Eve Testing

Testing Java Applications

Anton Liebetrau, iSolutions (VICS 3)
Version 1.02, December 2001

Translation/Editing:
George Raymond, Systor AG

Modifications

Version	Date	Description of the modification
1.02	Dec 2001	Small additions.
1.01	Nov 2001	Small corrections and additions.
1.00	Sep 2001	Small corrections and additions.
0.95	Sep 2001	Completion of chapters 1 and 3; distribution to various members of the VICS 11, VICS 3 and VINS 17 teams.
0.02	Jul 2001	Completion of chapter 2 (JUnit); distribution to various members of the VICS 3 and VINS 17 teams.

Contents

1. Introduction

It's fun to listen to software developers chatting. When they start talking about bugs (= simple software errors) or blunders (= bad software errors), the conversation can become particularly heated. Here is a conversation that we[1] (secretly) recorded during a crisis meeting and are please to relate to you in abridged form:[2]

Angela Angeli	*Manager, smoker, enraged.* – Who produced this blunder?
John Johnson	*Software developer, coffee drinker, easily agitated.* – You're asking the wrong question, Angie. The real question is: Who *didn't notice* this blunder?
Angela Angeli	*Still enraged.* – Didn't notice, didn't notice! What's that supposed to mean? Don't you test your software before giving it to people?
John Johnson	*Still agitated.* – Of course we do, Angie! But... Testing is so boring, so tedious, so tiring – it's no fun and makes for extra work, delays our releases and just completely...
Kent	*Software developer, consultant, gum chewer, finger waver.* – Heh heh ... Testing is fun and doesn't cost a drop of sweat. The extra effort quickly pays for itself. And testing doesn't delay software delivery – it speeds it up.
John Johnson	*With a dumb look on his face.* – Huh?
Martin	*Software developer, consultant, pencil chewer, pensive.* – If testing is so boring, then why not build a testing machine that you can just throw the tests into? What woman would wash clothes by hand today? – *He ducks a bit.* Sorry, Angie, I didn't mean you.
Angela Angeli	*With a sour look.* – At home my husband does the laundry. And as far as I can tell, it hasn't given him blisters yet.
Erich	*Software developer, consultant, surfboard enthusiast, suntanned.* – Exactly, if you don't want blisters on your hands, you need a machine. – *Excited.* – Kent, bring out our test machine!
Kent	*With his hand over his ears.* – Okay, Erich, okay, not so loud, please. – *Rummages in his pockets.* – Okay, well, you'll find our test machine on this diskette. It's called JUnit.

[1]By "we", I mean the "Eve Team" (= a few people from the iSolutions Team / VICS 3); this is not a "royal" use of the plural to designate the writer himself (*pluralismus majestatis*).

[2]For reasons of personal data protection, the names of all participants have been changed or at least greatly shortened.

Erich	*With a malicious grin.* – Indeed, it's a very special framework that's easy, I mean super-duper easy to use. Which is not usually the strong point of frameworks...
Kent	*Shrugs his shoulders regretfully.* – Uh, there's no documentation yet – sorry – but we'll deliver it, right Erich?
Erich	*With confidence.* – Yes, of course, we'll deliver it, for sure. – But first, we're building a documentation machine, and that could take awhile... Uh, Martin, have you maybe already written something about JUnit? You like to produce little buffer stocks, right, huh huh...
Martin	*Disapprovingly.* – That sunstroke of yours really worries me... You bet I've written something about JUnit – *Lays a book on the table.* – Here it is; it's entitled...
Erich	*Impatient.* – Please, it's not time for a commercial. We've all written wonderful books.
Kent	*Astonished.* – Hey Erich, you wrote the forward for Martin's book. – *Flips through the book.* – Looks like a good book – I'm always being quoted. – *Reads, laughs.* – Hoho, did I say that? Magnificent! Never knew I was so funny.
Martin	*Snatches the book back.* – Wait a minute. Don't get things mixed up. Read this here: I thought of all the funny parts; you thought of the ones that aren't so funny.
Erich	*Sarcastically.* – Look here, some parts have a gray background. Are they those appraisals you make, that everyone's so scared of?
Angela Angeli	*The lack of nicotine is causing discomfort.* – Look you guys, this meeting just started and I'm already exhausted. I need a break.
John Johnson	*Very interested, but unfortunately is starting to notice the five cups of coffee that constituted his breakfast:* – Listen, I've gotta go out for a minute... be right back.

These people are very nice, and it's interesting to listen to their daily problems, but it's about time we turned to the essential points of the discussion, which lasted some 320 minutes (of which you've just read the first three; we didn't have room for the rest):

- Applications are rarely without errors. To find disruptive errors before delivery, tests must be carried out (unfortunately, we know of no other method).
- Manual testing is boring and tiring (but you can do something about it).
- You can automate testing (and thus save everyone from boredom).
- Tests should be reproducible (so that, after the error has been corrected, you can check that it is really gone).
- Tests should be able to check their own results (and only bother us in case of an error).
- Tests increase software quality (massively).
- Tests reduce development time (even if managers and software engineers seldom believe it).

1.1 What we're talking about here…

In the present document, we will take a detailed look at the testing of Java applications. In practice, the following two kinds of tests are well-known. (See also [Beck2000], page 47 and [Fowler1999], pages 96–97):

- **Component tests** are written by the programmer. Such tests let you check whether individual program parts (classes, components) work as they should.
- **Functions tests** are written (or at least specified) by the customer. Such tests let you check whether the whole (finished) application works as it should.

In the present document, we will only talk about automated component tests, and recommend in this context the following open-source and freeware products:

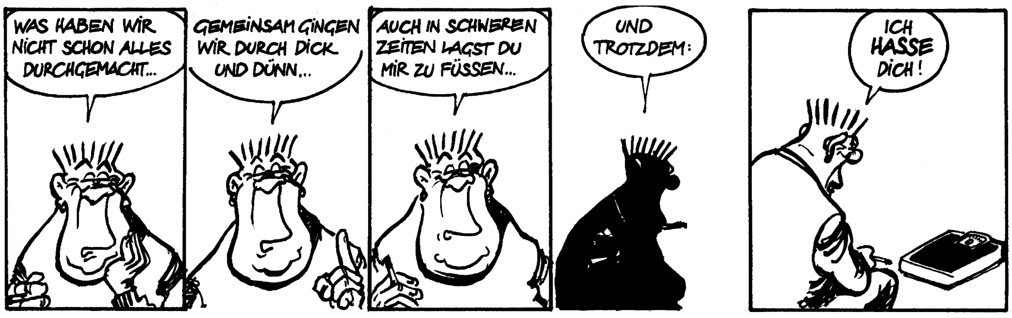
- **JUnit: Testing Java applications.** JUnit is a framework[3] with which you can write test classes. With these test classes, whenever you want, you can test whether Java classes are working properly. Martin Fowler has written about JUnit (see [Fowler1999], page 89 and following), and in the JUnit web site, Kent Beck and Erich Gamma offer a cookbook (see [JUnit]).
- **HttpUnit: Testing web applications.** HttpUnit is an API[4] that simulates a web browser. Using it, you can call dynamic web applications (applications based on servlets, for example). Together within JUnit, HttpUnit is a useful tool for testing dynamic web applications.

[3] A framework specifies a software architecture for a certain task. The framework usually does this by offering classes and relationships that developers then make more specialized for use in applications. (The framework thus offers semi-finished building blocks).

[4] API stands for *application programming interface* and normally means an interface that you can use to call classes, services, or applications from outside. An API can also mean a collection of pre-defined classes that carry out a specific task and that you can immediately put into use. Well-known standard Java APIs include JDBC, Java 2D Graphics, I/O (Streams), and Logging (in Java 1.4 and higher).

1.2 Why no one likes to test...

Test tools are seldom liked, much less loved. The main reason is that they are incorruptible, heartless, and honest. They slap us on the fingers when we make mistakes, prevent us from fudging results, and force us to produce quality. Eva, well-known as an overweight, crabby, stubborn, but nevertheless good-hearted mother and department-store employee, uses her test tool daily (albeit without success):


WAS HABEN WIR NICHT SCHON ALLES DURCHGEMACHT...
GEMEINSAM GINGEN WIR DURCH DICK UND DÜNN...
AUCH IN SCHWEREN ZEITEN LAGST DU MIR ZU FÜSSEN...
UND TROTZDEM:
ICH HASSE DICH!

What the two of us haven't been through...	Together, we've been through thick and thin...	Even in hard times, you've laid at my feet...	And still:	I **hate** you!

Figure 1: Eva talking with her test tool (Jaermann/Schaad in the Zurich *Tages-Anzeiger* newspaper of April 17, 2001).

2. JUnit: Testing Java applications

The Java framework JUnit was created by Kent Beck[5] and Erich Gamma[6]. Its purpose is to automate the testing of Java classes and applications. To do this, JUnit makes available a set of classes that help you write automated component tests.

A JUnit test consists of various test methods (= test routines). During the implementation phase of a project, the software developer writes and continuously extends a JUnit test. (The writing of the component-test thus occurs in parallel with code development itself.) The resulting test code is later executed at regular intervals and helps catch the errors that always try to slip in during program modifications and extensions.

In the ideal case, each class has an (independent) test-case class (= a subclass of *TestCase*), and each method its own test method in which several tests can be carried out.[7] Several related test cases are bundled together into a package called a test suite (= a subclass of *TestSuite*).

[5]Kent Beck is a recognized OO expert and has published various books (see [Beck2000]).

[6]Erich Gamma has made a name for himself as a specialist in design patterns (see [Gamma1996]).

[7]Practice has shown that for some classes (for example, servlet classes), a JUnit test isn't appropriate. A special API is, however, available for this purpose. Its name is HttpUnit and we will also describe it in this document.

The following UML class diagram illustrates this idea:

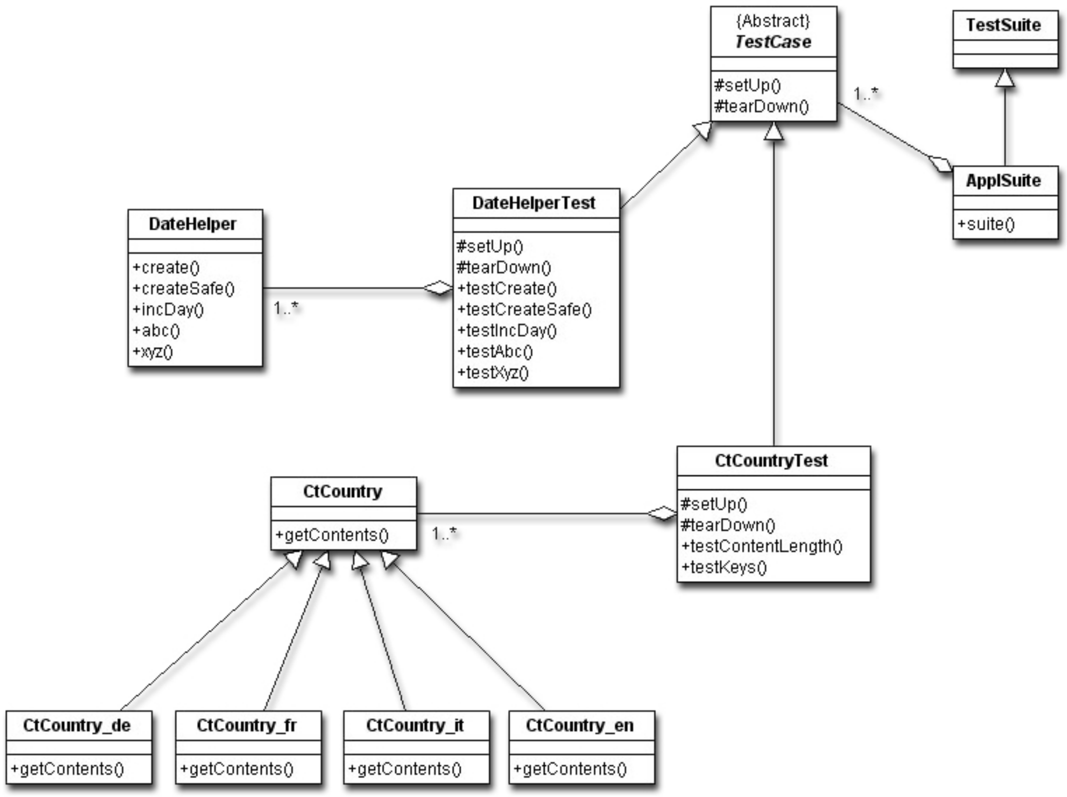


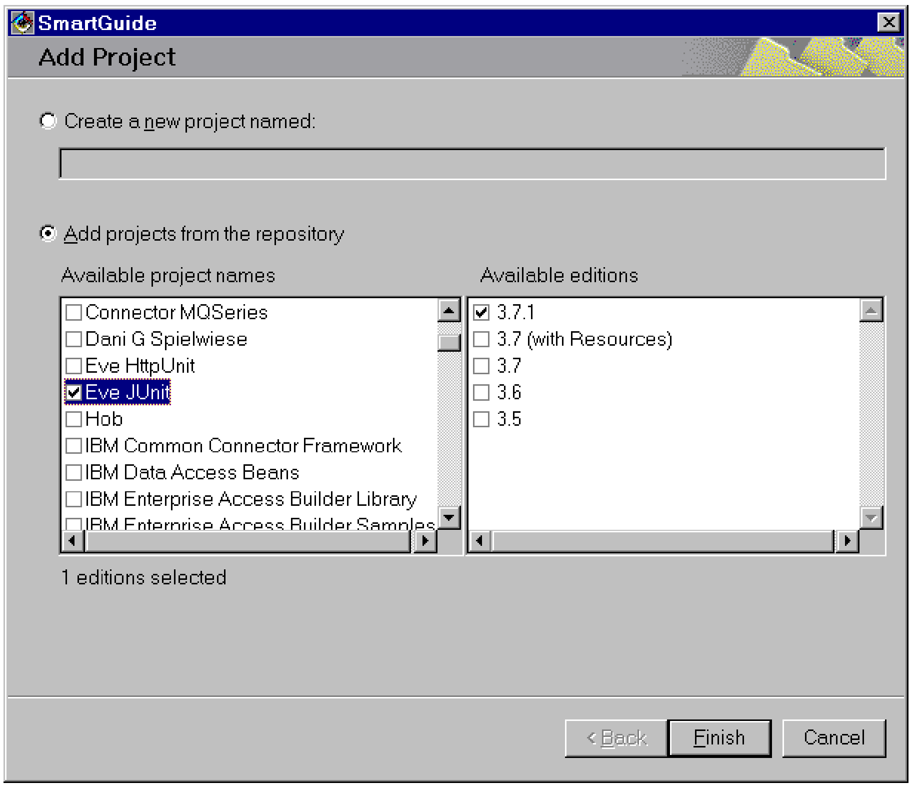
Figure 2: For the classes *DateHelper* and *CtCountry*, there are two test-case classes (*DateHelperTest* and *CtCountryTest*); they belong to the test suite *ApplSuite*.

The JUnit framework offers various classes. To write tests, however, the classes *TestCase* and *TestSuite* are all you need. The following table offers additional, short descriptions of the terms *test suite, test case,* and *test method*:

Element	**Subclass of**	**Identified with the**	**Contents**	**In Figure 2, see**
test suite	TestSuite	suffix *Suite*	several test cases	the class *ApplSuite*
test case	TestCase	suffix *Text*	several test methods	the classes *DateHelperTest* and *CtCountryTest*
test method		prefix *test*	code with which you can test certain methods	the methods *testCreate, testIncDay,* and *testContentLength*

2.1 Making JUnit available

Before you can start writing test classes, you must load either the project "Jackpot JUnit" or the project "Eve JUnit" in your own (local) workspace. (A reminder: within the VisualAge workbench, you do so by way of the menu items Projects > Add > Project...):



Create a new project named: ( )
Add projects from the repository (●)

Available project names:
- ☐ Connector MQSeries
- ☐ Dani G Spielwiese
- ☐ Eve HttpUnit
- ☑ Eve JUnit
- ☐ Hob
- ☐ IBM Common Connector Framework
- ☐ IBM Data Access Beans
- ☐ IBM Enterprise Access Builder Library
- ☐ IBM Enterprise Access Builder Samples

Available editions:
- ☑ 3.7.1
- ☐ 3.7 (with Resources)
- ☐ 3.7
- ☐ 3.6
- ☐ 3.5

1 editions selected

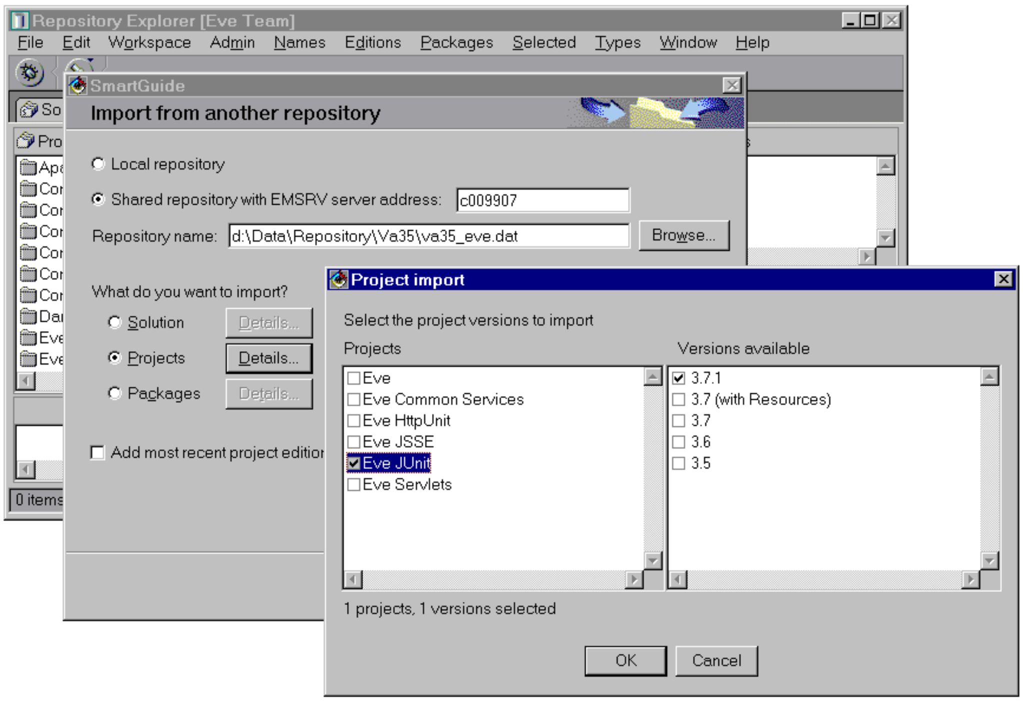
Figure 3: As soon as you select the field "Add projects from the repository" in this dialog, all projects appear that you can load into your (local) workspace.

Please note that you *cannot* simultaneously load both "Jackpot JUnit" and "Eve JUnit" into your workspace. These two JUnit projects differ as follows:

- **Eve JUnit** contains all the classes and interfaces of JUnit, as presented in http://www.junit.org (and nothing else).
- **Jackpot JUnit** contains, in addition to the JUnit classes and interfaces, the extensions *HttpUnit* and *JUnitPerf*. (At iSolution you'll find HttpUnit in the project *Eve HttpUnit*. *Unit-Perf*[8] is only available in *Jackpot JUnit*.**)**

[8] See also http://www.clarkware.com/software/JUnitPerf.html.

If neither of these two JUnit projects is in the current repository, you must import JUnit from another repository. (As of this writing, the official Eve repository is *va35_eve.dat* .) Within the Repository Explorer, you do this with the menu items File > Import...:


Repository Explorer [Eve Team]
File Edit Workspace Admin Names Editions Packages Selected Types Window Help
SmartGuide
Import from another repository
Browse...
Details...
OK
Cancel

Local repository
Shared repository with EMSRV server address: c009907
Repository name: d:\Data\Repository\Va35\va35_eve.dat
What do you want to import?
Solution
Projects
Packages
Add most recent project edition

Project import
Select the project versions to import

Projects	Versions available
☐ Eve	☑ 3.7.1
☐ Eve Common Services	☐ 3.7 (with Resources)
☐ Eve HttpUnit	☐ 3.7
☐ Eve JSSE	☐ 3.6
☑ Eve JUnit	☐ 3.5
☐ Eve Servlets	

1 projects, 1 versions selected

Figure 4: In the *Repository Explorer*, using the menu items File > Import..., you can import the contents of any repository. Here, the project *Eve JUnit 3.7.1* of repository *va35-eve.dat* is being loaded.

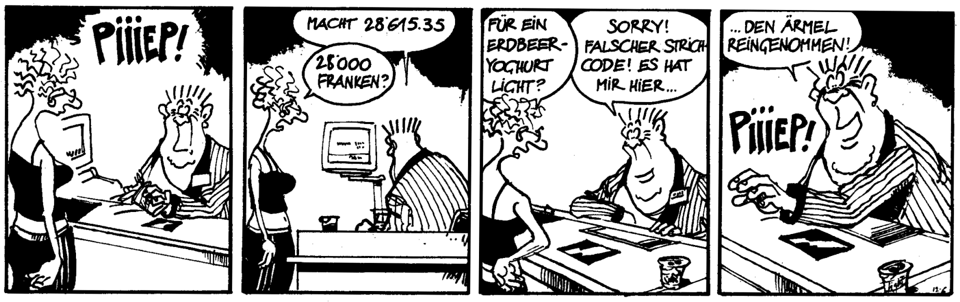
Important: After importing JUnit into the current Repository, you must also add this project to your workspace (see page 10).

2.2 Procedure for testing

When carrying out a component test, always proceed as follows:

1. Call the method that you want to test (with all needed input parameters).
2. On the basis of the input parameters, the called method returns a result.
3. Compare the returned result with what you expected.
4. If the returned result differs from the expected one, an error is probably present.
5. Look for and correct the error.
6. Repeat steps 1–5 until no more errors appear.

The following episode illustrates this procedure:


PIIIEP!
MACHT 28'615.35
28'000 FRANKEN?
FÜR EIN ERDBEER-YOGHURT LICHT?
SORRY! FALSCHER STRICH CODE! ES HAT MIR HIER...
... DEN ÄRMEL REINGENOMMEN!
PIIIEP!

That makes 28,615.35.

28,000 francs?

For strawberry yogurt light?

Sorry! Wrong bar code! The thing scanned...

...my sleeve!

Figure 5: Eva at work. – 1. Call to the method being tested (input parameter: strawberry yogurt). – 2. On the basis of the input parameters, the called method returns a result (CHF 28,615.35). – 3. Comparison of the returned and expected results. – 4. An error is present. – 5. Error localization and correction (roll up sleeve). – 6. Repeat the process (not shown here). (Jaermann/Schaad in the Zurich *Tages-Anzeiger* of June 12, 2001).

2.3 Test classes and test methods

As we've already mentioned, to carry out a test, you as a software developer need to check – with the help of test classes – whether certain classes really behave as you want. A test class contains several test methods, which in turn contain the code for the desired tests.

Concerning test classes and test methods, please note the following (see also **Figure 6**).

- A test class is a subclass of *TestCase*; append the suffix *Test* to it (for example *DateHelperTest* or *CtCountryTest*).
- For each class you want to test, define a specific test class. (For example, make *DateHelperTest* the test class for *DateHelper*.)
- To test how several classes work together, use specific test classes. (For example, *ProductAndPersonTest* would test the interplay of *Product* and *Person*.)
- One test class can contain any number of test methods.
- Ideally, you should give a test method the prefix *test* (as in *testIncMonth* or *testCreateSafe*). In this way, the JUnit framework recognizes the test methods automatically.
- You can give a test method either the same name as the original method you are testing (adding the prefix "test") or any other meaningful name. (See Figure 6.)

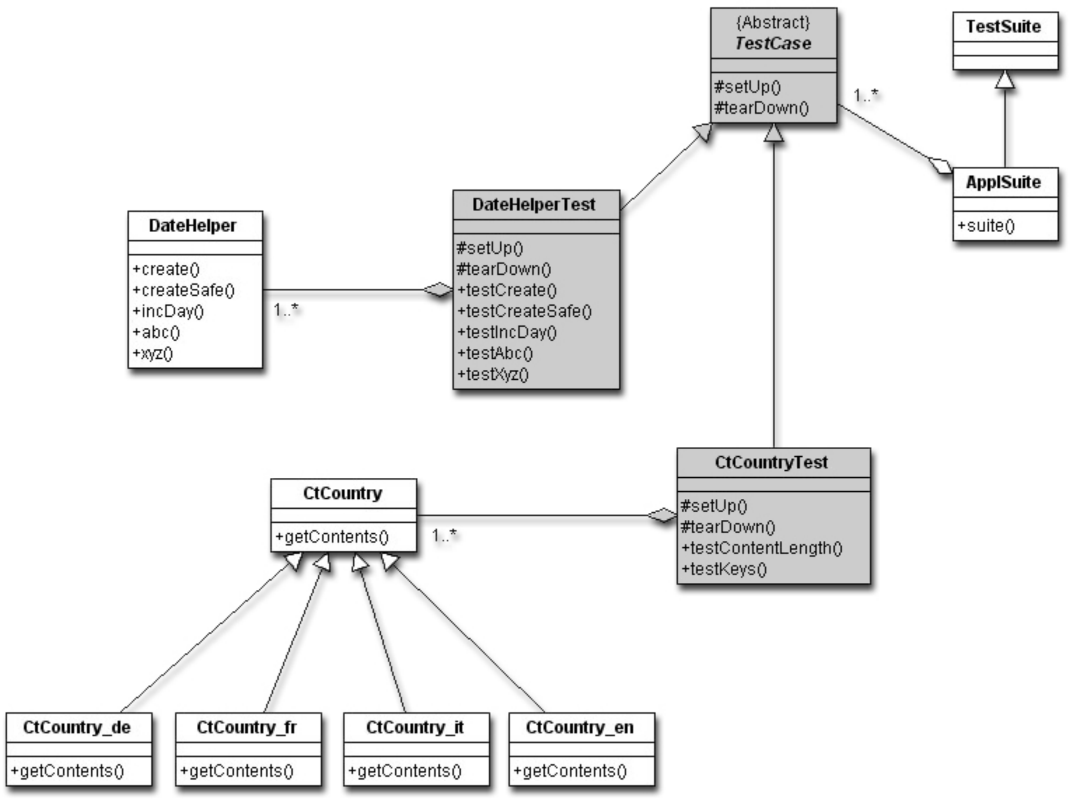


Figure 6: The gray-shaded test classes *DateHelperTest* and *CtCountryTest* permit checks of the classes *DateHelper* and *CtCountry* (and *CtCountry*'s four subclasses). In the case of *DateHelperTest*, the test methods bear the same name as the methods of *DateHelper* (plus the prefix *test*). In *CtCountryTest*, on the other hand, the names of the test look almost nothing like those of the original methods. (The methods *testContentLength* and *testKeys* check the original method *getContents*; for every original method, you can write as many test methods as you need.)

2.3.1 Packages for test classe s

Assign the test classes directly to the project you are testing, in the package named *unittest*.[9] (If needed, you can create additional, subpackages within *unittest*.) Here are few examples:

```
com.winterthur.iqskernel.unittest
com.winterthur.iqskernel.unittest.util
com.winterthur.iqskernel.unittest.resources
```

2.3.2 Creating a test class

As we already mentioned on page 12, you derive a test class from the class *TestCase;* the test class can look like this:

```
import junit.framework.TestCase;
```

[9] Each department can of course itself decide on the package name for the JUnit classes. Here at iSolutions, we call it *unittest*.

```
import com.winterthur.iqskernel.util.DateHelper;  // class being tested

/**
 * JUnit test class with which you can test "DateHelper".
 * For each method in "DateHelper", a test method (starting with "test") is
 * present
 *
 * Creation date: 12 Mar 2001
 * @author: John Johnson
 */
public class DateHelperTest extends TestCase {
  ...
}
```

2.3.3 Writing the constructor for the test case

Each test class has its own constructor. You can use the constructor to carry out one-time initializations:

```
public class DateHelperTest extends TestCase {
  ...

  public DateHelperTest(String name) {
    super(name);

    //-- Carry out one-time initializations
    ...
  }
}
```

2.3.4 Writing test methods

A test method returns no reply (= *void*) and bears the prefix *test*. Here's an example:

```
/**
 * This test method checks the method "DateHelper.decDay(...)".
 *
 * Creation date: 20 Mar 2001, John Johnson
 */
public void testDecDay() {
  Date dtAct;  // current date
  Date dtExp;  // expected date

  //-- Test: Start of year
  dtAct = DateHelper.createSafe(2001, 1, 1);
  dtAct = DateHelper.decDay(dtAct, 1);
  dtExp = DateHelper.createSafe(2000, 12, 31);
  assertEquals(dtExp, dtAct);

  //-- Test: start of March (during a leap year)
  dtAct = DateHelper.createSafe(2000, 3, 1);
  dtAct = DateHelper.decDay(dtAct, 1);
  dtExp = DateHelper.createSafe(2000, 2, 29);
  assertEquals(dtExp, dtAct);
}
```

2.3.5 *Assert* methods

To check (test) results, you should use so-called *assert* methods. The test class *DateHelperTest* would, for example, inherit these from the class *TestCase*.

The following table presents the available *assert* methods.

***Assert* method**	**Reports an error if this condition is not true:**	**Comments**
assertTrue([String message,] boolean condition)[10]	*condition* is true	
assertEquals([String message,] "simple type" expected, "simple type" actual)	the *expected* and *actual* parameters both contain the same value	By "simple types", we mean here: *boolean, char, byte, short, int* and *long.*
assertEquals([String message,] double expected, double actual, double delta)	the *expected* and *actual* parameters are equal (but a different of *delta* is permissible)	The same method exists for *float*, but we do not recommend you use this data type. See also Section *7.3, Prefixes for variables* in [EveProg2001].
assertEquals([String message,] Object expected, Object actual)	the *expected* and *actual* parameters are equal	Warning: This method uses the instance method *equals(...)* to check equality (like this: *expected.equals(actual)*). For many standard classes (for example, *String* or *Date*, but not *StringBuffer*) this method is already implemented, but you must write it yourself for your own classes.[11]
assertNotNull([String message,] Object object)	*object* does not equal null	
assertNull([String message,] Object object)	*object* equals null	
assertSame([String message,] Object expected, Object actual)	the *expected* and *actual* parameters both point to the same object (looks like this: *expected == actual*).	

[10] In version 3.4 of JUnit, the former method *assert(...)* was renamed *assertTrue(...).*

[11] In the Java literature, you'll often find statements saying that in addition to writing *equals(...)* you should also implement the method *hashCode()*. You mainly need *hashCode()* when inserting objects into hash tables (class *Hashtable*). For further information, see the source code of the methods *Object.equals(...)* and *Object.hashCode()*.

In these *assert* methods, the first parameter, *message*, is optional. You can use *message* to make the error report somewhat more specific. A reminder: The error report only appears in case of an error, and only when the assertion made by the asset message *is not* true. The following lines show some ways you can use assert methods:

```
//-- Test: Change to next day's date
Date dtAct1 = DateHelper.createSafe(2000, 10, 14);
Date dtAct2 = DateHelper.createSafe(2000, 10, 15);
assertTrue(!DateHelper.isAfter(dtAct1, dtAct2));
assertTrue(DateHelper.isAfter(dtAct2, dtAct1));

//-- Test: All days in January 2001
for (int i=-10; i<40; i++) {
  assertEquals(i>=1 && i<=31, DateHelper.isValid(2001, 1, i));
}

//-- Test: End of February (during a leap year)
Date dtAct = DateHelper.createSafe(2000, 2, 29);
assertEquals(20000229, DateHelper.date2Int(dtAct));

//-- Determine locales
Locale[] arLoc = needLocales();
assertNotNull("<needLocales>: Invalid return value.", arLoc);
```

2.3.6 *Fail* methods

In some situations, you need to provoke an error report. You can, of course, easily accomplish this with an *assert* method, for example, with:

assertTrue("A crash has occurred...", false)

We don't recommend this, however, because it's so hard to read. With the help of the following *fail* methods, you can force the occurrence of any error report you want.

***Fail* method**	**Always triggers an error report. Additional response:**
fail([String message])	[none]
failNotEquals(String message, Object expected, Object actual)	States that *expected* does not equal *actual* (in other words, that the statement *expected.equals(actual)* is not true).*
failNotSame(String message, Object expected, Object actual)	States that the two object instances *expected* and *actual* are not identical (in other words, the statement *expected == actual* is not true).*

*Important: Despite what their names might make you lead you to believe, these methods do not carry out comparisons.

Here as well, you can use the (sometimes optional) parameter *message* to make the error report somewhat more specific. The following lines show some ways you can use *fail* methods:

```
//-- Check whether a resource contains a key
try {
  stValue = actResource.getString(stKey);
} catch (Exception e) {
  fail("<" + actResource +"> -- Error concerning key: " + stKey);
}
```

```
//-- Transform a string into a double
double dVal = 0.0;
try {
  dVal = Double.parseDouble("Huhu");
  fail("No exception thrown.");
} catch (NumberFormatException e) {
  //-- very good, this exception is expected
} catch (Exception e) {
  fail("Unexpected exception thrown.");
}
```

2.3.7 Exceptions in test metho ds

When the test method throws an *exception* that a *try-catch block* doesn't catch, the JUnit-Framework recognizes the exception automatically and displays it as an error.[12] (See also the description of the *error* field in section *2.4.3, Carrying out the test methods of a test class*).

2.3.8 Initialization and cleanu p

With the help of the methods *setUp()* and *tearDown()*, which have no parameters, you can carry out certain initialization or cleanup tasks. You can use these methods either before or after executing test methods. In practice, you will often need *setUp()* but seldom *tearDown()*, since the garbage collector of Java takes care of most work for you (to close open files, you do need *tearDown()*, however).

Important: JUnit runs *setUp()* before and *tearDown()* after each execution of a test method, if these methods are present. (Thus, if a test class has 20 methods, you must call *setUp()* and *tearDown()* 20 times each.) This allows you to execute the various test methods independently from each other and in any order.

```
public class DateHelperTest extends TestCase {
  private Date mdtAct;
  private Date mdtEnd2001;
  private Date mdtStart2001;

  protected void setUp() {  // is called twice in this example
    System.out.println("-> setUp: " + getName());       // Display name of test
                                                        //method

    //-- Carry out initialization
    mdtStart2001 = DateHelper.createSafe(2001, 1, 1);
    mdtEnd2001 = DateHelper.createSafe(2001, 12, 31);
  }

  protected void tearDown() {  // is called twice in this example
    System.out.println("<- tearDown: " + getName());    // Display name of test
                                                        // method

    //-- Carry out cleanup
    ...
  }

  public void testCreateSafe() {
    //-- Test: Overshoot of December 2001
    mdtAct = DateHelper.createSafe(2001, 13, 77);
```

[12] In effect, all *fail* methods and unsuccessfull *assert* methods do nothing other than throw an exception named *AssertionFailedError*.

```
      assertEquals(mdtEnd2001, mdtAct);

      //-- Test: Undershoot of January 2001
      mdtAct = DateHelper.createSafe(2001, -3, -7);
      assertEquals(mdtStart2001, mdtAct);
    }

    public void testIncDay() {
      //-- Test: Overshoot of year's end
      mdtAct = DateHelper.createSafe(2000, 12, 31);
      mdtAct = DateHelper.incDay(mdtAct, 1);
      assertEquals(mdtStart2001, mdtAct);
    }
  }
```

As you can see, the *setUp()* and *tearDown()* methods are called twice: before and after *test-CreateSafe()* and *testIncDay()*.

With the help of the method *getName()*[13], which inherits from *TestCase*, you can always determine which test method is currently running. (If you need to, you can call the method *getName()* in *setUp()* and *tearDown()*.)

The following lines show the code of the method *CtCountryTest.setUp()*. This code uses the *fail* and *getName* methods to describe an error when it occurs:

```
  public class CtCountryTest extends TestCase {
    ...

    protected void setUp() {
      try {
        marRes[LANG_DE] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
          new Locale("de", "DE"));
        marRes[LANG_FR] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
          new Locale("fr", "FR"));
        marRes[LANG_IT] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
          new Locale("it", "IT"));
        marRes[LANG_EN] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
          new Locale("en", "GB"));
      } catch (Exception e) {
        fail("Error in <setUp> for <" + getName() + ">");
      }
    }
  }
```

A reminder: A test class does not necessarily need a *try-catch* block that catches *exceptions* (see section *2.3.7*, *Exceptions in test methods*). The *setUp* method above could also look as follows:

```
  protected void setUp() {
    marRes[LANG_DE] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
      new Locale("de", "DE"));
    marRes[LANG_FR] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
      new Locale("fr", "FR"));
    marRes[LANG_IT] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
      new Locale("it", "IT"));
    marRes[LANG_EN] = (ListBundleBase)ListBundleBase.getBundle(CLASS_TO_TEST,
      new Locale("en", "GB"));
  }
```

[13] JUnit version 3.7 renamed the method *getName()*. (It used to be *name()*).

2.4 Carrying out tests

To carry out tests, you need the application *TestRunner*. This application is available in three variants:

- *Text:* A rudimentary application without a graphical user interface that isn't much fun and that we won't discuss further here (class *junit.textui.TestRunner*).
- *AWT:* A practical testing application with a graphical AWT user interface that works with Java 1.1.x and higher (class *junit.awtui.TestRunner*).
- *Swing:* A comfortable testing application with a graphical Swing user interface that at iSolution only works flawlessly with Java 1.2 and higher (class *junit.swingui.TestRunner*).

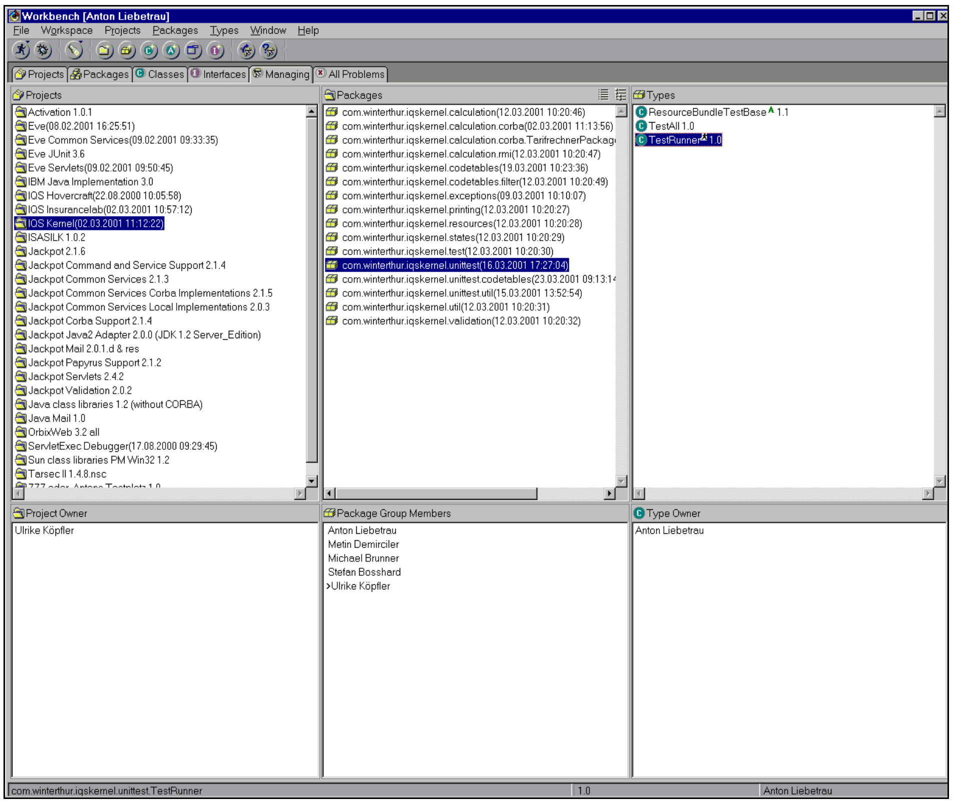
As is well-known, you cannot executive a Java application successfully unless you first correctly set the class path (see section *2.4.2, Setting the class path for the general TestRunner class*).

Given that the projects *Jackpot JUnit* and *Eve JUnit* are present in versioned form, you can't modify the class path in V*isualAge.* This is only possible in *open editions*. This has an unpleasant side effect: although you can run the application *TestRunner*, you can't load it with the test classes you have written (and thus *cannot* carry out the tests).

So what's the solution? This (simple) problem is very easy to solve. The following sections describe it in more detail.

2.4.1 Creating a general TestRunner class

An easy way to carry out the JUnit tests within *VisualAge* is to create your own *TestRunner* class in the project you want to test (store this class in the package *...unittest*).[14] An example:


Workbench [Anton Liebetrau]
File Workspace Projects Packages Types Window Help
Projects Packages Classes Interfaces Managing All Problems

Projects

Activation 1.0.1
Eve(08.02.2001 16:25:51)
Eve Common Services(09.02.2001 09:33:35)
Eve JUnit 3.6
Eve Servlets(09.02.2001 09:50:45)
IBM Java Implementation 3.0
IQS Hovercraft(22.08.2000 10:05:58)
IQS Insurancelab(02.03.2001 10:57:12)
IQS Kernel(02.03.2001 11:12:22)
ISASILK 1.0.2
Jackpot 2.1.6
Jackpot Command and Service Support 2.1.4
Jackpot Common Services 2.1.3
Jackpot Common Services Corba Implementations 2.1.5
Jackpot Common Services Local Implementations 2.0.3
Jackpot Corba Support 2.1.4
Jackpot Java2 Adapter 2.0.0 (JDK 1.2 Server_Edition)
Jackpot Mail 2.0.1.d & res
Jackpot Papyrus Support 2.1.2
Jackpot Servlets 2.4.2
Jackpot Validation 2.0.2
Java class libraries 1.2 (without CORBA)
Java Mail 1.0
OrbixWeb 3.2 all
ServletExec Debugger(17.08.2000 09:29:45)
Sun class libraries PM Win32 1.2
Tarsec II 1.4.8.nsc

Packages

com.winterthur.iqskernel.calculation(12.03.2001 10:20:46)
com.winterthur.iqskernel.calculation.corba(02.03.2001 11:13:56)
com.winterthur.iqskernel.calculation.corba.TarifrechnerPackag
com.winterthur.iqskernel.calculation.rmi(12.03.2001 10:20:47)
com.winterthur.iqskernel.codetables(19.03.2001 10:23:36)
com.winterthur.iqskernel.codetables.filter(12.03.2001 10:20:49)
com.winterthur.iqskernel.exceptions(09.03.2001 10:10:07)
com.winterthur.iqskernel.printing(12.03.2001 10:20:27)
com.winterthur.iqskernel.resources(12.03.2001 10:20:28)
com.winterthur.iqskernel.states(12.03.2001 10:20:29)
com.winterthur.iqskernel.test(12.03.2001 10:20:30)
com.winterthur.iqskernel.unittest(16.03.2001 17:27:04)
com.winterthur.iqskernel.unittest.codetables(23.03.2001 09:13:1
com.winterthur.iqskernel.unittest.util(15.03.2001 13:52:54)
com.winterthur.iqskernel.util(12.03.2001 10:20:31)
com.winterthur.iqskernel.validation(12.03.2001 10:20:32)

Types

ResourceBundleTestBase 1.1
TestAll 1.0
TestRunner 1.0

Project Owner

Ulrike Köpfler

Package Group Members

Anton Liebetrau
Metin Demirciler
Michael Brunner
Stefan Bosshard
>Ulrike Köpfler

Type Owner

Anton Liebetrau

com.winterthur.iqskernel.unittest.TestRunner 1.0 Anton Liebetrau

Figure 7: In the package *...unittest*, the project *IQS Kernel* has its own *TestRunner* class.

You can create the *TestRunner* class in either of two ways:

```
//-- First option: TestRunner with AWT-GUI.
public class TestRunner extends junit.awtui.TestRunner {
}

//-- Second option: TestRunner with Swing-GUI
public class TestRunner extends junit.swingui.TestRunner {
}
```

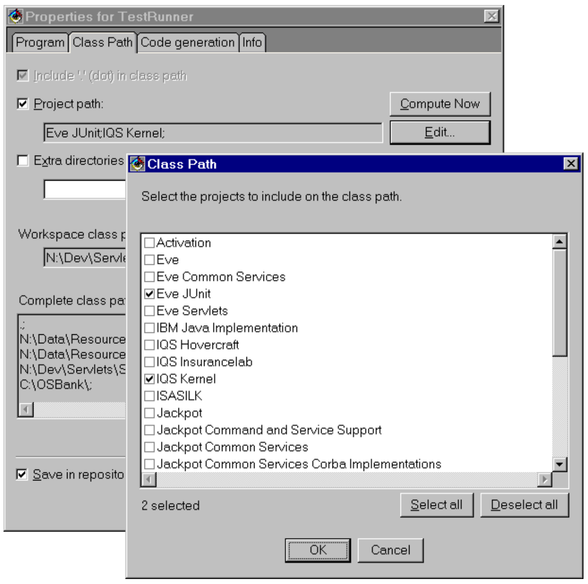
These *TestRunner* classes inherit a *main* method from their superclasses and you can thus start them as applications.

Another variant: Section *2.4.5, Giving a test class its own TestRunner*, describes how any test class can call a *TestRunner* on its own. (In this variant, you don't need a general *TestRunner*.)

[14] In other words: Each project has its own TestRunner class.

2.4.2 Setting the class path fo r the general TestRunner class

Before starting the *TestRunner* class for the first time, you must set the class path correctly. As is well-known, within *VisualAge* you do this in the Properties dialog of the class. (Call the dialog in **Figure 8** via the menu item *Properties* of the class context menu.)


Properties for TestRunner

Program | Class Path | Code generation | Info

☑ Include '.' (dot) in class path

☑ Project path: [Compute Now]

Eve JUnit;IQS Kernel; [Edit...]

☐ Extra directories

Workspace class p

N:\Dev\Servle

Complete class pa

.;
N:\Data\Resource
N:\Data\Resource
N:\Dev\Servlets\S
C:\OSBank\;

☑ Save in reposito

Class Path

Select the projects to include on the class path.

- ☐ Activation
- ☐ Eve
- ☐ Eve Common Services
- ☑ Eve JUnit
- ☐ Eve Servlets
- ☐ IBM Java Implementation
- ☐ IQS Hovercraft
- ☐ IQS Insurancelab
- ☑ IQS Kernel
- ☐ ISASILK
- ☐ Jackpot
- ☐ Jackpot Command and Service Support
- ☐ Jackpot Common Services
- ☐ Jackpot Common Services Corba Implementations

2 selected [Select all] [Deselect all]

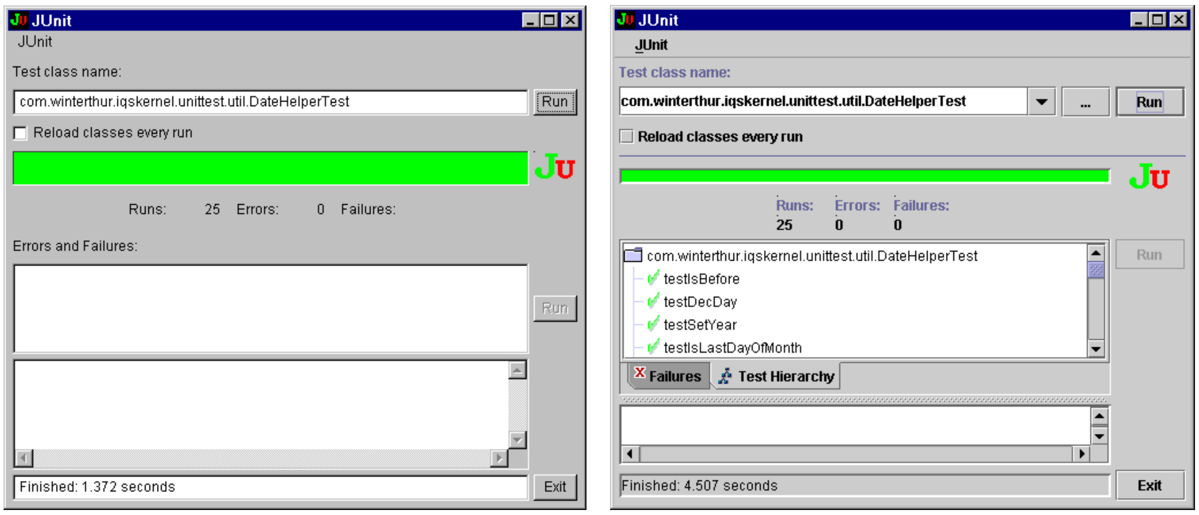
[OK] [Cancel]

Figure 8: In the Properities dialog of the *TestRunner* class, you must set the class path properly (do so with the button *Edit...* in the *Class Path* tab).

You can't, of course, start the newly-created *TestRunner* class until either the project *Jackpot JUnit* or the project *Eve JUnit* is in the class path.

2.4.3 Carrying out the test me thods of a test class

After running a test class successfully, the two *TestRunner* applications look like this:


JUnit

JUnit

Test class name:

com.winterthur.iqskernel.unittest.util.DateHelperTest [Run]

☐ Reload classes every run

Runs: 25 Errors: 0 Failures:

Errors and Failures:

Finished: 1.372 seconds [Exit]

JUnit

JUnit

Test class name:

com.winterthur.iqskernel.unittest.util.DateHelperTest [...] [Run]

☐ Reload classes every run

Runs: 25 Errors: 0 Failures: 0

com.winterthur.iqskernel.unittest.util.DateHelperTest
- ✓ testIsBefore
- ✓ testDecDay
- ✓ testSetYear
- ✓ testIsLastDayOfMonth

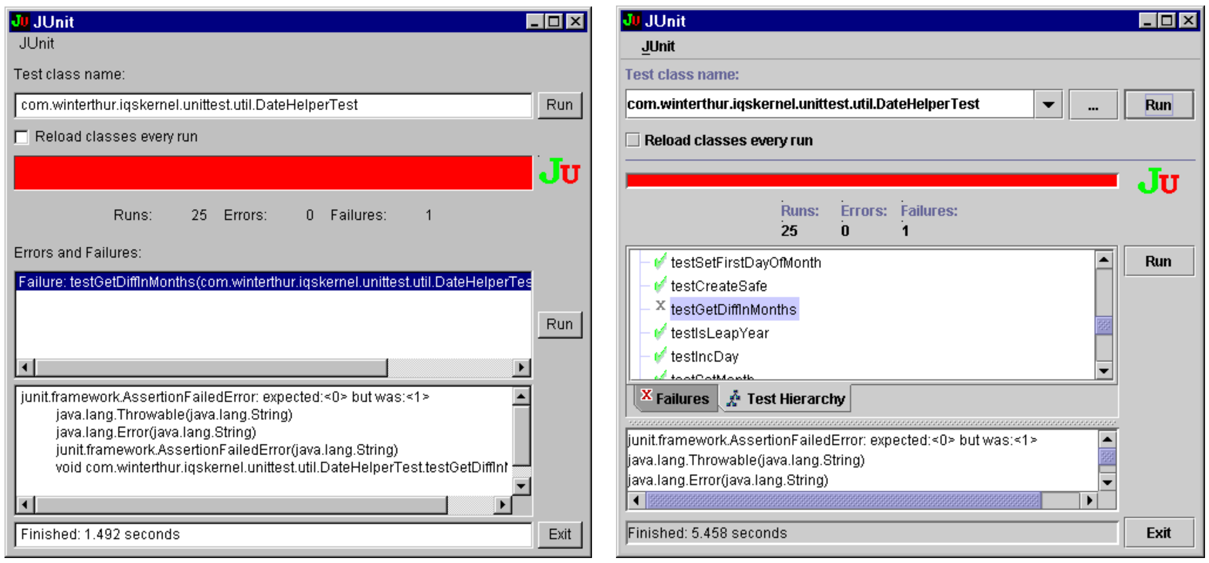
Failures | Test Hierarchy

Finished: 4.507 seconds [Exit]

Figure 9: The *TestRunner* in its AWT (left) and Swing versions after successfully executing all test methods.

In the field *Test class name*, enter by hand the name of the class to be checked (including its package name), then press Run. TestRunner automatically identifies and runs all the test methods (= methods with the prefix *test*) of the test class. You can use the second (lower) button *Run* to execute individual test methods separately.

If *all* test methods execute successfully, the progress bar is **green** (see Figure 9), otherwise it is **red**:



```
JUnit
JUnit
Test class name:
com.winterthur.iqskernel.unittest.util.DateHelperTest        Run
Reload classes every run

         Runs:   25  Errors:   0  Failures:   1

Errors and Failures:
Failure: testGetDiffInMonths(com.winterthur.iqskernel.unittest.util.DateHelperTes
                                                              Run
junit.framework.AssertionFailedError: expected:<0> but was:<1>
       java.lang.Throwable(java.lang.String)
       java.lang.Error(java.lang.String)
       junit.framework.AssertionFailedError(java.lang.String)
       void com.winterthur.iqskernel.unittest.util.DateHelperTest.testGetDiffInt
Finished: 1.492 seconds                                       Exit
```

```
JUnit
JUnit
Test class name:
com.winterthur.iqskernel.unittest.util.DateHelperTest   ...   Run
Reload classes every run

         Runs:  Errors:  Failures:
         25     0        1

 ✓ testSetFirstDayOfMonth                                     Run
 ✓ testCreateSafe
 X testGetDiffInMonths
 ✓ testIsLeapYear
 ✓ testIncDay
Failures   Test Hierarchy
junit.framework.AssertionFailedError: expected:<0> but was:<1>
java.lang.Throwable(java.lang.String)
java.lang.Error(java.lang.String)
Finished: 5.458 seconds                                       Exit
```

Figure 10: After running a test method indicating an error, the progress bar of *TestRunner* appears in **red**.

In case of an error, the cause of the error appears in the lower input window of *TestRunner*. (The Swing application offers more comfort here.) The fields *Runs*, *Errors* and *Failures* have the following meanings:

- **Runs:** Shows how many test methods have run (with and without errors).
- **Errors:** Shows the total number of *exceptions* that the test methods threw. Only unforeseen errors are shown.
- **Failures:** Shows how many *fail* methods have run and how many *assert* methods failed. Only expected errors are shown.

For complete information on *TestRunner*, see [JUnit].

2.4.4 When an error occurs

In Figure 10, an error has occurred in *DateHelperTest.testGetDiffInMonths()*[15]. This method contains various tests for the method *DateHelper.getDiffInMonths()* and can, for example, look like this:

```
    public void testGetDiffInMonths() {
```

[15] If the *TestRunner* detects an error, a developer usually analyses and corrects it with the help of the debugger.

```
    //-- Test: 0 months' difference
    mdtAct1 = DateHelper.createSafe(2000, 2, 20);
    mdtAct2 = DateHelper.createSafe(2000, 2, 19);
    assertEquals(0, DateHelper.getDiffInMonths(mdtAct1, mdtAct2));   // A
    assertEquals(0, DateHelper.getDiffInMonths(mdtAct2, mdtAct1));   // B

    //-- Test: 1 month's difference
    mdtAct1 = DateHelper.createSafe(2000, 2, 20);
    mdtAct2 = DateHelper.createSafe(2000, 3, 1);
    assertEquals(1, DateHelper.getDiffInMonths(mdtAct1, mdtAct2));   // C
    assertEquals(1, DateHelper.getDiffInMonths(mdtAct2, mdtAct1));   // D

    //-- Test: 12 months' difference
    mdtAct1 = DateHelper.createSafe(2000, 2, 20);
    mdtAct2 = DateHelper.createSafe(2001, 2, 20);
    assertEquals(12, DateHelper.getDiffInMonths(mdtAct1, mdtAct2));  // E
    assertEquals(12, DateHelper.getDiffInMonths(mdtAct2, mdtAct1));  // F

    //-- Test: 25 months' difference
    mdtAct1 = DateHelper.createSafe(2000, 1, 11);
    mdtAct2 = DateHelper.createSafe(2002, 2, 11);
    assertEquals(25, DateHelper.getDiffInMonths(mdtAct1, mdtAct2));  // G
    assertEquals(25, DateHelper.getDiffInMonths(mdtAct2, mdtAct1));  // H
  }
```

Using *assert* methods, lines A–H carry out the tests (in other words, they compare the expected results with the calculated ones). When an *assert* method encounters something wrong, it triggers an *exception* and aborts the execution of the current method. (Thus, for example, if an error appears in D, tests E through H will not run.)

2.4.5 Giving a test class its own TestRunner

If needed, each test class can call the *TestRunner* on its own. This occurs as follows, within the *main* method:

```
public class DateHelperTest extends TestCase {
  ...

  public static void main(String[] args) {
    //-- Call TestRunner, giving the name of the calling test class (= auto-
    //-- start)
    junit.swingui.TestRunner.main(new String[]{DateHelperTest.class.getName()});
  }
}
```

The advantage of this variant is that after the *main* method starts, the *TestRunner* automatically loads and immediately runs the test class. (This avoids your having to type the whole class name by hand within *TestRunner.*)

Class path: Please note that in this variant as well, either the project *Eve JUnit* or the project *Jackpot JUnit* must be in the class path. (See also section *2.4.2, Setting the class path for the general TestRunner class*).

2.5 Test suites

In Section *2.4, Carrying out tests*, we saw how to execute individual test classes. It is best to gather several test classes within a single test suite; the suite then executes all its test classes in one operation. Concerning test suites, please note the following (see also Figure 11):

- A test suite is a subclass of TestSuite; apply the suffix "Suite" to it (for example ApplSuite).
- One test suite can contain any number of test classes.

- If needed, you can assign several test suites to a project.
- In the method *suite()*, state the test classes and test suites that belong to the test suite.
- The application *TestRunner* helps run test suites (and test classes).

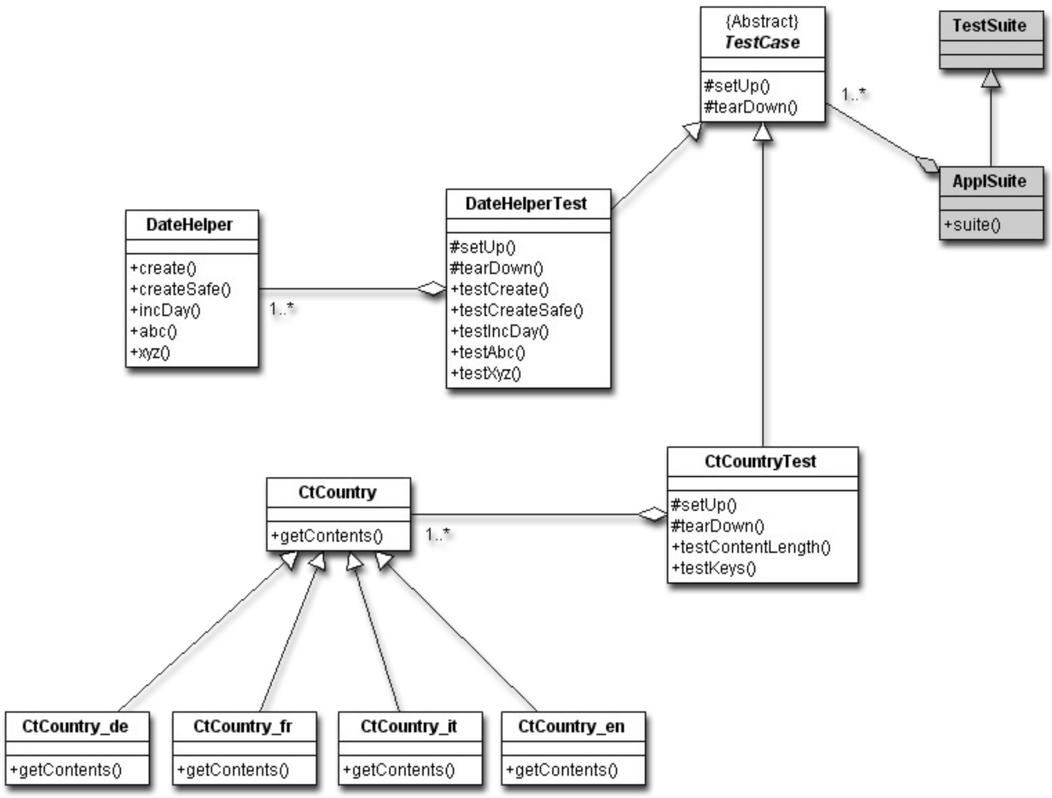


Figure 11: The test suite *ApplSuite* – which is marked in gray – can contain several test classes. You place these in the method *suite()*. (Examples here are the test classes *DateHelperTest* and *CtCountryTest.*)

2.5.1 Packages for test suites

Assign the test suites directly to the project you are testing, in a package named "unittest".[16] Here's an example:

```
com.winterthur.iqskernel.unittest
```

2.5.2 Creating a test suite

As we described at the start of section 2.5, a test suite inherits from the class *TestSuite* and looks, for example, like this:

```
import junit.framework.Test;
import junit.framework.TestSuite;

import com.winterthur.iqskernel.unittest.util.DateHelperTest;
```

[16] As in the case of the test classes (see section 2.3.1), each department can of course decide for itself what to call the package for the test suites. In VICS 3 (iSolutions) we again call it *unittest*.

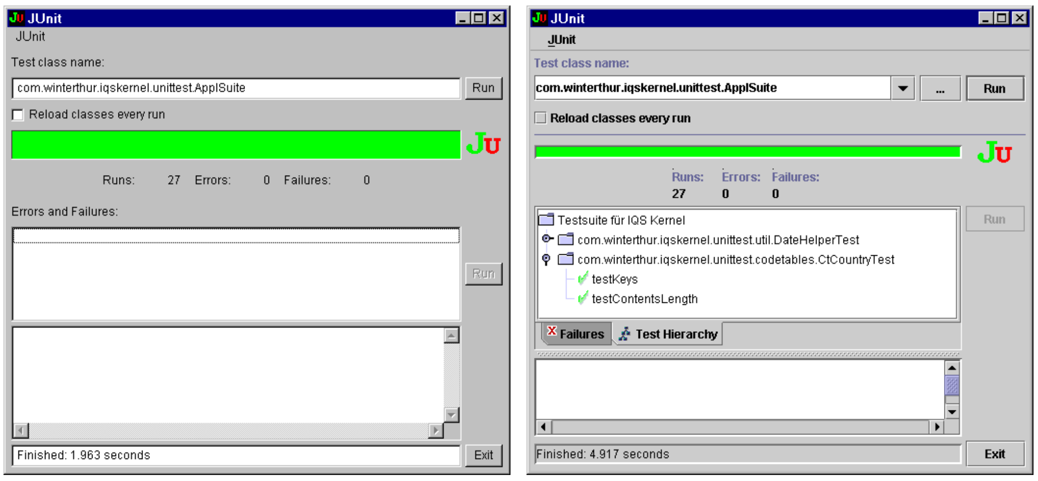
```
import com.winterthur.iqskernel.unittest.codetables.CtCountryTest;

/**
 * Brings together all test classes that the test suite runs
 *
 *
 * Creation date: (16 May 2001)
 * @author: John Johnson
 */
public class ApplSuite extends TestSuite { ... }
```

2.5.3 Bundling test classes together

Within the static method *suite()*, declare all the test classes belonging to a given test suite, as follows:

```
public class ApplSuite extends TestSuite {
  ...
  public static Test suite() {
    TestSuite suite = new TestSuite("Test suite for IQS Kernel");  // A
    suite.addTestSuite(DateHelperTest.class);                      // B
    suite.addTestSuite(CtCountryTest.class);                       // C
    return suite;                                                  // D
  }
}
```

In this method, line A creates the new test suite *suite* (including an explanatory text). Lines B and C add the test classes to be executed, and line D then returns, as a function value, the test suite *suite*. After running a test suite, *TestRunner* looks as follows:



Test class name: com.winterthur.iqskernel.unittest.ApplSuite

Runs: 27 Errors: 0 Failures: 0

Finished: 1.963 seconds

Test class name: com.winterthur.iqskernel.unittest.ApplSuite

Runs: 27 Errors: 0 Failures: 0

```
Testsuite für IQS Kernel
  com.winterthur.iqskernel.unittest.util.DateHelperTest
  com.winterthur.iqskernel.unittest.codetables.CtCountryTest
    testKeys
    testContentsLength
```

Finished: 4.917 seconds

Figure 12: The AWT TestRunner (left) says nothing about the contents of the test suite it has just carried out, whereas the Swing TestRunner shows – in a tree structure – all the test classes belonging to the suite. You can also open any test class and see its test methods.

Just as you can in the case of test classes, if you need to, you can again call the TestRunner within the main method (see also section *2.4.5, Giving a test class its own TestRunner*):

```
public class ApplSuite extends TestSuite {
  ...
  public static void main(String[] args) {
    //-- Call TestRunner, specifying your own class names (= Auto-Start)
    junit.swingui.TestRunner.main(new String[]{ApplSuite.class.getName()});
  }
}
```

2.6 Thoughts on testing

Kent Beck, Erich Gamma, and Martin Fowler have already given much thought to the question of automatic testing of applications (see [Beck2000] page 45 and following, page 115 and following; [Fowler1999], page 90 and following). Here are some major thoughts from Martin Fowler's book:

- If you want to refactor, the essential precondition is having solid tests (page 89).
- I've found that writing good tests greatly speeds my programming, even if I'm not refactoring (page 83).
- If you look at how most programmers spend their time, you'll find that writing code actually is quite a small fraction. Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging. [...] Every programmer can tell a story of a bug that took a whole day (or more) to find. Fixing the bug is usually pretty quick, but finding it is a nightmare (page 89).
- When tests are manual, they are gut-wrenchingly boring. But when they are automatic, tests can actually be quite fun to write (page 90).
- When I develop code, I write the tests as I go (page 91).
- When I'm working with people on refactoring, often we have a body of non-self-testing code to work on. So first we have to make the code self-testing before we refactor (page 91).
- I don't test accessors that just read and write a field. Because they are so simple, I'm not likely to find a bug there (page 97).
- The key is to test the areas that you are most worried about going wrong. That way you get the most benefit for your testing effort (page 98).
- I'm playing the part of an enemy to code. I'm actively thinking about how I can break it. I find that state of mind to be both productive and fun. It indulges the mean-spirited part of my psyche (page 100).
- There's always a risk that I'll miss something, but it is better to spend a reasonable time to catch most bugs than to spend ages trying to catch them all (page 102).

2.7 Guidelines for testin g

The following guidelines for automatic testing also come from Martin Fowler's book (see [Fowler1999], pages 90 and following):

- Make sure all tests are fully automatic and that they check their own results.
- A suite of tests is a powerful bug detector that decapitates the time it takes to find bugs.
- Run your tests frequently. Localize tests whenever you compile – every test at least every day.
- When you get a bug report, start by writing a unit test that exposes the bug.
- It is better to write and run incomplete tests than not to run complete tests.
- Think of the boundary conditions under which things might go wrong and concentrate your tests there.
- Don't forget to test that exceptions are raised when things are expected to go wrong.

- Don't let the fear that testing can't catch all bugs stop you from writing the tests that will catch most bugs.

3. HttpUnit: Testing servlet applications

The JUnit framework offers no special help for testing servlet applications. Although it's easy to use the standard package *java.net* to open an HTML connection and to read an HTML page from the server, you must take care of evaluating the data in the HTML page yourself.

An illustration: The following class, *HtmlReader*, displays an HTML page on the screen:

```
import java.net.*;
import java.io.*;

public class HtmlReader {

  public static void main(String[] args) {
    try {
      //-- create a URL instance
      URL url = new URL("http://www.winterthur-life.ch");

      //-- read the HTML document and, with the help of readers, make it legible
      InputStreamReader isr = new InputStreamReader(url.openStream());
      BufferedReader br = new BufferedReader(isr);

      //-- Display HTML document
      while (br.ready()) {
        System.out.println(br.readLine());
      }

      //-- Clean up
      br.close();
    } catch (Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

The *main* method above generates the following (meter-long) output:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<base href="http://www.winterthur-life.ch/">
  <title>Winterthur Life - Home</title>
  <!-- Meta-Tags -->
  <META HTTP-EQUIV="Pragma" CONTENT="No-Cache">
  <META HTTP-EQUIV="Content-Language" CONTENT="de">
  <META NAME="Description" LANG="de" CONTENT="Home">

... (etc.)
```

To correctly recognize the links, frames, forms, input fields, and buttons in this HTML code, you must analyze and evaluate the data. Without help classes, this requires a good deal of sweat[17].

[17] Sweat (noun): Liquid secretion from glands in the skin; is composed of water (99%), salt, urea, fats and (foul-smelling) volatile fatty acids; plays an important role in the temperature regulation of the body. Source: *Der Brockhaus in einem Band* (The One-Volume Brockhaus Encyclopedia), Leipzig, 2000.

Russell Gold recognized this problem and, as a response, developed the Java library *HttpUnit*[18], which makes it very easy to fill out HTML forms in a Java program, start server requests, receive server responses, and interpret HTML data. In combination with JUnit, HttpUnit is well-suited to writing automated texts for (dynamic) server applications.

The basic idea of HttpUnit is simulate, with the help of a Java class, an ordinary web browser such as Netscape Navigator, Internet Explorer, or Opera. Instead of a (graphic) user interface, HttpUnit makes available various classes and methods that let you very easily fill out forms, call up links, or send server requests.

3.1 Making HttpUnit available

To be able to use HttpUnit, you must load various projects into the local workspace of VisualAge. For this purpose, Eve offers two projects:

- **Eve HttpUnit:** Contains all classes belonging to the core of HttpUnit. As we just mentioned, these were mainly developed by Russell Gold (see also [HttpUnit]).
- **Eve JSSE:** JSSE (*Java Secure Socket Extension*) comes from the company Sun and is needed for web access via the protocol HTTPS (see also section *3.7, The protocols HTTP and HTTPS*).

In addition to these projects, HttpUnit will also need the following two projects. They must therefore be present in the local workspace of VisualAge (both these projects are part of the standard delivery package of the Jackpot framework):

- **SAX 2.0:** SAX (which is short for "Simple API for XML") is mostly the work of David Megginson; you need it in order to analyze and parse XML data (see also http://www.megginson.com/SAX/).
- **W3C DOM level 2 v. 1.1 a:** DOM (the Document Object Model) comes from the World Wide Web Consortium (W3C) and is an interface that lets you read and change the contents, structure, and formatting of documents. This interface is independent of platform and language. The documents typically contain HTML or XML data (see also http://www.w3.org/DOM/).

Warning: For trouble-free experimentation with HttpUnit, make sure the *ServletExecDebugger* 3.1.x is available in VisualAge (see section *3.9, Known problems in HttpUnit*).

3.2 A first program

With the help of the class *EveWebBrowser*, the following program generates a virtual web browser that can trigger a server request and then display the response on the screen:

```
import com.meterware.httpunit.*;
import com.winterthur.eve.httpunit.EveWebBrowser;

public class MyHttpTest {

  public static void main(String[] args) {
```

[18] See [HttpUnit].

```
        try {
          //-- A: Create a virtual web browser
          EveWebBrowser browser = new EveWebBrowser();

          //-- B: Generate "get" request
          WebRequest req = new GetMethodWebRequest("http://www.winterthur-life.ch");

          //-- C: Transmit "get" request and determine response
          WebResponse resp = browser.getResponse(req);

          //-- D: Display HTML document
          System.out.println(resp.getText());
        } catch (Exception e) {
          System.out.println("Exception: " + e.getMessage());
        }
      }
    }
```

In this program, line A generates a virtual web browser with which you can then submit whatever requests you want. Line B generates a *get* request that line C then sends to the virtual web browser. Finally, the lines in D display the response on the screen. The class *MyHttpTest* generates exactly the same output as the class *HtmlReader* (see the start of chapter 3).

A reminder: To be able to run this program correctly, you must of course first set the classpath correctly (see section *2.4.2, Setting the class path for the general TestRunner class*).

3.3 Packages for test programs

A project can have at its disposal several test classes that, with the help of virtual web browsers, can test various web sites. The test sites are stored in a (sub-)package called *unittest* (see section *2.3.1, Packages for test classes*). Examples:

```
com.winterthur.iqsinsurancelab.unittest
com.winterthur.iqsinsurancelab.unittest.lifefund
com.winterthur.iqsinsurancelab.unittest.lifestar3a
```

3.4 The classes of HttpUnit

In this section, we will discuss the most important classes of HttpUnit. You'll find most precise information directly in the source code of HttpUnit. To understand the classes and methods of HttpUnit, some knowledge of HTML is indispensable (see here the excellent book [Musc2000]).

3.4.1 Virtual browser (class "EveWebBrowser")

To generate a virtual web browser, you need the class *EveWebBrowser* (package *com.winterthur.eve.httpunit*), which makes available three constructors:

- *public EveWebBrowser()* – Generates a virtual browser for data that does not move via a proxy server. Using this method, within Winterthur, you can access data by way of the intranet, and not by way of Internet:

```
//-- Browser for intranet access (without a proxy server)
EveWebBrowser browser = new EveWebBrowser();
```

- *public EveWebBrowser(String userName, String password)*: Generates a virtual browser whose data does move by way of a proxy server. The method uses the standard proxy server "proxy001.winterthur.ch" with the port number 8080. (Using this method, from within Winterthur, you can access the Internet, but usually not the intranet.) The parameters *userName* and *password* must contain a valid user name and password; otherwise, the proxy server blocks Internet access:

```
//-- Browser for Internet access (by way of Winterthur's standard proxy server)
EveWebBrowser browser = new EveWebBrowser("c123456", "sockhole");
```

- *public EveWebBrowser(String userName, String password, ProxyServerData proxy):* Generates a virtual browser that, like the previous constructor, communicates by way of a proxy server. But here, you can specify the proxy server yourself with the parameter *proxy*.

```
import com.winterthur.eve.httpunit.ProxyServerData;
import com.winterthur.eve.httpunit.EveWebBrowser;

...

//-- Specify proxy server
ProxyServerData proxy = new ProxyServerData("proxy.eve.ch", 7777);

//-- Browser for Internet access (by way of any proxy server)
EveWebBrowser ewb1 = new EveWebBrowser("c123456", "solo", proxy);

//-- Browser for Internet access (by way of Winterthur's standard proxy server)
EveWebBrowser ewb2 = new EveWebBrowser("c123456", "solo");
EveWebBrowser ewb3 = new EveWebBrowser("c123456", "solo", ProxyServerData.WIN);
```

In this example, browsers *ewb2* and *ewb3* both work the same way. The class *ProxyServerData* makes available the constant *WIN*, which holds the data for Winterthur's standard proxy server. If users of HttpUnit so desire, iSolutions is willing to define additional constants here.

For your information: For problem-free web access by way of the proxy server, various properties for the system and communications must be set. The class that iSolutions has written, *EveWebBrowser*, automatically takes care of all the necessary details. *EveWebBrowser* inherits directly from the class *WebConversation*, which the father of HttpUnit (Russell Gold) foresees as a virtual web browser. Unfortunately, this class does not allow Internet access via a proxy server.

3.4.2 Server request (class "W ebRequest")

Before you can use an *EveWebBrowser* instance to trigger a server request, you must specify the server request more exactly with a *WebRequest* instance. For this purpose, the following two classes (among others) are available:

- *GetMethodWebRequest:* Subclass of *WebRequest*; generates a *get* request (in other words, sends form data in the URL directly to the server).
- *PostMethodWebRequest:* Subclass of *WebRequest*; generates a *post* request (in other words, sends form data to the server in a data flow that is separate from the URL).

Normally, you have to generate a *GetMethodWebRequest* instance "by hand" only once – when you access the home page of the web application you want to test. HttpUnit subsequently generates all other needed *GetMethodWebRequest* or *PostMethodWebRequest* instances. Some examples:

```
WebRequest req;

//-- "Get" request
req = new GetMethodWebRequest(
  "http://" +                                       // Protocol
```

```
    "www.winterthur-life.ch");                         // Hostname

  //-- "Get" request, including port
  req = new GetMethodWebRequest(
    "http://" +                                        // Protocol
    "marvin.winterthur.ch:8870");                      // Hostname and port

  //-- "Get" request, including parameters
  req = new GetMethodWebRequest(
    "http://" +                                        // Protocol
    "iqs.winterthur.ch/" +                             // Hostname
    "insurancelab/servlet/LifeStar3a?" +               // Servlet (including
                                                       // path)
    "language=en&transition=NewQuote&frame=Frameset");  // Parameters

  //-- "Get" request via "https" protocol, including parameters
  req = new GetMethodWebRequest(
    "https://" +                                       // Protocol
    "entry.winterthur.com/" +                          // Hostname
    "insurancelab/servlet/Lf2?" +                      // Servlet (including
                                                       // path)
    "country=CHE&language=0&transition=start");        // Parameters
```

The classes *GetMethodWebRequest* and *PostMethodWebRequest* both have various constructors. Normally, however, you will only need the constructor of the class *GetMethod-WebRequest* shown is the code just above. (You'll find additional information in the source code of these two classes.)

3.4.3 Server response (class "WebResponse")

The virtual web browser packs the response it receives from the server in an instance of the class *WebResponse*. You never need to generate this instance "by hand". Instead, the virtual web browser always generates and returns it. An example:

```
  EveWebBrowser browser = new EveWebBrowser();
  WebRequest req = new GetMethodWebRequest("http://www.winterthur-life.ch");

  //-- Transmit request and determine response
  WebResponse resp = browser.getResponse(req);
```

An instance of *WebResponse* offers various interesting methods. The following sections describe these.

3.4.4 Hyperlinks (class "WebLink")

The *WebLink* class corresponds to a link to an HTML page. The classes *WebResponse* and *WebLink* offer various methods that let you work with links.

"WebResponse" – Finding all links: Using the method *getLinks()*, you can find all the links on an HTML page:

```
  EveWebBrowser browser = new EveWebBrowser();
  WebRequest req = new GetMethodWebRequest("http://www...");
  WebResponse resp = browser.getResponse(req);

  //-- find all links in the HTML document
  WebLink[] links = resp.getLinks();
```

"WebResponse" – Finding certain links: In an HTML document, links can be in the form of either text or an image:

- *public WebLink getLinkWith(String text):* Finds a link containing the text *text*. The parameter *text* can correspond to either the complete text, or to a text fragment (uppercase and lowercase are irrelevant):

```
<!—HTML code -->
<a href="home/privateind.htm">Private individual</a>


//-- Java code
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);
WebLink link;
//-- Find text link (all calls lead to the same place)
link = resp.getLinkWith("Private individual");
link = resp.getLinkWith("Priv");
link = resp.getLinkWith("PRIV");
link = resp.getLinkWith("prIV");
link = resp.getLinkWith("ate ind");
```

- *public WebLink getLinkWithImageText(String text):* Returns a link that appears as an image in the HTML page and that has an associated text (specified by the *alt* attribute). Again here, the parameter *text* can correspond to either the complete text or to a text fragment (uppercase and lowercase are irrelevant):

```
<!-- HTML code -->
<a href="home/privateind.htm"><img src="img/head.gif" alt="Private
individual"></a>

//-- Java code
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);
WebLink link;

//-- Find image link (all calls lead to the same place)
link = resp.getLinkWithImageText("Private individual");
link = resp.getLinkWithImageText("Priv");
link = resp.getLinkWithImageText("PRIV");
link = resp.getLinkWithImageText("prIV");
link = resp.getLinkWithImageText("ate ind");
```

"WebLink" – Following a link: The class *WebLink* offers the following method, which lets you follow a link:

- *public WebRequest getRequest():* Returns a *WebRequest* instance that you can use to call the HTML page cited in the link:

```
WebRequest req;
WebResponse resp;
WebLink link;

EveWebBrowser browser = new EveWebBrowser();
req = new GetMethodWebRequest("http://www.winterthur-life.ch");

//-- Call first page
resp = browser.getResponse(req);     // Trigger request
link = resp.getLinkWith("privat");   // Find link
req = link.getRequest();             // Find the object that the link requests
```

```
//-- Call the second page (follow the link)
resp = browser.getResponse(req);    // Trigger new request
```

3.4.5 Tables (class "WebTable")

The class *WebTable* corresponds to a table in an HTML page. With the help of the classes *WebResponse* and *WebTable*, you can work with tables very comfortably. The following examples illustrate.

"WebResponse" – Finding all tables: The method *getTables()* delivers all tables on the top level of an HTML page (*getTables()* thus *ignores* nested inner tables):

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find all tables in the HTML document
WebTable[] tables = resp.getTables();
```

"WebResponse" – Finding tables with certain attributes: HTML code can identify a table with the attribute *id* or *summary*. These elements are very easy to find here:

- *public WebTable getTableWithID(String text):* Searches for a table with the *id* attribute *text* and returns it (or returns *null* if search fails). The search includes nested inner tables. (Uppercase and lowercase are irrelevant.)
- *public WebTable getTableWithSummary(String text):* This is just like the previous method, except that this method searches for a table with the specified *summary* attribute.

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find particular tables
WebTable tableId = resp.getTableWithID("base data");
WebTable tableSum = resp.getTableWithSummary("financing of life fund");
```

"WebResponse" – Finding tables with certain cell contents: Even if a table bears no *id* or *summary* attribute, you can still find the table by the contents of its cells. For this purpose, the methods *getTableStartingWith(String textOfFirstCell)* and *getTableStartingWithPrefix(String prefixTextOfFirstCell)* are available (see source code for further information).

"WebTable" – General methods: The class *WebTable* offers two methods that let you call up table attributes:

- *public String getID():* Returns the *id* attribute of the table (see also the method *WebResponse.getTableWithID(String id)* just above).
- *public String getSummary():* Returns the *summary* attribute of the table (see also the method *WebResponse.getTableWithSummary(String summary)* just above).

"WebTable" – Determining the size of a table: With the help of the following two methods, you can determine the number of columns and rows in a table:

- *public int getColumnCount()*: Returns the number of columns in a table.
- *public int getRowCount()*: Returns the number of rows in a table.

"WebTable" – Determining the contents of cells: Cell contents are very easy to call up:

- *public String getCellAsText(int row, int col)*: Returns the contents of a cell, where *getCellAsText(0, 0)* corresponds to the upper left corner of a table.
- *public TableCell getTableCell(int row, int col)*: Returns the contents of a cell as a *TableCell* instance, where *getTableCell(0, 0)* corresponds to the upper left corner of a table. (For further information about *TableCell*, see the source code.)

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find first table
WebTable tab = resp.getTables()[0];

//-- Determine number of rows and columns
int rows = tab.getRowCount();
int cols = tab.getColumnCount();

//-- Display table
for (int r=0; r<rows; r++) {
  System.out.print("Row " + r + ": ");
  for (int c=0; c<cols; c++) {
    System.out.print(tab.getCellAsText(r, c) + " / ");
  }
  System.out.println("");
}
```

3.4.6 Forms (Class "WebForm")

The class *WebForm* corresponds to a form in an HTML page. The classes *WebForm* and *WebResponse* let you work with forms.

"WebResponse" – Find all forms: The method *getForms()* returns all the forms in an HTML page.

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find all forms in the HTML document
WebForm[] forms = resp.getForms();
```

"WebResponse" – Finding forms with certain attributes: HTML code can identify a form with the attribute *id* or *name*; these elements are very easy to find here:

- *public WebForm getFormWithID(String id):* Returns the form with the desired *id* attribute (*id* must correspond exactly to the HTML "id" attribute; uppercase and lowercase are irrelevant).
- *public WebForm getFormWithName(String name)*: This is just like the previous method, except that this method searches for a form with the specified *name* attribute.

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find particular forms
WebForm formId = resp.getFormWithID("persdata");
WebForm formName = resp.getFormWithName("addrdata");
```

"WebForm" – General methods: The class "WebForm" offers a whole series of interesting methods.

- *public String getID()*: Returns the *id* attribute of the form (see also the method *WebResponse.getFormWithID(String id)* just above).
- *public String getName()*: Returns the *name* attribute of the form (see also the method *WebResponse.getFormWithName(String id)* just above).

"WebForm" – Calling up form parameters: Using the following methods, you can call up the currently-set form parameters (fields). You can also set these parameters: see page 38. Warning: The parameters of the following methods *are* case-sensitive (in other words, uppercase and lowercase *are* relevant):

- *public String[] getParameterNames()*: Returns the names of all the parameters in a form. Form parameters can include, for example, an input field, radio button, or selection list. (A request forwards these parameters and their assigned values to the server.)
- *public String getParameterValue(String paramName)*: Returns the value that is assigned to the parameter *paramName*.
- *public String[] getParameterValues(String paramName)*: Returns all values that are assigned to the parameter *paramName*.
- *public boolean isMultiValuedParameter(String paramName)*: Returns *true* if the form parameter can return several values (otherwise *false*).

```
<!-- HTML code -->
<form name="Order"
  action="/servlet/Order?transition=ViewCart&frame=Frameset"
  target="_top"
  method="post">
    <select name="SelFurniture" size=6 multiple>
      <option value='1'>Sofa</option>
      <option value='2'>Bed</option>
      <option value='3'>Table</option>
      <option value='4'>Cupboard</option>
    </select>
</form>

//-- Java code
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www...");
WebResponse resp = browser.getResponse(req);

//-- Find the form "Order" and call up its parameters
String[] multiSel = null;
String singleSel = null;
WebForm orderForm = resp.getFormWithName("Order");
if (isMultiValuedParameter("SelFurniture")) {
  multiSel = orderForm.getParameterValues("SelFurniture");
} else {
  singleSel = orderForm.getParameterValue("SelFurniture");
}
```

"WebForm" – Calling up the options for a form parameter: You can call up the options for form parameters with the following methods. (Warning: The parameters of the following methods *are* case-sensitive (in other words, uppercase and lowercase *are* relevant):

- *public String[] getOptions(String paramName)*: Returns the displayable texts of the form parameter *paramName.* (This method might, for example, return all the texts that appear in a selection list.)

- *public String[] getOptionValues(String paramName)*: Returns the values associated with the form parameter *paramName*. (A value can be assigned to each displayed text in a list; the method *getOptionValues(...)* returns all these values.)

```
//-- Java code
WebForm orderForm = resp.getFormWithName("Order");      // see HTML
                                                         // code just above

String[] opts = orderForm.getOptions("SelFurniture);
// <opts> contains: "Sofa", "Bed", "Table", "Cupboard"

String[] vals = orderForm.getOptionValues("SelFurniture");
// <vals> contains: "1", "2", "3", "4"
```

"WebForm" – Finding submit buttons: Various methods for finding *submit* buttons are available. A reminder: You use *submit* buttons to send form contents to the server.

- *public SubmitButton[] getSubmitButtons()*: Returns all the *submit* buttons that are present in a form.
- *public SubmitButton getSubmitButton(String name)*: Returns all *submit* buttons bearing the name *name*.
- *public SubmitButton getSubmitButton(String name, String value)*: Returns all *submit* buttons that bear the name *name* and that are associated with the value *value*.

```
<!-- HTML code: a few examples of submit buttons -->
<input type="submit">
<input type="submit" name="theMagicSubmitButton"
<input type="submit" name="theSubBut" value="nr1">
<input type="submit" name="theSubBut" value="nr2">
```

"WebForm" – Retrieving a form request: With the following methods, you can generate request instances that are associated with a form:

- *public WebRequest getRequest()*: Returns the request associated with a form (see also the form parameter *action* in the sample form *Order* on page 36).
- *public WebRequest getRequest(SubmitButton button)*: Returns the request that the submit button *button* triggers.
- *public WebRequest getRequest(SubmitButton button, int x, int y)*: Returns the request that the submit button *button* triggers at the coordinates (*x/y*). This method only makes sense for submit buttons that consist of an image.

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www.winterthur.com");

//-- Trigger request (first page)
WebResponse resp = browser.getResponse(req);

//-- Find forms
WebForm form = resp.getFormWithName("globe");

//-- Retrieve form request
req = form.getRequest();

//-- Trigger request (second page)
resp = browser.getResponse(req);
```

"WebRequest" – Setting form parameters: The following methods let you set the parameters (fields) within a form. (You must do this before triggering a request.) Warning: The parameters *paramName* of the following methods are case-sensitive (in other words, uppercase and lowercase are relevant):

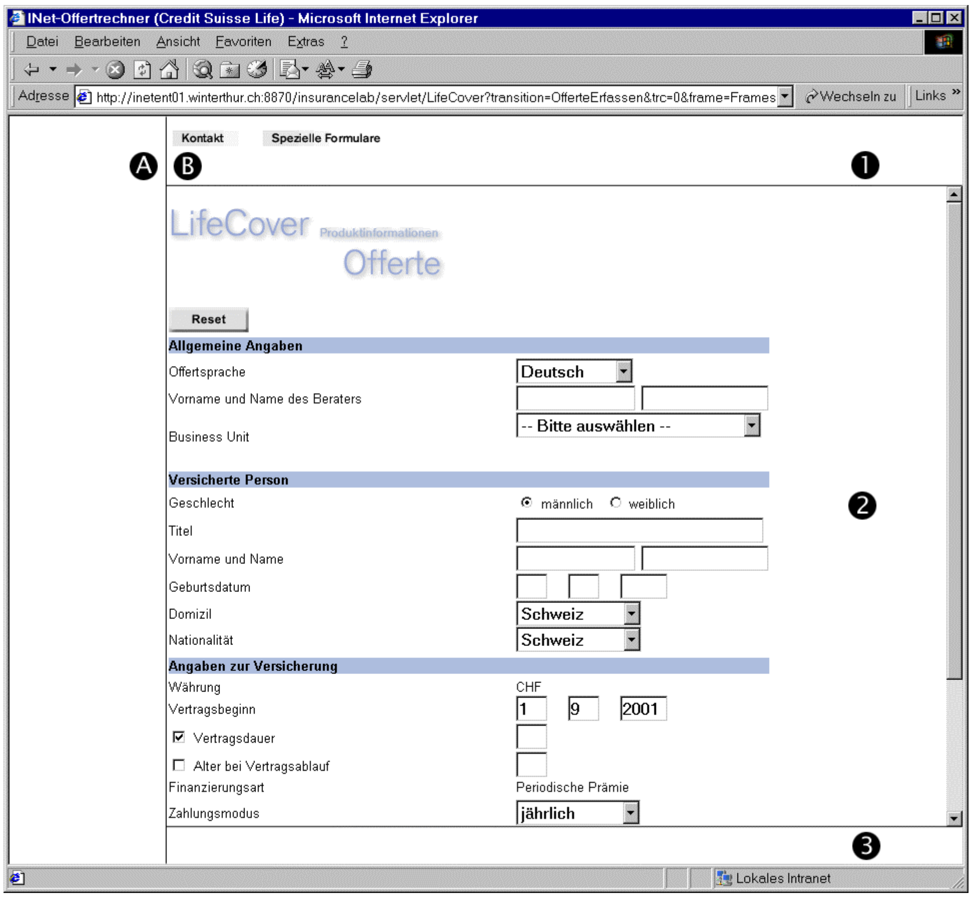
- *public void setParameter(String paramName, String value)*: Assigns the value *value* to the form parameter *paramName*.
- *public void setParameter(String paramName, String[] values)*: Assigns the values *values* to the form parameter. (In this case, the parameter must be a *multiple* one; see also the method *WebForm.isMultiValuedParameter(...),* on page 36).

```
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = new GetMethodWebRequest("http://www.winterthur.com");

//-- Trigger request (first page)
WebResponse resp = browser.getResponse(req);

//-- Find form "globe"
WebForm form = resp.getFormWithName("globe");

//-- Retrieve form request
req = form.getRequest();

//-- Set form parameters
req.setParameter("Make", "11");  // = IT - Italy

//-- Trigger request (second page)
resp = browser.getResponse(req);
```

3.4.7 Working with frames

Instead of a single HTML document, the presentation window of a browser can instead contain a so-called *frameset*. A frameset consists of several independent *frames*, each of which can display its own HTML document. One frame document can contain others (you can nest them at will), which allows creation of super-cool[19] user interfaces (with menu or navigation bars, headers and footers).

[19]In recent years, frames enjoyed a veritable boom. Today, however, many web designers are again managing to live without frames because the resulting "super-cool" navigation was at times "too super-cool" and turned out to be impractical or complicated. Nevertheless: As long as you don't overdo it, frames still let you do some pretty cool things.


INet-Offertrechner (Credit Suisse Life) - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Adresse
Wechseln zu
Links
Kontakt
Spezielle Formulare
A
B
1
2
3
Reset
Lokales Intranet

http://inetent01.winterthur.ch:8870/insurancelab/servlet/LifeCover?transition=OfferteErfassen&trc=0&frame=Frames

LifeCover Produktinformationen
Offerte

Allgemeine Angaben	
Offertsprache	Deutsch
Vorname und Name des Beraters	
Business Unit	-- Bitte auswählen --
Versicherte Person	
Geschlecht	[✓ männlich] weiblich
Titel	
Vorname und Name	
Geburtsdatum	
Domizil	Schweiz
Nationalität	Schweiz
Angaben zur Versicherung	
Währung	CHF
Vertragsbeginn	1 9 2001
[✓] Vertragsdauer	
☐ Alter bei Vertragsablauf	
Finanzierungsart	Periodische Prämie
Zahlungsmodus	jährlich

Figure 13: The main window of the product *LifeCover* contains a frameset consisting of the frames **A** and **B**. Frame **B** in turn contains another frameset containing the three frames **1**, **2** and **3**. In total, the main window displays four HTML documents (**A**, **1**, **2** and **3**). The documents have the following roles: A = Navigation bar (empty here), 1 = Header line, 2 = Data entry, 3 = Footer bar (empty here).

"EveWebBrowser" – Working with frames: The class *EveWebBrowser* offers two methods that make it easier to work with frames:

- *public String[] getFrameNames()*: Returns all available frame names; includes all frames on all nesting levels. Calling this method of course only makes sense when you have previously triggered a request (with *getResponse(...)*).
- *public WebResponse getFrameContents(String frameName)*: Triggers a request and returns the contents of the frame *frameName*. (This method also takes into account frames lying at a lower nesting level.) If *frameName* doesn't exist, the method throws the exception *NoSuchFrameException*.

```
//-- Generate web browser
EveWebBrowser browser = new EveWebBrowser();
WebRequest req = null;
WebResponse resp = null;

//-- Generate request
req = new GetMethodWebRequest(
  "http://" +
  "inetent01:8870/" +
  "insurancelab/servlet/LifeCover?" +
  "language=en&transition=NewQuote&frame=Frameset");
```

```
//-- Trigger first request (response: frameset)
resp = browser.getResponse(req);

//-- Trigger second request (response: document in the "content" frame)
resp = browser.getFrameContents("content");

//-- Continue to process the received HTML document
WebForm form = resp.getFormWithName("nar1");
```

3.5 Configuring HttpUnit (class "HttpUnitOptions")

Using the class *HttpUnitOptions*, you can influence the behavior of HttpUnit.

Forwarding of pages: In the HTML header, you can place a forwarding instruction that causes the browser to load another page after a certain number of seconds.

- *public static boolean getAutoRefresh()*: Returns *true* if HttpUnit follows the forwarding instruction in the HTML header (otherwise *false* = default). By default, a request will thus yield the forwarding page (*index.htm* in the following example) and not the target page (*yes.htm* in the example).
- *public static void setAutoRefresh(boolean autoRefresh)*: Turns the forwarding mode on (= *true*) or off (= *false*).

```
<!-- HTML code of "index.htm" -->
<html>
<head>
  <title>Refresh or stay</title>
  <meta http-equiv="refresh" content="1; URL=http://www.eve.ch/yes.htm">
</head>
<body bgcolor="#ffffff">
</body>
</html>

//-- Java code
HttpUnitOptions.setAutoRefresh(false);        // = Default
EveWebBrowser browser = new EveWebBrowser();

//-- Retrieve forwarding page "index.htm"
WebRequest req = new WebRequest("http://eve.winterthur.ch");
WebResponse resp = browser.getResponse(req);  // = "index.htm"

//-- Retrieve target page "yes.htm"
req = resp.getRefreshRequest();
resp = browser.getResponse(req);              // = "yes.htm"

//-- Activate auto-refresh
HttpUnitOptions.setAutoRefresh(true);

//-- Retrieve target page "yes.htm" directly
req = new WebRequest("http://eve.winterthur.ch");
resp = browser.getResponse(req);              // = "yes.htm"
```

Waiting period for forwarding: When you forward HTML pages, you can specify a waiting time (see also the method *getAutoRefresh()* in the example we just examined):

- *public static int getRedirectDelay()*: Returns the waiting time (in milliseconds) at the end of which the browser automatically loads the target page.
- *public static void setRedirectDelay(int delayInMilliseconds)*: Lets you set the waiting time (in milliseconds).

```
<!-- HTML code: Forwarding to target after waiting 1 second = 1000 milliseconds
-->
<html>
<head>
  <title>Refresh or stay</title>
  <meta http-equiv="refresh" content="1; URL=http://www.eve.ch/yes.htm">
</head>
<body bgcolor="#ffffff">
</body>
</html>
```

Default character set: Every HTML page can specify the character set that the browser should use to display the page. In the absence of this information, the browser uses a default character set.

- *public static String getDefaultCharacterSet()*: Returns the default character set that the browser will use for HTML pages that specify no character set.
- *public static void setDefaultCharacterSet(String characterSet)*: Lets you set the standard character set.
- *public static void resetDefaultCharacterSet()*: If the default character set was previously changed, this method sets it back to "iso-8859-1".

```
<!-- HTML code with specification of character set -->
<html>
<head>
  <title>Refresh or stay</title>
  <meta http-equiv="content-type" content="text/html;charset=ISO-8859-1">
</head>
<body bgcolor="#ffffff">
</body>
</html>
```

Treating images as text: In an HTML file, an image can have an *alt* attribute that describes the image in text form. HttpUnit can use such images as normal texts. (See also, in section *3.4.4, Hyperlinks (class "WebLink")*, the methods *WebResponse.getLinkWith(...)* and *WebResponse.getLinkWithImageText(...)*).

- *public static boolean getImagesTreatedAsAltText()*: Returns *true* if images are to handled like text (otherwise *false* = default); the text for each image is its *alt* attribute.
- *public static void setImagesTreatedAsAltText(boolean asText)*: Activate treatment of images as text (= *true*) or deactivate it (= *false* = default).

```
<!-- Image with "alt" attribute -->
<a href="home/privateind.htm"><img src="img/head.gif" alt="Private
individual"></a>
```

Uppercase and lowercase for names and IDs: In table, form, and frame names, uppercase and lowercase are normally irrelevant and you can thus ignore this issue. (Warning: In parameters within forms, uppercase and lowercase are *always* relevant):

- *public static boolean getMatchesIgnoreCase()*: Returns *true* (= default) if HttpUnit *should not* pay attention to uppercase and lowercase during entry of names and IDs (otherwise *false*; see *WebResponse* methods such as *getLinkWith(...)*, *getFormWithID(...)*, *get-FormWithName(...)*, *getTableWithID(...)*, and *getTableWithSummary(...)*).
- *public static void setMatchesIgnoreCase(boolean ignoreCase)*: Turns sensitivity to uppercase and lowercase on (= *true*) or off (= *false*); see previous method.

Validation when setting form parameters: Normally, HttpUnit checks efforts to set form parameters for permissibility and, in case of an error, throws one of the following exceptions: *IllegalFileParameterException*, *NoSuchParameterException*, or *IllegalParameterValueException*.

- *public static boolean getParameterValuesValidated()*: Returns *true* (= default), if HttpUnit is to check the form parameter inserted by the method *WebRequest.setParameter(...)* and, in case of error, throw exceptions (otherwise *false*).
- *public static void setParameterValuesValidated(boolean validated)*: Lets you activate or deactivate the checking mode.

Display of error messages when analyzing HTML pages: When analyzing (= parsing) HTML code, HttpUnit may encounter errors that can lead to an unusable result. Because HttpUnit reports errors, you easily correct HTML problems. (Judging by the error messages, all the HTML pages that we've examined abound with objectionable elements; but this normally doesn't keep them from working.)

- *public static boolean getParserWarningsEnabled()*: Returns *true* if HttpUnit is to display on the screen the errors it encounters when analyzing HTML code (otherwise *false* = default).
- *public static void setParserWarningsEnabled(boolean enabled)*: Lets you activate or deactivate error reports.

Example of an error report:

```
line 12 column 1 - Warning: <table> lacks "summary" attribute
line 14 column 20 - Warning: <img> lacks "alt" attribute
line 29 column 65 - Warning: trimming empty <font>
line 105 column 92 - Warning: discarding unexpected </a>
line 108 column 74 - Warning: trimming empty <b>
line 112 column 65 - Warning: missing </font> before </td>
line 113 column 54 - Warning: unescaped & or unknown entity "&country"
```

Display of header information: When you are trying to track down an error, it can be helpful to see the header information of an HTTP request on the computer screen.

- *public static boolean isLoggingHttpHeaders()*: Returns *true* if the HTTP header is to appear on the screen (otherwise *false* = default).
- *public static void setLoggingHttpHeaders(boolean enabled)*: Lets you activate or deactivate the screen display.

An example of displayed header information:

```
Header:: HTTP/1.1 200 OK
Header:: Date: Thu, 06 Sep 2001 13:02:27 GMT
Header:: Server: Apache/1.3.6 (Unix)
Header:: Last-Modified: Thu, 30 Aug 2001 06:44:32 GMT
Header:: ETag: "35215-15333-3b8de0d0"
Header:: Accept-Ranges: bytes
Header:: Content-Length: 86835
Header:: Connection: close
Header:: Content-Type: text/html
```

3.6 Access via a proxy server

As is well-known, at Winterthur Insurance, Internet communication occurs via a proxy server ("proxy001.winterthur.ch", Port 8080).

To communicate via this proxy server when accessing the *Internet*, you must give the proxy server a personal user ID and password (for example "c123456" and "solo").

In contrast, for local *intranet* access, you should *not* use the proxy server (which means you don't have to provide a user ID or password).

For problem-free web access via the proxy server, various properties for the system and communications must be set. A class we have written, *EveWebBrowser*, completely takes over this task (see section *3.4.1, Virtual browser (class "EveWebBrowser")*).

3.7 The protocols HTTP and HTTPS

HttpUnit supports the protocols HTTP and HTTPS (= access to a secure web server[20]). Use of HTTPS requires that a certificate be present on the client. Fortunately, when you access the URL "https://entry.winterthur.com/..."[21], the system uses a standard certificate from the company VeriSign, so you don't need to worry about a certificate.[22]

Important: Depending on which protocol you are using, a test class that uses HttpUnit must have the following projects in its classpath:

Protocol	Projects in classpath
HTTP	*Eve HttpUnit*
HTTPS	*Eve HttpUnit* and *Eve JSSE*[23]. Establishing a connection automatically activates a number of classes that help make the connection secure. (This is an automatic process that you don't need to worry about.)

Section *2.4.2, Setting the class path for the general TestRunner class*, describes how you can set the class path within VisualAge.

[20] Various approaches allow secure transactions on the web. Today, Secure Sockets Layer (SSL) is the main technology.

[21]A number of applications that iSolutions have developed and that are now in live operation, including *wincoLink, OGL* and *IQS*, are addressed via this URL.

[22] All certificates are registered in the file *cacerts*. On every client, this file is located in a directory of the Jave runtime environment (for example "c:\dev\visualage for java\ide\program\lib\security\" or "c:\jdk1.3\jre\lib\security\"). You can write a pseudo-certificate and register it in the file *cacerts*. See "http://httpunit.sourceforge.net/doc/sslfaq.html" for a full description.

[23] JSSE stands for *Java Secure Socket Extension*. This extension consists of a collection of packages and permits encrypted communication (via the protocol HTTPS). According to Sun, JSSE will become an integral part of Java 1.4 (probably in fall 2001). As soon as VisualAge offers Java 1.4, the project *Eve JSSE* will become redundant and we will remove it.

3.8 Possible errors while establishing connection and reading data

During attempts to access web pages by using HttpUnit, various errors may appear and various exceptions may be thrown. The following example shows the establishment of a connection and the display of an error message:

```
try {
  EveWebConversation browser = new EveWebConversation("user", "password");
  WebRequest req = new GetMethodWebRequest("http://where.are.you");
  WebResponse resp = browser.getResponse(req);
  ...
} catch (Exception e) {
  System.err.println("Exception: " + e);
}
```

A resulting error message could, for example, look as follows:

```
Exception: ... Error on HTTP request: 404 [http://where.are.you]
```

Within the error message, the HTTP status code (= 404 in this example) is of particular interest, because it indicates why HttpUnit couldn't make the connection or read the data.

The following table presents the most important status codes (see also [Hunter2001], page 685 and following).

HTTP status code	Possible reason for error

404 (Bad Request)	Internet access: You have tried to access the Internet without a user name and password (which you must supply, since access is only via proxy server).
	
	`//-- Wrong (Access attempt without user name and password)`
	`EveWebBrowser browser = new EveWebBrowser();`
	
	`//-- Correct (Access attempt with user name and password)`
	`EveWebBrowser browser = new EveWebBrowser("user", "password");`
	
	If you supplied the correct user name and password, then the URL is probably wrong (see also errors 500 and 503).
	
	Or you could be trying to access a web page via HTTPS without a certificate (see also section *3.7, The protocols HTTP and HTTPS*).

407 (Proxy Authentication Required)	Internet access: User name or password is wrong.
	
	`//-- Wrong (wrong user name and password)`
	`EveWebBrowser browser =`
	`  new EveWebBrowser("wrongUser", "wrongPassword");`
	
	`//-- Correct (correct user name and password)`
	`EveWebBrowser browser = new EveWebBrowser("user", "password");`

500 (Internal Server Error)	Access using a user name and password; the URL is probably wrong (see also errors 404 and 503).
	
	`//-- Wrong (URL does not exist)`
	`EveWebBrowser browser = new EveWebBrowser("user", "password");`
	`WebRequest request =`
	`  new GetMethodWebRequest("http://where.are.you");`

HTTP status code	**Possible reason for error**

503 (Service Unavailable)	Local *intranet* access: You are trying to access the local intranet by using a user name and password (this corresponds to access via proxy server, which is impossible in the case of the intranet).
	
	`//-- Wrong (Access attempt with user name and password)`
	`EveWebBrowser browser = new EveWebBrowser("user", "password");`
	`//-- Correct (Access attempt without user name and password)`
	`EveWebBrowser browser = new EveWebConversation();`
	
	If you didn't supply a user name and password, then the URL is probably wrong (see also errors 404 and 500).

3.9 Known problems in HttpUnit

While trying out HttpUnit, we noticed various problems; we will describe them in the following sections.

3.9.1 *Post* requests

Currently, the servlet engine *ServletExec 2.2* is in use on various servers (both test and live operations) at iSolution. This server is unable to correctly process *post* requests sent by HttpUnit. (Specifically, the server doesn't recognize the form parameters that come with a request. This problem doesn't exist with *get* requests.) In version 3.1 and later of *ServletExec*, however, this problem is gone.[24] Despite this blemish, you can already start using HttpUnit today, as follows:

- **Local environment:** Using HttpUnit, you can easily test servlet applications on your local computer (*http://localhost...*), but version 3.1 or above of the ServletExec debugger must be installed.
- **Test and production environments:** Until *ServletExec* in Version 3.1 or higher is available on the iSolution test and operational servers, HttpUnit will remain of only limited use in the testing of test and operational applications. (You can send *get* requests, but not *post* requests.)

3.9.2 JavaScript

HttpUnit can't deal with *JavaScript*. Should an HTML button try to make a server request via *JavaScript*, HttpUnit throws the exception *java.net.MalformedURLException*:

```
try {
  EveWebBrowser browser = new EveWebBrowser();

  //-- Trigger the request, receive the response
  WebRequest req = new GetMethodWebRequest("http://www.winterthur.com");
```

[24] According to Russell Gold, the servlet engine *Tomcat 3.0* has this same problem. (In *Tomcat 3.2* and higher, this problem is gone.)

```
      WebResponse resp = browser.getResponse(req);

      //-- Find link
      WebLink link = resp.getLinkWith("take me there");

      //-- Trigger a new request, receive the response
      req = link.getRequest();
      resp = browser.getResponse(req);
    } catch (MalformedURLException e) {
      System.out.println("Exception: " + e.getMessage());
    }
```

The above program creates the message "Exception: unknown protocol: javascript". This is certainly unpleasant, given that the server applications that we've developed use *JavaScript* in many places. Russell Gold is aware of this problem and promises corresponding extensions in the future.[25]

3.9.3 Sloppily-coded HTML forms

In an HTML document, when a form contains several fields (parameters) with identical names, the fields cannot be correctly queried or set. The following form[26] contains several fields with the identical name *symbol.* (This is an example of a sloppily-coded HTML form.) In this case, using HttpUnit, you can only read or fill in one *symbol* field:

```
<!-- HTML code -->
<form name="frmQuotes" method=get action="http://quotes.nasdaq.com/Quote.dll">
  <td><input maxlength="10" name="symbol" type="text" size="6"></td>
  <td><input maxlength="10" name="symbol" type="text" size="6"></td>
  <td><input maxlength="10" name="symbol" type="text" size="6"></td>
...
</form>

//-- Java code
EveWebBrowser browser = new EveWebBrowser("c156817", "igelst");

//-- Transmit request and store response
WebRequest req = new GetMethodWebRequest("http://www.nasdaq.com");
WebResponse resp = browser.getResponse(req);

//-- Retrieve form and determine request
WebForm form = resp.getFormWithName("frmQuotes");
req = form.getRequest("quick");

//-- Set a form parameter, trigger the request, and store the response
req.setParameter("symbol", "snic"); // snic = Sonic Solutions
resp = browser.getResponse(req);
```

3.9.4 HTML documents that cannot be processed

The code example we just saw assigns the stock price (USD 1.46) of "Sonic Solutions" to a JavaScript variable. HttpUnit is therefore unable to read it:

---

[25] See *http://httpunit.sourceforge.net/doc/faq.html#javascript*.

[26] See *http://www.nasdaq.com*.

```
//-- Java code
req.setParameter("symbol", "snic"); // snic = Sonic Solutions
resp = browser.getResponse(req);


<!-- HTML code that "resp" delivers -->
...
var data=[[0,"SNIC",19,"$&nbsp;1.46",-1,"0.17","10.43%","21,900","N"]];
...
```

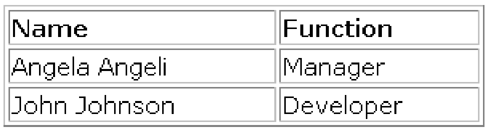
With the following proposed solutions – which are admittedly hard to implement – you can access the desired information:

- **HTML code:** The method *resp.getText()* gives you access to the entire delivered HTML code. You can analyse this HTML code whenever you want and in this way obtain the information you need.
- **DOM tree:** The method *resp.getDOM()* gives you a DOM tree (DOM = *Document Object Model)* representing the HTML document; you can navigate through it as you please. (See also the project *W3C DOM*, which section *3.1, Making HttpUnit available* describes briefly.) The DOM tree can possibly lead you to the data you want.

3.10 The interplay betwee n JUnit and HttpUnit

JUnit and HttpUnit work well together when the goal is to test dynamic web applications. HttpUnit navigates through the application and obtains various values in the HTML pages; JUnit then checks these values.

As an example, let's check the size and contents of the following HTML table:



Name	Function
Angela Angeli	Manager
John Johnson	Developer

An appropriate JUnit test class could look as follows:

```
...
public class PersDataTest extends TestCase {
  ...
  public void testTableContent () {
    ...

    //-- Retrieve table and check size
    WebTable table = ...
    assertEquals("Wrong number of rows", 3, table.getRowCount());
    assertEquals("Wrong number of columns", 2, table.getColumnCount());

    //-- Check the content of the table
    assertEquals("Name", table.getCellAsText(0, 0));
    assertEquals("Function", table.getCellAsText(0, 1));

    assertEquals("Angela Angeli", table.getCellAsText(1, 0));
    assertEquals("Manager", table.getCellAsText(1, 1));

    assertEquals("John Johnson", table.getCellAsText(2, 0));
    assertEquals("Developer", table.getCellAsText(2, 1));
  }
}
```

3.11 Application area for HttpUnit

We are fully aware that writing HttpUnit can be laborious and that using them to test large applications would probably turn into dog work.

Small applications: In our view, HttpUnit represents a practical toolbox that offers a great way to write automated tests for small applications or to tap into web services (such as current stock-market prices and currency rates).

Large applications: HttpUnit is not as well suited, however, to testing large applications (such as *wincoLink*, *OGL* or *IQS*). Here, we recommend the use of a test tool that "remote controls" our web applications with the help of scripts and normal web browsers.[27]

[27] iSolutions plans to evaluate a test tool.

4. For further reading

[AmbySoft2000]	Scott W. Ambler: *Writing Robust Java Code;* 2000 (see *http://www.ambysoft.com/javaCodingStandards.pdf*).
[Beck2000]	K. Beck: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.
[e-Platform1999]	e-Platform-Team: *Coding Guidelines*; Winterthur Insurance, 1999 (see *http://c004020/ECPReference/jackpot/Codierungsrichtlinien/Codierungsrichtlinien.html*).
[EveAppl2001]	iSolutions-Team: *Eve – The application framework.* Winterthur Life, 2001.
[EveProg2001]	iSolutions-Team: *Eve – Guidelines for Java Programming.* Winterthur Life, 2001.
[Fowler1999]	M. Fowler: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[Gamma1995]	E. Gamma et al: *Design patterns;* Addison-Wesley, 1995.
[HttpUnit]	R. Gold: *HttpUnit* (see *http://httpunit.sourceforge.net*).
[Hunter2001]	J. Hunter, W. Crawford: *Java Servlet Programming;* O'Reilly, 2001.
[JUnit]	K. Beck, E. Gamma: *JUnit, Testing Resources for Extreme Programming;* JUnit.org (see *http://www.junit.org*).
[Maguire1993]	S. Maguire: *Writing Solid Code.* Microsoft Press, 1993.
[Musc2000]	Ch. Musciano, B. Kennedy: *HTML & XHTML – The Definitive Guide;* O'Reilly, 2000.
[RoqueWave2000]	Rogue-Wave-Team: *The Elements of Java Style*; Cambridge University Press, 2000.
[Sun1999]	Sun-Team: *Code Conventions for the Java Programming Language;* Sun Microsystems, 1999 (see *http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html*).