

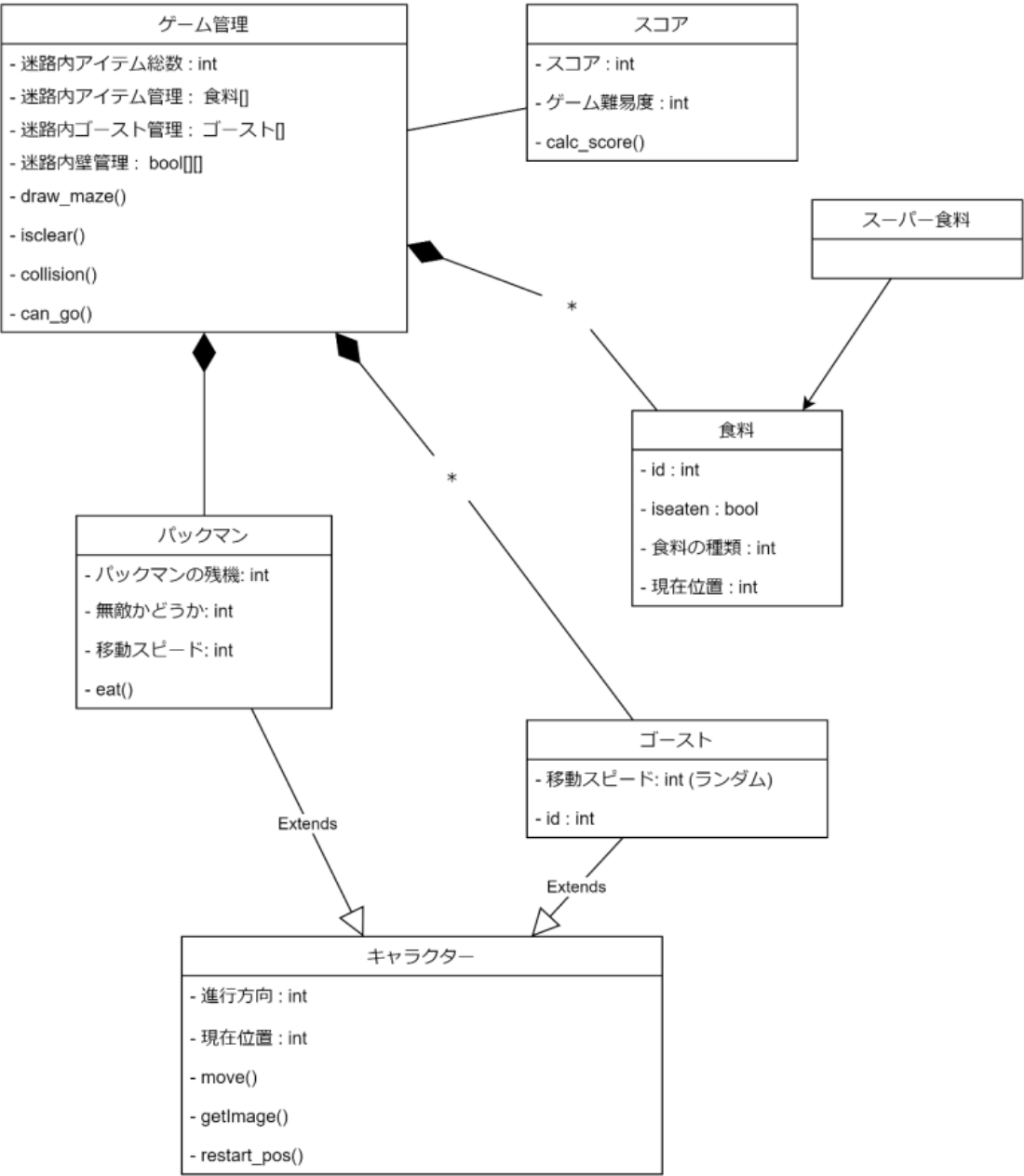
# ソフトウェア品質と設計

## 最終課題

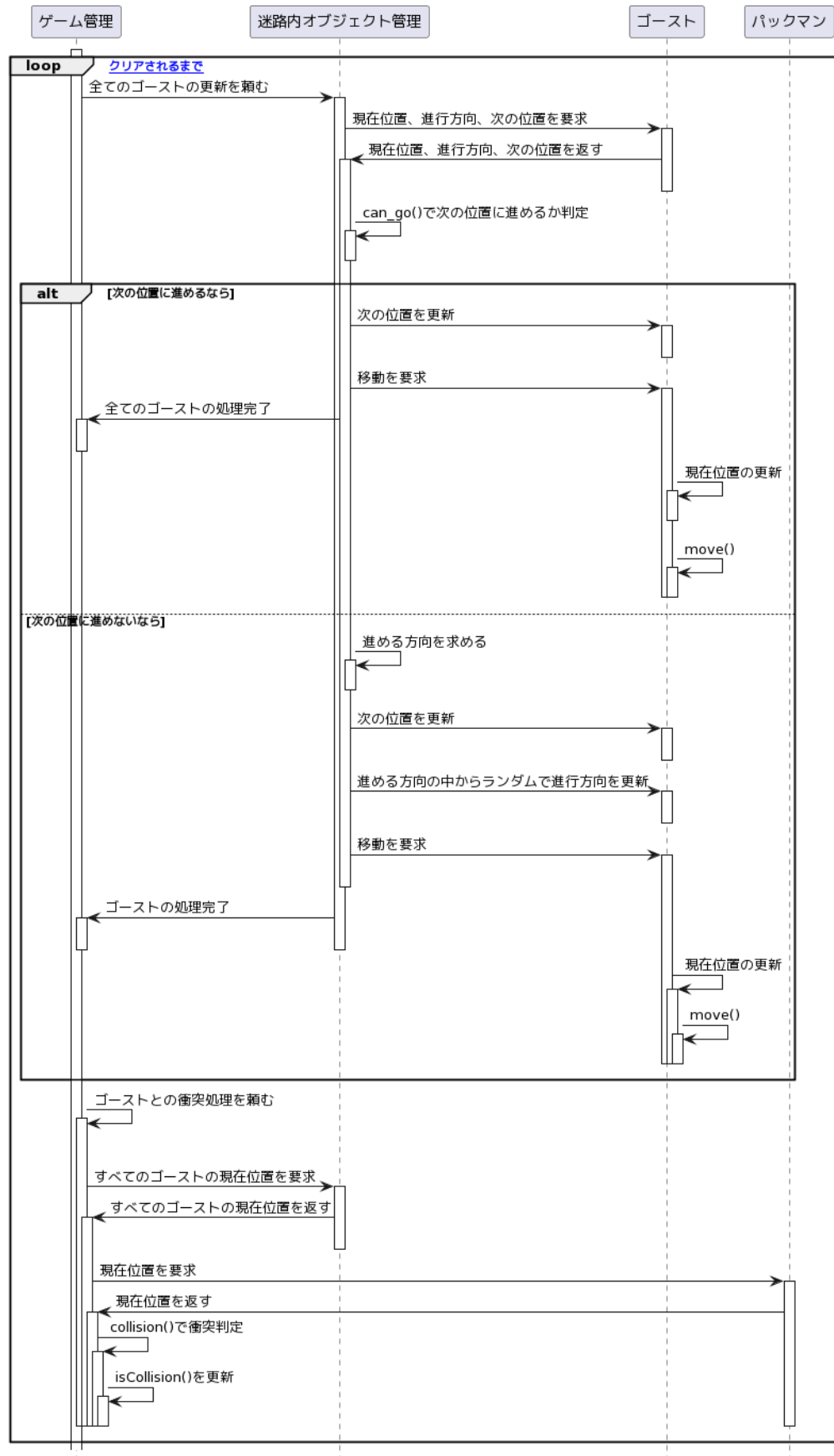
20B11261 小島悠介

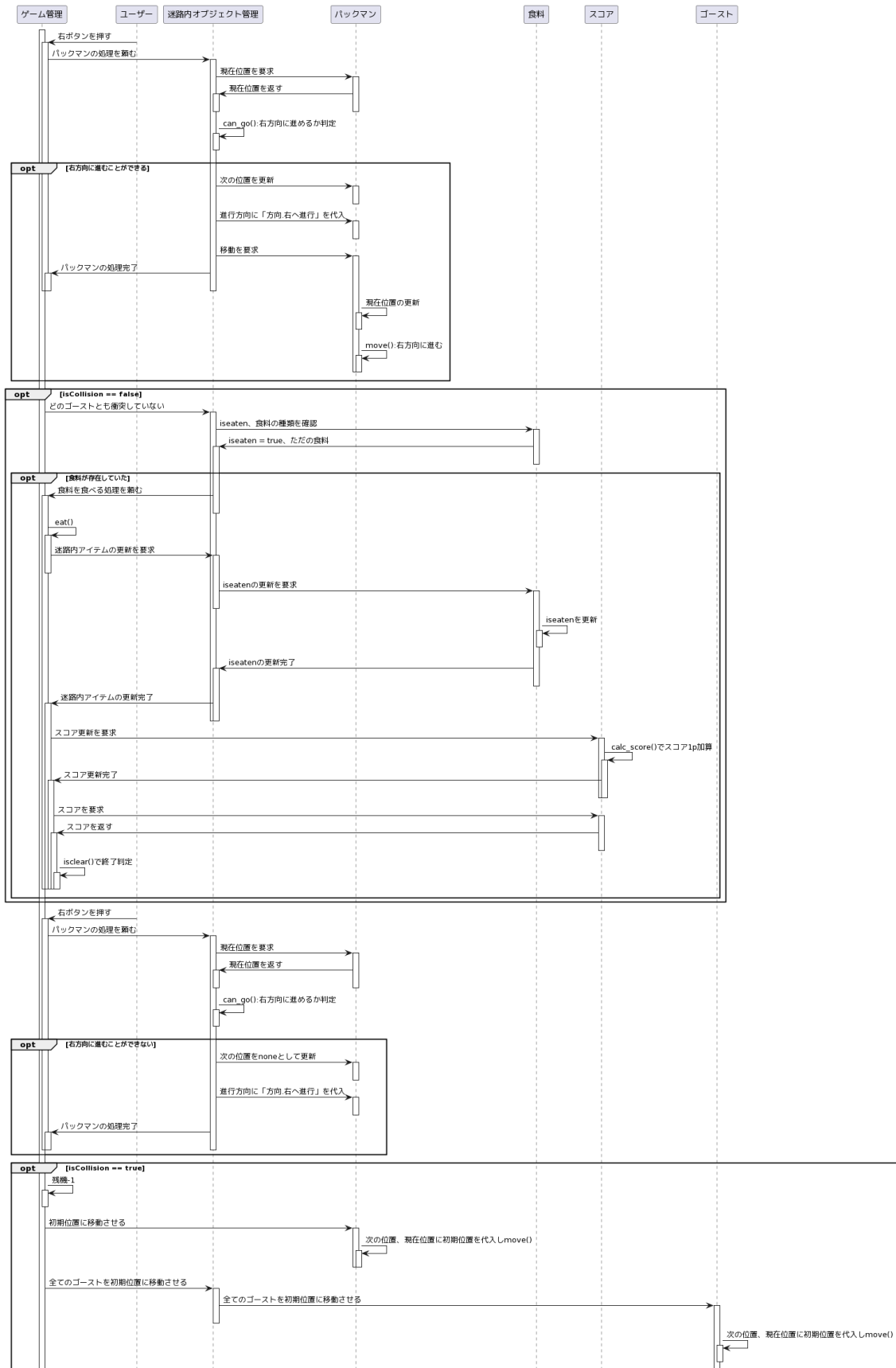
0. 中間レポートにおける設計

クラス図



シーケンス図

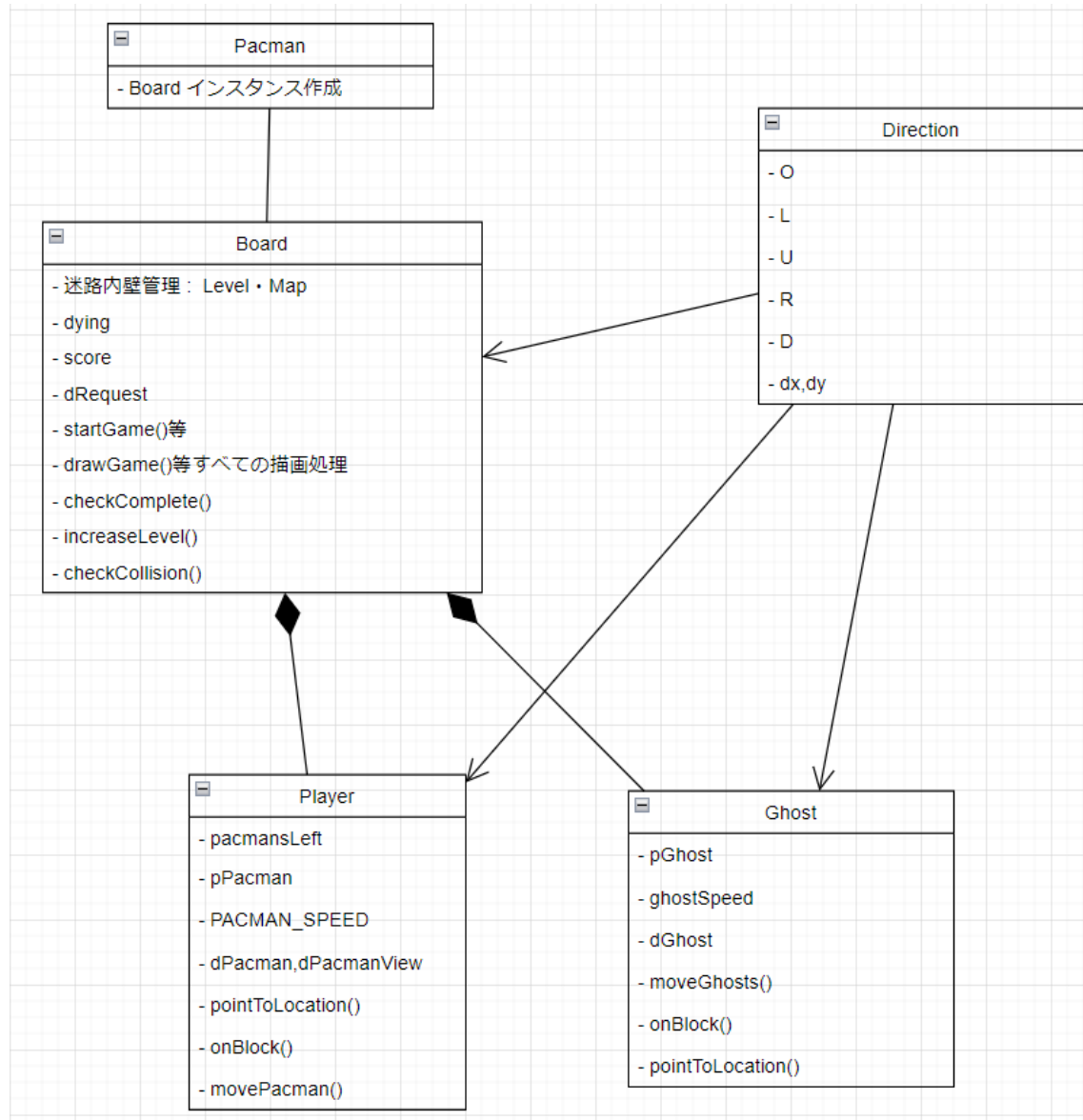




## 1. 設計 1 とその実装

(ex1 ブランチに設計 1 の実装コードあります。)

設計 1 でのクラス図



## 各クラスの説明

- ・ Pacman クラス：

元の Pacman.java のコードから何も変えていない。

- ・ Player クラス：

パックマンについてのクラスであり、中間レポートにおける「パックマン」と「キャラクター」に相当するフィールドやメソッド（主に残機、位置、速さ、向き、move()）がある。

- ・ Ghost クラス：

ゴーストについてのクラスであり、中間レポートにおける「ゴースト」と「キャラクター」に相当するフィールドやメソッド（主に位置、速さ、向き、move()）がある。

- ・ Board クラス：

元の Board.java のコードから上記 Player クラスと Ghost クラスに相当する部分を抜いたものである。中間レポートにある通り、ゲームマネージャーの役割、迷路の管理に加え、元の Board.java のコードに入っていたキー入力、描画処理をすべて行っている。

- ・ Direction クラス：

元の Direction.java のコードから何も変えていない。

動作例ですがもともとのプログラムから動作自体は変わっていないので載せていません。

## 2. 設計 1 の品質

問題点とその対策を箇条書きにしてまとめた。

- ・実際にゲームが動かすためには中間レポートの時には把握できなかった処理が必要になってくる。例えば描画処理は中間レポートでは全く考えていなかったため、設計 1 ではすべての描画処理を Board クラスで行っている。明らかに Board クラスはゲームマネージャーの役割も背負っているため、「巨大なクラス」となっている。

⇒設計 2 ではパックマンの描画は player クラスで、ゴーストの描画は Ghost クラスで行うことにした。またゲームマネージャーに関する責務は新たに GameManager クラスで行うこととした。

- ・元のコードからクラスを分けた方がいいもの、public static のオンパレードになってしまった。Player クラスを例に挙げてみるとこんな感じである。

```
public class Player {  
    private static final int PACMAN_SPEED = 6;  
    public static Point pPacman;  
    public static Direction dPacman,dPacmanView;  
    public static int pacmansLeft;  
    public static void continueLevelPacman(){}  
    public static boolean onBlock(Point p) {}  
    public static int pointToLocation(Point p) {}  
    public static void movePacman() {}  
}
```

⇒設計 2 では Player クラスのインスタンスを Board クラスで作成することで、Player クラスのフィールドを private にした。

- ・設計 1 の Board.java のコードでは LEVEL という変数にマップ内の配置情報が入ってるが、初見でこれが何を意味しているのかが分かりにくい。

⇒設計 2 では LEVEL という変数名を ARRANGEMENTS にした（複数形なのは様々なレベルに対応して複数の ARRANGEMENT をもつため）。また level というレベル管理に必要な変数を新たに作成し、increaseLevel() メソッドなどでその変数をインクリメントするようにしている。

・設計 1 の Board.initLevel()メソッドでやっていることは LEVEL から map を作り、continueLevel メソッドを読んでいる。「初期化する」という名前のメソッドの中に「初期化」+「継続」という責務が入っているため、誤解を生んでしまう。そもそも continueLevel()メソッドはレベルに関するメソッドというよりは、death()メソッドの中で死んだあと or initLevel()メソッドの中で一番初めにレベル配列から MAP を作ったあとで使用していて、意味的にはゲームを初めからスタートさせるメソッドになっている。

⇒設計 2 では continueLevel()の名前を continueGame()と変更し、initLevel()の名前を initMap()と変更した。また、initMap()のなかで continueGame()を呼ぶのではなく、以下のように initGame()のなかで initMap()を呼んでから continueGame()を呼ぶことにした。

```
private void initGame() {
    player.pacmansLeft = 3;
    score = 0;
    ghost.numGhosts = 0;
    ghost.ghostSpeedRank = 3;
    board.initMap();
    continueGame();
}
private void continueGame() {
    player.continueGame();
    dying = false;
    ghost.continueGame();
}
```

・Player クラスと Ghost クラスに分割したのはよいが、onBlock()と pointToLocation()というどちらにも共通するメソッドがある。

⇒設計 2 ではあらたに抽象クラスである Character クラスを用意し、これは Player クラスと Ghost クラスの親クラスとして定義した。これを継承することであらたに特殊なゴーストやパックマンを作成する際のコード記述量が減るというメリットもある。

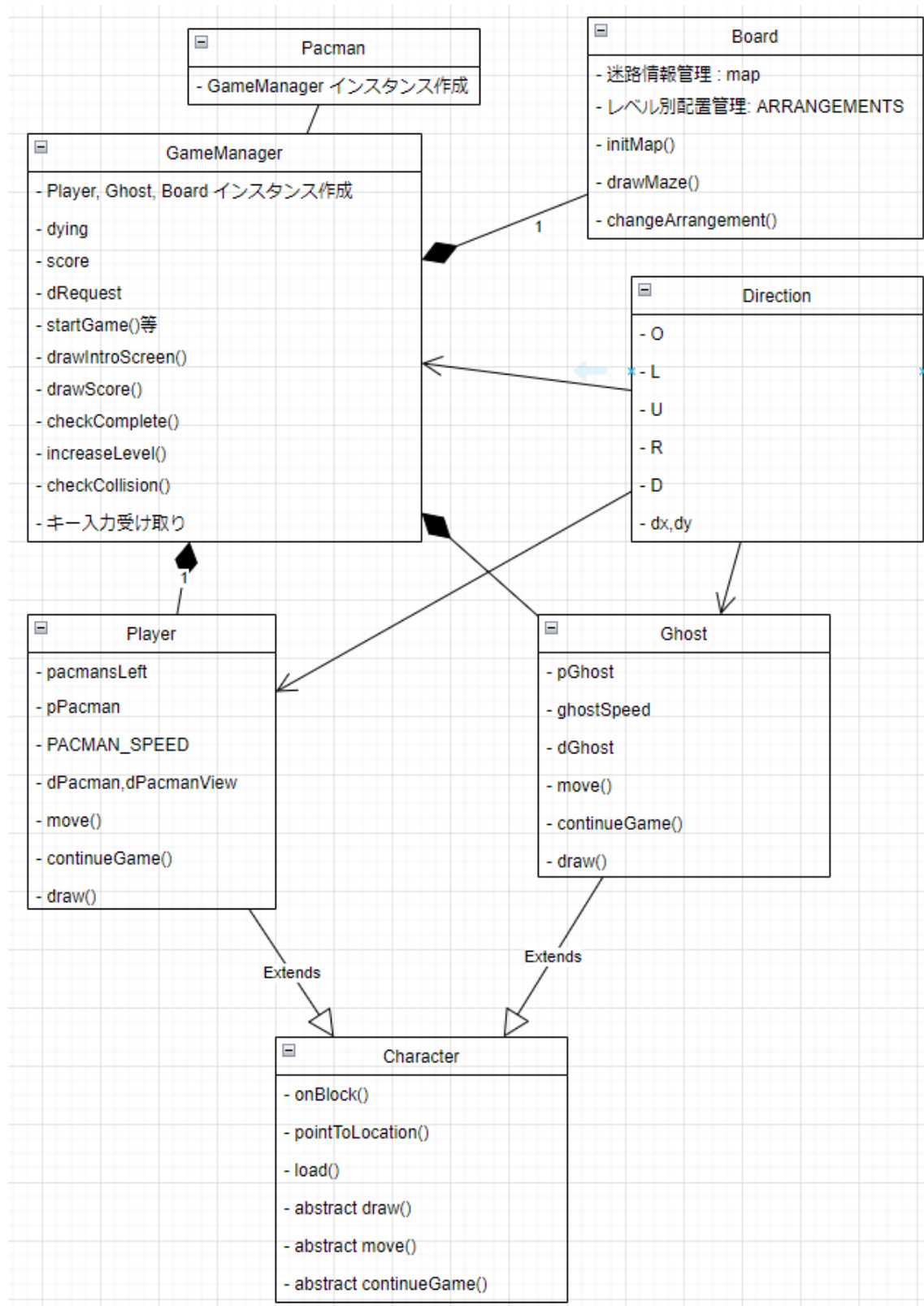


### 3. 設計 2 とその実装

どのフィールドがどのクラスにあるのかパッとわかりやすいように意味でクラス分けをした。個人的にスコアはゲームマネージャーでやるような気がしたので、今回は Score クラスというものを作成していない。ただ今後スコア加算の方法が複雑になったり、スコアの色をスコアによって変えたりして GameManager クラスのコード量が多くなることが考えられるので、以下のようにクラスを分けておくのもよかったと思う。

```
J Score.java U ×
src > main > java > pacman > J Score.java > ...
1  package pacman;
2
3  import java.awt.*;
4  import javax.swing.JPanel;
5  public class Score extends JPanel {
6      private static final Color SCORE_COLOR = new Color(r: 96, g: 128, b: 255);
7      private static final Font SCORE_FONT = new Font(name: "Helvetica", Font.BOLD, size: 14);
8
9      int score = 0;
10     /**
11      * Draws the score and the pacmans left.
12      */
13     void drawScore(Graphics2D g, Player player) {
14         g.setFont(SCORE_FONT);
15         g.setColor(SCORE_COLOR);
16         g.drawString("Score: " + score, Board.BOARD_SIZE.width / 2 + 96, Board.BOARD_SIZE.height + 16);
17         for (int i = 0; i < player.pacmansLeft; i++) {
18             g.drawImage(Player.PACMAN_IMAGE_L[3], i * 28 + 8, Board.BOARD_SIZE.height + 1, this);
19         }
20     }
21
22 }
23
```

設計 2 のクラス図



## 各クラスの説明

- Pacman クラス：

設計 1 から何も変えていない。

- Player クラス：

設計 1 のフィールドやメソッドに加え、draw メソッドを持つようにした。これによりほぼすべてのパックマンに対する操作や動作がこのクラスでできるようにした。ほぼすべてとしたのは、実装はできなかったが本当はパックマンに対するキー入力 Player クラスにいられたかったからだ。そうすると dRequest も Player クラスに入ってすべてのパックマンに対する操作や動作が Player クラスで可能となる。

- Ghost クラス：

設計 1 のフィールドやメソッドに加え、draw メソッド、ゴーストの数を増やしたりする increaseLevel メソッドを持つようにした。

- Character クラス：

新しく作成した抽象クラス。パックマンとゴーストに共通する処理や、共通して必要なメソッドを定義した。

```
abstract class Character extends JPanel{
    boolean onBlock(Point p) {
        return p.x % Board.BLOCK_SIZE == 0 && p.y % Board.BLOCK_SIZE == 0;
    }
    int pointToLocation(Point p) {
        return (p.y / Board.BLOCK_SIZE) * Board.MAP_SIZE.width + (p.x / Board.BLOCK_SIZE);
    }
    static Image load(String filename) {
        URL url = GameManager.class.getClassLoader().getResource("images/" + filename);
        assert url != null;
        return new ImageIcon(url).getImage();
    }

    abstract void draw(Graphics2D g);
    abstract void move();
    abstract void continueGame();
}
```

これによりあらたに特別なゴーストや特別なパックマンを作るときも作りやすくなったし、Player クラスと Ghost クラスの記述量を少し減らすことができる。

- Direction クラス：

設計 1 から何も変えていない。

・ GameManager クラス：

設計 1 では Board クラスの責務が多すぎた。ゲームマネージャーに関する役割を抽出したクラスである。player などのインスタンスを作り、startGame()や initGame()などのメソッドを持つ。パックマンの描画処理などある特定のクラスに意味的に関連付けられる描画処理以外の描画処理はここで行っている。具体的には drawIntroScreen()と drawScore()メソッドだ。また先ほどの考察からスコアの加算も今はここで行うように実装した。キー入力受付もここで行う。

・ Board クラス：

盤面に関する責務を任せている。まずは元のコードで LEVEL だったマップの壁の位置の配列は ARRANGEMENTS として持ち、それから map をつくることもこのクラスで行っている。盤面に関する描画、つまり drawMaze()も担当している。レベルに応じて盤面を変える処理 (changeArrangement()) も行っている。

以上が設計 2 におけるクラスである。

また、実装する際に特に気を付けたことは以下の二点である。

- ・一つのメソッドで行うことは意味的に一つにする
- ・よっぽどのことがない限りクラス間で共通のフィールドを共有しない

ここでは二つ目について議論する。

まず「よっぽどのことがない限り」の「よっぽどのこと」とは map 配列のことである。それはさておき、仮に共通のフィールドをクラス間で共有すると、例えば以下に挙げる increaseLevel()メソッドで危ないことが起きる。

```
private void increaseLevel() {  
    level +=1;  
    ghost.increaseLevel();  
    board.changeArrangement();  
    board.initMap();  
    continueGame();  
}
```

もしここで Board クラスの changeArrangement()メソッドでも level という変数を用いてマップ配置を変更していると危ない。バグが起きても気づかない可能性がある。なぜなら、もし level をインクリメントするのが changeArrangement()メソッドを呼ぶよりも後になってしまうと、レベルは上がったのにマップ配置は変わらないということが起きてしまうからだ。今回はこれを防ぐため、以下のように Board クラスでは level とは別で numOfArr という

う変数を作り、それをインクリメントすることでARRANGEMENTを変更するようにした。

```
void changeArrangement() {  
    if(numOfArr< Max_Arr){  
        numOfArr += 1;  
    }  
}
```

最後に追加実装に関する話題を扱う。

- ・無敵状態

無敵状態の管理は Player クラスで行い、GameManager クラスで衝突判定をしないようにすればよいだろう。

- ・スーパーアイテムの追加

アイテムの追加は map の右から六番目以降にビットを立てていく。

map 配列の実装ではすでに以下のような仕組みになっている。

- 一番右のビットが立っているなら左に行けない
- 右から二番目のビットが立っているなら上に行けない
- 右から三番目のビットが立っているなら右に行けない
- 右から四番目のビットが立っているなら下に行けない
- 右から五番目のビットが立っているならそこに食料がある（もし食料をパックマンが食べたら立たなくなる）

となっているから、食料やアイテムの追加を行う場合は右から六番目以降のビットを立てていけばよいことがわかる。今回は試しに super dot の実装だけ行った。

Super dot の挙動は、それを食べるとスコア+10 にするようにした。以下に実装例を挙げる（super dot を作った際に変更したすべてのファイルを記載しているわけではない）。

Board.java- initMap

```
if (ARRANGEMENTS[numOfArr][i] == 2) {  
    c |= 32; // super dot  
}
```


Player -move()

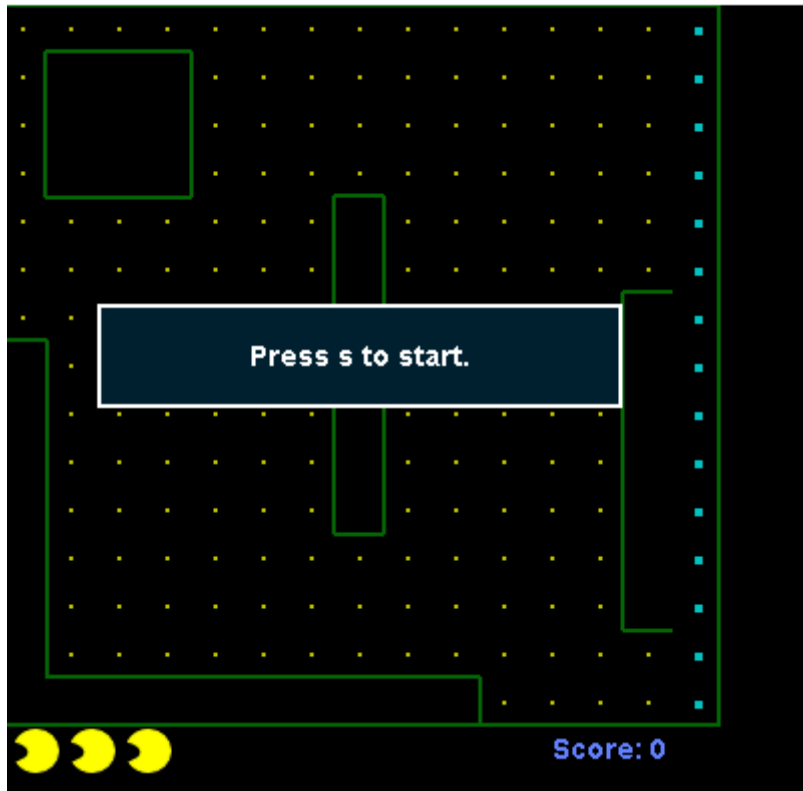
```
if ((1 & 32) != 0) { //右から 6 番目のビットが少なくとも 1 が立ってたら、  
つまりまだ食料を食べていなかったら  
    // eat super dot  
    Board.map[loc] = 1 & 31; //6 桁目だけを 0 に変える
```

```
GameManager.score += 10;
```

```
}
```

また、実行するとわかるが super dot は以下のように大きめの青い点になっている

 Pacman



#### 4 章（感想）

map 配列一つで上下左右進行可否やアイテムの存在を管理する仕組みはとても勉強になりました。またゲーム作りを通してオブジェクト指向の楽しさに触れられてとても楽しかったです。