**17 вопрос. Структура тестов.**

**Какими отличительными признаками характеризуется чистый тест? Тремя: удобочитаемостью, удобочитаемостью и удобочитаемостью.** Вероятно, удобочитаемость в модульных тестах играет еще более важную роль, чем в коде продукта. Что делает тестовый код удобочитаемым? То же, **что делает удобочитаемым любой другой код: ясность, простота и выразительность. В тестовом коде необходимо передать максимум информации минимумом выразительных средств.**

В листинге 9.1 приведен фрагмент кода из проекта FitNesse. Эти три теста трудны для понимания; несомненно, их можно усовершенствовать. Прежде всего, повторные вызовы addPage и assertSubString содержат огромное количество повторяющегося кода [G5]. Что еще важнее, код просто забит второстепенными подробностями, снижающими выразительность теста.

**Листинг 9.1.** SerializedPageResponderTest.java

public void testGetPageHieratchyAsXml() throws Exception {

crawler.addPage(root, PathParser.parse("PageOne"));

crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));

crawler.addPage(root, PathParser.parse("PageTwo"));

request.setResource("root");

request.addInput("type", "pages");

Responder responder = new SerializedPageResponder();

SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request);

String xml = response.getContent();

assertEquals("text/xml", response.getContentType());

assertSubString("PageOne", xml); assertSubString("PageTwo", xml); assertSubString("ChildOne", xml);

}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks() throws Exception {

WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));

crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));

crawler.addPage(root, PathParser.parse("PageTwo"));

PageData data = pageOne.getData();

WikiPageProperties properties = data.getProperties();

WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY\_NAME);

symLinks.set("SymPage", "PageTwo");

pageOne.commit(data);

request.setResource("root");

request.addInput("type", "pages");

Responder responder = new SerializedPageResponder();

SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request);

String xml = response.getContent();

assertEquals("text/xml", response.getContentType());

assertSubString("PageOne", xml);

assertSubString("PageTwo", xml);

assertSubString("ChildOne", xml);

assertNotSubString("SymPage", xml);

}

public void testGetDataAsHtml() throws Exception {

crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

request.setResource("TestPageOne");

request.addInput("type", "data");

Responder responder = new SerializedPageResponder();

SimpleResponse response =

(SimpleResponse) responder.makeResponse( new FitNesseContext(root), request); String xml = response.getContent();

assertEquals("text/xml", response.getContentType());

assertSubString("test page", xml);

assertSubString("<Test", xml);

}

В общем, этот код не предназначался для чтения. На несчастного читателя обрушивается целый водопад мелочей, в которых необходимо разобраться, чтобы уловить в тестах хоть какой-то смысл.

Теперь рассмотрим усовершенствованные тесты в листинге 9.2. Они делают абсолютно то же самое, но код был переработан в более ясную и выразительную форму.

**Листинг 9.2.** SerializedPageResponderTest.java (переработанная версия)

public void testGetPageHierarchyAsXml() throws Exception {

makePages("PageOne", "PageOne.ChildOne", "PageTwo");

submitRequest("root", "type:pages");

assertResponseIsXML();

assertResponseContains(

"<name>PageOne<name>","<name>PageTwo<name>", "<name>ChildOne<name>"

);

}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {

WikiPage page = makePage("PageOne");

makePages("PageOne.ChildOne", "PageTwo");

addLinkTo(page, "PageTwo", "SymPage");

submitRequest("root", "type:pages");

assertResponseIsXML();

assertResponseContains(

"<name>PageOne<name>","<name>PageTwo<name>", "<name>ChildOne<name>"

);

assertResponseDoesNotContain("SymPage");

}

public void testGetDataAsXml() throws Exception {

makePageWithContent("TestPageOne", "test page");

submitRequest("TestPageOne", "type:data");

assertResponseIsXML();

assertResponseContains("test page", "<Test”);

}

В структуре тестов очевидно воплощен **паттерн ПОСТРОЕНИЕ-ОПЕРАЦИИ-ПРОВЕРКА** . **Каждый тест четко делится на три части. Первая часть строит тестовые данные, вторая часть выполняет операции с тестовыми данными, а третья часть проверяет, что операция привела к ожидаемым результатам.**

**Предметно-ориентированный язык тестирования**

Тесты в листинге 9.2 демонстрируют методику построения предметно-ориентированного языка для программирования тестов. **Вместо вызова функций API, используемых программистами для манипуляций с системой, мы строим набор функций и служебных программ, использующих API; это упрощает написание и чтение тестов. Наши функции и служебные программы образуют специализированный API, то есть по сути — язык тестирования, который программисты используют для упрощения работы над тестами, а также чтобы помочь другим программистам, которые будут читать эти тесты позднее.** Тестовый API не проектируется заранее; он развивается на базе многократной переработки тестового кода, перегруженного ненужными подробностями. По аналогии с тем, как я переработал листинг 9.1 в листинг 9.2, дисциплинированные разработчики перерабатывают свой тестовый код в более лаконичные и выразительные формы.

**Одна проверка на тест**

**Существует точка зрения , согласно которой каждая тестовая функция в тесте JUnit должна содержать одну — и только одну — директиву assert.** Тесты приводят к одному выводу, который можно быстро и легко понять при чтении.

Но что вы скажете о листинге 9.2? В нем объединена проверка двух условий: что выходные данные представлены в формате XML и они содержат некоторые подстроки. На первый взгляд такое решение выглядит сомнительно. Впрочем, тест можно разбить на два отдельных теста, каждый из которых имеет собственную директиву assert, как показано в листинге 9.7.

**Листинг 9.7.** SerializedPageResponderTest.java (одна директива assert)

public void testGetPageHierarchyAsXml() throws Exception {

givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

whenRequestIsIssued("root", "type:pages");

thenResponseShouldBeXML();

}

public void testGetPageHierarchyHasRightTags() throws Exception {

givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

whenRequestIsIssued("root", "type:pages");

thenResponseShouldContain(

"<name>PageOne<name>","<name>PageTwo<name>", "<name>ChildOne<name>"

);

}

Обратите внимание: я переименовал функции в соответствии со **стандартной схемой given-when-then** [RSpec]. Это еще сильнее упрощает чтение тестов. К сожалению, такое разбиение приводит к появлению большого количества дублирующегося кода.

**Я думаю, что правило «одного assert» является хорошей рекомендацией.** Обычно я стараюсь создать предметно-ориентированный язык тестирования, который это правило поддерживает, как в листинге 9.5. Но при этом я не боюсь включать в свои тесты более одной директивы assert. Вероятно, лучше всего сказать, что количество директив assert в тесте должно быть сведено к минимуму.