オブジェクト指向設計について

Object Oriented Design

小林　健一郎
Ken-ichiro KOBAYASHI

（平成21年10月７日受理）

要旨

オブジェクト指向は、決して目新しい、あるいは、難解な哲学ではなく、ごく自然に活用される技法である。

にもかかわらず、必ずしも、教えやすいものではない。本小論では、この困難を示し、シューティングゲーム等を教材として使う解決例を示したい。

１．はじめに

「オブジェクト指向」という言葉は、日本語としてわかりにくい表現であるが、「オブジェクトを重視する」という程度の意味でしかない。ここでオブジェクトとは、「データと機能を合わせたもの」つまり「機能をもったデータ構造」のことである。

これは、一般に、次のように説明される。

> オブジェクト指向とは、次の３つの機能を利用したプログラミング技法である。
>
> １．データ・機能のカプセル化（クラス）
> ２．データ・機能を他のデータ構造に引き継がせる（クラスの継承）
> ３．継承したクラスの統一的扱い（ポリモーフィズム）
>
> 一般に、データ・機能をカプセル化するのに、クラスが使われる。クラスとは、「機能を持ったデータ構造」の設計図のことであり、その設計図をもとに作られた具体的なデータ構造をオブジェクトという。

この説明は、必ずしもわかりやすいものではないが、具体例を見せれば、カプセル化と継承までを理解することは、それほど難しいことではない。適当な機能をもったクラスを具体的に示し、そのオブジェクトの生成と利用を示せばよいのである。この点に関しては、著者はゼミ等も含めて、数年来C++[１]の講義を持っているが、指導上の困難を感じたことはない。

一方、ポリモーフィズムの説明はなかなか難しいが、それは、必要になったときに理解

すればよいと思われる。

しかし、簡単なのはオブジェクト指向の定義の理解であって、オブジェクト指向を自分で利用することは簡単ではない。学生が自分でオブジェクト指向プログラミングができるようになるためには、より深い理解が必要なのである。

一般に、簡単な例を見てクラス等の定義を理解した学生の多くは、次に、「なぜクラスというものが必要なのか」と疑問に思うようだ。当然、意義がわからなければ使うこともできない。

クラスを書く利点は、上記の説明の中にある。つまり、「データを保護する（カプセル化）のに役立つ、また、継承・ポリモーフィズムにより再利用がしやすい」である。しかし、初心者がオブジェクト指向の世界に入っていくためには、わかりやすい実用例が必要である。

著者は、オブジェクト指向の実用的な利点のうち、「現実をシミュレートするようにプログラムが書けること」が、非常に重要だと考える。一般に、手続き指向では、論理が優先される。しかし、プログラムが長くなると論理的なミスによるバグを取り除くことが困難になってくる。オブジェクト指向型プログラミングは、「現実をシミュレートする」ことで、プログラムに「直感的な理解」を与え、それにより、より間違いを少なくできるのである。

しかし、一方で、「直感的な理解」は、コードをかえって複雑化させたり、また、効率を落としてしまうこともある。もちろん、そのまま現実をシミュレートすることもできない。

したがって、実際のプログラミングでは、このような長所・短所を理解した上で、最適なコードを判断をしていくことになる。

ところが、初心者は、このような説明を理解し、それを実践しようと思っていても、具体的にはどうしてよいかわからず、途方にくれてしまうことが多い。本小論では、オブジェクト指向を「使えるまで理解する方法」を、ゲームプログラミングを例に考察する。

ここで、教える対象は、C++の基礎を学んでいる学生とする。

なお、本学は、2005年度より、著者の所属する情報デザイン学科を開設した。関連して、「情報をデザインする」ということの概念説明が求められることも多い。本小論は、プログラムの詳細を見なければ、ソフトウェアの非専門家に対しても、システム設計の立場からの「情報をデザインする」を示すことにもなると考える

2．シューティングゲーム

オブジェクト指向がその良さを発揮するのは、主に、大きなプログラムである。しかし、教育に使えるプログラムのサイズは、どうしても限られてしまう。そのため、プログラムの題材（教材）選びが難しい[2][3]。

著者のゼミでは、2007年度より、ゲームプログラムを題材にしている。ゲームは

DirectX[４]を利用したWindows上のもので、言語はC++とした。また、DirectXを直接利用するのではなく、DirectX APIを簡単に呼び出すための独自ライブラリを作った。このライブラリには、ゲーム自体のフレームワーク、画像表示、音声機能などがある（付録を参照のこと)。

以下では、シューティングゲームを例に、「オブジェクト指向をどう教えるか」を説明したい。

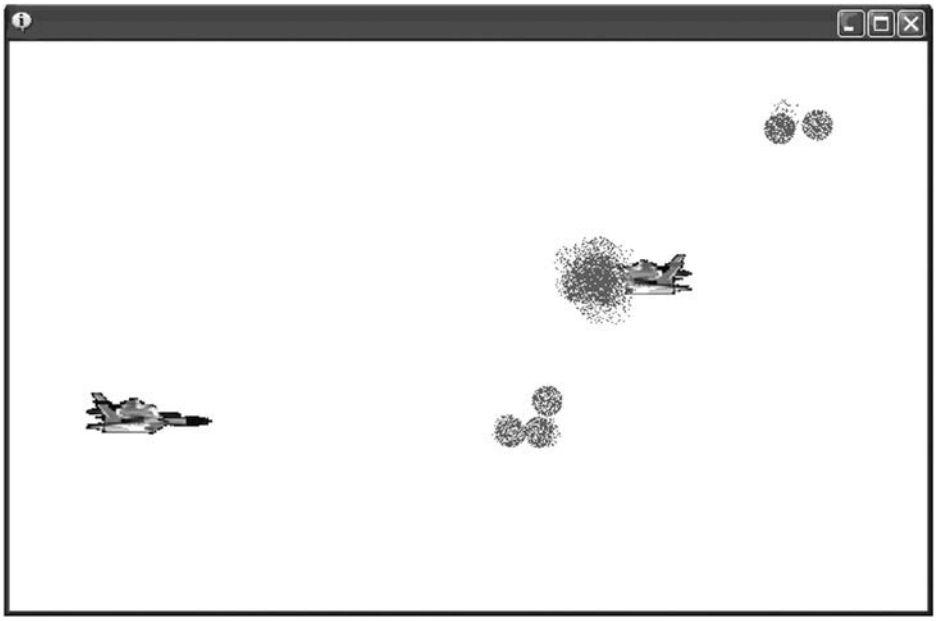

図２．１　シューティングゲーム

ここで言うシューティングゲームとは次のようなものである。

１．１台のパソコンにゲームコントローラを２つ接続し、２人で遊ぶ。
２．それぞれのコントローラで、「戦闘機」を操る。
３．戦闘機は、十字キーによる移動とボタンによる弾丸の発射ができる。
４．相手の戦闘機に弾丸を当てれば、その人の勝ちになる。

図２．２　ゲームコントローラ
左にあるのが十字キー、ボタンは各所にある

B.Stroustrupの「プログラミング言語C++」には、「はじめに使いたいクラスを決めよ」というアドバイスがある（[１]のp72。原文では、「使いたい型を決めよ」）。このアドバイスを実行できるようになれば、オブジェクト指向の初心者を卒業したことになるだろう。このゲームは、そのためのわかりやすい例になっていると考える。

オブジェクト指向プログラミングでは、クラスの候補として、「登場物」から考えることが多い。もちろん、そうでない場合もあるが、多くの場合、この方法はうまく機能する。それが、前節で書いた「現実をシミュレートするように考える」ということである。ここでは、その方針に従う。

シューティングゲームの登場物は、「戦闘機」と「弾丸」である。また、「背景」や「障害物」などを考えることもできる。さらに、「ゲーム（ゲーム自体）」も登場物と考えることができる。

オブジェクト指向設計では、これらの関係を精査し、淘汰・統合・拡充していくことになるが、ここでは、「戦闘機」（Fighter）「弾丸」（Bullet）「ゲーム」（Game）を考えることにする。そして、これらに対応するクラスを考える。そのクラスのオブジェクトが、ゲームプログラム内で活躍することになるわけである。

著者によるライブラリを使うと、これらのクラスを、たとえば、次のように作ることになる。

```
class Fighter : public KObject
{
```

```
public:
    void init(const std::string& fn);
};

class Bullet : public KVelObject
{
public:
    void init();
};

class Game : public KGameFrame
{
    Fighter fs_[2];
    int bn_;
    Bullet bs_[100];
public:
    void init();
    void onJoyStick(int i);
    void show();
};
```

ここでKObjectは「ゲーム内の物体」の基底クラス、KVelObjectは、「速度を持つゲーム内の物体」の基底クラス（KVelObjectはKObjectの派生クラス）、KGameFrameはゲームの基底クラスである。

これらのクラスのオブジェクトは、initで初期化を行う。Bulletの場合、画像ファイル名をbullet.bmpとすると、

```
void Bullet::init() {
    setTex("bullet.bmp");
    setValid(false);
}
```

などと書くことができる。

ここで、setTexはKObjectの画像を設定する関数（テクスチャのセット）である。また、setValidは、物体の有効／無効（表示／非表示）を設定する関数で、setValid(false);により、弾丸を無効（非表示）にしている。弾丸は、発射されたときにはじめて有効化（表示）したいからである。なお、KObjectはデフォルトで有効（表示）になっている。

KObjectには、他に、x座標y座標を設定する関数setXYやx座標y座標に加算するaddX、addY、角度（水平方向右向きを 0 度とする）を設定するsetDegなどがある。また、KObjectには、画像を表示するshowという関数もある。

そして、これらの関数はKVelObjectに継承されている。また、KVelObjectは、「速度を持つゲーム内の物体」を表しており、その速度（x方向y方向）は、setVXVYで設定することができる。

FighterもBulletとほぼ同様に書くことができるが、ここでは、initが画像ファイル名を受け取るようにした。すると、関数の定義は次のように書ける。

```
void Fighter::init(const string& fn) {
    setTex(fn);
}
```

Gameは、これらの弾丸や戦闘機を持つクラスである。２人対戦であるので、戦闘機は２機、弾丸は簡単に100発とした。実際には、「どちらの戦闘機が発射する弾丸か」の区別が重要になってくるが、それは次の節以降で議論することにし、ここでは、両者が同じ弾丸の配列bs_から、弾丸を使っていくことにする。また、上のコードでは、さらに「次に何発目の弾丸を発射するか」を保持するbn_というメンバ変数を持たせることにした。弾丸は、配列の先頭から使っていくことにしたいので、bn_は０に初期化することにする。すると、Game初期化initは次のように書くことができる。

```
void Game::init() {
    fs_[0].init("fighter1.bmp");
    fs_[0].setXY(100, 300);
    fs_[1].init("fighter1.bmp");
    fs_[1].init(500, 300);
    bn_ = 0;
    for(int i = 0; i < 100; ++i)
        bs_[i].init();
}
```

このような準備でGameの表示関数showは次のように書くことができる。

```
void Game::show() {
    for(int i = 0; i < 100; ++i)
        bs_[i].show();
    fs_[0].show();
    fs_[1].show();
}
```

次に戦闘機の操作であるが、それは、KGameFrameから継承するonJoyStickを上書きして行う。これはゲームコントローラの状態を調べ、それにより、ゲームを進行させる関数である。このゲームでは、コントローラを２つパソコンに接続するが、それらは、

onJoyStickのint型引数で区別される。
すると、onJoyStickは、次のように書ける。

```
void Game::onJoyStick(int i) {
    fs_[i].addX(jsX());
    fs_[i].addY(jsY());
    if(onButton(0)) {
        if(bn_ >= 100) return;
        bs_[bn_].setXY(fs[i].getX(), fs[i].getY());
        bs_[bn_].setVXVY(10 * dcos(fs[i].getDeg()), 10*dsin(fs[i].getDeg()));
        bs_[bn_].setValid(true);
        ++bn_;
    }
};
```

ここで、jsX()、jsY()は、ゲームコントローラの十字キーで指定された、x方向y方向への移動距離を表している。この値を、戦闘機の座標に足すことで、戦闘機は位置を変えるのである。

また、一般に、ゲームコントローラには、複数のボタンがついており、それらは整数で識別される。上記のonButton(0)は、「第 0 ボタンが押されているかどうか」を判定する関数であり、このようにif文を書くことで、第 0 ボタンが押されたときの動作を指定するとができるのである。

上のコードでは、第 0 ボタンが押された場合、押した側の戦闘機から、戦闘機の方向に弾丸が出るようにしている。getX、getYで戦闘機の位置を取得し、setXYで弾丸をその位置に置き、setVXVYで弾丸に戦闘機の向きに沿った速度を与えている。（dcos、dsinは、360度法によるコサイン関数、サイン関数である。）これにより、弾丸であるKVelObjectオブジェクトは、自動的にその速度で動き出すのである。

ただし、弾丸を発射後は「次に発射する弾丸の番号bn_」を 1 増やし、if文のはじめではbn_をチェックし、これが100以上になった場合は、弾丸を発射しないようにしている。撃てる弾丸は100発までにしたのである。（ちなみに、「if(bn_ >= 100) return;」を「if(bn_ >= 100) bn_ = 0;」とすれば、弾丸を無尽蔵に発射できるようになる。その場合、画面中に表示できる弾丸の最大数が100になる。）

これだけのコードで、ゲーム画面（戦闘機の表示と弾丸の発射）が表示できるのである。これは、一般的なDirectXプログラミングとしては、非常に少ないコード量であると言えるだろう（一般に、画像を表示するだけのサンプルプログラムでも200行を超える[4]）。それは、もちろん、複雑な部分を独自ライブラリの中に押し込んだためであるが、それが成功しているという事実が、オブジェクト指向の有用性を示していると考えられる。また、上記のコードは、 1 行 1 行の意味が明確であると思う。このことも、オブジェクト指向初心者にとって、よい教材になっていると考えるのである。

ゲームとして次に必要な機能は、「弾丸が戦闘機に当たったときの処理」だろう。独自ライブラリには、アニメ機能もあるので、「当たり」を判定できれば、爆発シーンのアニメを表示することも簡単にできる。

問題は「当たり判定」である。「当たり判定」は、ゲームプログラミングにおいてさまざまに考察される重要な対象である。たとえば、「弾丸が戦闘機をすり抜ける」などの現象が起こらないようにしなければならないからである。しかし、本小論では、そのように一般に議論されている問題は取り上げず、「オブジェクト指向の教材」としての側面を議論したい。それは、設計の議論になる。

3．設計について

プログラミング言語やライブラリにある機能を無反省に利用するだけでは、オブジェクト指向プログラミングにはならない。「オブジェクト指向を利用する」ということは、「プログラムをオブジェクト指向的に設計する」ということである。

前節のプログラミングは、無反省的であるが、一定の成功を収めているように見える。最初に「登場物」を考察し、対応するクラスを書き、継承を利用してコードを簡単で明瞭に書くことができたからである。

ただし、「それは、まだプログラムが複雑でないから」ということもできる。ここまでのことなら、（ライブラリを利用すれば）どのように書いても簡単に書けてしまうのである。しかし、ゲームに「当たり判定」を組み込む辺りから、コードが次第に複雑化し、オブジェクト指向の１つの大きな目標である「保守管理しやすいプログラム」（「すっきりして直しやすいプログラム」）でなくなってくることが多い。

初心者は、そのような「モンスタープログラム」を一度書いた後の方が、設計の重要性を認識するようなので、時間があれば、「自由に書かせる」ということも教育的にはよいかと思う。ただ、いずれにしても、最終的に重要なことは、「設計をすること」である。

実は、第２節でも設計は行っている。ただ、ごく簡単に述べ、実際のコードを書いていったのだ。これは、本小論の説明上の都合でもあるが、初心者のプログラミングの実際に非常に近いものでもある。

しかし、以後は、設計に焦点を当てていく。前節で考えたシューティングゲームは、実は、以下のような設計（構造図）によるものである。

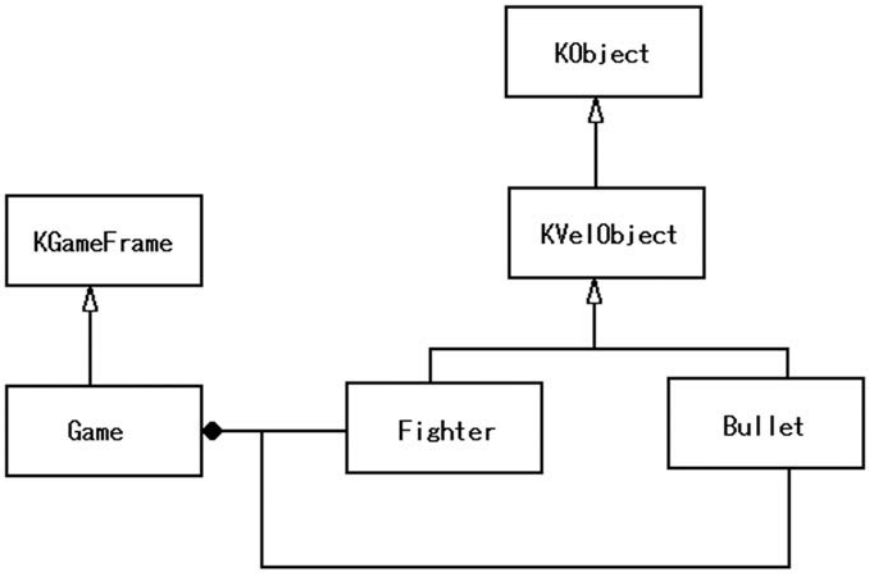

図３．１　第２節のゲームの設計

ここで、白抜きの三角を持つ線は継承を表し、黒いひし形を持つ線は合成を表している。このゲームの基本部分は、「GameがFighterとBulletを（内部に）持つ」ということになる。

このような設計が良いか悪いかを、できるだけコーディングの前に考えておかなければならないのである。

ただし、一般に、「（設計によっても、何によっても）プログラムの複雑さはなくならない」ということは指摘しておきたい。プログラムが複雑な仕事をするものであれば、その複雑さは、どうやっても消えてなくならない。ただ、分類され、整理され、分割される（その結果、薄められる）だけなのである。初心者の多くは、「設計＝複雑さの回避」と考えるようだが、実際には、「設計＝不要な複雑さの回避」ではあっても、そのプログラムにおける本質的な複雑さはなくならないと教えるべきである。設計において重要なことは、「本質的な複雑さを、どのように整理し分割するか」について、決断をするということである。

決断をするためには、いくつか仮りの設計を行い、比較する必要がある。それを、本小論では、シューティングゲームの例で考えたいのである。

４．「当たり判定」と設計

実際のプログラミングは、仕様の詳細を決め、設計してから行われる。したがって、第２節のプログラミングは、仕様・設計が極めて簡単で、非現実的なアプローチであったと言える。（ただし、実際には、多くのプログラムがこのようなアプローチで書かれている

ことも事実である。）

この節では、主に設計を議論したい。ただし、ゲーム全体を議論すると、膨大なものになるので、「当たり判定」に付随する部分のみを取り上げて考察することにする。

今の例のシューティングゲームにおいて、「当たり判定」では、たとえば、次のような仕様が考えられる。

仕様案１
　自分の弾丸は敵にしか当たらない。
　つまり、自分の弾丸は自分に当たらない。

仕様案２
　自分の弾丸は敵にも自分にも当たる。

第２節のプログラムでは、弾丸は、戦闘機の方向に向かって発射するので、自分に当たる可能性はなさそうだが、実際には、「自分が発射した弾丸を戦闘機が追い抜く」などの理由により、ゲーム上では、（もちろん、それも仕様によるわけだが）自分の発射した弾丸と自分の戦闘機が交差することはよくあることである。

したがって、本来は、このような仕様を、はじめに決める必要があったのだ。ここでは、仕様案１を採用することにする。

第２節では、「実際のゲーム」をイメージしながら、「登場物」を特定し、それらのクラスを作り、その組み合わせでゲーム（の土台）を作った。「現実をシミュレートする」ように考えたのである。ゲームをおもしろくするには、（実際には決してあってほしくないが）現実の空中戦をイメージすることもできるだろう。しかし、「自分の弾丸は自分には当たらない」というのは、現実的ではない制約である。現実の戦闘機なら、自分の発射した弾丸に当たれば大損害になるはずだからだ。このような制約は、単に「プログラムは現実とは違う」という理由から、頻繁に起こるものである。そのような場合、プログラマは、現実をヒントにしながら、そのプログラム独自の構造を考えていくことになる。

一般に、オブジェクト指向設計で重要なことは、「どのクラスが何を持ち、どのクラスが何を実行するか」である。今の場合、「どのクラスが当たりを判定するか（つまり、弾丸と戦闘機の交差を判定するか）」が問題になる。そして、それは「どのクラスが弾丸を持つのか」ということにも関係する。

図３．１の設計はどうだろうか。この図は「当たり判定を誰がするか」までは示してはいない。したがって、図３．１は、「設計の一部」なのである。この足りない分を、これから考えていくわけだ。それにより、図３．１を変更していくこともある。

図３．１は「当たり判定を誰がするか」までは示してはいないと書いたが、このような構造からもっとも自然なのは、「Gameが当たり判定を行う」だろう。Gameがすべて（戦

闘機と弾丸）を持ち、したがって、すべての情報を持っているからだ。Gameが、各弾丸について「誰が発射したのか」を識別し、弾丸と戦闘機の当たり判定を行えばよい。識別と言っても、弾丸の配列を２つにし、それらをそれぞれの戦闘機に割り当てれば簡単にできる。

当たり判定が単純である場合（実際のところ、今の場合がそれにあたる）、この設計でも問題はない。これで、「当たり判定に関する設計」は、ひとまず、終了である。

本小論は単純な設計を否定するものではない。ただし、初心者には「単純な設計は、単純なプログラムにしか有効ではない」と教えるべきである。

上述の設計では、Gameが「弾丸の発射」と「当たり判定」を受け持つことになるが、それらに関連する機能を増やそうとすると、すべてがGameに書き込まれ、したがってGameのコードのみが複雑化していく可能性が高いと考えられる。

たとえば、戦闘機がいろいろなモード（高速モード、連射モード、無敵モードなど）になるとし、そのモードに応じた「弾丸発射」をするようにしても、そのコードを書き込むのは、FighterではなくGameになる可能性が高い。弾丸にいろいろな種類をつけて、弾丸の飛び方や戦闘機へのダメージを変えたい場合でも、それらのコードもGameに書き込むことになる可能性が高い。もちろん、これらの場合でも、意識してコードを整理し分散させて書くことは可能である。しかし、それがうまくいかないことも多いのである。

図３．１の「改良版」としては、たとえば、次のような設計も考えられる。

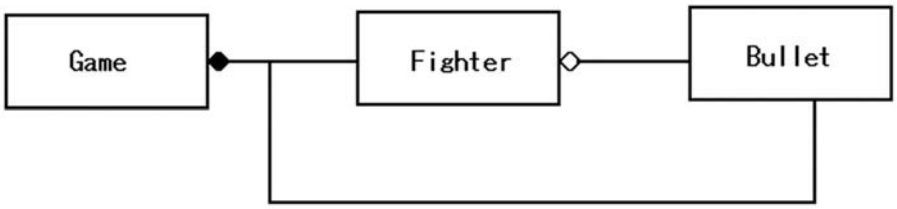

図４．１　ゲームがすべてを持っていて、戦闘機は弾丸のことを知っている

ここで、KObject等の継承関係の部分は省略した。以下でも同様である。

白抜きのひし形付きの線は、「相手の情報（ポインタなど）を持つ」という意味である。Fighterが「弾丸の情報」を持つので、Fighterに弾丸を発射する関数を持たせることができる。

しかしまた、もっと別の設計を考えることもできる。仕様１の「自分の弾丸は自分には当たらない」をもっともシンプルに実現するにはどうすればよいだろうか。「自分の弾丸」を自然に考えれば、「戦闘機が弾丸を持つ」ということになるだろう。それは、次のような構造を考えることになる。

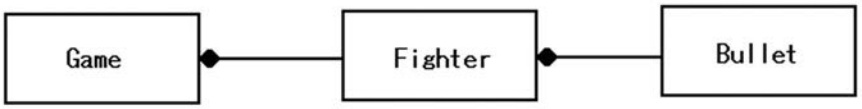

図４．２　ゲームが戦闘機を持ち、戦闘機が弾丸を持つ

現実の戦闘機は弾丸を積んでいるのであるから、これは、現実に近い形でもあり、プログラムを単純にする可能性を持っている。このようにすれば、たとえば、Fighterに弾丸発射関数（shootとする）を

```
void Fighter::shoot() {
    if(bn_ >= 100) return;
    bs_[bn_].setXY(getX(), getY());
    bs_[bn_].setVXVY(10* dcos(getDeg()), 10*dsin(getDeg()));
    bs_[bn_].setValid(true);
    ++bn_;
}
```

などと書いて、持たせることができるだろう。ここで、bs_はFighterが持つ弾丸の配列、bn_はそのFighterが次に発射する弾丸の番号を表すこととする。つまり、bs_、bn_は、Fighterのメンバ変数としたのである。

すると、GameのonJoyStickは次のようになるだろう。

```
void Game::onJoyStick(int i) {
    fs_[i].addX(jsX());
    fs_[i].addY(jsY());
    if(onButton(0)) fs_[i].shoot();
};
```

もちろん、複雑さは、プログラム全体からは消えていない。単に、GameのonJoyStickのコードの一部がFighterに移動しただけである。しかし、こうすることによって、次のようなメリットがある。

1．onJoyStickのコードがわかりやすくなった。
（「if(onButton(0)) fs_[i].shoot();」は、「第０ボタンが押されたら戦闘機が弾丸を発射する」と読める。）
2．「弾丸発射」のコードが独立したため、その部分の変更が簡単になった。
（たとえば、モードによって、発射する弾丸を２つにしたり、散乱させたり、ということが、Fighterの変更だけで行える。）

今は、２人対戦ゲームを考えているが、３人以上の対戦ゲームに変更する場合も、Fighterオブジェクトを増やせば弾丸も付随して増えるので、簡単である。

つまり、「戦闘機が弾丸を持つ」とすることで、コードが整理され、わかりやすくなり、その結果、変更しやすくなったのである。

ただし、この場合でも、「誰が（何が）当たり判定を行うか」に、３つの可能性がある。それは、

１．弾丸
２．戦闘機
３．ゲーム

のどれか、である。

弾丸自身が当たり判定をするとは、つまり、「戦闘機に当たったか」を判定する関数をBulletにつけるということである。戦闘機が判断する場合は、「弾丸が当たったか」を判定する関数をFighterにつけることになる。

前者の場合、弾丸が戦闘機の位置を知らなければならない。すると、「誰が（何が）弾丸に戦闘機の位置を教えるか」が問題になる。もちろん、後者の場合は、「誰が（何が）戦闘機に弾丸の位置を教えるか」が問題になる。

１つには、はじめから、戦闘機の「情報」を弾丸に教えておく（あるいは、弾丸の「情報」を戦闘機に教えておく）ということも考えられる。たとえば、ポインタを渡しておくのである。これは、次のような構造を考えるということである。

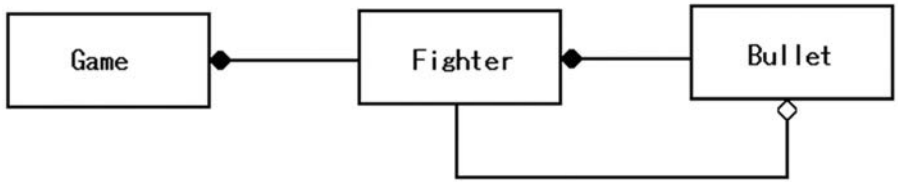

図４．３　弾丸が戦闘機の情報を持つ

このようにすれば、Bulletに当たり判定の関数を書くことは、容易である。それは、たとえば、次のようになるだろう。

```
bool Bullet::isHit() const {
    return intersect(*ef_);
}
```

ここで、ef_を「敵の戦闘機のポインタ」とした。intersectは、KObjectの関数で、他のKObjectと交差するときにtrueを戻す関数である。

この関数をどこから呼び出すかも問題になるが、「当たり判定は完全にBulletの仕事」と考えるなら、一定時間ごとにBulletの関数内から呼び出すことも可能である。

この設計は、弾丸が当たった場合の処理がBulletのコードだけになるので、いろいろな種類の弾丸を考える場合には魅力的である。他の部分を一切変えることなく、弾丸のコードだけで弾丸の効果を変更することも可能だからである。
しかし、「100発の弾丸があれば、そのすべてが敵の戦闘機のポインタを持つ」ということになるので、メモリ的にはあまり効率がよいとは言えない。また、戦闘機が後から登場するような場合、改めて弾丸に新しい戦闘機のデータを与えなければならなくなる。
もちろん、これらのデメリットが大きな問題にならなければ、これらの設計でよいわけである。（戦闘機が当たり判定をする場合も同様に考えることができる。）

ただし、もう１つの可能性があった。ゲームが当たり判定を行う場合である。これは、「現実をシミュレートする」、つまり、「現実をヒントにする」ということには、沿っているかと思う。ただし、「ゲームがどうやって弾丸の情報を知るか」が問題になる。
そこで、たとえば、GameがFighterの持つBulletの情報（ポインタ）を持つという設計も考えられる。

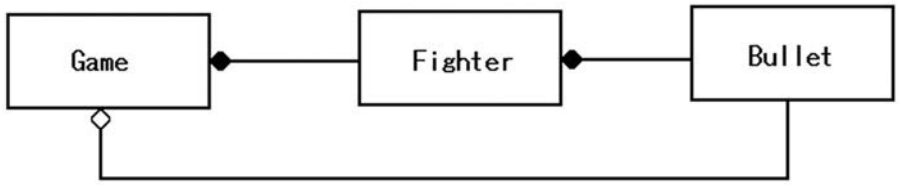

図４．４　ゲームがすべてを持っているわけではないがすべてを知っている

Gameは弾丸の配列の先頭のアドレスのみを持っていればよいので、これなら、メモリ的な効率もよい。
このような設計で、当たり判定をし、その後の処理を行うGameの関数は、単純に書くと、次のようになる。

```
void Game::hitProcess() {
    for(int i = 0; i < 100; ++i) {
        if(bs_[0][i].intersect(fs_[1])) {
            戦闘機 0 の弾丸が戦闘機 1 に当たったときの処理
        }
        if((bs_ [1].intersect(fs_[0])) {
            戦闘機 1 の弾丸が戦闘機 0 に当たったときの処理
        }
    }
}
```

ここで、bs_[0]、bs_[1]は、戦闘機 0 、 1 の弾丸の配列へのポインタとしているのであ

る。戦闘機0と1に関しての処理は、コードをわかりやすくするために分けて書いた。
あるいは、Fighterに自分の持つBulletの先頭のアドレスを戻す関数（getBulsとする）をつければ、Gameはそれを利用できる。その場合、構造は図4．2のままでよいことになる。
こちらの方が、構造はすっきりしているが、コードが今後複雑になっていくかもしれない。
いずれにしても、「当たり判定をGameがする設計」では、弾丸の種類を増やしたり、当たりの条件を単なる交差のみでなくする（特別な条件のときのみ当たるようにする）とした場合、Gameのコードが大きくなる傾向がある。その場合でも、コードを整理し、Game以外のオブジェクトにうまく分配できればよいが、それがうまくいかなければ、他の設計を考えることになるだろう。

ここで示したかったことは、どの設計がよいかではなく、いろいろな設計が可能だということである。プログラムを大きくしていくときに、設計の視点がなければ、いずれ破綻してしまうのである。
そして、本小論で紹介したゲームプログラムは、設計をオブジェクト指向的に考える良い教材ではないだろうか。

5．結論

初心者にオブジェクト指向、特に、その設計を教える方法を考えてきた。そのために、利用したのは、

1．独自ライブラリによるフレームワーク
2．ゲーム製作の具体例

である。
オブジェクト指向を利用したライブラリを利用することで、長いプログラムが簡単になる。その事実が「オブジェクト指向の利点」を明確にし、習得意欲を高めることになると考える。
また、本小論で示したのは、ゲーム製作の一部に過ぎないが、これだけでもいろいろな選択があることを示すことができたと思う。しかも、アルゴリズムの選択などといったものより、具体的に考えることができる点が、初心者に向いていると考える。

謝辞
本研究は、2007年度、2008年度、2009年度の小林ゼミにおける「ゲーム製作研究」に基づいている。ライブラリを利用しながら、独自のゲームを製作し、そのライブラリの不備を指摘してくれた学生諸君に感謝したい。
また、いつもご指導・ご鞭撻くださる静岡産業大学情報学部の諸先生方に感謝致します。

付録　独自ライブラリの概要

```
// テクスチャ（画像）、直接使うことはあまりない
class KTexture { /* 省略 */ };

// ゲーム内のオブジェクト
class KObject
{
    static KGameFrame* pGame_; // ゲームオブジェクトへのポインタ
    const KTexture* pKTex_;    // テクスチャへのポインタ
    float x_, y_;              // 位置
    float deg_;                // 水平線に対する角度
    RECT rc_;                  // テクスチャ内の表示の範囲
    int thru_;                 // 透過度
    float scale_;              // 表示倍率
    int ref_;                  // 反転するかどうか
    bool val_;                 // 有効・無効
public:
    KObject();
    virtual ～KObject() {}
    /* setter getter等は省略 */
    bool intersect(const KObject& o) const; // 他のオブジェクトと重なっていればtrue
    virtual void show();                    // 表示
};

// 一定速度で動き続ける物体
class KVelObject : public KObject
{
    float vx_, vy_;            // 速度
public:
    KVelObject() : KObject(), vx_(0), vy_(0) {}
    void setVXY(float vx, float vy) {vx_=vx;vy_=vy;}
    virtual void show();       // 表示（自動的にvx_、vy_で動く）
};

// アニメーション ct_がect_になるまでアニメーション
class KAnime : public KObject
{
    /* 省略 */
public:
```

```
    KAnime();
    void setTex(const std::string& n, int e);  // 画像とコマ数をセット
    void setPlayRatio(int r);                  // 再生速度がr倍になる
    void setLoop(bool b) { loop_=b; }          // アニメをループにする
    void start(float x, float y);              // スタート
    void stop();                               // ストップ
    bool animeIsOver() const;                  // 終了していればtrueを戻す
    void show();                               // 表示
};

// 効果音（音楽でも同様のクラスあり）
class KSound
{
    /* 省略 */
public:
    KSound();
    ～KSound();
    // ロード　nは音源ファイル名
    bool load(LPDIRECTSOUND8 pDSound, const std::string&n);
    void unload();                             // アンロード
    void start();                              // スタート
    void stop();                               // ストップ
};

// ゲームの基本クラス
class KGameFrame
{
    /* 省略 */
public:
    KGameFrame(int w, int h);
    virtual ~KGameFrame();
    // ウィンドウにメッセージを送る
    void SendMessage(UINT m);
    // タイトルバーのテキストをセット
    void SetWindowText(const std::string& n);
    LPD3DXFONT getFont(int size);              // フォントの生成
    // ゲーム画面に文字を表示
    void text(LPD3DXFONT p, int x, int y, const std::string& n, int r, int g, int b);
    void setBG(const std::string& n);          // 背景のセット　nは画像ファイル名
    // 音の初期化（sndを初期化）
```

```
    void initSnd(KSound& snd, const std::string&n, int e);
    // 音楽の初期化（mscを初期化）
    void initMsc(KMusic& msc, const std::string&n);
    void moveCFrame(int x, int y);     // カメラフレームの移動
    // ゲームコントローラのd番ボタンが押されているか（rがtrueなら連射モード）
    bool onButton(int d, bool r=false);
    // ゲームコントローラの十字キーの値
    int jsX() const;
    int jsY() const;
    virtual void init() {}                  // 初期化
    virtual void onJoyStick(int i) {}    // ゲームコントローラによるゲームの進行
    virtual void onKeyDown(WPARAM w) {}   // キーボードによるゲームの進行
    virtual void showText() {}                  // ゲームにおける文字の表示
    virtual void show() {}                      // ゲームにおける画像の表示
};
```

文献

［1］ B.Stroustrup　『プログラミング言語C++』　アジソンウェスレイパブリッシャーズジャパン

［2］ 小林健一郎『初心者を対象としたC++プログラミング教育について』静岡産業大学情報学部紀要第１号

［3］ 小林健一郎『講義における実用サイトの構築』静岡産業大学情報学部紀要第11号

［4］ http://msdn.microsoft.com/ja-jp/directx/default.aspx