Labbdokumentation

MOO projekt

Av: Alexander Skogsmo, Claes Wikman

Namn

Vi har genomgående i programmet ändrat namn till mer deskriptiva och meningsfulla namn från förkortningar och allmänt otydliga namn.

```
string playerGuess = ReadUserGuess();

int numberOfGuesses = 1;
string bullsAndCows = CheckBullsAndCows(target, playerGuess);
```

Klasser

Eftersom programmet som vi fick var väldigt stort och gjorde på tok för mycket (allt), så bröt vi ner programmet i mindre klasser.

Klasser ska vara korta! Precis som funktioner ska vara korta, så ska klasser också vara det. Detta på grund av att det gör det enklare att förstå vad funktion/klassen gör, samt att det blir enklare att felsöka. [2,3]

Enligt kurslitteraturen ska varje klass ha ett enskilt ansvarsområde (*single responsibility principle*), det vill säga att en klass ska enbart göra en sak. [4]

Exempel: Ursprungsprogrammet hade både spellogik och spelardata i samma fil.

Kommentarer

Enligt kurslitteraturförfattaren är varje kommentar ett misslyckande, då man inte har lyckats skriva begriplig kod. [5]

Över tid så blir kommentarer felaktiga eftersom kod inte är beständig, utan ändras. Om den relaterade kommentaren inte ändras med kodändringarna, leder det till att kommentaren, så småningom, blir missvisande. [5]

Exempel: Det fanns en kommentar till en kodrad som skrev värdet av variabeln *goal* till konsolen. Den tog vi bort, bröt ut koden i en egen funktion, och ersatte med ett metodanrop till nämnda funktion.

```
1 reference
public void ShowTarget(string target)
{
    ui.Write("For practice, number is: " + target + "\n");
}
```

Funktioner

Funktioner ska vara små, precis som klasser. Funktioner ska göra en sak, och göra det bra, också det precis som klasser. [2]

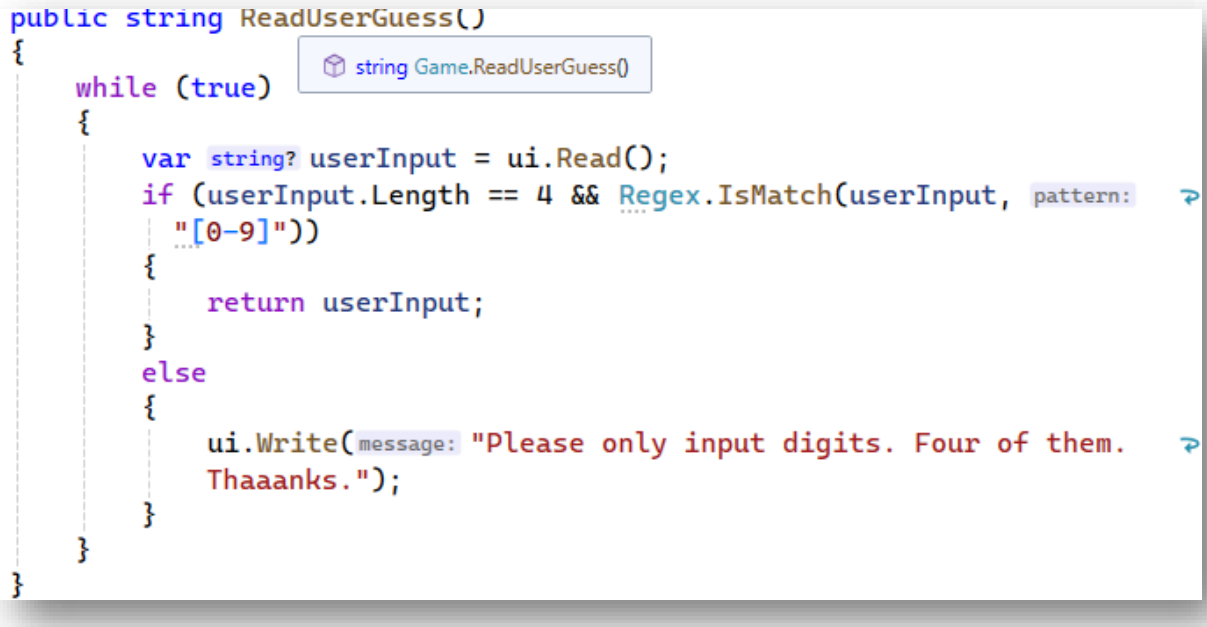
Exempel: Innehållet i spelloopen var väldigt lång. Därför bröt vi ut en delmängd av koden till IO.ShowTopList(). En annan anledning till refaktoreringen var att kodstycket utförde uppgifter som inte berörde spelmekaniken. Vi tyckte att kodstycket var en enhetlig subrutin som berör filhanteringen. Därför fick den flytta över till klassen IO. Den kvarvarande spelloopen och den nya IO-metoden blev både mer läsbara.

```
public void ShowTopList()
{
    List<PlayerData> results = new List<PlayerData> ();
    // TODO: Borde vi använda dependency injection här?
    ConsoleUI ui = new ConsoleUI();
    results = ReadPlayersFromFile(file: "result.txt");
    results.Sort((PlayerData p1, PlayerData p2) => p1.Average().CompareTo
      (p2.Average()));
    ui.Write(message: "Player   games average");
    ViewPlayerResults(results);
}
```

Felhantering

Eftersom felhantering var en punkt i labben så valde vi att implementera det på det ställe där vi tar in input från användaren.

Vi returnerade inte *null*, då det är jobbigt och tidskrävande att kontrollera för *null* vid metodanrop. Allt enligt författaren och dennes användning av **Java** i kurslitteraturens kodexempel. [6]



Rena tester

Robert Martin skriver i kapitlet om rena tester, att BUILD-OPERATE-CHECK är ett mönster man kan använda för att skapa ett lättöverskådligt och välstrukturerat test.

Vad vi har gjort: För att visa på skillnaden mellan ett sämre och ett bättre strukturerat test, har vi valt att göra hälften av våra tester kompletta utifrån BUILD-OPERATE-

CHECK-mönstret, och hälften där operationsdelen saknas. [6,7]

```
[Test]
0 references
public void Generate_Target_ExpectedBehavior()
{
    var string? target = Game.GenerateTarget();

    Assert.That(target.Length, Is.EqualTo(expected: 4));
}

[Test]
0 references
public void BullsAndCows_Success()
{
    var string? target = "abcd";
    string playerGuess = "abcd";
    var string? sut = Game.CheckBullsAndCows(target, playerGuess);

    Assert.That(sut, Is.EquivalentTo(expected: "BBBB,"));
}
```

Designmönster och programstruktur

Vi har använt oss av konstruktorinjektion i Gameklassen.

Vi har delat upp systemet i olika skikt. Användargränssnittet består av ett interface (Iui.cs) och en implementation (ConsoleUI.cs).

Programlogiken ligger i Game.cs, och statistikinsamling/redovisning finns i IO.cs.

```
Iui ui;
IO io;
1 reference
public Game(Iui ui, IO io)
{
    this.ui = ui;
    this.io = io;
}
```

Referenser

[1] Clean Code; Martin, Robert; sid 18

[2] Clean Code; Martin, Robert; sid 34

[3] Clean Code; Martin, Robert; sid 136

[4] Clean Code; Martin, Robert; sid 138

[5] Clean Code; Martin, Robert; sid 54

[5] Clean Code; Martin, Robert; sid 110

[6] https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

[7] Clean Code; Martin, Robert; sid 127