Unit Testing in Java: How Tests Drive the Code
by Johannes Link, Peter Frolich (Contributor), Peter Frohlich, Peter Fröhlich
Publisher: Morgan Kaufmann; (April 2003)
ISBN: 1558608680

Chapter 13

Graphical User Interfaces

Readers who have gotten this far probably noticed that the "special cases" discussed in the second part of this book are nothing more than the same issue in recurring variations: the structure or interface design of a ready-to-use framework, a tool, or an API hampers the testing of the actual functionality.

Unclear testing criteria

Graphical user interfaces (GUIs) are no exception in this respect, because their implementation relies on the use of AWT or Swing, as well as additional components (e.g., Java2D). This situation is topped by an important aspect we already saw in connection with Web applications: A "good" user interface not only has to have clearly testable properties, it is also subject to unclear evaluation criteria, such as ergonomics, intuitive use, and esthetics, or in short *usability* [Nielsen94]. In contrast to what's required by the test-first approach, these criteria cannot be specified in advance. For this reason, it is not surprising that the test-first development of a GUI cannot cover all desirable aspects.

Nevertheless, we as test-first representatives are not left empty-handed in this issue either, as we will see in this chapter.

13.1 The Direct Way

The first idea that a deeply rooted test-first enthusiast will naturally follow is the direct way: why not develop a GUI exactly like all other classes, namely in small steps and with corresponding test cases before writing the

application code? Many test-first critics are doubtful about this—a good reason to prove the opposite in a detailed example.

We will use the development of a simple user interface for a product catalog as our case study. The product catalog itself can be addressed over the following interface:

Domain model

```
public interface ProductCatalog {
   void addProduct(Product product);
   void removeProduct(Product product);
   Set getProducts();
   Set getAvailableCategories();
}
```

This interface is slim and allows us to add and remove a product and to output all products and all permitted product categories. This means that, to develop the GUI, we don't need a fully functioning implementation, including persistence and other trinkets. In fact, the simplest conceivable implementation will be sufficient, and with `SimpleProductCatalog`, we have it already (in the codebase on the Web site). But there is still a dependence upon the Product class and the product Category class:

```
public class Product {
   public Product(String pid) {...}
   public String getPID() {...}
   public Category getCategory() {...}
   public void setCategory(Category category) {...}
   public String getDescription() {...}
   public void setDescription(String description) {...}
}

public class Category {
   public Category(String name) {...}
   public String getName() {...}
}
```

Separating the logic from the interface

Encapsulating the entire specialist logic in a separate domain model—ideally in an interface—represents important heuristics: the GUI classes should contain as little logic as possible. This means that the correct coupling between user interface and specialist logic is the only functionality

that remains to be tested. In addition, encapsulation of the logic in interfaces rather than in classes contributes to independence and facilitates testing. Thus, improved testability is an additional argument in favor of using the MVC pattern (see also Chapter 12, Section 12.5).

Figure 13.1 shows the design of our graphical user interface for adding and deleting products. Our goal is to create a product editor (`ProductEditor`) and to ensure the following things:

Testing goals

- All important elements of the interface are in place, and they are correctly initialized and visible.
- All products available from the catalog are displayed in the list.
- Selecting a product from the list and subsequently clicking on *Delete* removes the product from the catalog and from the list.
- Clicking *Add* opens a dialog for the user to add a product.

The exact pixel representation of the interface is not to be tested. Although this would be possible, experience has shown that the layout of an interface is subject to frequent changes. For this reason, the test cases targeted to details would have to be modified often, such that it wouldn't be worth automating them. A visual inspection of the interface within the acceptance tests is better suited to this task.

Creation testing

But let's get down to work now. In the first step, we want to test that the editor is properly created and displayed:

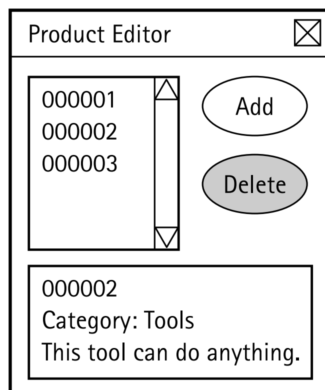

Figure 13.1 The desired `Product Editor` layout.

```
public class CatalogEditorTest extends TestCase {
   public void testCreation() {
      CatalogEditor editor = new CatalogEditor();
      editor.show();
      assertTrue(editor.isShowing());
      assertEquals("Product Editor", editor.getTitle());
   }
}
```

Our implementation is clearly shorter than the test code:

```
import javax.swing.*;
public class CatalogEditor extends JFrame {
   public CatalogEditor() {
      super("Product Editor");
   }
}
```

One reviewer commented here that the lines,

```
editor.show();
assertTrue(editor.isShowing());
```

check for nothing else but correct functioning of the class `javax.swing.JFrame`. This is correct, but these lines serve a purpose: they enable us to derive the editor from `JFrame`. Someone with no idea about Swing would not (be able to) opt for this solution. This shows once more that a known goal, namely to create a Swing application, influences both the design and the implementation.

When running the first test, we notice a small flaw: the editor window remains open after the test. So let's add a `dispose()` to the test and make sure that it's called even when the test fails:

```
   public void testCreation() {
      try {
         CatalogEditor editor = new CatalogEditor();
         editor.show();
         assertTrue(editor.isShowing());
         assertEquals("Product Editor", editor.getTitle());
```

```
    } finally {
        editor.dispose();
    }
}
```

Display testing

Next, we will deal with testing the product list, including the known refactoring in `setUp()` and `tearDown()`:

```
public class CatalogEditorTest extends TestCase {
    ...
    private CatalogEditor editor;
    protected void setUp() {
        editor = new CatalogEditor();
        editor.show();
    }
    protected void tearDown() {
        editor.dispose();
    }

    public void testCreation() {
        assertTrue(editor.isShowing());
        assertEquals("Product Editor", editor.getTitle());
    }

    public void testProductList() {
        assertTrue(editor.productList.isShowing());
    }
}
```

To obtain access to the list widget `productList` within the test, we have to make it available in an instance variable, either *protected* or *package scope*. As an alternative, we could search the component tree for the matching widget. We will see this approach later when testing with JFC-Unit (see Section 13.2).

The implementation is straightforward, totally ignoring the esthetics of the visible layout:

```
public class CatalogEditor extends JFrame {
    JList productList = new JList();
    public CatalogEditor() {
```

```
      super("Product Editor");
      getContentPane().add(productList);
   }
}
```

Link testing

Now let's test whether the product list really fills with products from the catalog. We need to make a design decision for this purpose: a `ProductCatalog` instance is passed to the editor within the constructor. The simple `SimpleProductCatalog` will do for testing here:

```
public class CatalogEditorTest extends TestCase {
   ...
   private ProductCatalog catalog;

   protected void setUp() {
      catalog = new SimpleProductCatalog();
      catalog.addProduct(new Product("123456"));
      catalog.addProduct(new Product("654321"));
      editor = new CatalogEditor(catalog);
      editor.show();
   }

   public void testProductList() {
      assertTrue(editor.productList.isShowing());
      ListModel model =
         (ListModel) editor.productList.getModel();
      assertEquals(2, model.getSize());
   }
}
```

Testing for `getSize()` alone is relatively weak. We could think of a helper method, which compares the content of a `ListModel` with the content of a `Collection`. For the moment, however, we are brave and contented with simple things, leading to the following implementation:

```
public class CatalogEditor extends JFrame {
   JList productList;
   public CatalogEditor(ProductCatalog catalog) {
      super("Product Editor");
```

```
      productList =
         new JList(catalog.getProducts().toArray());
      getContentPane().add(productList);
   }
}
```

This test tells us nothing about how the single products should be represented. And the implementation relies entirely on the `toString()` method in `Product`. By the way, this is a general weakness of the approach described here: we are merely testing the model of the graphical components and not what is really displayed. The reason is that the Swing widgets (i.e., the `JList` class in this case) don't let us access the actual representation.

Delete button testing

Let's see how products are deleted, the Delete button test is easy:

```
public class CatalogEditorTest extends TestCase {
   ...

   public void testDeleteButton() {
      assertTrue(editor.deleteButton.isShowing());
      assertEquals("Delete", editor.deleteButton.getText());
   }
}
```

and so is the implementation:

```
public class CatalogEditor extends JFrame {
   ...
   JButton deleteButton;

   public CatalogEditor(ProductCatalog catalog) {
      super("Product Editor");
      productList =
         new JList(catalog.getProducts().toArray());
      getContentPane().add(productList);
      deleteButton = new JButton("Delete");
      getContentPane().add(deleteButton);
   }
}
```

Delete function testing

For this reason, we want to proceed to the actual Delete function, or actually to the corresponding test:

```
public class CatalogEditorTest extends TestCase {
   ...

   private ListModel getListModel() {
      return (ListModel) editor.productList.getModel();
   }

   public void testDeleteProduct() {
      editor.productList.setSelectedIndex(0);
      editor.deleteButton.doClick();
      assertEquals(1, getListModel().getSize());
      assertEquals(new Product("654321"),
         getListModel().getElementAt(0));
      assertEquals(1, catalog.getProducts().size());
      assertTrue(catalog.getProducts().contains(
         new Product("654321")));
   }
}
```

Note that both the `ListModel` and the catalog itself have to be tested, where the number of products alone is not sufficient this time. The test in this form only makes sense provided that `Product.equals()` has been previously implemented, namely, when the equality of two products is determined by their product IDs.

We quickly dare a first implementation attempt:

```
public class CatalogEditor extends JFrame
   implements ActionListener {
   ...
   private ProductCatalog catalog;

   public CatalogEditor(ProductCatalog catalog) {
      super("Product Editor");
      this.catalog = catalog;
      productList =
         new JList(catalog.getProducts().toArray());
      getContentPane().add(productList);
```

```
      deleteButton = new JButton("Delete");
      deleteButton.addActionListener(this);
      getContentPane().add(deleteButton);
   }

   public void actionPerformed(ActionEvent e) {
      deleteButtonClicked();
   }
   private void deleteButtonClicked() {
      Product toDelete =
         (Product) productList.getSelectedValue();
      catalog.removeProduct(toDelete);
      productList.setListData(
         catalog.getProducts().toArray());
   }
}
```

This solution gets by without separate implementation of the `ListModel` interface. But the experienced Swing programmer's hair stands on end; he or she wants a "neat and tidy" `ListModel`. We want that too, but because the code will *communicate* better, rather than because it will make the code shorter.

ListModel introduction

For this reason, we will turn our attention away from the editor and concentrate on a `ProductsListModel` instead. And because we are right in the swing of it,[1] let's take bigger steps. Our first test focuses on creating things:

```
public class ProductsListModelTest extends TestCase {
   public void testCreation() {
      ProductCatalog catalog = new SimpleProductCatalog();
      catalog.addProduct(new Product("123456"));
      catalog.addProduct(new Product("654321"));
      ProductsListModel model =new ProductsListModel(catalog);
      assertEquals(2, model.getSize());
      assertEquals(new Product("123456"), model.getElementAt(0));
      assertEquals(new Product("654321"), model.getElementAt(1));
   }
}
```

1. `countPuns++`

Sort order testing

Considering that the test requires lexicographic sorting of the products, while the catalog supplies an (unordered) `Set`, we realize that everything is harder than expected:

```
public class ProductsListModel extends AbstractListModel {
   private ProductCatalog catalog;
   private List sortedProducts;

   public ProductsListModel(ProductCatalog catalog) {
      this.catalog = catalog;
      sortedProducts = new ArrayList(catalog.getProducts());
      Collections.sort(sortedProducts, new Comparator() {
         public int compare(Object o1, Object o2) {
            Product p1 = (Product) o1;
            Product p2 = (Product) o2;
            return p1.getPID().compareTo(p2.getPID());
         }
      });
   }

   public Object getElementAt(int index) {
      return sortedProducts.get(index);
   }

   public int getSize() {
      return sortedProducts.size();
   }
}
```

Deleting from the model

Our next testing step is targeted to delete products from `ListModel`:

```
public class ProductsListModelTest extends TestCase {
   ...

   private ProductCatalog catalog;
   private ProductsListModel model;
   protected void setUp() {
      catalog = new SimpleProductCatalog();
      catalog.addProduct(new Product("123456"));
      catalog.addProduct(new Product("654321"));
```

```
      model = new ProductsListModel(catalog);
   }

   public void testDeleteProduct() {
      model.deleteProduct(0);
      assertEquals(1, model.getSize());
      assertEquals(new Product("654321"),model.getElementAt(0));
      assertEquals(1, catalog.getProducts().size());
      assertTrue(catalog.getProducts().contains(
         new Product("654321")));
   }
}
```

It is no coincidence that this test case is very similar to the delete test in `CatalogEditorTest`. It should occasionally cause us to consider whether or not the old test has become pointless by the new one. The interface decision was taken in favor of `delete(int index)`, because we can easily determine the selected index from `JList`. This makes the implementation pretty easy:

```
public class ProductsListModel extends AbstractListModel {
   ...

   public void deleteProduct(int index) {
      Product toDelete = (Product) sortedProducts.get(index);
      sortedProducts.remove(toDelete);
      catalog.removeProduct(toDelete);
   }
}
```

Embedding the model

And now back to the editor to embed the `ProductListModel`:

```
public class CatalogEditor extends JFrame implements ActionListener {
   ...

   public CatalogEditor(ProductCatalog catalog) {
      super("Product Editor");
      ProductsListModel model =
         new ProductsListModel(catalog);
```

```
      productList = new JList(model);
      ...
   }

   public void actionPerformed(ActionEvent e) {
      deleteButtonClicked();
   }
   private void deleteButtonClicked() {
      int deleteIndex = productList.getSelectedIndex();
      ((ProductsListModel) productList.
        getModel()).deleteProduct(deleteIndex);
   }
}
```

Isn't this wonderful? All tests are running! It's about time we take a look at all of this "for real." All we need first is a `main()` method and a rudimentary layout:

```
public class CatalogEditor extends JFrame implements ActionListener {
   ...

   public CatalogEditor(ProductCatalog catalog) {
      super("Product Editor");
      createWidgets(catalog);
   }

   private void createWidgets(ProductCatalog catalog) {
      getContentPane().setLayout(new FlowLayout());
      setSize(150, 150);
      ProductsListModel model = new ProductsListModel(catalog);
      productList = new JList(model);
      getContentPane().add(productList);
      deleteButton = new JButton("Delete");
      deleteButton.addActionListener(this);
      getContentPane().add(deleteButton);
   }

   public static void main(String[] args) {
      SimpleProductCatalog catalog = new SimpleProductCatalog();
      catalog.addProduct(new Product("1000001"));
```

```
        catalog.addProduct(new Product("2000002"));
        CatalogEditor editor = new CatalogEditor(catalog);
        editor.show();
    }
}
```

Unfortunately, visual testing has an unpleasant surprise in store for us: Although all tests are running, clicking the Delete button once does not make the selected product disappear! How could this happen?

Liar View bug pattern

The answer is that we were taken in by the Liar View bug pattern [Allen01]: although the model behind `JList` is updated, the view itself is not informed about it. The correct thing to do would be to inform all `ListDataListeners` registered with `ProductsListModel` about a successful deletion. So let's extend a corresponding test case:

```
import javax.swing.event.ListDataEvent;
public class ProductsListModelTest extends TestCase {
    ...
    private boolean intervalRemovedCalled = false;
    public void testDeleteProduct() {

        ListDataListener listener = new ListDataListener() {
            public void intervalAdded(ListDataEvent e) {}
            public void intervalRemoved(ListDataEvent e) {
                intervalRemovedCalled = true;
            }
            public void contentsChanged(ListDataEvent e) {}
        };

        model.addListDataListener(listener);
        model.deleteProduct(0);
        assertTrue(intervalRemovedCalled);
        assertEquals(1, model.getSize());
        assertEquals(new Product("654321"),model.getElementAt(0));
        assertEquals(1, catalog.getProducts().size());
        assertTrue(catalog.getProducts().contains(
            new Product("654321")));
    }
}
```

Similar to examples in other chapters, we are using a *poor-house mock object,* that is, an anonymous class. The implementation needs one small addition:

```
public class ProductsListModel extends AbstractListModel {
   ...

   public void deleteProduct(int index) {
      Product toDelete =
         (Product) sortedProducts.get(index);
      sortedProducts.remove(toDelete);
      catalog.removeProduct(toDelete);
      fireIntervalRemoved(this, index, index);
   }
}
```

Now, the visual test works as expected. And just to make sure nobody can say we are careless, let's see a few functionality chunks in express testing.

Correct Enable status

Chunk 1: Test to ensure that the Delete button is *enabled* only when a product is selected:

```
   public void testDeleteButton() {
      assertTrue(editor.deleteButton.isShowing());
      assertEquals("Delete", editor.deleteButton.getText());
      assertFalse(editor.deleteButton.isEnabled());
      editor.productList.setSelectedIndex(0);
      assertTrue(editor.deleteButton.isEnabled());
      editor.productList.clearSelection();
      assertFalse(editor.deleteButton.isEnabled());
   }
```

Product details

Chunk 2: Test to ensure that the product detail view works correctly:

```
   public void testProductDetails() throws Exception {
      assertTrue(editor.productDetails.isShowing());
      assertEquals("", editor.productDetails.getText());
      Product product =
         (Product) getListModel().getElementAt(0);
```

```
        product.setCategory(new Category("Records"));
        product.setDescription("The best of Kenny Haye");
        editor.productList.setSelectedIndex(0);
        assertProductDetails(product);
        editor.productList.setSelectedIndex(1);
        assertProductDetails(
            (Product) getListModel().getElementAt(1));
        editor.productList.clearSelection();
        assertEquals("", editor.productDetails.getText());
        editor.productList.setSelectedIndex(0);
        assertProductDetails(product);
    }
    private void assertProductDetails(Product product)
        throws IOException {
        BufferedReader reader = new BufferedReader(
            new StringReader(editor.productDetails.getText()));
        assertEquals("PID: " + product.getPID(),
                     reader.readLine());
        assertEquals("Category: " +
                     product.getCategory().getName(),
                     reader.readLine());
        assertEquals(product.getDescription(),
                     reader.readLine());
        assertNull(reader.readLine());
    }
```

This test case already considers deselection of a product and sequential selection of multiple products. The test in this form hasn't been built in one go, but piece by piece. It also shows that visually controlling the components from time to time can be useful to find new test cases.

Let's have another look at the current implementation state of `CatalogEditor`:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class CatalogEditor extends JFrame
         implements ActionListener, ListSelectionListener {
```

```
JList productList;
JButton deleteButton;
JTextArea productDetails;
public CatalogEditor(ProductCatalog catalog) {
   super("Product Editor");
   createWidgets(catalog);
}
private void createWidgets(ProductCatalog catalog) {
   getContentPane().setLayout(new FlowLayout());
   setSize(150, 150);
   addProductList(catalog);
   addProductDetails();
   addDeleteButton();
}
private void addProductList(ProductCatalog catalog) {
   ProductsListModel model =
      new ProductsListModel(catalog);
   productList = new JList(model);
   productList.setSelectionMode(
      ListSelectionModel.SINGLE_SELECTION);
   productList.addListSelectionListener(this);
   getContentPane().add(productList);
}
private void addProductDetails() {
   productDetails = new JTextArea();
   getContentPane().add(productDetails);
}
private void addDeleteButton() {
   deleteButton = new JButton("Delete");
   deleteButton.setEnabled(false);
   deleteButton.addActionListener(this);
   getContentPane().add(deleteButton);
}
public void actionPerformed(ActionEvent e) {
   deleteButtonClicked();
}
private void deleteButtonClicked() {
```

```
        int deleteIndex = productList.getSelectedIndex();
        ((ProductsListModel) productList.getModel()).
            deleteProduct(deleteIndex);
    }
    public void valueChanged(ListSelectionEvent e) {
        if (productList.getSelectedValue() == null) {
            clearProductSelection();
        } else {
            selectProduct(
                (Product) productList.getSelectedValue());
        }
    }
    private void clearProductSelection() {
        deleteButton.setEnabled(false);
        productDetails.setText("");
    }
    private void selectProduct(Product product) {
        deleteButton.setEnabled(true);
        productDetails.setText("");
        productDetails.append("PID: " +
            product.getPID() + "\n");
        productDetails.append("Category: " +
            product.getCategory().getName() + "\n");
        productDetails.append(
            product.getDescription() + "\n");
    }
}
```

We find nothing that could be a big surprise for Swing programmers, perhaps with the only exception of gradually losing patience as the layout still leaves a great deal to be desired. However, one important piece of functionality is still missing: the Add button. First, the button itself in a trivial test:

```
public void testAddButton() {
    assertTrue(editor.addButton.isShowing());
    assertEquals("Add", editor.addButton.getText());
}
```

Opening a dialog

And now let's see what's behind it. The problem we have to deal with here is that another dialog should open for the user to enter new product data. But how can we test such a thing?

Here is our answer: not at all in the test suite for `CatalogEditor`! In this case, we are only interested in having the editor somehow get a hold of a new product instance:

```
public void testAddProduct() {
   editor.setProductCreator(new ProductCreator() {
      public Product create() {
         return new Product("333333");
      }
   });

   editor.addButton.doClick();
   assertEquals(3, getListModel().getSize());
   assertEquals(new Product("333333"),
      getListModel().getElementAt(1));
   assertEquals(3, catalog.getProducts().size());
   assertTrue(catalog.getProducts().contains(
      new Product("333333")));
}
```

This approach deserves a closer look. We have decided to move the creation of a new product to a `ProductCreator` interface. It allows us to use a dummy implementation for testing purposes. Another trick is the PID we have actually chosen, which means that we concurrently test for correct sorting of the list.

To keep it happy, the test requires a considerable amount of code:

```
public class CatalogEditor extends JFrame
   implements ActionListener, ListSelectionListener {
   ...
   JButton addButton;
   private ProductCreator productCreator;
   private void createWidgets(ProductCatalog catalog) {
      ...
      addAddButton();
   }
   public void actionPerformed(ActionEvent e) {
```

```
      if (e.getSource() == deleteButton) {
         deleteButtonClicked();
      } else {
         addButtonClicked();
      }
   }
   private ProductsListModel getProductsListModel() {
      return ((ProductsListModel) productList.getModel());
   }
   private void addAddButton() {
      addButton = new JButton("Add");
      addButton.addActionListener(this);
      getContentPane().add(addButton);
   }
   private void addButtonClicked() {
      Product newProduct = productCreator.create();
      getProductsListModel().addProduct(newProduct);
   }
   protected void setProductCreator(
      ProductCreator creator) {
      productCreator = creator;
   }
}
public class ProductsListModel extends AbstractListModel {
   ...
   private void sortProducts() {
      Collections.sort(sortedProducts, new Comparator() {
         public int compare(Object o1, Object o2) {
            Product p1 = (Product) o1;
            Product p2 = (Product) o2;
            return p1.getPID().compareTo(p2.getPID());
         }
      });
   }
   public void addProduct(Product newProduct) {
      sortedProducts.add(newProduct);
      sortProducts();
      catalog.addProduct(newProduct);
   }
}
```

We can see that two classes had to be modified. The test chunk was actually too big for one single iteration. But everything seems to have gone well.

Liar View number two!

Nope! Actually, the Liar View bug pattern we saw before sneaked in again: a corresponding test is still missing in `ProductsListModelTest`. To make sure our readers won't sink into a testing monotony, we will turn the tables this time—we will give only the bug fix and leave it up to the readers to write a matching test as an exercise:

```
public void addProduct(Product newProduct) {
   sortedProducts.add(newProduct);
   sortProducts();
   catalog.addProduct(newProduct);
   int index = sortedProducts.indexOf(newProduct);
   fireIntervalAdded(this, index, index);
}
```

Things unfinished

Let's stop the development of this graphical product catalog editor at this point.[2] The following remains to be done or tested:

- A `ProductCreatorDialog` class that implements the `ProductCreator` interface is missing.
- We have to build this dialog class into our `CatalogEditor`.
- What happens if `ProductCreator.create()` returns `null`, that is, when the Cancel button is pressed in `ProductCreatorDialog`?

These requirements do not produce new test-first problems, so we can confidently leave them up to the experienced reader. Besides, nobody has to be ashamed of using a "visual" test to actually discover additional test cases or errors and to then translate them back into JUnit test cases. To work with an example based on this approach, we recommend our readers delete a product and then look at the buttons and product details. You will see that several things are wrong!

2. The code available from our Web site includes the complete example.

Brief Summary

As with our test-first development of servlets (see Chapter 12, Section 12.3), we have gotten to a point where the full functionality of the graphical editor is not yet available on a functional level, because a certain component (i.e., `ProductCreatorDialog` in this case) is still missing. But still, we can test the behavior depending on this in unit tests.

However, the layout of the product catalog editor is still unsatisfactory. The current layout is anything but pretty (Figure 13.2). Theoretically, we could check everything by automated testing. For example, [URL:WakeGUI] also tests for the relative position of widgets to each other. However, such test cases are often very unstable, because the layout of a GUI normally changes more often than its functionality. For this reason, we should carefully consider whether or not this type of test automation would better be implemented as part of the functional test suite. On the one hand, this would mean using different testing tools; on the other hand, we would get direct feedback from the users.

We will content ourselves with a (slightly) improved and refactored implementation of the layout at this point. This improved version takes us a step closer to the desired layout, as shown in Figure 13.3:

```
public class CatalogEditor extends JFrame
   implements ActionListener, ListSelectionListener {
   ...
   private void createWidgets(ProductCatalog catalog) {
```

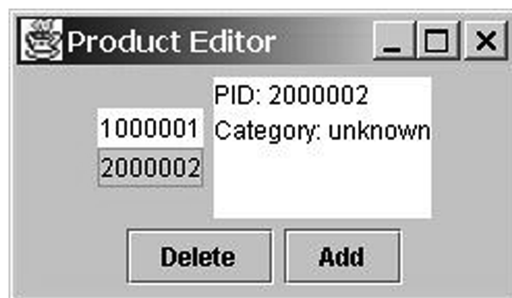

Figure 13.2 A "rudimentary" layout.

}

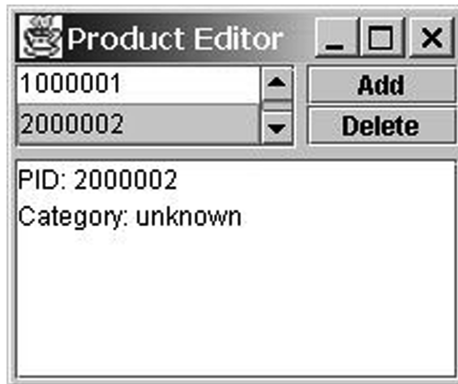

Figure 13.3 The improved layout.

```
      buildProductDetails();
      buildProductList(catalog);
      buildAddButton();
      buildDeleteButton();
      buildLayout();
   }
   private void buildLayout() {
      getContentPane().setLayout(new BorderLayout(5,5));
      setSize(300, 300);
      JPanel buttonPane = new JPanel();
      buttonPane.setLayout(new GridLayout(2,1));
      getContentPane().add(new JScrollPane(productDetails),
         BorderLayout.SOUTH);
      getContentPane().add(new JScrollPane(productList),
         BorderLayout.CENTER);
      buttonPane.add(addButton);
      buttonPane.add(deleteButton);
      getContentPane().add(buttonPane, BorderLayout.EAST);
   }
   private void buildProductDetails() {
      productDetails = new JTextArea();
      productDetails.setPreferredSize(
         new Dimension(50,100));
   }
```

```
    private void buildProductList(ProductCatalog catalog) {
        ProductsListModel model =
            new ProductsListModel(catalog);
        productList = new JList(model);
        productList.setSelectionMode(
            ListSelectionModel.SINGLE_SELECTION);
        productList.addListSelectionListener(this);
    }
    private void buildDeleteButton() {
        deleteButton = new JButton("Delete");
        deleteButton.setEnabled(false);
        deleteButton.addActionListener(this);
    }
    private void buildAddButton() {
        addButton = new JButton("Add");
        addButton.addActionListener(this);
    }
}
```

Keeping the GUI Clear

The basic principle to facilitate GUI testing or to even make it almost superfluous is to keep all logic out of the GUI classes. In the preceding example we still had two kinds of logic within those classes:

- Interdependencies between two or more GUI elements, for example, selecting an item in the product list had to enable the Delete button.
- The GUI widgets are actually wired to the underlying models, for example, the product list model.

If we could eliminate the first point by extracting the interdependency logic into a class of its own, we could argue that the actual wiring is mere delegation, which is not really worthwhile testing. This is exactly the approach that Michael Feathers [02b] describes in *The Humble Dialog Box*. He presents a test-first technique to extract all user interface logic from the GUI class itself into a "smart object," which collaborates with an abstract

Humble Dialog

view and can thus be developed and tested independently. The actual "humble" view implementation does nothing more than provide simple, delegating getter and setter methods.

13.2 Short Detours

The path we have taken so far in this chapter has a few hidden problems:

Problems with the above approach

- All GUI widgets important for the tests have to be made visible in non-private instance variables or over corresponding getter methods. This conflicts with the style propagated so far, which tells us that instance variables should always be private, and access methods should only be offered if they are used by clients of that class.
- The GUI component does not "operate" over the actual event mechanism, but uses special methods of the widgets. The drawback is that certain error types will not be revealed, for example:
  - Even invisible and `disabled` components can be addressed in the unit test.
  - Errors originating from the Swing thread problem[3] will slip through.

The first point, visible variables, is connected to the non-public class properties discussed in Chapter 8, Section 8.2. As mentioned in that section, there are pros and cons about the question as to whether or not "internal things" should be made visible for testing purposes only. In any event, it means that we are coupling the tests to internal things of the implementation, making it more fragile. Although moving the widget creation and the widget referencing to a dedicated class can solve the visibility problem, it often does not lead to the simplest design.

The second point is actually a limitation of the mightiness and thus effectiveness of the test cases. For this reason, we will discuss two approaches to solve this point in the following sections.

3. Most method calls of the Swing components are not thread-safe, so that they have to be passed explicitly to the AWT thread over `SwingUtilities.invokeLater(Runnable)` for handling, unless they are invoked from the event-handling code.

JFCUnit

Swing-based testing

JFCUnit is an open source project aimed at creating a framework for unit testing of Swing applications. You can download the latest version from [URL:JFCUnit] and actively contribute to the product's further development. JFCUnit provides support for the following:

- Locating `java.awt.Window` instances (e.g., frames and dialogs) opened by the code under test.
- Locating Swing components within the component tree of a window based on type, name, or other properties.
- Sending targeted events in the AWT event handling thread, clicking a button or selecting a `JTree` subtree, for example.
- Handling thread-safe testing of components.

The process of creating a JFCUnit test suite does not differ much from programming a normal suite. First of all, we have to derive our test classes from `junit.extensions.jfcunit.JFCTestCase`. This gives us access to a helper class that supplies methods for finding components and triggering events. Internally, JFCUnit takes care of switching between the test thread and the AWT event thread.

To understand this better, we will translate the above test case, `testDeleteProduct()`, into a JCFUnit test case:

```
import javax.swing.*;
import junit.extensions.jfcunit.*;
public class CatalogEditorJFCTest extends JFCTestCase {
   public CatalogEditorJFCTest(String name) {...}
   private JFCTestHelper helper;
   private ProductCatalog catalog;
   protected void setUp() {
      helper = new JFCTestHelper();
      catalog = new SimpleProductCatalog();
      catalog.addProduct(new Product("123456"));
      catalog.addProduct(new Product("654321"));
      new CatalogEditor(catalog).show();
   }
```

```
   protected void tearDown() {
      helper.cleanUp(this);
   }
   public void testDeleteProduct() {
      CatalogEditor editor =
         (CatalogEditor) helper.getWindow("Product Editor");
      JList productList =
         (JList) helper.findComponent(JList.class, editor, 0);
      AbstractButtonFinder deleteButtonFinder =
         new AbstractButtonFinder("Delete");
      JButton deleteButton = (JButton) helper.findComponent(
         deleteButtonFinder, editor, 0);
      JListMouseEventData listClick =
         new JListMouseEventData(this, productList, 0, 1);
      helper.enterClickAndLeave(listClick);
      MouseEventData deleteClick =
         new MouseEventData(this, deleteButton);
      helper.enterClickAndLeave(deleteClick);
      ListModel listModel = productList.getModel();
      assertEquals(1, listModel.getSize());
      assertEquals(new Product("654321"),
                   listModel.getElementAt(0));
      assertEquals(1, catalog.getProducts().size());
      assertTrue(catalog.getProducts().contains(
         new Product("654321")));
   }
}
```

First, we instantiate `JFCTestHelper` in `setUp()`, then we create and display the product editor under test. The method `cleanUp(...)` in `tearDown()` removes all remaining windows and dialogs from the desktop. However, the actual events happen in `testDeleteProduct()`: the starting-point into a JFCUnit test case is normally from the main window of an application (i.e., `ProductEditor` object in our example). Next, we determine the `productList` and `deleteButton` widgets as subcomponents of the editor, and then feed them with the product selection and mouse clicking

events. Note that we adopt the actual `assert` commands virtually as they are from `ProductEditorTest` of the corresponding test.

Differences to the first implementation

One important difference to the test approach discussed in the previous subsection is that we search for specific widgets by their types, names, or labels, instead of having to rely on the visibility in the class. Another important difference is that we do not call any methods directly from the widgets, but rather set those events in the event queue that will be triggered from within the GUI in the real world. Note that creating events with all their data is much more complex than a simple method invocation.

Benefits and drawbacks

The testing approach with JFCUnit has benefits and drawbacks. One major benefit is that the tests are very close to the actual events on a GUI. This means that we can test the behavior in several open windows and dialogs and discover Swing-typical multi-thread errors at the same time. The drawback is that the test cases have to make detours to access the actual components and widgets. Therefore, a JFCUnit test case is too heavy for the role of fine-grained unit test, but it is ideally suited for interaction and integration tests. Note that the current release—0.3 beta—supports only the most important Swing features; unusual things like drag and drop are not yet included.

The AWT Robot

From JDK 1.3 up

Since JDK 1.3, Java offers a class explicitly conceived for GUI testing: `java.awt.Robot`. Instances of this class offer ways to simulate mouse and keyboard actions, including the following commands: `mouseMove(...)`, `mousePress(...)`, `mouseRelease(...)`, `keyPress(...)`, and `keyRelease(...)`. However, this interface is on too low an abstraction level for unit tests. For example, mouse movements have to be positioned on exact pixels and each touch of keys has to be determined individually. The `Robot` class was mainly designed for the development of pure Java solutions in the capture and replay approach.

It is also feasible to use an AWT robot to build a test framework similar to the abstraction level of JFCUnit. This relieves the user from routine work like positioning the mouse pointer over a specific component or pressing several keys to enter a character string. Abbot is such a tool.

Other Tools

Abbot

Abbot [URL:Abbot] is a GUI testing tool built on `java.awt.Robot` that works on two levels of abstraction:

- Functional testing, by writing Swing test cases in a script language which can then be invoked from within JUnit.
- Programmatic unit level testing.

Jemmy

Another Swing GUI testing tool has been developed by Sun's Netbeans team. *Jemmy* [URL:Jemmy] is technically similar to JFCUnit since it emulates events being posted to the AWT event queue.

13.3 Summary

This chapter showed how you can use the test-first approach to create graphical user interfaces (GUIs). Currently, the testing community uses mainly two variants:

- Fine-grained unit tests, by directly addressing components and widgets.
- JFCUnit for interaction tests that simulate the actual GUI operation.
- Either of the two approaches reaches its limits when trying to realize certain layout standards and nonspecifiable properties like ergonomics or esthetics.

In practice, many teams create user interfaces by use of special tools (GUI builders). Such tools can speed up the GUI implementation and open the way for a twofold approach: using the GUI builder to create the GUI's layout and developing the connection to the logic layer in the test-first approach.