第5章

软件测试工具

本章将对多种测试工具进行详细介绍，并通过具体实例对 BoundsChecker、JUnit、LoadRunner、Monkey 等测试工具，以及测试管理工具禅道进行演示。

5.1 白盒测试工具 BoundsChecker

BoundsChecker 是一种常用的动态白盒测试工具，其常用于单元测试中的代码错误检查，使用简单且高效。下面将对其进行详细介绍，帮助读者更快地了解以及学会使用该工具。

5.1.1 安装

BoundsChecker 是集成在 Visual C++ 上的一个插件，因此在安装 BoundsChecker 之前，首先要确保计算机中已经安装了 Visual C++。在获得 BoundsChecker 安装程序后，以管理员的身份运行 setup.exe 文件，按照提示进行安装即可。安装成功以后，可在 Visual C++ 环境下看到名为 BoundsChecker 的菜单，如图 5.1 所示。

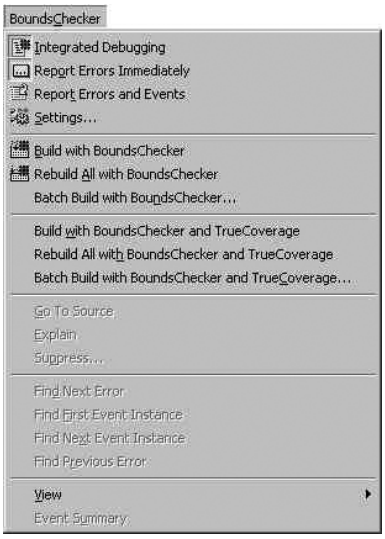

图 5.1 BoundsChecker 在 Visual C++ 环境下的菜单

5.1.2 功能与模式

BoundsChecker 是一个功能强大、使用方便的代码检查工具，在程序运行期间，它可以跟踪以下 3 种程序错误。

（1）指针操作和内存、资源泄露错误。例如，对指针变量的错误操作、内存泄漏等问题。

（2）内存的错误操作。例如，未初始化内存空间即使用，内存的读写溢出等错误。

（3）API 函数使用错误。

BoundsChecker 提供两种模式给用户进行错误检测：一种为 ActiveCheck；另一种为 FinalCheck。与 FinalCheck 模式相比，ActiveCheck 检测的错误类型有限，一般为内存泄漏错误、资源泄露错误、API 函数使用错误。而 FinalCheck 则包含了 BoundsChecker 可以检测的所有的错误类型，除 ActiveCheck 可以检测的错误类型之外，还可以对指针错误操作、内存溢出等内存操作错误进行检测，提供更详细的错误信息，其是 ActiveCheck 的超集。

5.1.3 ActiveCheck 模式

使用 ActiveCheck 模式检查的步骤如下。

（1）在 Visual C++ 环境下打开要测试的程序，并使程序处于 Debug 状态下。

（2）选择 BoundsChecker 菜单中的 Integrated Debugging 与 Report Errors and Events 命令，确保 BoundsChecker 可以发挥作用，如图 5.2 所示。

（3）选择 Visual C++ Build 菜单的 Start Debug→Go 命令，使程序在 Debug 状态下运行，ActiveCheck 也将在后台下启动并进行错误检测。当程序运行结束后，BoundsChecker 将给出错误报告。

（4）如果选择 Report Errors Immediately 命令，如图 5.3 所示，则当检测到错误发生时会立即停止运行，并弹出错误提示框，如图 5.4 所示。

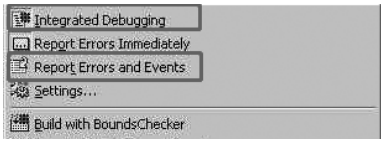

图 5.2 BoundsChecker 菜单选项（一）

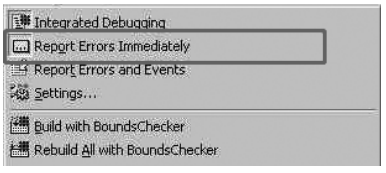

图 5.3 BoundsChecker 菜单选项（二）

单击第 1 个按钮表示暂时忽略这个错误，继续运行程序。

单击第 2 个按钮 BoundsChecker 会跳转到程序出现问题的代码处。待处理完问题，可选择 Debug→go 命令继续运行。

单击第 3 个按钮表示将该错误添加至可忽略列表，对以后出现的此错误不予报告。

单击第 4 个按钮表示终止程序的运行。

单击第 5 个按钮会显示当前的内存使用情况。

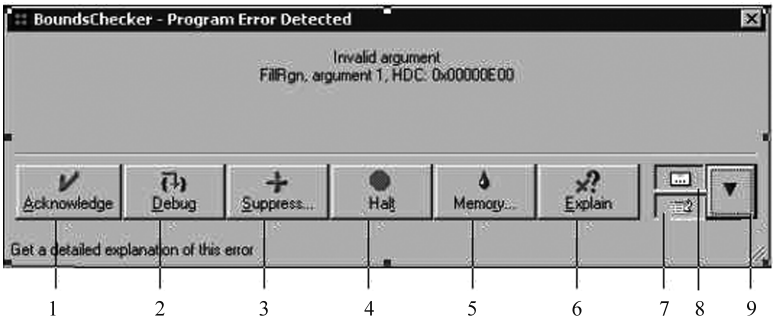

图 5.4　错误提示框

单击第 6 个按钮会显示当前错误的有关帮助信息。

单击第 7 个按钮即代表选择 Report Errors Immediately 命令。

单击第 8 个按钮代表选择 Report Errors and Events 命令。

单击第 9 个按钮会显示或者隐藏与错误相关的函数调用堆栈情况。

5.1.4　FinalCheck 模式

如需在 FinalCheck 模式下测试程序，则不能使用 Visual C++ 集成开发环境提供的编译连接器来构造程序，而必须要使用 BoundsChecker 提供的编译连接器。当 BoundsChecker 的编译连接器编译连接程序时，会向程序中插桩一些错误检测代码，因此 FinalCheck 能够比 ActiveCheck 找到更多的错误。

使用 FinalCheck 模式进行错误检查的步骤如下。

(1) 在 Visual C++ 环境下打开要测试的程序。

(2) 选择 Build 菜单的 Configurations 命令。

(3) 在弹出的对话框中单击 Add 按钮，在 Configuration 文本框中添加需要创建的文件夹名称。

(4) 在 Copy settings from 组合框中选择×××-Win32 Debug 选项，单击 OK 按钮，然后再单击 Close 按钮即可。

(5) 选择 Build 菜单的 Set Active Configuration 命令，选中步骤(3)中新建的文件夹，单击 OK 按钮，这样 BoundsChecker 的编译连接器编译连接程序时生成的中间文件、可执行程序都会被放到该文件夹下。

(6) 选择 BoundsChecker 菜单的 Rebuild All with BoundsChecker 命令，对程序重新进行编译连接。在这个过程中，BoundsChecker 将会向被测程序的代码中加入错误检测码。

(7) 选择 BoundsChecker 菜单中的 Integrated Debugging 与 Report Errors and Events 命令。

(8) 运行 Visual C++ 环境中 Build 菜单的 Start Debug→Go 命令，程序将在 Debug 状态下运行。

两种模式都可以对程序进行检测，但正如 5.1.2 节所述，ActiveCheck 所能检测的错

误类型有限，而 FinalCheck 模式下可以检测 BoundsChecker 所能支持检测的所有错误类型，与此同时，FinalCheck 需要付出运行速度变慢的代价。

5.1.5 结果分析

在程序结束以后，BoundsChecker 会给出一份错误列表，其将包含程序中出现的内存泄漏、指针操作错误、API 函数使用错误以及资源泄漏等错误，如图 5.5 所示。测试人员需要根据错误列表进行分析，找到错误原因以及位置。

从图 5.5 中可以看出，左边的窗格中逐条给出了程序在内存、资源、API 函数使用上的问题，包括问题的类型、出现次数、具体问题描述等。在单击某条问题时，右边窗格会显示与该问题相关的函数调用堆栈情况，双击某条问题时，BoundsChecker 会定位到引起该问题的源代码处。

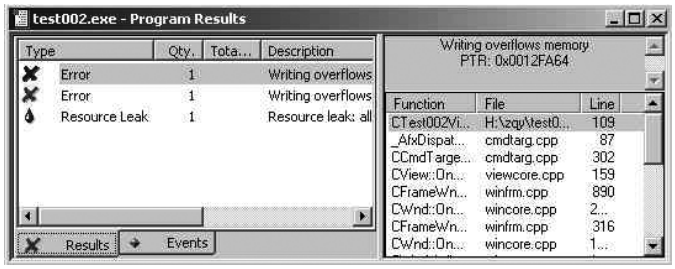

图 5.5 错误列表

虽然 BoundsChecker 给出了较为详尽的错误报告，但测试人员仍然需要发挥主观能动性进行分析。因为工具的使用只能帮助测试人员更快捷方便地检测，但无法保证其结果是完全正确的。在使用中 BoundsChecker 会存在误报的情况，其可能有以下两种情况：第一种是由于工具算法问题将正常的代码检测为错误；第二种是由于 BoundsChecker 指出的问题存在于第三方代码中，如第三方的程序库等。对于错误列表中的错误，测试人员应该进行仔细分析，在确认非程序出错时可将错误设为忽略，以防下次再提醒。

5.2 单元测试工具 JUnit

5.2.1 JUnit 简介

JUnit 是一个可编写重复测试的简单框架，是基于 XUnit 架构的单元测试框架实例。它由 Kent Beck 和 Erich Gamma 建立，并逐渐成为源于 Kent Beck，SUnit 的 XUnit 家族中最为成功的一个。

JUnit 测试主要用于程序员测试，即白盒测试，因为程序员知道被测试的软件如何完成功能和完成什么样的功能。JUnit 是一套框架，其继承 TestCase 类，现在大多数 Java 开发环境都已经集成了 JUnit 作为自带单元测试的工具。

5.2.2　JUnit 的优势与核心功能

1. JUnit 的优势

(1) 简化测试代码的编写，每个单元测试用例相对独立并由 JUnit 启动，自动调用，不需要添加额外的调用语句。

(2) 可以书写一系列的测试方法，对项目所有的接口或者方法进行单元测试。

(3) 添加、删除、屏蔽某条测试方法时不影响其他的测试方法，几乎所有开源框架都对 JUnit 有相应的支持。

(4) 能使测试单元保持持久性。

2. JUnit 的核心功能

(1) 测试用例(TestCase)：创建和执行测试用例。

(2) 断言(Assert)：自动校检测试结果。

(3) 测试结果(TestResult)：测试的执行结果。

(4) 测试运行器(Runner)：组织和执行测试。

5.2.3　根据血糖判断健康状况

本案例主要根据空腹血糖以及餐后两小时血糖联合进行健康判断，判断标准如表 2.1。

1. 创建测试类

首先创建被测类，部分核心代码如下：

```
public class BloodSugar {
    private double empty;
    private double later;

    public double getEmpty() {
        return empty;
    }

    public void setEmpty(double empty) {
        this.empty = empty;
    }

    public double getLater() {
        return later;
    }

    public void setLater(double later) {
        this.later = later;
```

```
    }

    public void setParams(double e,double l){
        this.empty=e;
        this.later=l;
    }
    public BloodSugar(double e,double l){
        this.empty=e;
        this.later=l;
    }
}
public String GetBloodSugarType() {
        String result = "";
            if (empty >7. 0 & later >11. 1) {
                result = "糖尿病";
            } else if (empty <6. 1 & later <7. 8) {
                result = "正常血糖";
            }else if (empty < 7. 0 & later <=11. 1 & later >=7. 8) {
                result = "糖耐量减低";
            } else if (empty >=6. 1 & empty <= 7. 0 & later <7. 8) {
                result = "空腹血糖受损";
            } else {
                result ="血糖值错误";
            }
        return result;
    }
```

编写完成后可以写几个简单的测试用例测试一下。

```
public static void main(String[] args){
    BloodSugar test=new BloodSugar(7.5,12);
    System.out.println(test.GetBloodSugarType());

    test.setParams(5,6);
    System.out.println(test.GetBloodSugarType());

    test.setParams(6,8);
    System.out.println(test.GetBloodSugarType());

    test.setParams(6.5,7.5);
    System.out.println(test.GetBloodSugarType());

    test.setParams(1,200);
    System.out.println(test.GetBloodSugarType());
}
```

单击运行后结果如图 5.6 所示。

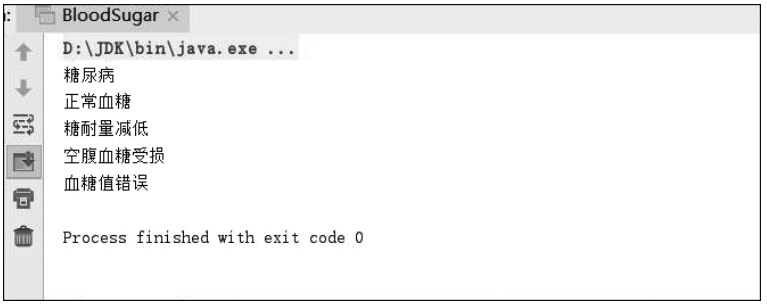

图 5.6　测试结果

可以看到，程序的功能已经基本实现，可以根据输入的空腹血糖值和餐后两小时血糖值判断人的健康情况。

在被测类代码处右击，在弹出的快捷菜单中选择 Go To→Test 命令，如图 5.7 所示，创建测试脚本。

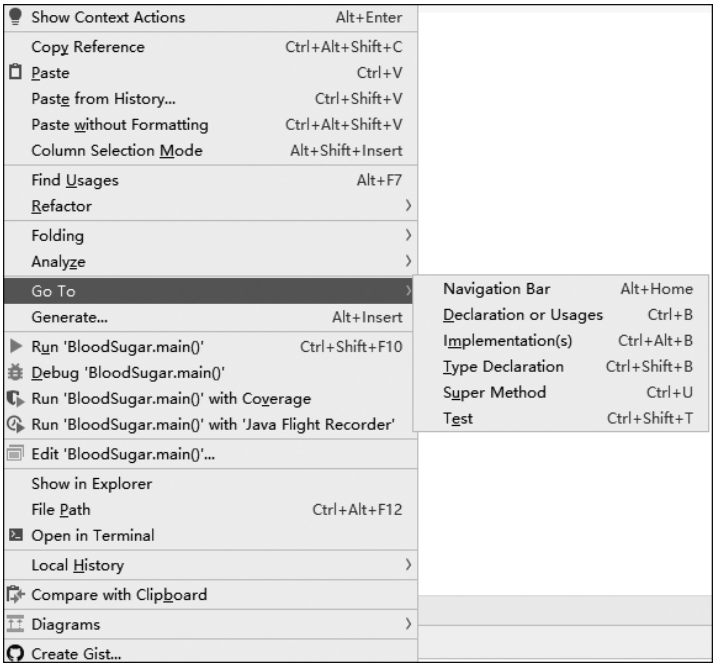

图 5.7　创建测试脚本

继续选择 Create New Test 命令，新建脚本，如图 5.8 所示。

图 5.8　新建脚本

选用 JUnit4 测试库进行测试，如图 5.9 所示，创建 JUnit 测试类。

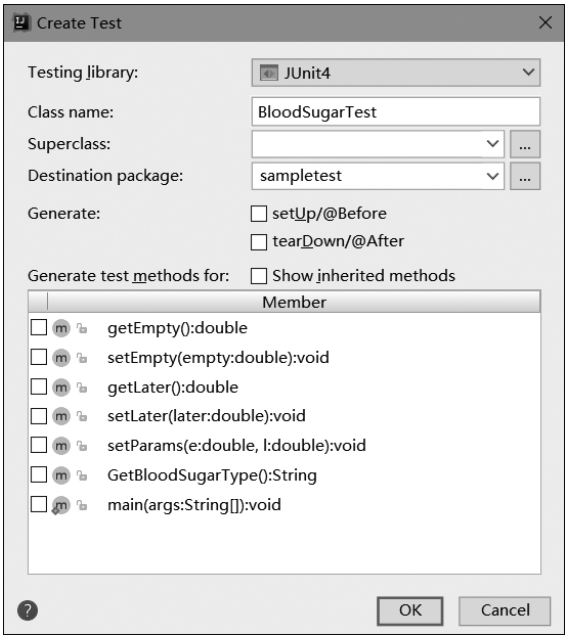

图 5.9　创建 JUnit 测试类

选中需要测试的类和方法后，即可自动生成 JUnit 测试类，部分代码如下所示：

```
@Test
public void getBloodSugarType() {
}
@Test
public void main() {
}
```

2. 编写测试方法

使用 JUnit 生成测试脚本，并编写部分代码如下：

```
public class BloodSugarTest {
    BloodSugar testObj;

    @Before
    public void setUp() throws Exception {
        testObj=new BloodSugar();
        System.out.println("Run @Before method");
    }
```

```
    @After
    public void tearDown() throws Exception {
        testObj = null;
        System.out.println("Run @After method");
    }

    @BeforeClass
    public static void prepareEnvironment(){
        System.out.println("Run @BeforeClass method");
    }

    @AfterClass
    public static void RestoreEnvironment(){
        System.out.println("Run @AfterClass method");
    }
}
```

测试用例的执行顺序如下。

(1) @Before：在每个测试用例执行之前执行一次。

(2) @After：在每个测试用例执行之后执行一次。

(3) @BeforeClass：在测试类的所有测试用例执行之前执行一次。

(4) @AfterClass：在测试类的所有测试用例执行之后执行一次。

继续编写测试用例，部分代码如下：

```
@Test
public void getBloodSugarType() {
    System.out.println("Run getBloodSugarType");
    testObj.setParams(6,7);
    String actual=testObj.GetBloodSugarType();
    String expect="正常血糖";
    assertTrue(expect==actual);
}
```

其中，GetBloodSugarType 的作用即调用被测方法。

assertTrue 即断言，其可以判断预期结果 expect 和运行结果 actual 的差别。

```
@Category({EPTest.class})
@Test
public void getBloodSugarType_1(){
    System.out.println("Run getBloodSugarType1");

    testObj.setParams(6,8);
    assertTrue(testObj.GetBloodSugarType()=="糖耐量减低");
```

```
    }
    @Category({EPTest.class})
    @Test
    public void getBloodSugarType_2(){
        System.out.println("Run getBloodSugarType2");

        testObj.setParams(6.5,7);
        assertTrue(testObj.GetBloodSugarType()=="空腹血糖受损");
    }
    @Category({EPTest.class})
    @Test
    public void getBloodSugarType_3(){
        System.out.println("Run getBloodSugarType3");

        testObj.setParams(7,11.1);
        assertTrue(testObj.GetBloodSugarType()=="糖尿病");
    }
```

这里用了@Category 注解，JUnit 提供了一种分类运行器，其包含在 org.junit.experimental.categories.Category 中，基本步骤分成两步。

（1）创建新的测试类，并配置该测试类。

（2）修改已有测试类，定义具有特定分类的方法。

采用这种分类器可以将测试用例分类，以满足不同的测试类型。

在应用到测试用例之前首先完成第一步，创建新的测试类，其代码如下。

```
    @RunWith(Categories.class)
    @Categories.IncludeCategory({EPTest.class})
    @Suite.SuiteClasses({BloodSugarTest.class})
    public class CategoryTest {
    }
```

@RunWith 注解指定运行器；IncludeCategory 设置要执行的测试特性为 EPTest，测试特性也可被理解为专有一类测试的标签；SuiteClasses 设置候选测试集为 BloodSugarTest。

第二步修改已有测试类，在@Category 注解后加上{EPTest.class}以表明该测试用例的标签。

注意在使用 EPTest 标签之前必须要先定义 EPTest 接口类，其代码如下。

```
    public interface EPTest {
    }
```

之后该代码即可执行，执行结果如图 5.10 所示。