**Metody LogData**

IBall: void LogData(bool reverse = false);

ICollisionManager: void LogData();

IPad: void LogData();

Można zrobić osobny interfejs z metodą LogData (SOLID - Interface segregation principle**)**.

Zmieniłbym tez nazwę na LogDiagnostics(), LogDebugInfo() albo coś w tym stylu.

BallInGameState

Dostaje ICollisionManager w konstruktorze, ale dodatkowo w tym konstruktorze tworzy sobie nową instancję CollisionManager?

IBall i IPad mają w metody:

void SetSize(int width, int height);

void GetSize(out int width, out int height);

Jeżeli by przerobić to na property Size (struct) dostępne przez IElement to można te metody usunąć i kod będzie bardziej czytelniejszy.

IElement można dorzucić do interfejsów IBall, IPad itd. zamiast ustawiać je bezpośrednio na obiektach Ball i Pad.

public interface IBall : IElement

Można wtedy usunąć kod wykonujący rzutowanie który występuje w wielu miejscach:

IElement ballElement = ball as IElement;

IElement padElement = pad as IElement;

IBrick

bool IsBeatable();

bool IsVisible();

Property byłoby bardziej odpowiednie od metod.

ICollisionManager

Te metody robią dokładnie to samo.

bool Detect(IBorder first, IPad second);

bool Detect(IBorder first, IBall second);

bool Detect(IPad first, IBall second);

bool Detect(IBrick first, IBall second);

Wystarczyłaby jedna metoda

bool Detect(IElement first, IElement second);

public class Logger

To chyba miał być Singleton, ale w takim razie set nie powinien być publicznie dostępny.

Nie ma też lock na tworzeniu new Logger(). Może to nie problem, bo jest tylko jeden wątek, ale może lepiej byłoby tworzyć instancję w polu klasy static readonly – to daje pewność, że zostanie utworzone tylko jedna instancja.

Interfejs ILoggerBase nie pełni żadnej roli i można go usunąć.

TailManager

public ITail Find(IBall ball)

{

if (tails.ContainsKey(ball))

{

return tails[ball];

}

return null;

}

Zamiast ContainsKey i odwołania po indeksie lepiej stosować TryGetValue – wtedy przeszukujesz kolekcję raz zamiast dwa razy.

public class Game

List<IBrick> bricks = new List<IBrick>();

foreach (var value in bricksHit)

{

bricks.Add(value.Brick);

}

return bricks;

Jedna linijka z wykorzystaniem Linq

return bricksHit.Select(b => b.Brick).ToList();

Podobnie:

private List<IBrick> ParseBlocks(IEnumerable<XElement> blockElements, int rowNumber)

{

var blocks = blockElements.Elements("Brick").Select(d => new Brick(0,0,0)

{

…

}).ToList();

List<IBrick> result = new List<IBrick>();

foreach (var block in blocks)

{

if (block.IsBeatable())

{

BeatableBricksNumber++;

}

result.Add(block);

}

return result;

}

LINQ

var blocks = blockElements.Elements("Brick").Select(d => new Brick(0, 0, 0)

{

…

}).Cast<IBrick>().ToList();

BeatableBricksNumber = blocks.Count(b => b.IsBeatable());

return blocks;

public bool IsBeatable()

{

if (Type != BrickType.Solid)

{

return true;

}

return false;

}

Można uprościć:

public bool IsBeatable()

{

return Type != BrickType.Solid;

}

new Brick(0, 0, 0)

Konstruktor z parametrami nie ma za bardzo sensu, skoro przekazujesz zera, a potem ustawiasz wartości przez properties.

public double CalculateNewDegree(double Degree)

{

double degreeRangeMax = 30;

do

{

double degree = random.Next((int)degreeRangeMax);

Degree += degree - degreeRangeMax / 2;

Degree = Math.Abs(Degree);

}

while (Degree < 10.0 || Degree > 80.0);

return Degree;

}

To jest mało czytelne. Masz zmienne Degree i degree i modyfikujesz wartość Degree która jest parametrem metody.

W ogóle jakiś dziwny ten algorytm. Losujesz wartość 0-30 i zawsze odejmujesz 15.

Ta pętla while tez mi nie pasuje, dałoby się to napisać bez while ustawiając odpowiedni zakres dla ranodm.Next(), tak żeby uzyskać wartość z odpowiedniego zakresu.

BallStateMachine, BallInMenuState

Metody powinny mieć nazwy z dużej litery