Experimento de Refatoração de Test Smells Parte 2

Nome: Gabriel Maia Gondim - 478943

Projeto: https://github.com/wilhelmSt/Trab_02QS/

Eu estou atualmente trabalhando na refatoração dos test smells seguintes:

5 Test smells do tipo ignored test no arquivo
commons-cli\src\test\java\org\apache\commons\cli\GnuParserTest.java

Código original:

```
123     @Override
124     @Test
125     @Ignore("not supported by the GnuParser")
126     public void testStopBursting2() throws Exception {
127     }
128
129     @Override
130     @Test
131     @Ignore("not supported by the GnuParser")
132     public void testUnambiguousPartialLongOption1() throws Exception {
133     }
134
135     @Override
136     @Test
137     @Ignore("not supported by the GnuParser")
138     public void testUnambiguousPartialLongOption2() throws Exception {
139     }
140
141     @Override
142     @Test
143     @Ignore("not supported by the GnuParser")
144     public void testUnambiguousPartialLongOption3() throws Exception {
145     }
146
147     @Override
148     @Test
149     @Ignore("not supported by the GnuParser")
150     public void testUnambiguousPartialLongOption4() throws Exception {
151     }
```

Código refatorado:
Apenas removi os testes, pois estavam vazios.

5 Test smells do tipo magic number test no arquivo
commons-pool\src\test\java\org\apache\commons\pool2\impl\TestEvictionTimer.java

Código original:

```
68          final ThreadPoolExecutor evictionExecutor = (ThreadPoolExecutor) evictorExecutorField.get(null);
69          assertEquals(2, evictionExecutor.getQueue().size()); // Reaper plus one eviction task
70          assertEquals(1, EvictionTimer.getNumTasks());
71
72          // Start evictor #2
73          final BaseGenericObjectPool<String, RuntimeException>.Evictor evictor2 = pool.new Evictor();
74          EvictionTimer.schedule(evictor2, TestConstants.ONE_MINUTE_DURATION, TestConstants.ONE_MINUTE_DURATION);
75
76          // Assert that eviction objects are correctly allocated
77          // 1 - the evictor timer task is created
78          sf = (ScheduledFuture<?>) evictorTaskFutureField.get(evictor2);
79          assertFalse(sf.isCancelled());
80          // 2- and, the eviction action is added to executor thread pool
81          assertEquals(3, evictionExecutor.getQueue().size()); // Reaper plus 2 eviction tasks
82          assertEquals(2, EvictionTimer.getNumTasks());
83
84          // Stop evictor #1
85          EvictionTimer.cancel(evictor1, BaseObjectPoolConfig.DEFAULT_EVICTOR_SHUTDOWN_TIMEOUT, false);
86
87          // Assert that eviction objects are correctly cleaned
88          // 1 - the evictor timer task is cancelled
89          sf = (ScheduledFuture<?>) evictorTaskFutureField.get(evictor1);
90          assertTrue(sf.isCancelled());
91          // 2- and, the eviction action is removed from executor thread pool
92          final ThreadPoolExecutor evictionExecutorOnStop = (ThreadPoolExecutor) evictorExecutorField.get(null);
93          assertEquals(2, evictionExecutorOnStop.getQueue().size());
94          assertEquals(1, EvictionTimer.getNumTasks());
95
```

Código refatorado:

```
70          int expectedQueueSize = 2;
71          int expectedNumberTasks = 1;
72
73          assertEquals(expectedQueueSize, evictionExecutor.getQueue().size()); // Reaper plus one eviction task
74          assertEquals(expectedNumberTasks, EvictionTimer.getNumTasks());
75
76          // Start evictor #2
77          final BaseGenericObjectPool<String, RuntimeException>.Evictor evictor2 = pool.new Evictor();
78          EvictionTimer.schedule(evictor2, TestConstants.ONE_MINUTE_DURATION, TestConstants.ONE_MINUTE_DURATION);
79
80          // Assert that eviction objects are correctly allocated
81          // 1 - the evictor timer task is created
82          sf = (ScheduledFuture<?>) evictorTaskFutureField.get(evictor2);
83          assertFalse(sf.isCancelled());
84          // 2- and, the eviction action is added to executor thread pool
85
86          expectedQueueSize = 3;
87          expectedNumberTasks = 2;
88
89          assertEquals(expectedQueueSize, evictionExecutor.getQueue().size()); // Reaper plus 2 eviction tasks
90          assertEquals(expectedNumberTasks, EvictionTimer.getNumTasks());
91
92          // Stop evictor #1
93          EvictionTimer.cancel(evictor1, BaseObjectPoolConfig.DEFAULT_EVICTOR_SHUTDOWN_TIMEOUT, false);
94
95          // Assert that eviction objects are correctly cleaned
96          // 1 - the evictor timer task is cancelled
97          sf = (ScheduledFuture<?>) evictorTaskFutureField.get(evictor1);
98          assertTrue(sf.isCancelled());
99          // 2- and, the eviction action is removed from executor thread pool
100         final ThreadPoolExecutor evictionExecutorOnStop = (ThreadPoolExecutor) evictorExecutorField.get(null);
101
102         expectedQueueSize = 2;
103         expectedNumberTasks = 1;
104
105         assertEquals(expectedQueueSize, evictionExecutorOnStop.getQueue().size());
106         assertEquals(expectedNumberTasks, EvictionTimer.getNumTasks());
```

5 Test smells do tipo lazy test no arquivo commons-math-legacy\src\test\java\org\apache\commons\math4\legacy\ode\nonstiff\AdamsBashforthIntegratorTest.java

Código original:

```java
@Test(expected = MaxCountExceededException.class)
public void exceedMaxEvaluations() throws DimensionMismatchException, NumberIsTooSmallException, MaxCountExceededException,
    NoBracketingException {

    TestProblem1 pb  = new TestProblem1();
    double range = pb.getFinalTime() - pb.getInitialTime();

    AdamsBashforthIntegrator integ = new AdamsBashforthIntegrator(2, 0, range, 1.0e-12, 1.0e-12);
    TestProblemHandler handler = new TestProblemHandler(pb, integ);
    integ.addStepHandler(handler);
    integ.setMaxEvaluations(650);
    integ.integrate(pb,
                    pb.getInitialTime(), pb.getInitialState(),
                    pb.getFinalTime(), new double[pb.getDimension()]);
}

@Test
public void backward() throws DimensionMismatchException, NumberIsTooSmallException, MaxCountExceededException, NoBracketingException {

    TestProblem5 pb = new TestProblem5();
    double range = JdkMath.abs(pb.getFinalTime() - pb.getInitialTime());

    AdamsBashforthIntegrator integ = new AdamsBashforthIntegrator(4, 0, range, 1.0e-12, 1.0e-12);
    integ.setStarterIntegrator(new PerfectStarter(pb, (integ.getNSteps() + 5) / 2));
    TestProblemHandler handler = new TestProblemHandler(pb, integ);
    integ.addStepHandler(handler);
    integ.integrate(pb, pb.getInitialTime(), pb.getInitialState(),
                    pb.getFinalTime(), new double[pb.getDimension()]);

    Assert.assertEquals(0.0, handler.getLastError(), 4.3e-8);
    Assert.assertEquals(0.0, handler.getMaximalValueError(), 4.3e-8);
    Assert.assertEquals(0, handler.getMaximalTimeError(), 1.0e-16);
    Assert.assertEquals("Adams-Bashforth", integ.getName());
}

@Test
public void polynomial() throws DimensionMismatchException, NumberIsTooSmallException, MaxCountExceededException, NoBracketingException
    {
    TestProblem6 pb = new TestProblem6();
    double range = JdkMath.abs(pb.getFinalTime() - pb.getInitialTime());

    for (int nSteps = 2; nSteps < 8; ++nSteps) {
        AdamsBashforthIntegrator integ =
            new AdamsBashforthIntegrator(nSteps, 1.0e-6 * range, 0.1 * range, 1.0e-4, 1.0e-4);
        integ.setStarterIntegrator(new PerfectStarter(pb, nSteps));
        TestProblemHandler handler = new TestProblemHandler(pb, integ);
        integ.addStepHandler(handler);
        integ.integrate(pb, pb.getInitialTime(), pb.getInitialState(),
                        pb.getFinalTime(), new double[pb.getDimension()]);
        if (nSteps < 5) {
            Assert.assertTrue(handler.getMaximalValueError() > 0.005);
        } else {
            Assert.assertTrue(handler.getMaximalValueError() < 5.0e-10);
        }
    }
}
```

Código refatorado:

```java
@Test(expected = MaxCountExceededException.class)
public void exceedMaxEvaluationsPolynomialBackward() throws DimensionMismatchException, NumberIsTooSmallException,
    MaxCountExceededException, NoBracketingException {

    // test if it exceeds max evaluations
    TestProblem6 pb = new TestProblem6();
    double range = JdkMath.abs(pb.getFinalTime() - pb.getInitialTime());

    for (int nSteps = 2; nSteps < 8; ++nSteps) {
        AdamsBashforthIntegrator integ =
            new AdamsBashforthIntegrator(nSteps, 1.0e-6 * range, 0.1 * range, 1.0e-4, 1.0e-4);
        integ.setStarterIntegrator(new PerfectStarter(pb, nSteps));
        TestProblemHandler handler = new TestProblemHandler(pb, integ);
        integ.addStepHandler(handler);
        integ.integrate(pb, pb.getInitialTime(), pb.getInitialState(),
                        pb.getFinalTime(), new double[pb.getDimension()]);
        if (nSteps < 5) {
            Assert.assertTrue(handler.getMaximalValueError() > 0.005);
        } else {
            Assert.assertTrue(handler.getMaximalValueError() < 5.0e-10);
        }
    }

    // test backward computation
    TestProblem5 pb2 = new TestProblem5();
    double range2 = JdkMath.abs(pb2.getFinalTime() - pb2.getInitialTime());

    AdamsBashforthIntegrator integ2 = new AdamsBashforthIntegrator(4, 0, range2, 1.0e-12, 1.0e-12);
    integ2.setStarterIntegrator(new PerfectStarter(pb2, (integ2.getNSteps() + 5) / 2));
    TestProblemHandler handler2 = new TestProblemHandler(pb2, integ2);
    integ2.addStepHandler(handler2);
    integ2.integrate(pb2, pb2.getInitialTime(), pb2.getInitialState(),
                     pb2.getFinalTime(), new double[pb2.getDimension()]);

    Assert.assertEquals(0.0, handler2.getLastError(), 4.3e-8);
    Assert.assertEquals(0.0, handler2.getMaximalValueError(), 4.3e-8);
    Assert.assertEquals(0, handler2.getMaximalTimeError(), 1.0e-16);
    Assert.assertEquals("Adams-Bashforth", integ2.getName());

    // test polynomial computation
    TestProblem1 pb3  = new TestProblem1();
    double range3 = pb3.getFinalTime() - pb3.getInitialTime();

    AdamsBashforthIntegrator integ3 = new AdamsBashforthIntegrator(2, 0, range3, 1.0e-12, 1.0e-12);
    TestProblemHandler handler3 = new TestProblemHandler(pb3, integ3);
    integ3.addStepHandler(handler3);
    integ3.setMaxEvaluations(650);
    integ3.integrate(pb3,
                     pb3.getInitialTime(), pb3.getInitialState(),
                     pb3.getFinalTime(), new double[pb3.getDimension()]);
}
```

Minhas principais dificuldades ao remover essas anormalidades foram:

Os três testes foram bem simples de resolver, especialmente o primeiro que por serem um conjunto de testes vazios era só apagar os testes.

Eu estou usando as seguintes técnicas de refatoração para remover test smells:

Ignored test: Remover método fazio (safe delete empty method), removendo os testes desnecessários.
Magic number test: Extrair variável (extract variable), guardando o valor mágico em uma variável com um nome relevante e explicativo.
Lazy test: Método em linha (inline method), unindo as funções de teste separadas em uma só.

De 0 a 10, quão prejudicial é esse test smell para o sistema? Por que?

Ignored test: 4.

Ele não faz tanta diferença por ser simplesmente um método vazio, mas polui o código desnecessariamente e a visualização dos testes passados e falhos.

Magic number test: 7.
Apesar de ser simplesmente um número introduzido magicamente e os testes passarem, ele dificulta o entendimento do teste, pois o programador não vai saber exatamente qual valor está sendo testado e porque esse é o valor esperado.

Lazy test: 2.
Muitas vezes, funções de um objeto de produção tem vários casos diferentes que precisam ser testados. Testar esses casos em funções separadas dá benefícios como poder isolar casos que tiveram sucesso ou falha, podendo detectar mais facilmente em quais partes do código está o erro. Testar esses casos em uma mesma função segue a ideia de coesão, onde uma função de teste testa todos os casos para uma função de um objeto de produção. Acredito que para cada caso um vale mais a pena que o outro, não tendo, na maior parte dos casos, tanto impacto assim escolher um no lugar do outro.