Visual Component Integration and Regression Test

Ludger Martin

Darmstadt University of Technology
Department of Computer Science
Wilhelminenstr. 7, 64283 Darmstadt, Germany
Tel: +49 (0)6151 16 6710, Fax: +49 (0)6151 16 6707
lumartin@gkec.informatik.tu-darmstadt.de,

Abstract. To ensure that components are reusable and reliable, it is important to verify their functionality. In this paper, we present a tool for component integration testing. The components are tested as black boxes. The test tool HotAgent Test enables the tester to specify a test case in a visual way. The tool HotAgent Regression is presented to run the specified tests in regression.

Keywords: Component, Integration Test, Visual Test

1 Introduction

Recently, component technology is being used extensively to build many kinds of software. Frequently, components from different providers are used to assemble component-based applications. To ensure the reliability of these applications, the components need to be well tested.

In [Mar01b] the HotAgent [Mar] development environment is presented. It is a component framework to construct agents for electronic commerce. The framework provides a special set of components to support the easy construction of agents. The agents could undertake the task of doing routine work. Those could analyze electronic documents, extract important information, and compose an answer or a form. The HotAgent development environment offers different tools to develop components, to test components, and to compose agents using components. This paper focuses on a tool for component integration test. HotAgent Test enables the tester to specify a test case in a visual way (Burnett *et al.* [BRCR01,RLDB98]: "What You See Is What You Test"). Using this tool, reuseability and reliability of a component can be verified.

In Section 2, we discuss how to test components. In Section 3, the HotAgent Test and HotAgent Regression tools are presented. In Section 4, other parts of the HotAgent development environment are introduced. Section 5 shows some related work. The paper concludes with a summary and a discussion of further work in Section 6.

2 Testing Component Software

To discuss testing of components, it is necessary to first define the term component. Later on testing in general and testing of components can be discussed.

2.1 Components

Szyperski [Szy98] defines: "A *component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A *component* consists of several classes which are encapsulated by a set of well defined interfaces. The *interface* is the sole public part of the component. All other classes of the component are private. Therefore no knowledge about the component structure and behavior can be assumed for dynamic analysis. The interfaces ensure the *communication* between all the components.

The HotAgent development environment is not based on one of the well-known component models like EJB (Enterprise Java Beans), CCM (CORBA Component Model), or COM+ (Common Object Model). Instead, a small component model is used. The advantage of a small model is that it is easier to use with a lower overhead, which allows a user to focus on component usage. Inter-component communication relies only on a *single common interface* each component has to implement. It is also based on an event mechanism which is common to all component models mentioned above. So it is not hard to transfer the small model to a more complex one.

The component model is implemented in VisualWorks Smalltalk. The common interface that realizes the communication among the components is the sole public part of all components. All other classes that are part of a component are private within that component. This approach enables all components to communicate with each other independently of their tasks.

Lüer and Rosenblum [LR00,LR01] demand self documentation of components. The HotAgent component model offers three kinds of documentation. The first is a short unique *descriptive name* of the component, e.g., eMail, which is stored in a global name space. To have the opportunity to sort the components into different categories, every component has one *category name*, e.g., data management, that is also used to locate components in an easy way. If a component is chosen, it is necessary to check that it is the right one for the task at hand. For this purpose, a component has a *descriptive text* to give an impression of its features.

Whenever an instance of a component is created, it is important to assign a unique *instance name*. This name is only known in the composed agent and can be used to identify the component during runtime.

The interface of a component ensures the communication between the components. Every component provides so called entrances and exits. An *entrance* can receive data and produce a result, depending on the component functionality. An *exit* is activated if a component has to notify its environment about a change in

its internal state. It supplies all necessary data and can process a possible result. Lüer and Rosenblum [LR00,LR01] call the entrances *requires ports* and the exits *provides ports*. Since Smalltalk is an untyped programming language, the data can be of any type. If a wrong type is provided by an exit or an entrance needs other data, a failure will occur during runtime. It is only possible to communicate using these public entrances and exits. To compose components, exits can be connected with entrances. Similar to the component, every entrance and exit has a *name* and a *descriptive text* to allow self documentation. It is important to describe the data provided by an exit and needed for an entrance, respectively.

2.2 Testing

Now testing can be discussed. Weyuker [Wey01] explains that *integration test* focuses on the interface test and is mainly done by the *programmer team* or a *test team*.

Beck [Bec94] defines two kinds of faults: "A *failure* is an anticipated problem. When you write tests, you check for expected results. If you get a different answer, that is a failure. An *error* is more catastrophic, an error condition you didn't check for." The tester determines an instruction block with a predictable result, called *test fixture*, to find failures. When writing test fixtures, he has to supply the correct results of the fixture.

McCarthy [McC97] defines *unit test* as: "testing a function, module, or object in isolation from the rest of the program." A *unit* is normally the smallest unit of programming. In object-oriented programming, the smallest unit is a class.

2.3 Component Test

Tests for component software need to regard components as *black boxes*. In component technology, the smallest unit is a component. It is not useful to break down the components and test the encapsulated classes to test the functionality of a program based on components.

Since we understand a component as a block with entrances and exits, it is practicable to test it using a combination of *control* and *data flow test*: A message or data can be sent into the entrances and can be tested against the data coming from the exits.

In this context, a possible test fixture is to send data to an entrance and to check the responding exits of the component. As usual in data flow tests, the tester must define the data of the individual exits in advance to check the answers. If an exit has a wrong value or is not activated, a failure occurred.

Component developers have to test the components in isolation from other components. They need to make sure that the component works correctly in all possible scenarios. *Component assemblers* can test the separated component to verify the component for the concrete application. Harrold *et al.* [HLS99] describe that it is sufficient for *component assemblers* to test only functions they need. If they test more functions, they cannot guarantee the correct functioning

in the component's special application. Some additional calls could set variables, which could be unset in other situations.

Another good opportunity for this kind of test is to validate the component specification. It is possible to test a component to find out if it conforms to the documented functionality. If the *component assembler* wants to determine whether the assembled components fit their responsibilities in the context of the whole application, the composition of components must be tested.

3 HotAgent Test and Regression

HotAgent Test is one part of the HotAgent development environment. It enables the tester to specify a test case in a visual way. It is based on the HotAgent Assembly editor (see section 3.1). The HotAgent Test tool is described in section 3.2. To allow regression tests, the tool HotAgent Regression was developed. It is explained in section 3.3.

To show component testing, two example components are used. The Data Base component provides a simple data base. It has an entrance init to set the initial content. The entrance add can be used to add a data set. The entrance query can be used to query data. The second component is the Split Result component. This component has one entrance query to define a request. This request will be passed through the exit query to another component. After receiving the result of the request, the value is published by the exit result.

3.1 HotAgent Assembly Editor

The HotAgent Assembly editor, presented in [Mar01a], can be used for application composition. Agents can be composed visually using predefined components (see Figure 1).

The first step when designing an application, is to choose a position for a new component on a workspace. Components can be classified into two different types: (i) visible components representing an element of a graphical user interface (GUI) and (ii) invisible components like a database without any visual representation. The workspace of the HotAgent Assembly editor consists of two parts: the main part that accepts only invisible components and a frame on which visible and invisible components can be placed.

Once the developer has indicated a position on workspace, a dialog window for the selection of a component appears. The dialog shows lists to choose a component and an entry field to choose the component instance names.

When all required components have been placed onto the workspace, it is necessary to connect them. In order to create a connection, a source component needs to be selected. Once this has been done, the middle handle of the selected component can be dragged onto the target component (see Figure 2). After finishing the drag, HotAgent checks whether there is at least one source component exit and one target component entrance. If this is the case, another dialog window pops up which can be used to select the corresponding exits and

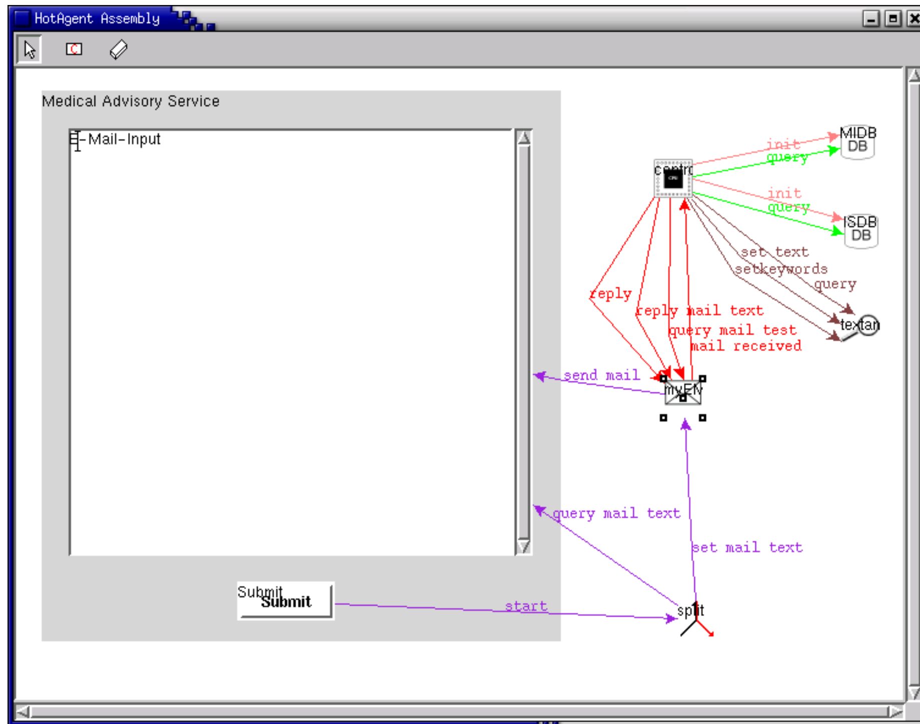

Fig. 1. Workspace with Components and Connections

entrances. In this window, the exit and entrance can be chosen from a list. In addition, connections can be labeled and colored. Figure 2 shows how to create a connection from the query exit of the split component to the get random entrance of the random component.

As an example for an agent, Figure 1 displays the workspace of the Medical Advisory Service agent [Cla01]. The agent receives patients' symptom descriptions via email and analyzes them for matches with symptoms of known medical diseases. If the agent finds a disease, it answers the email with a proposal for a drug. To simplify the email handling in the prototype program, there is a user interface to simulate email-sending and receiving. The user types the email-text in a dialog box and the "sending" of the "email" is done by pushing a button. The dialog box is also used to show the answer-email of the agent. Therefore, the agent has a visible part which consists of these two user interface components. The rest of the agent consists of several invisible components.

3.2 HotAgent Test Tool

The HotAgent Test tool operates similarly to the HotAgent Assembly editor. The difference is that the workspace consists only of the main part, but

Fig. 2. Connecting Components

all types of components can be placed on this part. The tool bar on top of the window has two additional buttons, which are described later.

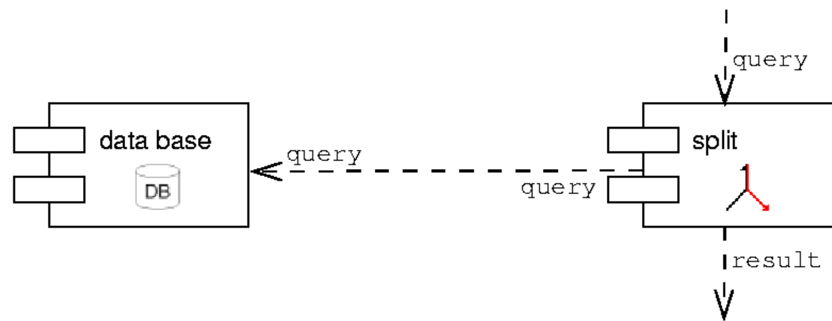

Fig. 3. Composition of Components (modified UML notation)

In the test case example, the component Data Base with instance name data base is tested. First the data base component needs to be initialized with an empty data set (data 0 component). After this, three illness symptoms and deceases are added to the data base (data 1...3 component). Now the real test can be specified. To do this, the Split Result component with instance name split is used as shown in Figure 3. To test the query entrance of the data base, a proper query needs to be generated. A possible data block is (in Smalltalk notation):

```
[ #(#onlyObjects: #(#cough)) ]
```

This data is sent to the the entrance query of the split component. It forwards the request to the data base component and this component processes the query. The result is returned to the split component. The split component provides the data at the exit result. To test this data, an assertion block with boolean result needs to be defined. If it is evaluated as true, the test is positive. A possible assertion block is:

```
[ :testData |
(testData includes: #cold) and:
[testData includes: #bronchitis] ]
```

In the example, deseases with the symptom *cough* are queried. Expected is a list with the two deseases *cold* and *bronchitis*. In this example only a simple list is tested. It is also possible to test structured values like instances of trees. In the test block, any method call is possible. This way, it is feasible to test for the existence of a specific object.

As shown above, it is not very complicated to create a test fixture. Every test fixture can be created with the same scheme:

1. create all components
2. connect the components
3. send data to a component entrance
4. define an assertion for a component exit

To create a test case using HotAgent Test, it is necessary to place all components to test on the workspace. In our example, the components data base and split are used and connected as mentioned previously.

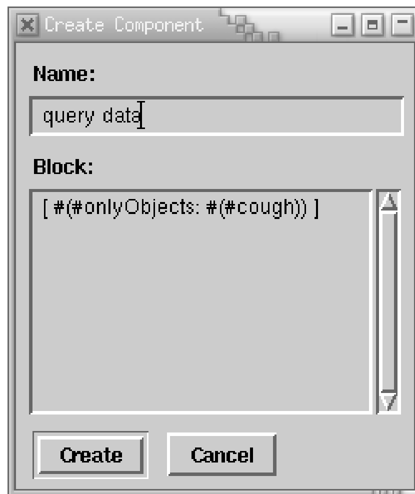

Fig. 4. Data Creation Dialog

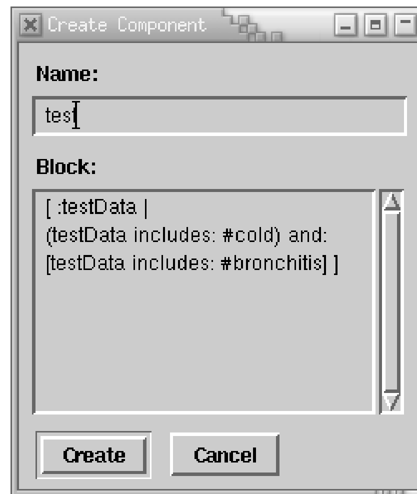

Fig. 5. Test Creation Dialog

Once all components have been created and connected, the data for the entrances to test need to be created. The HotAgent Test tool provides a special button for this purpose. Similar to a normal component, the position on the workspace needs to be chosen. A dialog window pops up (see Figure 4) to specify the instance name of the data component and a block which will be evaluated to gain the data.

If there are several data components, the name of the component specifies the call sequence. The alphabetic first one will be sent first. To connect the data to a component, the data component provides an exit names data.

Similar to this, a button to create a test assertion is provided. After selecting a position a dialog appears (see Figure 5). In this dialog an instance name and also an assertion block have to be specified.

The last thing to do is to connect a component exit with the related assertion component. For this purpose, the assertion component provides an entrance with the name test.

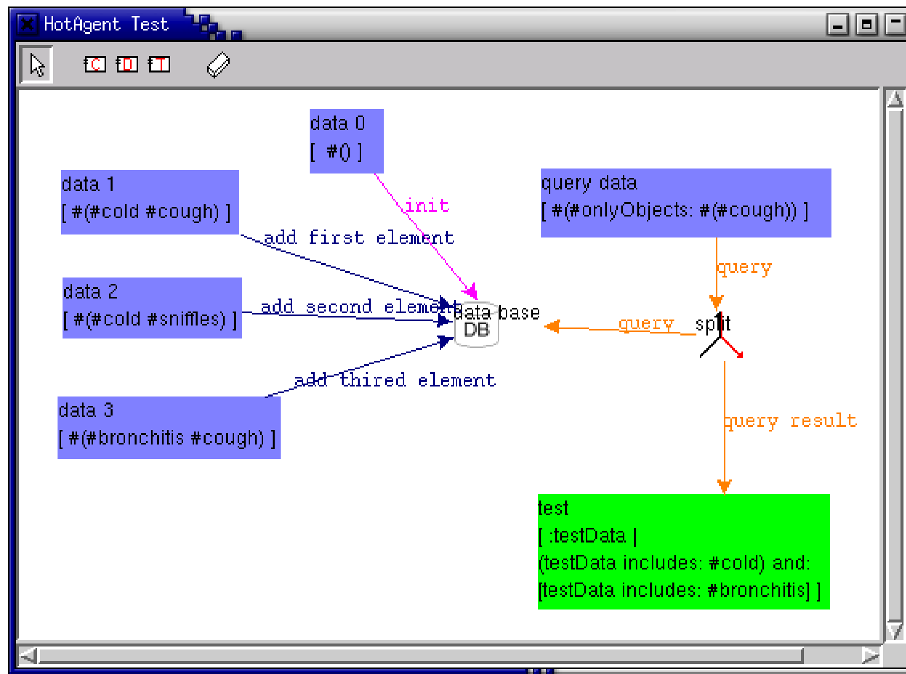

Fig. 6. HotAgent Test Tool

The complete test fixture is shown in Figure 6. When the test is executed the color of the assertion component changes. Burnett *et al.* [BRCR01] mention that different colors are necessary to show different states of a test. In HotAgent Test the assertion has a dark cyan background if it is untested. It changes to yellow if the test is in process. If the assertion succeeds, the background gets green otherwise it is red.

Weyuker [Wey01] mentions that test suites have to be stored along with a component. HotAgent Test offers the opportunity to store a test fixture with every component or component program. Poulin [Pou01] postulates that a test tool has to be simple and meaningful to ensure that it is used. HotAgent Test visualizes a test fixture in a clear way and the usage is easy.

3.3 HotAgent Regression

Siepmann and Newton [SN94] mention that it is important to run regression tests. *Regression testing* is the rerunning of tests after changes on the source code. If the source code of a component is changed, it is necessary to prove that the behavior is unchanged resp. a failure is eliminated. Pauli [Pau01] adds “Test everything you can, and test it often”.

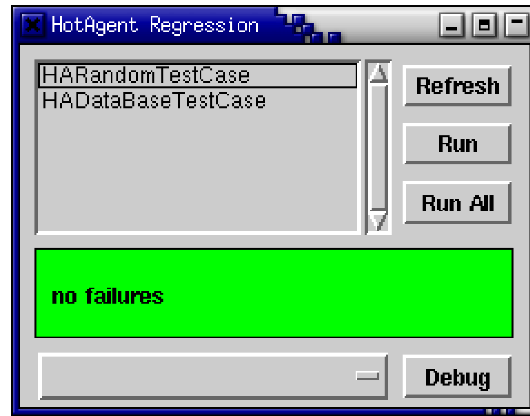

Fig. 7. HotAgent Regression Tool

HotAgent Regression is a tool to run regression tests. It is similar to the SUnit test runner [Cam]. Figure 7 shows a screenshot fo the tool. The list in the top left corner shows all possible test cases. By selecting one test case and choosing the button *Run*, this specific test case is processed. When choosing *Run All*, all test cases in the list are executed. The color area symbolizes the state of the tests. Initially it has a dark cyan background. It changes to yellow if tests are in process. If all assertions succeed, the background becomes green, otherwise it is red. In the case of a failure, the faulty test cases are shown in the drop down list at the bottom. After selecting one faulty test case and pushing the button *Debug*, HotAgent Test is called to examine the faulty test case and to correct the fault.

4 Other HotAgent Parts

In the previous section, tools to test components are discussed. HotAgent offers some additional tools [Mar01b] to work with components.

The simplest method to create new visual components is to use the HotAgent Pattern Component editor. The first step is to place graphical elements into a workspace. The elements can have different shapes and colors. The second step is to assign behavioral patterns to the elements, e.g., an element

can be moved on a line or plane. The editor creates all necessary classes for the new component on its own. Using this editor, new control mechanisms for a user interface can be created.

The HotAgent Component editor is an all-purpose editor. It is good for constructing invisible components. It supports the programmer by creating new components using other existing components. They can be placed on a workspace and can be connected with each other and new component entrances and exits. Using this editor, a programmer can construct components without knowing very much about the underlying component model.

Using the constructed components, it is possible to build complete applications. This editor is presented in Section 3.1.

The HotAgent Visualize tool [MGM02] provides a three-dimensional visualization of component programs' runtime behavior based on dynamic analysis. The communication between the components is dynamically analyzed and the gathered data can be filtered. The visualization is three dimensional and the focus can be chosen: either the whole program execution can be viewed at the same time or the developer can zoom into special regions. HotAgent Visualize is also useful for testing component programs.

5 Related Work

Weyuker [Wey98] illustrates that a component system needs to be tested at least in three ways.

- *unit tests* are good for testing individual components
- *integration testing* is used to verify compositions of tests
- *system tests* are used to show the correctness of complete systems

Also, *component providers* and *component assemblers* need different kinds of tests: Harrold *et al.* [HLS99] analyze two groups of testers. The first group contains the *component providers*. They have to perform mainly unit tests to ensure correct functionality in all possible contexts. The *provider* can not test components in all possible scenarios. Therefore the second group, the *assemblers*, has to test the components, too. *Component assemblers* also have the possibility to test components in the context of the concrete program (*integration* and *system test*). They check if a component is correct, and whether a component is good for the special application. Harrold *et al.* [HLS99] try to generate test cases for components automatically. The component provider has to give language independent summary information with every component. Using this information, the user can split his application into smaller parts and create test cases with the same information. Unfortunately, the authors do not tell very much about the implementation.

Siepmann and Newton [SN94] present the system TOBAC. It is a user interface to create and run test cases. The tester is guided through menus to create objects to test and test cases. A browser lists all test suites and allows the execution of regression test.

In Burnett *et al.* [BRCR01] a mechanism to test spread sheets is presented. The user can define tests by arranging cells of a spread sheet on a workspace. The test state is also visualized using different colors.

6 Summary and Further Work

In this paper, the HotAgent Test tool to test components is presented. It shows an extension of the component composition editor HotAgent Assembly. The tool supports the *component developer* and *component assembler* by testing components as black boxes using defined entrances and exits. The example shows that the solution is suitable for *integration testing*. The tool is simple and the visualization is meaningful to ensure easy usage.

In addition HotAgent Regression is presented to support regression tests. The tool allows to run several test cases at the same time and to inspect a possibly faulty test case.

A hard question is how to know whether a test suite is complete. McCarthy [McC97] claims that it is always uncertain if a test suite is complete. He proposes to test until finding several failures. It is necessary to also investigate a different solution. Another interesting question is how much testing is needed for a composition of components if the separated components are already tested.

References

[Bec94] Kent Beck. Simple Smalltalk Testing: With Patterns. *Smalltalk Report*, October 1994. <http://www.xprogramming.com/testfram.htm> (February 2002).

[BRCR01] Margaret Burnett, Bing Ren, Curtis Cook, and Gregg Rothermel. Visually Testing Recursive Programms in Spreadsheet Languages. In *IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 288 – 295, September 2001.

[Cam] Camp Smalltalk. *ANSI-ST-tests Project*. <http://ANSI-ST-tests.sourceforge.net/> (February 2002).

[Cla01] Thorsten Clausius. Komponenten zur Konstruktion von Agenten. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, 2001.

[HLS99] Mary Jean Harrold, Donglin Liang, and Saurabh Sinha. An Approach to Analyzing and Testing Component-Based Systems. *Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems*, May 1999.

[LR00] Chris Lüer and David S. Rosenblum. Wren – An Environment for Component-Based Development. Technical report, Department of Information and Computer Science, University of California, Irvine, September 2000.

[LR01] Chris Lüer and David S. Rosenblum. Wren – An Environment for Component-Based Development. In *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 207 – 217, September 2001.

[Mar] Ludger Martin. *HotAgent homepage.* <http://www.gkec.informatik.tu-darmstadt.de/HotAgent/> (February 2002).

[Mar01a] Ludger Martin. HotAgent Component Assembly Editor. In Jean-Guy Schneider and Markus Lumpe, editors, *Proceedings Workshop on Component Composition Languages*, pages 25 – 32, September 2001.

[Mar01b] Ludger Martin. Visual Development Environment Based on Component Technique. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 346 – 347, September 2001.

[McC97] Adrian McCarthy. Unit and Regression Testing. *Dr. Dobbs Journal*, Februar 1997.

[MGM02] Ludger Martin, Anke Giesl, and Johannes Martin. Dynamic Component Program Visualization. to be submitted, 2002.

[Pau01] Kevin Pauli. Pattern your way to automated regression testing. *JavaWorld*, September 2001. <http://www.javaworld.com/javaworld/jw-09-2001/jw-0921-test.html> (February 2002).

[Pou01] Jeffrey S. Poulin. *Component-Based Software Engineering*, chapter Measurement and Metrics for Software Components, pages 435 – 452. Addison Wesley, 2001.

[RLDB98] Gregg Rothermel, Lixin Li, Christopher DuPuis, and Margaret Burnett. What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. In *1998 International Conference on Software Engineering*, April 1998.

[SN94] Ernst Siepmann and A. Richard Newton. TOBAC: a test case browser for testing object-oriented software. In *Proceedings of the 1994 international symposium on software testing and analysis*, pages 154 – 168, 1994.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Wey98] Elaine J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, pages 54 – 59, September/October 1998.

[Wey01] Elaine J. Weyuker. *Component-Based Software Engineering*, chapter The Trouble with Testing Components, pages 499 – 512. Addison Wesley, 2001.