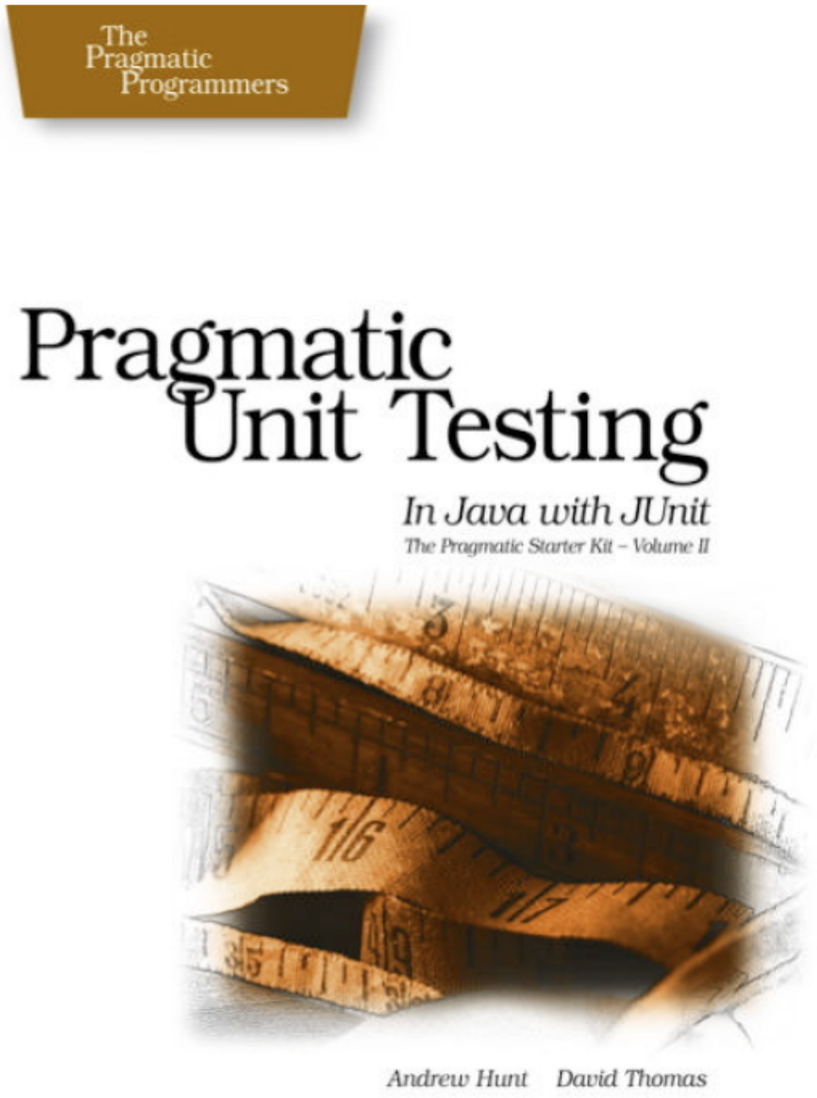
Examples

- Stack
- VideoPlayer



Stack Interface

```
public interface Stack
{
  public String pop ();

  public void push (String item);

  public String top ();

  public boolean isEmpty ();
}
```

ArrayStack

```
public class ArrayStack implements Stack
{
  private int      next_index;
  private String[] stack;

  public ArrayStack()
  {
    stack = new String[100];
    next_index = 0;
  }

  public String pop ()
  {
    if (next_index == 0)
    {
      throw new RuntimeException ("empty stack");
    }
    return stack[--next_index];
  }

  public void delete (int n)
  {
    next_index -= n;
  }

  public void push (String aString)
  {
    stack[next_index++] = aString;
  }

  public String top ()
  {
    if (next_index == 0)
    {
      throw new
          RuntimeException ("empty stack");
    }
    return stack[next_index - 1];
  }

  public boolean isEmpty ()
  {
    return next_index == 0;
  }
}
```

StackTest

- Test Fixture

```
public class StackTest
{
  private Stack testStack;

  @Before
  public void setUp()
  {
    testStack = new ArrayStack();
  }

  @After
  public void tearDown()
  {
    testStack = null;
  }
```

Test Specifications

1. Starting with an empty stack, call push() to push a test string onto the stack. Verify that top() returns that string several times in a row, and that isEmpty() returns false
2. For a brand-new stack, isEmpty() should be true, top() and pop() should throw exceptions.
3. Call pop() to remove the test string, and verify that it is the same string. isEmpty() should now be true. Call pop() again verify an exception is thrown.
4. Now do the same test again, but this time add multiple items to the stack - each of them strings which have the same value (say all "test"). Make sure you get the rights ones back, in the right order (the most recent item added should be the one returned). In this case, assertEquals() isn't good enough; you need assertSame() to ensure it's the same object
5. Push a null onto the stack and pop it; confirm you get a null back.
6. Ensure you can use the stack after it has thrown exceptions

1.

Starting with an empty stack, call push() to push a test string onto the stack. Verify that top() returns that string several times in a row, and that isEmpty() returns false

```
@Test
public void top()
{
  testStack.push("Item 1");
  assertEquals("Item 1", testStack.top());
  assertEquals("Item 1", testStack.top());
  assertEquals("Item 1", testStack.top());
  assertFalse(testStack.isEmpty());
}
```

2.

- For a brand-new stack, isEmpty() should be true, top() and pop() should throw exceptions.

```
public void testEmptyStack ()
{
  assertTrue(testStack.isEmpty());
  try
  {
    testStack.top();
    fail("should throw empty stack exception");
  }
  catch (Exception e)
  {
    assertTrue(true);
  }
  try
  {
    testStack.pop();
    fail("should throw empty stack exception");
  }
  catch (Exception e)
  {
    assertTrue(true);
  }
}
```

```
// Case 2
@Test
public void emptyStack()
{
  assertTrue(testStack.isEmpty());
}

// Case 2
@Test(expected = Exception.class)
public void testTopException()
{
  testStack.top();
}

// Case 2
@Test(expected = Exception.class)
public void testPopException()
{
  testStack.pop();
}
```

3.

- Call pop() to remove a test string, and verify that it is the same string. isEmpty() should now be true. Call pop() again verify an exception is thrown.

```
public void testPop() throws Exception
{
  String testStr = new String ("test");
  testStack.push(testStr);
  assertEquals(testStr, testStack.pop());
  try
  {
    testStack.pop();
    fail("Pop should throw exception");
  }
  catch (Exception e)
  {
    assertTrue(true);
  }
}
```

4.

- Now do the same test again, but this time add multiple items to the stack - each of them strings which have the same value (say all "test"). Make sure you get the right ones back, in the right order (the most recent item added should be the one returned). In this case, assertEquals() isn't good enough; you need assertSame() to ensure it's the same object

```
public void testPopDuplicate()
{
  String test1 = new String ("test");
  String test2 = new String ("test");
  String test3 = new String ("test");

  testStack.push(test1);
  testStack.push(test2);
  testStack.push(test3);

  assertSame(test3, testStack.pop());
  assertSame(test2, testStack.pop());
  assertSame(test1, testStack.pop());
}
```

5.

Push a null onto the stack and pop it; confirm you get a null back.

```
public void testNull()
{
  testStack.push(null);
  assertEquals (null, testStack.pop());
}
```

6.

- Ensure you can use the stack after it has thrown exceptions

```
public void testException()
{
  try
  {
    testStack.pop();
    fail("Pop should throw exception");
  }
  catch (Exception e)
  {
    assertTrue(true);
  }
  testStack.push("test");
  assertEquals ("test", testStack.top());
  assertFalse  (testStack.isEmpty());
}
```

CollectionStack

```
public class StackTest extends TestCase
{
  private Stack s;

  public void setUp() throws Exception
  {
    super.setUp();
    //s = new ArrayStack();
    s = new CollectionStack();
  }
```

```
public class CollectionStack implements Stack
{
  private java.util.Stack<String> stack;

  public CollectionStack()
  {
    stack = new java.util.Stack<String>();
  }

  public boolean isEmpty ()
  {
    return stack.isEmpty();
  }

  public String pop ()
  {
    return stack.pop();
  }

  public void push (String item)
  {
    stack.push(item);
  }

  public String top ()
  {
    return stack.peek();
  }
}
```

VideoPlayer Specification

- Implementations will provide methods to control a VCR or tape deck.
- There's the notion of a current position that lies somewhere between the beginning of tape (considered to be zero) and the end of tape - considered to be the tape duration in seconds

```
public interface VideoPlayer
{
  public void fastForward (int seconds);

  public void rewind (int seconds);

  public int currentTimePosition ();

  public void markTimePosition (String name);

  public void gotoMark (String name);
}
```

- You can ask for the current position and move from there to another given position. Fast-forward moves from current position toward end by some amount. Rewind moves from current position toward the begining by some amount. When tapes are first loaded, they are positioned at beginning automatically.
- A "Mark" can be established at any location - and this is remembered by the player. Going to a mark should make the location of the mark the current position.

VideoPlayer

- Initial implementation....

```
public class VideoPlayerImpl implements VideoPlayer
{
  private int duration;
  private int currentPosition;
  private Map<String, Integer> marks;

  public VideoPlayerImpl(int length)
  {
    this.duration = length;
    marks = new HashMap<String, Integer>();
  }

  public int currentTimePosition ()
  {
    return currentPosition;
  }

  public void gotoMark (String name)
  { }

  public void markTimePosition (String name)
  { }

  public void rewind (int seconds)
  { }

  public void fastForward (int seconds)
  {
  }
}
```

Test Case Specifications

1. Verify that the initial position is 0.
2. Fast forward by some allowed amount (not past end of tape), then rewind by same amount. Should be at initial location.
3. Rewind by some allowed amount amount (not past beginning of tape), then fast forward by same amount. Should be at initial location.
4. Fast forward past end of tape, then rewind by same amount. Should be before the initial location by an appropriate amount to reflect the fact that you can't advance the location past the end of tape.
5. Try the same thing in the other direction (rewind past beginning of tape).
6. Mark various positions and return to them after moving the current position around.
7. Mark a position and return to it without moving in between

Test Fixture

```
public class VideoPlayerTest
{
  private VideoPlayer editor;

  @Before
  public void setUp() throws Exception
  {
    editor = new VideoPlayerImpl(100);
  }

  @After
  public void tearDown() throws Exception
  {
    editor = null;
  }
```

1.

```
public void testBOT()
{
  assertEquals(0, editor.currentTimePosition());
}
```

- Verify that the initial position is 0

```
public class VideoPlayerImpl implements VideoPlayer
{
  private int duration;
  private int currentPosition;
  private Map<String, Integer> marks;

  public VideoPlayerImpl(int length)
  {
    this.duration = length;
    marks = new HashMap<String, Integer>();
  }

  public int currentTimePosition ()
  {
    return currentPosition;
  }
```

2.

- Fast forward by some allowed amount (not past end of tape), then rewind by same amount. Should be at initial location.

```
public void testFF()
{
  editor.fastForward(30);
  editor.rewind(30);
  assertEquals(0, editor.currentTimePosition());
}
```

```
public void rewind (int seconds)
{
   currentPosition -= seconds;
}

public void fastForward (int seconds)
{
  currentPosition += seconds;
}
```

3.

- Rewind by some allowed amount amount (not past beginning of tape), then fast forward by same amount. Should be at initial location

```
public void testRewind()
{
  editor.fastForward(30);

  editor.rewind(20);
  editor.fastForward(20);
  assertEquals(30, editor.currentTimePosition());
}
```

4.

- Fast forward past end of tape, then rewind by same amount. Should be before the initial location by an appropriate amount to reflect the fact that you can't advance the location past the end of tape

```
public void testFFBeyondEnd()
{
  editor.fastForward(50);

  editor.fastForward(60);
  editor.rewind(60);
  assertEquals(40, editor.currentTimePosition());
}
```

```
public void fastForward (int seconds)
{
  if ((currentPosition + seconds) < duration)
  {
    currentPosition += seconds;
  }
  else
  {
    currentPosition = 100;
  }
}
```

5

- Try the same thing in the other direction (rewind past beginning of tape)

```
public void testRewindBeyondStart()
{
  editor.fastForward(50);

  editor.rewind(60);
  editor.fastForward(60);
  assertEquals(60, editor.currentTimePosition());
}
```

```
public void rewind (int seconds)
{
  if ((currentPosition - seconds) >= 0)
  {
    currentPosition -= seconds;
  }
  else
  {
    currentPosition = 0;
  }
}
```

6,7

- Mark various positions and return to them after moving the current position around.
- Mark a position and return to it without moving in between

```
public void testMark()
{
  editor.fastForward(30);
  editor.markTimePosition("one");
  editor.fastForward(30);
  editor.markTimePosition("two");

  editor.gotoMark("one");
  assertEquals(30, editor.currentTimePosition());
  editor.gotoMark("two");
  assertEquals(60, editor.currentTimePosition());
}
```

```
public void gotoMark (String name)
{
  if (marks.containsKey(name))
  {
    currentPosition = marks.get(name);
  }
}

public void markTimePosition (String name)
{
  marks.put(name, currentPosition);
}
```