Engineer Notebook: An Extreme Programming Episode

by Robert C. Martin and Robert S. Koss

Figure 1

Figure 2

Note

About the Authors

Design and programming are human activities; forget that and all is lost.

Bjarne Stroustrup, 1991

In order to demonstrate XP (eXtreme Programming) practices, Bob Koss (RSK) and Bob Martin (RCM) will pair program a simple application while you watch like a fly on the wall. We will use test first design and a lot of refactoring to create our application. What follows is a faithful re-enactment of a programming episode that the two Bob's actually did.

RCM: "Will you help me write a little application that calculates bowling scores?"

RSK: (Reflects to himself: The XP practice of pair programming says that I can't say no, when asked to help. I suppose that's especially true when it is your boss who is asking.) "Sure Bob, I'd be glad to help."

RCM: "OK, Great. What I'd like to do is write an application that keeps track of a bowling league. It needs to record all the games, determine the ranks of the teams, determine the winners and losers of each weekly match, and accurately score each game."

RSK: "Cool. I used to be a pretty good bowler. This will be fun. You rattled off several user stories, which one would you like to start with."

RCM: "Let's begin with scoring a single game."

RSK: "Okay. What does that mean? What are the inputs and outputs for this story?"

RCM: "It seems to me that the inputs are simply a sequence of throws. A throw is just an integer that tells how many pins were knocked down by the ball. The output is the data on a standard bowling score card, a set of frames populated with the pins knocked down by each throw, and marks denoting spares and strikes. The most important number in each frame is the current game score."

RSK: "Let me sketch out a little picture of this score card to give us a visual reminder of the requirements." (See Figure 1.)

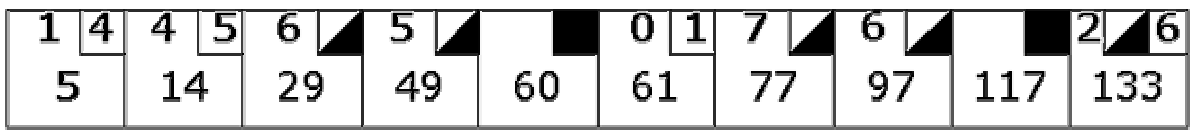

Figure 1

RCM: "That guy is pretty erratic."

RSK: "Or drunk, but it will serve as a decent acceptance test."

RCM: "We'll need others, but let's deal with that later. How should we start? Shall we come up with a design for the system?"

RSK: "Well, don't hate me, but I wouldn't mind a UML diagram showing the problem domain concepts that we might see from the score card. That will give us some candidate objects that we can explore further in code."

RCM: (Putting on his powerful object designer hat) "OK, clearly a game object consists of a sequence of ten frames. Each frame object contains one, two, or three throws."

RSK: "Great minds. That was exactly what I was thinking. Let me quickly draw that, but if you tell Kent, I'll deny it." (See Figure 2.)

Figure 2

Kent: *"I'm always watching."*

RSK: "Well, pick a class ... any class. Shall we start at the end of the dependency chain and work backwards? That will make testing easier."

RCM: "Sure, why not. Let's create a test case for the **Throw** class."

RSK: (Starts typing)

```
//TestThrow.java---------------------------------
import junit.framework.*;

public class TestThrow extends TestCase
{
  public TestThrow(String name)
  {
    super(name);
  }
// public void test????
}
```

RSK: "Do you have a clue what the behavior of a **Throw** object should be?"

RCM: "It holds the number of pins knocked down by the player."

RSK: "Okay, you just said in not so many words that it doesn't really do anything. Maybe we should come back to it and focus on an object that actually has behavior, instead of one that's just a data store."

RCM: "Hmm. You mean the **Throw** class might not really exist?"

RSK: (Starts to sweat. This is my boss I'm working with.) "Well, if it doesn't have any behavior, how important can it be? I don't know if it exists or not yet. I'd just feel more productive if we were working on an object that had more than setters and getters for methods. But if you want to drive..." (slides the keyboard to RCM).

RCM: "Well, let's move up the dependency chain to **Frame** and see if there are any test cases we can write that will force us to finish **Throw**." (Pushes the keyboard back to RSK.)

RSK: (Wondering if RCM is leading me down a blind alley to educate me or if he is really agreeing with me) "Okay, new file, new test case."

```
//TestFrame.java-------------------------------------
import junit.framework.*;

public class TestFrame extends TestCase
{
  public TestFrame( String name )
  {
    super( name );
  }

  //public void test???
}
```

RCM: "OK, that's the second time we've typed that. Now, can you think of any interesting test cases for **Frame**?"

RSK: "A frame might provide its score, the number of pins on each throw, whether there was a strike or a spare..."

RCM: "Too much talk, not enough code. Type!"

RSK: (types)

```
//TestFrame.java---------------------------------
import junit.framework.*;

public class TestFrame extends TestCase
{
  public TestFrame( String name )
  {
    super( name );
  }

  public void testScoreNoThrows()
  {
    Frame f = new Frame();
    assertEquals( 0, f.getScore() );
  }
}

//Frame.java---------------------------------------
public class Frame
{
```

```
  public int getScore()
  {
    return 0;
  }
}
```

RCM: "OK, the test case passes. But **Score** is a really stupid function. It will fail if we add a throw to the frame. So let's write the test case that adds some throws and then checks the score."

```
//TestFrame.java--------------------------------

public void testAddOneThrow()
{
  Frame f = new Frame();
  f.add(5);
  assertEquals(5, f.getScore());
}
```

RCM: "That doesn't compile. There's no **add** method in **Frame**."

RSK: "I'll bet if you define the method it will compile."

RCM:

```
//Frame.java-----------------------------------------
public class Frame
{
  public int getScore()
  {
    return 0;
  }

  public void add(Throw t)
  {
  }
}
```

RCM: (Thinking out loud) "This doesn't compile because we haven't written the **Throw** class."

RSK: "Talk to me, Bob. The test is passing an integer, and the method expects a **Throw** object. You can't have it both ways. Before we go down the **Throw** path again, can you describe its behavior?"

RCM: "Wow! I didn't even notice that I had written **f.add(5)**. I should have written **f.add(new Throw(5))**, but that's ugly as hell. What I *really* want to write is **f.add(5)**."

RSK: "Ugly or not, let's leave aesthetics out of it for the time being. Can you describe any behavior of a **Throw** object — binary response, Bob?"

RCM: "101101011010100101. I don't know if there is any behavior in **Throw**; I'm beginning to think a **Throw** is just an **int**. However, we don't need to consider that yet, since we can write **Frame.add** to take an **int**."

RSK: "Then I think we should do that for no other reason than it's simple. When we feel pain, we can

do something more sophisticated."

RCM: "Agreed."

```
//Frame.java----------------------------------------
public class Frame
{
  public int getScore()
  {
    return 0;
  }

  public void add(int pins)
  {
  }
}
```

RCM: "OK, this compiles and fails the test. Now, let's make the test pass."

```
//Frame.java----------------------------------------
public class Frame
{
  public int getScore()
  {
    return itsScore;
  }

  public void add(int pins)
  {
    itsScore += pins;
  }
  private int itsScore = 0;
}
```

RCM: "This compiles and passes the tests. But it's clearly simplistic. What's the next test case?"

RSK: "Can we take a break first?"

RCM: "That's better. **Frame.add** is a fragile function. What if you call it with an 11?"

RSK: "It can throw an exception if that happens. But who is calling it? Is this going to be an application framework that thousands of people will use and we have to protect against such things, or is this going to be used by you and only you? If the latter, just don't call it with an 11" (chuckle).

RCM: "Good point, the tests in the rest of the system will catch an invalid argument. If we run into trouble, we can put the check in later. So, the add function doesn't currently handle strikes or spares. Let's write a test case that expresses that."

RSK: "Hmmmm... if we call **add(10)** to represent a strike, what should **getScore** return? I don't know how to write the assertion, so maybe we're asking the wrong question. Or we're asking the right question to the wrong object."

RCM: "When you call **add(10)**, or **add(3)** followed by **add(7)**, then calling **getScore** on the **Frame** is meaningless. The frame would have to look ahead at later frames to calculate its score. If those later frames don't exist, then it would have to return something ugly like -1. I don't want to return -1."

RSK: "Yeah, I hate the -1 idea too. You've introduced the idea of frames knowing about other frames. Who is holding these different frame objects?"

RCM: "The **Game** object."

RSK: "So **Game** depends on **Frame**, and **Frame** in turn depends on **Game**. I hate that."

RCM: "**Frame**s don't have to depend upon **Game**; they could be arranged in a linked list. **Each frame could hold pointers to its next and previous frames.** To get the score from a frame, the frame would look backwards to get the score of the previous frame and look forwards for any spare or strike balls it needs."

RSK: "Okay, I'm feeling kind of dumb because I can't visualize this. Show me some code, boss."

RCM: "Right. So, we need a test case first."

RSK: "For **Game** or another test for **Frame**?"

RCM: "I think we need one for **Game**, since it's **Game** that will build the frames and hook them up to each other."

RSK: "Do you want to stop what we're doing on **Frame** and do a mental long jump to **Game**, or do you just want to have a **MockGame** object that does just what we need to get **Frame** working?"

RCM: "No, let's stop working on **Frame** and start working on **Game**. The test cases in **Game** should prove that we need the linked list of **Frame**s."

RSK: "I'm not sure how they'll show the need for the list. I need code."

RCM:

```
//TestGame.java----------------------------------------
import junit.framework.*;

public class TestGame extends TestCase
{
  public TestGame(String name)
  {
    super(name);
  }

  public void testOneThrow()
  {
    Game g = new Game();
    g.add(5);
    assertEquals(5, g.score());
  }
}
```

RCM: "Does that look reasonable?"

RSK: "Sure, but I'm still looking for proof for this list of **Frame**s."

RCM: "Me too. Let's keep following these test cases and see where they lead."

```
//Game.java---------------------------------
public class Game
{
  public int score()
  {
    return 0;
  }

  public void add(int pins)
  {
  }
}
```

RCM: "OK, this compiles and fails the test. Now let's make it pass."

```
//Game.java---------------------------------
public class Game
{
  public int score()
  {
    return itsScore;
  }

  public void add(int pins)
  {
    itsScore += pins;
  }
  private int itsScore = 0;
}
```

RCM: "This passes. Good."

RSK: "I can't disagree with it. But I'm still looking for this great proof of the need for a linked list of frame objects. That's what led us to **Game** in the first place."

RCM: "Yeah, that's what I'm looking for too. I fully expect that once we start injecting spare and strike test cases, we'll have to build frames and tie them together in a linked list. **But I don't want to build that until the code forces us to.**"

RSK: "Good point. Let's keep going in small steps on **Game**. What about another test that tests two throws but with no spare?"

RCM: "OK, that should pass right now. Let's try it."

```
//TestGame.java-----------------------------------------

public void testTwoThrowsNoMark()
```

```
  {
    Game g = new Game();
    g.add(5);
    g.add(4);
    assertEquals(9, g.score());
}
```

RCM: "Yep, that one passes. Now let's try four balls, with no marks."

RSK: "Well, that will pass too. I didn't expect this. We can keep adding throws, and we don't ever even need a **Frame**. But we haven't done a spare or a strike yet. Maybe that's when we'll have to make one."

RCM: "That's what I'm counting on. However, consider this test case:"

```
//TestGame.java----------------------------------------
public void testFourThrowsNoMark()
  {
    Game g = new Game();
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
  }
```

RCM: "Does this look reasonable?"

RSK: "It sure does. I forgot that we have to be able to show the score in each frame. Ah, our sketch of the score card was serving as a coaster for my Diet Coke. Yeah, that's why I forgot."

RCM: (Sigh) "OK, first let's make this test case fail by adding the **scoreForFrame** method to **Game**."

```
//Game.java---------------------------------

public int scoreForFrame(int frame)
{
  return 0;
}
```

RCM: "Great, this compiles and fails. Now, how do we make it pass?"

RSK: "We can start making frame objects. But is that the simplest thing that will get the test to pass?"

RCM: "No, actually, we could just create an array of integers in **Game**. Each call to **add** would append a new integer onto the array. Each call to **scoreForFrame** will just work forward through the array and calculate the score."

```
//Game.java---------------------------------
public class Game
{
```

```
  public int score()
  {
    return itsScore;
  }

  public void add(int pins)
    {
      itsThrows[itsCurrentThrow++]=pins;
      itsScore += pins;
    }

  public int scoreForFrame(int frame)
    {
    int score = 0;
    for ( int ball = 0;
      frame > 0 && (ball < itsCurrentThrow);
      ball+=2, frame--)
    {
      score += itsThrows[ball] + itsThrows[ball+1];
    }
    return score;
  }
  private int itsScore = 0;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;
}
```

RCM: (Very satisfied with himself) "There, that works."

RSK: "Why the magic number 21?"

RCM: "That's the maximum possible number of throws in a game."

RSK: "Yuck. Let me guess, in your youth you were a Unix hacker and prided yourself on writing an entire application in one statement that nobody else could decipher.

"**scoreForFrame** needs to be refactored to be more communicative. But before we consider refactoring, let me ask another question: Is **Game** the best place for this method? In my mind, **Game** is violating Bertrand Meyer's SRP (Single Responsibility Principle) [1]. It is accepting throws *and* it knows how to score for each frame. What would you think about a **Scorer** object?"

RCM: (Makes a rude oscillating gesture with his hand) "I don't know where the functions live now; right now I'm interested in getting the scoring stuff to work. Once we've got that all in place, *then* we can debate the values of the SRP.

"However, I see your point about the Unix hacker stuff; let's try to simplify that loop."

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
```

```
  {
    score += itsThrows[ball++] + itsThrows[ball++];
  }

  return score;
}
```

RCM: "That's a little better, but there are side-effects in the **score+=** expression. They don't matter here because it doesn't matter which order the two **addend** expressions are evaluated in." (Or does it? It's possible that the two increments could be done before either array operations.)

RSK: "I suppose we could do an experiment to verify that there aren't any side-effects, but that function isn't going to work with spares and strikes. Should we keep trying to make it more readable or should we push further on its functionality?"

RCM: "The experiment would only have meaning on certain compilers. Other compilers might use different evaluation orders. Let's get rid of the order dependency and then push on with more test cases."

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    int firstThrow = itsThrows[ball++];
    int secondThrow = itsThrows[ball++];
    score += firstThrow + secondThrow;
  }

  return score;
}
```

RCM: "OK, next test case. Let's try a spare."

```
public void testSimpleSpare()
{
  Game g = new Game();
}
```

RCM: "I'm tired of writing this. Let's refactor the test and put the creation of the game in a **setUp** function."

```
//TestGame.java-----------------------------------------
import junit.framework.*;

public class TestGame extends TestCase
{
  public TestGame(String name)
  {
    super(name);
  }
```

```
  private Game g;

  public void setUp()
  {
    g = new Game();
  }

  public void testOneThrow()
  {
    g.add(5);
    assertEquals(5, g.score());
  }

  public void testTwoThrowsNoMark()
  {
    g.add(5);
    g.add(4);
    assertEquals(9, g.score());
  }

  public void testFourThrowsNoMark()
  {
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
  }

  public void testSimpleSpare()
  {
  }
}
```

RCM: "That's better, now let's write the spare test case."

RSK: "I'll drive."

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    int firstThrow = itsThrows[ball++];
    int secondThrow = itsThrows[ball++];

    int frameScore = firstThrow + secondThrow;
    // spare needs next frames first throw
    if ( frameScore == 10 )
      score += frameScore + itsThrows[ball++];
    else
```

```
      score += frameScore;
  }

  return score;
}
```

RCM: (Grabbing the keyboard) "OK, but I think the increment of ball in the **frameScore==10** case shouldn't be there. Here's a test case that proves my point."

```
public void testSimpleFrameAfterSpare()
{
  g.add(3);
  g.add(7);
  g.add(3);
  g.add(2);
  assertEquals(13, g.scoreForFrame(1));
  assertEquals(18, g.score());
}
```

RCM: "Ha! See, that fails. Now if we just take out that pesky extra increment..."

```
  if ( frameScore == 10 )
    score += frameScore + itsThrows[ball];
```

RCM: "Uh, it still fails.... Could it be that the **score** method is wrong? I'll test that by changing the test case to use **scoreForFrame(2)**."

```
public void testSimpleFrameAfterSpare()
{
  g.add(3);
  g.add(7);
  g.add(3);
  g.add(2);
  assertEquals(13, g.scoreForFrame(1));
  assertEquals(18, g.scoreForFrame(2));
}
```

RCM: "Hmmmm.... That passes. The **score** method must be messed up. Let's look at it."

```
public int score()
{
  return itsScore;
}

public void add(int pins)
{
  itsThrows[itsCurrentThrow++]=pins;
  itsScore += pins;
}
```

RCM: "Yeah, that's wrong. The **score** method is just returning the sum of the pins, not the proper score. What we need **score** to do is call **scoreForFrame** with the current frame."

RSK: "We don't know what the current frame is. Let's add that message to each of our current tests, one at a time, of course."

RCM: "Right."

```
//TestGame.java-----------------------------------------
  public void testOneThrow()
  {
    g.add(5);
    assertEquals(5, g.score());
    assertEquals(1, g.getCurrentFrame());
    }

//Game.java---------------------------------
  public int getCurrentFrame()
  {
    return 1;
  }
```

RCM: "OK, that works. But it's stupid. Let's do the next test case."

```
public void testTwoThrowsNoMark()
{
  g.add(5);
  g.add(4);
  assertEquals(9, g.score());
  assertEquals(1, g.getCurrentFrame());
}
```

RCM: "That one's uninteresting; let's try the next."

```
public void testFourThrowsNoMark()
{
  g.add(5);
  g.add(4);
  g.add(7);
  g.add(2);
  assertEquals(18, g.score());
  assertEquals(9, g.scoreForFrame(1));
  assertEquals(18, g.scoreForFrame(2));
  assertEquals(2, g.getCurrentFrame());
}
```

RCM: "This one fails. Now let's make it pass."

RSK: "I think the algorithm is trivial. Just divide the number of throws by two, since there are two throws per frame. Unless we have a strike ... but we don't have strikes yet, so let's ignore them here too."

RCM: (Flails around adding and subtracting one until it works)

```
public int getCurrentFrame()
{
```

```
  return 1 + (itsCurrentThrow-1)/2;
}
```

RCM: "That isn't very satisfying."

RSK: "What if we don't calculate it each time? What if we adjust a **currentFrame** member variable after each throw?"

RCM: "OK, let's try that."

```
//Game.java----------------------------------
  public int getCurrentFrame()
  {
    return itsCurrentFrame;
  }

  public void add(int pins)
  {
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
    if (firstThrow == true)
    {
      firstThrow = false;
      itsCurrentFrame++;
    }
    else
    {
      firstThrow=true;;
    }
  }

  private int itsCurrentFrame = 0;
  private boolean firstThrow = true;
}
```

RCM: "OK, this works. But it also implies that the current frame is the frame of the last ball thrown, not the frame that the next ball will be thrown into. As long as we remember that, we'll be fine."

RSK: "I don't have that good of a memory, so let's make it more readable. But before we go screwing around with it some more, let's pull that code out of **add** and put it in a private member function called **adjustCurrentFrame** or something."

RCM: "OK, that sounds good"

```
  public void add(int pins)
  {
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
    adjustCurrentFrame();
  }

  private void adjustCurrentFrame()
  {
    if (firstThrow == true)
    {
```

```
      firstThrow = false;
      itsCurrentFrame++;
    }
    else
    {
      firstThrow=true;;
    }
  }
```

RCM: "Now let's change the variable and function names to be more clear. What should we call **itsCurrentFrame**?"

RSK: "I kind of like that name. I don't think we're incrementing it in the right place though. The current frame, to me, is the frame number that I'm throwing in. So it should get incremented right after the last throw in a frame."

RCM: "I agree. Let's change the test cases to reflect that; then we'll fix **adjustCurrentFrame**."

```
//TestGame.java-----------------------------------------
  public void testTwoThrowsNoMark()
  {
    g.add(5);
    g.add(4);
    assertEquals(9, g.score());
    assertEquals(2, g.getCurrentFrame());
  }

  public void testFourThrowsNoMark()
  {
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(3, g.getCurrentFrame());
  }
//TestGame.java-----------------------------------------
  private void adjustCurrentFrame()
  {
    if (firstThrow == true)
    {
      firstThrow = false;
    }
    else
    {
      firstThrow=true;
      itsCurrentFrame++;
    }
  }

  private int itsCurrentFrame = 1;
}
```

RCM: "OK, that's working. Now let's test **getCurrentFrame** in the two spare cases."

```
  public void testSimpleSpare()
  {
    g.add(3);
    g.add(7);
    g.add(3);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(2, g.getCurrentFrame());
  }

  public void testSimpleFrameAfterSpare()
  {
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(3, g.getCurrentFrame());
  }
```

RCM: "This works. Now, back to the original problem. We need **score** to work. We can now write **score** to call **scoreForFrame(getCurrentFrame()-1)**.

```
  public void testSimpleFrameAfterSpare()
  {
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(18, g.score());
    assertEquals(3, g.getCurrentFrame());
  }

//Game.java---------------------------------
  public int score()
  {
    return scoreForFrame(getCurrentFrame()-1);
  }
```

RCM: "This fails the **TestOneThrow** test case. Let's look at it."

```
  public void testOneThrow()
  {
    g.add(5);
    assertEquals(5, g.score());
    assertEquals(1, g.getCurrentFrame());
  }
```

RCM: "With only one throw, the first frame is incomplete. The **score** method is calling **scoreForFrame(0)**. This is yucky."

RSK: "Maybe, maybe not. Who are we writing this program for, and who is going to be calling **score**? Is it reasonable to assume that it won't get called on an incomplete frame?"

RCM: "Yeah. But it bothers me. To get around this, we have take the **score** out of the **testOneThrow** test case. Is that what we want to do?"

RSK: "We could. We could even eliminate the entire **testOneThrow** test case. It was used to ramp us up to the test cases of interest. Does it really serve a useful purpose now? We still have coverage in all of the other test cases."

RCM: "Yeah, I see your point. OK, out it goes." (Edits code, runs test, and gets green bar.) "Ahhh, that's better.

"Now, we'd better work on the strike test case. After all, we want to see all those **Frame** objects built into a linked list, don't we?" (snicker).

```
  public void testSimpleStrike()
  {
    g.add(10);
    g.add(3);
    g.add(6);
    assertEquals(19, g.scoreForFrame(1));
    assertEquals(28, g.score());
    assertEquals(3, g.getCurrentFrame());
  }
```

RCM: "OK, this compiles and fails as predicted. Now we need to make it pass."

```
//Game.java---------------------------------
public class Game
{
  public void add(int pins)
  {
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
    adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
    if (firstThrow == true)
    {
      if( pins == 10 ) // strike
        itsCurrentFrame++;
      else
        firstThrow = false;
    }
    else
    {
```

```
      firstThrow=true;
      itsCurrentFrame++;
    }
  }

  public int scoreForFrame(int theFrame)
  {
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      int firstThrow = itsThrows[ball++];
      if (firstThrow == 10)
      {
        score += 10 + itsThrows[ball] + itsThrows[ball+1];
      }
      else
      {
        int secondThrow = itsThrows[ball++];

        int frameScore = firstThrow + secondThrow;
        // spare needs next frames first throw
        if ( frameScore == 10 )
          score += frameScore + itsThrows[ball];
        else
        score += frameScore;
      }

    }

    return score;
  }
  private int itsScore = 0;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;
  private int itsCurrentFrame = 1;
  private boolean firstThrow = true;
}
```

RCM: "OK, that wasn't too hard. Let's see if it can score a perfect game."

```
  public void testPerfectGame()
  {
    for (int i=0; i<12; i++)
    {
      g.add(10);
    }
    assertEquals(300, g.score());
    assertEquals(10, g.getCurrentFrame());
  }
```

RCM: "Urg, it's saying the score is 330. Why would that be?"

RSK: "Because the current frame is getting incremented all the way to 12."

RCM: "Oh! We need to limit it to 10."

```
  private void adjustCurrentFrame(int pins)
  {
    if (firstThrow == true)
    {
      if( pins == 10 ) // strike
        itsCurrentFrame++;
      else
        firstThrow = false;
    }
    else
    {
      firstThrow=true;
      itsCurrentFrame++;
    }
    itsCurrentFrame = Math.min(10, itsCurrentFrame);
  }
```

RCM: "Damn, now it's saying that the score is 270. What's going on?"

RSK: "Bob, the **score** function is subtracting one from **getCurrentFrame**, so it's giving you the score for frame 9, not 10."

RCM: What? You mean I should limit the current frame to 11 not 10? I'll try it.

itsCurrentFrame = Math.min(**11**, itsCurrentFrame);

RCM: "OK, so now it gets the score correct, but fails because the current frame is 11 and not 10. Ick! this current frame thing is a pain in the butt. We want the current frame to be the frame the player is throwing into, but what does that mean at the end of the game?"

RSK: "Maybe we should go back to the idea that the current frame is the frame of the last ball thrown."

RCM: "Or maybe we need to come up with the concept of the last *completed* frame? After all, the score of the game at any point in time is the score in the last completed frame."

RSK: "A completed frame is a frame that you can write the score into, right?"

RCM: "Yes, a frame with a spare in it completes after the next ball. A frame with a strike in it completes after the next two balls. A frame with no mark completes after the second ball in the frame.

"Wait a minute.... We are trying to get the **score** method to work, right? All we need to do is force **score** to call **scoreForFrame(10)** if the game is complete."

RSK: "How do we know if the game is complete?"

RCM: "If **adjustCurrentFrame** ever tries to increment **itsCurrentFrame** past the 10th frame, then the game is complete."

RSK: "Wait. All you are saying is that if **getCurrentFrame** returns 11, the game is complete; that's the way the code works now!"

RCM: "Hmm. You mean we should change the test case to match the code?"

```
public void testPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    g.add(10);
  }
  assertEquals(300, g.score());
  assertEquals(11, g.getCurrentFrame());
}
```

RCM: "Well, that works. I suppose it's no worse than **getMonth** returning zero for January, but I still feel uneasy about it."

RSK: "Maybe something will occurr to us later. Right now, I think I see a bug. May I?" (Grabs keyboard.)

```
public void testEndOfArray()
{
  for (int i=0; i<9; i++)
  {
    g.add(0);
    g.add(0);
  }
  g.add(2);
  g.add(8); // 10th frame spare
  g.add(10); // Strike in last position of array.
  assertEquals(20, g.score());
}
```

RSK: "Hmm. That doesn't fail. I thought since the 21st position of the array was a strike, the scorer would try to add the 22nd and 23rd positions to the score. But I guess not."

RCM: "Hmm, you are still thinking about that **scorer** object, aren't you? Anyway, I see what you were getting at, but since **score** never calls **scoreForFrame** with a number larger than 10, the last strike is not actually counted as a strike. It's just counted at a 10 to complete the last spare. We never walk beyond the end of the array."

RSK: "OK, let's pump our original score card into the program."

```
public void testSampleGame()
{
  g.add(1);
  g.add(4);
  g.add(4);
  g.add(5);
  g.add(6);
  g.add(4);
```

```
    g.add(5);
    g.add(5);
    g.add(10);
    g.add(0);
    g.add(1);
    g.add(7);
    g.add(3);
    g.add(6);
    g.add(4);
    g.add(10);
    g.add(2);
    g.add(8);
    g.add(6);
    assertEquals(133, g.score());
  }
```

RSK: "Well, that works. Are there any other test cases that you can think of?"

RCM: "Yeah, let's test a few more boundary conditions. How about the poor schmuck who throws 11 strikes and then a final 9."

```
  public void testHeartBreak()
  {
    for (int i=0; i<11; i++)
      g.add(10);
    g.add(9);
    assertEquals(299, g.score());
  }
```

RCM: "That works. OK, how about a 10th frame spare?"

```
  public void testTenthFrameSpare()
  {
    for (int i=0; i<9; i++)
      g.add(10);
    g.add(9);
    g.add(1);
    g.add(1);
    assertEquals(270, g.score());
  }
}
```

RCM: (Staring happily at the green bar) "That works too. I can't think of any more, can you."

RSK: "No, I think we've covered them all. Besides I really want to refactor this mess. I still see the **scorer** object in there somewhere."

RCM: "OK, well, the **scoreForFrame** function is pretty messy. Let's consider it."

```
  public int scoreForFrame(int theFrame)
  {
    int ball = 0;
```

```
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    int firstThrow = itsThrows[ball++];
    if (firstThrow == 10)
    {
      score += 10 + itsThrows[ball] + itsThrows[ball+1];
    }
    else
    {
      int secondThrow = itsThrows[ball++];

      int frameScore = firstThrow + secondThrow;
      // spare needs next frames first throw
      if ( frameScore == 10 )
        score += frameScore + itsThrows[ball];
      else
        score += frameScore;
    }

  }

  return score;
}
```

RCM: "I'd really like to extract the body of that else clause into a seperate function named **handleSecondThrow**, but I can't because it uses **ball**, **firstThrow**, and **secondThrow** local variables."

RSK: "We could turn those locals into member variables."

RCM: "Yeah, that kind of reinforces your notion that we'll be able to pull the scoring out into its own **scorer** object. OK, let's give that a try."

RSK: (Grabs keyboard.)

```
private void adjustCurrentFrame(int pins)
{
  if (firstThrowInFrame == true)
  {
    if( pins == 10 ) // strike
      itsCurrentFrame++;
    else
      firstThrowInFrame = false;
  }
  else
  {
    firstThrowInFrame=true;
    itsCurrentFrame++;
  }
  itsCurrentFrame = Math.min(11, itsCurrentFrame);
}
```

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    firstThrow = itsThrows[ball++];
    if (firstThrow == 10)
    {
      score += 10 + itsThrows[ball] + itsThrows[ball+1];
    }
    else
    {
      secondThrow = itsThrows[ball++];

      int frameScore = firstThrow + secondThrow;
      // spare needs next frames first throw
      if ( frameScore == 10 )
        score += frameScore + itsThrows[ball];
      else
        score += frameScore;
    }
  }

  return score;
}
private int ball;
private int firstThrow;
private int secondThrow;

private int itsScore = 0;
private int[] itsThrows = new int[21];
private int itsCurrentThrow = 0;
private int itsCurrentFrame = 1;
private boolean firstThrowInFrame = true;
```

RSK: "I hadn't expected the name collision. We already had an instance variable named **firstThrow**. But it is better named **firstThrowInFrame**. Anyway, this works now. So we can pull the else clause out into its own function."

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    firstThrow = itsThrows[ball++];
    if (firstThrow == 10)
    {
      score += 10 + itsThrows[ball] + itsThrows[ball+1];
```

```
      }
      else
      {
        score += handleSecondThrow();
      }
    }

    return score;
  }

  private int handleSecondThrow()
  {
    int score = 0;
    secondThrow = itsThrows[ball++];

    int frameScore = firstThrow + secondThrow;
    // spare needs next frames first throw
    if ( frameScore == 10 )
      score += frameScore + itsThrows[ball];
    else
      score += frameScore;
    return score;
  }
```

RCM: "Look at the structure of **scoreForFrame**! In pseudocode, it looks something like this:"

```
if strike
  score += 10 + nextTwoBalls();
else
  handleSecondThrow.
```

RCM: "What if we changed it to:"

```
if strike
  score += 10 + nextTwoBalls();
else if spare
  score += 10 + nextBall();
else
  score += twoBallsInFrame()
```

RSK: "Geez! That's pretty much the rules for scoring bowling isn't it? OK, let's see if we can get that structure in the real function. First let's change the way the **ball** variable is being incremented so that the three cases manipulate it independently."

```
  public int scoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      firstThrow = itsThrows[ball];
      if (firstThrow == 10)
      {
```

```
      ball++;
      score += 10 + itsThrows[ball] + itsThrows[ball+1];
    }
    else
    {
      score += handleSecondThrow();
    }
  }

  return score;
}

private int handleSecondThrow()
{
  int score = 0;
  secondThrow = itsThrows[ball+1];

  int frameScore = firstThrow + secondThrow;
  // spare needs next frames first throw
  if ( frameScore == 10 )
  {
    ball+=2;
    score += frameScore + itsThrows[ball];
  }
  else
  {
    ball+=2;
    score += frameScore;
  }
  return score;
}
```

RCM: (Grabs keyboard.) "OK, now let's get rid of the **firstThrow** and **secondThrow** variables and replace them with appropriate functions."

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    firstThrow = itsThrows[ball];
    if (strike())
    {
      ball++;
      score += 10 + nextTwoBalls();
    }
    else
    {
      score += handleSecondThrow();
    }
  }
```

```
    return score;
  }

  private boolean strike()
  {
    return itsThrows[ball] == 10;
  }

  private int nextTwoBalls()
  {
    return itsThrows[ball] + itsThrows[ball+1];
  }
```

RCM: "That step works; let's keep going."

```
  private int handleSecondThrow()
  {
    int score = 0;
    secondThrow = itsThrows[ball+1];

    int frameScore = firstThrow + secondThrow;
    // spare needs next frames first throw
    if ( spare() )
    {
      ball+=2;
      score += 10 + nextBall();
    }
    else
    {
      ball+=2;
      score += frameScore;
    }
    return score;
  }

  private boolean spare()
  {
    return (itsThrows[ball] + itsThrows[ball+1]) == 10;
  }

  private int nextBall()
  {
    return itsThrows[ball];
  }
```

RCM: "OK, that works too. Now let's deal with **frameScore**."

```
  private int handleSecondThrow()
  {
    int score = 0;
    secondThrow = itsThrows[ball+1];

    int frameScore = firstThrow + secondThrow;
    // spare needs next frames first throw
    if ( spare() )
```

```
    {
      ball+=2;
      score += 10 + nextBall();
    }
    else
    {
      score += twoBallsInFrame();
      ball+=2;
    }
    return score;
  }

  private int twoBallsInFrame()
  {
    return itsThrows[ball] + itsThrows[ball+1];
  }
```

RSK: "Bob, you aren't incrementing **ball** in a consistent manner. In the spare and strike case, you increment before you calculate the score. In the **twoBallsInFrame** case, you increment *after* you calculate the score. And the code *depends* upon this order! What's up?"

RCM: "Sorry, I should have explained. I'm planning on moving the increments into **strike**, **spare**, and **twoBallsInFrame**. That way they'll disappear from the **scoreForFrame** function, and the function will look just like our pseudocode."

RSK: "OK, I'll trust you for a few more steps, but remember, I'm watching."

Kent: *"So am I."*

RCM: "OK, now since nobody uses **firstThrow**, **secondThrow**, and **frameScore** anymore, we can get rid of them."

```
  public int scoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      if (strike())
      {
        ball++;
        score += 10 + nextTwoBalls();
      }
      else
      {
        score += handleSecondThrow();
      }
    }

    return score;
  }
```

```
private int handleSecondThrow()
{
  int score = 0;
  // spare needs next frames first throw
  if ( spare() )
  {
    ball+=2;
    score += 10 + nextBall();
  }
  else
  {
    score += twoBallsInFrame();
    ball+=2;
  }
  return score;
}
```

RCM: (The sparkle in his eyes is a reflection of the green bar.) "Now, since the only variable that couples the three cases is **ball**, and since **ball** is dealt with independently in each case, we can merge the three cases together."

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    if (strike())
    {
      ball++;
      score += 10 + nextTwoBalls();
    }
    else if ( spare() )
    {
      ball+=2;
      score += 10 + nextBall();
    }
    else
    {
      score += twoBallsInFrame();
      ball+=2;
    }
  }

  return score;
}
```

RSK: (Peter Lorrie Gasp) "Master... Master... Let me do it. *Please*, let me do it."

RCM: "Ah, Igor, you would like to move the increments?"

RSK: "Yes, master. Oh *yes* master." (Grabs keyboard.)

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
    if (strike())
      score += 10 + nextTwoBalls();
    else if (spare())
      score += 10 + nextBall();
    else
      score += twoBallsInFrame();
  }

  return score;
}

private boolean strike()
{
  if (itsThrows[ball] == 10)
  {
    ball++;
    return true;
  }
  return false;
}

private boolean spare()
{
  if ((itsThrows[ball] + itsThrows[ball+1]) == 10)
  {
    ball += 2;
    return true;
  }
  return false;
}

private int nextTwoBalls()
{
  return itsThrows[ball] + itsThrows[ball+1];
}

private int nextBall()
{
  return itsThrows[ball];
}

private int twoBallsInFrame()
{
  return itsThrows[ball++] + itsThrows[ball++];
}
```

RCM: "Well *done* Igor!"

RSK: "Thank you master."

RCM: "Look at that **scoreForFrame** function. That's the rules of bowling stated about as succinctly as possible."

RSK: "But, Bob, what happened to the linked list of **Frame** objects?" (snicker, snicker)

RCM: (Sigh) **"We were bedevilled by the daemons of diagramatic overdesign. My God, three little boxes drawn on the back of a napkin, Game, Frame, and Throw, and it was still too complicated and just plain wrong."**

RSK: "We made a mistake starting with the **Throw** class. We should have started with the **Game** class first!"

RCM: "Indeed! So, next time let's try starting at the highest level and work down."

RSK: (Gasp) "Top Down Design!??!?! Could DeMarco have been right all along?"

RCM: "Correction: Top Down *Test First* Design. Frankly, I don't know if this is a good rule or not. It's just what would have helped us in this case. So next time, I'm going to try it and see what happens."

RSK: "Yeah, OK. Anyway we still have some refactoring to do. The **ball** variable is just a private iterator for **scoreForFrame** and its minions. They should all be moved into a different object."

RCM: "Oh, yes, your **Scorer** object. You were right after all. Let's do it."

RSK: (Grabs keyboard and takes several small steps punctuated by tests to create the following.)

```
//Game.java--------------------------------
public class Game
{
  public int score()
  {
    return scoreForFrame(getCurrentFrame()-1);
  }

  public int getCurrentFrame()
  {
    return itsCurrentFrame;
  }

  public void add(int pins)
  {
    itsScorer.addThrow(pins);
    itsScore += pins;
    adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
```

```
    if (firstThrowInFrame == true)
    {
      if( pins == 10 ) // strike
        itsCurrentFrame++;
      else
        firstThrowInFrame = false;
    }
    else
    {
      firstThrowInFrame=true;
      itsCurrentFrame++;
    }
    itsCurrentFrame = Math.min(11, itsCurrentFrame);
  }

  public int scoreForFrame(int theFrame)
  {
    return itsScorer.scoreForFrame(theFrame);
  }

  private int itsScore = 0;
  private int itsCurrentFrame = 1;
  private boolean firstThrowInFrame = true;
  private Scorer itsScorer = new Scorer();
}

//Scorer.java-----------------------------------
public class Scorer
{
  public void addThrow(int pins)
  {
    itsThrows[itsCurrentThrow++] = pins;
  }

  public int scoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      if (strike())
        score += 10 + nextTwoBalls();
      else if (spare())
        score += 10 + nextBall();
      else
        score += twoBallsInFrame();
    }

    return score;
  }

  private boolean strike()
  {
    if (itsThrows[ball] == 10)
    {
```

```
      ball++;
      return true;
    }
    return false;
  }

  private boolean spare()
  {
    if ((itsThrows[ball] + itsThrows[ball+1]) == 10)
    {
      ball += 2;
      return true;
    }
    return false;
  }

  private int nextTwoBalls()
  {
    return itsThrows[ball] + itsThrows[ball+1];
  }

  private int nextBall()
  {
    return itsThrows[ball];
  }

  private int twoBallsInFrame()
  {
    return itsThrows[ball++] + itsThrows[ball++];
  }

  private int ball;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;

}
```

RSK: "That's much better. Now **Game** just keeps track of frames, and **Scorer** just calculates the score. The SRP rocks!"

RCM: "Whatever. But it is better. Did you notice that the **itsScore** variable is not being used anymore?"

RSK: "Ha! You're right. Let's kill it." (Gleefully starts erasing things.)

```
  public void add(int pins)
  {
    itsScorer.addThrow(pins);
    adjustCurrentFrame(pins);
  }
```

RSK: "Not bad. Now should we clean up the **adjustCurrentFrame** stuff?"

RCM: "OK, let's look at it."

```
private void adjustCurrentFrame(int pins)
{
  if (firstThrowInFrame == true)
  {
    if( pins == 10 ) // strike
      itsCurrentFrame++;
    else
      firstThrowInFrame = false;
  }
  else
  {
    firstThrowInFrame=true;
    itsCurrentFrame++;
  }
  itsCurrentFrame = Math.min(11, itsCurrentFrame);
}
```

RCM: "OK, first let's extract the increments into a single function that also restricts the frame to 11. (Brrrr. I still don't like that 11.)"

RSK: "Bob, 11 means end of game."

RCM: "Yeah. Brrrr." (Grabs keyboard, makes a couple of changes punctuated by tests.)

```
private void adjustCurrentFrame(int pins)
{
  if (firstThrowInFrame == true)
  {
    if( pins == 10 ) // strike
      advanceFrame();
    else
      firstThrowInFrame = false;
  }
  else
  {
    firstThrowInFrame=true;
    advanceFrame();
  }
}

private void advanceFrame()
{
  itsCurrentFrame = Math.min(11, itsCurrentFrame + 1);
}
```

RCM: "OK, that's a little better. Now let's break out the strike case into its own function." (Takes a few small steps and runs tests between each.)

```
private void adjustCurrentFrame(int pins)
{
  if (firstThrowInFrame == true)
  {
    if (adjustFrameForStrike(pins) == false)
```

```
        firstThrowInFrame = false;
    }
    else
    {
      firstThrowInFrame=true;
      advanceFrame();
    }
  }

  private boolean adjustFrameForStrike(int pins)
  {
    if (pins == 10)
    {
      advanceFrame();
      return true;
    }
    return false;
  }
```

RCM: "That's pretty good. Now, about that 11."

RSK: "You really hate that don't you."

RCM: "Yeah, look at the **score** function:"

```
  public int score()
  {
    return scoreForFrame(getCurrentFrame()-1);
  }
```

RCM: "That -1 is odd. It's the only place we truly use **getCurrentFrame**, and yet we need to adjust what it returns."

RSK: "Damn, you're right. How many times have we reversed ourselves on this?"

RCM: "Too many. But there it is. The code wants **itsCurrentFrame** to represent the frame of the last thrown ball, not the frame we are about to throw into."

RSK: "Sheesh, that's going to break lots of tests cases."

RCM: "Actually, I think we should remove **getCurrentFrame** from all the test cases and remove the **getCurrentFrame** function itself. Nobody really uses it."

RSK: "OK, I get your point. I'll do it. It'll be like putting a lame horse out of its misery." (Grabs keyboard.)

```
//Game.java---------------------------------
  public int score()
  {
    return scoreForFrame(itsCurrentFrame);
  }
```

```
  private void advanceFrame()
  {
    itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
  }
```

RCM: "Oh for crying out loud. You mean to tell me that we were fretting over *that*. All we did was change the limit from 11 to 10 and remove the -1. Cripe."

RSK: "Yeah, Uncle Bob, it really wasn't worth all the angst we gave it."

RCM: "OK, looks like we are done. Let's just read through the whole program and see if it's as simple and communicative as it can be."

```
//Game.java---------------------------------
public class Game
{
  public int score()
  {
    return scoreForFrame(itsCurrentFrame);
  }

  public void add(int pins)
  {
    itsScorer.addThrow(pins);
    adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
    if (firstThrowInFrame == true)
    {
      if (adjustFrameForStrike(pins) == false)
        firstThrowInFrame = false;
    }
    else
    {
      firstThrowInFrame=true;
      advanceFrame();
    }
  }

  private boolean adjustFrameForStrike(int pins)
  {
    if (pins == 10)
    {
      advanceFrame();
      return true;
```

```
    }
    return false;
  }

  private void advanceFrame()
  {
    itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
  }

  public int scoreForFrame(int theFrame)
  {
    return itsScorer.scoreForFrame(theFrame);
  }

  private int itsCurrentFrame = 0;
  private boolean firstThrowInFrame = true;
  private Scorer itsScorer = new Scorer();
}

//Scorer.java-----------------------------------
public class Scorer
{
  public void addThrow(int pins)
  {
    itsThrows[itsCurrentThrow++] = pins;
  }

  public int scoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
      if (strike())
        score += 10 + nextTwoBalls();
      else if (spare())
        score += 10 + nextBall();
      else
        score += twoBallsInFrame();
    }

    return score;
  }

  private boolean strike()
  {
    if (itsThrows[ball] == 10)
    {
      ball++;
      return true;
    }
    return false;
  }

  private boolean spare()
```

```
  {
    if ((itsThrows[ball] + itsThrows[ball+1]) == 10)
    {
      ball += 2;
      return true;
    }
    return false;
  }

  private int nextTwoBalls()
  {
    return itsThrows[ball] + itsThrows[ball+1];
  }

  private int nextBall()
  {
    return itsThrows[ball];
  }

  private int twoBallsInFrame()
  {
    return itsThrows[ball++] + itsThrows[ball++];
  }

  private int ball;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;
}
```

RCM: "OK, that looks pretty good. I can't think of anything else to do."

RSK: "Yeah, it's pretty. Let's look over the tests for good measure."

```
//TestGame.java-----------------------------------------
import junit.framework.*;

public class TestGame extends TestCase
{
  public TestGame(String name)
  {
    super(name);
  }

  private Game g;

  public void setUp()
  {
    g = new Game();
  }

  public void testTwoThrowsNoMark()
```

```
{
  g.add(5);
  g.add(4);
  assertEquals(9, g.score());
}

public void testFourThrowsNoMark()
{
  g.add(5);
  g.add(4);
  g.add(7);
  g.add(2);
  assertEquals(18, g.score());
  assertEquals(9, g.scoreForFrame(1));
  assertEquals(18, g.scoreForFrame(2));
}

public void testSimpleSpare()
{
  g.add(3);
  g.add(7);
  g.add(3);
  assertEquals(13, g.scoreForFrame(1));
}

public void testSimpleFrameAfterSpare()
{
  g.add(3);
  g.add(7);
  g.add(3);
  g.add(2);
  assertEquals(13, g.scoreForFrame(1));
  assertEquals(18, g.scoreForFrame(2));
  assertEquals(18, g.score());
}

public void testSimpleStrike()
{
  g.add(10);
  g.add(3);
  g.add(6);
  assertEquals(19, g.scoreForFrame(1));
  assertEquals(28, g.score());
}

public void testPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    g.add(10);
  }
  assertEquals(300, g.score());
}

public void testEndOfArray()
{
  for (int i=0; i<9; i++)
```

```
    {
      g.add(0);
      g.add(0);
    }
    g.add(2);
    g.add(8); // 10th frame spare
    g.add(10); // Strike in last position of array.
    assertEquals(20, g.score());
   }

  public void testSampleGame()
  {
    g.add(1);
    g.add(4);
    g.add(4);
    g.add(5);
    g.add(6);
    g.add(4);
    g.add(5);
    g.add(5);
    g.add(10);
    g.add(0);
    g.add(1);
    g.add(7);
    g.add(3);
    g.add(6);
    g.add(4);
    g.add(10);
    g.add(2);
    g.add(8);
    g.add(6);
    assertEquals(133, g.score());
  }

  public void testHeartBreak()
  {
    for (int i=0; i<11; i++)
      g.add(10);
    g.add(9);
    assertEquals(299, g.score());
  }

  public void testTenthFrameSpare()
  {
    for (int i=0; i<9; i++)
      g.add(10);
    g.add(9);
    g.add(1);
    g.add(1);
    assertEquals(270, g.score());
  }
}
```

RSK: "That pretty much covers it. Can you think of any more meaningful test cases?"

RCM: "No, I think that's the set. There aren't any there that I'd be comfortable removing at this point."

RSK: "Then we're done."

RCM: "I'd say so. Thanks a lot for you help."

RSK: "No problem, it was fun."

Note

[1] Bertrand Meyer. *Object-Oriented Software Construction* (Prentice Hall, 1999).

About the Authors

Robert C. Martin has been a software professional since 1970. He is president of Object Mentor Inc., a firm of highly experienced experts that offers high level object-oriented software design consulting, training, and development services to major corporations around the world. In 1995, he authored the best-selling book: Designing Object Oriented C++ Applications using the Booch Method, published by Prentice Hall. In 1997, he was chief editor of the book: Pattern Languages of Program Design 3, published by Addison Wesley. In 2000, he was editor of the book More C++ Gems, published by Cambridge Press. From 1996 to 1998, he was the editor-in-chief of the C++ Report. He has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He is currently working in the area of lightweight productive processes and is a strong advocate and supporter of Extreme Programming.

Dr. Robert S Koss has been programming since 1973, starting in Fortran, migrating to C, then to C++, and lately to Java, Smalltalk, and Python. He has been a senior consultant at Object Mentor Inc. for the last two years, where his duties include writing and teaching courses, mentoring clients, and participating in the development of projects that are outsourced to the company. He has developed many systems in a variety of areas, from real-time embedded controllers to large database systems. He is also a strong proponent of lightweight processes and a very vocal advocate of Extreme Programming.