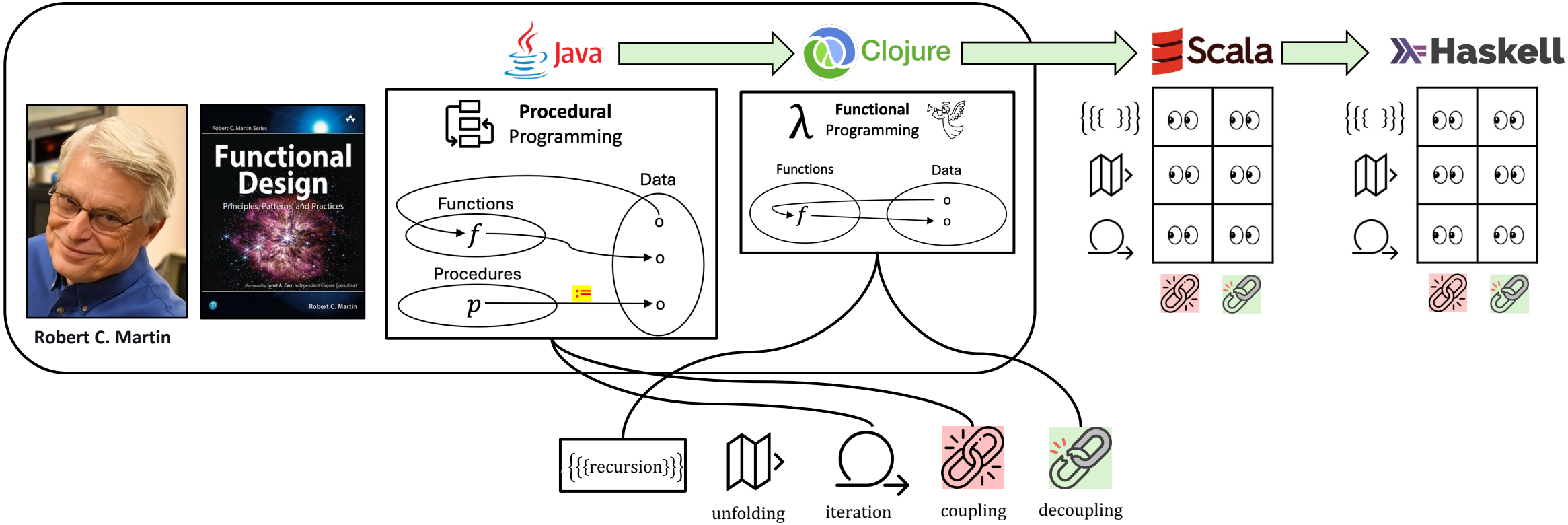


The Bowling Game
From Imperative to Functional Programming

Part 4



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

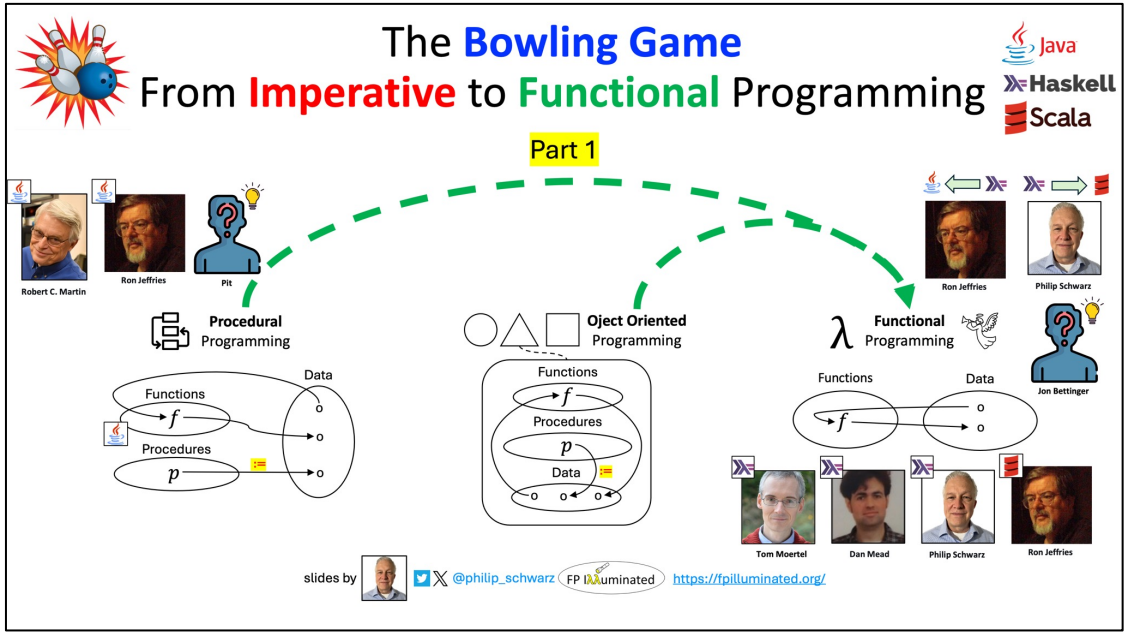
Welcome to **part 4** of this **series** of **decks**.

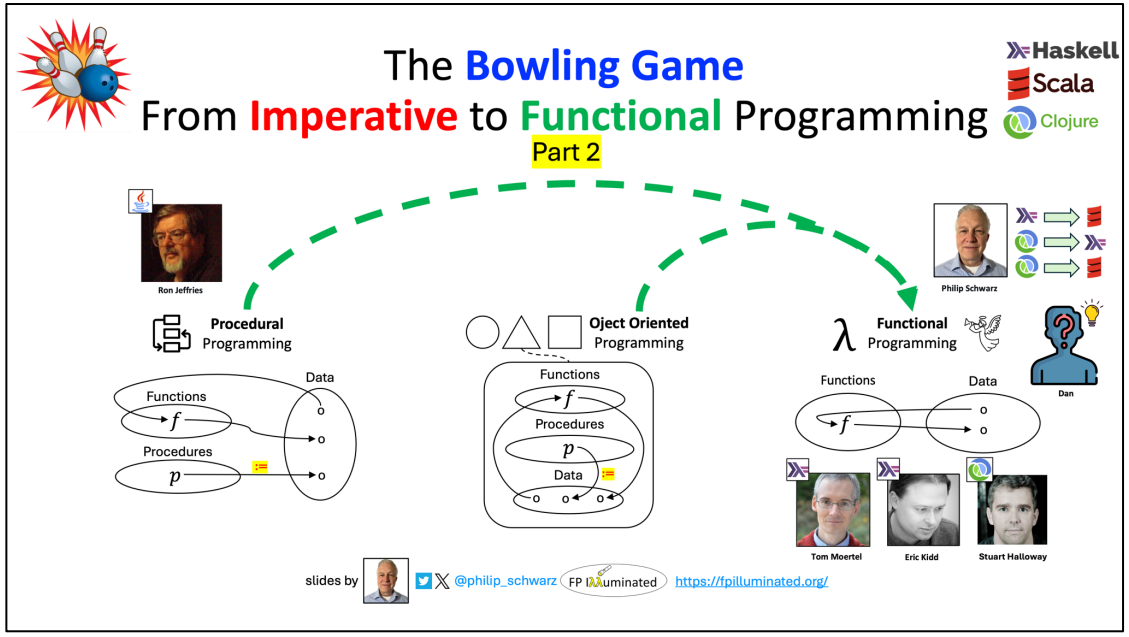


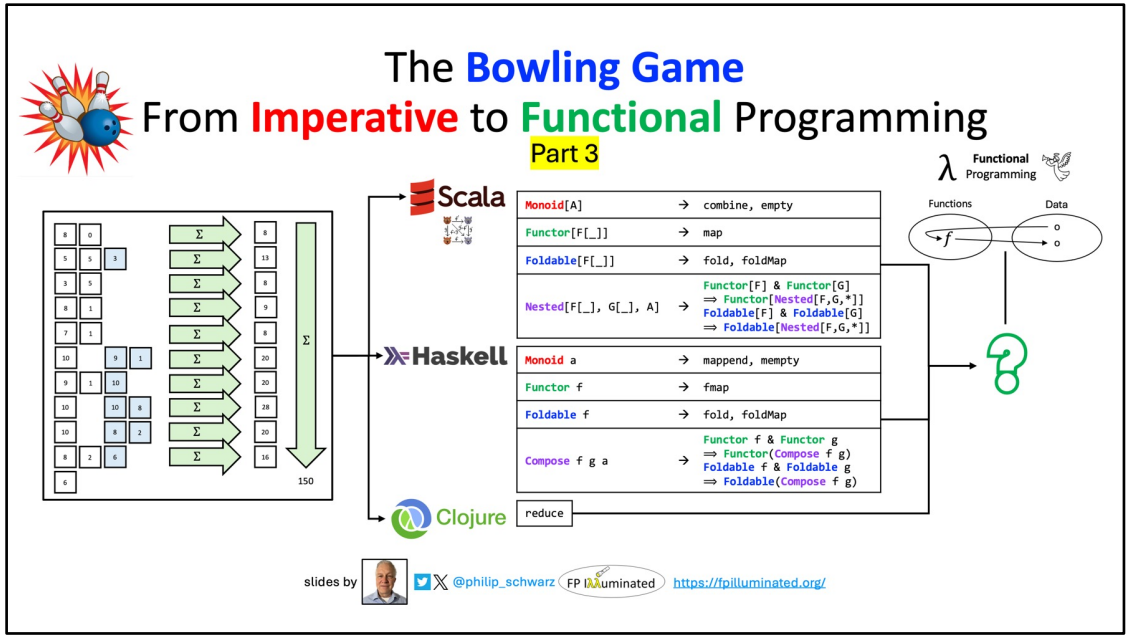
@philip_schwarz

FP IλλuminatedFP Illuminated

https://fpilluminated.org







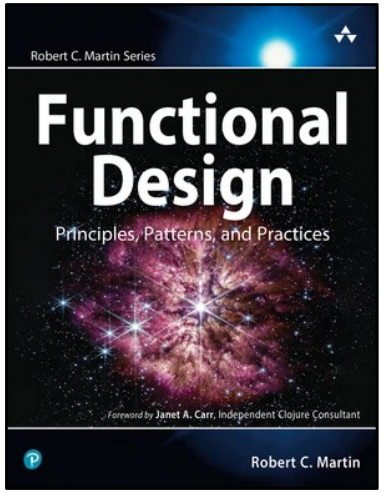


In **part 1** we mentioned that in **Functional Design - Principles, Patterns, and Practices** (**FD-PPP**) there is a chapter in which **Robert Martin** does the **Bowling Game kata**, first in **Java** and then in **Clojure**.

We then looked at the resulting **Java program**, which can be seen again on the next slide.

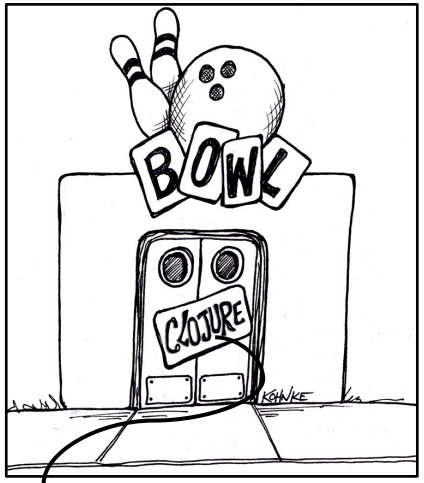
Let's now see how **Robert Martin** arrives at his **Clojure program**.

Chapter 7
Bowling Game





Robert C. Martin



Clojure

```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;
  }

}
```



Things start out quite differently in **Clojure**. We have **no classes** to create, and there is **no need** for a **roll method**. So our **first test** is the **gutter game**:

```
(should= 0 (score (repeat[2] 20 0)))

(defn score [rolls] 0)
```

2. The **repeat function** returns a **sequence** of **repeating values**. In this case, it is a sequence of 20 zeros.

Followed quickly by all ones:

```
(should= 20 (score (repeat 20 1)))

(defn score [rolls]
  (reduce[3] + rolls))
```

3. You will want to **look this function up**. It does much more than this paragraph suggests. But you'll see that soon enough.



No surprises here. The **reduce function** simply applies the **+ function** across the **entire list**. So our **next test** is **one spare**:

```
(should= 24 (score (concat [5 5 7] (repeat 17 0)))))
```

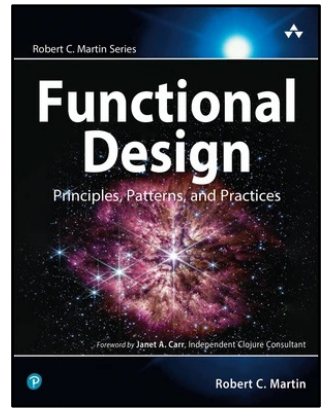
To make this **pass**, we go through **several steps**. The **first** is to **break** the **rolls array up into frames** and **sum up** the **frames**. At first, we assume that **frames** have just **two rolls**:

```
(defn to-frames [rolls]
  (partition[4] 2 rolls))

(defn add-frame [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frame 0 (to-frames rolls)))
```

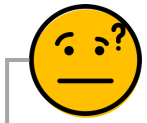
4. The **partition function breaks** the **rolls list** into a **list** of **pairs**. So `[1 2 3 4 5 6]` becomes `[[1 2][3 4][5 6]]`.





Now the **reduce function** has come into its own. It **cycles** through the **pairs** of **rolls**, **accumulating** them into a **score**.

This **change** keeps all the **previous tests passing**, but it **still fails** the **spare test**. To **pass** that we have to **add special processing** to the **to-frames** and **add-frame** functions. Our **goal** is to **put** all the **rolls** needed to **calculate** a **frame** into the **frame data**.



```
(defn to-frames [rolls]
  (let [frames (partition 2 rolls)
        possible-bonuses (map #(take 1 %)[5] (rest frames))
        possible-bonuses[6] (concat[7] possible-bonuses [[0]])]
    (map concat frames possible-bonuses)))

(defn add-frame [score frame-and-bonus]
  (let [frame (take 2 frame-and-bonus)]
    (if (= 10 (reduce + frame))
      (+ score (reduce + frame-and-bonus))
      (+ score (reduce + frame)))))

(defn score [rolls]
  (reduce add-frame 0 (to-frames rolls)))
```

5. The **#(...)** form creates an **anonymous function**. The **% symbol** is the **argument** to that function. You can also use **%n**, where **n** is an integer representing the **nth argument**. So **#(take 1 %)** is a function that returns a list containing the first element of its argument.

6. This is **not a reassignment**, or even a **reinitialization**. The second possible-bonuses value is **distinct** from the first. Think of it like a **local variable** in Java **hiding** a function argument or a member variable of the same name.

7. The **concat function** concatenates lists together. So (concat [1 2] [3 4]) returns [1 2 3 4].

Look closely at this code. There are **lots** of **little tricks** and **workarounds** in it. Why? Because **Clojure** is **full** of **lots** of **lovely**, **tempting little tools** that you can use to get data into almost the form you want, and then use **little tricks** to maneuver the data into exactly the form you want. If you aren't **careful**, those **little tricks** can start to **dominate** the **code**.

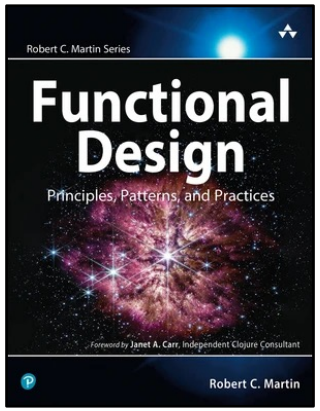
So, for example, see if you can figure out **why** I am passing [[0]] into the **concat** function in **to-frames**.[8] As another example, ask yourself **why** I used **#(take 1 %)** instead of just **first**.[9]

8. Since **bonuses** are based on the next frame, **possible-bonuses** had one too few elements. That would have terminated the final call to **map** one element too early.

9. **(take 1 x)** returns a list containing the first element in **x**. **first** returns the first element.

Because of the **trickiness** in this **code**, don't be too concerned if you are **struggling** to **understand** it. I **struggled** too when looking back over it. And so...





...when these **little tricks proliferate** it's time to **rethink** the **solution**.



So I **refactored** the **solution** into a **simple loop**:

```
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))
      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

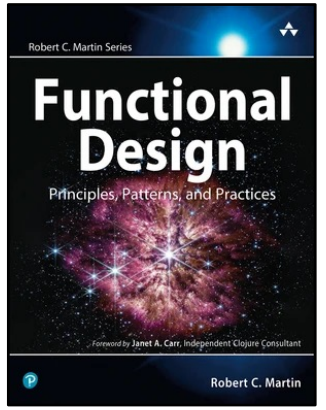
(defn score [rolls]
  (reduce add-frames 0 (to-frames rolls)))
```

① ② ③

{{{recursion}}}

This is **looking a lot better**. Moreover, it's starting to look **a bit like** the **Java solution**.





① ② see next two slides for details of **loop** and **recur**

③ see slide after that for details of the **conj** function

In general, Clojure allows 3 ways to recur:

1. A call to the function from within the function itself. No loop or recur is used in this case. The recursive call can appear anywhere, not just as last instruction, like in this example returning the n-th element in the Fibonacci series :

```
(defn fib [n]
  (if (<= 0 n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2))))) ①

(map fib (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

① Note that despite the line is the last in a mutually exclusive if branch, the first fib invocation is not in tail position, because there is a second (- n 2) evaluation that follows.

2. The recur special form invoked without a loop target will use the innermost containing function definition. Here's is the same example re-written to avoid recurring twice:

```
(defn fib [a b cnt]
  (if (zero? cnt)
    b
    (recur (+ a b) a (dec cnt))))

(map (partial fib 1 0) (range 10)) ①
;; (0 1 1 2 3 5 8 13 21 34)
```

① recur target here is the top function definition fib, because there is no other inner-most definition and there is no loop instruction either.

3. The recur special form invoked without a loop target will restart computation from the innermost loop form. The following is probably the most effective of the 3 fib versions presented so far. The presence of loop takes care of some initialization that doesn't need to be outside the function and at the same time we are not recurring twice:

```
(defn fib [n]
  (loop [a 1 b 0 cnt n] ①
    (if (zero? cnt)
      b
      (recur (+ a b) a (dec cnt)))))

(map fib (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

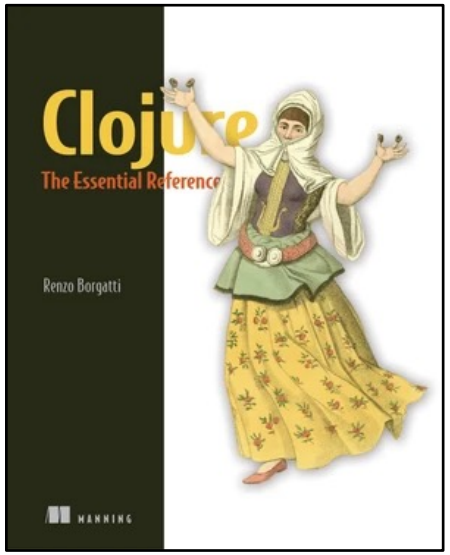
① Note that 0 and 1 initialization parameters had to be sent in when invoking the function. They are now handled by loop without being exported in the function parameters.



Renzo Borgatti



reborg





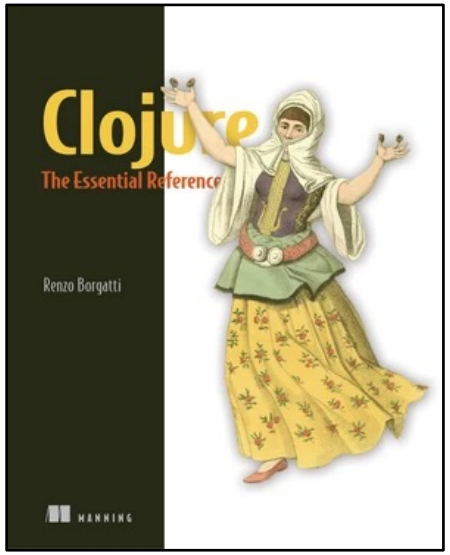
Rachel Bowyer



rachbowyer

These are the **main differences** between **calling the function recursively** and **using** the **recur variants**:

- The **compiler ensures** that if **recur** is present, it needs to be in **tail position**. The compiler throws exception if **recur** is in **not** in **tail position**.
- The **compiler enables** a **special optimizatio**n when using **recur** that **doesn't consume** the **stack** (see the tail-recursion section below for a detailed explanation).
- **loop** offers **additional control over local bindings** without interfering with the function arguments, for example to **initialize values** or to **add additional parameters** to the **recursion**.
- **loop** is also the **main choice for iteration when speed is important**. **loop** takes care of propagating type information to avoid unnecessary boxing/unboxing steps, which is usually an important factor for fast code execution[64]. We are going to illustrate additional aspects about **loop-recur** speed further down the section.





Rachel Bowyer



rachbowyer



Renzo Borgatti



reborg



The **to-frames** function on the previous slide **grows** the **frames collection** by repeatedly using the **conj** function to add a new **list** of **rolls** to the **collection**. e.g. `(conj frames (take 3 remaining-rolls))`.

The **behaviour** of the **conj** function depends on the **collection type** that it operates on. While it **adds elements** to the **front** of a **list**, it **adds** them at the **end** of a **vector**.

The **initial value** of the **frames collection** in **to-frames** is `[ ]`, whose type may **incorrectly** be assumed to be that of a **list**, due to an **inaccuracy** in the following **footnotes** accompanying **Robert Martin**'s program:

4. The **partition function breaks** the **rolls list** into a **list** of **pairs**. So `[1 2 3 4 5 6]` becomes `[[1 2][3 4][5 6]]`.
5. The **concat function** concatenates lists together. So `(concat [1 2] [3 4])` returns `[1 2 3 4]`.

The **problem** is that while the two **footnotes** speak of **lists**, the examples they provide contain **vectors**. That's because in **Clojure**, **lists** are delimited by **round brackets**, whereas **vectors** are delimited by **square brackets**.

Here are examples of using **conj** with a **list** and with a **vector**:

```
(conj '(1 2) 3) → (3 1 2))

(conj  [1 2] 3) → [1 2 3])
```



Given that the **initial value** of the **frames collection** in the **to-frames** function is an **empty vector**, rather than an **empty list**, we conclude that the **to-frames** function **adds** each new **frame** at the **back** of the **frames vector**. The **correctness** of the **program** depends on it: **adding frames** to the **front** of a **frames list** would make the program **incorrect**, in that when the time came for the program to **enforce ten-ness**, it would be **incorrectly `taking`** the **last ten frames**, rather than **correctly `taking`** the **first ten frames**.

The **next test** is **one strike**:

```
(should= 20 (score (concat [10 2 3] (repeat 16 0)))))
```

And we make that **pass** by **adding one more case** to the **cond**:

```
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

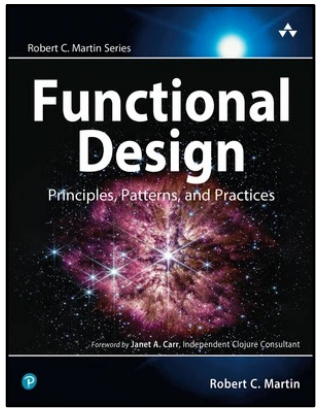
(defn add-frames [score frame]
  (+ score (reduce + frame)))

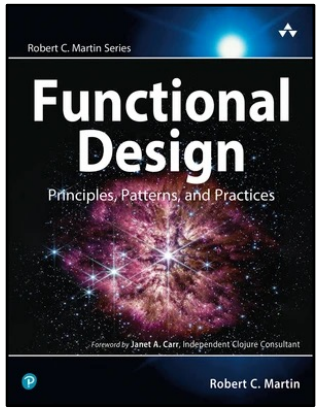
(defn score [rolls]
  (reduce add-frames 0 (to-frames rolls)))
```

Trivial, right?

{{{recursion}}}







So all that's left is the **perfect game**. And if this goes like the **Java version**, this **test** should just **pass** without modification:

```
(should= 300 (score (repeat 12 10))))
```

But it doesn't! Can you see why? Perhaps the **fix** will elucidate that for you:

```
(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

The **to-frames** function happily **creates** more than **ten frames**. It just runs to the end of the **rolls list** making as many **frames** as it can. But a **game** of **bowling** is **only ten frames**.







Here is how **Robert Martin**'s **Clojure program** looks next to a **refactored version** of his **Java program** in which functions invoked by the **score method** have been **inlined**. As **Martin** says, it does look **a bit like** his **Java program**.

Also, how would the **Clojure program** look if we made it **more domain-rich** by **extracting** some of the **logic** of the **to-frames function** into **subordinate functions** named after **domain concepts**? See the **next two slides** for how such a **refactored program** looks next to **Martin**'s original **Java program**.

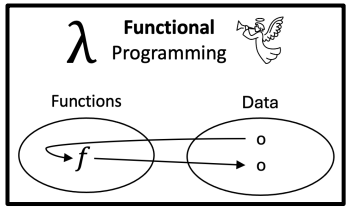
```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))
      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

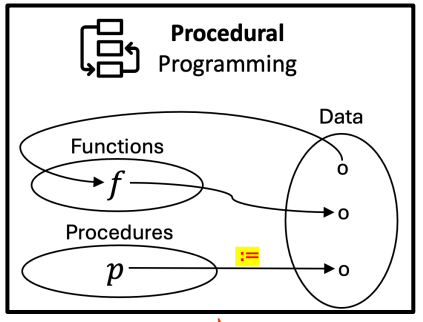


Version 9 - **Clojure** – **Robert Martin**

```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 1 - **Java** - **Robert Martin** in FD-PPP —— **REFACTORED**



```clojure
(defn nextFrameIsSpare [rolls]
  (= 10 (reduce + (take 2 rolls))))

(defn nextFrameIsStrike [rolls]
  (= 10 (first rolls)))

(defn takeSpareBonusRolls [rolls]
  (take 1 (drop 2 rolls)))

(defn takeStrikeBonusRolls [rolls]
  (take 2 (rest rolls)))

(defn takeRollsInOpenFrame [rolls]
  (take 2 rolls))

(defn dropStrikeFrameRolls [rolls]
  (rest rolls))

(defn dropSpareFrameRolls [rolls]
  (drop 2 rolls))
```

REFACTORED

Version 9 - Clojure – Robert Martin

```java
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];
}
```



ORIGINAL

Version 1 - Java - Robert Martin in FD-PPP

```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (nextFrameIsStrike remaining-rolls)
      (recur (dropStrikeFrameRolls remaining-rolls)
             (conj frames
                   (cons 10 (takeStrikeBonusRolls remaining-rolls))))

      (nextFrameIsSpare remaining-rolls)
      (recur (dropSpareFrameRolls remaining-rolls)
             (conj frames
                   (cons 10 (takeSpareBonusRolls remaining-rolls))))

      :else ; next frame is open
      (recur (dropOpenFrameRolls remaining-rolls)
             (conj frames
                   (takeRollsInOpenFrame remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```



REFACTORED

Version 9 - Clojure – Robert Martin

```java
public class Game {
  private int rolls[] = new int[21];
  private int currentRoll = 0;

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

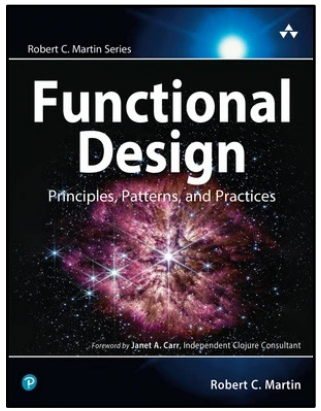
  public int score() {
    int score = 0;
    int frameIndex = 0;
    for (int frame = 0; frame < 10; frame++) {
      if (isStrike(frameIndex)) {
        score += 10 + strikeBonus(frameIndex);
        frameIndex ++;
      } else if (isSpare(frameIndex)) {
        score += 10 + spareBonus(frameIndex);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

}
```



ORIGINAL

Version 1 - Java - Robert Martin in FD-PPP



There are **quite** a **few interesting differences** between the **Java** and **Clojure versions** of this problem.

First, the **Clojure** version has **no Game class**. So all the **machinations** we used to create that **class** in the **Java** version simply don't occur in the **Clojure** version.

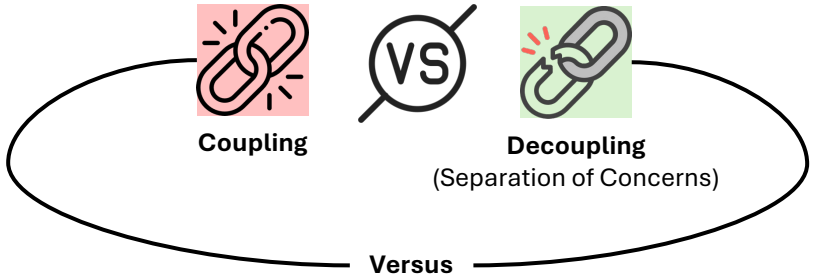
You might think that the loss of the **Game class** is a **weakness** of the **Clojure version**. After all, it's **convenient** to be able to just create a **Game**, toss it a bunch of **rolls**, and then get the **score**.

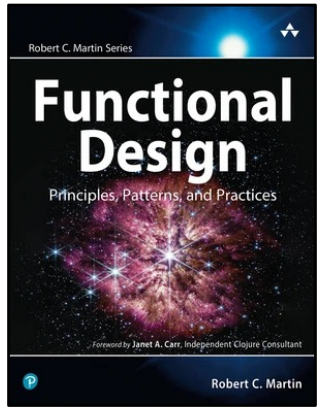
However, the **Clojure version** has **decoupled** the **accumulation** of the **rolls** from the **computation** of the **score**. Those **concepts** are **not bound together** in the **Clojure version**. And that makes me think that the **Java version** has a **subtle violation** of the **Single Responsibility Principle**.[10]

10 See Robert C. Martin, ***Clean Architecture*** (Pearson, 2017).

Second, as we tried to solve the **one spare case**, we saw how the **Clojure version** got **polluted** with all those **nasty little tricks**. This is a real problem with **Clojure** programs (or perhaps **Clojure** programmers). It's just too easy to add one more **nasty little trick** to get things to work.







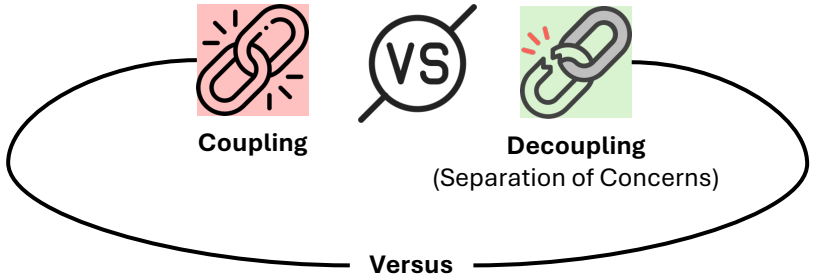
Third, the Clojure solution is significantly different from the Java solution. Oh, there are some points of similarity, to be sure. That cond structure in the Clojure version is very reminiscent of the if/else structure in the Java version. However, those two similar structures produced radically different results. The Java version produced the score. The Clojure version produced a frame that included the bonus balls for spares and strikes.

This is an interesting separation of concerns. It is a fact that computing the score forces both versions to identify all the rolls that impact each frame. However, the Java version does this in situ, whereas the Clojure version nicely separates those two concerns.

Which of these versions is better? The Java version ended up a bit simpler than the Clojure version; but it was also a bit more coupled. The separation of concerns in the Clojure version convinces me that between the two, it would be more flexible and useful.

But, of course, we are only talking about a dozen lines of code.







Decoupling
(Separation of Concerns)



```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))
      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

Mapping of rolls to frames

Computation of score

Version 9 - **Clojure** – **Robert Martin**



Coupling



```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;
  }
}
```

Mapping of rolls to frames and computation of score

Version 1 - **Java** - **Robert Martin** in FD-PPP —— **REFACTORED**



Here we have a go at writing the **Scala equivalent** of **Robert Martin**'s **Clojure program**.

Since **appending** an **item** to a **list** is **more costly** than **prefixing** the **list** with the **item**, in the **Scala program**, instead of **adding new frames** at the **end** of a **frames list**, we **add them** at the **front** of the **frame list**, and later **reverse** the **frames list**.

{{{recursion}}}

```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

```scala
type Roll = Int
type Frame = List[Roll]
```

```scala
def toFrames(
  remainingRolls : List[Roll],
  frames: List[Frame] = Nil
): List[Frame] =

    if remainingRolls.isEmpty
    then frames.reverse

    else if remainingRolls.head == 10
    then toFrames(remainingRolls.tail,
                  remainingRolls.take(3)::frames)

    else if remainingRolls.take(2).sum == 10
    then toFrames(remainingRolls.drop(2),
                  remainingRolls.take(3)::frames)

    else toFrames(remainingRolls.drop(2),
                  remainingRolls.take(2)::frames)

def addFrames(score: Int, frame: Frame): Int =
  score + frame.sum

def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).foldLeft(0)(addFrames)
```

```scala
if remainingRolls.isEmpty
then frames.reverse

else if remainingRolls.head == 10
then toFrames(remainingRolls.tail,
              remainingRolls.take(3)::frames)

else if remainingRolls.take(2).sum == 10
then toFrames(remainingRolls.drop(2),
              remainingRolls.take(3)::frames)

else toFrames(remainingRolls.drop(2),
              remainingRolls.take(2)::frames)
```

Let's **refactor** the body of the **Scala to-frames** function so that it exploits **pattern matching**.

In our programs in part one, **Dan Mead** and I named the **current three rolls x**, **y** and **z**.

In our **first refactoring** below, we name the **rolls** the way **Eric Kidd** did in his program in part two: **x1**, **y1**, **y2** for a **strike**, **x1**, **x2**, **y1** for a **spare**, and **x1**, **x2**, for an **open frame**. On the one hand, this **naming scheme** is preferable in that it closely models the **domain logic**: rolls **x1** and **x2** belong to the **current frame**, whereas rolls **y1** and **y2** are **bonus rolls** belonging to the **next frame**. On the other hand, it could be argued that the scheme is **overcomplex**, that a **single letter scheme** is **good enough**.

In the **second refactoring**, we revert to a **single letter scheme**, except that we use names **a**, **b** and **c**, because I believe they have a **simplifying**, **calming effect** on the **reader**, whereas I believe **x**, **y** and **z** to have the **opposite effect**.



```scala
remainingRolls match
  case Nil | List(_) => frames.reverse
  case x1::y1::y2::rs if x == 10       => toFrames(y1::y2::rs, List(x1,y1,y2)::frames)
  case x1::x2::y1::rs if x1 + x2 == 10 => toFrames(    y1::rs, List(x1,x2,y1)::frames)
  case     x1::x2::rs                  => toFrames(        rs,    List(x1,x2)::frames)
```

```scala
remainingRolls match
  case Nil | List(_) => frames.reverse
  case a::b::c::rs if a == 10     => toFrames(b::c::rs, List(a,b,c)::frames)
  case a::b::c::rs if a + b == 10 => toFrames(   c::rs, List(a,b,c)::frames)
  case a::b::rs                   => toFrames(      rs, List(a,b)::frames)
```



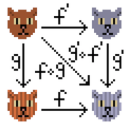
Let's also **refactor** the **logic** that **computes the score**, by exploiting the **simplification** that we came up with in **part 3**.

```
def addFrames(score: Int, frame: Frame): Int =
  score + frame.sum

def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).foldLeft(0)(addFrames)
```



```
def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).nested.sumAll
```



```
import cats.implicits.*
```



The next slide shows **the wole Scala program after the refactoring** seen on this slide and the preceding one.

Version 9 - Clojure – Robert Martin

```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

```scala
type Roll = Int
type Frame = List[Roll]
```

```scala
import cats.implicits.*
```



```scala
def toFrames(
  remainingRolls : List[Roll],
  frames: List[Frame] = Nil
): List[Frame] =

  remainingRolls match
    case Nil | List(_) => frames.reverse
    case a::b::c::rs if a == 10     => toFrames(b::c::rs, List(a,b,c)::frames)
    case a::b::c::rs if a + b == 10 => toFrames(   c::rs, List(a,b,c)::frames)
    case a::b::rs                   => toFrames(      rs, List(a,b)::frames)

def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).nested.sumAll
```



REFACTORED

On the next slide we translate the Scala program into Haskell.



{{{recursion}}}

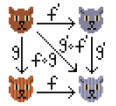
{{{recursion}}}

```scala
def toFrames(remainingRolls: List[Roll], frames: List[Frame] = Nil): List[Frame] =
  remainingRolls match
    case Nil | List(_) => frames.reverse
    case a::b::c::rs if a == 10     => toFrames(b::c::rs, List(a,b,c)::frames)
    case a::b::c::rs if a + b == 10 => toFrames(   c::rs, List(a,b,c)::frames)
    case a::b::rs                   => toFrames(      rs, List(a,b)::frames)

def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).nested.sumAll
```

```scala
type Roll = Int
type Frame = List[Roll]
```

```scala
import cats.implicits.*
```



Scala translation of 'Version 9 - **Clojure** – **Robert Martin**'

```haskell
frames :: [Roll] -> [Frame]
frames rolls = toFrames rolls []
  where toFrames rs         fs | length rs < 2 = reverse fs
        toFrames (a:b:c:rs) fs | a == 10       = toFrames (b:c:rs) ([a,b,c]:fs)
        toFrames (a:b:c:rs) fs | a + b == 10   = toFrames   (c:rs) ([a,b,c]:fs)
        toFrames (a:b:rs)   fs                 = toFrames       rs   ([a,b]:fs)

score :: [Roll] -> Int
score rolls = sum (Compose (take 10 (frames rolls)))
```

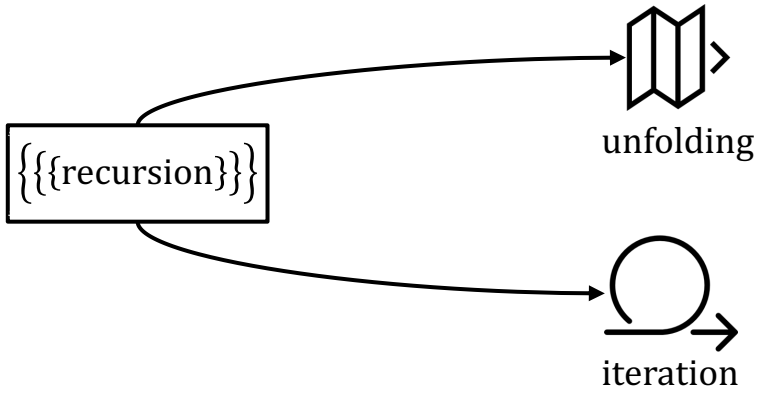
```haskell
type Roll = Int
type Frame = [Roll]
```

```haskell
import Data.Functor.Compose
```

Haskell translation of 'Version 9 - **Clojure** – **Robert Martin**'



In the next three slides we **explore replacing recursion** with **unfolding** and **iteration**, first in **Scala** and then in **Haskell**.



```scala
def toFrames(remainingRolls : List[Roll], frames: List[Frame] = Nil): List[Frame] =
  remainingRolls match
    case Nil | List(_) => frames.reverse
    case a::b::c::rs if a == 10   => toFrames(b::c::rs, List(a,b,c)::frames)
    case a::b::c::rs if a+b == 10 => toFrames(   c::rs, List(a,b,c)::frames)
    case a::b::rs                 => toFrames(      rs, List(a,b)::frames)

def score(rolls: List[Roll]): Int =
  toFrames(rolls).take(10).nested.sumAll
```

1..10 ten-ness

{{{recursion}}}

```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.unfold(rolls,0) : (rolls,frames) =>
    if frames == 10 then None
    else rolls match
      case a::b::c::rs if a == 10     => Some(List(a,b,c),(b::c::rs,frames+1))
      case a::b::c::rs if a + b == 10 => Some(List(a,b,c),   (c::rs,frames+1))
      case a::b::rs                   => Some(List(a,b),        (rs,frames+1))

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```

1..10 ten-ness

$$unfold :: (\beta \to Maybe\ (\alpha, \beta)) \to \beta \to List\ \alpha$$
$$unfold\ f\ u = \textbf{case}\ f\ u\ \textbf{of}$$
$$\quad Nothing \to Nil$$
$$\quad Just\ (x, v) \to Cons\ x\ (unfold\ f\ v)$$



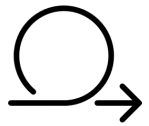
unfolding

```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.iterate((rolls, List.empty[Roll]), 11) {
    case (a::b::c::rs,_) if a == 10   => (b::c::rs, List(a,b,c))
    case (a::b::c::rs,_) if a+b == 10 =>    (c::rs, List(a,b,c))
    case (a::b::rs,_)                 =>       (rs, List(a,b))
  }.map(_.last).tail

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```

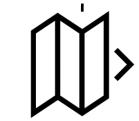
1..10 ten-ness

$$iterate :: (\alpha \to \alpha) \to \alpha \to [\alpha]$$
$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$



iteration

iterating and unfolding
variants of **Scala** translation of
'Version 9 - **Clojure** – **Robert Martin**'



unfolding

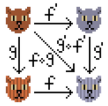
```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.unfold(rolls,0) : (rolls,frames) =>
    if frames == 10 then None
    else rolls match
      case a::b::c::rs if a == 10     => Some(List(a,b,c), (b::c::rs,frames+1))
      case a::b::c::rs if a + b == 10 => Some(List(a,b,c),    (c::rs,frames+1))
      case a::b::rs                   => Some(List(a,b),         (rs,frames+1))

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```



```scala
type Roll = Int
type Frame = List[Roll]
```

```scala
import cats.implicits.*
```





unfolding variant of **Scala** translation of
'Version 9 - **Clojure** – **Robert Martin**'

```haskell
toFrames :: [Roll] -> [Frame]
toFrames rolls = unfoldr (\case
    (_,        fs) | fs == 10     -> Nothing
    (a:b:c:rs, fs) | a == 10      -> Just ([a,b,c], (b:c:rs, fs+1))
    (a:b:c:rs, fs) | a + b == 10 -> Just ([a,b,c],   (c:rs, fs+1))
    (a:b:rs,   fs)               -> Just (  [a,b],     (rs, fs+1))
  )(rolls, 0)

score :: [Roll] -> Int
score rolls = sum (Compose (toFrames rolls))
```



```haskell
type Roll = Int
type Frame = [Roll]
```

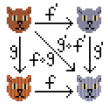
```haskell
{-# LANGUAGE LambdaCase #-}
import Data.Functor.Compose
import Data.List (unfoldr)
```

```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.iterate((rolls, List.empty[Roll]), 11) {
    case (a::b::c::rs,_) if a == 10   => (b::c::rs, List(a,b,c))
    case (a::b::c::rs,_) if a+b == 10 =>    (c::rs, List(a,b,c))
    case (a::b::rs,_)                 =>       (rs, List(a,b))
  }.tail.map(_.last)

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```

```scala
type Roll = Int
type Frame = List[Roll]
```

```scala
import cats.implicits.*
```



iterating variant of **Scala** translation of
'Version 9 - **Clojure** – **Robert Martin**'

iteration

```haskell
frames :: [Roll] -> [Frame]
frames rolls = map snd $ tail $ iterate (\case
   (a:b:c:rs, _) | a == 10     -> (b:c:rs, [a,b,c])
   (a:b:c:rs, _) | a + b == 10 -> (  c:rs, [a,b,c])
   (a:b:rs,   _)               -> (    rs,   [a,b])
  )(rolls, [])

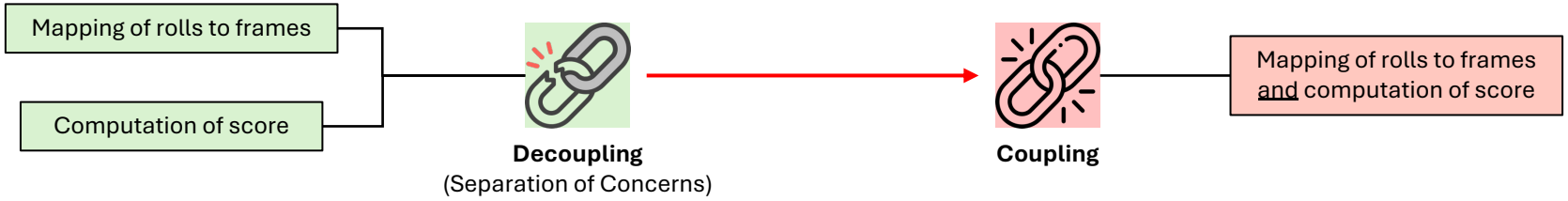
score :: [Roll] -> Int
score rolls = sum (Compose (take 10 (frames rolls)))
```

```haskell
type Roll = Int
type Frame = [Roll]
```

```haskell
{-# LANGUAGE LambdaCase #-}
import Data.Functor.Compose
Import Data.List (iterate)
```

And now, for completeness, let's also **explore** a **regression** from **decoupling** to **coupling**, first in **Scala**, and then in **Haskell**.





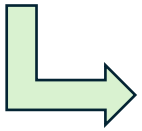
```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.unfold(rolls,0) : (rolls,frames) =>
    if frames == 10 then None
    else rolls match
      case a::b::c::rs if a == 10     => Some(List(a,b,c),(b::c::rs,frames+1))
      case a::b::c::rs if a + b == 10 => Some(List(a,b,c),   (c::rs,frames+1))
      case a::b::rs                   => Some(List(a,b),        (rs,frames+1))

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```



unfolding





```scala
def score(rolls: List[Roll]): Int =
  List.unfold(rolls, 0){
    case (_,           frame:10)              => None
    case (a::b::c::rs, frame   ) if a == 10   => Some(a+b+c, (b::c::rs, frame+1))
    case (a::b::c::rs, frame   ) if a+b == 10 => Some(a+b+c,    (c::rs, frame+1))
    case (a::b::rs,    frame   )              => Some(a+b,        (rs, frame+1))
  }.sum
```

coupled iterating and unfolding variants of **Scala** translation of 'Version 9 - **Clojure** – **Robert Martin**'

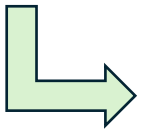
```scala
def toFrames(rolls: List[Roll]): List[Frame] =
  List.iterate((rolls, List.empty[Roll]), 11) {
    case (a::b::c::rs,_) if a == 10   => (b::c::rs, List(a,b,c))
    case (a::b::c::rs,_) if a+b == 10 =>    (c::rs, List(a,b,c))
    case (a::b::rs,_)                 =>       (rs, List(a,b))
  }.map(_.last).tail

def score(rolls: List[Roll]): Int =
  toFrames(rolls).nested.sumAll
```



iteration



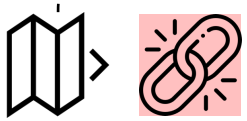


```scala
def score(rolls: List[Roll]): Int =
  List.iterate((rolls, 0), 11) {
    case (a::b::c::rs, _) if a == 10   => (b::c::rs, a+b+c)
    case (a::b::c::rs, _) if a+b == 10 =>    (c::rs, a+b+c)
    case (a::b::rs,    _)              =>       (rs, a+b)
  }.map(_.last).sum
```

```scala
def score(rolls: List[Roll]): Int =
  List.unfold(rolls, 0):
    case (_,           frame:10)             => None
    case (a::b::c::rs, frame   ) if a == 10  => Some(a+b+c, (b::c::rs, frame+1))
    case (a::b::c::rs, frame   ) if a+b == 10 => Some(a+b+c,    (c::rs, frame+1))
    case (a::b::rs,    frame   )             => Some(a+b,         (rs, frame+1))
  .sum
```

```scala
type Roll = Int
```

coupled unfolding
variant of **Scala** translation of
'Version 9 - **Clojure** – **Robert Martin**'



unfolding



```haskell
score :: [Roll] -> Int
score rolls = sum $ unfoldr (\case
    (_,        fs) | fs == 10     -> Nothing
    (a:b:c:rs, fs) | a == 10      -> Just (a+b+c, (b:c:rs, fs+1))
    (a:b:c:rs, fs) | a + b == 10 -> Just (a+b+c,   (c:rs, fs+1))
    (a:b:rs,   fs)               -> Just (  a+b,     (rs, fs+1))
  )(rolls, 0)
```

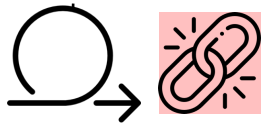
```haskell
type Roll = Int
```

```haskell
{-# LANGUAGE LambdaCase #-}
import Data.List (unfoldr)
```

```scala
def score(rolls: List[Roll]): Int =
  List.iterate((rolls, 0), 11) {
    case (a::b::c::rs, _) if a == 10   => (b::c::rs, a+b+c)
    case (a::b::c::rs, _) if a+b == 10 =>    (c::rs, a+b+c)
    case (a::b::rs,    _)              =>       (rs, a+b  )
  }.map(_.last).sum
```

```scala
type Roll = Int
```

coupled iterating
variant of **Scala** translation of
'Version 9 - **Clojure** – **Robert Martin**'





```haskell
score :: [Roll] -> Int
score rolls = sum $ take 10 $ map snd $ tail $ iterate (\case
    (a:b:c:rs, _) | a == 10     -> (b:c:rs, a+b+c)
    (a:b:c:rs, _) | a + b == 10 -> (  c:rs, a+b+c)
    (a:b:rs,   _)               -> (    rs,  a+b )
  )(rolls, 0)
```

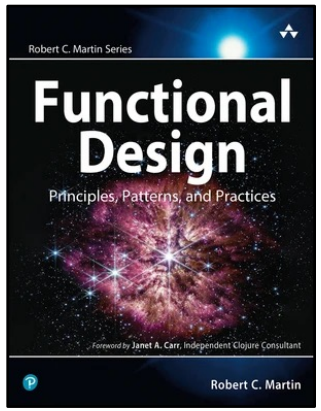
```haskell
type Roll = Int
```

```haskell
{-# LANGUAGE LambdaCase #-}
import Data.List (unfoldr)
import Data.List (iterate)
```

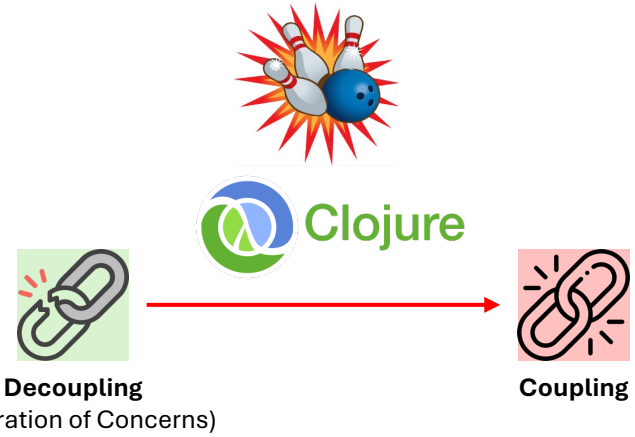


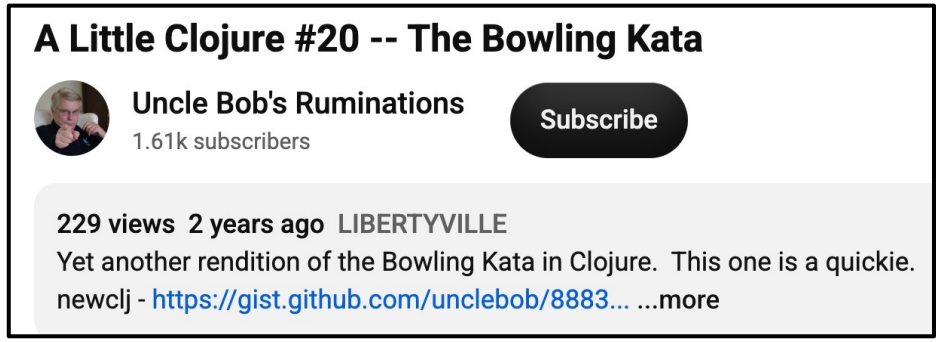
It turns out that, in a way, **Robert Martin** himself **explored** a **regression** from **decoupling** to **coupling** in **Clojure** **bowling programs**.

While in his **2023** **Clojure** **program** in **Functional Design** the mapping of rolls to frames was **decoupled** from the computation of the score, the two were **coupled** in his **2024** **Bowling Kata** video on **YouTube**.



Oct 2023





Mar 2024



Uncle Bob's Ruminations

@ubruminations • 1.71k subscribers • 48 videos

Random videos about software and other topics. ...more

Subscribed

YouTube



A Little Clojure

Playlist



Clojure



A Little Clojure #20 -- The Bowling Kata



Uncle Bob's Ruminations

1.61k subscribers

Subscribe

229 views 2 years ago LIBERTYVILLE

Yet another rendition of the Bowling Kata in Clojure. This one is a quickie. newclj - https://gist.github.com/unclebob/8883... ...more

https://youtu.be/cZ_mPksVAvM



Here is the **Bowling Game program** in **Robert Martin**'s **video**.

On the next slide we **compare** that program with the one in his **Functional Design** book.



IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools VCS Window Help — Robert C Martin — Sun Mar 10 9:37:46

bowling – bowling-spec

Project: bowling ~/junk/alc20/bowling
- .cpcache
- .idea
- spec
  - bowling_spec.clj
- src
  - bowling.clj
- bowling.iml
- deps.edn
- External Libraries
- Scratches and Consoles

bowling.clj

```
(ns bowling)

(defn score [rolls]
  (loop [frames rolls
         score 0
         frame 1]
    (if (> frame 10)
      score
      (let [[a b c] frames]
        (cond
          (= 10 a)
          (recur (drop 1 frames) (+ score a b c) (inc frame))

          (= 10 (+ a b))
          (recur (drop 2 frames) (+ score a b c) (inc frame))

          :else
          (recur (drop 2 frames) (+ score a b) (inc frame)))))))
```

bowling_spec.clj

```
(ns bowling-spec
  (:require [speclj.core :refer :all]
            [bowling :refer :all]))

(describe "bowling"
  (it "scores gutter game"
    (should= 0 (score (repeat 20 0))))

  (it "scores all ones"
    (should= 20 (score (repeat 20 1))))

  (it "scores one spare"
    (should= 16 (score (concat [5 5 3] (repeat 17 0)))))

  (it "scores one strike"
    (should= 24 (score (concat [10 3 4] (repeat 16 0)))))

  (it "scores perfect game"
    (should= 300 (score (repeat 12 10))))
)
```

Version 9 - Clojure – Robert Martin

```clojure
(defn to-frames [rolls]
  (loop [remaining-rolls rolls
         frames []]
    (cond
      (empty? remaining-rolls)
      frames

      (= 10 (first remaining-rolls))
      (recur (rest remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      (= 10 (reduce + (take 2 remaining-rolls)))
      (recur (drop 2 remaining-rolls)
             (conj frames (take 3 remaining-rolls)))

      :else
      (recur (drop 2 remaining-rolls)
             (conj frames (take 2 remaining-rolls))))))

(defn add-frames [score frame]
  (+ score (reduce + frame)))

(defn score [rolls]
  (reduce add-frames 0 (take 10 (to-frames rolls))))
```

Functional Design
Principles, Patterns, Practices

Version 10 - Clojure – Robert Martin

```clojure
(defn score [rolls]
  (loop [frames rolls
         score 0
         frame 1]
    (if (> frame 10)
        score
        (let [[a b c] frames]
          (cond (= 10 a)
                (recur (drop 1 frames)(+ score a b c)(inc frame))

                (= 10 (+ a b))
                (recur (drop 2 frames)(+ score a b c)(inc frame))

                :else
                (recur (drop 2 frames)(+ score a b)(inc frame))))))
)
```

A Little Clojure #20

{{{recursion}}}



The next slide shows the Scala translation of that last version of Robert Martin's Clojure program.

The subsequent slide shows the Haskell translation.



```clojure
(defn score [rolls]
  (loop [frames rolls, score 0, frame 1]
    (if (> frame 10)
      score
      (let [[a b c] frames]
        (cond (= 10 a)       (recur (drop 1 frames)(+ score a b c)(inc frame))
              (= 10 (+ a b)) (recur (drop 2 frames)(+ score a b c)(inc frame))
              :else          (recur (drop 2 frames)(+ score a b)(inc frame))))))
)
```



Version 10 - **Clojure** – **Robert Martin**





```scala
def score(rolls: List[Roll], frame: Int = 0, total: Int = 0): Int =
  if frame == 10 then total else rolls match
    case a::b::c::rs if a == 10   => score(b::c::rs, frame+1, total+a+b+c)
    case a::b::c::rs if a+b == 10 =>    score(c::rs, frame+1, total+a+b+c)
    case a::b::rs                 =>       score(rs, frame+1, total+a+b)
```



Scala translation of 'Version 10 - Clojure – Robert Martin'

```scala
def score(rolls: List[Roll], frame: Int = 0, total: Int = 0): Int =
  if frame == 10 then total else rolls match
    case a::b::c::rs if a == 10   => score(b::c::rs, frame+1, total+a+b+c)
    case a::b::c::rs if a+b == 10 =>    score(c::rs, frame+1, total+a+b+c)
    case a::b::rs                 =>       score(rs, frame+1, total+a+b)
```





```haskell
scoreGame :: [Roll] -> Int
scoreGame rolls = score rolls 0 0
  where score _          frame total | frame == 10 = total
        score (a:b:c:rs) frame total | a == 10     = score (b:c:rs) (frame+1) (total+a+b+c)
        score (a:b:c:rs) frame total | a + b == 10 = score   (c:rs) (frame+1) (total+a+b+c)
        score (a:b:rs)   frame total               = score      rs  (frame+1) (total+a+b)
```

reddit r/Clojure Search in r/Clojure

r/Clojure • 2mo ago
u/philip_schwarz

The Bowling Game - From Imperative to Functional Programming - Part 2

axvr • 2mo ago

I've found this and part 1 to be fascinating read on the history and variety of possible solutions. It was interesting to see the different coding styles generally.

I solved it myself a couple of times and found it was really well suited to recursion and teaching recursion. This was my second solution from early 2023 or late 2022 in 7 lines of Clojure: https://github.com/axvr/archive/blob/master/2023/misc/bowling.cljc

I did not write it using TDD and I've just added some additional test cases from the OP to confirm that it does actually work. My solution was also prior to having seen any of the solutions included by the OP. Mine appears to be quite close to the domain-free solutions, but does have some domain in there.

3 Reply Share ...

philip_schwarz OP • 2mo ago

u/axvr Thank you so much for taking the time to let me know.

Very interesting solution you shared, and it turns out I had already planned to look at other Clojure `loop` examples in part 3 !

1 Reply Share 26 ...

https://github.com/axvr

axvr

On the next slide, we compare **Martin**'s **Clojure** **program** with that of **Alex Vear**, who shared it in his reddit comment on part 2 of this deck series.



Version 10 - Clojure – Robert Martin



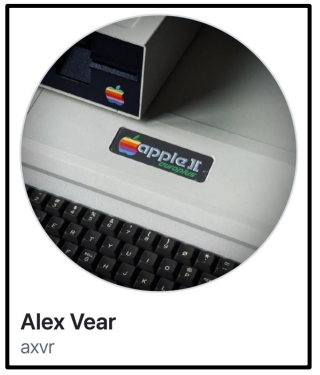
```clojure
(defn score [rolls]
  (loop [frames rolls, score 0, frame 1]
    (if (> frame 10)
        score
        (let [[a b c] frames]
          (cond (= 10 a)       (recur (drop 1 frames)(+ score a b c)(inc frame))
                (= 10 (+ a b)) (recur (drop 2 frames)(+ score a b c)(inc frame))
                :else          (recur (drop 2 frames)(+ score a b)(inc frame))))))
)
```



Version 11 - Clojure – Alex Vear



```clojure
(defn score [rolls]
  (loop [turn 1, total 0, [r1 r2 r3 :as rs] rolls]
    (cond
      (= 10 turn)      (apply + total rs)
      (= 10 r1)        (recur (inc turn) (+ total r1 r2 r3) (next rs))
      (= 10 (+ r1 r2)) (recur (inc turn) (+ total r1 r2 r3) (nnext rs))
      :else            (recur (inc turn) (+ total r1 r2)    (nnext rs)))))
```



{{{recursion}}}





On the next and final slide we compare **Alex Vear**'s very succinct **Clojure** program with the equally succinct **Scala** and **Haskell** translations of **Martin**'s **Clojure program** shown on the previous slide.

{{{recursion}}}



Version 11 - **Clojure** – **Alex Vear**



```clojure
(defn score [rolls]
  (loop [turn 1, total 0, [r1 r2 r3 :as rs] rolls]
    (cond
      (= 10 turn)      (apply + total rs)
      (= 10 r1)        (recur (inc turn) (+ total r1 r2 r3) (next rs))
      (= 10 (+ r1 r2)) (recur (inc turn) (+ total r1 r2 r3) (nnext rs))
      :else            (recur (inc turn) (+ total r1 r2)    (nnext rs)))))
```



```scala
def score(rolls: List[Roll], frame: Int = 0, total: Int = 0): Int =
  if frame == 10 then total else rolls match
    case a::b::c::rs if a == 10   => score(b::c::rs, frame+1, total+a+b+c)
    case a::b::c::rs if a+b == 10 =>    score(c::rs, frame+1, total+a+b+c)
    case a::b::rs                 =>       score(rs, frame+1, total+a+b)
```

Scala translation of ’Version 10 - **Clojure** – **Robert Martin**’



```haskell
scoreGame :: [Roll] -> Int
scoreGame rolls = score rolls 0 0
  where score _          frame total | frame == 10 = total
        score (a:b:c:rs) frame total | a == 10     = score (b:c:rs) (frame+1) (total+a+b+c)
        score (a:b:c:rs) frame total | a + b == 10 = score   (c:rs) (frame+1) (total+a+b+c)
        score (a:b:rs)   frame total               = score      rs  (frame+1) (total+a+b)
```

Haskell translation of ’Version 10 - **Clojure** – **Robert Martin**’



That’s all for **part four**.

I hope you liked it.

See you in **part five**.