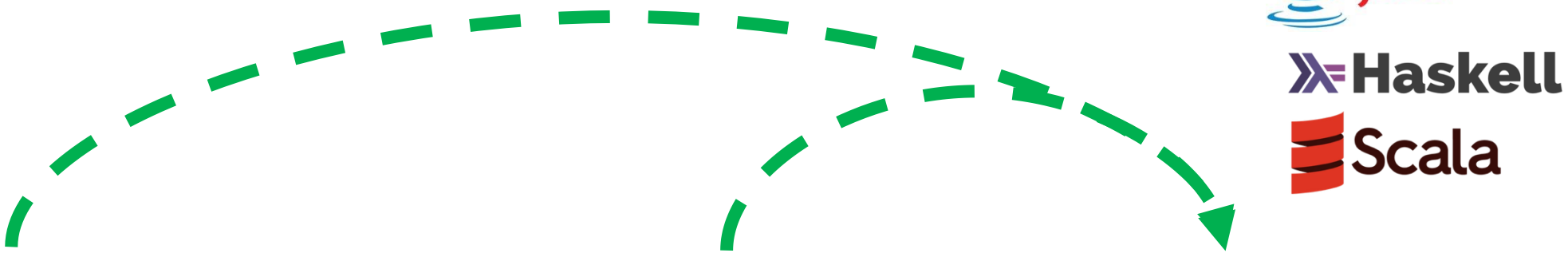
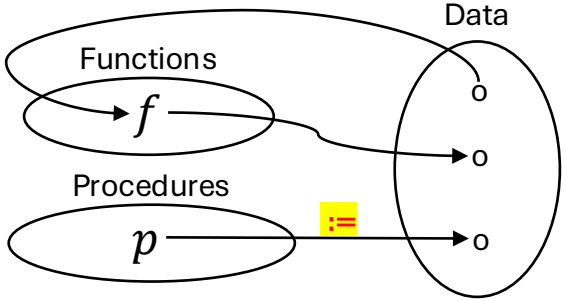
The Bowling Game
From Imperative to Functional Programming

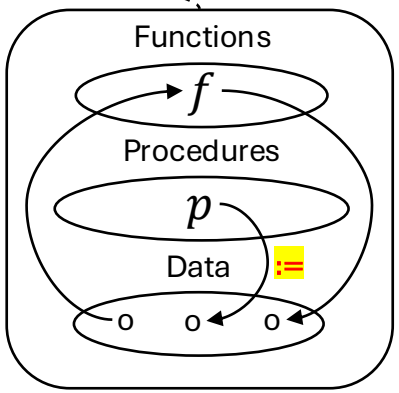
Part 1



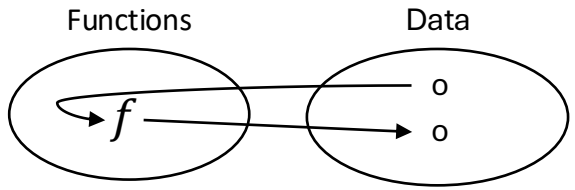
Procedural Programming



Oject Oriented Programming



Functional Programming



slides by  @philip_schwarz FP Illuminated https://fpilluminated.org/

@philip_schwarz

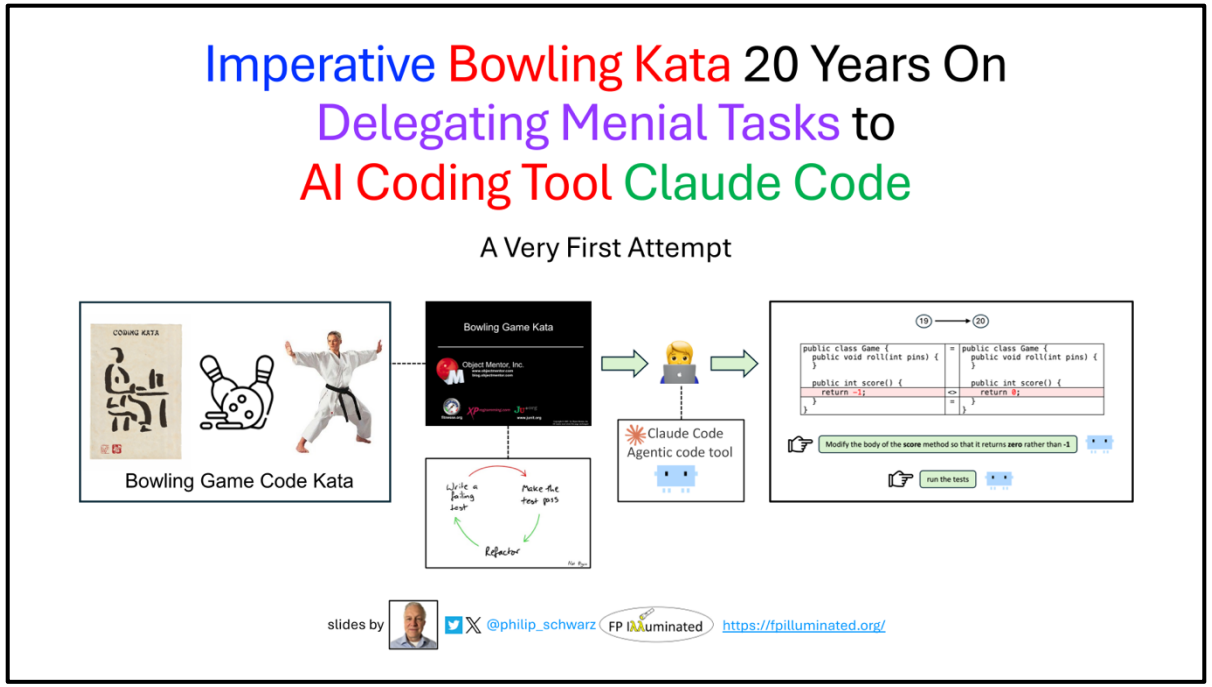
One of the **top five most popular** and **highly recommended programming katas** over the past 20 years has been the **Bowling Game Kata**, in which **TDD** is used to write a program that computes the **score** of a **Ten Pin Bowling Game**.

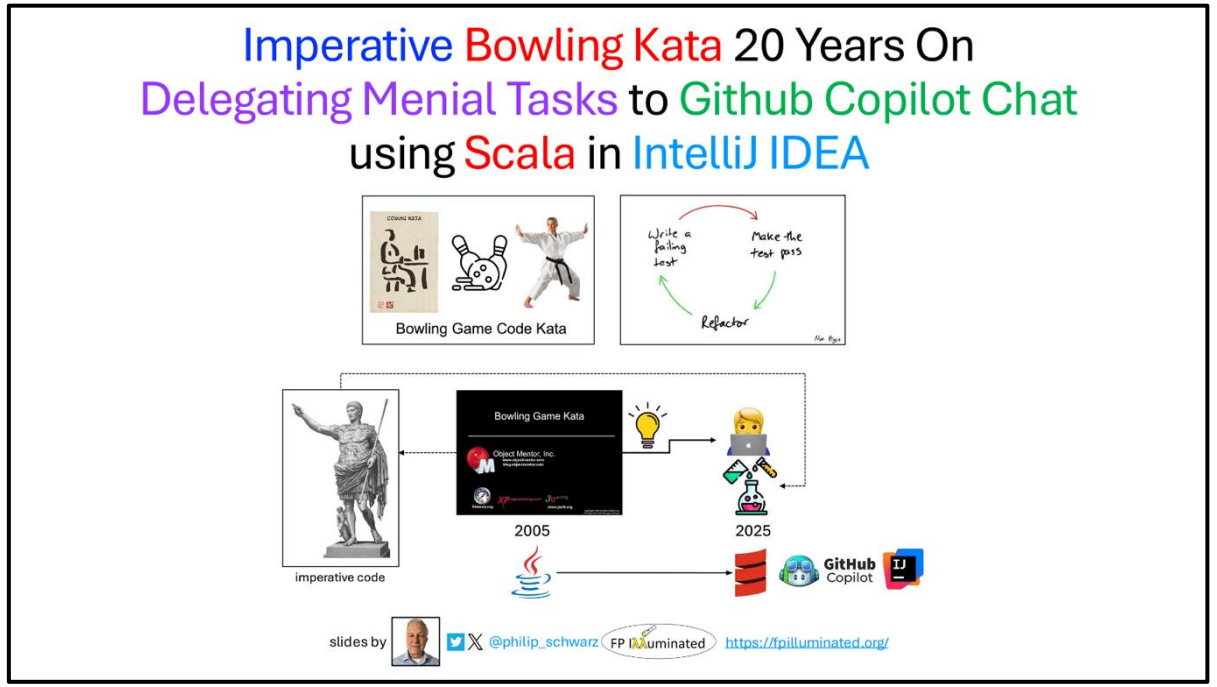
In this deck we are going to explore how such a program may look when coded using **different programming paradigms**.

If you could do with an introduction to (or a refresher of) the **Kata** then see the following web page and deck

- http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
- http://butunclebob.com/files/downloads/Bowling Game Kata.ppt

By the way, if you happen to be interested in the idea of **experimenting** with **using AI to do the kata**, then maybe consider checking out the decks below.





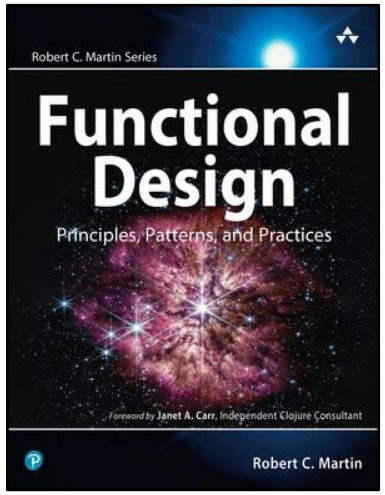
Let's begin by looking at a **Java** version of the **bowling game program** whose **code style** is **procedural**, in the sense that it is only **nominally object-oriented** in that it consists of a **single object**.

In the book **Functional Design - Principles, Patterns, and Practices** (**FD-PPP**) there is a chapter in which **Robert Martin** does the **Bowling Game kata**, first in **Java** and then in **Clojure**.

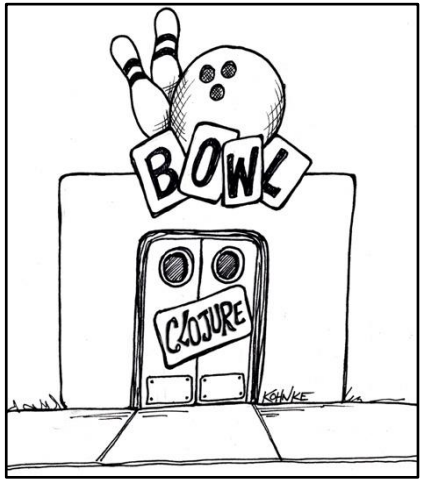
At some point in this deck series we will be looking at his **Clojure** version of the **program**, so when it comes to picking a **Java** version, it makes sense to pick the one he discusses in his book.

Now let's look at another traditional **TDD** exercise: the **Bowling Game kata**. What follows is a much-abbreviated version of that **kata** that appeared in *Clean Craftsmanship*.[1] A related video, ***Bowling Game***, is also available. You can access the video by registering at https://informit.com/functionaldesign.

1. Robert C. Martin, *Clean Craftsmanship* (Addison-Wesley, 2021).

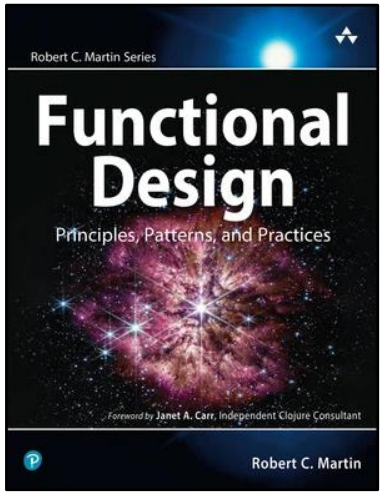


Robert C. Martin

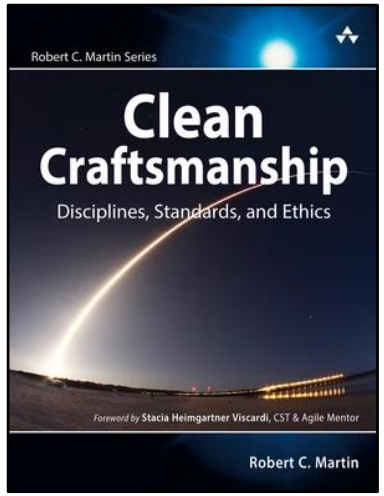


Robert Martin's **Java** **program** can be seen in the next two slides.

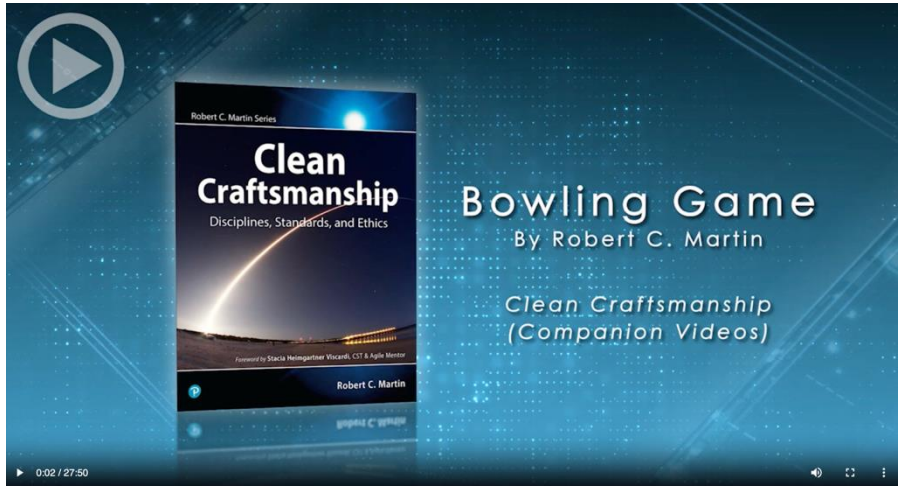
Because not all of its code is shown in **FD-PPP**, I have reconstructed it using the resources show below



FD-PPP



CC-DSE



Bowling Game Video

```java
public class Game {

  private int rolls[] = new int[21];
  private int currentRoll = 0;

  private boolean isSpare(int frameIndex) {
    return rolls[frameIndex] + rolls[frameIndex + 1] == 10;
  }

  private boolean isStrike(int frameIndex) {
    return rolls[frameIndex] == 10;
  }

  private int spareBonus(int frameIndex) {
    return rolls[frameIndex + 2];
  }

  private int strikeBonus(int frameIndex) {
    return rolls[frameIndex + 1] + rolls[frameIndex + 2];
  }

  private int twoBallsInFrame(int frameIndex) {
    return rolls[frameIndex] + rolls[frameIndex + 1];
  }

  public void roll(int pins) {
    rolls[currentRoll++] = pins;
  }

  public int score() {
    int score = 0;
    int frameIndex = 0;
    for (int frame = 0; frame < 10; frame++) {
      if (isStrike(frameIndex)) {
        score += 10 + strikeBonus(frameIndex);
        frameIndex ++;
      } else if (isSpare(frameIndex)) {
        score += 10 + spareBonus(frameIndex);
        frameIndex += 2;
      } else {
        score += twoBallsInFrame(frameIndex);
        frameIndex += 2;
      }
    }
    return score;
  }

}
```

```java
import org.example.Game;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class BowlingTest {
  private Game g;

  @Before
  public void setUp() throws Exception {
    g = new Game();
  }

  private void rollMany(int n, int pins) {
    for (int i=0; i<n; i++) {
      g.roll(pins);
    }
  }

  private void rollSpare() {
    rollMany(2, 5);
  }

  private void rollStrike() {
    g.roll(10);
  }

 @Test
  public void gutterGame() throws Exception {
    rollMany(20, 0);
    assertEquals(0, g.score());
  }

@Test
public void allOnes() throws Exception {
  rollMany(20, 1);
  assertEquals(20, g.score());
}

@Test
public void oneSpare() throws Exception {
  rollSpare();
  g.roll(7);
  rollMany(17, 0);
  assertEquals(24, g.score());
}

@Test
public void oneStrike() throws Exception {
  rollStrike();
  g.roll(2);
  g.roll(3);
  rollMany(16, 0);
  assertEquals(20, g.score());
}

@Test
public void perfectGame() throws Exception {
  rollMany(12, 10);
  assertEquals(300, g.score());
}
```

The next slide shows the same **Game class** that we have just seen, but with all **function calls inlined**.

```java
public class Game {

  private int rolls[] = new int[21];
  private int currentRoll = 0;

  public void roll(int pins) {
    rolls[currentRoll++] = pins;
  }

  public int score() {
    int score = 0;
    int frameIndex = 0;
    for (int frame = 0; frame < 10; frame++) {
      if (rolls[frameIndex] == 10) {
        score += 10 + rolls[frameIndex + 1] + rolls[frameIndex + 2];
        frameIndex ++;
      } else if (rolls[frameIndex] + rolls[frameIndex + 1] == 10) {
        score += 10 + rolls[frameIndex + 2];
        frameIndex += 2;
      } else {
        score += rolls[frameIndex] + rolls[frameIndex + 1];
        frameIndex += 2;
      }
    }
    return score;
  }

}
```

Version 1b - Java - Inlined variant of `Version1 - Java - **Robert Martin** in FD-PPP`

Back in 2009, I wanted to get some practice using **Haskell**.

I had recently been doing **Robert Martin**'s **Bowling Game Code Kata** in **Java**, so I said to myself: why not have a go at coding the game in **Haskell**?

To do that, I needed to find the **Haskell** equivalent of the **XUnit framework**, and learn how to use it.

I decided to take a shortcut by **Googling** for the `**Bowling Game Kata in Haskell**`, in the hope of finding a blog post whose tests I could reuse.

Luckily I found a great 2006 blog post by **Tom Moertel** (see next slide).

Tom Moertel's Blog **Home About Archive**

The Bowling Game Kata in Haskell

By Tom Moertel

Posted on April 5, 2006
Tags: haskell, kata, bowling

On the WPLUG mailing list I came across a post about the formation of a Pittsburgh Coding Dojo. The idea is to get a bunch of hackers together and have them work on solving a challenge problem with the goal of sharpening their programming skills and learning from each other.

There was a trial meeting on 31 March that focused on The Bowling Game Kata. The challenge was essentially to write some code that scores a full (ten-frame) game of bowling. A game is represented by a series of "rolls," each being the number of pins knocked down by a roll of the bowling ball. The scoring function must determine frame boundaries from the sequence of rolls and score all ten frames according to the rules of bowling, i.e., taking into account spares and strikes and the final frame.

The challenge sounded like a fun lunch-break problem, and so I whipped up the following solution in Haskell. (You might find it interesting to compare this solution to the Java-based solutions on the web.)

I forced myself to skip past **Tom**'s **program**, lest it influence my upcoming efforts to **code the game unaided by any existing example**, and having made a mental note to **return to it later**, since it looked very interesting, I moved on to **the program's tests** (see next slide).

```haskell
{-
   My solution to "The Bowling Game Kata"
   Tom Moertel <tom@moertel.com>
   2006-04-05

   See http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
-}

module Bowling (score) where

import Test.HUnit

-- | Compute the score for the list of rolls 'rs'

score rs = sc 0 1 rs

-- accumulate the score 's' and frame count 'f' while consuming a
-- list of rolls 'rs' one frame at a time

sc s 11 _   = s           -- frame 11 means all done; return score
sc s f rs   = case rs of  -- otherwise, consume the frame & recurse
    10:rs'                 -> sc' 3 rs'   -- strike
    x:y:rs' | x + y == 10 -> sc' 3 rs'   -- spare
            | otherwise    -> sc' 2 rs'   -- normal
    _                      -> error "ill-formed sequence of rolls"
  where
    -- accumulate the next 'n' rolls into the score and recurse
    sc' n rs' = sc (s + sum (take n rs)) (f + 1) rs'
```

Tom Moertel
@tmoertel

2006 https://blog.moertel.com/posts/2006-04-05-the-bowling-game-kata-in-haskell.html

Tom Moertel

@tmoertel

Here are my unit tests:

```
{-
                    *** Unit tests ***

            *Bowling> runTestTT tests
            Cases: 9  Tried: 9  Errors: 0  Failures: 0
-}

tests = test
    [ "gutters"       ~: score  (rep 20  0)          ~?=   0
    , "ones"          ~: score  (rep 20  1)          ~?=  20
    , "fives"         ~: score  (rep 22  5)          ~?= 150
    , "strikes"       ~: score  (rep 12 10)          ~?= 300
    , "1 + gutters"   ~: score  (1 : rep 19 0)       ~?=   1
    , "first spare"   ~: score  (5:5:5 : rep 17 0)   ~?=  20
    , "first strike"  ~: score  (10:5:5 : rep 17 0)  ~?=  30
    , "last spare"    ~: rscore (5:5:5 : rep 18 0)   ~?=  15
    , "last strike"   ~: rscore (5:5:10 : rep 18 0)  ~?=  20
    ]
  where
    rep    = replicate
    rscore = score . reverse  -- reverse list and then score it
```

If you have a little free time, code up a solution in your favorite language.

I then wrote **my own version** of the **Haskell program** (see next slide), **verifying** its **correctness** by running **Tom**'s **test suite**.

Philip Schwarz

17 years ago

2009

Hi Tom,

I wanted to get some practice in Haskell, so I thought I'd look for some Kata, and since I had recently been going over Robert Martin's Java Bowling Kata, I said to myself, what the hell, why not try the Bowling Kata in Haskell?...but how can I do TDD without XUnit? Let me just Google that just in case someone has already tried it, and sure enough, you had. Thanks for your post. I was pleasantly surprised to see that there is such a thing as HUnit.

Here is my attempt:

```
score [x, y]                          = x + y      -- Normal Frame
score [10, x, y]                      = 10 + x + y -- Strike
score [x, y, z]                       = 10 + z     -- Spare
score (10:(x:(y:rest)))               = 10 + x + y + score (x:(y:rest)) -- Strike
score (x:(y:(z:rest)))  | (x + y) == 10  = 10 + z + score (z:rest)         -- Spare
score (x:(y:rest))      | otherwise      = x + y + score rest              -- Normal Frame
```

What do you think? I think it should be possible to simplify this further (say from 6 lines down to 5 or 4), but I haven't got there yet.

It passes all your tests, except for 'fives' and 'first strike', whose frames need (IMHO) to be corrected to the following respectively:

```
    , "fives"        ~: score  (rep 21  5)          ~?= 150
    , "first strike" ~: score  (10:5:5 : rep 16 0)  ~?=  30
```

In "fives", I have taken away the 22nd roll, because you cannot have more than 21 rolls in a game.
In "first strike", I have removed the 17 gutter because after the strike (1 roll) and the spare (2 rolls), there are 8 normal frames, and therefore 8x2=16 rolls.

When Tom responded, he offered two stylistic suggestions. Here is my program again after applying his suggestions.

I have annotated the program with a bug icon because as we'll see later, the program would not pass a test that has yet to be introduced. Tom's program *would* pass the test.

Tom Moertel

17 years ago

2009

Philip, I believe you are right about my "fives" and "first strike" tests: they are incorrect. Luckily, my code passes the corrected tests, too. Whew!

Your code looks pretty good. I'll offer a couple of stylistic suggestions, though. First, the "cons" operator `(:)` is right associative, so you don't need to write `x:(y:rest)`; just `x:y:rest` will do. Second, the _otherwise_ guard on the final case isn't needed because there's no other guard for that case.

Cheers,
Tom

```
score [x, y]                          = x + y       -- Normal Frame
score [10, x, y]                      = 10 + x + y  -- Strike
score [x, y, z]                       = 10 + z      -- Spare
score (10:x:y:rest)                   = 10 + x + y + score (x:y:rest) -- Strike
score (x:y:z:rest)  | (x + y) == 10   = 10 + z + score (z:rest)       -- Spare
score (x:y:rest)                      = x + y + score rest            -- Normal Frame
```

A year after writing his **program**, in a blog post that we won't be dealing with until later in this series, **Tom** left a comment in which he referred to the **program** as being a **somewhat golfed solution**.

See below for what he means by **'golfed'**.

AI Overview +7

Code golf is a recreational programming competition where developers write the shortest possible source code to solve a specific problem. Just like in traditional golf where you want the lowest score, code golfers aim for the lowest character count. While highly entertaining, this practice deliberately ignores readability and maintainability in favor of pure density. Wikipedia +3

Key Concepts

- **Character Count:** The "score" is the total number of bytes or characters used in the code. Wikipedia +1
- **Languages:** While languages like **Python** or **Perl** are popular for their brevity, esoteric languages specifically designed for minimum syntax (like **GolfScript**) are widely used. YouTube · Charles Pandian
- **Optimization:** Golfers often exploit language quirks, obscure operator precedence, and implicit type conversions, producing code that looks entirely unreadable to standard software engineering standards. Code Golf Stack Exchange +2

In the paper **Programming as if the Domain (and Performance) Mattered** †, **Carlo Pescio** discusses the use of **two opposing code styles** to solve the **Water Between Towers** problem:

> "You are given an input array whose each element represents the height of a line of towers.
> The width of every tower is 1. It starts raining. How much water is collected between the towers?"

Here is the solution written in the **first style**, which the paper refers to as **hipster** code:

```
rainfall :: [Int] -> Int
rainfall xs = sum (zipWith (-) mins xs)
    where mins = zipWith min maxl maxr
          maxl = scanl1 max xs
          maxr = scanr1 max xs
```

See next slide for some of **Pescio**'s many observations regarding the code. See the slide after next for a definition of **hipster**.

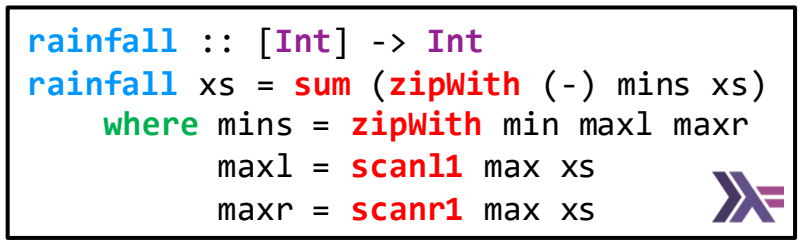
† https://drive.google.com/file/d/0B59Tysg-nEQZSXRqVjJmQjZyVXc

- That code **actively removes domain knowledge**. The idea that we have towers and water is **removed** from the table and replaced with a list of integers. The clever **removal** of **variable** and **function names**, which is part of the **strategy** to get **short/cool code**, also **removes** any possibility to use **domain terminology**. To make things worse, even the input is called **xs**, neglecting our only chance to **explain** it was an array of tower heights. But anyway, **xs** is there just because the OP didn't write the entire thing point free, which would have been considered even better (and even less informative, but who cares).

- In fact, the only reminiscence of the **domain** is in the **function name** (**rainfall**). However, given that code alone, without an explanation of the problem to be solved, you would be able to understand the **mechanical semantics** (the way data is mechanically transformed) but not the **domain semantics**. There is nothing to guide you to an **understanding** of water collected between towers.

- That **approach** (solve the exact problem, then make it **loop free** and **math-looking**) tends to create **extremely fragile code**. This has always been known as a weakness of straightforward **functional decomposition**, but among many other things it is being ignored by **the latest generation of functional programmers**, who seem **unaware** of what we have learnt in the past 30-40 years.

- This **unawareness** is causing a **funny revisionism**
  - It's ok if the code is **hard to read**: it's called **CODE** so you have to **DECODE** it.
  - The idea that **code** should reflect the **domain** is a **notion of the past**.
  - Since **naming** is **hard**, **functional code** that **gets away without variable and function names** helps the programmer being **more productive** (don't mind the fact that thinking about **good names** helps **understanding** the **problem** and that good names give hints to the reader about our solution **they are supposed to DECODE it anyway**)




```haskell
rainfall :: [Int] -> Int
rainfall xs = sum (zipWith (-) mins xs)
    where mins = zipWith min maxl maxr
          maxl = scanl1 max xs
          maxr = scanr1 max xs
```

@CarloPescio

Here is a definition of **hipster code**.

AI Overview +2

"Hipster code" generally refers to a lighthearted, esoteric programming aesthetic that prioritizes minimalist elegance, functional purity, and esoteric languages. It embraces the pursuit of "code poetry" over conventional engineering standards. secretartofscience.com

Aesthetically and structurally, hipster code often revolves around specific conventions:

The Modern "Hipster" Aesthetic

- **Functional Purity:** A heavy preference for paradigms that avoid side effects, often favoring immutable variables (`const`) and functional array methods (e.g., `.map()` , `.filter()` , `.reduce()`).
- **Minimalist Syntax:** Embracing new language features that require fewer lines, such as arrow functions, template literals, and object destructuring.
- **Quirky Languages:** Leaning into alternative or esoteric programming languages that aren't tied to corporate tech stacks, such as Clojure, Elixir, or Rust.

A "Hipster" JavaScript Example

A classic example of this clean, esoteric styling uses modern JavaScript to create a simple sequence, a look popularized by developers who view coding as a form of art:

javascript

```
// A "poetic" way to generate ten squared numbers
const generateNaturals = function* (count) {
    let i = 0;
    while (i < count) yield i++;
};

const tenSquares = [...generateNaturals(10)].map(x => x * x);
```

Use code with caution.

As for the other **code style** shown in **Programming as if the Domain (and Performance) Mattered**, which is the **opposite** of the **style** used in the **hipster code** of the **rainfall function**, see the next slide.

```
class Program
{
    static void Main(string[] args)
    {

        World w = new World(5, 3, 7, 2, 6, 4, 5, 9, 1, 2) ;
        w.Drain();
        Debug.Assert(w.Water() == 14);
    }
}


class Slice
{
    private int wall;
    private int water;
    private int air;

    public Slice(int wall = 0, int water = 0, int air = 0 )
    {
        this.wall = wall;
        this.water = water;
        this.air = air;
    }

    public bool LeakInto(Slice target)
    {
        if(air != 0)
            return false;

        int newWater = Math.Max(target.wall + target.water - wall,0);
        air = water - newWater;
        water = newWater ;
        return air > 0;
    }

    public int Water()
    {
        return water;
    }
}
```

```
class World
{
    private int width;
    private Slice[] slices;

    public World(params int[] towerHeights)
    {
        width = towerHeights.Length + 2;
        int worldHeight = towerHeights.Max();
        slices= new Slice[width];

        int x = 0;
        slices[x++] = new Slice(air:worldHeight);
        foreach( int h in towerHeights )
            slices[x++] = new Slice(wall:h, water:worldHeight -h);
        slices[x] = new Slice(air: worldHeight);
    }

    public void Drain()
    {
        for(int x = 0; x < width - 1; ++x)
            if( ! slices[x + 1].LeakInto(slices[x]) )
                break;

        for(int x = width - 2; x >= 0; --x)
            if( ! slices[x].LeakInto(slices[x+1]) )
                break;
    }

    public int Water()
    {
        return slices.Sum(s => s.Water());
    }
}
```

@CarloPescio

With the following in mind

1. **Golfed code** uses as few characters as possible, so none of its **variables**, **function names**, et cetera, are ever given **names** reflecting **domain knowledge**

2. **Hipster** code like that shown in **Carlo Pescio**'s paper has had all **domain knowledge removed** from it

3. Code written `**as if the domain mattered**` is **rich** in **domain knowledge**

let us **coin** the following **terms** (purely for use within this series of decks) :

- **domain-free code**: **code** whose **naming** reveals a **negligible amount** of **domain knowledge**
- **domain-rich code**: **code rich** in **domain knowledge** (written **as if the domain matters**)

The program on the previous slide is **domain-rich** code, whereas I think it is fair to say that my program below is **domain-free** code, even though it uses some **comments** to **communicate** a **smidgen** of **domain terminology**.

```
score [x, y]                          = x + y      -- Normal Frame
score [10, x, y]                      = 10 + x + y -- Strike
score [x, y, z]                       = 10 + z     -- Spare
score (10:x:y:rest)                   = 10 + x + y + score (x:y:rest) -- Strike
score (x:y:z:rest)  | (x + y) == 10   = 10 + z + score (z:rest)       -- Spare
score (x:y:rest)                      = x + y + score rest            -- Normal Frame
```

Tom wrote his program in April 2006. In November that same year, **Ron Jeffries** wrote a blog post called **Haskell Bowling**, in which he discusses the following **Haskell Bowling Game program** written by **Dan Mead**.

Ron Jeffries

About Search Site Categories

Haskell Bowling

© Nov 1, 2006 • [XProgramming]

At the Simple Design and Testing conference, Dan Mead bravely agreed to implement the infamous Bowling Game exercise, TDD style, in Haskell. It was fun and interesting. Here I present some discussion, his program, and a Java program in a similar style.

2006

https://ronjeffries.com/xprog/articles/dbchaskellbowling/

```haskell
score ([]) = 0
score (x:[]) = x
score (x:y:[]) = x + y
score (x:y:z:[]) = x + y + z
score (x:y:z:xs) = if  (x == 10)             then x + y + z +  score(y:z:xs)
                   else if (((x + y) == 10)) then x + y + z +  score(z:xs)
                   else x + y + score(z:xs)
```

Dan Mead
daniel-mead-95a4b6379

Version 3 - **Haskell** - **domain-free** - **Dan Mead**

Dan Mead's version of the program, just like mine, is in domain-free style, so it is not surprising that, as shown on the next slide, Ron Jeffries finds that the whole program, and in particular the line of code highlighted below, is not obvious or communicative, that it obscures/obfuscates the rules of the game.

```
score [x, y]                          = x + y       -- Normal Frame
score [10, x, y]                      = 10 + x + y  -- Strike
score [x, y, z]                       = 10 + z      -- Spare
score (10:x:y:rest)                   = 10 + x + y + score (x:y:rest) -- Strike
score (x:y:z:rest)  | (x + y) == 10   = 10 + z + score (z:rest)       -- Spare
score (x:y:rest)                      = x + y + score rest            -- Normal Frame
```


Version 2 - Haskell - domain-free - Philip Schwarz

```
score ([]) = 0
score (x:[]) = x
score (x:y:[]) = x + y
score (x:y:z:[]) = x + y + z
score (x:y:z:xs) = if  (x == 10)             then x + y + z +  score(y:z:xs)
                   else if (((x + y) == 10)) then x + y + z +  score(z:xs)
                   else x + y + score(z:xs)
```

Version 3 - Haskell - domain-free - Dan Mead

The bottom program is annotated with a bug icon for the same reason as the top one.

The **x:y:z:[] rule** handles the **case** of **three rolls** at the **end of the game**, i.e. a **bonus roll**.

This rule is **not**, in my opinion, **obvious** or **communicative**.

What it's relying on is that it happens that if you roll a **strike**, you get ten + the next two rolls, but the first roll (x) equals ten, and if you roll a **spare**, the score is ten plus the next one roll, but the first two (x + y) sum to ten.

When I observe this truth in my demonstrations, a significant portion of the audience **objects**, saying that it **obscures** the **rules** of the **game**.

Ron Jeffries

```
score ([]) = 0
score (x:[]) = x
score (x:y:[]) = x + y
score (x:y:z:[]) = x + y + z
score (x:y:z:xs) = if  (x == 10)            then x + y + z +  score(y:z:xs)
                   else if (((x + y) == 10)) then x + y + z +  score(z:xs)
                   else x + y + score(z:xs)
```

Dan Mead

This is all pretty straightforward, I think, **except** for the **obfuscation** of the **rules of bowling** that appears in the **x+y+z**, as already mentioned.

Dan's assertion was that the **Haskell programs** we get **rather directly reflect our thoughts,** and that they are **compact** and **easy to understand**.

I would agree that **with what we just figured out fresh in our mind,** the code is **clear** – and it's certainly **compact**.

However, walking up to it, I do not find it clear. It is **recursive**, and its **rules** are a bit odd, mostly due to the need in the **x:y:z rules** to look for an item that may not be there, resulting in the **special ending rules** like x:y:[] and x:y:z:[]. And we are left with that **tag rule** x:[], which **appears to be redundant**, but it's really hard to be sure.

Nonetheless, it's an **interesting implementation** that **seems to get the right answers**, in very few lines. And it was quite brave of **Dan** to sit in front of a bunch of people and try to improvise a solution. I find that scary when I do it, at least. So thanks to **Dan** for an interesting and thought-provoking session.

While **Ron Jeffries** was intent on writing the **Java program** on the next two slides in a **`similar style`** to **Dan Mead**'s **Haskell program**, which is reflected, for example, in the fact that they both exploit **recursion**, we can see that while the code of the **Haskell program** is **domain-free**, the code in the **Java** program is **domain-rich**.

By the way, I have annotated the **Java program** with the **bug icon** because as we'll see later, the program suffers from the same problem as the two **Haskell programs**!

```java
package haskellBowling;

import java.util.Arrays;
import java.util.List;

public class BowlingGame {
    List<Integer> rolls;

    public Integer score(Integer[] rollsArray) {
        rolls = Arrays.asList(rollsArray);
        return score();
    }

    private Integer score() {
        if (rolls.size() == 0) return 0;
        if (rolls.size() == 3) return thisFrameScore();
        return thisFrameScore() + remainingScore();
    }

    private Integer remainingScore() {
        setRemainingRolls();
        return score();
    }

    private Integer thisFrameScore() {
        Integer frameScore = 0;
        for (Integer roll : thisFramesRolls())
            frameScore += roll;
        return frameScore;
    }

    private List<Integer> thisFramesRolls() {
        return rolls.subList(0, numberOfRollsToScore());
    }

    private int numberOfRollsToScore() {
        return (isMark()) ? 3: 2;
    }

    private boolean isMark() {
        return isStrike() || isSpare();
    }

    private boolean isStrike() {
        return rolls.get(0) == 10;
    }

    private boolean isSpare() {
        return rolls.get(0) + rolls.get(1) == 10;
    }

    private void setRemainingRolls() {
        rolls =  rolls.subList(frameSize(rolls), rolls.size());
    }

    private int frameSize(List<Integer> rolls) {
        return isStrike()? 1: 2;
    }
}
```

The program is annotated with a **bug icon** for the same reason as the previous two **Haskell** programs.

Version 4 - **Java** - **domain-rich** - inspired by **Dan Mead**'s **Haskell** version - **Ron Jeffries**

```java
package haskellBowling;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class BowlingGameTest {
    BowlingGame game;

    @Before public void setUp() throws Exception {
        game = new BowlingGame();
    }

    @Test public void hookup() {
        assertTrue(true);
    }

    @Test public void opens() throws Exception {
        assertEquals(60, game.score(new Integer[]
          {3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3}));
    }

    @Test public void spare() throws Exception {
        assertEquals(22, game.score(new Integer[]
          {6,4,5,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}));
    }

    @Test public void perfect() throws Exception {
        assertEquals(300, game.score(new Integer[]
          {10,10,10,10,10,10,10,10,10,10,10,10}));
    }

    @Test public void alternatingStrikeSpare() throws Exception {
        assertEquals(200, game.score(new Integer[]
          { 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10}));
    }

    @Test public void alternatingSpareStrike() throws Exception {
        assertEquals(200, game.score(new Integer[]
          { 5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5}));
    }

    @Test public void trailingSpare() throws Exception {
        assertEquals(20, game.score(new Integer[]
          { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,5,5}));
    }
}
```

Version 4 - Java - domain-rich - inspired by Dan Mead's Haskell version - Ron Jeffries

See next slide for some of **Ron Jeffries**‘ observations as he compares the **Haskell program** with the **Java** one.

Ron Jeffries

Assessing the Java, and the Haskell Approach

2006

The Java is certainly much more code, and while it expresses some ideas that the Haskell doesn't, such as strike and spare and mark, the Haskell would be much shorter even if it were extended to address those concerns. And shorter is generally preferable.

The recursive solution, however, is questionable on more fundamental grounds. A game of bowling consists of ten frames, not less or more, and the "ten-ness" of the game is not represented in the recursive solutions at all. Even if we let that slide, the recursive solutions make it a bit hard to understand what's going on. And, as I also mentioned, the x+y+z trick raises objections from a number of observers, in that the rules of the game do not make clear that the rolls scored for strike and spare are the same rolls, yet the code takes advantage of that fact.

Dan's assertion, as I recall it, was that Haskell lets us express the program "in the way we think". On the contrary, what Haskell does in my opinion is let us express the program in the way Haskell thinks, in a concise form. And we learn to think the way Haskell thinks, not because humans normally think in terms of pattern matches and recursion, but because we learn to work that way. And having learned it, it seems natural. Well, having learned Java and C# and SNOBOL and twenty other languages, they all seemed natural too, but I confess that I have liked the expressive ones like LISP and Smalltalk more than I did, say FORTRAN.

I'm certainly not knocking Haskell: it looks like it would be fun, and for some class of problems, perhaps a good way to go. I'm concerned that no one who wasn't there may ever be able to read another person's Haskell program, but lord knows it's not easy to write code in any language that is easy to understand for a new reader.

What we see here is that it is possible to emulate part of the Haskell solution in Java fairly easily, but with lots more code, and that other parts, such as the pattern matching, are quite difficult. It might be amusing to try to do the same solution in, say, Ruby, which would probably be more compact and perhaps would cleave closer to the Haskell style.

Both the **Haskell** program and the **Java** program use **recursion**, about which **Ron Jeffries** says the following:

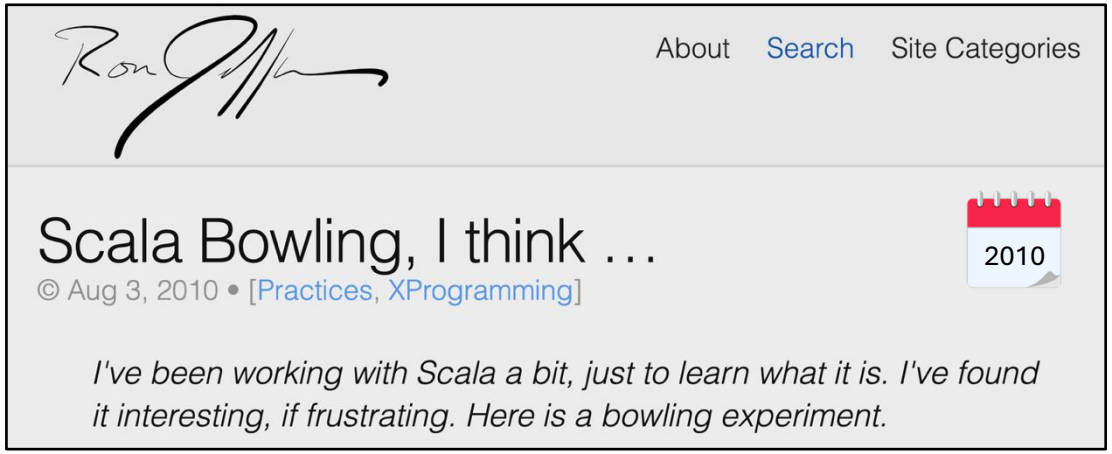
- The **recursive** solutions makes it a bit **hard** to **understand** what's going on
- A **game** of **bowling** consists of **ten frames**, **not less or more**, and the **"ten-ness"** of the **game** is not represented in the **recursive solutions** at all

One of the many things I like about **Ron Jeffries** is his way with language, thanks to which we can speak of the **ten-ness** of **Ten Pin Bowling**.

As we'll soon see, **ten-ness** is **important**: a **ten-pin bowling game program** not **representing** the **ten-ness** of the **game** is a **bug**.

Four years later, in 2010, **Ron Jeffries** wrote a series of articles about coding the **Bowling Game** in **Scala**, with the first one titled **`Scala Bowling, I think...`**.

So I dug up my **Haskell program**, translated it to **Scala**, and showed it to **Ron** in a comment on the article.



https://ronjeffries.com/xprog/articles/scala-bowling-i-think/

Ron Jeffries

Hi **Ron**,

I have not touched **Scala** for months now, still smarting from the problems I had in Eclipse. I should be sleeping by now, but I couldn't resist translating my **Haskell** attempt into **Scala**. It seems to pass your tests. Here is the **Haskell** version:

```
score [x, y]                          = x + y      — Normal Frame
score [10, x, y]                      = 10 + x + y — Strike
score [x, y, z]                       = 10 + z     — Spare
score (10:(x:(y:rest)))               = 10 + x + y + score (x:(y:rest)) — Strike
score (x:(y:(z:rest))) | (x + y) == 10 = 10 + z + score (z:rest)        — Spare
score (x:(y:rest))     | otherwise     = x + y + score rest             — Normal Frame
```

Version 2 - **Haskell** - **domain-free** - **Philip Schwarz**

2009

This is my **Haskell** program *before* applying **Tom**'s two stylistic suggestions

And here is the corresponding **Scala** version:

```
def score(rolls : List[Int]) : Int = rolls match {
  case List(x,y)         => x + y
  case List(10,x,y)      => 10 + x + y
  case List(x,y,z)       => 10 + z
  case (10::x::y::rest) => 10 + x + y + score(x::y::rest)
  case (x::y::z::rest)  => if ((x + y) == 10) 10 + z + score(z::rest)
                           else x + y + score(z::rest)
}
```

Version 2b - **Scala** - **domain-free** - **Philip Schwarz**

2010

Ron liked my **Scala program** and decided to **build on its ideas**.

But later **Jon Bettinger** **found a bug** affecting both my **program** and **Ron**'s **program**. It is this **bug** that caused me, in this deck, to tag my **program** (and a few others) with the **bug icon**.

see next two slides

see subsequent four slides

Scala ... a failing test

Jon Bettinger has found a failing test! Excellent!

(Aug 5, 2010) [Practices, XProgramming]

Scala Bowling … more function cowbell

My critiXXXXX advisors expected a more function-oriented solution. Here's something a bit better, perhaps ...

(Aug 4, 2010) [Practices, XProgramming]

Scala Bowling … building on Philip's approach

Philip Schwarz provided a nice-looking implementation. Let's look at it and try to build on his ideas.

(Aug 4, 2010) [Practices, XProgramming]

Scala Bowling, I think …

I've been working with Scala a bit, just to learn what it is. I've found it interesting, if frustrating. Here is a bowling experiment.

(Aug 3, 2010) [Practices, XProgramming]

2010

```scala
object Game {

   def score(rolls: List[Int]):Int = {

      rolls match {

           case List(first, second)
             => first+second

           case List(first, second, third)
             => first + second + third

           case 10::second::third::theRest
             => 10 + second + third + score(second::third::theRest)

           case first::second::third::theRest if (first + second == 10)
             => 10 + third + score(third::theRest)

           case first::second::theRest
             => first + second + score(theRest)
      }
   }
}
```

Ron Jeffries

Version 5 - Scala - domain-free - Ron Jeffries

```
import org.scalatest.Spec

class ExampleSpec extends Spec {

    describe("A Bowling Game") {

        it("should score all gutters as zero") {
            expect(0)(Game.score(List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score all 5,4 as 90") {
            expect(90) (Game.score(List(5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4)))
        }

        it("should score spare game 6, 4, 5, 2 as 22") {
            expect(22)(Game.score(List(6,4,5,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score strike game 10, 5, 2, 7 as 31") {
            expect(31)(Game.score(List(10,5,2,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score perfect game as 300") {
            expect(300)(Game.score(List(10,10,10,10,10,10,10,10,10,10,10,10)))
        }

    }
}
```

Ron Jeffries

Version 5 - Scala - domain-free - Ron Jeffries

Jon Bettinger's comment in one of the articles in **Ron**'s **Scala Bowling** series

I don't know **how you can get away with not counting frames**. Consider **8 frames of gutters**; **strike** in 9th; 1,0 in tenth should yield **12**. However, **9 frames of gutters**; **strike**, 1, 0 in tenth should yield **11**.

I guess I wasn't clear enough. You can't reliably recognize it. There is **no recognizable difference** between 8 frames of gutter followed by 10, 1, 0 vs 9 frames of 0 followed by 10, 1, 0. Both are valid scores for a game.

This test passes:

```
it("should count not double count tenth frame bonus rolls" ) {
  val game = new Game(List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,1,0))
  expect(11)(game.score)
}
```

Here's the **failing test**. 

```
it("should count non-mark tenth frame after strike in ninth" ) {
  val game = new Game(List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,1,0))
  expect(12)(game.score)
}
```

Jon Bettinger

Both my Scala program and Ron's Scala program were failing Jon's second test.

```
...
[info] ExampleSpec:
[info] A Bowling Game
[info] - should score all gutters as zero
[info] - should score all 5,4 as 90
[info] - should score spare game 6, 4, 5, 2 as 22
[info] - should score strike game 10, 5, 2, 7 as 31
[info] - should score perfect game as 300
[info] - should count not double count tenth frame bonus rolls
[info] - should count non-mark tenth frame after strike in ninth *** FAILED ***
[info]   Expected 12, but got 11 (ExampleSpec.scala:37)
[error] Failed: Total 7, Failed 1, Errors 0, Passed 6
[error] Failed tests:
[error]         ExampleSpec
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 4 s, completed 27 Jun 2026, 20:32:55
```

I was surprised that my **Scala program** failed one of **Jon**'s **two tests**, because the **program** did **pass**, not just **Ron**'s tests, but also the following tests by **Robert Martin**.

```
// Robert Martin's (10-frame) tests in his book ASDPP
// http://www.amazon.co.uk/Software-Development-Principles-Patterns-Practices/dp/0135974445
assert (300 == score(List(10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10)))             // TestPerfectGame
assert (133 == score(List(1, 4, 4, 5, 6, 4, 5, 5, 10, 0, 1, 7, 3, 6, 4, 10, 2, 8, 6))) // TestSampleGame
assert (299 == score(List(10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 9)))                  // TestHeartBreak
assert (270 == score(List(10, 10, 10, 10, 10, 10, 10, 10, 10, 9, 1, 1)))                // TestTenthFrameSpare

// Robert Martin's tests in his bowling kata slides
// http://butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt
assert (  0 == score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0)))           // testGutterGame
assert ( 20 == score(List(1,1, 1,1, 1,1, 1,1, 1,1, 1,1, 1,1, 1,1, 1,1, 1,1)))           // testAllOnes
assert ( 16 == score(List(5,5, 3,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0)))           // testOneSpare
assert ( 24 == score(List(10,  3,4, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0)))           // testOneStrike
assert (300 == score(List(10,  10,  10,  10,  10,  10,  10,  10,  10,  10,10,10)))      // testPerfectGame
```

Since my **Scala program** was just a straightforward translation of my **Haskell program**, the same considerations also hold for the latter.

About Search Site Categories

Scala ... a failing test

© Aug 5, 2010 • [Practices, XProgramming]

2010

Jon Bettinger has found a failing test! Excellent!

Ron Jeffries

https://ronjeffries.com/xprog/articles/scala-a-failing-test/

In a comment on an earlier article, **Jon** points out that the **special check** for a **final bonus frame** can trigger on the **9th frame**, not just the **10th**. The game can end **10-x-y** with a **strike** in the **9th frame** and an **open frame** in the **10th**! A test that **succeeds** and one that **fails** are:

```
it("should count not double count tenth frame bonus rolls") {
    expect(11)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,1,0)))
}

// fails !!!
it("should count non-mark tenth frame after strike in ninth") {
    expect(12)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 1,0)))
}
```

My initial–but still thoughtful–reaction to this is that the current **tail-recursive** approach is **doomed** to **fail** without **frame counting**. I have a fairly simple counting fix in hand. And I **suspect** that a **two-phase approach**, parsing frames from the left and then summing, might be made to work **without counting**. Stylistic arguments aside, I think these **count-free solutions** are getting **hard to understand**.

I'd like to see one that shows up as clean and yet in **full functional style**. I'll show my current solution below…

see next slide

Hi **Ron**, You said "Jon Bettinger found a failing test! Excellent!". **Excellent indeed!** This **series** of **posts** is turning into a **learning experience**.

I only allowed myself to show you my **Scala program** on the grounds that it **passes** your **tests**. I wrote the **Haskell program** back in 2009, as a response to **Tom Moertel**'s 2006 blog post.

It was the first time I had done the **bowling kata** with a **functional language**, and while I was pleased with the simplicity of the solution, I also feared that **some corner case might break it**. But I told myself: no need to worry, if it **passes** all the **nine tests** [in **Tom Moertel**'s **test suite**], it meets its **requirements**…

Instead, I should have heeded **Kent Beck**'s advice: "test until fear turns into boredom".

Those **nine tests** were **not enough** to eliminate all traces of **fear**, and I wasn't **bored** yet either.

Here are the **changes** that **Ron** made to his **program** to get it to **pass** the **failing test** provided by **Jon**.

The way he got the **program** to **represent** the **ten-ness** of the **bowling game** is by adding **frame counting**.

See next two slides for the **corrected program** and an updated **test suite** with **Jon**'s **two tests** added to it.

Ron Jeffries

```
object Game {                                                    =  object Game {

   def score(rolls: List[Int]):Int = {                           <>    def score(frame: Int, rolls: List[Int]):Int = {

      rolls match {                                              =        rolls match {

           case List(first, second)                                          case List(first, second)
             => first+second                                                   => first+second

           case List(first, second, third)                       <>          case List(first, second, third) if frame == 10
             => first + second + third                           =             => first + second + third

           case 10::second::third::theRest                                   case 10::second::third::theRest
             => 10 + second + third + score(second::third::theRest)  <>        => 10 + second + third + score(frame+1, second::third::theRest)

           case first::second::third::theRest if (first + second == 10)  =     case first::second::third::theRest if (first + second == 10)
             => 10 + third + score(third::theRest)               <>          => 10 + third + score(frame+1, third::theRest)

           case first::second::theRest                           =           case first::second::theRest
             => first + second + score(theRest)                  <>          => first + second + score(frame+1, theRest)
      }                                                          =        }
   }                                                                    }
                                                                 -+    def score(rolls: List[Int]):Int = score(1, rolls)
}                                                                =  }
```

```scala
object Game {

   def score(frame: Int, rolls: List[Int]):Int = {

      rolls match {

           case List(first, second)
             => first+second

           case List(first, second, third) if frame == 10
             => first + second + third

           case 10::second::third::theRest
             => 10 + second + third + score(frame+1, second::third::theRest)

           case first::second::third::theRest if (first + second == 10)
             => 10 + third + score(frame+1, third::theRest)

           case first::second::theRest
             => first + second + score(frame+1, theRest)
      }
   }
   def score(rolls: List[Int]):Int = score(1, rolls)
}
```

Version 5b - Scala - domain-free - Version 5 corrected to model the ten-ness of the game - Ron Jeffries

Ron Jeffries

Frame counting could also be used to fix my buggy Haskell program, and Dan's.

```scala
import org.scalatest.Spec

class ExampleSpec extends Spec {

    describe("A Bowling Game") {

        it("should score all gutters as zero") {
            expect(0)(Game.score(List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score all 5,4 as 90") {
            expect(90) (Game.score(List(5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4,5,4)))
        }

        it("should score spare game 6, 4, 5, 2 as 22") {
            expect(22)(Game.score(List(6,4,5,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score strike game 10, 5, 2, 7 as 31") {
            expect(31)(Game.score(List(10,5,2,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)))
        }

        it("should score perfect game as 300") {
            expect(300)(Game.score(List(10,10,10,10,10,10,10,10,10,10,10,10)))
        }

        // Jon Bettinger's two tests

        it("should count not double count tenth frame bonus rolls") {
            expect(11)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,1,0)))
        }

        it("should count non-mark tenth frame after strike in ninth") {
            expect(12)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 1,0)))
        }
    }
}
```

Version 5b - **Scala** - **domain-free** - Version 5 corrected to model the **ten-ness** of the game - **Ron Jeffries**

Ron Jeffries

Ron's corrected Scala program passes both Jon's tests.

```
...
[info] ExampleSpec:
[info] A Bowling Game
[info] - should score all gutters as zero
[info] - should score all 5,4 as 90
[info] - should score spare game 6, 4, 5, 2 as 22
[info] - should score strike game 10, 5, 2, 7 as 31
[info] - should score perfect game as 300
[info] - should count not double count tenth frame bonus rolls
[info] - should count non-mark tenth frame after strike in ninth
[info] Passed: Total 7, Failed 0, Errors 0, Passed 7
[success] Total time: 5 s, completed 27 Jun 2026, 20:44:31
```

So thanks to Ron's 2010 Scala Bowling blog series, we learnt about the need for bowling game programs to represent the ten-ness of the game, and also that some bowling game test suites may be lacking in that they don't verify that ten-ness is represented.

The first program that we saw in this deck was Robert Martin's Java program, which represents ten-ness using the following loop:

```
for (int frame = 0; frame < 10; frame++) { … }
```

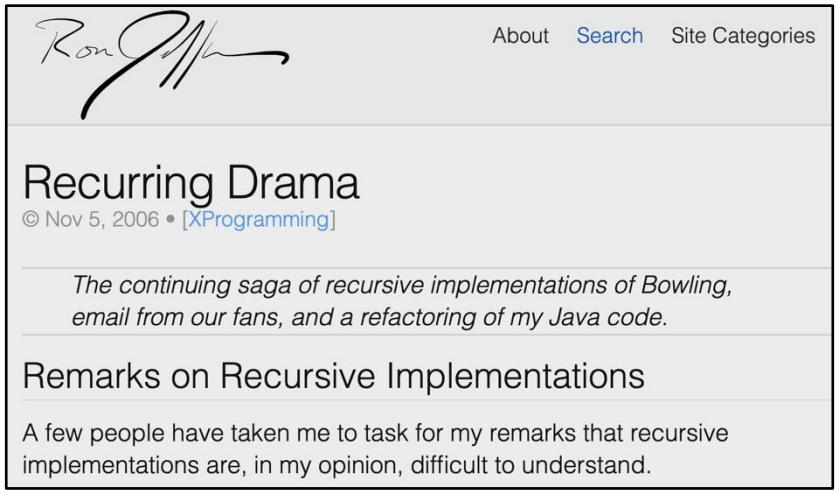
What about the Haskell and Java programs that we saw in Ron's 2006 Haskell Bowling blog series, do they represent ten-ness?

And what about the first Haskell program that we saw in this deck, Tom Moertel's program, does it represent ten-ness?

It turns out that, back in 2006, **Ron** didn't just write a single **Haskell Bowling blog post**, there was also a **follow-up post**, called **Recurring Drama**, in which the following happened:

- **Dan mead**'s **Haskell program**, and **Ron**'s translation into **Java**, were found to **fail** one of **two new tests** written by someone called **Pit**.
- **Tom Moertel**, author of the first **Haskell** program seen in this deck (the examination of which was postponed), contacted **Ron** privately and said, amongst other things, that the **problem** with the other **implementations** (i.e. **recursive implementations**, other than his, which fail the test, e.g. that by **Dan Mead**) isn't that they are **recursive**, but that they use an **unreliable termination test**.
- **Tom** referred to **Pit**'s **failing test** as the **ninth-frame-strike test**.

Ron Jeffries



https://ronjeffries.com/xprog/articles/dbcrecurringdrama/

Tom Moertel

...I also heard privately from **Tom Moertel** on the subject, also expressing the notion that **recursion** is not the **culprit**. He offers:

> The **problem** with the other **implementations** isn't that they are **recursive**, but that they use an **unreliable termination test**. Thus, I wouldn't characterize this **flaw** as being **attributable** to **recursion**. (I hope you will update your article to make this clear.)
>
> Rather, the **culprit** is that these **implementations** rely upon a **particular sequence** of **final rolls** to determine when to end the **scoring process**. As you have pointed out, the **final rolls**, when **considered by themselves**, **can be ambiguous**. Any implementation that relied upon them, **recursive or not**, would fail the **ninth-frame-strike test**.

The **problem** with the **"other" implementations**, as **Tom** suggests, certainly is that **they have improper termination conditions**. I think that **concern** is **more endemic** to **recursive solutions** than to **iterative ones**. In **iterative solutions**, we tend to **loop very simply**, over some integer sequence (1 to number of frames, for example), or over all the elements of a list. In **recursive ones**, the **approved style** is to **recur until some termination condition arises**. When all we're doing is processing a list, this can generate the same result. When we really should do something exactly ten times, it's common in writing recursive approaches to try to terminate on some condition in the structure we're processing, not on a count. Tom's implementation, it seems to me, is not typical of the way one tries to program with recursive algorithms.

(His implementation has the advantage of being the one that works, mind you, which is rather an important consideration by any standards!)

Tom goes on to say:

> Also, you brought up an interesting hypothesis: "Pit hypothesizes, and I'm inclined to agree, that you simply must deal with the tenth frame differently in all these recursive versions." I should point out that my implementation serves as a counterexample to this hypothesis: it is recursive and yet deals with the tenth frame the same as the rest.

I can't entirely agree with **Tom** here. **His implementation does in fact deal with the tenth frame differently**. If it did not, it would have the same defect that **Pit** discovered, dealing with 10, 2,3 at the end of the game. One way or another, any implementation has to recognize that it should not treat the trailing 2,3 as a new frame, but should instead stop.

Ron Jeffries

Dan Mead

Stop Marker

Dan Mead reports that he has a version that doesn't use a frame count, but that has a stop marker of 0,0 on the end of the list, and that it is passing the tests. I haven't seen it yet, but it seems to me that with the stop marker added, at least some of the other termination conditions are impossible, and I also grant that I don't see why the addition of an empty frame would make the program work.

Now I suppose it could just be a personal disability not to be able to see why that sort of a thing would work. Certainly adding a stop marker to a list is a common recursive programming technique. But frankly, I don't think it's just a personal problem. I've been writing recursive code, in many languages, since the early 60's. Wrote a master's thesis on list processing, as a matter of fact. Still, perhaps I'm just past it.

I would bet, though, that if we were to take some set of problems that can be solved in a natural fashion both recursively and iteratively, and had a bunch of programmers write both versions, they would have more trouble with the recursive ones, on the average. And I'd bet that if we had people read the resulting programs, they would have more trouble understanding the recursive implementations, and convincing themselves that they work.

I could, of course, be wrong – I frequently am. But I think we'd find that recursion is harder for most folks than iteration.

https://ronjeffries.com/xprog/articles/dbcrecurringdrama/

The Java Program

Moving right along … let's look at my Java program and see about making it actually work. I've added Pit's two tests to the list: …

The final test fails in my version, as it does in the Haskell and Ruby versions as well. (But not in Tom's or Alistair's, mind you.) Let's see about fixing up the Java code.

…

Today's Summary

There has been more trouble in this sequence of implementations than we usually see. I attribute that trouble to inherently higher complexity in recursive solutions than iterative ones, though some readers do not agree. One way or another, something went not so smoothly, and the experience, while it wouldn't cause me not to try a recursive solution, would certainly cause me to operate more carefully when working with this kind of thing. I would be inclined to use more tests – especially more tests that offer variation at the end of the rolls list. Other forms of care are probably needed … perhaps some work on paper or the like.

So far, though rumors of a version that use a fake frame as a stop marker are out there, we've not seen running tested code that uses that approach. Perhaps a reader will provide one, or perhaps I'll play with that another day. For now … it's time to wait for another round of comments, which are, as always, solicited.

The Code

here are the tests and code as they now stand:

Ron Jeffries

2006

see next three slides

https://ronjeffries.com/xprog/articles/dbcrecurringdrama/


```
package haskellBowling;

import java.util.Arrays;
import java.util.List;

public class BowlingGame {
    List<Integer> rolls;

    public Integer score(Integer[] rollsArray) {
        rolls = Arrays.asList(rollsArray);
        return score();
    }

    private Integer score() {
        if (rolls.size() == 0) return 0;
        if (rolls.size() == 3) return thisFrameScore();
        return thisFrameScore() + remainingScore();
    }

    private Integer remainingScore() {
        setRemainingRolls();
        return score();
    }

    private Integer thisFrameScore() {
        Integer frameScore = 0;
        for (Integer roll : thisFramesRolls())
            frameScore += roll;
        return frameScore;
    }

    private List<Integer> thisFramesRolls() {
        return rolls.subList(0, numberOfRollsToScore());
    }

    private int numberOfRollsToScore() {
        return (isMark()) ? 3: 2;
    }

    private boolean isMark() {
        return isStrike() || isSpare();
    }

    private boolean isStrike() {
        return rolls.get(0) == 10;
    }

    private boolean isSpare() {
        return rolls.get(0) + rolls.get(1) == 10;
    }

    private void setRemainingRolls() {
        rolls =  rolls.subList(frameSize(rolls), rolls.size());
    }

    private int frameSize(List<Integer> rolls) {
        return isStrike()? 1: 2;
    }
}
```


```
package haskellBowling;

import java.util.Arrays;
import java.util.List;

public class BowlingGame {

  public Integer score(Integer[] rollsArray) {
    return score(Arrays.asList(rollsArray), 10);
  }

  private Integer score(List<Integer> rolls, int framesRemaining) {
    if (framesRemaining == 0) return 0;

    return thisFrameScore(rolls) + score(getRemainingRolls(rolls), framesRemaining-1);
  }

  private Integer thisFrameScore(List<Integer> rolls) {
    Integer frameScore = 0;
    for (Integer roll : thisFramesRolls(rolls))
      frameScore += roll;
    return frameScore;
  }

  private List<Integer> thisFramesRolls(List<Integer> rolls) {
    return rolls.subList(0, numberOfRollsToScore(rolls));
  }

   private int numberOfRollsToScore(List<Integer> rolls) {
    return (isMark(rolls)) ? 3: 2;
  }

  private boolean isMark(List<Integer> rolls) {
    return isStrike(rolls) || isSpare(rolls);
  }

  private boolean isStrike(List<Integer> rolls) {
    return rolls.get(0) == 10;
  }

  private boolean isSpare(List<Integer> rolls) {
    return rolls.get(0) + rolls.get(1) == 10;
  }

  private List<Integer> getRemainingRolls(List<Integer> rolls) {
    return rolls.subList(frameSize(rolls), rolls.size());
  }

  private int frameSize(List<Integer> rolls) {
    return isStrike(rolls)? 1: 2;
  }
}
```


```java
package haskellBowling;

import java.util.Arrays;
import java.util.List;

public class BowlingGame {
    List<Integer> rolls;

    public Integer score(Integer[] rollsArray) {
        return score(Arrays.asList(rollsArray), 10);
    }

    private Integer score(List<Integer> rolls, int framesRemaining) {
        if (framesRemaining == 0) return 0;
        return thisFrameScore(rolls)
               + score(getRemainingRolls(rolls), framesRemaining-1);
    }

    private Integer thisFrameScore(List<Integer> rolls) {
        Integer frameScore = 0;
        for (Integer roll : thisFramesRolls(rolls))
            frameScore += roll;
        return frameScore;
    }

    private List<Integer> thisFramesRolls(List<Integer> rolls) {
        return rolls.subList(0, numberOfRollsToScore(rolls));
    }

    private int numberOfRollsToScore(List<Integer> rolls) {
        return (isMark(rolls)) ? 3: 2;
    }

    private boolean isMark(List<Integer> rolls) {
        return isStrike(rolls) || isSpare(rolls);
    }

    private boolean isStrike(List<Integer> rolls) {
        return rolls.get(0) == 10;
    }

    private boolean isSpare(List<Integer> rolls) {
        return rolls.get(0) + rolls.get(1) == 10;
    }

    private List<Integer> getRemainingRolls(List<Integer> rolls) {
        return rolls.subList(frameSize(rolls), rolls.size());
    }

    private int frameSize(List<Integer> rolls) {
        return isStrike(rolls)? 1: 2;
    }
}
```

Version 4b - Java - domain-rich - Version 4 (inspired by Dan Mead's Haskell version) corrected to represent ten-ness - Ron Jeffries

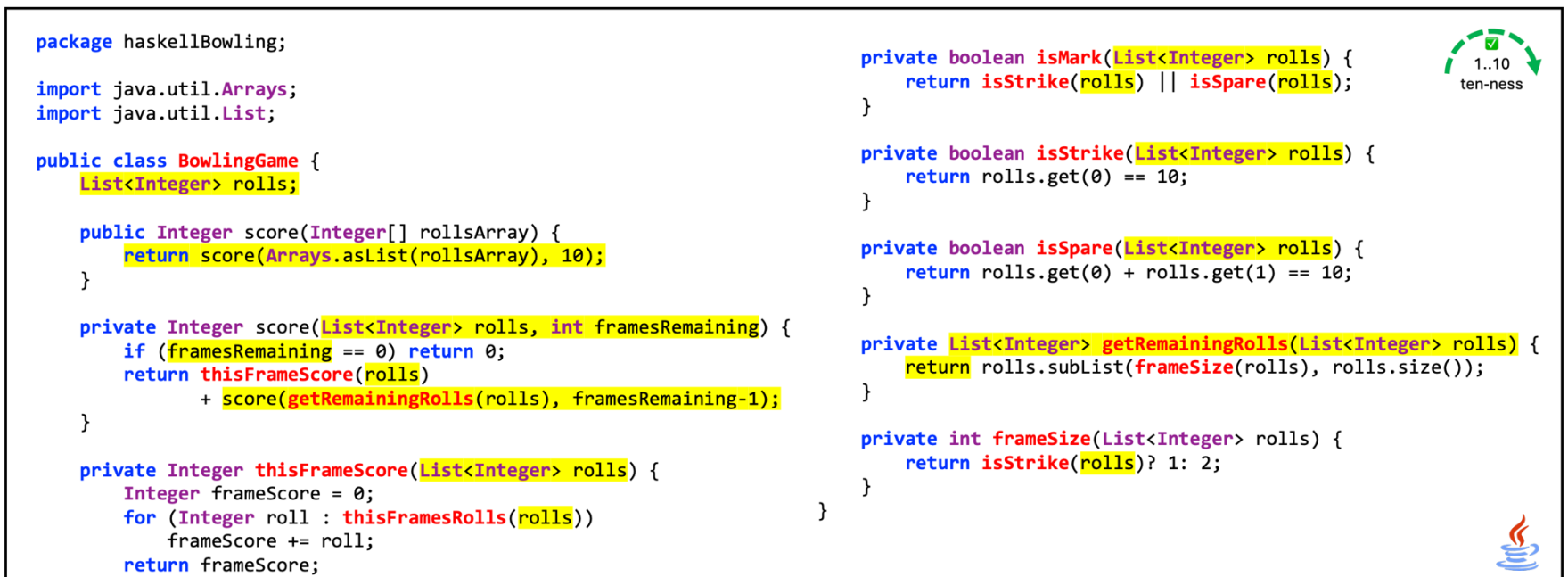
```java
package haskellBowling;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class BowlingGameTest {
    BowlingGame game;

    @Before public void setUp() throws Exception {
        game = new BowlingGame();
    }

    @Test public void hookup() {
        assertTrue(true);
    }

    @Test public void opens() throws Exception {
        assertEquals(60, game.score(new Integer[]
          {3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3}));
    }

    @Test public void spare() throws Exception {
        assertEquals(22, game.score(new Integer[]
          {6,4,5,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}));
    }

    @Test public void perfect() throws Exception {
        assertEquals(300, game.score(new Integer[]
          {10,10,10,10,10,10,10,10,10,10,10,10}));
    }

    @Test public void alternatingStrikeSpare() throws Exception {
        assertEquals(200, game.score(new Integer[]
          { 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10}));
    }

    @Test public void alternatingSpareStrike() throws Exception {
        assertEquals(200, game.score(new Integer[]
          { 5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5}));
    }

    @Test public void trailingSpare() throws Exception {
        assertEquals(20, game.score(new Integer[]
          { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,5,5}));
    }

    @Test public void pitStrikeFinalFrame() throws Exception {
        assertEquals(15, game.score(new Integer[]
          { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3}));
    }

†   @Test public void pitStrikeNinthFrame() throws Exception {
        assertEquals(20, game.score(new Integer[]
          { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3}));
    }
}
```


Pit

Version 4b - Java - domain-rich - Version 4 (inspired by Dan Mead's Haskell version) corrected to include Pit's two new tests - Ron Jeffries

† This test is referred to by Tom Moertel as the ninth-frame-strike test.

```java
package haskellBowling;

import java.util.Arrays;
import java.util.List;

public class BowlingGame {
    List<Integer> rolls;

    public Integer score(Integer[] rollsArray) {
        return score(Arrays.asList(rollsArray), 10);
    }

    private Integer score(List<Integer> rolls, int framesRemaining) {
        if (framesRemaining == 0) return 0;
        return thisFrameScore(rolls)
               + score(getRemainingRolls(rolls), framesRemaining-1);
    }

    private Integer thisFrameScore(List<Integer> rolls) {
        Integer frameScore = 0;
        for (Integer roll : thisFramesRolls(rolls))
            frameScore += roll;
        return frameScore;
    }

    private List<Integer> thisFramesRolls(List<Integer> rolls) {
        return rolls.subList(0, numberOfRollsToScore(rolls));
    }

    private int numberOfRollsToScore(List<Integer> rolls) {
        return (isMark(rolls)) ? 3: 2;
    }

    private boolean isMark(List<Integer> rolls) {
        return isStrike(rolls) || isSpare(rolls);
    }

    private boolean isStrike(List<Integer> rolls) {
        return rolls.get(0) == 10;
    }

    private boolean isSpare(List<Integer> rolls) {
        return rolls.get(0) + rolls.get(1) == 10;
    }

    private List<Integer> getRemainingRolls(List<Integer> rolls) {
        return rolls.subList(frameSize(rolls), rolls.size());
    }

    private int frameSize(List<Integer> rolls) {
        return isStrike(rolls)? 1: 2;
    }
}
```



Version 5b – Java – domain-rich - Ver

While we are here, we may as well compare the level of **domain-richness** of **Ron Jeffries'** and **Robert Martin's** Java programs. Not too dissimilar.

```java
public class Game {

  private int rolls[] = new int[21];
  private int currentRoll = 0;

  private boolean isSpare(int frameIndex) {
    return rolls[frameIndex] + rolls[frameIndex + 1] == 10;
  }

  private boolean isStrike(int frameIndex) {
    return rolls[frameIndex] == 10;
  }

  private int spareBonus(int frameIndex) {
    return rolls[frameIndex + 2];
  }

  private int strikeBonus(int frameIndex) {
    return rolls[frameIndex + 1] + rolls[frameIndex + 2];
  }

  private int twoBallsInFrame(int frameIndex) {
    return rolls[frameIndex] + rolls[frameIndex + 1];
  }

  public void roll(int pins) {
    rolls[currentRoll++] = pins;
  }

  public int score() {
    int score = 0;
    int frameIndex = 0;
    for (int frame = 0; frame < 10; frame++) {
      if (isStrike(frameIndex)) {
        score += 10 + strikeBonus(frameIndex);
        frameIndex ++;
      } else if (isSpare(frameIndex)) {
        score += 10 + spareBonus(frameIndex);
        frameIndex += 2;
      } else {
        score += twoBallsInFrame(frameIndex);
        frameIndex += 2;
      }
    }
    return score;
  }
}
```

Version 1 – Java - Robert Martin in FD-PPP

By the way, when I labelled **Robert Martin**'s **Java program** with the **ten-ness** indicator shown on the right, I added to its **test suite** (reproduced on the next slide) the following four tests (shown on the slide after next):

- **Pit**'s two tests (the second one verifies **ten-ness**)
- Two of the tests found in **Ron Jeffries**' **Java program**

Also by the way, **Pit**'s **two tests**, added by **Ron Jeffries** to his 2006 **Java program**, are equivalent to **Jon Bettinger**'s **two tests**, added by **Ron Jeffries** to his 2010 **Scala program**.

Pit

```
@Test public void pitStrikeFinalFrame() throws Exception {
  assertEquals(15, game.score(new Integer[]
   { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3}));
}

@Test public void pitStrikeNinthFrame() throws Exception {
  assertEquals(20, game.score(new Integer[]
    { 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3}));
}
```

Pit's two tests

Jon Bettinger

```
it("should count not double count tenth frame bonus rolls") {
  expect(11)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,1,0)))
}

it("should count non-mark tenth frame after strike in ninth") {
    expect(12)(Game.score(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 1,0)))
}
```

Jon Bettinger's two tests

```java
import org.example.Game;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class BowlingTest {
  private Game g;

  @Before
  public void setUp() throws Exception {
    g = new Game();
  }

  private void rollMany(int n, int pins) {
    for (int i=0; i<n; i++) {
      g.roll(pins);
    }
  }

  private void rollSpare() {
    rollMany(2, 5);
  }

  private void rollStrike() {
    g.roll(10);
  }

 @Test
  public void gutterGame() throws Exception {
    rollMany(20, 0);
    assertEquals(0, g.score());
  }

@Test
public void allOnes() throws Exception {
  rollMany(20, 1);
  assertEquals(20, g.score());
}

@Test
public void oneSpare() throws Exception {
  rollSpare();
  g.roll(7);
  rollMany(17, 0);
  assertEquals(24, g.score());
}

@Test
public void oneStrike() throws Exception {
  rollStrike();
  g.roll(2);
  g.roll(3);
  rollMany(16, 0);
  assertEquals(20, g.score());
}

@Test
public void perfectGame() throws Exception {
  rollMany(12, 10);
  assertEquals(300, g.score());
}
```

```java
@Test
public void alternatingStrikeSpare() throws Exception {

  rollStrike(); rollSpare();
  rollStrike(); rollSpare();
  rollStrike(); rollSpare();
  rollStrike(); rollSpare();
  rollStrike(); rollSpare();

  g.roll(10);

  assertEquals(200, g.score());
}
```

```java
@Test
public void alternatingSpareStrike() throws Exception {

  rollSpare(); rollStrike();
  rollSpare(); rollStrike();
  rollSpare(); rollStrike();
  rollSpare(); rollStrike();
  rollSpare(); rollStrike();

  g.roll(5);
  g.roll(5);

  assertEquals(200, g.score());
}
```

```java
@Test
 public void pitStrikeFinalFrame() throws Exception {

  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);
  rollMany(2,0);

  rollStrike();

  g.roll(2);
  g.roll(3);

  assertEquals(15, g.score());
 }
```

```java
@Test
public void pitStrikeNinthFrame() throws Exception {

  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);
  rollMany(2,0); rollMany(2,0);

  rollStrike();

  g.roll(2);
  g.roll(3);

  assertEquals(20, g.score());
 }
```

To **conclude** this **series**' **first deck**, let's finally have a look at **Tom Moertel**'s **Haskell program**, which was shown at the beginning of the deck, and whose **test suite** we **reused**, but whose **code** we **skipped**, preferring to **postpone** its **examination**.

Tom Moertel

𝕏@tmoertel

```haskell
-- | Compute the score for the list of rolls 'rs'

score rs = sc 0 1 rs

-- accumulate the score 's' and frame count 'f' while consuming a
-- list of rolls 'rs' one frame at a time

sc s 11 _  = s           -- frame 11 means all done; return score
sc s f rs  = case rs of  -- otherwise, consume the frame & recurse
    10:rs'                -> sc' 3 rs'  -- strike
    x:y:rs' | x + y == 10 -> sc' 3 rs'  -- spare
            | otherwise   -> sc' 2 rs'  -- normal
    _                     -> error "ill-formed sequence of rolls"
  where
    -- accumulate the next 'n' rolls into the score and recurse
    sc' n rs' = sc (s + sum (take n rs)) (f + 1) rs'
```

In my **humble opinion**, the **program**'s **`golfed` style** makes it **harder** to **understand**, so here is a slightly **`ungolfed` version** in which I have **renamed** its **functions**, **function parameters**, et cetera, so that they are **more intention-revealing**. Hopefully you'll agree that the **resulting program** is **easy to understand**.

See how the **scoreRolls function** maintains a **frameCount** that **begins** at **one** and is **incremented ten times**? That's how the **program represents** the **ten-ness** of the **game**.


```haskell
-- | Compute the score for the list of rolls 'rs'

score rolls = scoreRolls 0 1 rolls

-- accumulate the score 's' and frame count 'f' while consuming a
-- list of rolls 'rs' one frame at a time

scoreRolls score 11 _              = score          -- frame 11 means all done; return score
scoreRolls score frameCount rolls = case rolls of  -- otherwise, consume the frame & recurse
    10:restOfRolls                 -> scoreNextFrameAndOtherRolls 3 restOfRolls -- strike
    x:y:restOfRolls | x + y == 10  -> scoreNextFrameAndOtherRolls 3 restOfRolls -- spare
                    | otherwise    -> scoreNextFrameAndOtherRolls 2 restOfRolls -- normal
    _                              -> error "ill-formed sequence of rolls"
  where
    -- accumulate the next 'n' rolls into the score and recurse
    scoreNextFrameAndOtherRolls frameRollsCount restOfRolls =
      scoreRolls (score + sum (take frameRollsCount rolls)) (frameCount + 1) restOfRolls
```

Version 6b - **Haskell** - **Tom Moertel**'s version but slightly **ungolfed** by **Philip Schwarz**

Here is **Tom Moertel**'s **test suite** again, but this time I have added to it the following:

- The tests seen in **Robert Martin**'s **Java** version of the program
- Two tests seen in **Ron Jeffries**' **Java** version of the program
- **Pit**'s two tests (added by **Ron Jeffries** to his corrected **Java** program), the second of which verifies **ten-ness**

```
import Test.HUnit

tests = test
    [ "gutters"       ~: score  (rep 20  0)          ~?=   0
    , "ones"          ~: score  (rep 20  1)          ~?=  20
    , "fives"         ~: score  (rep 22  5)          ~?= 150
    , "strikes"       ~: score  (rep 12 10)          ~?= 300
    , "1 + gutters"   ~: score  (1 : rep 19 0)       ~?=   1
    , "first spare"   ~: score  (5:5:5 : rep 17 0)   ~?=  20
    , "first strike"  ~: score  (10:5:5 : rep 17 0)  ~?=  30
    , "last spare"    ~: rscore (5:5:5 : rep 18 0)   ~?=  15
    , "last strike"   ~: rscore (5:5:10 : rep 18 0)  ~?=  20

    , "gutter game"  ~: score (rep 20  0)            ~?=   0
    , "all ones"     ~: score (rep 20  1)            ~?=   20
    , "one spare"    ~: score (5:5:7 : rep 17  0)    ~?=   24
    , "one strike"   ~: score (10:2:3 : rep 16  0)   ~?=   20
    , "perfect-game" ~: score (rep 12  10)           ~?=   300
    , "alternating strike spare" ~: score [10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10] ~?= 200
    , "alternating spare strike" ~: score [5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5] ~?= 200
    , "strike final frame"       ~: score [0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3] ~?= 15
    , "strike ninth frame"       ~: score [0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3] ~?= 20

    ]
  where
    rep    = replicate
    rscore = score . reverse  -- reverse list and then score it
```

Robert Martin

Ron Jeffries

Pit

That’s all for part one.

I hope you liked it.

See you in part two.