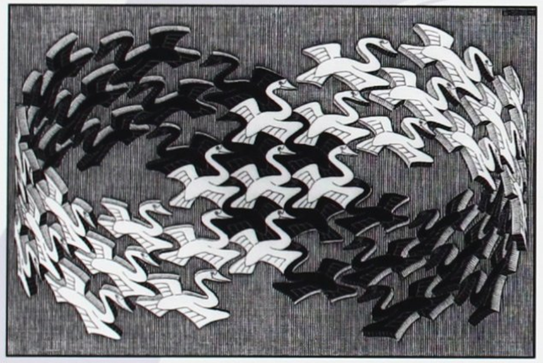

2017
Elixir
conf
usa
Bellevue, WA.
THE NAME Bellevue IS FRENCH FOR Beautiful View
WASHINGTON
SEPT 6 7 8
Micros


2017
Elixir
conf
usa
Bellevue, WA.
THE NAME Bellevue IS FRENCH FOR Beautiful View
WASHINGTON
SEPT 6 7 8
Micros

Design Patterns

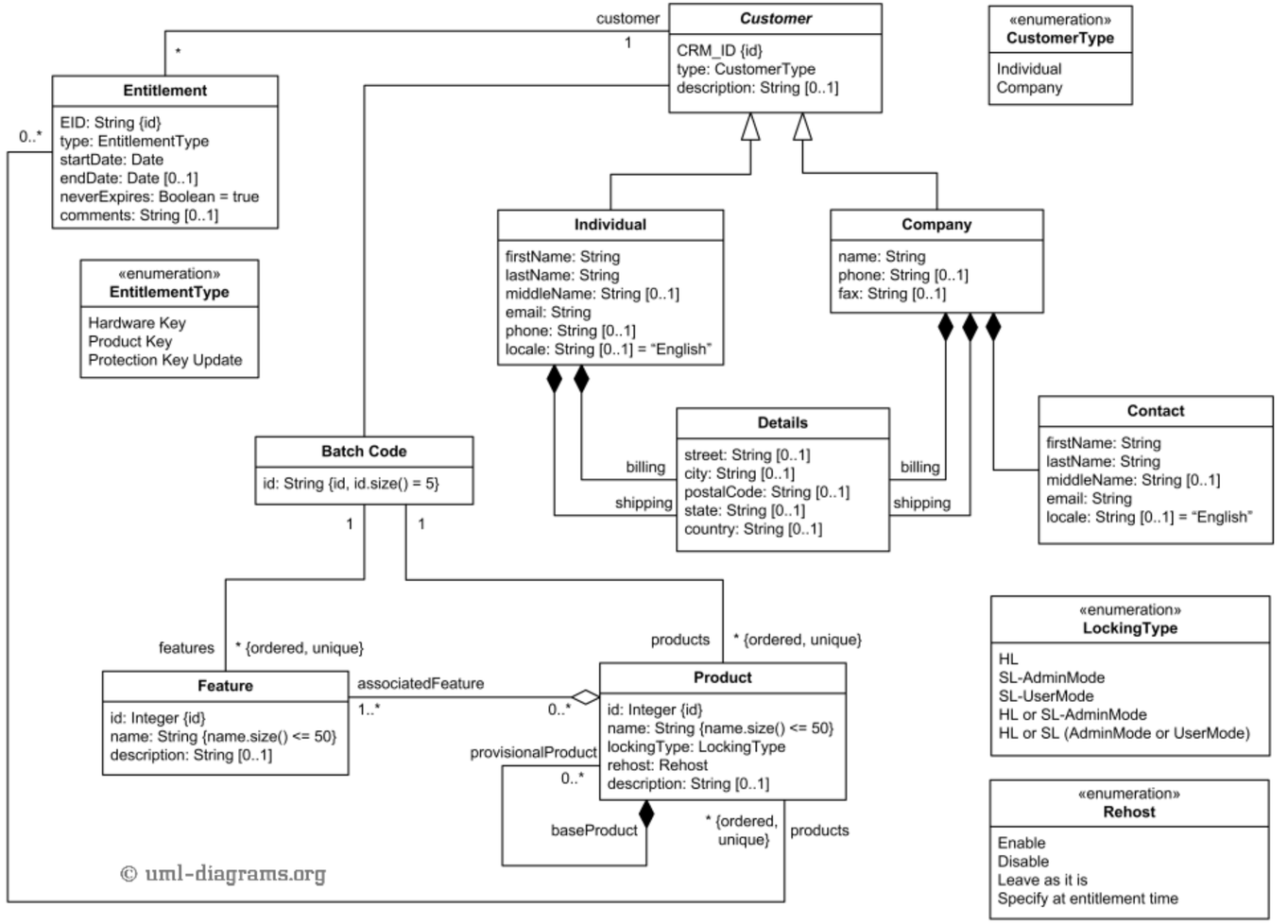
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Every application is a collection of code; the code's arrangement is the design.

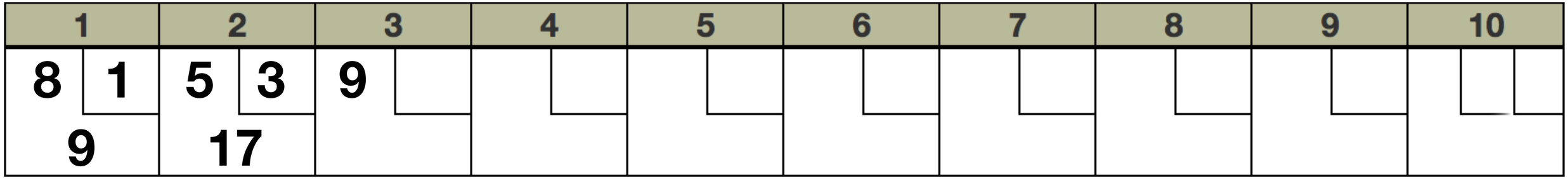
— Sandi Metz

Practical Object-Oriented Design in Ruby

Design == Coding

Katas

Open frames



- In 2 rolls, < 10 pins knocked down
- Score is sum of pinfall

Spares

1	2	3	4	5	6	7	8	9	10
8 /	5								
15									

- In 2 rolls, 10 pins knocked down
- Score is 10 + next roll

Strikes

1	2	3	4	5	6	7	8	9	10
X	5 3								
18	26								

- 10 pins knocked down on first roll
- Score is 10 + next 2 rolls

1	2	3	4	5	6	7	8	9	10
X	5 /	7 1							
20	37	45							

- Last frame"extra" slots for bonus rolls

bowling kata

bowling kata
bowling kata **c#**
bowling kata **java**
bowling kata **ruby**

Google Search | I'm Feeling Lucky

Report inappropriate predictions

Bowling in Ruby Challenges

- Score after every roll
- No conditionals

```ruby
require "forwardable"

module Bowling
  class Game
    attr_reader :frames

    def initialize
      @frames = Array.new(10) { Frame.new }
    end

    def roll(pinfall)
      frames_handling_roll.each { |f| f.roll(pinfall) }
    end

    def score
      frames.sum(&:score)
    end

    private

    def frames_handling_roll
      frame_handling_normal_roll + frames_handling_bonus_roll
    end

    def frame_handling_normal_roll
      Array(frames.detect(&:handles_normal_roll?)).compact
    end

    def frames_handling_bonus_roll
      frames.select(&:handles_bonus_roll?)
    end
  end

  class Frame
    extend Forwardable
    def_delegators :@state,
      :score,
      :handles_normal_roll?,
      :handles_bonus_roll?

    attr_reader :rolls
    attr_accessor :state

    def initialize
      @rolls = []
      @state = FrameState.initial_state(self)
    end

    def roll(pinfall)
      rolls << pinfall
      state.transition!
    end

    def strike?
      rolls.first == 10
    end

    def spare?
      !strike? && rolls.take(2).sum == 10
    end

    def bonus?
      strike? || spare?
    end

    def open?
      !strike? && !spare? && rolls_count == 2
    end

    def rolls_count
      rolls.length
    end
  end

  class FrameState
    def self.initial_state(frame)
      PendingState.new(frame)
    end

    attr_reader :frame

    def initialize(frame)
      @frame = frame
    end

    def transition!
      frame.state = [BonusState, CompleteState, PendingState].detect do |klass|
        klass.state_for?(frame)
      end.new(frame)
    end
  end

  class PendingState < FrameState
    def self.state_for?(frame)
      !frame.bonus? && !frame.open?
    end

    def score
      0
    end

    def handles_normal_roll?
      true
    end

    def handles_bonus_roll?
      false
    end
  end

  class BonusState < FrameState
    def self.state_for?(frame)
      frame.bonus? && frame.rolls_count < 3
    end

    def score
      0
    end

    def handles_normal_roll?
      false
    end

    def handles_bonus_roll?
      true
    end
  end

  class CompleteState < FrameState
    def self.state_for?(frame)
      (frame.bonus? && frame.rolls_count == 3) ||
        (frame.open? && frame.rolls_count == 2)
    end

    def score
      frame.rolls.sum
    end

    def handles_normal_roll?
      false
    end

    def handles_bonus_roll?
      false
    end
  end
end
```

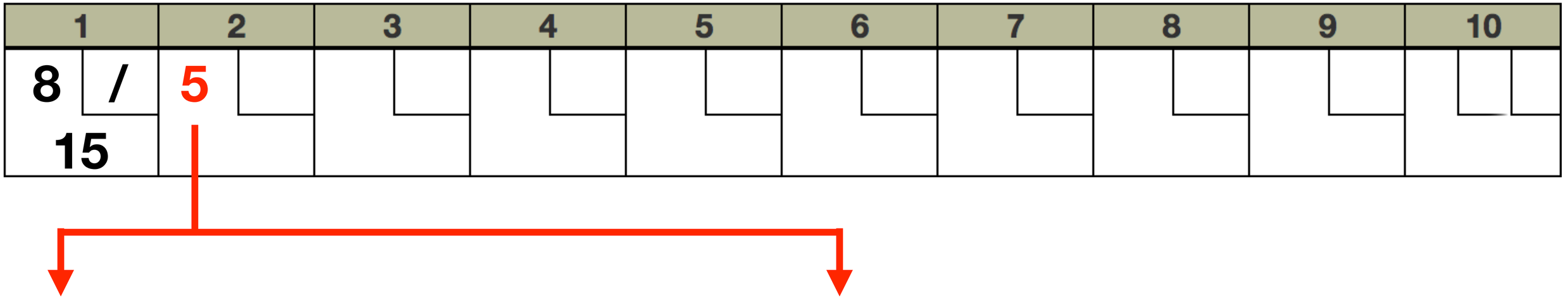
Ruby: Summary

- Public API: Game#roll(pinfall), Game#score
- Frame
- FrameState
  - PendingState
  - BonusState
  - CompleteState

```ruby
class Game
  attr_reader :frames

  def initialize
    @frames = Array.new(10) { Frame.new }
  end

  def roll(pinfall)
    frames_handling_roll.each { |f| f.roll(pinfall) }
  end

  def score
    frames.sum(&:score)
  end

  private

  def frames_handling_roll
    frame_handling_normal_roll + frames_handling_bonus_roll
  end

  def frame_handling_normal_roll
    Array(frames.detect(&:handles_normal_roll?)).compact
  end

  def frames_handling_bonus_roll
    frames.select(&:handles_bonus_roll?)
  end
end
```

Ruby: Game

Ruby: roll distribution



Frame 1

handles_bonus_roll? # => true

rolls: [8,2,5]

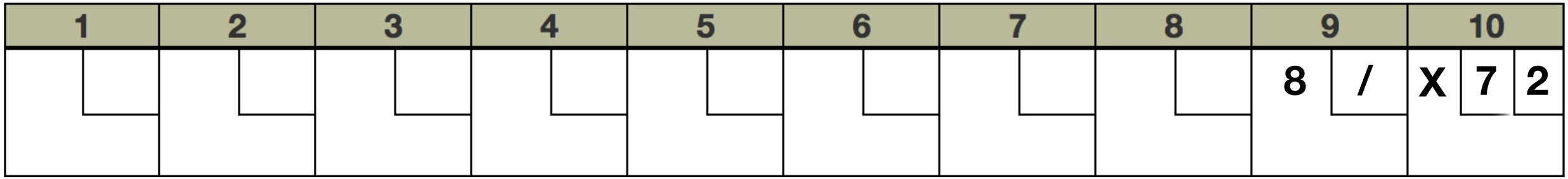
Frame 2

handles_normal_roll? # => true

rolls: [5]

Ruby: Design

- Frames are completely decoupled
- Scoring is simple



Frame 9: [8,2,10].sum # => 20

Frame 10: [10,7,2].sum # => 19

Tyranny of DRY

1	2	3	4	5	6	7	8	9	10
X	5 /	7 1							
20	37	45							

```ruby
class Frame
  extend Forwardable
  def_delegators :@state, :score, :handles_normal_roll?, :handles_bonus_roll?

  attr_reader :rolls
  attr_accessor :state

  def initialize
    @rolls = []
    @state = FrameState.initial_state(self)
  end

  def roll(pinfall)
    rolls << pinfall
    state.transition!
  end

  def strike?
    rolls.first == 10
  end

  def spare?
    !strike? && rolls.first(2).sum == 10
  end

  def bonus?
    strike? || spare?
  end

  def open?
    !strike? && !spare? && rolls_count == 2
  end

  def rolls_count
    rolls.length
  end
end
```

Ruby: Frame

```ruby
class FrameState
  def self.initial_state(frame)
    PendingState.new(frame)
  end

  attr_reader :frame

  def initialize(frame)
    @frame = frame
  end

  def transition!
    frame.state =
      [BonusState, CompleteState, PendingState].detect do |klass|
        klass.state_for?(frame)
      end.new(frame)
  end
end
```

Ruby: FrameState

```
class PendingState < FrameState
  def self.state_for?(frame)
    !frame.bonus? && !frame.open?
  end

  def score
    0
  end

  def handles_normal_roll?
    true
  end

  def handles_bonus_roll?
    false
  end
end
```

Ruby: PendingState

```ruby
class BonusState < FrameState
  def self.state_for?(frame)
    frame.bonus? && frame.rolls_count < 3
  end

  def score
    0
  end

  def handles_normal_roll?
    false
  end

  def handles_bonus_roll?
    true
  end
end
```

Ruby: BonusState

```ruby
class CompleteState < FrameState
  def self.state_for?(frame)
    (frame.bonus? && frame.rolls_count == 3) ||
      (frame.open? && frame.rolls_count == 2)
  end

  def score
    frame.rolls.sum
  end

  def handles_normal_roll?
    false
  end

  def handles_bonus_roll?
    false
  end
end
```

Ruby: CompleteState

Ruby: Qualities

- Long-ish
- Small methods, thoroughly decomposed
- No conditionals!
- State pattern (emerged)
- Names tell a story about the domain

Ruby: Lessons

- State pattern
- Reinforced: UI is not your software design
- Ask better questions

Elixir: Challenges

- Language learning, idioms
- Embrace functional programming
- Lists and recursion
- Stateless game
- Score any number of rolls
- No conditionals!

```elixir
defmodule Bowling do
  # strike in last frame
  def score([10, bonus1, bonus2 | _tail = []]) do
    10 + bonus1 + bonus2
  end

  # strike
  def score([10, bonus1, bonus2 | tail]) do
    10 + bonus1 + bonus2 + score([bonus1, bonus2 | tail])
  end

  # spare
  def score([roll1, roll2, bonus | tail]) when roll1 + roll2 == 10 do
    10 + bonus + score([bonus | tail])
  end

  # open frame
  def score([roll1, roll2 | tail]) when roll1 + roll2 < 10 do
    roll1 + roll2 + score(tail)
  end

  def score([_|_]), do: 0 # incomplete frame
  def score([]), do: 0 # no rolls
end
```

Elixir

```elixir
defmodule Bowling do
  defstruct rolls: [], score: 0

  def new_game do
    %Bowling{}
  end

  def roll(game = %Bowling{}, pinfall) do
    rolls = append_pinfall(game.rolls, pinfall)
    %{ game | rolls: rolls, score: score(rolls) }
  end

  ### scoring code elided...

  defp append_pinfall(rolls, pinfall) do
    [ pinfall | Enum.reverse(rolls) ]
    |> Enum.reverse
  end
end
```

Elixir similar API

```elixir
defmodule Bowling do
  # strike in last frame
  def score([10, bonus1, bonus2 | _tail = []]) do
    10 + bonus1 + bonus2
  end

  # strike
  def score([10, bonus1, bonus2 | tail]) do
    10 + bonus1 + bonus2 + score([bonus1, bonus2 | tail])
  end

  # spare
  def score([roll1, roll2, bonus | tail]) when roll1 + roll2 == 10 do
    10 + bonus + score([bonus | tail])
  end

  # open frame
  def score([roll1, roll2 | tail]) when roll1 + roll2 < 10 do
    roll1 + roll2 + score(tail)
  end

  def score([_|_]), do: 0 # incomplete frame
  def score([]), do: 0 # no rolls
end
```

Elixir

Elixir: Qualities

- Brevity
- Higher per-line cognitive load to read
- Doesn't name domain concepts
- No conditionals!
- Functional features
  - Lists
  - Pattern matching (order-dependent)
  - Guard clauses
  - Recursion

Functional Ruby?

```elixir
defmodule Bowling do
  # strike in last frame
  def score([10, bonus1, bonus2 | _tail = []]) do
    10 + bonus1 + bonus2
  end

  # strike
  def score([10, bonus1, bonus2 | tail]) do
    10 + bonus1 + bonus2 + score([bonus1, bonus2 | tail])
  end

  # spare
  def score([roll1, roll2, bonus | tail]) when roll1 + roll2 == 10 do
    10 + bonus + score([bonus | tail])
  end

  # open frame
  def score([roll1, roll2 | tail]) when roll1 + roll2 < 10 do
    roll1 + roll2 + score(tail)
  end

  def score([_|_]), do: 0 # incomplete frame
  def score([]), do: 0 # no rolls
end
```

Lists, pattern matching

```elixir
defmodule Bowling do
  # strike in last frame
  def score([10, bonus1, bonus2 | _tail = []]) do
    10 + bonus1 + bonus2
  end

  # strike
  def score([10, bonus1, bonus2 | tail]) do
    10 + bonus1 + bonus2 + score([bonus1, bonus2 | tail])
  end

  # spare
  def score([roll1, roll2, bonus | tail]) when roll1 + roll2 == 10 do
    10 + bonus + score([bonus | tail])
  end

  # open frame
  def score([roll1, roll2 | tail]) when roll1 + roll2 < 10 do
    roll1 + roll2 + score(tail)
  end

  def score([_|_]), do: 0 # incomplete frame
  def score([]), do: 0 # no rolls
end
```

Guard clauses

```elixir
defmodule Bowling do
  # strike in last frame
  def score([10, bonus1, bonus2 | _tail = []]) do
    10 + bonus1 + bonus2
  end

  # strike
  def score([10, bonus1, bonus2 | tail]) do
    10 + bonus1 + bonus2 + score([bonus1, bonus2 | tail])
  end

  # spare
  def score([roll1, roll2, bonus | tail]) when roll1 + roll2 == 10 do
    10 + bonus + score([bonus | tail])
  end

  # open frame
  def score([roll1, roll2 | tail]) when roll1 + roll2 < 10 do
    roll1 + roll2 + score(tail)
  end

  def score([_|_]), do: 0 # incomplete frame
  def score([]), do: 0 # no rolls
end
```

Recursion

Affordances

Open frames

```
def score(rolls)
  if rolls.length.even?
    rolls.sum
  else
    rolls.pop
    rolls.sum
  end
end
```

```
def score([roll1, roll2 | tail]) do
  roll1 + roll2 + score(tail)
end

def score([_|_]), do: 0
def score([]), do: 0
```

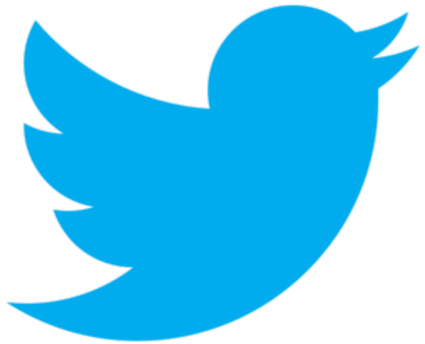
Affordances in Elixir for avoiding conditionals

- Lists
- Pattern matching
- Guard clauses
- Recursion

The End


User
Testing

Questions?



@tjstankus