命名規則 C#の命名規則ではClass名、メソッド名、プロパティ名はPascalCase、変数名はcamelCase、定数CONSTANT_CASEである。JavaともC++とも異なるが、C#の標準であるため、これに従う。

第2章

2.1

Point 《盤面の座標》

- 盤面は横7マス、縦6マスとし、左上を原点とする。
- Pointは構造体とする。
- 座標x,yを外部から書き換えられないように、readonlyにしておく。
- 引数なしコンストラクタは定義しない。

Color 《石の色》

- 定数値ではなく、列挙型とする。
- 色の入れ替えは符号反転ではなく、拡張メソッドで行う。
- 画面への表示もここで定義。
- 「-1倍で色反転」のようなハックは基本すべきじゃない。したかったら–演算子をオーバーロードすべき。

Disc 《石？》

- いらないと思う。
- 石の配置は後述のBoardで管理する。

2.3

Direction列挙型はBoard外にあるべきなので、Utils.csに移動。Connect4では使わないけど。

2.4

Connect4では石を置けるかどうかの判断は簡単。オセロならば例えば以下のようにする。

```
readonly int[] xk = { 0, 0, -1, 1, 1, -1, -1, 1 };
readonly int[] yk = { -1, 1, 0, 0, -1, -1, 1, 1 };
readonly Direction[] dk = {
    Direction.UPPER, Direction.LOWER,
    Direction.LEFT, Direction.RIGHT,
    Direction.UPPER_RIGHT, Direction.UPPER_LEFT,
    Direction.LOWER_LEFT, Direction.LOWER_RIGHT
};
private Direction checkMobility(int px, int py, Stone stone) {
    if (rawBoard[px, py] != Stone.EMPTY) return Direction.NONE;
    int x, y;
    Direction dir = Direction.NONE;

    for (int k = 0; k < 8; k++) {
```

```
            if (rawBoard[px + xk[k], py + yk[k]] == stone.Reverse()) {
                x = px + 2 * xk[k];
                y = py + 2 * yk[k];
                while (rawBoard[x, y] == stone.Reverse()) {
                    x += xk[k];
                    y += yk[k];
                }
                if (rawBoard[x, y] == stone) dir |= dk[k];
            }
        }
        return dir;
    }
```

2.5

- C#のサイズ可変配列は`List<T>`である。なおC#ではテンプレートとは言わず、ジェネリックという。
- connect4では打てる場所を調べるのが低コストなので、`movablePos`は不要かもしれないが、オセロでは重要なので利用しておく。
- ColorStorageも不要。オセロでもわざわざclassを作らず、Dictionary<Stone,int>に保存すれば良い。
- UpdateLogも`List<Point>`で管理できるが、最後尾への追加と削除しかしないので、より低機能の`Stack<Point>`で管理する。Stackによるデータ管理は、*皿を積み重ねるイメージ*。一番最後に置いたデータしか操作できないし、データ追加も一番上に積むだけである。

2.6

- flipDiscsという名前はConnet4には不適切なので、PlaceDiscsに変更。
- 最上段（Y=1）が空なのは確認済み。あとは直下がEmptyじゃなくなるまで、y座標をインクリメントしている。

2.7

- 最大ターン以外で、石が置けないってことはない。
- 石が4つ揃う時、必ず最後に置いた石が関与しているので、その周囲だけ調べれば十分。
- ゲームの終了を調べた瞬間に勝敗もわかるので`IsGameOver`ではなく、`CheckWinner`という関数にして、勝敗を返すことにする。

2.8

- connect4にパスはない

2.9

- connect4にパスはないので、簡単

2.11

- CountDiscは不要。
- C#のGetMovablePosではそもそもコピーは発生しない。変更を阻止するためにListではなくIReadOnlyListを返すようにする。

- updateLogが空なのに、GetUpdateが呼ばれることはないはずなので、if文で分岐せず、エラーで落とす。

2.12

- Connect4ではPointを文字から作る機会はない。
- 動作確認用のコードは`Program.cs`に書いてみた。

第6章?

とりあえず、動かないとプログラミングしにくいので、本の順序を変更し、Playerから実装する。

6.1

人間とAIのプレイヤーを抽象化して、インタフェース`IPlayer`を定義する。`IPlayer`は盤面をもとに、次の一手を決定する`OnTurn`メソッドを持つ。

```
interface IPlayer{
    public Action OnTurn(Board board);
}
```

本のC++では`IPlayer`の代わりに`Player`クラスを使っているが、インタフェースの方が適切である。C#の命名規則に則って`I`を先頭につけている。

本ではPlayerが`void OnTurn(Board board)`メソッドにて、ゲーム終了などの分岐処理に例外（Exception）を使っているが、これは不適切である。例外は予期せぬエラーを扱うものであり、プログラムの制御を変えるものではない。そこで、動作を報告する`Action`列挙型を`Utils.cs`に作り、`OnTurn`の戻り値を`Action`に変更する。単に継続かどうかのbool型でもよかったかもしれない。

IPlayer実装するクラスは`HumanPlayer`と`AIPlayer`である。`HumanPlayer`はコンソールから入力を受け取る。`AIPlayer`は`AI`クラスをのインスタンスを持ち、`AI`の指示に従って手を打つ。もし2つのプログラム同士で戦わせるのなら、別途`IOPlayer`みたいなのを作るべき。

`AI`は抽象クラスとして定義する。動作確認用のAIとして、打てる場所へランダムに打つAI`RandomAI`と、ほとんどランダムだけど、次手で勝てる時だけ勝ちに行くAI`WeakestAI`をまず作成する。次手の勝利を確認する方法は単純である。とりあえず打ち、勝敗を確認し、勝っていればその手を打つ。勝っていなければ、待ったをして、違う手を打ってみる、というのを全ての手で試す方法である。

```
public override void Move(Board board){
    Color myColor = board.CurrentColor;
    var pos = board.GetMovablePos();
    foreach(int x in pos){
        board.Move(x);
        if(board.CheckWinner() == myColor){
            return;
        }
        board.Undo();
    }
```

```
        board.Move(pos[random.Next(pos.Count)]);
    }
```

第4章

本では第3章で、Minmax法などの実装を説明しているが、評価値の計算がされていないため、実際には動かない。そこでここでも順番を入れ替え、まずは評価値の計算を可能にする。評価値を単純にヒントとして、現在の盤面へ表示するようにしてみる。

4.9

評価値を計算する関数Evaluateは仮想関数ではなく、インタフェースで実装する。

4.11

とりあえず完全読み切り用`PerfectEvaluator`と、普通の読み`PatternEvaluator`を作成する。

`PatternEvaluator`では縦横斜めの4つ並びのパターンをもとに評価値を計算する。相手の石が入らず、自分の石と空白マスだけで4つ並んでいる箇所を全て列挙し、それに応じた得点を加算するようにしている。

さらに`PatternEvaluator`の動作確認用に、次の一手まで読むAI`NextMoveAI`を作成。

第3章

ようやく探索アルゴリズムを実装。Evalueatorが先手であることを前提にしていないので、相手番では評価値を反転させる必要があるので注意。

せっかくなので`AI`クラスではevalulateの呼び出し回数と、所要時間を計算することにする。

3.2

使うことはないけど、一番簡単なので動作確認用にMinmax法を実装。5手先も読めば人間には負けない気がする。

3.3

同じく、動作確認用にAlphabeta法を実装。Minmax法と同じ結果になっているかどうかを、確認するためのTestも作成。

3.4

本ではnegamax法と書いてあるが、一般的にはNegaAlpha法と呼ばれる。

3.5

NegaScout法は実装の面倒さの割に、効果が薄いので省略。その代わり次の手の評価値が高い順に並び替えてから、本探索するようにNegaAlpha法を改良する。