象型なので Enemy 型そのものであることはあり得ません。

<sup>39</sup> 他にも効率は良くありませんが、実際にダウンキャストして例外が飛んでこないかどうかで判定する方法もあります。

State パターンを適用した場合のクラス図



今回は三つの命令 : start、update、initVelocity を持つインターフェイスを作成して、それを実装することにより、「何だか良く解らないけれど、とにかくアクティブにできて、更新できて、そして進行方向を初期化できる」オブジェクトを作成します。それらのオブジェクトには各敵機の固有機能が含まれています。そしてそのオブジェクトを実行時に挿げ替えて使います。このようにオブジェクトを一つの状態とみなす使い方を State パターン、または

Strategy パターン<sup>40</sup>と呼びます。

左に State パターンを適用したクラス図を示していますが、今まで出てきたサンプルで似たようなロジックがあることに気付きましたか？実はシーン管理の部分がこの State パターンとやや似通っています。要はシーンの考え方をさらに拡張して、全ての敵キャラにシーン（正確に自機めがけ直進するシーン、ホーミングするシーン、大雑把に自機めがけ直進するシーン）を持たせるような、そんなイメージですね。今回の State パターンではさらにそれを拡張して、誰がシーンを動かしているかをシーン側に渡しています。従来のシーン管理クラスは各シーンの中で完全に完結していて、シーン管理クラスに何か直接干渉するよう

```

interface IEnemyState
{
    /// <summary>
    /// 敵機がアクティブになったときに呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
    void onStartd(Enemy entity);

    /// <summary>
    /// 1 フレーム分の更新を行う時に呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
    void onUpdate(Enemy entity);

    /// <summary>
    /// 移動速度と方角が初期化された時に呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
    void onInitializedVelocity(Enemy entity);
}
  
```

<sup>40</sup> 両者の違いは使い方や考え方の違いであり、プログラム構造は全く同じです。興味ある人は調べても良いかもしれません、現段階では余り気にする必要はないでしょう。

なことはありませんでしたよね？一方今回の State パターンでは管理側がシーンに対して自分自身のオブジェクトを渡して、シーンに対して積極的に干渉してもらおうとします。そこが State パターンと従来のシーン管理との大きな違いです。

それでは実際に Enemy クラスと各固有実装ロジックのクラスを改良し、State パターンを実装してみます。(Sample1\_13)

```
// Enemy.cs

/// <summary>色。</summary>
public Color color;

/// <summary>移動速度と方角。</summary>
public Vector2 velocity;

/// <summary>ホーミング有効時間。</summary>
public int homingAmount;

/// <summary>現在の敵機の状態。</summary>
public IEnemyState currentState;

/// <summary>
/// 現在活動中かどうかを取得します。
/// </summary>
/// <value>現在活動中である場合、true。</value>
public bool active
{
    get
    {
        return Game1.SCREEN.Contains((int)position.X, (int)position.Y);
    }
}
```

```

///<summary>
/// 1フレーム分の更新を行います。
///</summary>
public void update()
{
    position += velocity;
    currentState.onUpdate(this);
}

///<summary>
/// 敵機を強制的にアクティブにします。
///</summary>
///<param name="speed">基準速度。</param>
public void start(float speed)
{
    float AROUND_HALF = Game1.SCREEN.Width + Game1.SCREEN.Height;
    float AROUND_HALF_QUARTER = Game1.SCREEN.Width * 2 + Game1.SCREEN.Height;
    int AROUND = (int)AROUND_HALF * 2;
    int p = rnd.Next(AROUND);
    Vector2 pos;
    if (p < Game1.SCREEN.Width || p >= AROUND_HALF &&
        p < AROUND_HALF_QUARTER)
    {
        pos.X = p % Game1.SCREEN.Width;
        pos.Y = p < AROUND_HALF ? 0 : Game1.SCREEN.Height;
    }
    else
    {
        pos.X = p < AROUND_HALF ? 0 : Game1.SCREEN.Width;
        pos.Y = p % Game1.SCREEN.Height;
    }
    position = pos;
    currentState.onStart(this);
    initVelocity(rnd.Next(1, 3) + speed);
}

```

```
/// <summary>
/// 敵機の移動速度と方角を初期化します。
/// </summary>
/// <param name="speed">速度。</param>
public void initVelocity(float speed)
{
    Vector2 v = Player.instance.position - position;
    if (v == Vector2.Zero)
    {
        // 長さが 0 だと単位ベクトル計算時に NaN が出るため対策
        v = Vector2.UnitX;
    }
    v.Normalize();
    velocity = v * speed;
    currentState.onInitializedVelocity(this);
}
```

```
// EnemyManager.cs

/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="speed">基準速度。</param>
public bool create(float speed)
{
    bool result = false;
    int percentage = rnd.Next(100);
    if (percentage - HOMING_PERCENTAGE < 0)
    {
        result = create(speed, EnemyHoming.instance);
    }
    else if (percentage - INFERIORITY_PERCENTAGE < 0)
    {
        result = create(speed, EnemyInferiority.instance);
    }
}
```

```
        else
        {
            result = create(speed, EnemyStraight.instance);
        }
        return result;
    }

/// <summary>
/// 敵機を作成します。
/// </summary>
/// <param name="speed">基準速度。</param>
/// <param name="state">敵機の状態。</param>
/// <returns>敵機を作成できた場合、true。</returns>
public bool create(float speed, IEnemyState state)
{
    bool result = false;
    int length = tasks.Count;
    for (int i = 0; !result && i < length; i++)
    {
        result = !tasks[i].active;
        if (result)
        {
            Enemy enemy = tasks[i];
            enemy.currentState = state;
            enemy.start(speed);
        }
    }
    if (!result)
    {
        Enemy enemy = new Enemy(state);
        enemy.start(speed);
        tasks.Add(enemy);
        result = true;
    }
    return result;
}
```

```
// EnemyStraight.cs

/// <summary>
/// 自機めがけて直進する敵機の状態。
/// </summary>
sealed class EnemyStraight
: IEnemyState
{

    /// <summary>クラス インスタンス。</summary>
    public static readonly IEnemyState instance = new EnemyStraight();

    /// <summary>
    /// コンストラクタ。
    /// </summary>
    private EnemyStraight()
    {

    }

    /// <summary>
    /// 敵機がアクティブになったときに呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
    public void onStart(Enemy entity)
    {
        entity.color = Color.Red;
    }

    /// <summary>
    /// 1 フレーム分の更新を行う時に呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
    public void onUpdate(Enemy entity)
    {
    }
}
```

```
/// <summary>
/// 敵機の移動速度と方角が初期化された時に呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onInitializedVelocity(Enemy entity)
{
}
```

```
// EnemyHoming.cs

/// <summary>
/// 自機めがけてホーミングする敵機の情報。
/// </summary>
sealed class EnemyHoming
: IEnemyState
{

    /// <summary>ホーミング時間。</summary>
private const int HOMING_LIMIT = 60;

    /// <summary>クラス インスタンス。</summary>
public static readonly IEnemyState instance = new EnemyHoming();

    /// <summary>
    /// コンストラクタ。
    /// </summary>
private EnemyHoming()
{
}

    /// <summary>
    /// 敵機がアクティブになったときに呼び出されます。
    /// </summary>
    /// <param name="entity">対象の敵機。</param>
public void onStart(Enemy entity)
```

```

        entity.color = Color.Orange;
        entity.homingAmount = HOMING_LIMIT;
    }

/// <summary>
/// 1 フレーム分の更新を行う時に呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onUpdate(Enemy entity)
{
    if (--entity.homingAmount > 0)
    {
        entity.initVelocity(entity.velocity.Length());
    }
}

/// <summary>
/// 敵機の移動速度と方角が初期化された時に呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onInitializedVelocity(Enemy entity)
{
}
}

```

```

// EnemyInferiority.cs

/// <summary>
/// 粗悪な精度で自機めがけて直進する敵機の状態。
/// </summary>
sealed class EnemyInferiority
: IEnemyState
{

/// <summary>クラス インスタンス。</summary>
public static readonly IEnemyState instance = new EnemyInferiority();
}

```

```
/// <summary>疑似乱数ジェネレータ。</summary>
private readonly Random rnd = new Random();

/// <summary>
/// コンストラクタ。
/// </summary>
private EnemyInferiority()
{
}

/// <summary>
/// 敵機がアクティブになったときに呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onStart(Enemy entity)
{
    entity.color = Color.Magenta;
}

/// <summary>
/// 1フレーム分の更新を行う時に呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onUpdate(Enemy entity)
{
}

/// <summary>
/// 敵機の移動速度と方角が初期化された時に呼び出されます。
/// </summary>
/// <param name="entity">対象の敵機。</param>
public void onInitializedVelocity(Enemy entity)
{
    // ここでベクトルをわざと乱して、精度を落とす
    Quaternion q = Quaternion.CreateFromAxisAngle(
        Vector3.UnitZ, (float)rnd.NextDouble() - 0.5f);
}
```

```
        entity.velocity = Vector2.Transform(entity.velocity, q);  
    }  
}
```

今回のサンプルでは Enemy クラスを実体に、各固有機能を State、即ち「状態」としてコードを組みました。状態と実態を完全に断ち切ることにより、すべての敵機は Enemy クラスになり、「Enemy クラスを継承したクラス」ではなくなりました。これによって Flyweight パターンは休眠している敵機オブジェクトを見つけたらとにかく特定の敵機の固有状態を突っ込んで、叩き起こしてやりさえすれば良いので、無駄なオブジェクトが増えることはありません。

ここでどうしても State パターンのイメージがつかない人のために、一つのたとえ話をしましょう。皆さんは「ガチャピン」と言うキャラクターをご存知でしょうか？テレビ番組「ポンキッキ」シリーズの初期の頃からいる定番キャラクターですね。彼はいつ頃からか「ガチャピンチャレンジ」と称して、スノーボードや空手、ロッククライミング、果ては宇宙旅行などと、色々なことをやっています。さて、そんなガチャピンですが、実はチャレンジの種目ごとに着ぐるみの中の人が違う<sup>41</sup>ということをご存知でしょうか？ガチャピン単体では動くことはできず、中に人が入ることでいろんな行動ができます。さらに入る人によっては空手ができたり、また別人ではロッククライミングができたりなどと、中の人によって同じチャレンジでも出来る種目が違ってきます。この場合ガチャピンの着ぐるみが実体(サンプルで言うところの Enemy クラス)で、ガチャピンの中の人が状態(サンプルで言うと三種の固有機能クラス)、そしてもしほかの固有機能を実装したければ IEnemyState インターフェイスを実装さえすれば誰でもガチャピンの中に入ることができる、と例えることができます。

もしかしたらガチャピンを知らない人もいるかも知れないので、もう一つたとえ話を出しておきましょう。ファミリーコンピュータ、略してファミコンをご存知ですか？知らない人は現行のコンシューマ ゲーム機(以下ゲーム機、PLAYSTATION3 や XBOX360、Wii などを指します)をイメージしてください。流石に本書を読んでいる方でゲーム機を知らない人はいないでしょう。ゲーム機は、原則ゲームソフト(カートリッジやディスクなど)がないとゲームができません。ですが、ゲームソフトを差し替えることでその数の分だけゲームが楽しめるようになります。State パターンも同じことが言えます。ゲーム機が実体(サンプルで言うところの Enemy クラス)で、ゲームソフトが状態(各敵機の固有機能クラス)です。そして、そのゲーム機のカートリッジは、ほかのゲーム機では原則プレイできません。たとえば GAMEBOY ADVANCE のカートリッジは、NINTENDO 3DS に挿入できませんが、これはそもそもカートリッジのコネクタ形状が違うから当然ですよね。同様に PlayStation2 のゲーム ディスクを XBOX360 に挿入しても、当然ですが起動できません。これは、ディスク自体は同じ DVD でもソフトウェア的なコネクタ形状が違うため起動できないのです。そしてプログラムの世界では、このソフトウェア的なコネクタ形状となる

<sup>41</sup> 建前上、私個人の考えとしては「中の人などいない！」ですが……

ものがインターフェイス、今回のサンプルで言うところの `IEnemyState` なのです。

そう言えば、これとよく似た説明をポリモーフィズムの時にも説明しましたね。この概念はゲームプログラマムに限らず、オブジェクト指向プログラミングの中で非常に重要……もとい、最も重要とまで言ってしまっても過言ではないところなので、絶対に覚えておきましょう。そんなオブジェクト指向プログラミングの神髄とも言えるような `State` パターンですが、私の周りだけでしょうか、妙にマイナーな感じが伺えます。これは私なりの仮説で、何らかの裏付けや根拠があるわけではないのですが、極論 `State` パターンを知らずともポリモーフィズムまでマスターしてしまえば、たいていのゲームは作れるようになってしまいます。一方 `State` パターンは意図して勉強しないと中々自然には思いつくものではありません<sup>42</sup>。そのため、ポリモーフィズムまでマスターしてしまえば、あとは自力でやっていけるため、ここまで辿り着く人が少ないのではないか、と勝手に推測してみたりしています。

ところでそんな `State` パターンにも、実は一つだけ欠点があります。それは「完全なカプセル化を実現するのがちょっと大変になる」ことです。`State` パターンでは実体クラスが状態クラスに対して積極干渉してもらうのが前提であるうえ、C#の言語仕様上特定のクラスにのみ公開と言うアクセス制御が存在しないため、「現在適用されている状態クラスにしか弄らせてあげない」というロジックを実装するのが難しくなります<sup>43</sup>。厳密には全く出来ないわけではないのですが、隠蔽したいフィールドを別クラスに持たせたりなどの一工夫が必要で、どうしても実装が複雑化することは避けられません。また、コードを一目見ただけでは、そのプロパティやメソッドが外部からの読み書き対応しているかどうか、状態からのみ読み書き対応なのかの区別がつきにくくなります。そもそも状態クラス自体が実体クラスの外部にあるものなので、外部のクラスに内部を弄らせてしまうと、その時点でカプセル化は半分崩壊してしまっていることになり、ある意味 `State` パターンをとるか、カプセル化をとるか、でトレードオフな関係になっています。

本書では、カプセル化よりも `State` パターンを優先したサンプルコードを掲載しています。但し、`Sample1_15` だけは `Sample1_14` をベースにカプセル化と極力両立したサンプルとなっていますので、カプセル化しないと気持ち悪いと思う方は、よく見比べて扱い方を覚えておくと良いでしょう。

さて、前回敵機だけに適用していた `State` パターンですが、`State` パターンの部分だけを切り出して汎用化することにより、殆どのタスクで使えるようになります。次ページに汎用化したコードを掲載します。(Sample1\_14)

---

<sup>42</sup> シーン管理ロジックまで自力で出来ていた私でも、2009年の暮れに会社の同僚のコードが `State` パターンで組まれていて、そのコードに触れてからある程度操れるようになるまで、二か月はかかりました。

<sup>43</sup> C++ではフレンドクラスを使う「逃げ道」もあるのですが……

```
// Istate.cs

/// <summary>状態表現のためのインターフェイス。</summary>
interface IState
{
    /// <summary>
    /// <para>状態が開始された時に呼び出されます。</para>
    /// <para>このメソッドは、遷移元の<c>teardown</c>よりも後に呼び出されます。</para>
    /// </summary>
    /// <param name="entity">この状態を適用されたオブジェクト。</param>
    void setup(Entity entity);

    /// <summary>1 フレーム分の更新処理を実行します。</summary>
    /// <param name="entity">この状態を適用されたオブジェクト。</param>
    void update(Entity entity);

    /// <summary>1 フレーム分の描画処理を実行します。</summary>
    /// <param name="entity">この状態を適用されたオブジェクト。</param>
    /// <param name="graphics">グラフィック データ。</param>
    void draw(Entity entity, Graphics graphics);

    /// <summary>
    /// <para>状態が開始された時に呼び出されます。</para>
    /// <para>このメソッドは、遷移元の<c>teardown</c>よりも後に呼び出されます。</para>
    /// </summary>
    /// <param name="entity">この状態を適用されたオブジェクト。</param>
    void teardown(Entity entity);
}
```

```
// StateEmpty

/// <summary>何もしない状態。</summary>
sealed class StateEmpty
    : Istate
{
```

```
/// <summary>クラス オブジェクト。</summary>
public static readonly IState instance = new StateEmpty();

/// <summary>コンストラクタ。</summary>
private StateEmpty()
{
}

// 以下、IState の殻の実相につき、省略。
}
```

```
// Entity.cs

/// <summary>State パターンにおける実体となるクラス。</summary>
class Entity
: ITask
{

/// <summary>次に変化する状態。</summary>
public IState nextState;

/// <summary>汎用カウンタ。</summary>
public int counter;

/// <summary>コンストラクタ。</summary>
/// <param name="firstState">初期の状態。</param>
public Entity(IState firstState)
{
    currentState = StateEmpty.instance;
    nextState = firstState;
}

/// <summary>現在の状態を取得します。</summary>
/// <value>現在の状態。初期値は<c>CState.empty</c>。</value>
public IState currentState
{
```

```
get;
private set;
}

/// <summary>1 フレーム分の更新を行います。</summary>
public virtual void update()
{
    currentState.update(this);
    commitNextState();
    counter++;
}

/// <summary>1 フレーム分の描画を行います。</summary>
/// <param name="graphics">グラフィック データ。</param>
public virtual void draw(Graphics graphics)
{
    currentState.draw(this, graphics);
}

/// <summary>オブジェクトをリセットします。</summary>
public virtual void reset()
{
    nextState = StateEmpty.instance;
    commitNextState();
    counter = 0;
}

/// <summary>予約していた次の状態を強制的に確定します。</summary>
public void commitNextState()
{
    if (nextState != null)
    {
        IState prev = currentState;
        currentState = nextState;
        nextState = null;
    }
}
```

```
    prev.teardown(this);  
    currentState.setup(this);  
}  
}  
}
```

## ステップ 5:もっとスマートに美しく書こう

このステップまで到達できた方、即ちポリモーフィズムと Flyweight パターン、そして State パターンをマスターするに至った方は、殆どのゲームプログラムの基礎部分をマスターしたも同然と言ってしまっても過言ではないでしょう。3D ゲームを作りたければ、この基礎部分に 3D 描画機能を乗せれば済みますし、オンラインゲームを作りたければ、同じく通信処理を乗せてしまえば済んでしまいます。それどころか、うまくいけば殆どコーディングせずに済んでしまうかもしれません。State パターンは各状態で干渉が発生しにくく、また今回のサンプルにおいては構造が Game1 クラスに似ているため、Web や書籍などに散在している各種サンプルのソースコードを転用しやすいのです。

しかし、実際に企業など団体の中でゲームプログラマとして活躍するためには、これだけだと少々力不足です。勿論、他にも一般常識やコミュニケーション能力などの重要な要素がありますが、本書の趣旨から逸れてしまうためここではゲームプログラミングに直接関係のある部分だけを取り上げます。

## コーディングスタイルを統一する

企業などでゲームを開発する場合、一人だけで制作することは滅多にありません。本当に工数が一日程度など特別に小さいプロジェクトの場合一人で開発することもありますが、多くの場合そのゲームに関わるプログラマも二人か三人で組んで作業します。

そのような集団開発の現場で問題になるのが「コーディングスタイル<sup>44</sup>」です。プログラマによっては、例えば関数の中括弧の位置や構文中におけるスペースの空け方、インデントの方法などプログラミングの「癖」のことを示します。これがバラバラだとお互いに仲間のコードが大変見づらくなり<sup>45</sup>、精神衛生上もあまり良いとは言えないでしょう。

もし集団で開発する場合、コーディングスタイルはできるだけプログラミングを始める前に統一するとよいでしょう。コーディングスタイルに優劣はほとんどありません。相當に奇抜なスタイルの場合は問題になることもあるかもしれません、ある程度メジャーなスタイルの場合は、可読性や利便性など、どのスタイルでもあまり優劣はなく、もはや「なんか知らんけど（あるいはあまり客観的でない理由で）、今使ってるやり方がいい」と言った、例えるなら宗教論争と同レベルになっています。

本書でもコーディングスタイルの優劣を論ずるつもりはありません。筆者のコーディングスタイルを布教したいわけでもありません。本書において特に訴えたいのは「今いるチーム間でコーディングスタイルを統一すること」なのです。

---

<sup>44</sup> コーディングルール、コーディング規約などと称されることもあります。

<sup>45</sup> ひどくなると、まるで「散らかった部屋」を見ているような錯覚に襲われることもあります。

## ネーミングルールを決めよう

コーディングスタイルごとに優劣はあまりないと前項で述べましたが、一部においては可読性に影響が出てくる部分もあります。その一つはネーミングルールです。

例えばサンプル 1「360° 弾避けゲーム」において、HUD を描画する機能に「drawHUD()」と命名しました<sup>46</sup>。しかし十人いれば十通りの命名の仕方があると思います。以下に例を幾つか示します。

- drawHUD()
- DrawHUD()
- \_DrawHud()
- drawString()
- func011()

入口 1 つに出口 1 つ

たとえば、弾を作る処理

コピペはバグの温床

**Builder パターン**

せっかちな人は **Façade パターン**

バグ対策

テストをしよう

手動より自動

ユニットテストツール紹介

---

<sup>46</sup> Sample1\_03 以降参照。すぐにスコア機能とプレイヤー機能に分けられて、それぞれに統合してしまいましたが……

## 第4章 - あとがき

ゲームプログラミングの心得

できるだけ小さく作れ

小さなゲームでも大きなゲームの作り方を

ただし学習中に限る

バージョン管理ツールを使おう

プログラムに限った話ではない

バシバシ公開して叩かれろ

ソースも公開してしまえ

パフォーマンス改善は最後に

## 付録

UML の解説

サンプルプログラム紹介

- ゲーム 1:「360° 弾避けゲーム」(ゲーム 1 は駄目なコーディング例付き)
- ゲーム 2:「マリオブラザーズのクローン」

SVN リポジトリ(仮設/仮設版は一般非公開): <http://3535.rei.mu/GameOOP/>  
TODO: 本公開はエクスポートの上 Zip 化する。

開発環境の設定

## 索引

- Flyweight パターン, 155, 157
- GC, 152
- Hello, world, 10
- null, 152
- Singleton パターン, 41, 149
- State パターン, 150, 166
- Strategy パターン, 166
- 委譲, 147
- インターフェイス, 89
- インヘリタンス, 88
- ガーベージコレクション, 152
- カプセル化, 65, 69, 88
- 継承, 88
- 工期, 9
- 工数, 9
- 工数見積もり, 9
- 構造体, 44, 65
- コーディングスタイル, 182
- サブルーチン, 35
- ジェネリックス, 141
- 実績, 11
- 車輪の再発明, 57
- 仕様書, 12
- スタック, 152
- 総称型, 141
- 定数化, 41
- ドキュメンテーション, 26
- 名前空間, 45, 136
- 人月計算, 9
- ヒープ, 152
- ポリモーフィズム, 65, 88, 136
- マジックナンバー, 41, 136
- 予実管理, 11

## 筆者自己紹介

野村 周平 P/N:真久 秀(略してまく)

三十路間近の公私共にプログラマ。中学校の部活動で触られた N-88 日本語 BASIC がきっかけで、それ以来今に至るまでどっぷりとゲームプログラミングの世界に入り込み、同人やフリーでいくつかの作品をリリースした。

業務での開発歴は五年程度。ウェブアプリとPCゲームを2:1位の比率で開発してきた。副業で専門学校にてゲームプログラミングの講師をやっている。こちらはまだ始めて一年の新人です。

筆者 Web サイト「danmaq」

<http://danmaq.com/>

大きなゲームの為のプログラミング手法  
ゲームプログラマのためのデザインパターン for XNA/C#  
2011年5月9日 初版発行

著者:野村 周平

発行元:danmaq

東京都墨田区東向島 2-36-12

オリエントビル 57 1115号室

内容の不明点などご意見は

メールでお気軽にどうぞ。

[info@danmaq.com](mailto:info@danmaq.com)

