Fluent BDD

a composed approach to behaviours

From the outside in

I make my customer happy by providing features

This keeps me in a job and pays for the stuff I want to have/do

So let's make a feature:

Imagine we are providing a system for a bowling alley…

From the outside in

Let’s look at scoring

Feature: Score Calculation

From the outside in

Let’s look at scoring

Feature: Score Calculation

Why do I want this feature?

So players get competitive and keep buying games,
but don’t fight about the scoring rules – so I don’t have to call the cops!

From the outside in

So,

Feature: Score Calculation

For the bowling alley owner

Who is the stakeholder for ***this feature***, and what do they want from this feature?

From the outside in

So,

Feature: Score Calculation

For the bowling alley owner

To show players their scores

I want the system to calculate player’s total score

Great! A feature. So now I need to implement it.

First question: how do I know when I’m done?

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

In BDD, we start with a **Behaviour**:

When I play a game, **then** I should get the score

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

Next, as developers, we need something that will do the work for us:

With a **Game Scorer**

When I play a game, **then** I should get the score

This is our **Subject**

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

Let's now look at two assumptions we've made:

With a **Game Scorer**

When I play a **game**, **then** I should get the **score**

"Game" and "score" are things we expect to be related by the behaviour.

They could have any range of values and still fit the behaviour.

We will call this pair our **Expectations**

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

So we have a Behaviour, a Subject and some Expectations.

With a **Game Scorer**

When I play a **game**, **then** I should get the **score**

At this point we reach a problem: our subject isn't referenced in the behaviour.

Time to refactor!

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

Let's ask the game scorer for the score:

With a **Game Scorer**

When I ask for the score,

then I should get the **score**

Ok, now our behaviour references the subject, but we've lost one of our expectations!

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

Put it back, and

With a **Game Scorer** that has had a **game** played

When I ask for the score,

then I should get the **score**

Now we’ve got a Subject that has had something done to it,

But we haven’t said *what* or *how*…

Let’s do that now

From the outside in

A game scorer that records pins knocked down:

Given a new **Game Scorer**

For every ball thrown in the **game**

Tell the **Game Scorer** how many pins were knocked down

This is everything that it takes to get from nothing up until we ask for the score

We call this the **Context**

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

Let’s put our Context back in:

Given *A game scorer that records pins knocked down*

When I ask for the score,

then I should get the **score**

Great, but we’ve lost that Expectation again.

It needs to be given to the Context, but it belongs with it’s other half (“score”)

Let’s put it back

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

The Context and Action are **using** our Expectation, so let's call it that

Given *A game scorer that records pins knocked down*

When I ask for the score,

Using a **game** played,

Then I should get the **score**

That's looking pretty complete, and it appears to have 4 separate parts:

With **Context**, when **Action**, using **Expectation,** then **Assert**

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

We need a name for this thing we have composed.

Given *A game scorer that records pins knocked down*

When I ask for the score,

Using a **game** played,

Then I should get the **score**

We call this the **Scenario**

From the outside in

To recap:

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

Scenario:

Given *A game scorer that records pins knocked down*

When I ask for the score,

Using a game played,

Then I should get the score

We'll need a few more scenarios before we have an entire feature

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

Scenario: a complete game

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played

Then I should get the score

Scenario: too many throws

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played with too many throws

I should get an exception telling me I’ve thrown too many balls

And so on.

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players’ scores

I want the system to calculate players’ total score

Scenario: a complete game

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played

Then I should get the score

Scenario: too many throws

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played with too many throws

I should get an exception telling me I’ve thrown too many balls

Notice that some parts of the scenarios are entirely common…

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

Scenario: a complete game

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played

Then I should get the score

Scenario: too many throws

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played with too many throws

I should get an exception telling me I've thrown too many balls

…some are *nearly* common…

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores

I want the system to calculate players' total score

Scenario: a complete game

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played

Then I should get the score

Scenario: too many throws

Given a game scorer that records pins knocked down

When I ask for the score

Using a game played with too many throws

I should get an exception telling me I've thrown too many balls

…and others are totally different.

From the outside in

Feature: Score Calculation

For the bowling alley owner; To show players' scores
I want the system to calculate players' total score

Scenario: a complete game

Given a game scorer that records pins knocked down
When I ask for the score
Using a game played
Then I should get the score

Scenario: too many throws

Given a game scorer that records pins knocked down
When I ask for the score
Using a game played with too many throws
I should get an exception telling me I've thrown too many balls

Scenario: a half finished game

Given a game scorer that records pins knocked down
When I ask for the score
Using a game played that stops half way through
Then I should get the score so far

Scenario: a correction

Given a game scorer that records pins knocked down
When I correct the last throw
Using a game played with too many throws
I should get a corrected score

If we have to write out each scenario in full every time, things quickly get out of hand…

From the outside in

```
Feature: Score Calculation
        For the bowling alley owner; To show players' scores
        I want the system to calculate players' total score
Scenario: a complete game
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played
        Then I should get the score
Scenario: too many throws
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played with too many throws
        I should get an exception telling me I've thrown too many balls
Scenario: a half finished game
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played that stops half way through
        Then I should get the score so far
Scenario: a correction
        Given a game scorer that records pins knocked down
        When I correct the last throw
        Using a game played with too many throws
        I should get a corrected score
Scenario: manager's override
        Given a game scorer that records pins knocked down
        When manager changes score
        Using a game played
        Then I should get the new score
Scenario: a player quits
        Given a game scorer that records pins knocked down
        When player leaves the game
        Using a game played
        I should get a short final score
        Other players can keep playing
```

…and each of these would need some matching code…

From the outside in

```
Feature: Score Calculation
        For the bowling alley owner; To show players' scores
        I want the system to calculate players' total score
Scenario: a complete game
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played
        Then I should get the score
Scenario: too many throws
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played with too many throws
        I should get an exception telling me I've thrown too many balls
Scenario: a half finished game
        Given a game scorer that records pins knocked down
        When I ask for the score
        Using a game played that stops half way through
        Then I should get the score so far
Scenario: a correction
        Given a game scorer that records pins knocked down
        When I correct the last throw
        Using a game played with too many throws
        I should get a corrected score
Scenario: manager's override
        Given a game scorer that records pins knocked down
        When manager changes score
        Using a game played
        Then I should get the new score
Scenario: a player quits
        Given a game scorer that records pins knocked down
        When player leaves the game
        Using a game played
        I should get a short final score
        Other players can keep playing
Scenario: a correction
        Given a game scorer that records pins knocked down
        When I correct the last throw
        Using a game played with too many throws
        I should get a corrected score
Scenario: yes, some of these are duplicates
        Given a game scorer that records pins knocked down
        How good is your eyesight?
        When manager changes score
        Then I should get the new score
Scenario: a player quits
        Given a game scorer that records pins knocked down
        When player leaves the game
        Using a game played
        I should get a short final score
        Other players can keep playing
```

…which is the shortcoming of most fluent / clear text BDD frameworks.

Summary

- Developing software outside-in: GOOD
- Writing readable, behavioural tests: GOOD

- Copy-and-Pasting specs: BAD
- A sea of specs so large you can't read it: BAD
- Repetition: BAD BAD BAD BAD BAD!

Fluent BDD

Getting all the good with the least bad

What we want to achieve

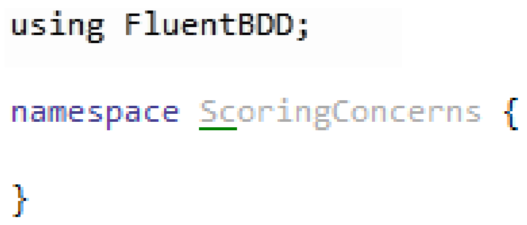
1. Write code that is close to plain English
2. Type as little as possible

What we want to achieve

1. Write code that is close to plain English
   - As little code over-head as possible
   - Fluent interface
   - Good conventions for common cases
2. Type as little as possible
   - Reusable contexts
   - Reusable expectations
   - More than one test per scenario
   - Good interaction with ReSharper

Using FluentBDD

Let’s go back to the bowling alley, this time in code

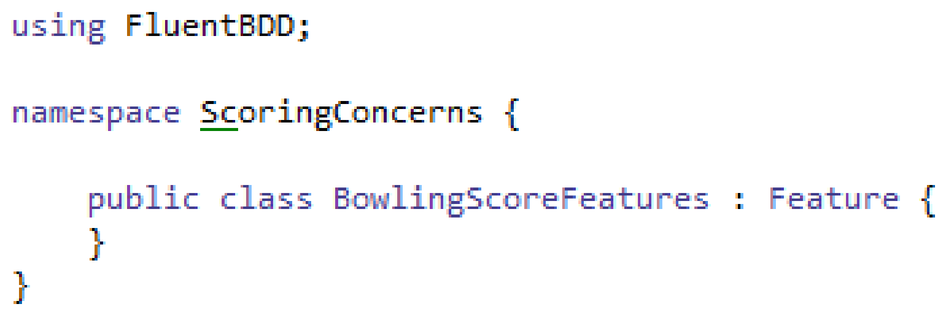


```
using FluentBDD;

namespace ScoringConcerns {

}
```

First, we start with a nice empty code file.

Note “using FluentBDD”

Using FluentBDD



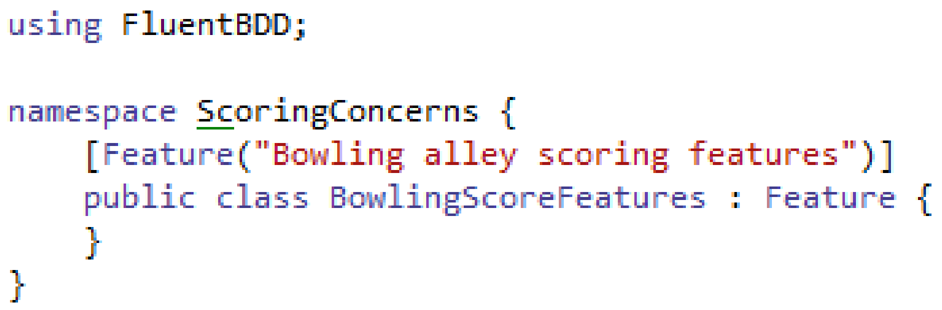
```
using FluentBDD;

namespace ScoringConcerns {

    public class BowlingScoreFeatures : Feature {
    }
}
```

Then we create a class for our feature.

We inherit from "Feature" as that gives us our fluent interface.

The class doesn't need to be public, but it doesn't hurt.

Using FluentBDD

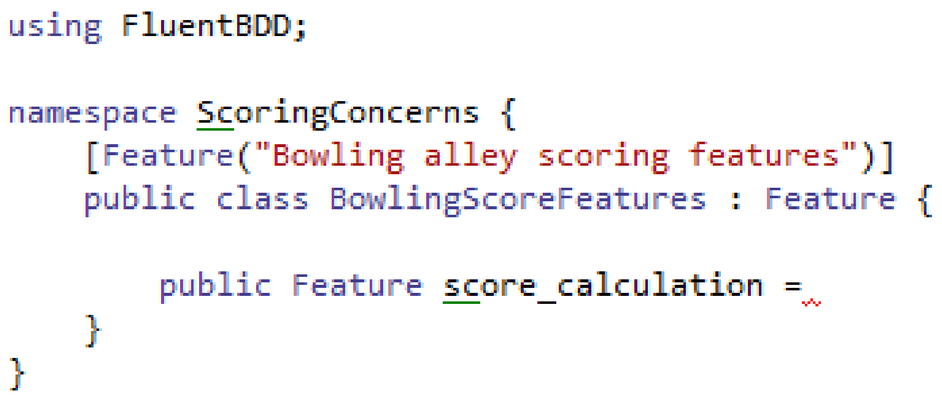


```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {
    }
}
```

Next we mark the class with the feature attribute. This means the features we create will be appear in NUnit.

We give the feature a name, which will be part of the unit test tree.

Using FluentBDD

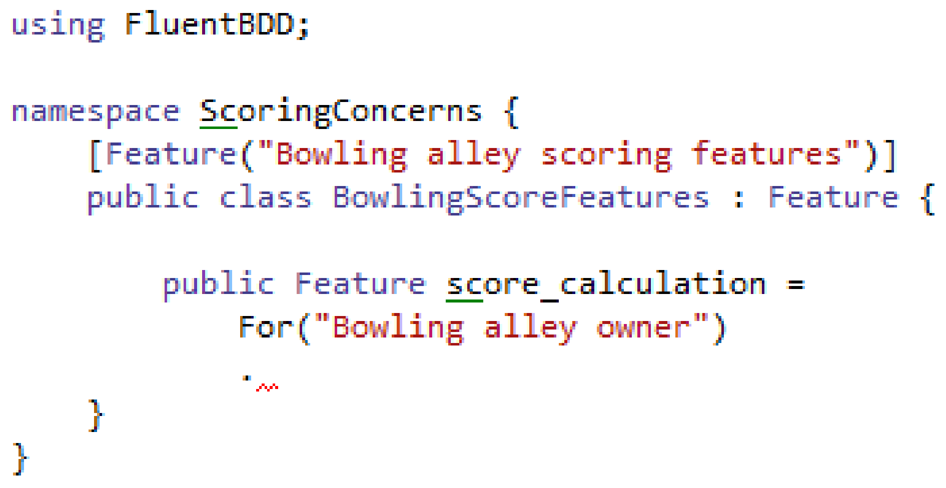


```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
    }
}
```

Next we add a feature giving it a name.

The layout of a feature follows the pattern–

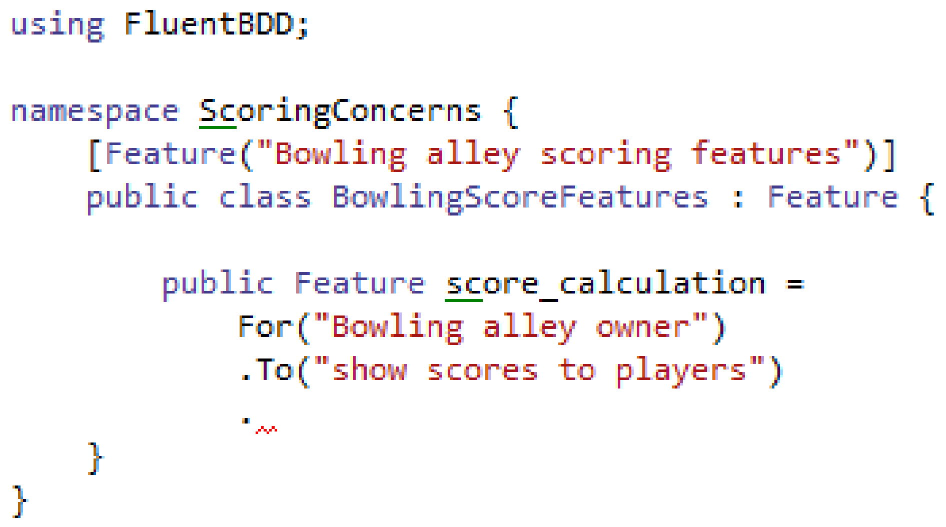
Using FluentBDD



```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
            For("Bowling alley owner")
            .
    }
}
```

For the stakeholder(s),

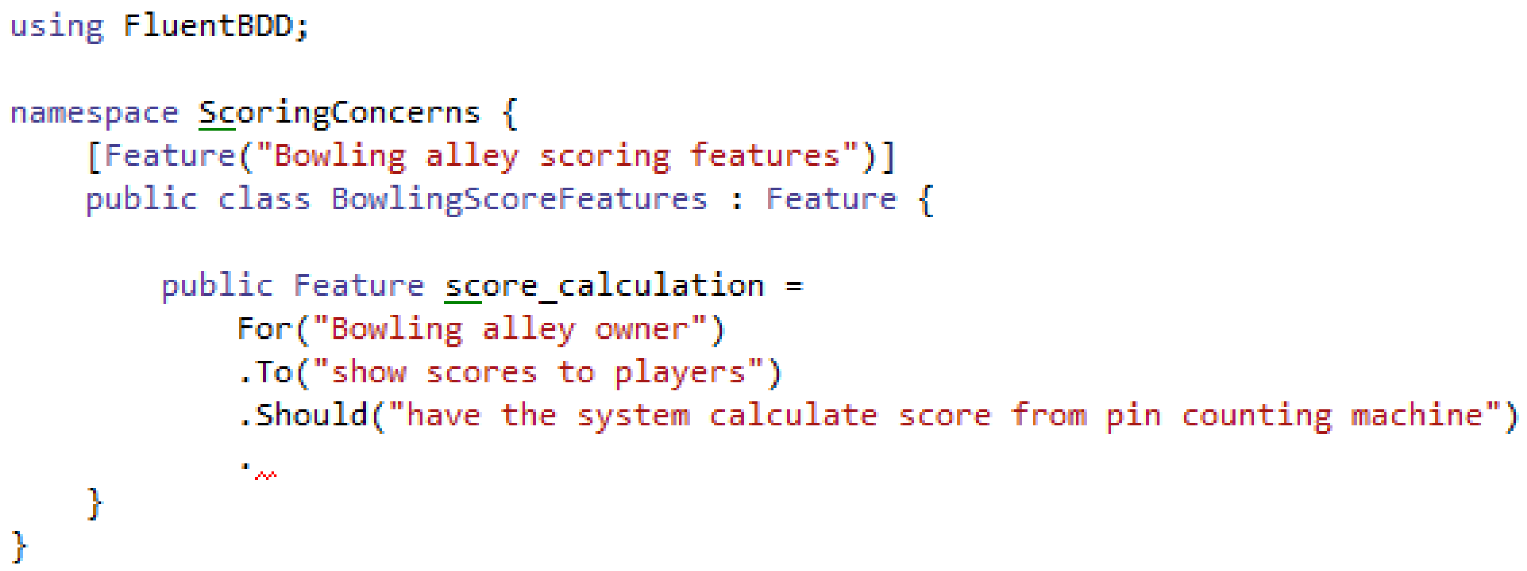
Using FluentBDD



```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
            For("Bowling alley owner")
            .To("show scores to players")
            .
    }
}
```

To achieve a goal,

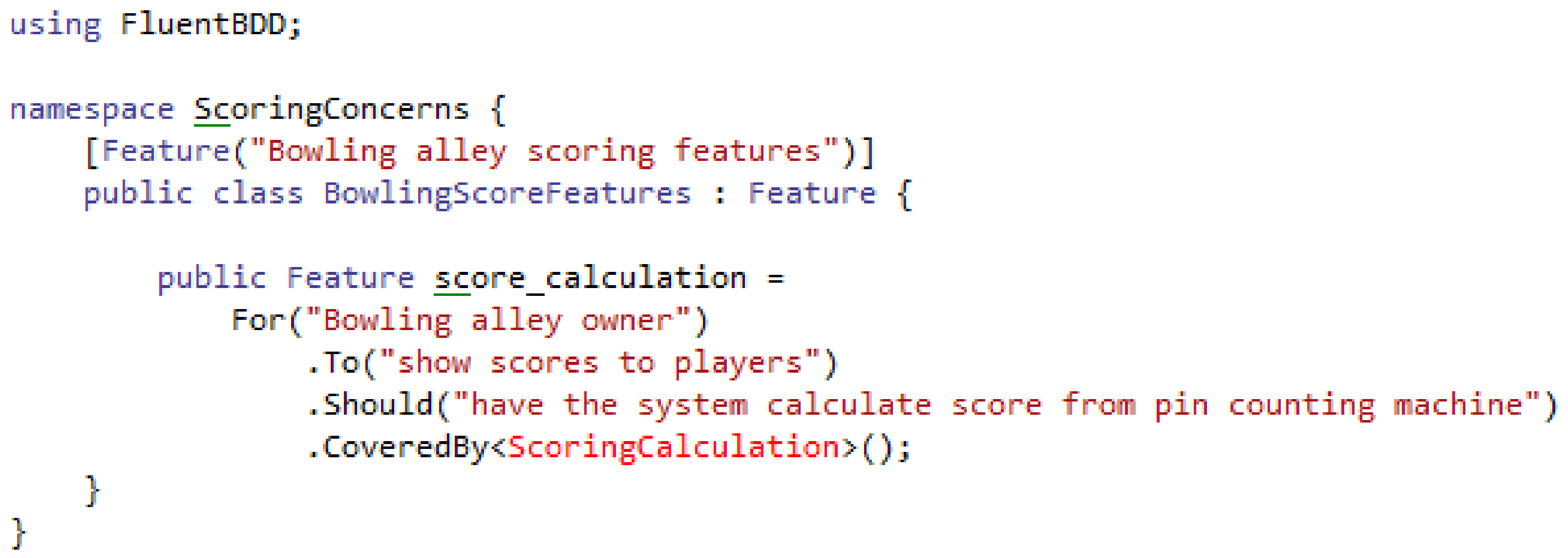
Using FluentBDD



```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
            For("Bowling alley owner")
            .To("show scores to players")
            .Should("have the system calculate score from pin counting machine")
            .
    }
}
```

Should introduce a solution.

Using FluentBDD

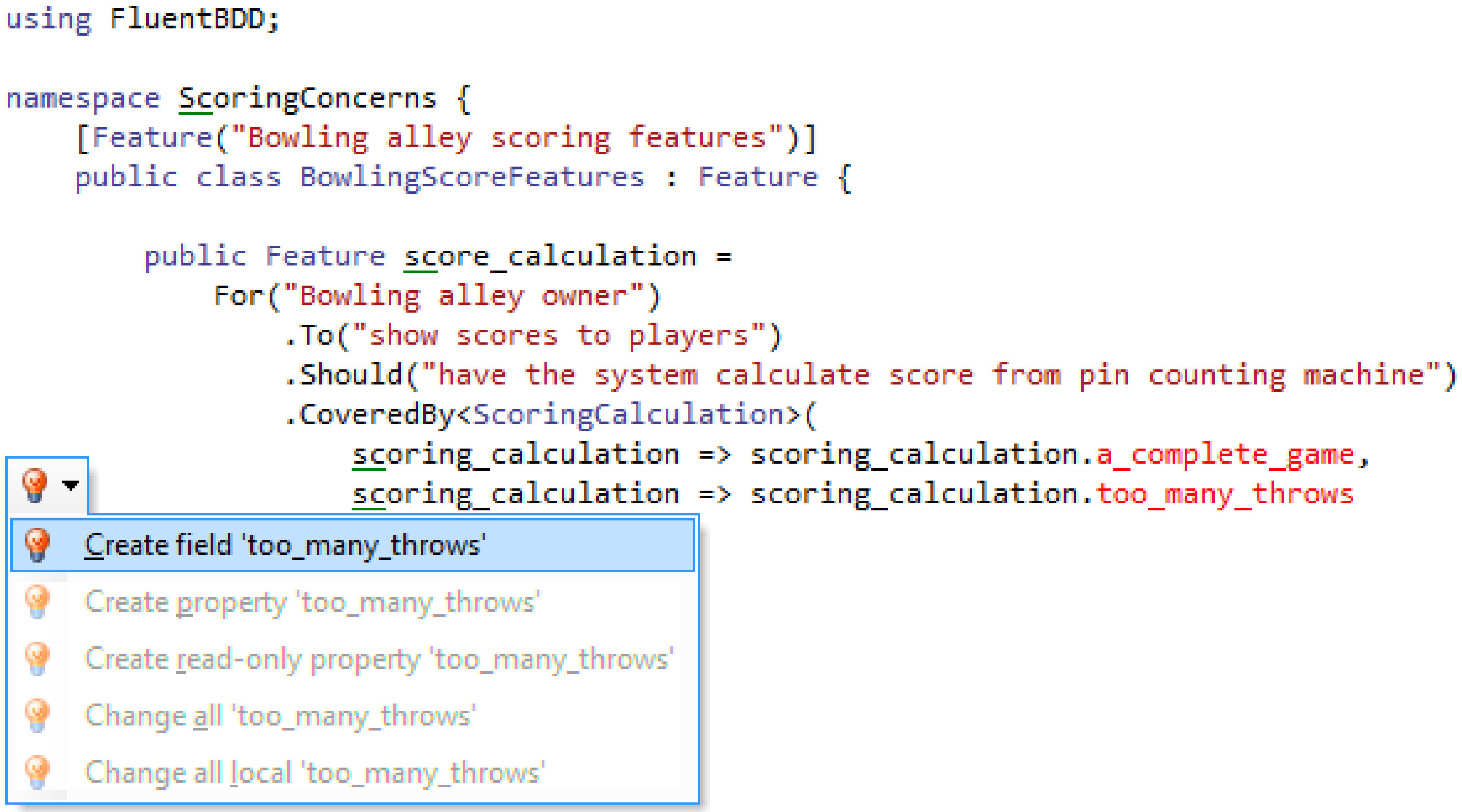


```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
            For("Bowling alley owner")
                .To("show scores to players")
                .Should("have the system calculate score from pin counting machine")
                .CoveredBy<ScoringCalculation>();
    }
}
```

Now we must somehow test this feature has the correct **Behaviour**.

To do this, we include a coverage test.

Using FluentBDD



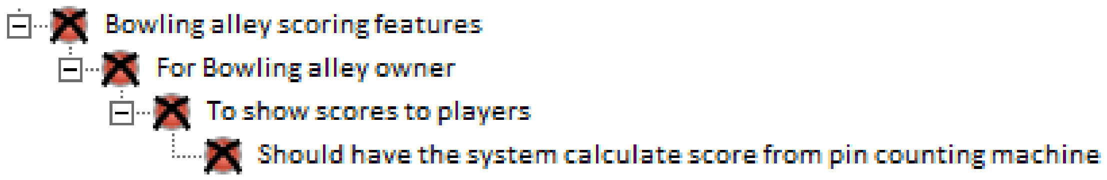
```
using FluentBDD;

namespace ScoringConcerns {
    [Feature("Bowling alley scoring features")]
    public class BowlingScoreFeatures : Feature {

        public Feature score_calculation =
            For("Bowling alley owner")
                .To("show scores to players")
                .Should("have the system calculate score from pin counting machine")
                .CoveredBy<ScoringCalculation>(
                    scoring_calculation => scoring_calculation.a_complete_game,
                    scoring_calculation => scoring_calculation.too_many_throws
```

Create field 'too_many_throws'
Create property 'too_many_throws'
Create read-only property 'too_many_throws'
Change all 'too_many_throws'
Change all local 'too_many_throws'

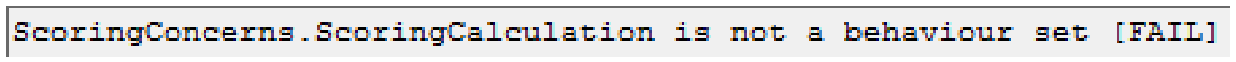
We let ReSharper generate the class – we'll get back to it later.

Now let's rough out some scenarios, and let ReSharper generate those too.

Using FluentBDD



```
Bowling alley scoring features
  For Bowling alley owner
    To show scores to players
      Should have the system calculate score from pin counting machine
```

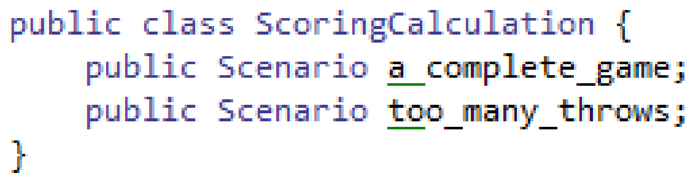


```
ScoringConcerns.ScoringCalculation is not a behaviour set [FAIL]
```

Testing in NUnit, we have our first red test.

Let’s go take a look at our auto-generated behaviours

Using FluentBDD

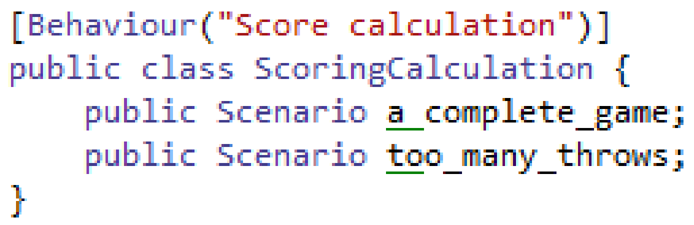


```
public class ScoringCalculation {
    public Scenario a_complete_game;
    public Scenario too_many_throws;
}
```

The basic generated class is quite bare.

We need to do some decoration before we start filling out the scenarios.

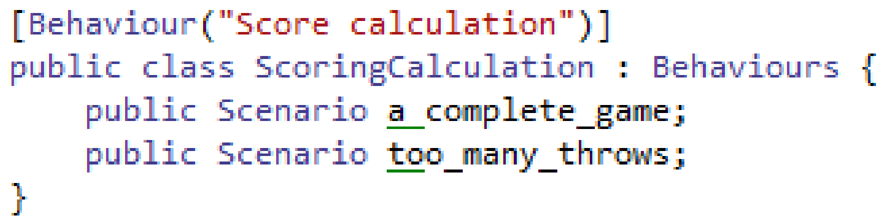
Using FluentBDD



```
[Behaviour("Score calculation")]
public class ScoringCalculation {
    public Scenario a_complete_game;
    public Scenario too_many_throws;
}
```

The **Behaviour** attribute makes scenarios available to NUnit.

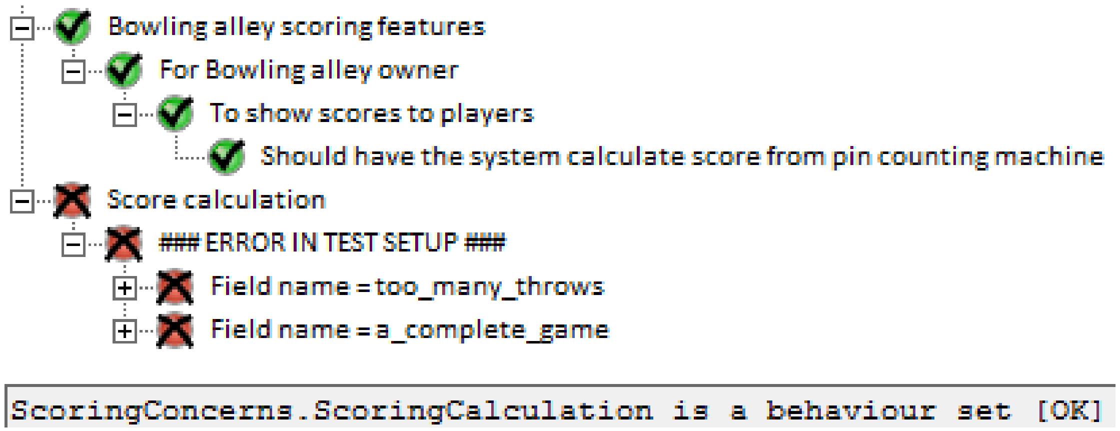
Using FluentBDD



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game;
    public Scenario too_many_throws;
}
```

The **Behaviours** parent class provides our fluent interface.

Using FluentBDD



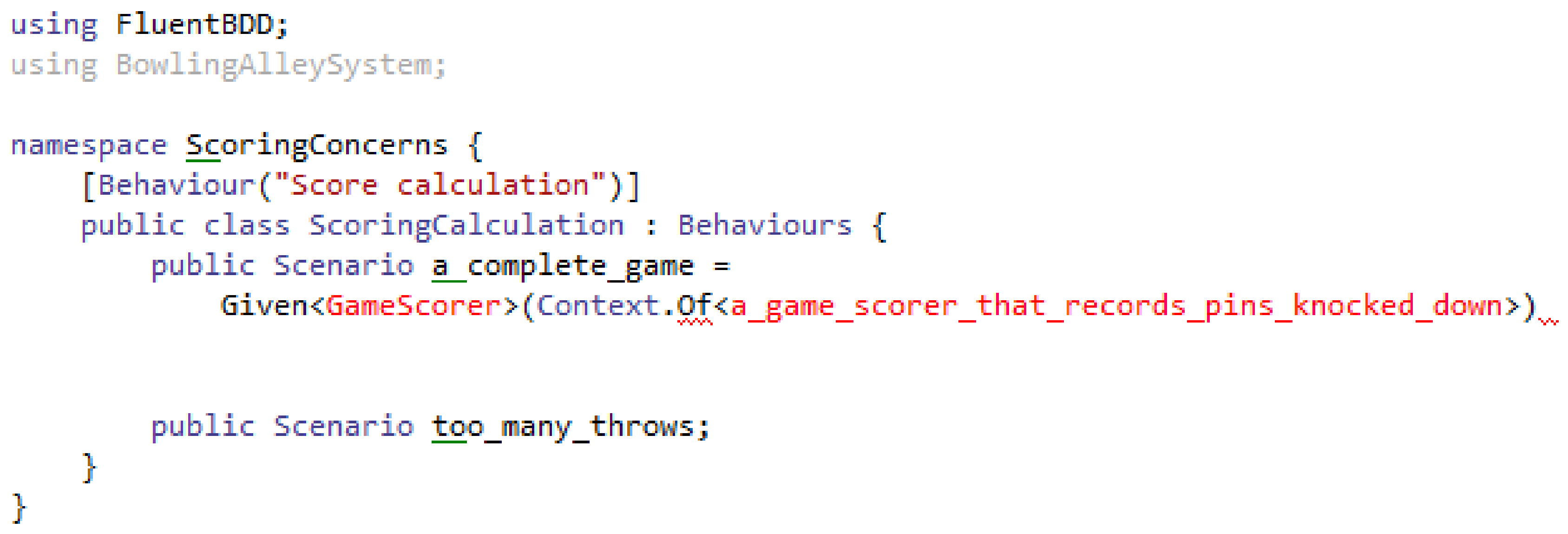
```
⊟ ✓ Bowling alley scoring features
    ⊟ ✓ For Bowling alley owner
        ⊟ ✓ To show scores to players
            ✓ Should have the system calculate score from pin counting machine
⊟ ✗ Score calculation
    ⊟ ✗ ### ERROR IN TEST SETUP ###
        ⊞ ✗ Field name = too_many_throws
        ⊞ ✗ Field name = a_complete_game
```

```
ScoringConcerns.ScoringCalculation is a behaviour set [OK]
```

Now our feature description is green, and we can see the score calculation behaviour tests are red.

Time to move in a level!

Using FluentBDD



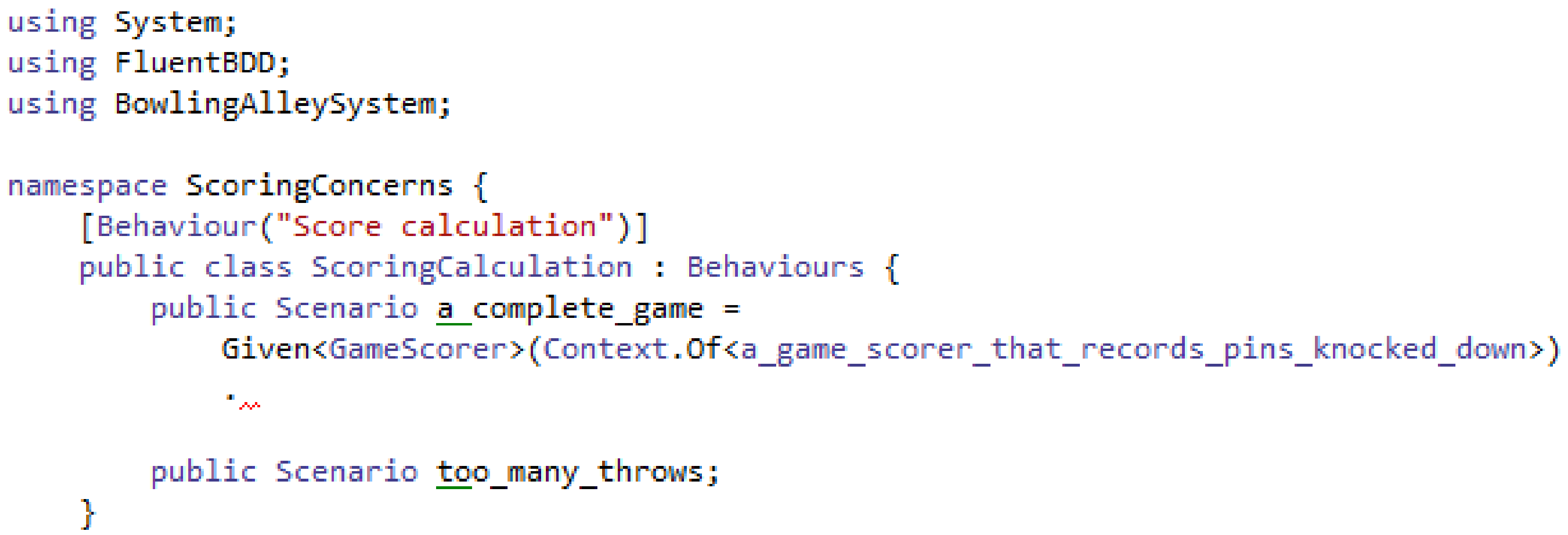
```
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)


        public Scenario too_many_throws;
    }
}
```

With our behaviour, we take the first scenario, and start defining it:

Given < *subject type* **> (Context.Of <** *context type* **>)**

The subject type will go in our production code.

Using FluentBDD

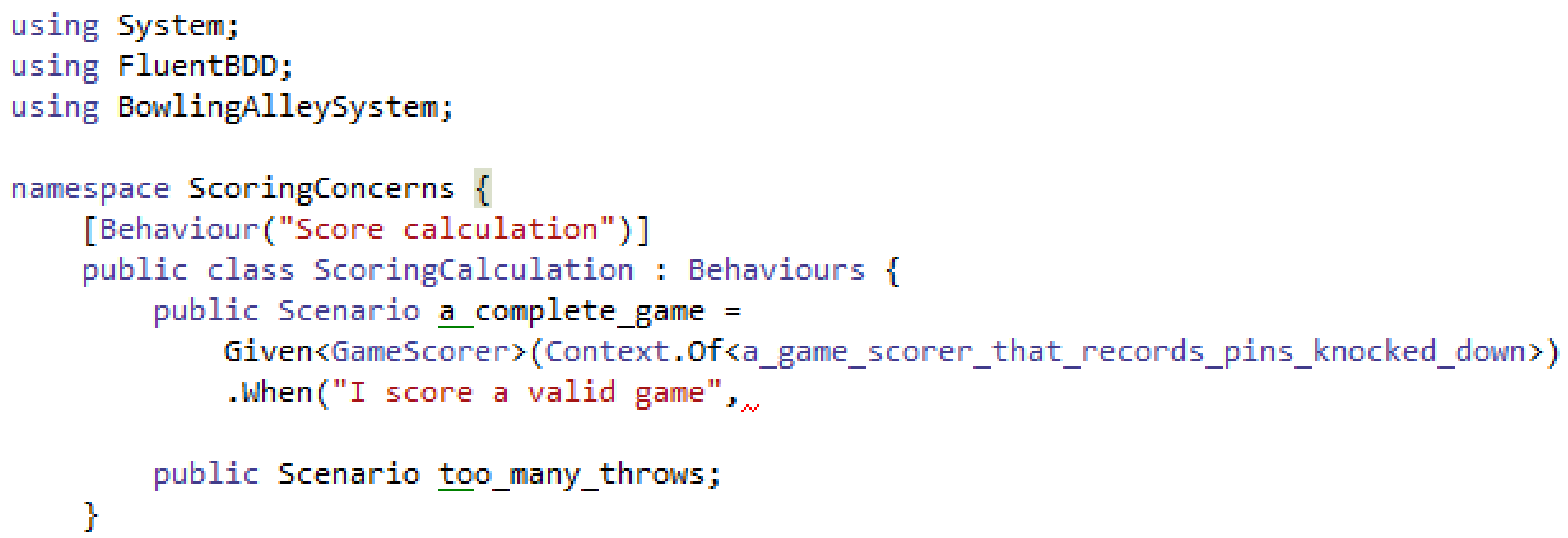


```
using System;
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .

        public Scenario too_many_throws;
    }
```

Use ReSharper to generate those classes and we'll worry about them later.

The compiler may complain until you make your context inherit from "`Context<GameScorer>`"

Using FluentBDD

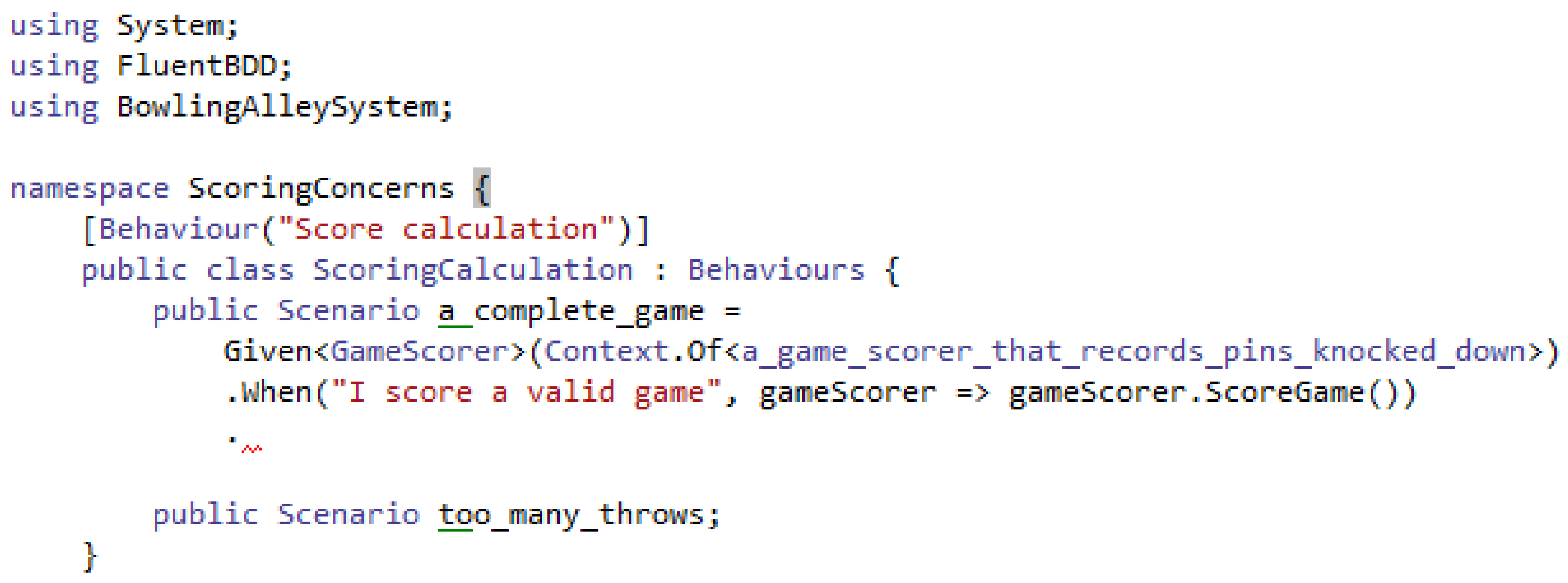


```
using System;
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game",

        public Scenario too_many_throws;
    }
```

Next, let's declare our Action.

We give it a plain English name in a string. This will appear in the unit tests.

Using FluentBDD

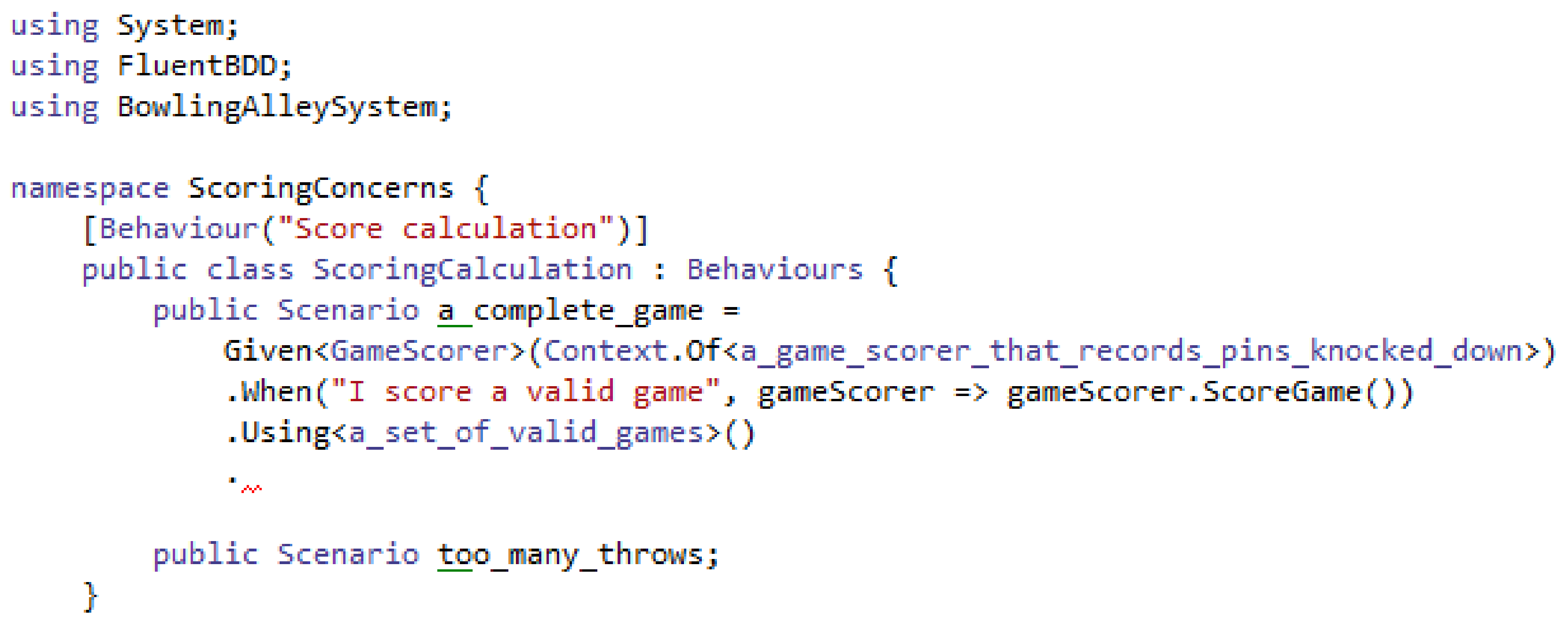


```
using System;
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .

        public Scenario too_many_throws;
    }
```

To complete the action, we say what scoring means in code.

We let ReSharper generate the method and we'll get back to it later.

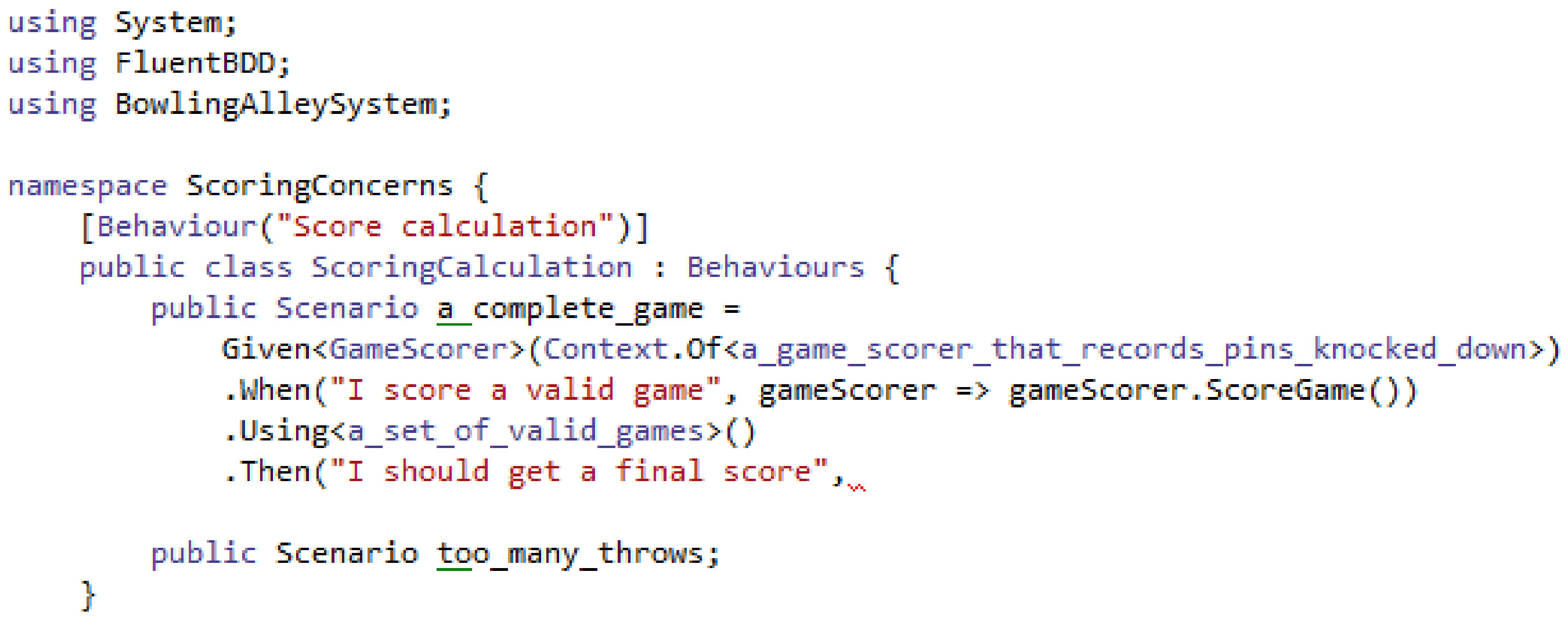
Using FluentBDD



```
using System;
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<a_set_of_valid_games>()
            .

        public Scenario too_many_throws;
    }
```

We add in a set of expectations, and let ReSharper create the class.

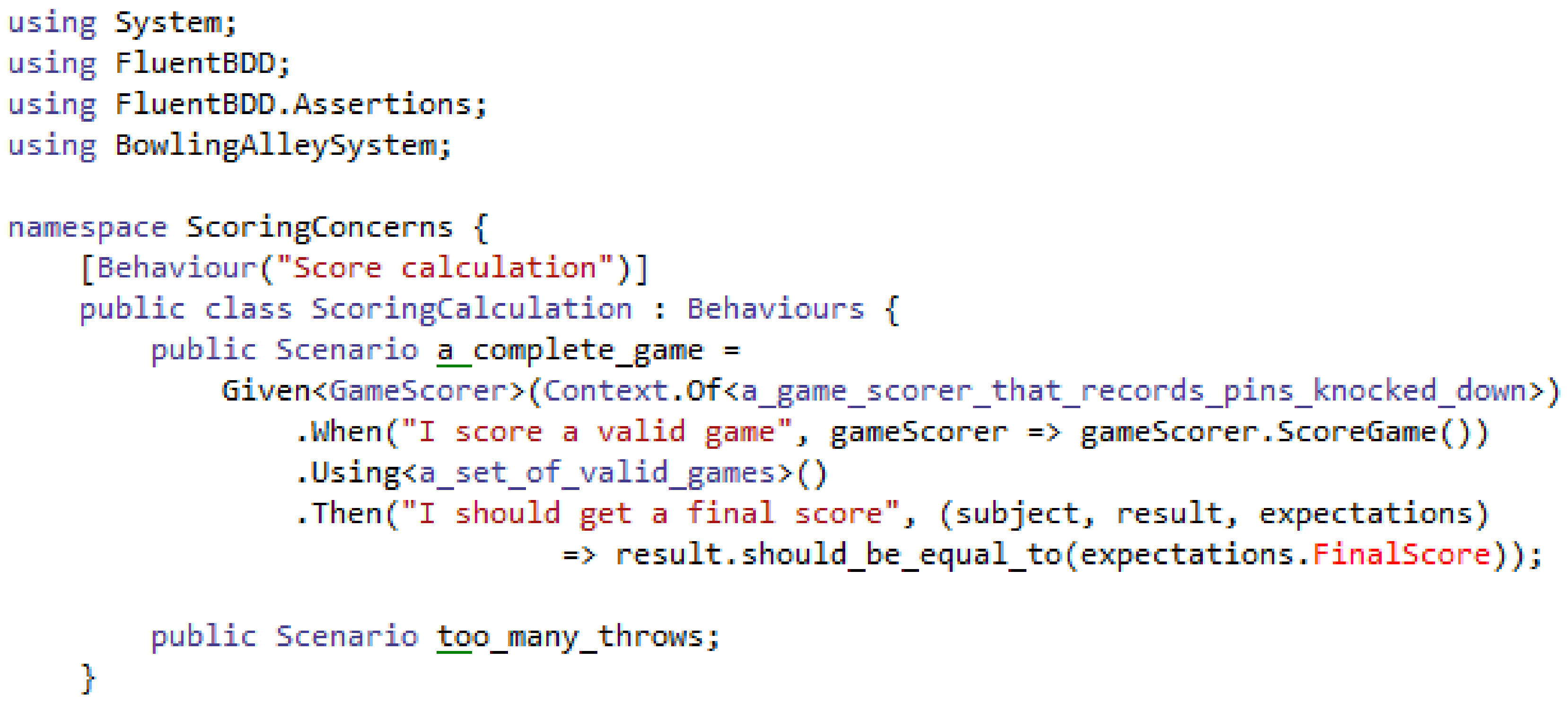
Using FluentBDD



```
using System;
using FluentBDD;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<a_set_of_valid_games>()
            .Then("I should get a final score",

        public Scenario too_many_throws;
    }
```

Finally we assert what should happen…

Using FluentBDD



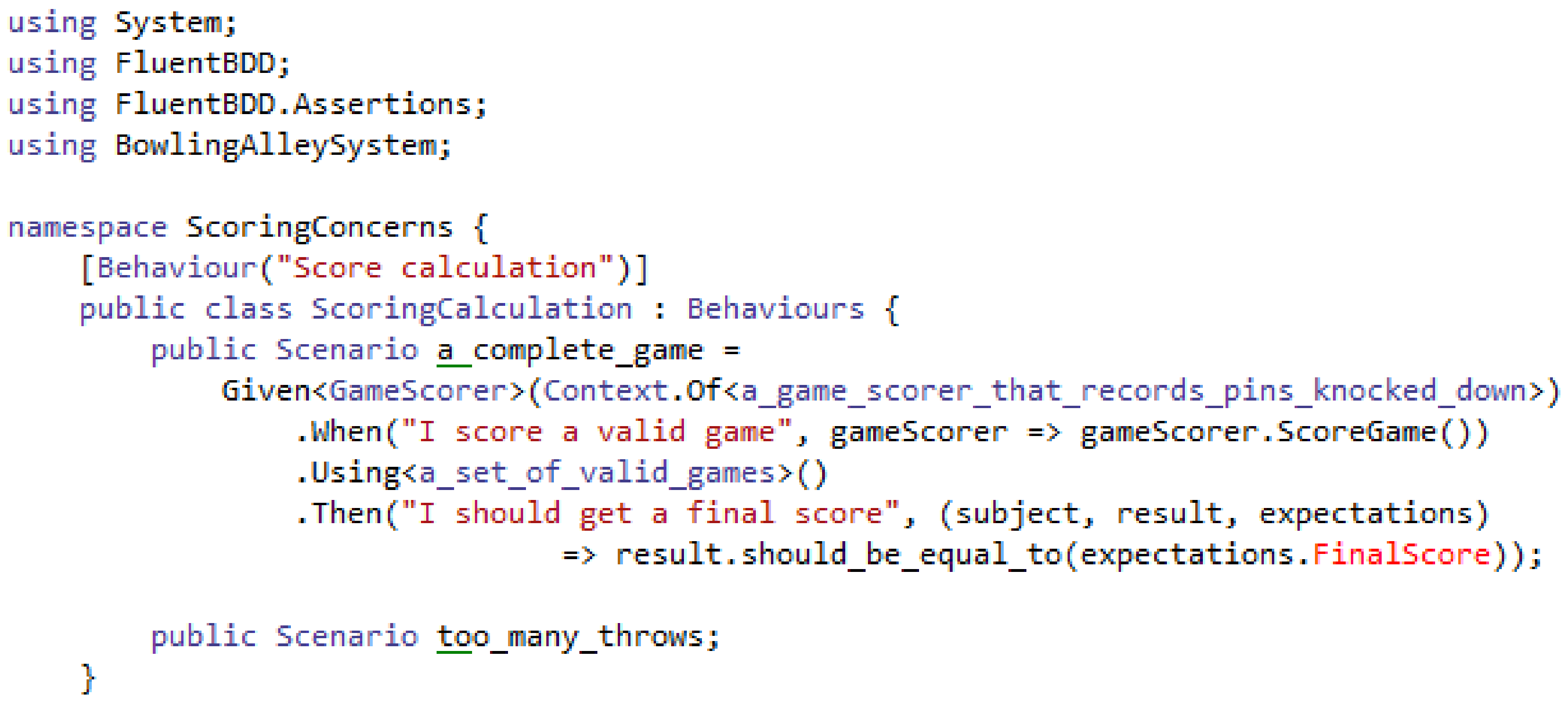
```
using System;
using FluentBDD;
using FluentBDD.Assertions;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                                => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }
```

… and explain this in code.

subject is the GameScorer being tested. result is returned from “ScoreGame()”.

expectations is the specific instance of our expectations being tested.

Using FluentBDD

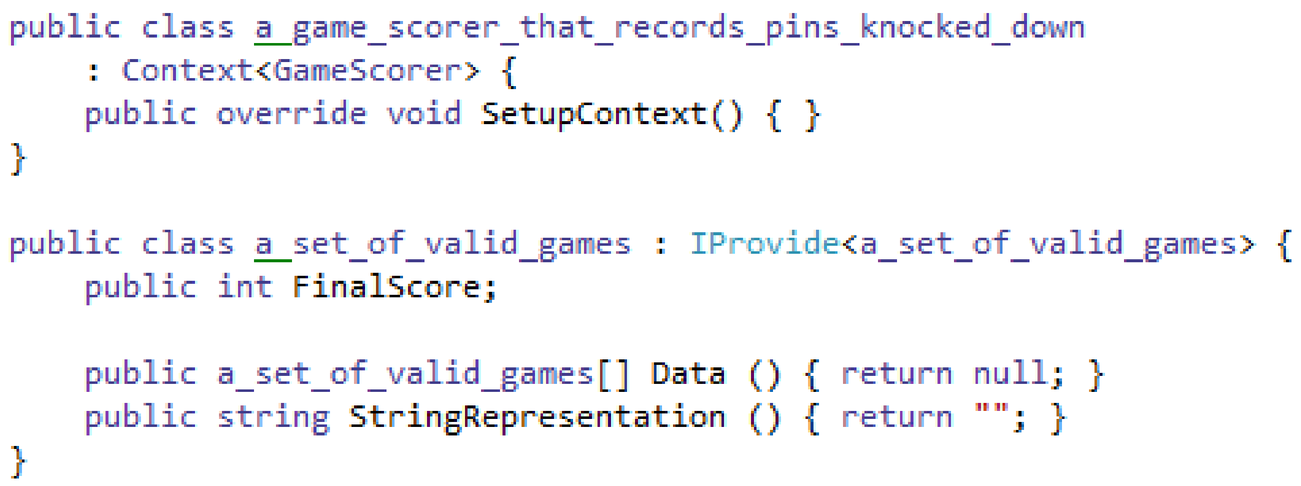


```
using System;
using FluentBDD;
using FluentBDD.Assertions;
using BowlingAlleySystem;

namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                                => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }
```

We also now have our first red field in our expectations: FinalScore.

We'll have ReSharper add this and then move on to the context.

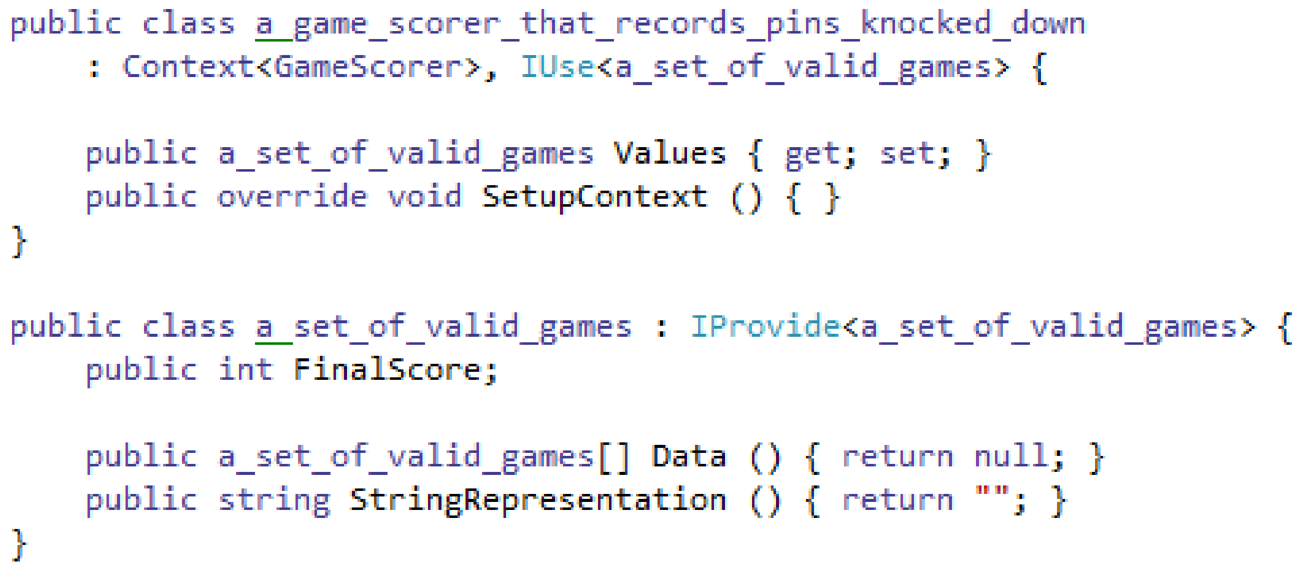
Using FluentBDD



```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer> {
    public override void SetupContext() { }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

We should have something that looks like this: a Context for a `GameScorer` class
And a set of expectations that implement `IProvide`.

Using FluentBDD

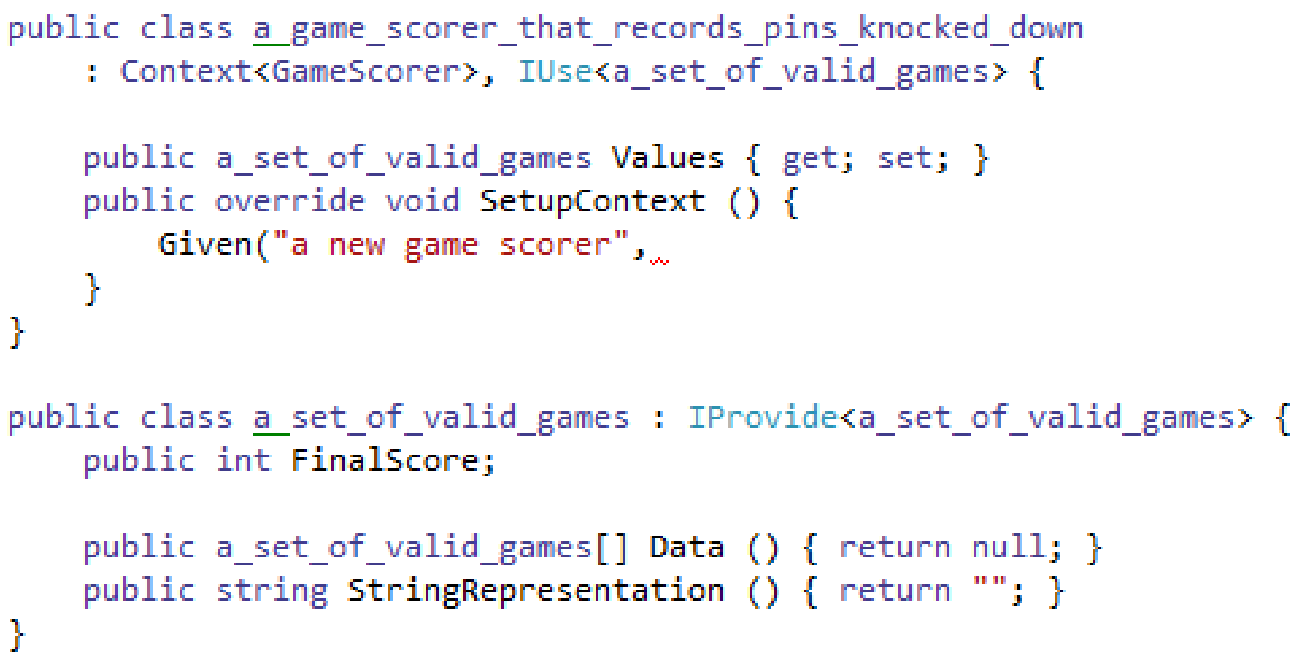


```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () { }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

Before we start building our context, let's use our expectations by adding `IUse` for our expectation type.

This interface requires a Values property we can use to build our context.

Using FluentBDD



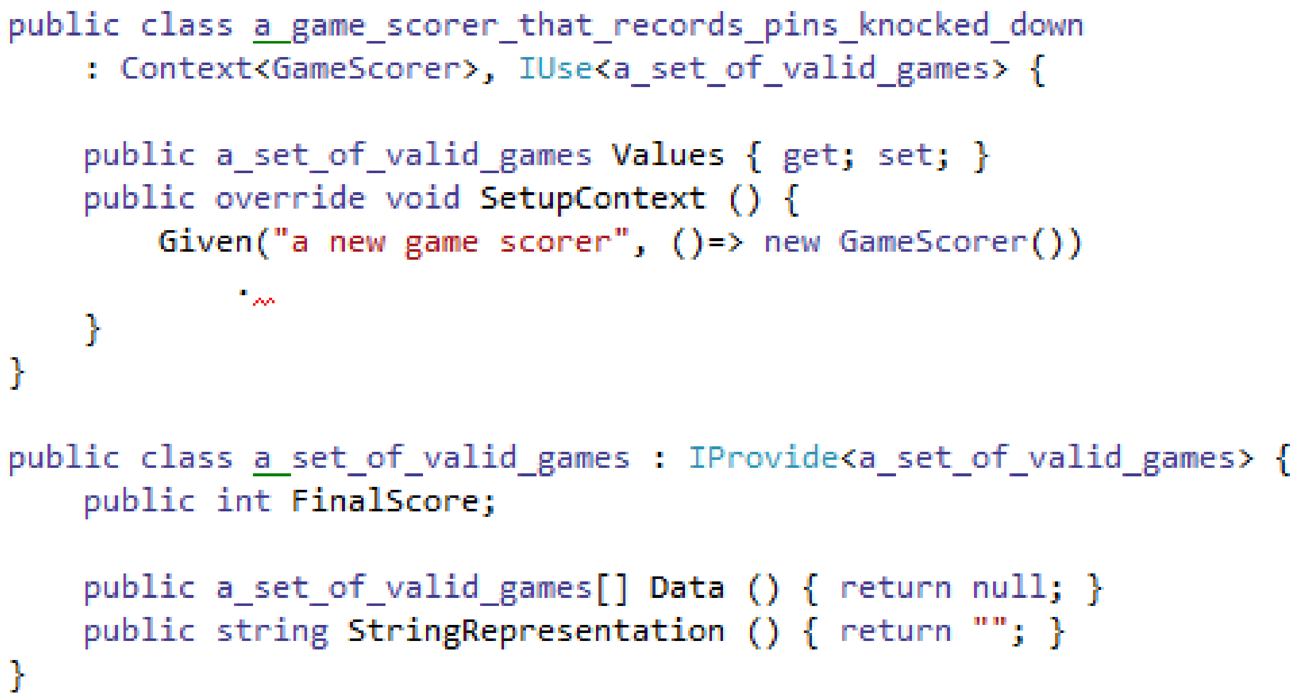
```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer",
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

Now we can define our context.

Context gives us the Given() method, which we use to start contexts.

The description we provide will appear in test results.

Using FluentBDD

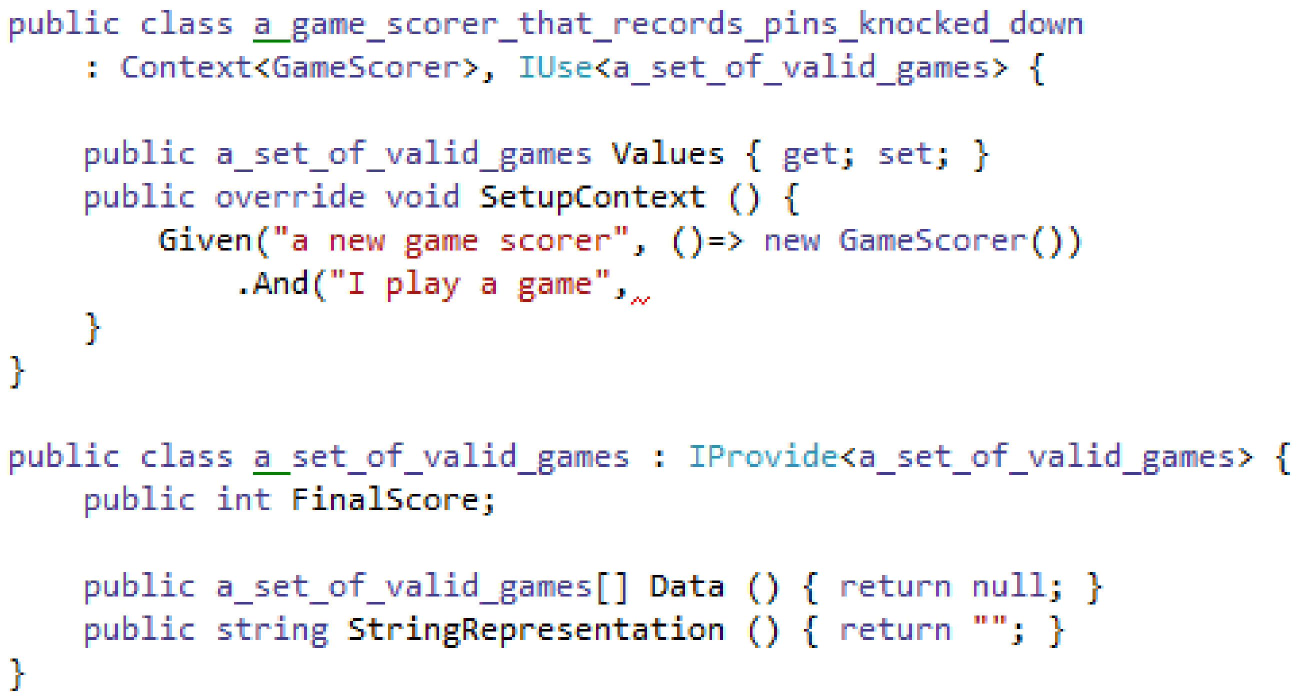


```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", ()=> new GameScorer())
            .
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

We complete the Given with a code explanation. This **must** return an instance of the subject.

Our context now needs to interact with the subject, to inject our expectations.

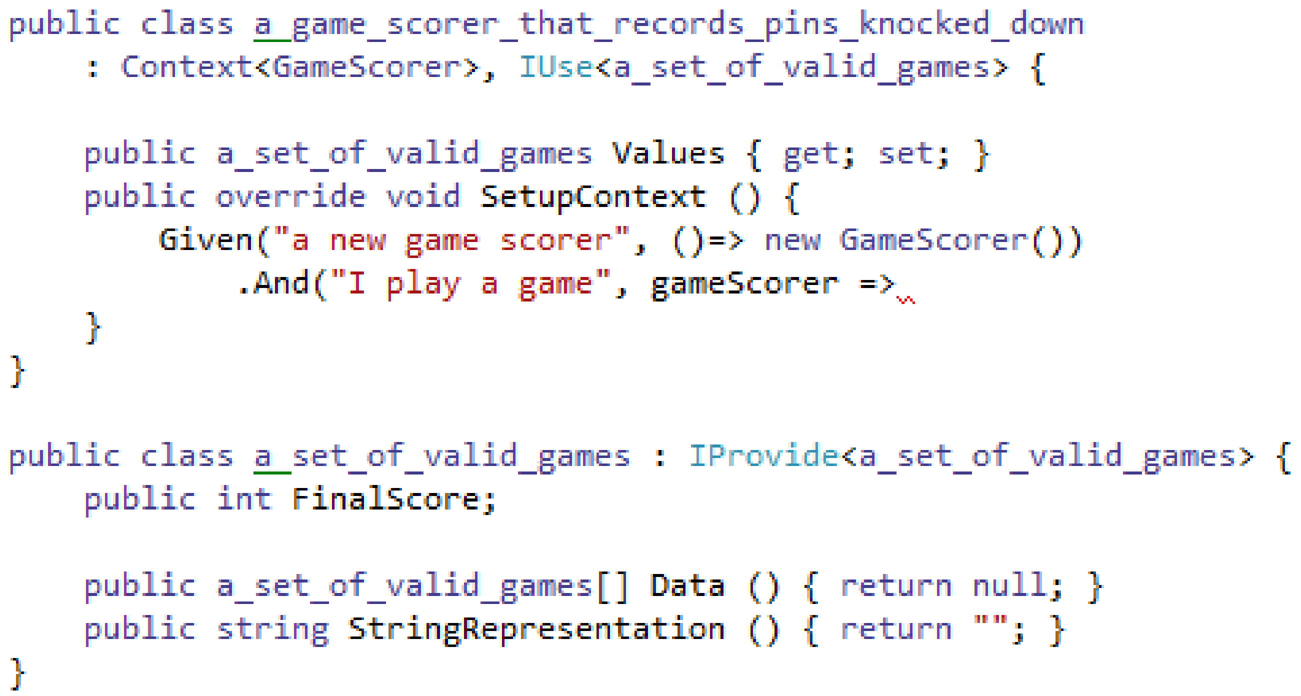
Using FluentBDD



```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", ()=> new GameScorer())
            .And("I play a game",
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

We can add an And to the end of our given, with a description (which will appear in the tests)…

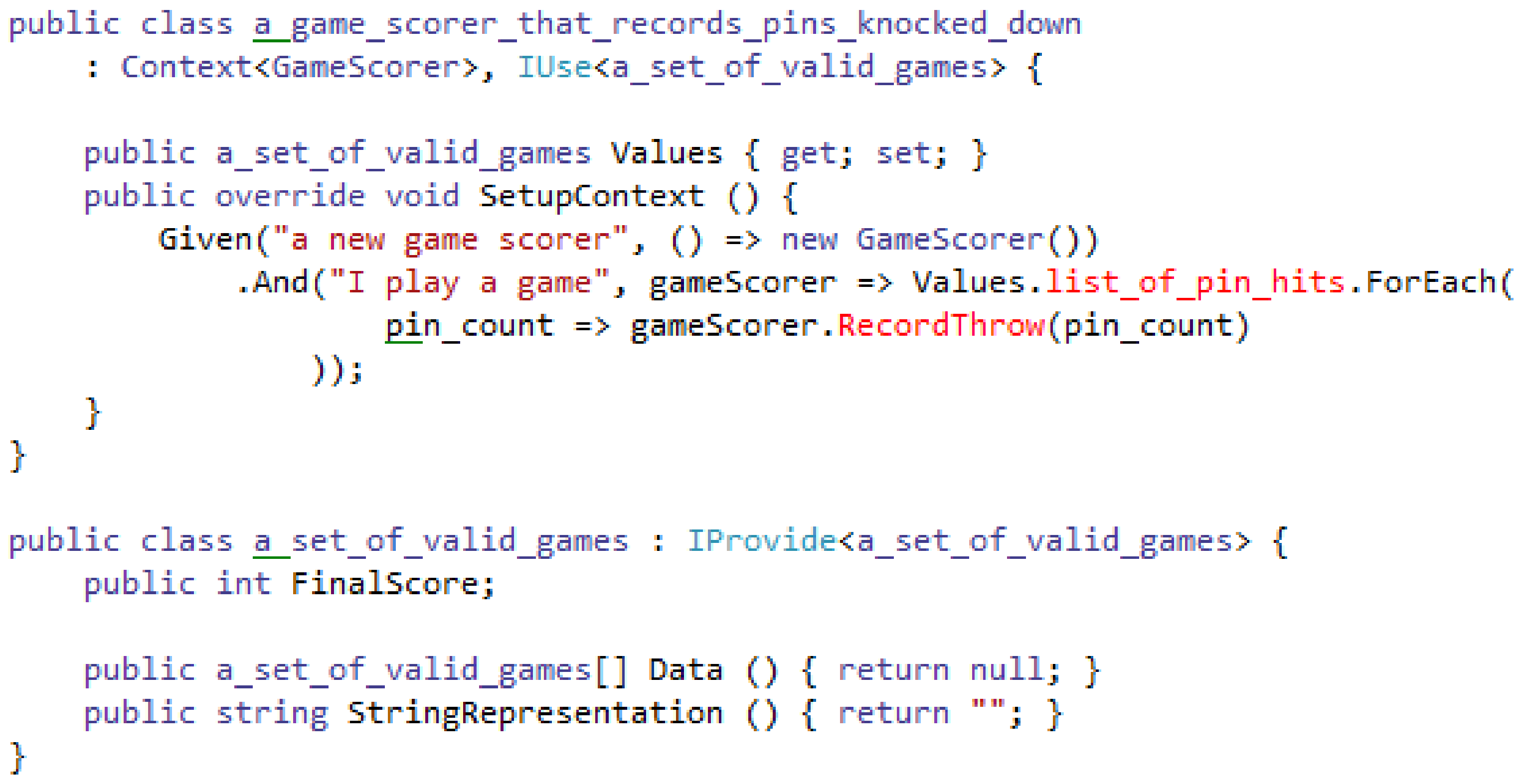
Using FluentBDD



```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", ()=> new GameScorer())
            .And("I play a game", gameScorer =>
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

…and a code explanation, which takes the subject we’ve made.

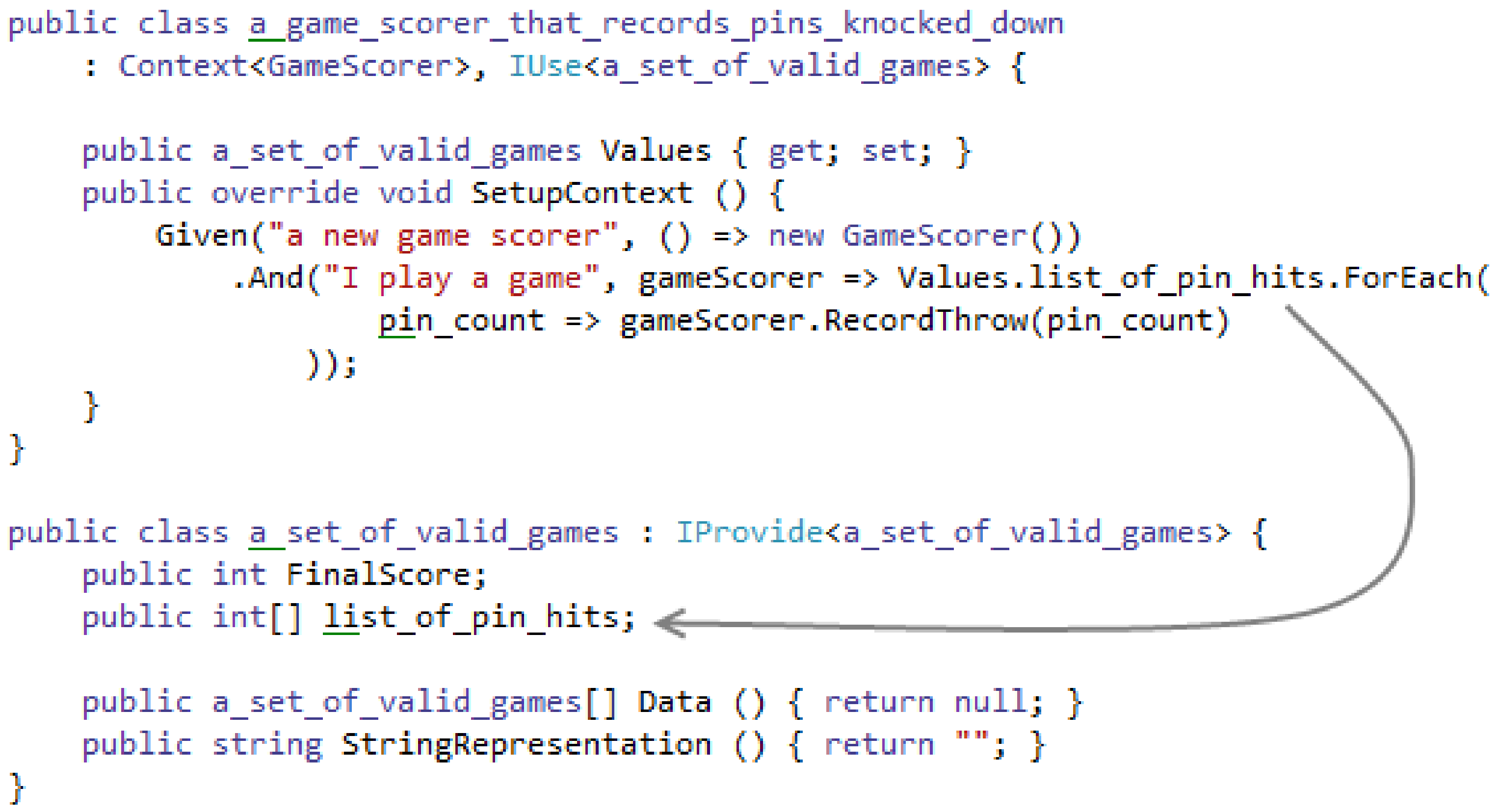
Using FluentBDD



```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", () => new GameScorer())
            .And("I play a game", gameScorer => Values.list_of_pin_hits.ForEach(
                    pin_count => gameScorer.RecordThrow(pin_count)
                ));
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

Let’s fill this out with some ‘wish code’ and let ReSharper generate the required parts.

Using FluentBDD

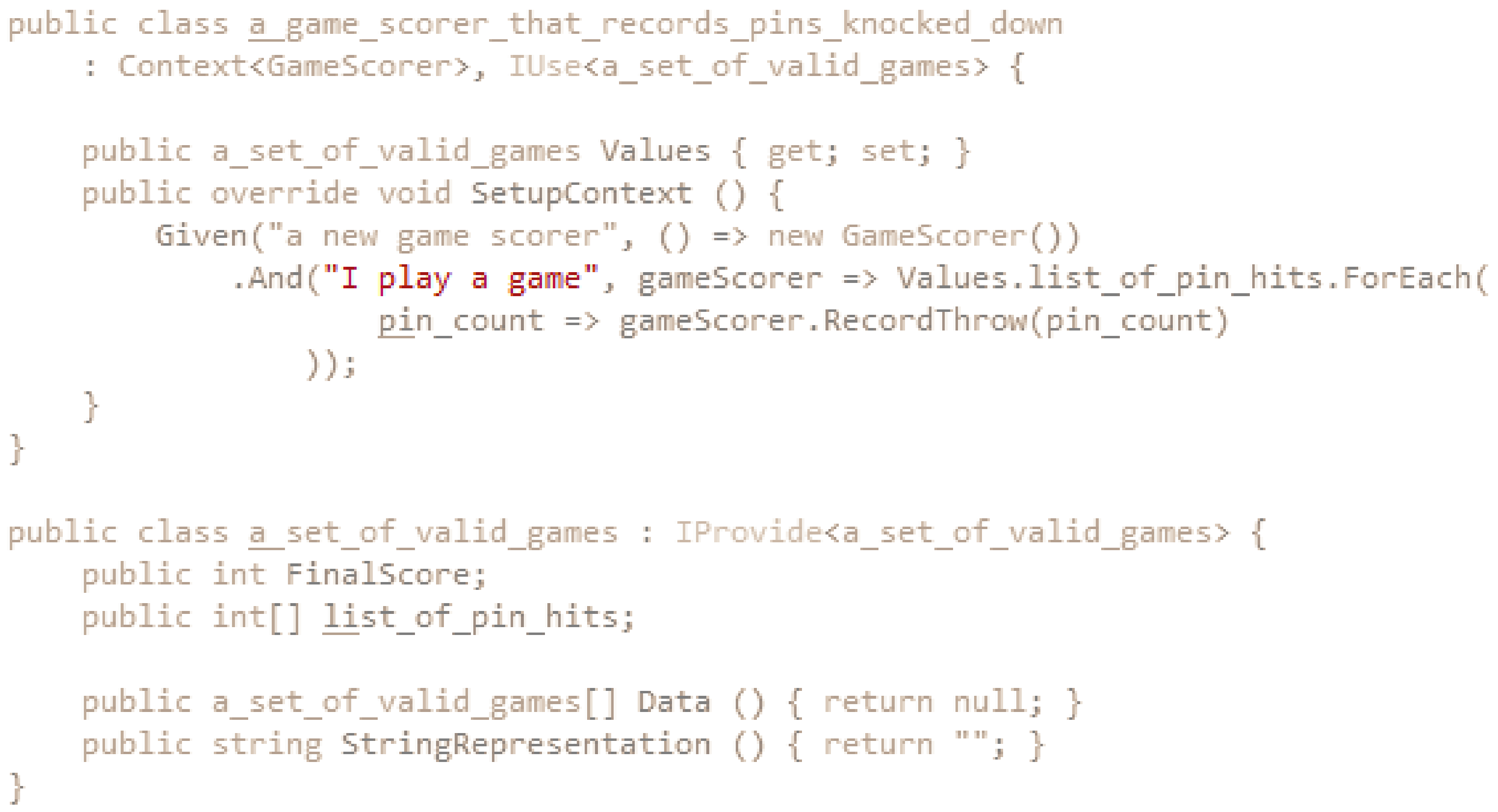


```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", () => new GameScorer())
            .And("I play a game", gameScorer => Values.list_of_pin_hits.ForEach(
                    pin_count => gameScorer.RecordThrow(pin_count)
                ));
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

Note the new list of pin hits in our expectations.

You can chain as many `.And()`s together as you need. Try to keep a balance between number of ands and complexity of the code.

Using FluentBDD

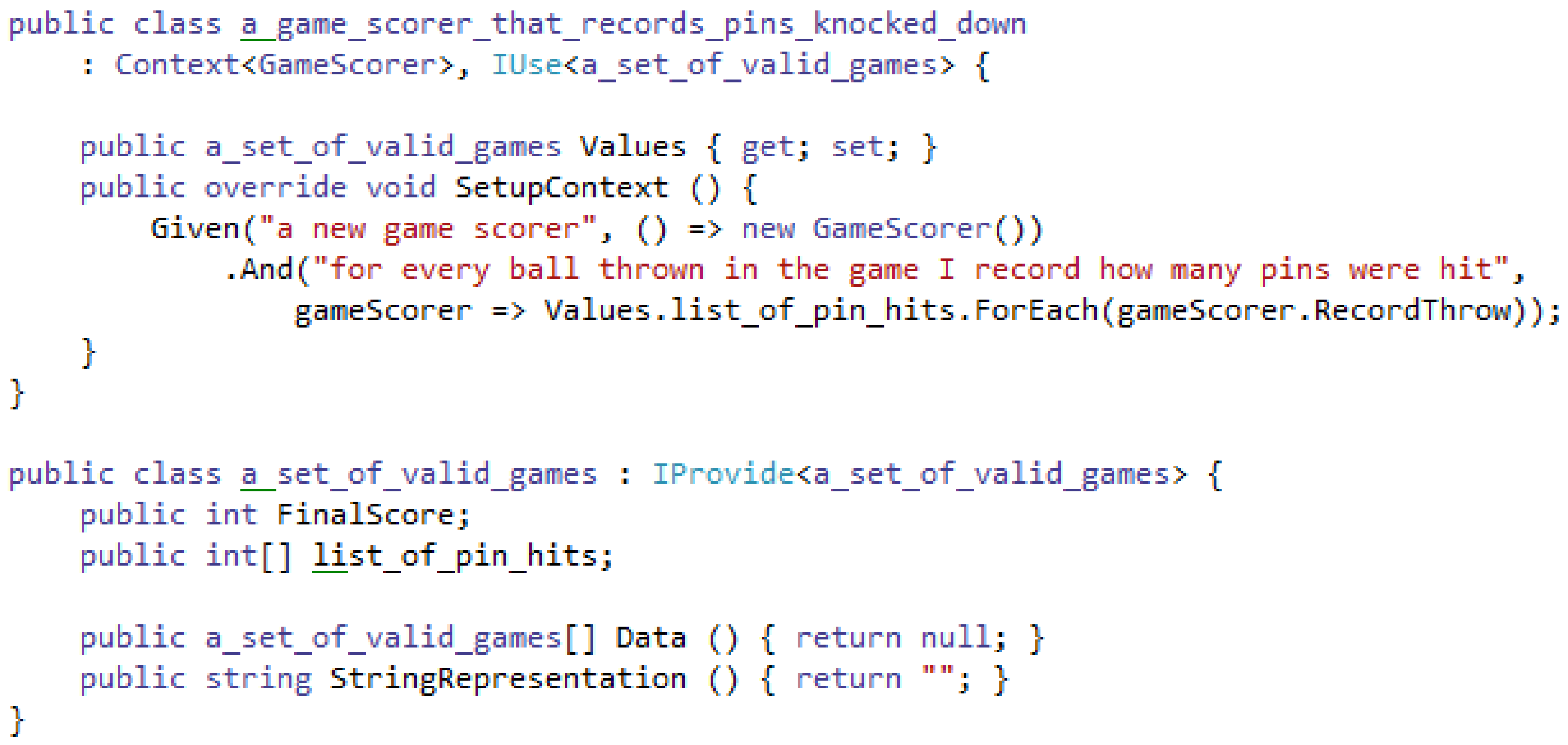


```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", () => new GameScorer())
            .And("I play a game", gameScorer => Values.list_of_pin_hits.ForEach(
                    pin_count => gameScorer.RecordThrow(pin_count)
                ));
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

Next, I review my code context to make sure everything is good.

I think the description could be better.

Using FluentBDD

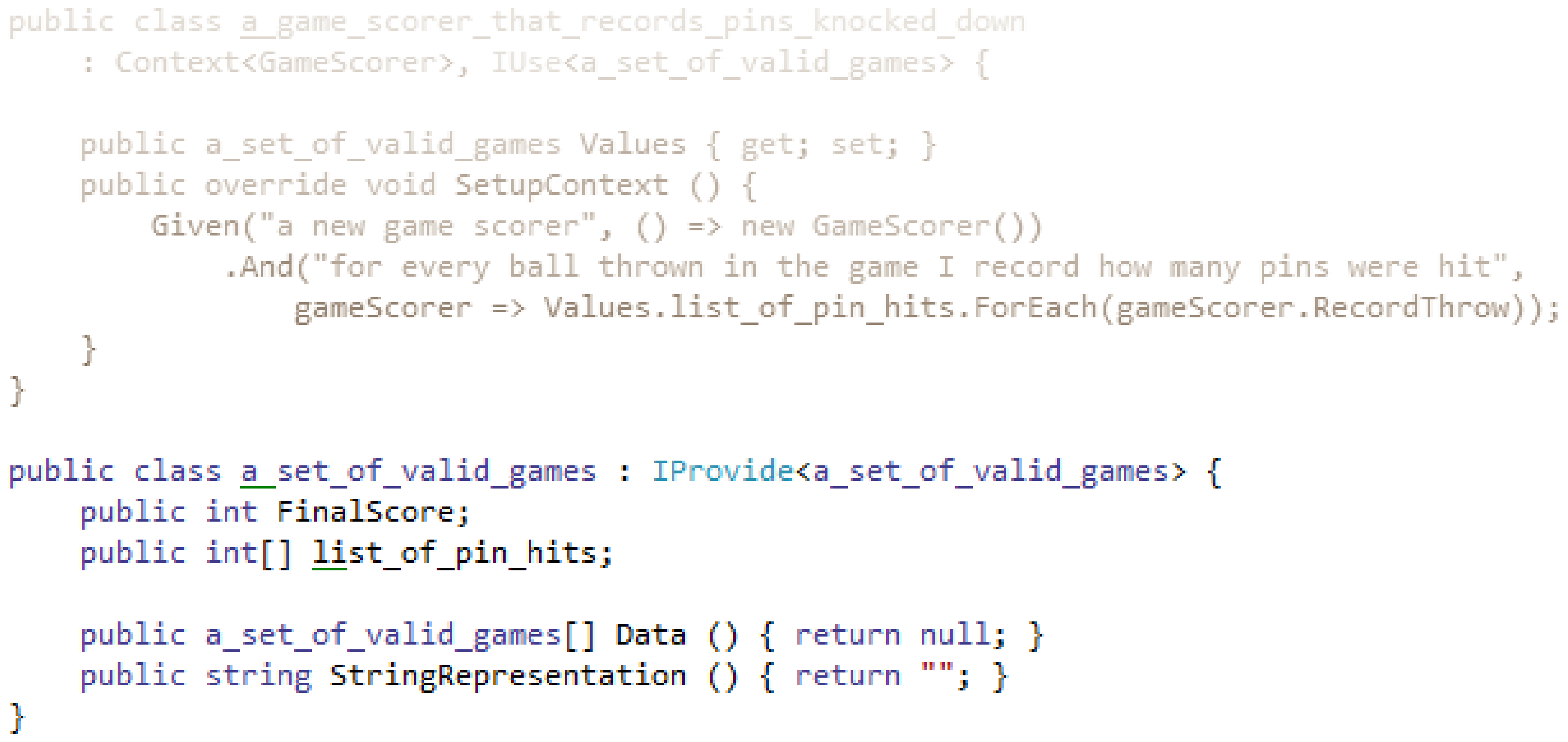


```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", () => new GameScorer())
            .And("for every ball thrown in the game I record how many pins were hit",
                gameScorer => Values.list_of_pin_hits.ForEach(gameScorer.RecordThrow));
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

It’s a bit wordy, but much better describes what is happening.

The goal with FluentBDD is clarity, so make sure to review and refactor your specifications like you would with your code.

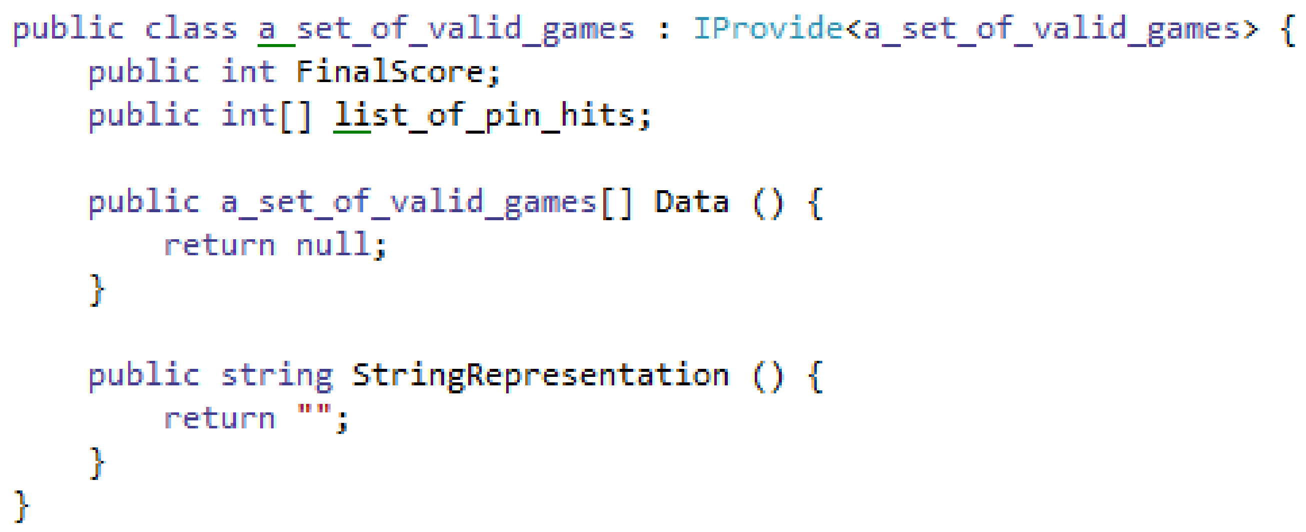
Using FluentBDD



```
public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<a_set_of_valid_games> {

    public a_set_of_valid_games Values { get; set; }
    public override void SetupContext () {
        Given("a new game scorer", () => new GameScorer())
            .And("for every ball thrown in the game I record how many pins were hit",
                gameScorer => Values.list_of_pin_hits.ForEach(gameScorer.RecordThrow));
    }
}

public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () { return null; }
    public string StringRepresentation () { return ""; }
}
```

That’s the context complete, now let’s have a look at our expectations.

Using FluentBDD



```
public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () {
        return null;
    }

    public string StringRepresentation () {
        return "";
    }
}
```

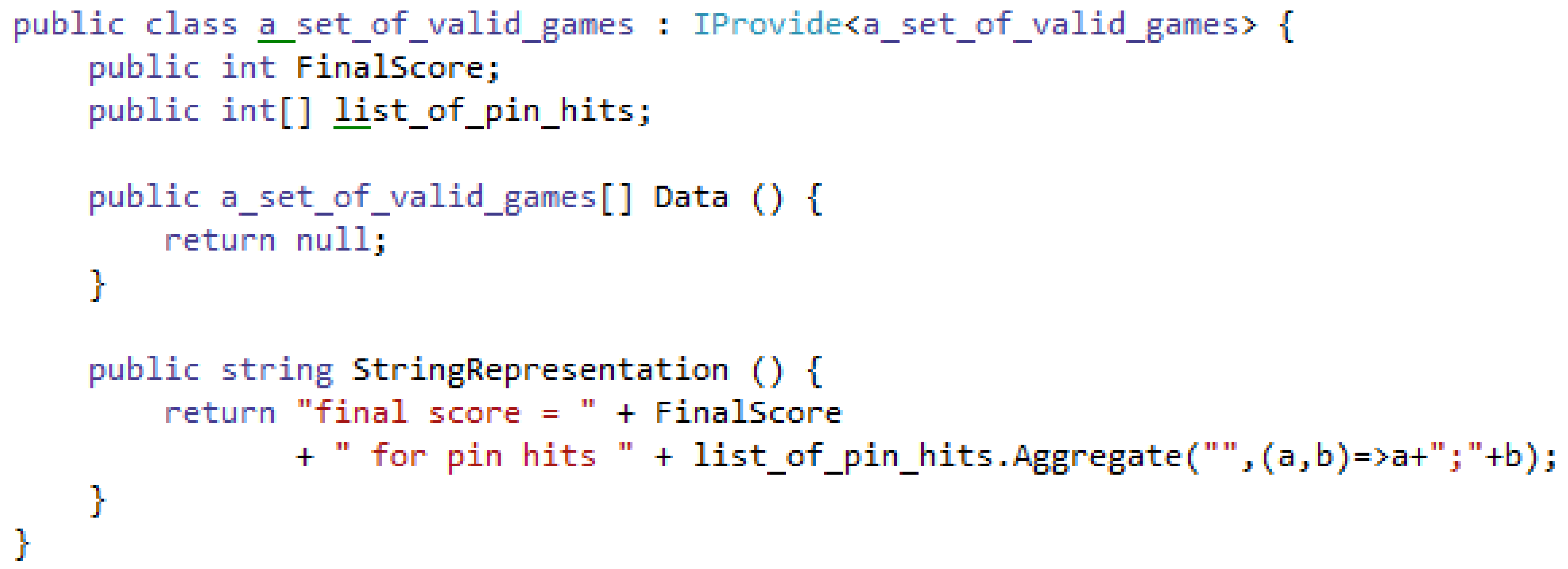
Apart from the public fields we’ve auto-generated from our context and scenario, there are two methods required by `IProvide`:

Data

and

StringRepresentation

Using FluentBDD

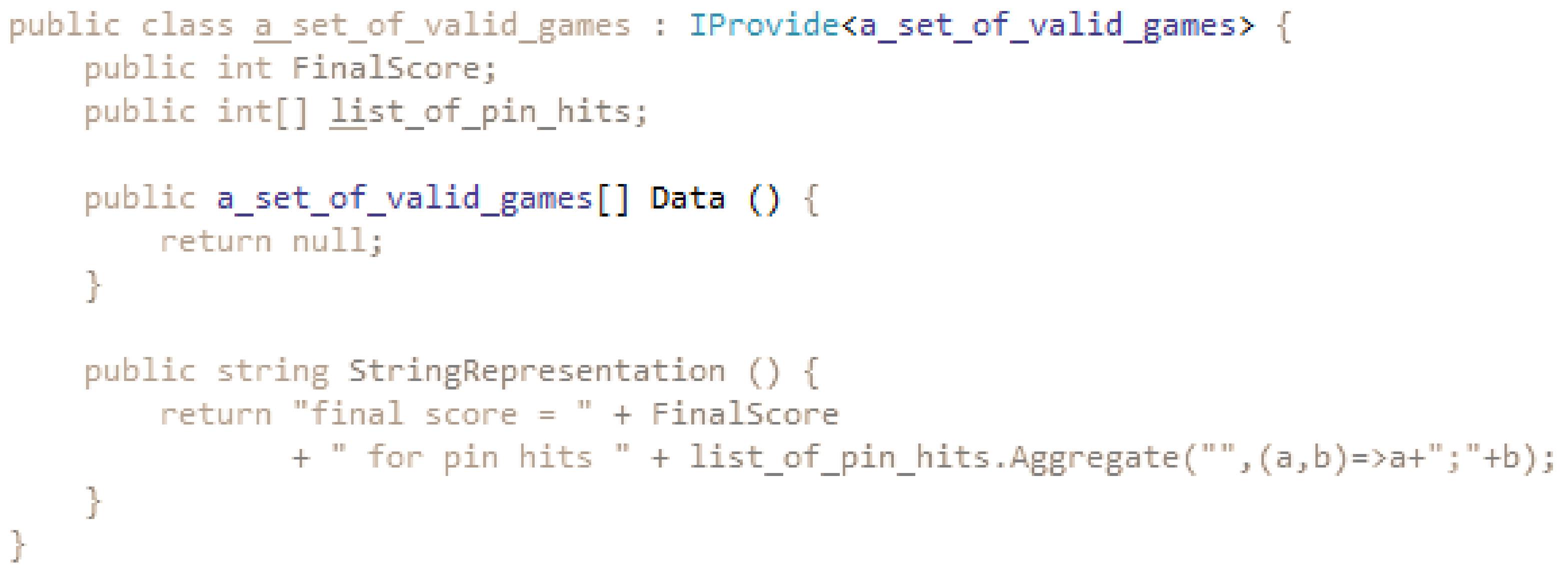


```
public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () {
        return null;
    }

    public string StringRepresentation () {
        return "final score = " + FinalScore
              + " for pin hits " + list_of_pin_hits.Aggregate("",(a,b)=>a+";"+b);
    }
}
```

StringRepresentation is used to name each expectation in the unit tests. The name is entirely up to you, but should be enough to identify the test case that has passed or failed.

For now, let's just show the pin hits and score.

Using FluentBDD

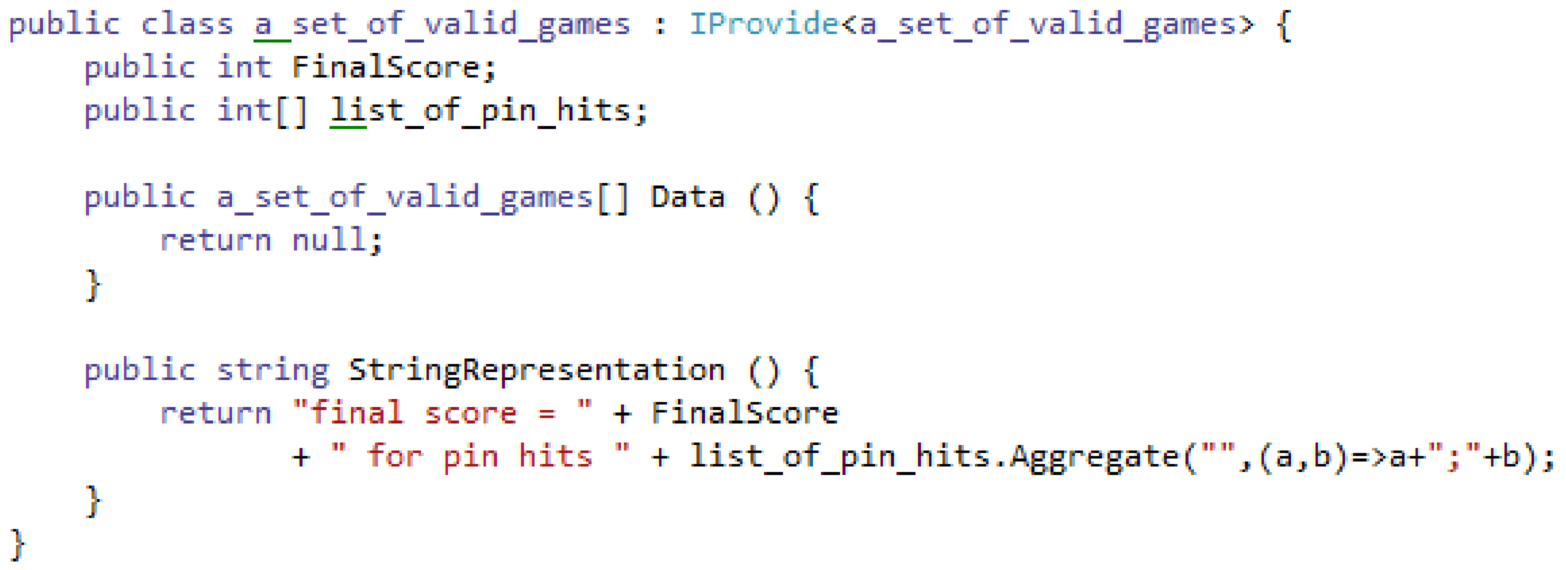


```
public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () {
        return null;
    }

    public string StringRepresentation () {
        return "final score = " + FinalScore
               + " for pin hits " + list_of_pin_hits.Aggregate("",(a,b)=>a+";"+b);
    }
}
```

Next, **Data** returns an array of instances of what we claim to provide.

Note that the return type of `Data` matches our `IProvide` type.

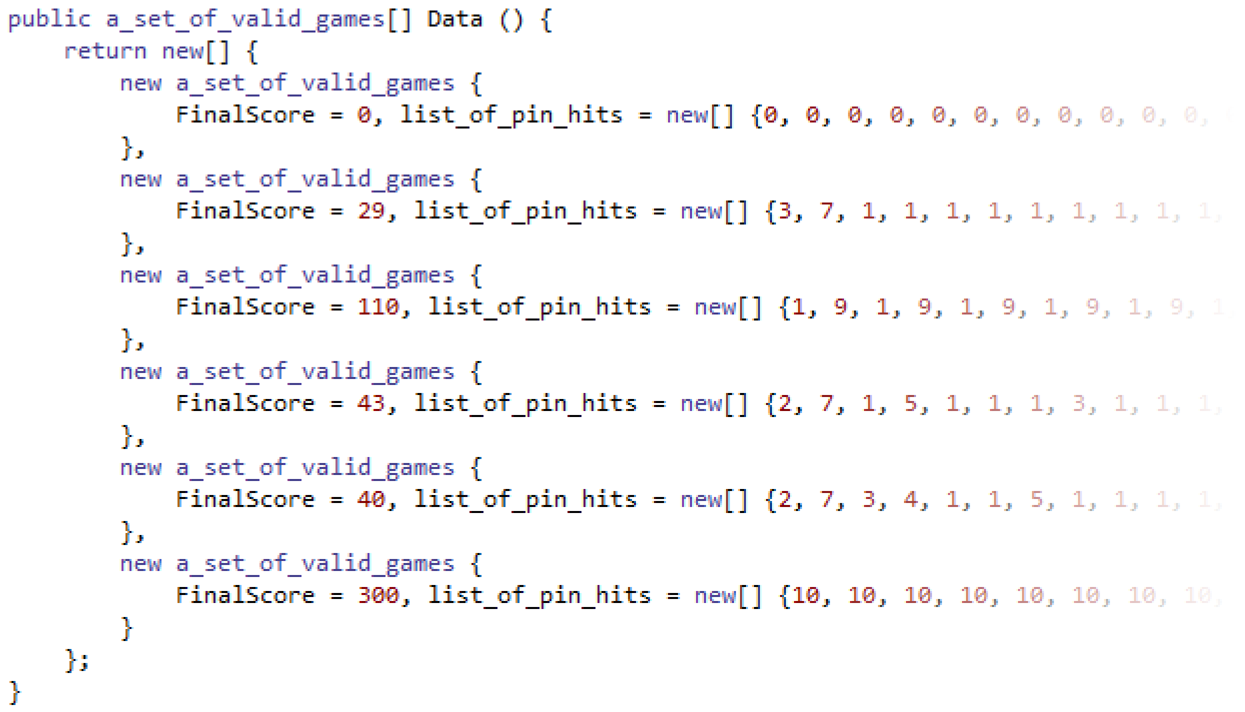
Using FluentBDD



```
public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
    public int FinalScore;
    public int[] list_of_pin_hits;

    public a_set_of_valid_games[] Data () {
        return null;
    }

    public string StringRepresentation () {
        return "final score = " + FinalScore
               + " for pin hits " + list_of_pin_hits.Aggregate("",(a,b)=>a+";"+b);
    }
}
```

For this expectation, we will go the simple route of creating and returning a set of example games.

Using FluentBDD



```
public a_set_of_valid_games[] Data () {
    return new[] {
        new a_set_of_valid_games {
            FinalScore = 0, list_of_pin_hits = new[] {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        },
        new a_set_of_valid_games {
            FinalScore = 29, list_of_pin_hits = new[] {3, 7, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        },
        new a_set_of_valid_games {
            FinalScore = 110, list_of_pin_hits = new[] {1, 9, 1, 9, 1, 9, 1, 9, 1, 9, 1,
        },
        new a_set_of_valid_games {
            FinalScore = 43, list_of_pin_hits = new[] {2, 7, 1, 5, 1, 1, 1, 3, 1, 1, 1,
        },
        new a_set_of_valid_games {
            FinalScore = 40, list_of_pin_hits = new[] {2, 7, 3, 4, 1, 1, 5, 1, 1, 1, 1,
        },
        new a_set_of_valid_games {
            FinalScore = 300, list_of_pin_hits = new[] {10, 10, 10, 10, 10, 10, 10, 10,
        }
    };
}
```

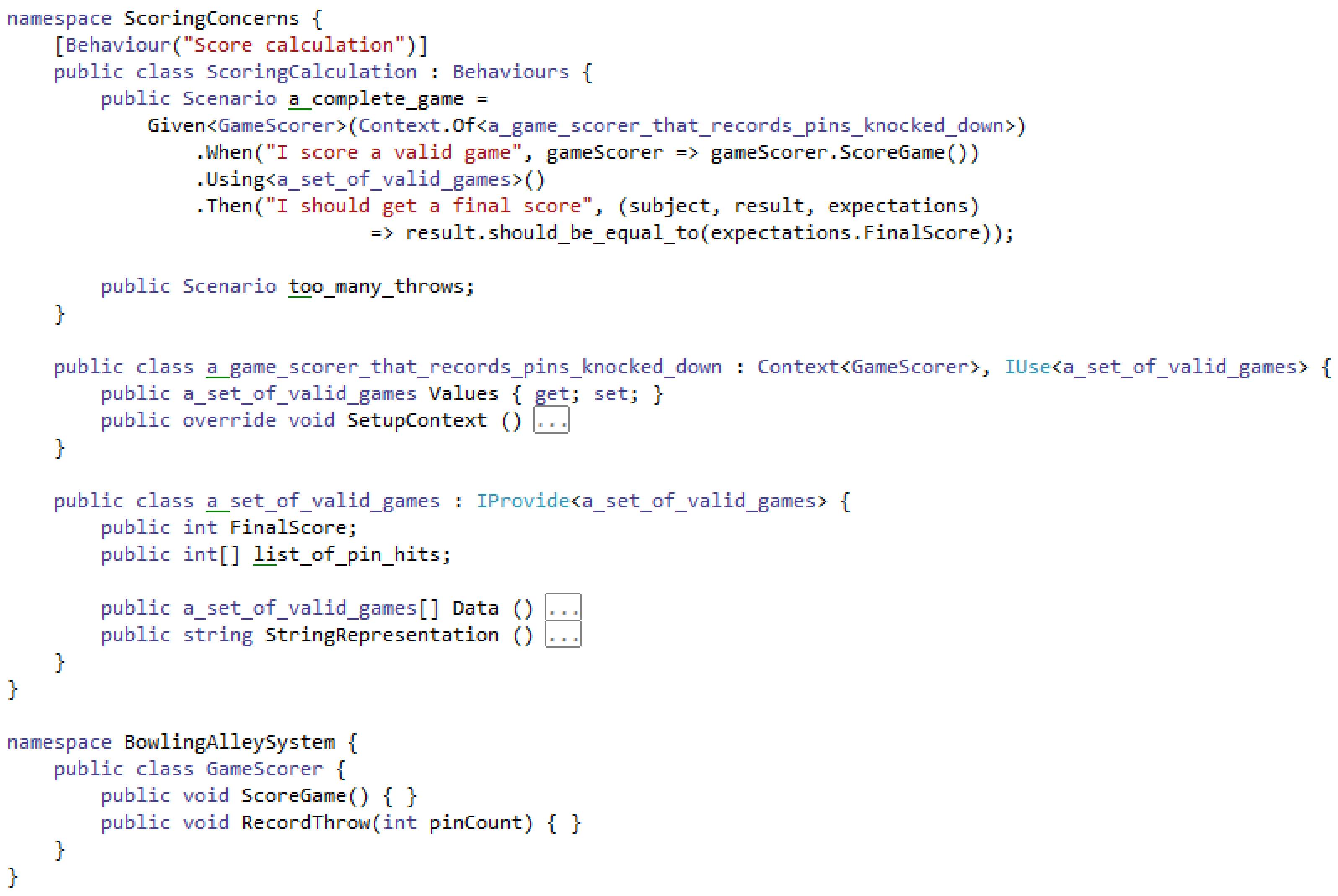
Giving us a Data() method like this.

It doesn’t matter how you generate the output for Data(), as long as it’s an array of your provided type.

Using FluentBDD

Now we’ve filled in all the blanks, it’s time to review where we are.

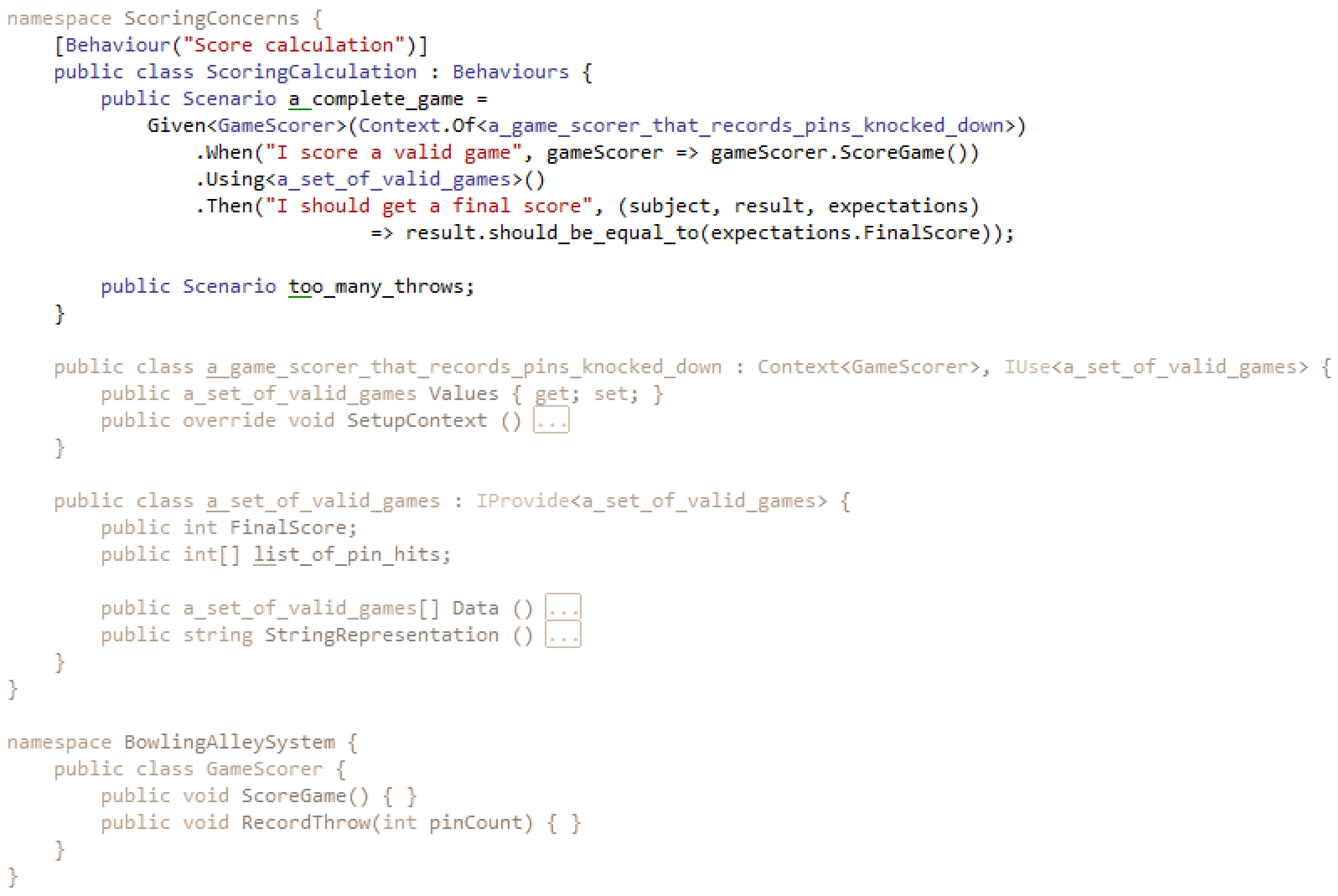
Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

We should have some code that looks like this

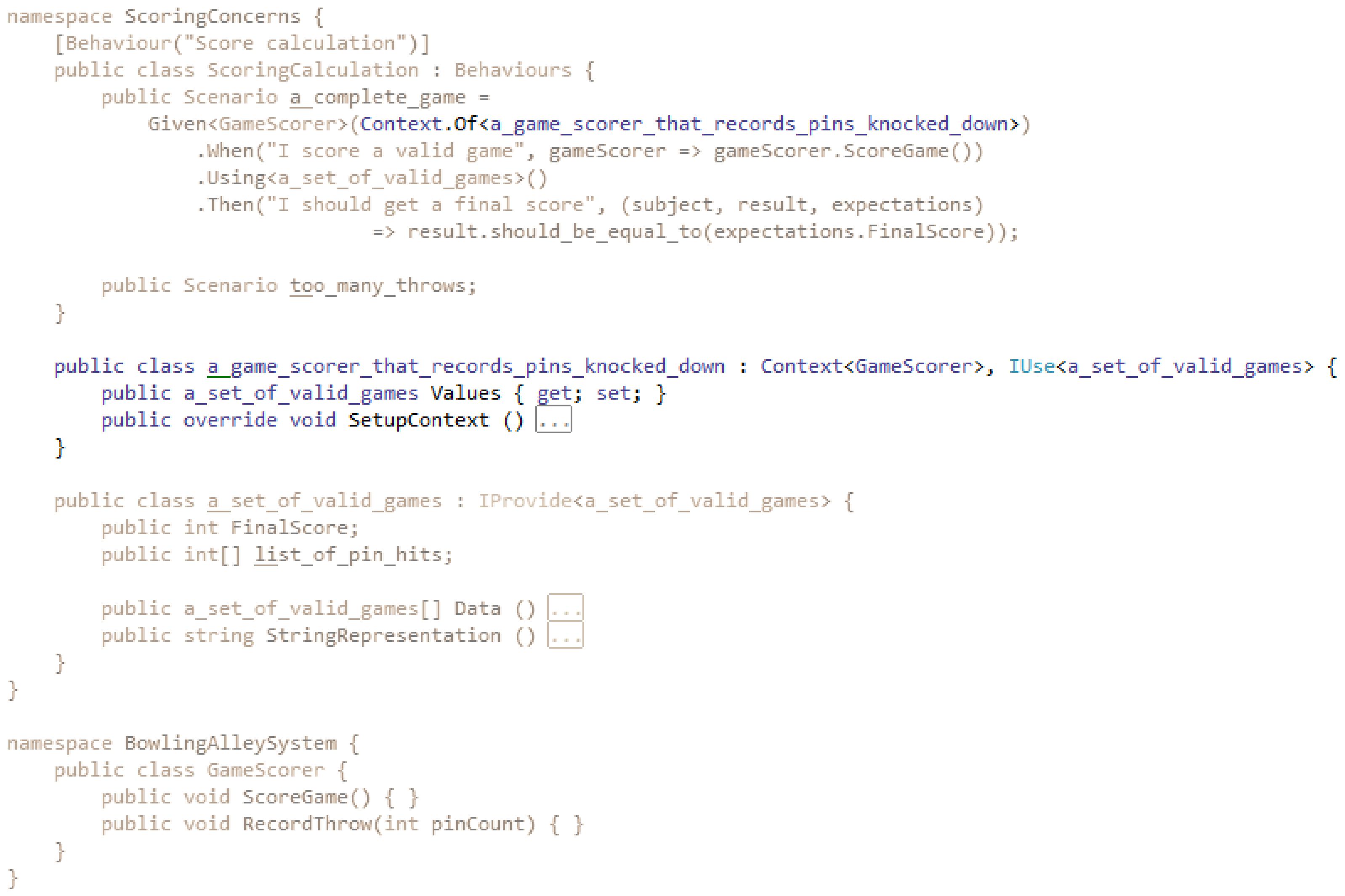
Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

We have a **Behaviour** set with one completed **Scenario** and one empty scenario.

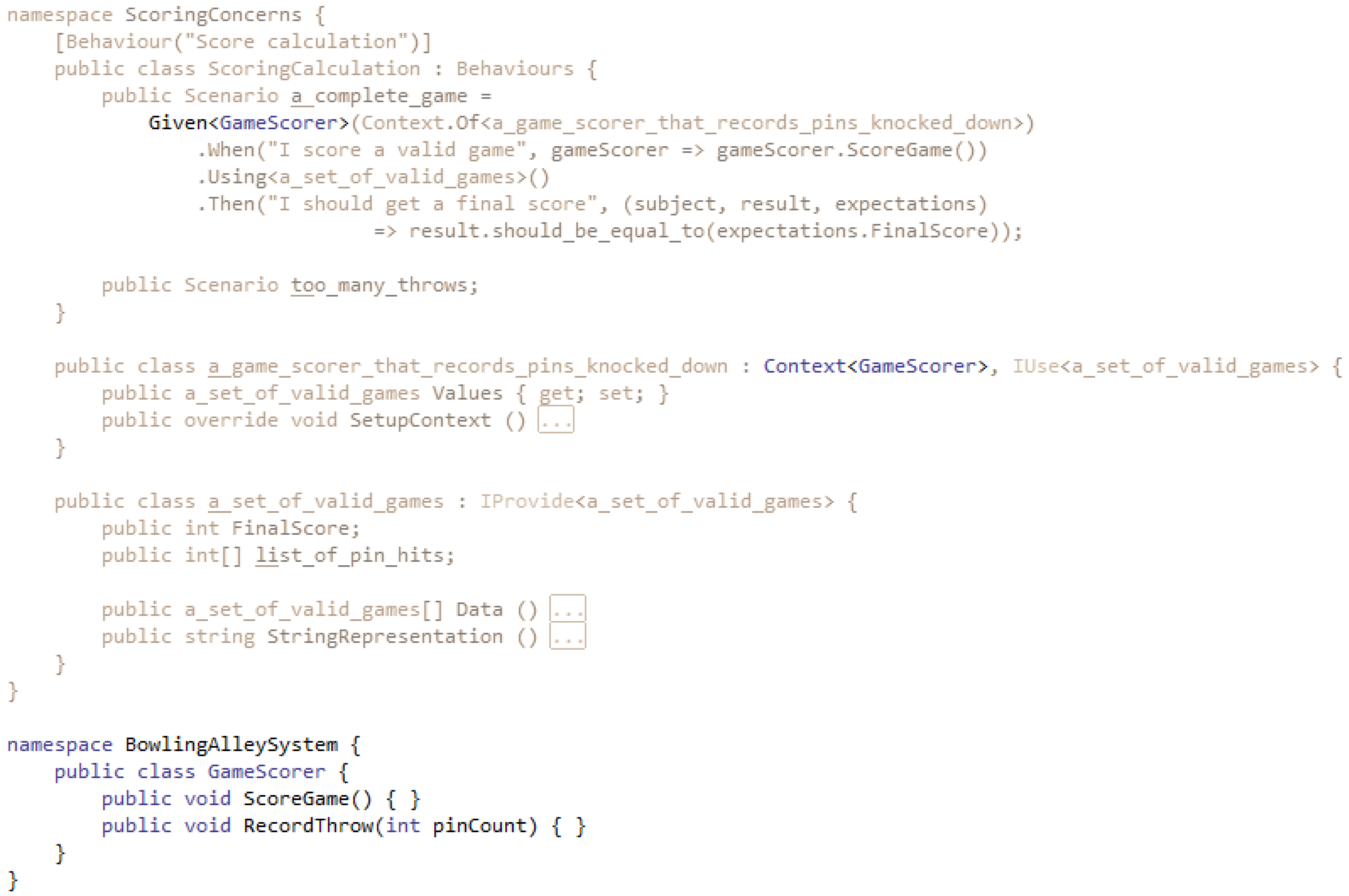
Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

The **Scenario** has a **Context**

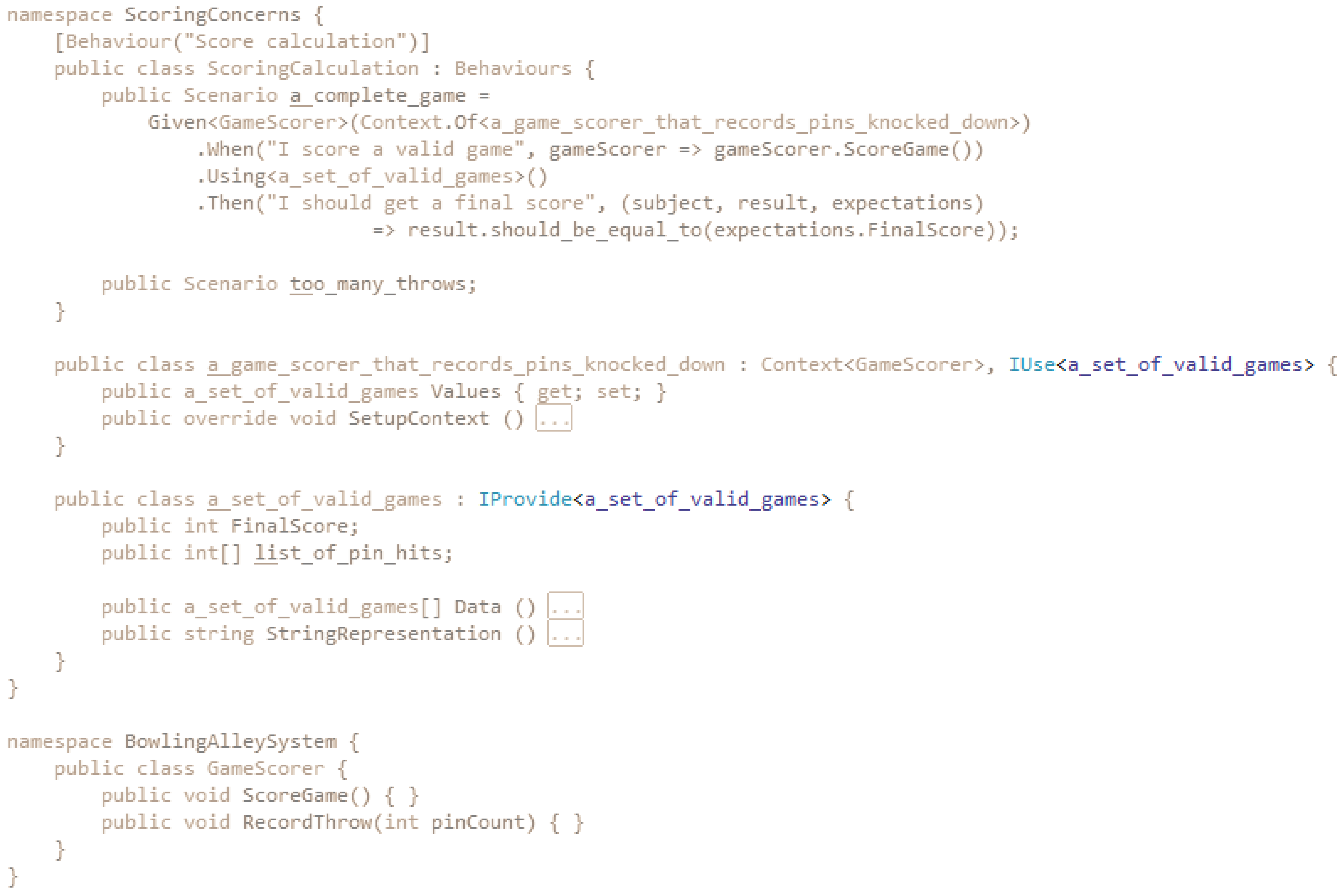
Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

The **Context** relates to a **Subject**

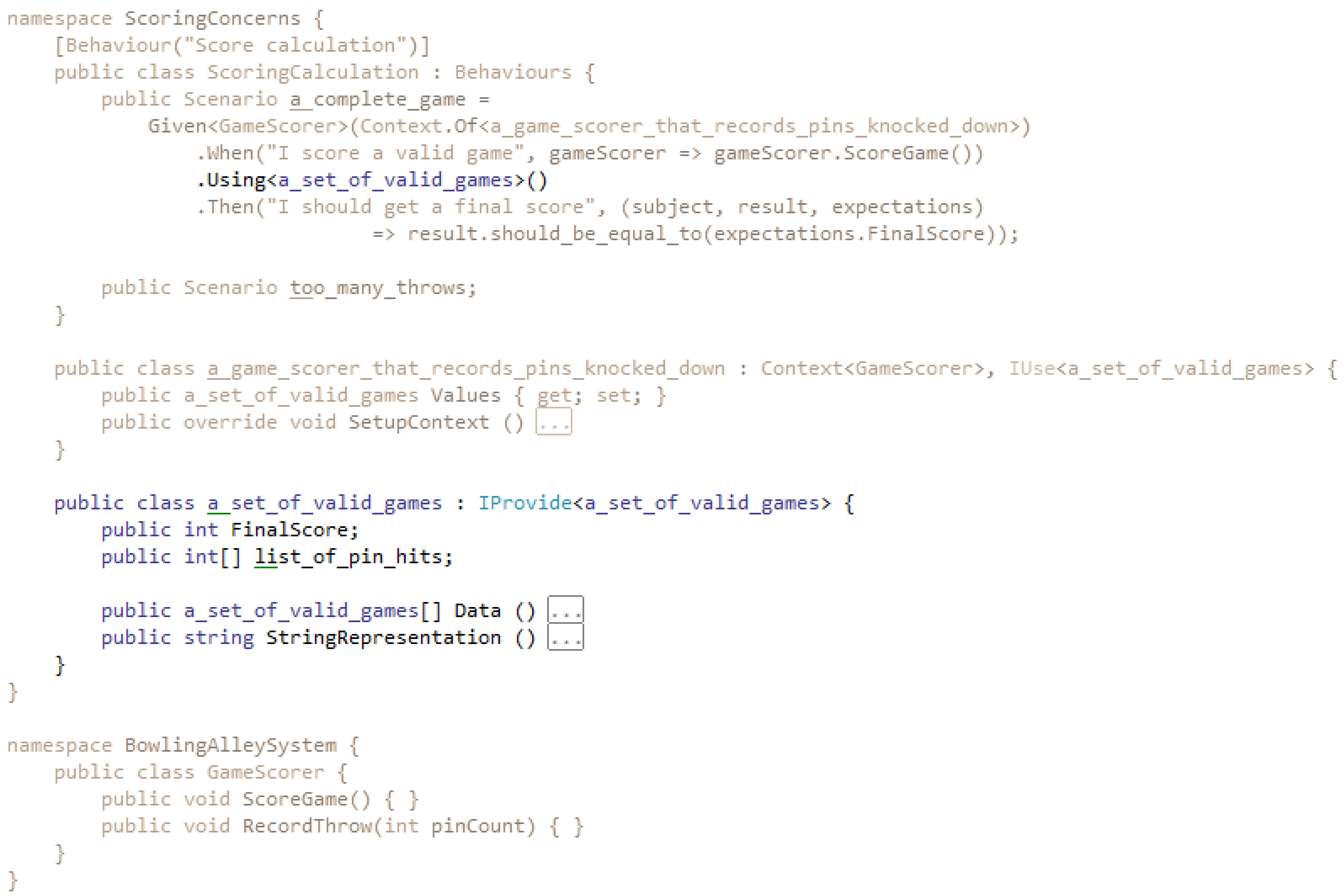
Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

The **Context** makes use of an **Expectation** provider type

Using FluentBDD



```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

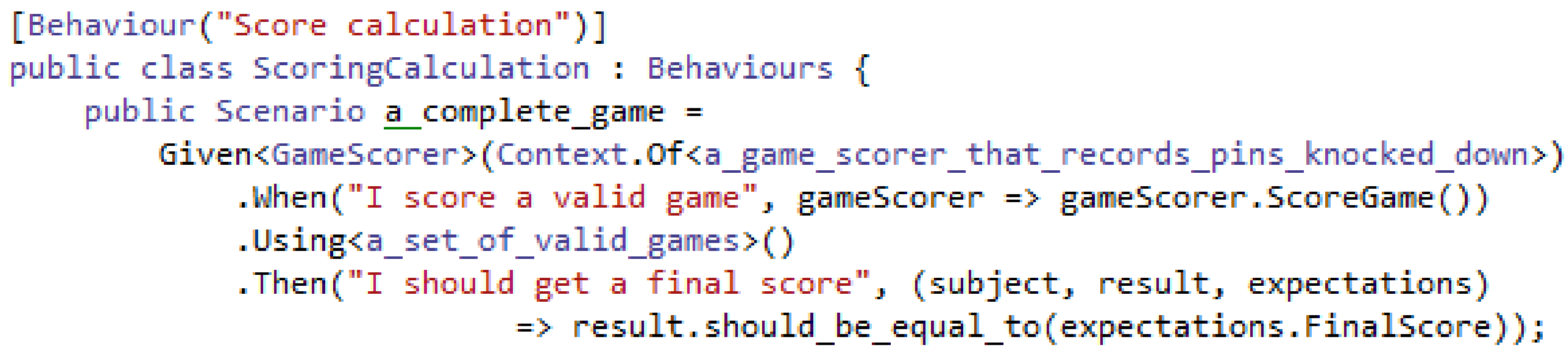
The **Scenario** ‘uses’ a specific **Expectation** provider

Using FluentBDD

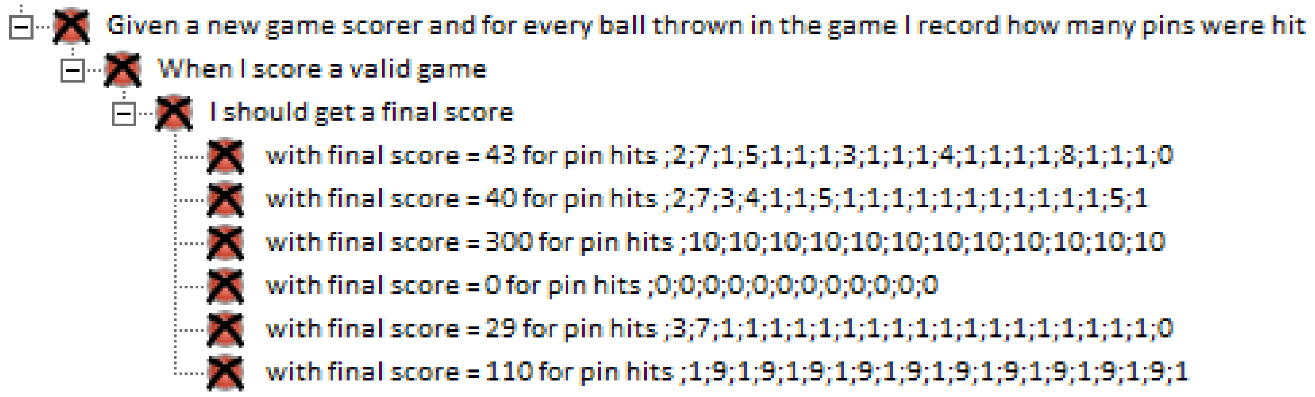
```
namespace ScoringConcerns {
    [Behaviour("Score calculation")]
    public class ScoringCalculation : Behaviours {
        public Scenario a_complete_game =
            Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
                .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
                .Using<a_set_of_valid_games>()
                .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));

        public Scenario too_many_throws;
    }

    public class a_game_scorer_that_records_pins_knocked_down : Context<GameScorer>, IUse<a_set_of_valid_games> {
        public a_set_of_valid_games Values { get; set; }
        public override void SetupContext () ...
    }

    public class a_set_of_valid_games : IProvide<a_set_of_valid_games> {
        public int FinalScore;
        public int[] list_of_pin_hits;

        public a_set_of_valid_games[] Data () ...
        public string StringRepresentation () ...
    }
}

namespace BowlingAlleySystem {
    public class GameScorer {
        public void ScoreGame() { }
        public void RecordThrow(int pinCount) { }
    }
}
```

That’s all we need to run our feature as a set of unit tests. Let’s take a look.

Using FluentBDD



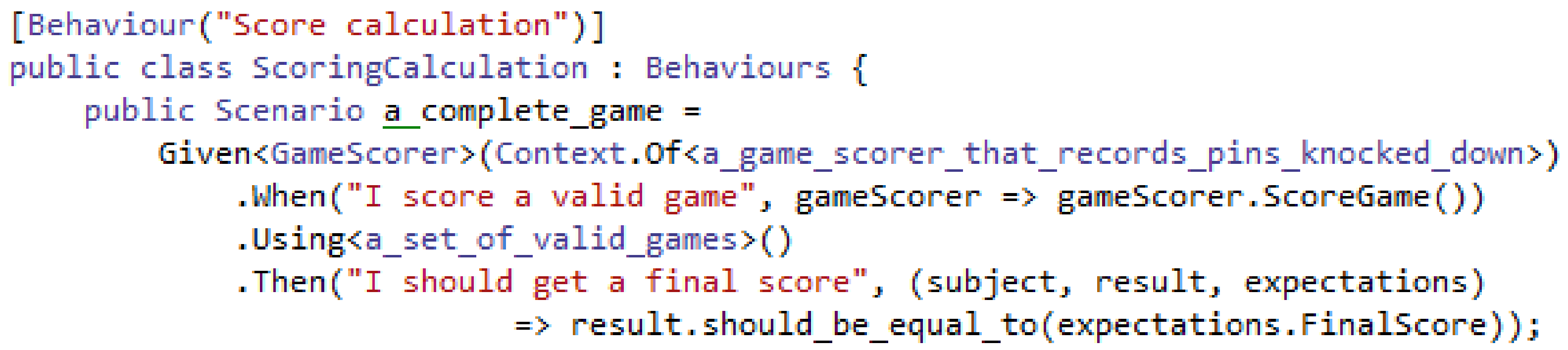
```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                               => result.should_be_equal_to(expectations.FinalScore));
```



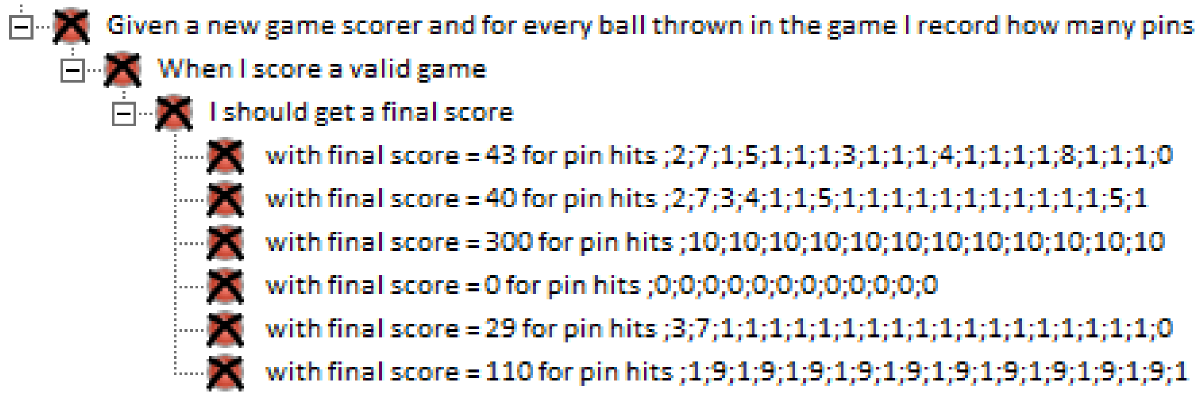
```
Given a new game scorer and for every ball thrown in the game I record how many pins were hit
  When I score a valid game
    I should get a final score
      with final score = 43 for pin hits ;2;7;1;5;1;1;1;3;1;1;1;4;1;1;1;1;8;1;1;1;0
      with final score = 40 for pin hits ;2;7;3;4;1;1;5;1;1;1;1;1;1;1;1;1;1;1;5;1
      with final score = 300 for pin hits ;10;10;10;10;10;10;10;10;10;10;10;10
      with final score = 0 for pin hits ;0;0;0;0;0;0;0;0;0;0;0;0
      with final score = 29 for pin hits ;3;7;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0
      with final score = 110 for pin hits ;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1
```

All red, as we should expect – we haven't implemented `GameScorer` yet

Using FluentBDD



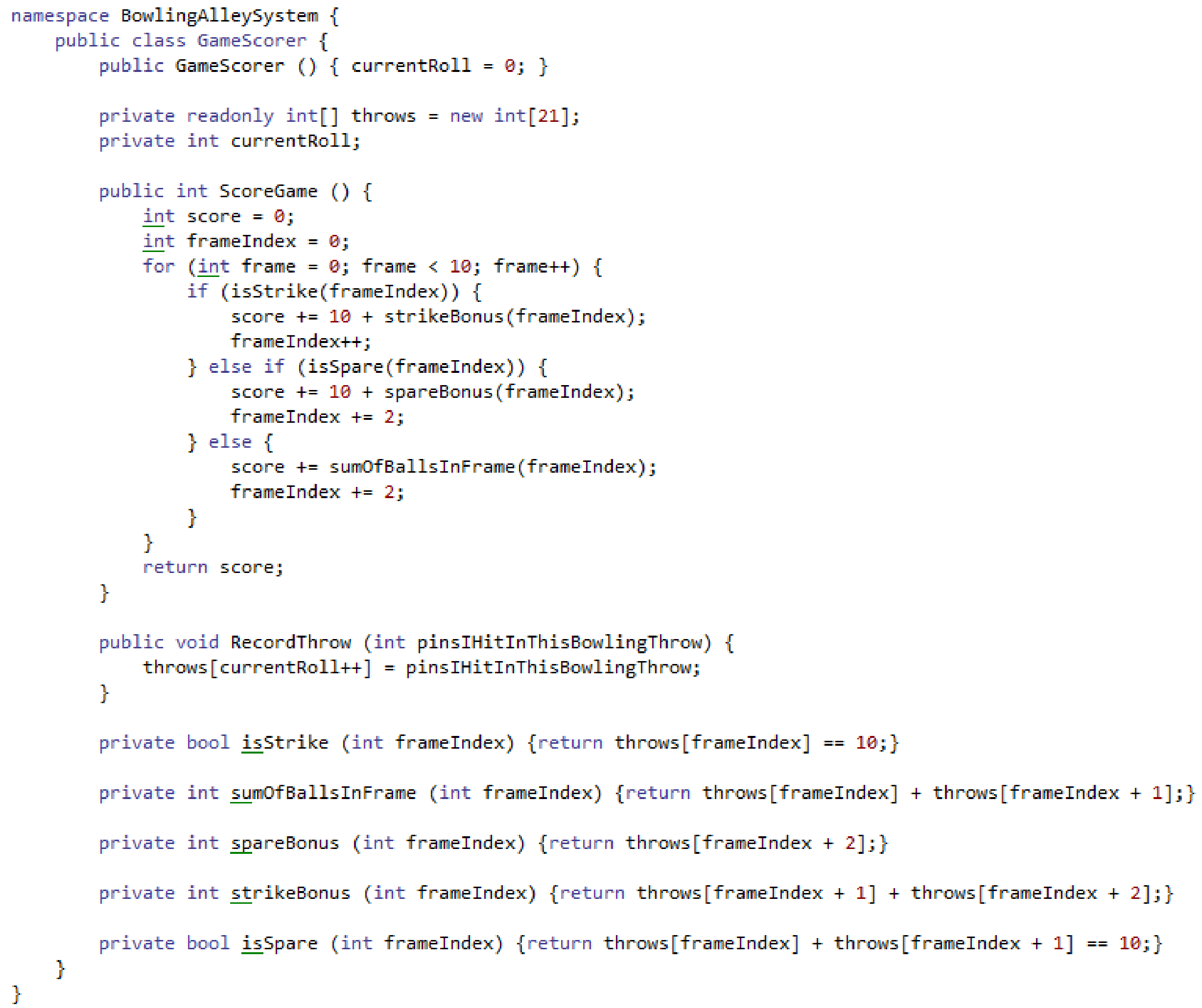
```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                           => result.should_be_equal_to(expectations.FinalScore));
```



```
Given a new game scorer and for every ball thrown in the game I record how many pins were hit
  When I score a valid game
    I should get a final score
      with final score = 43 for pin hits ;2;7;1;5;1;1;1;3;1;1;1;4;1;1;1;1;8;1;1;1;0
      with final score = 40 for pin hits ;2;7;3;4;1;1;5;1;1;1;1;1;1;1;1;1;1;1;5;1
      with final score = 300 for pin hits ;10;10;10;10;10;10;10;10;10;10;10;10
      with final score = 0 for pin hits ;0;0;0;0;0;0;0;0;0;0;0;0
      with final score = 29 for pin hits ;3;7;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0
      with final score = 110 for pin hits ;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1
```

Notice the progression in the unit tests follows that of the scenario,
and that our one scenario results in six unit tests (one per ‘then’ per expectation)

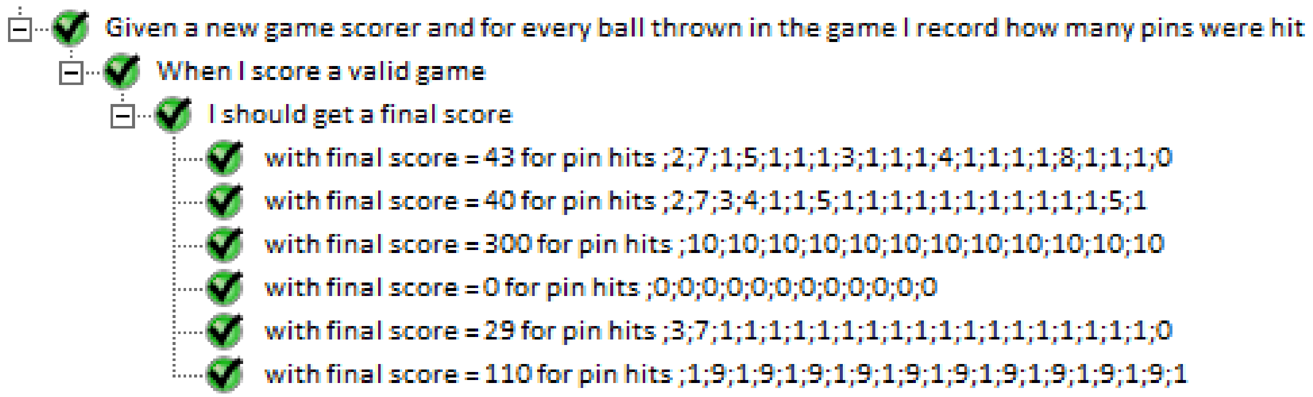
Using FluentBDD



```
namespace BowlingAlleySystem {
    public class GameScorer {
        public GameScorer () { currentRoll = 0; }

        private readonly int[] throws = new int[21];
        private int currentRoll;

        public int ScoreGame () {
            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 += sumOfBallsInFrame(frameIndex);
                    frameIndex += 2;
                }
            }
            return score;
        }

        public void RecordThrow (int pinsIHitInThisBowlingThrow) {
            throws[currentRoll++] = pinsIHitInThisBowlingThrow;
        }

        private bool isStrike (int frameIndex) {return throws[frameIndex] == 10;}

        private int sumOfBallsInFrame (int frameIndex) {return throws[frameIndex] + throws[frameIndex + 1];}

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

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

        private bool isSpare (int frameIndex) {return throws[frameIndex] + throws[frameIndex + 1] == 10;}
    }
}
```

Now if we implement the game scorer…

Using FluentBDD



```
Given a new game scorer and for every ball thrown in the game I record how many pins were hit
  When I score a valid game
    I should get a final score
      with final score = 43 for pin hits ;2;7;1;5;1;1;1;3;1;1;1;4;1;1;1;1;8;1;1;0
      with final score = 40 for pin hits ;2;7;3;4;1;1;5;1;1;1;1;1;1;1;1;1;1;5;1
      with final score = 300 for pin hits ;10;10;10;10;10;10;10;10;10;10;10;10
      with final score = 0 for pin hits ;0;0;0;0;0;0;0;0;0;0;0;0
      with final score = 29 for pin hits ;3;7;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0
      with final score = 110 for pin hits ;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1;9;1
```

…we are rewarded with a sea of green.

What we want to achieve

1. Write code that is close to plain English
    - As little code over-head as possible
    - Fluent interface
    - Good conventions for common cases
2. Type as little as possible
    - Reusable contexts
    - Reusable expectations
    - More than one test per scenario
    - Good interaction with ReSharper

What we want to achieve

1. Write code that is close to plain English
   - As little code over-head as possible
   - Fluent interface
   - Good conventions for common cases
     - Exceptions, Reflection, Creation
2. Type as little as possible
   - Reusable contexts
   - Reusable expectations
   - More than one test per scenario
   - Good interaction with ReSharper

Using FluentBDD: Exceptions

If we assumed that we'd never get an invalid call to GameScorer, we'd probably be done now. But reality is rarely so kind.

Let's build in some range checking. This is part of our Score calculation feature, and will require a new scenario:

Scenario: too many throws

With a game scorer that records pins knocked down

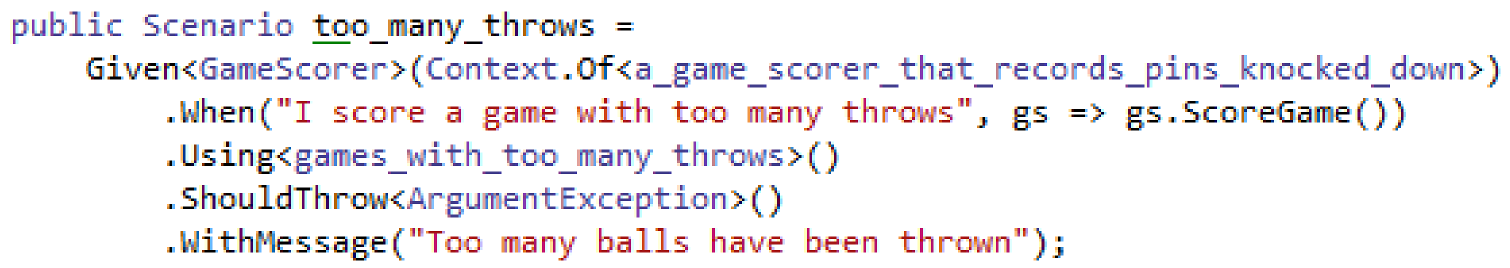
When I ask for the score

Using a game played with too many throws

I should get an exception telling me I've thrown too many balls

Using FluentBDD: Exceptions

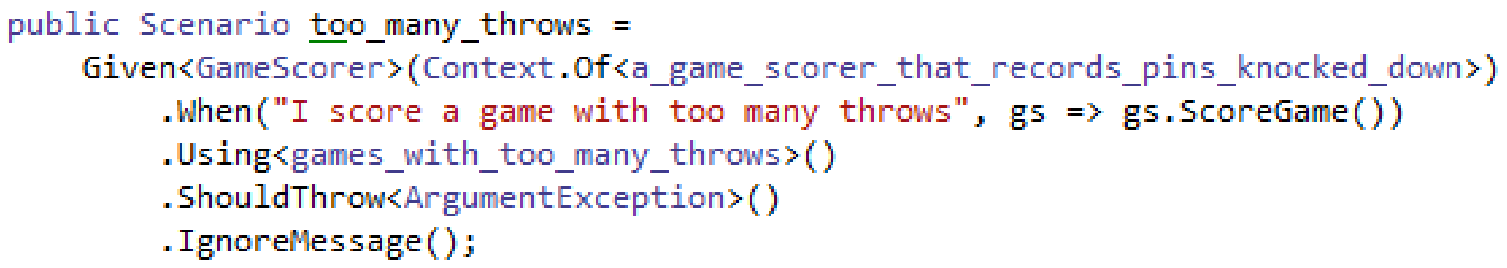
In code we write it like this:



```
public Scenario too_many_throws =
    Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
        .When("I score a game with too many throws", gs => gs.ScoreGame())
        .Using<games_with_too_many_throws>()
        .ShouldThrow<ArgumentException>()
        .WithMessage("Too many balls have been thrown");
```

Note, we have no “Then”, but instead we state the scenario should throw a specified type of exception, with an expected message.

Using FluentBDD: Exceptions



```
public Scenario too_many_throws =
    Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
        .When("I score a game with too many throws", gs => gs.ScoreGame())
        .Using<games_with_too_many_throws>()
        .ShouldThrow<ArgumentException>()
        .IgnoreMessage();
```

If we don’t care about the message returned with the exception, we must state this explicitly.

What we want to achieve

1. Write code that is close to plain English
    - As little code over-head as possible
    - Fluent interface
    - Good conventions for common cases
        - Exceptions, Reflection, Creation
2. Type as little as possible
    - Reusable contexts
    - Reusable expectations
    - More than one test per scenario
    - Good interaction with ReSharper

Using FluentBDD: Reflection

When dealing with various parts of the .Net ecosystem, class and field attributes can be critically important (e.g. WCF, messaging systems, Automapper)

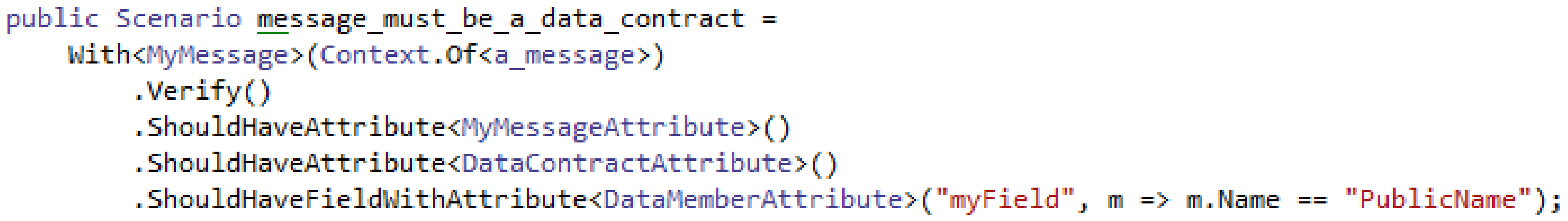
Scenario: message must be a data contract

With a message object

- It should have a “MyMessage” attribute
- It should have a “DataContract” attribute
- It should have a field “myField” with a “DataMember” attribute and it should be called “PublicName”

Using FluentBDD: Exceptions

In code we write it like this:



```
public Scenario message_must_be_a_data_contract =
    With<MyMessage>(Context.Of<a_message>)
        .Verify()
        .ShouldHaveAttribute<MyMessageAttribute>()
        .ShouldHaveAttribute<DataContractAttribute>()
        .ShouldHaveFieldWithAttribute<DataMemberAttribute>("myField", m => m.Name == "PublicName");
```

Note, we use “Verify” rather than “When” – this means we don’t need to give an action to perform.

What we want to achieve

1. Write code that is close to plain English
    - As little code over-head as possible
    - Fluent interface
    - Good conventions for common cases
        - Exceptions, Reflection, Creation
2. Type as little as possible
    - Reusable contexts
    - Reusable expectations
    - More than one test per scenario
    - Good interaction with ReSharper

Using FluentBDD: Creation

A lot of guard exceptions are placed around the creation of classes. There are a few other cases where we won't have a pre-made context or subject

Feature: Calculator creation

Scenario: creating a calculator

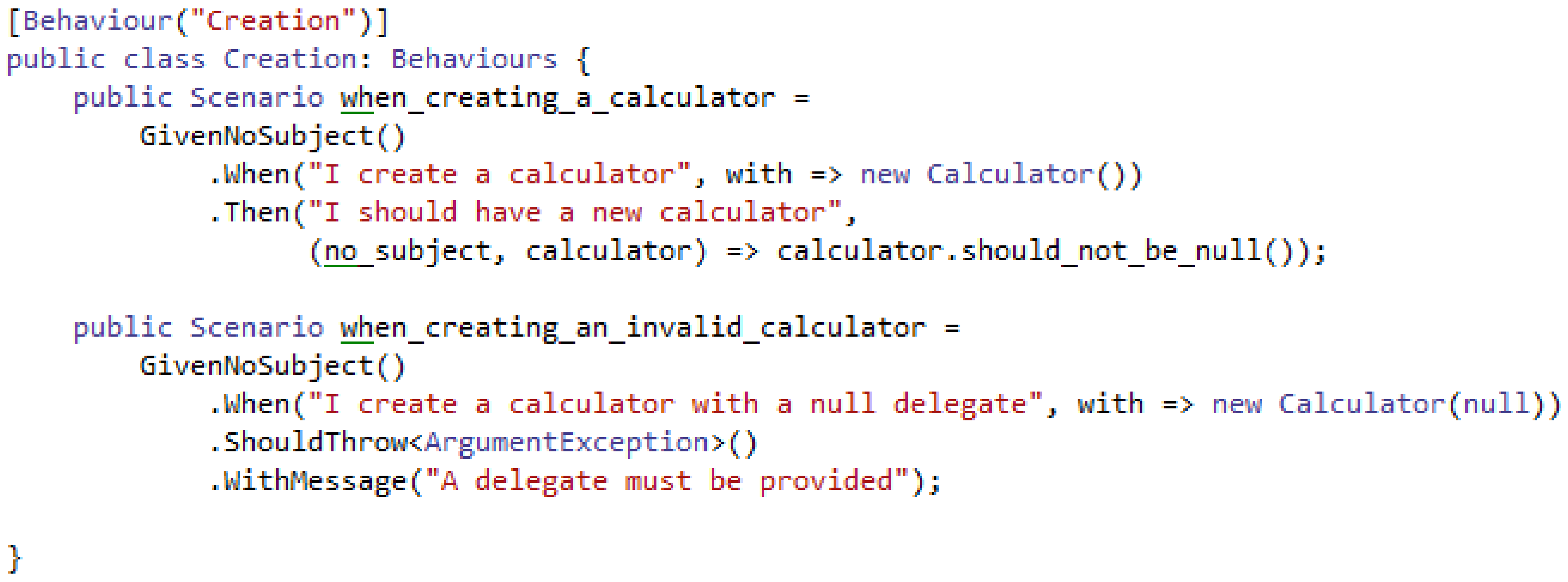
    When I create a calculator
    I should get an non-null object

Scenario: creating a calculator with invalid parameters

    When I create a calculator with a null delegate
    I should get an argument exception
    With message "A delegate must be provided"

Using FluentBDD: Exceptions

In code we write it like this:



```
[Behaviour("Creation")]
public class Creation: Behaviours {
    public Scenario when_creating_a_calculator =
        GivenNoSubject()
            .When("I create a calculator", with => new Calculator())
            .Then("I should have a new calculator",
                  (no_subject, calculator) => calculator.should_not_be_null());

    public Scenario when_creating_an_invalid_calculator =
        GivenNoSubject()
            .When("I create a calculator with a null delegate", with => new Calculator(null))
            .ShouldThrow<ArgumentException>()
            .WithMessage("A delegate must be provided");

}
```

What we want to achieve

1. Write code that is close to plain English
   - As little code over-head as possible
   - Fluent interface
   - Good conventions for common cases
     - Exceptions, Reflection, Creation
2. Type as little as possible
   - Reusable contexts
   - Reusable expectations
   - More than one test per scenario
   - Good interaction with ReSharper

Fluent BDD

Advanced topics: Templating

Advanced FluentBDD: Templating

When we write a specification template, we are trying to achieve one of two goals:

1. Supply different sets of Expectations to the same Context for multiple Scenarios

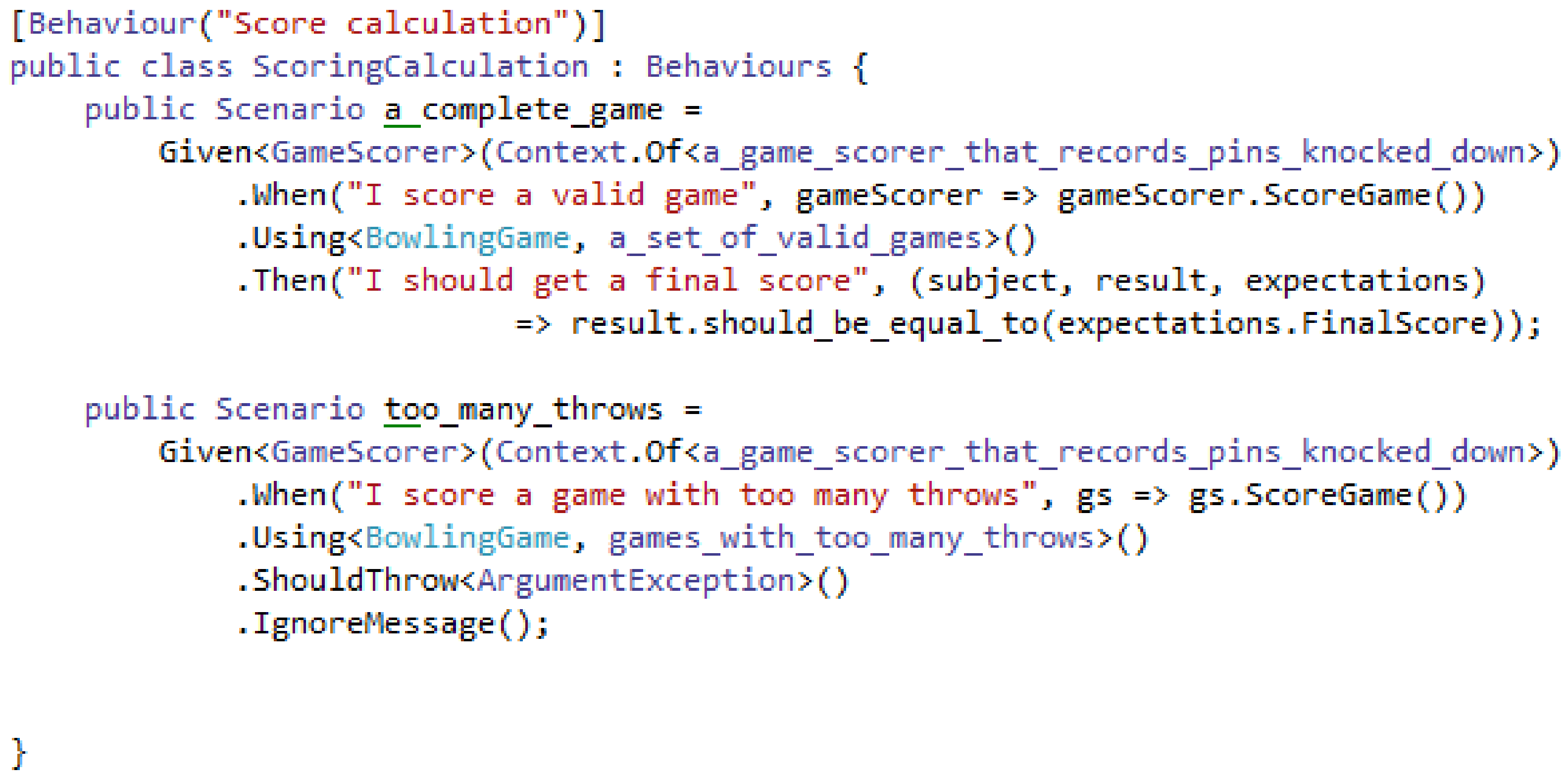
   We do this when we expect different outcomes from a scenario based solely on differing input.

2. Expecting different subjects to have the same behaviour

   For example, different implementations of an interface or two features which have scenarios in common

The for the first case, we head back to the bowling alley…

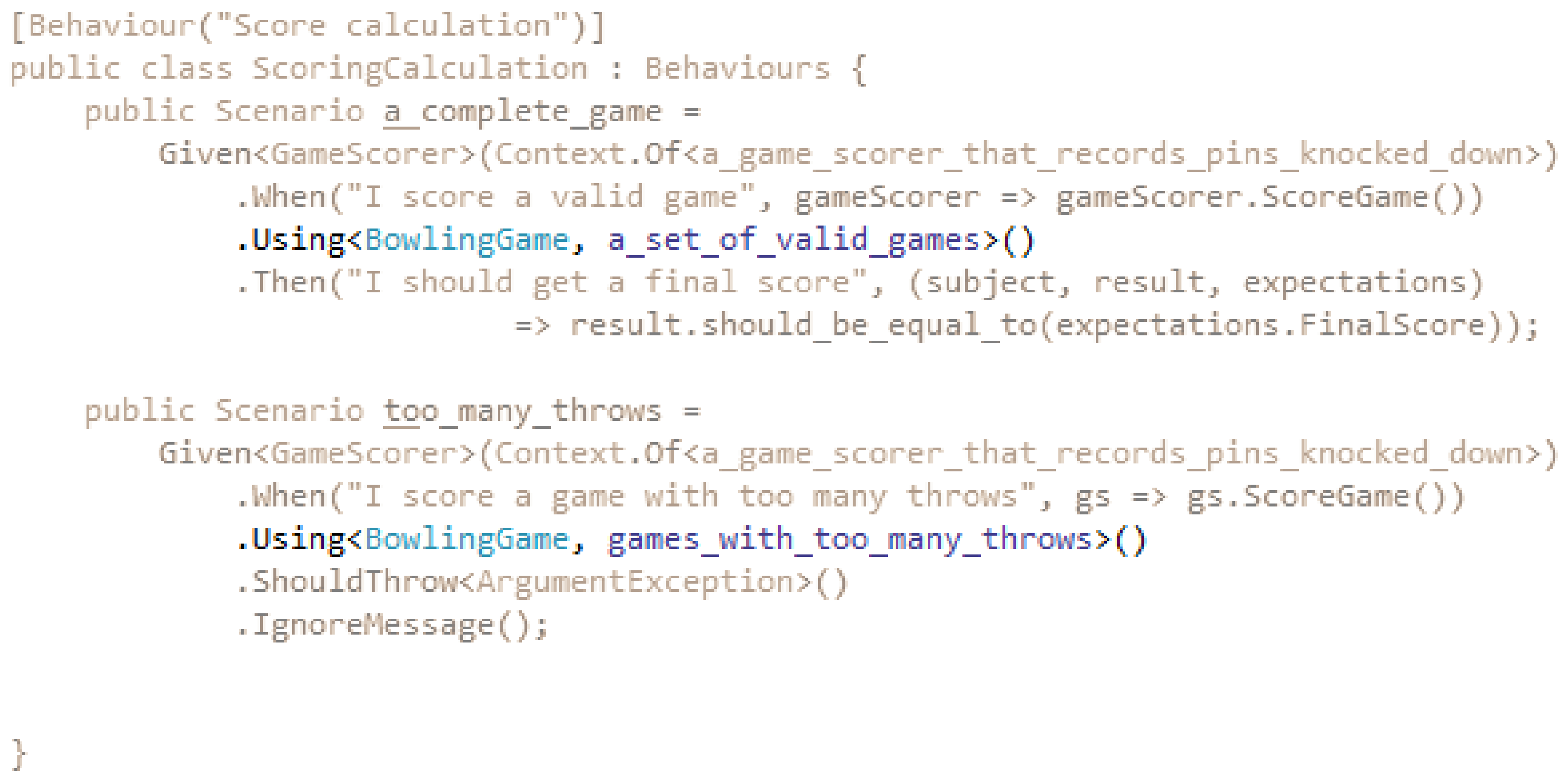
Advanced FluentBDD: Templating



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<BowlingGame, a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                           => result.should_be_equal_to(expectations.FinalScore));

    public Scenario too_many_throws =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a game with too many throws", gs => gs.ScoreGame())
            .Using<BowlingGame, games_with_too_many_throws>()
            .ShouldThrow<ArgumentException>()
            .IgnoreMessage();


}
```

Here we have our scoring feature, with two scenarios.

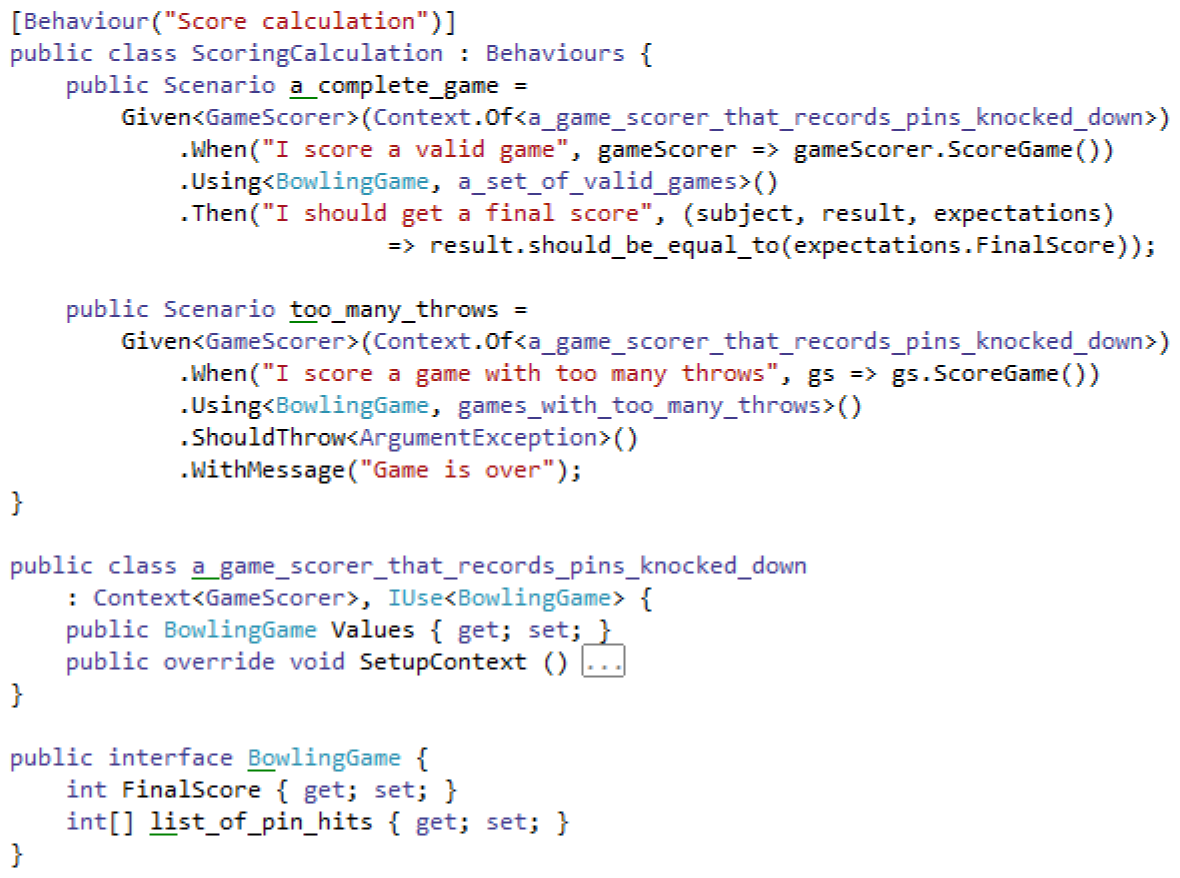
Advanced FluentBDD: Templating



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<BowlingGame, a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                        => result.should_be_equal_to(expectations.FinalScore));

    public Scenario too_many_throws =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a game with too many throws", gs => gs.ScoreGame())
            .Using<BowlingGame, games_with_too_many_throws>()
            .ShouldThrow<ArgumentException>()
            .IgnoreMessage();


}
```

Note the “Using” clause now has both an interface name and an implementation name.

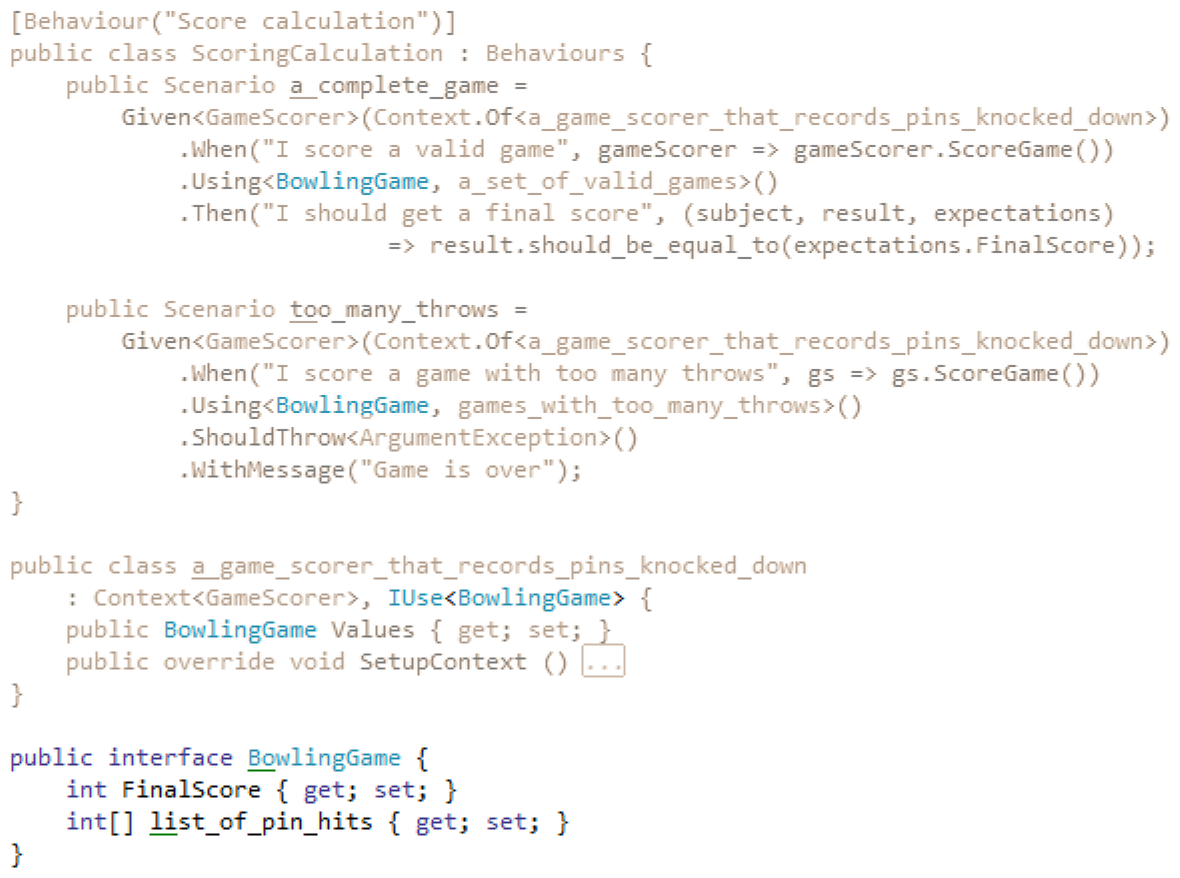
Advanced FluentBDD: Templating



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<BowlingGame, a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                          => result.should_be_equal_to(expectations.FinalScore));

    public Scenario too_many_throws =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a game with too many throws", gs => gs.ScoreGame())
            .Using<BowlingGame, games_with_too_many_throws>()
            .ShouldThrow<ArgumentException>()
            .WithMessage("Game is over");
}

public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<BowlingGame> {
    public BowlingGame Values { get; set; }
    public override void SetupContext () ...
}

public interface BowlingGame {
    int FinalScore { get; set; }
    int[] list_of_pin_hits { get; set; }
}
```

The context has a normal subject, and implements the IUse interface.

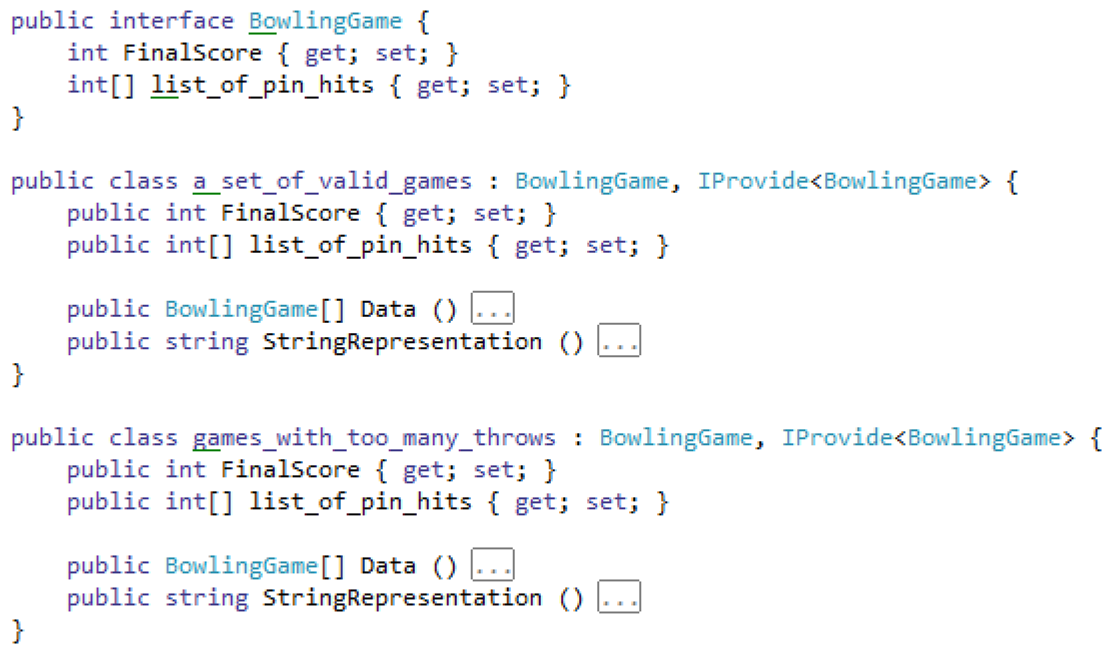
Advanced FluentBDD: Templating



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<BowlingGame, a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                           => result.should_be_equal_to(expectations.FinalScore));

    public Scenario too_many_throws =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a game with too many throws", gs => gs.ScoreGame())
            .Using<BowlingGame, games_with_too_many_throws>()
            .ShouldThrow<ArgumentException>()
            .WithMessage("Game is over");
}

public class a_game_scorer_that_records_pins_knocked_down
    : Context<GameScorer>, IUse<BowlingGame> {
    public BowlingGame Values { get; set; }
    public override void SetupContext () ...
}

public interface BowlingGame {
    int FinalScore { get; set; }
    int[] list_of_pin_hits { get; set; }
}
```

The BowlingGame interface provides the expectations used in the context and assertions, with no references to concretes outside of “Using”

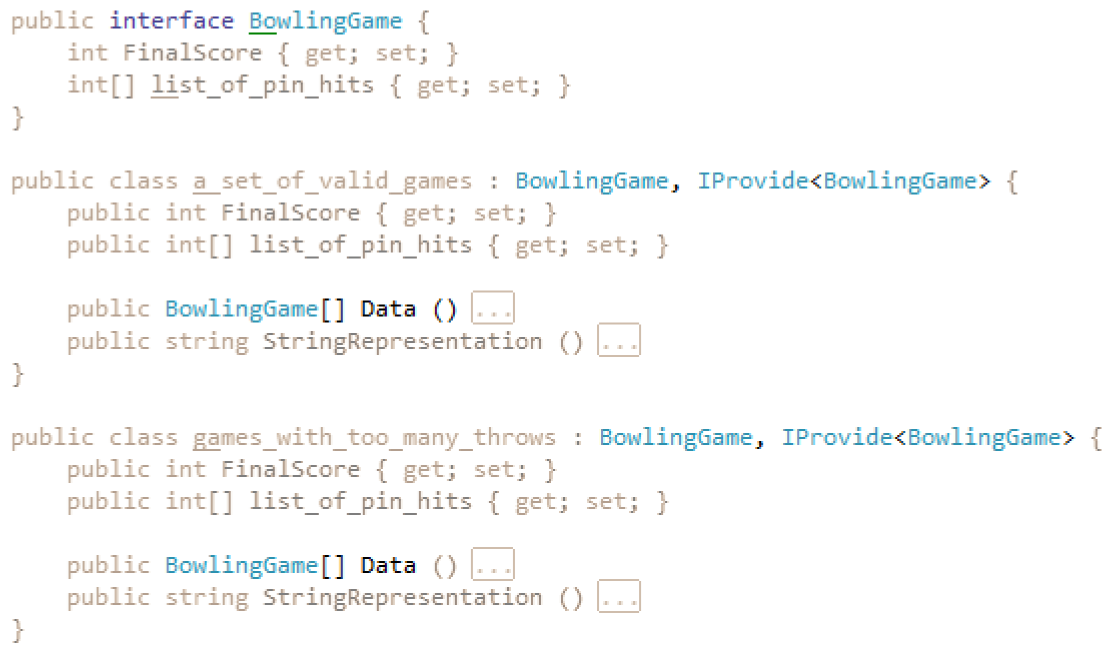
Advanced FluentBDD: Templating



```
public interface BowlingGame {
    int FinalScore { get; set; }
    int[] list_of_pin_hits { get; set; }
}

public class a_set_of_valid_games : BowlingGame, IProvide<BowlingGame> {
    public int FinalScore { get; set; }
    public int[] list_of_pin_hits { get; set; }

    public BowlingGame[] Data () ...
    public string StringRepresentation () ...
}

public class games_with_too_many_throws : BowlingGame, IProvide<BowlingGame> {
    public int FinalScore { get; set; }
    public int[] list_of_pin_hits { get; set; }

    public BowlingGame[] Data () ...
    public string StringRepresentation () ...
}
```

The concrete expectations implement both the `BowlingGame` interface and the `IProvide` interface.

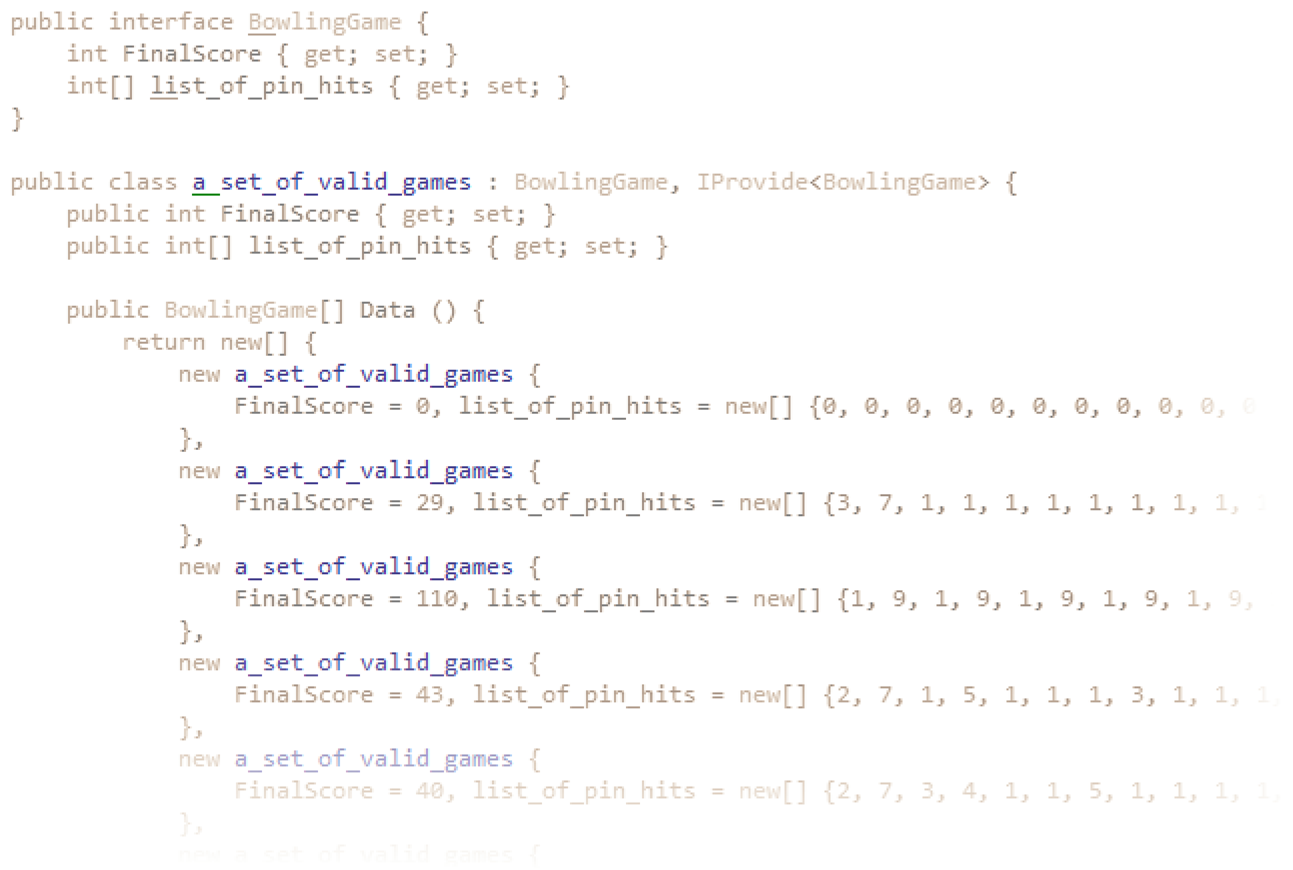
Advanced FluentBDD: Templating



```
public interface BowlingGame {
    int FinalScore { get; set; }
    int[] list_of_pin_hits { get; set; }
}

public class a_set_of_valid_games : BowlingGame, IProvide<BowlingGame> {
    public int FinalScore { get; set; }
    public int[] list_of_pin_hits { get; set; }

    public BowlingGame[] Data () ...
    public string StringRepresentation () ...
}

public class games_with_too_many_throws : BowlingGame, IProvide<BowlingGame> {
    public int FinalScore { get; set; }
    public int[] list_of_pin_hits { get; set; }

    public BowlingGame[] Data () ...
    public string StringRepresentation () ...
}
```

Note that it's the `BowlingGame` interface that is being provided…

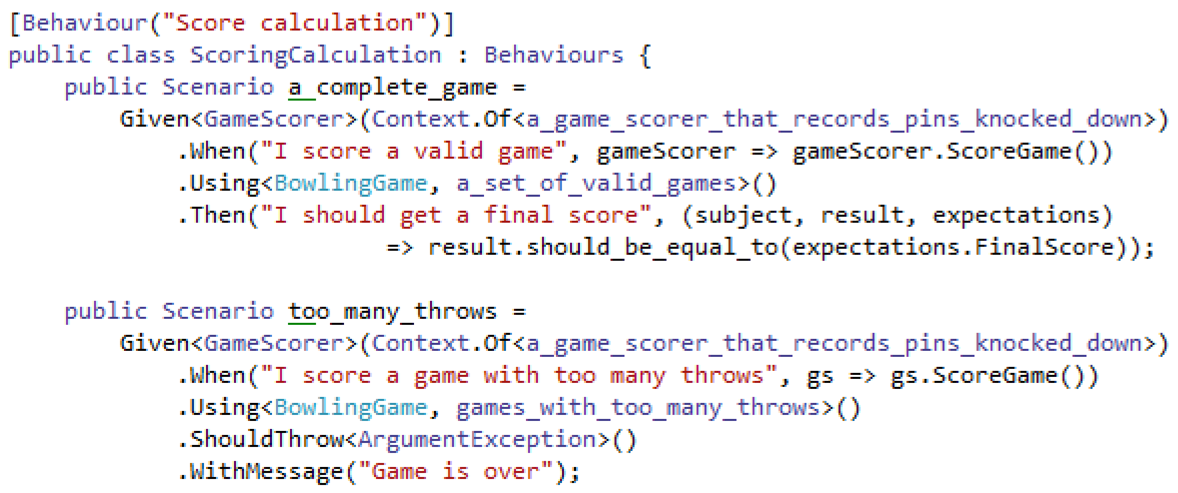
Advanced FluentBDD: Templating



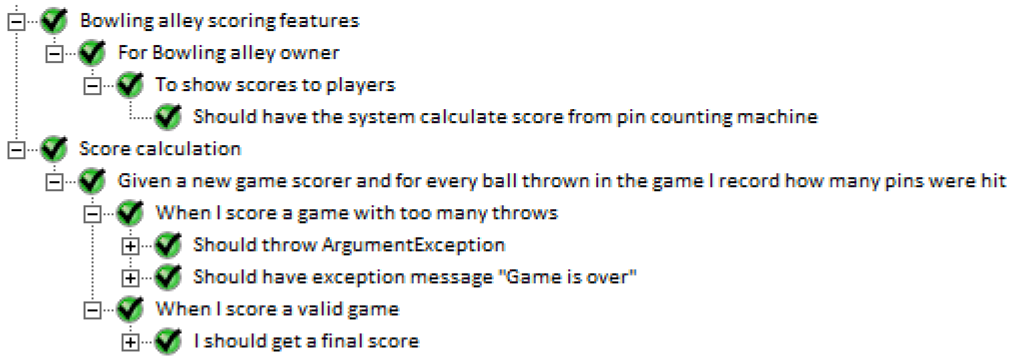
```
public interface BowlingGame {
    int FinalScore { get; set; }
    int[] list_of_pin_hits { get; set; }
}

public class a_set_of_valid_games : BowlingGame, IProvide<BowlingGame> {
    public int FinalScore { get; set; }
    public int[] list_of_pin_hits { get; set; }

    public BowlingGame[] Data () {
        return new[] {
            new a_set_of_valid_games {
                FinalScore = 0, list_of_pin_hits = new[] {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
            },
            new a_set_of_valid_games {
                FinalScore = 29, list_of_pin_hits = new[] {3, 7, 1, 1, 1, 1, 1, 1, 1, 1, 1
            },
            new a_set_of_valid_games {
                FinalScore = 110, list_of_pin_hits = new[] {1, 9, 1, 9, 1, 9, 1, 9, 1, 9,
            },
            new a_set_of_valid_games {
                FinalScore = 43, list_of_pin_hits = new[] {2, 7, 1, 5, 1, 1, 1, 3, 1, 1, 1,
            },
            new a_set_of_valid_games {
                FinalScore = 40, list_of_pin_hits = new[] {2, 7, 3, 4, 1, 1, 5, 1, 1, 1, 1,
            },
            new a_set_of_valid_games {
```

…but instances of the concrete are being returned.

Advanced FluentBDD: Templating



```
[Behaviour("Score calculation")]
public class ScoringCalculation : Behaviours {
    public Scenario a_complete_game =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a valid game", gameScorer => gameScorer.ScoreGame())
            .Using<BowlingGame, a_set_of_valid_games>()
            .Then("I should get a final score", (subject, result, expectations)
                           => result.should_be_equal_to(expectations.FinalScore));

    public Scenario too_many_throws =
        Given<GameScorer>(Context.Of<a_game_scorer_that_records_pins_knocked_down>)
            .When("I score a game with too many throws", gs => gs.ScoreGame())
            .Using<BowlingGame, games_with_too_many_throws>()
            .ShouldThrow<ArgumentException>()
            .WithMessage("Game is over");
```



```
⊟ ✓ Bowling alley scoring features
   ⊟ ✓ For Bowling alley owner
      ⊟ ✓ To show scores to players
           ✓ Should have the system calculate score from pin counting machine
⊟ ✓ Score calculation
   ⊟ ✓ Given a new game scorer and for every ball thrown in the game I record how many pins were hit
      ⊟ ✓ When I score a game with too many throws
         ⊞ ✓ Should throw ArgumentException
         ⊞ ✓ Should have exception message "Game is over"
      ⊟ ✓ When I score a valid game
         ⊞ ✓ I should get a final score
```

Notice that the test results have only one ‘Given’, but two ‘When’s

What we want to achieve

1. Write code that is close to plain English
    - As little code over-head as possible
    - Fluent interface
    - Good conventions for common cases
        - Exceptions, Reflection, Creation
2. Type as little as possible
    - Reusable contexts
    - Reusable expectations
    - More than one test per scenario
    - Good interaction with ReSharper

Fluent BDD

any questions?

FluentBDD is a work in progress, and will most likely expand as it is used more extensively.