Tempto – An Object-Oriented Test Framework And Test Management Infrastructure Based On Fine-Grained XML-Documents

[1]Ralf-Dieter Schimkat[τ,υ] Martin Schmidt-Dannert[τ] Wolfgang Küchlin[τ]
Rainer Krautter[υ]

[τ] Wilhelm-Schickard-Institute for Computer Science
Symbolic Computation
Sand 13, D-72074 Tübingen, Germany

[υ] debis Systemhaus Industry GmbH
Competence Center DMS
Erich-Herion-Str.11-13, D-70736 Fellbach, Germany

Abstract. This paper describes Tempto, an object-oriented framework that supports creating, running and managing automated tests in Java. Key objectives of Tempto are the uniform handling of both, unit and acceptance tests within an object-oriented framework, the establishment of an XML-based repository for test reports, and the unified management of test reports within a distributed infrastructure. This management is based on concepts of modern document management systems such as the archiving, accessing and presenting of fine-grained documents. We argue that the design of test reports and test information as structured fine-grained XML-documents empowers an object-oriented infrastructure for generating test reports and deploying test management facilities in large-scale application domains, even within the Web.

1 Introduction

Extensive testing is seen as critical for the development of high quality software and for the maintenance and improvement of code [7]. In practice however, testing ranges from end tests done in a hurry shortly before delivery, to testing as soon as some part has been developed, or even concurrently with the development, by means of creating growing automated test suites that are required to pass thoroughly.

Testing as an integral part of software development rather than an activity done after the core development is getting emphasized in recent software development approaches as in eXtreme Programming (XP) [1, 10]. Two different kinds of tests are distinguished by XP: Unit tests and acceptance tests, formerly called functional tests.

[1] Supported by debis Systemhaus Industry

The unit tests are constructed by the software developer while, or even before, writing the code. Their purpose is to test software units, like single Classes or Objects, isolated from the context they will act in. According to Beck [1], unit tests are “tests written from the perspective of the programmer”. In contrast, the acceptance tests are targeting the functionality of software modules or the whole system. They simulate application situations, ideally using customer data, and check the output against expected results. Acceptance tests are “tests written from the perspective of the customer” [1].

Unit Tests and acceptance tests closely work together, although they may get developed independently by different people. While unit tests assure the correct behaviour of software units in isolation, the co-operating behaviour of these units can not be verified by unit tests. This is the main focus of the acceptance tests.

The unit tests are closely tied together with the source code. The test writer is usually also the author of the code. The knowledge about the underlying implementation may be used in unit tests, which is characteristically for “white box” tests. To avoid testing the code against itself, unit tests can easily be created before the implementation, assuring additional “black box” treatment of the code. In fact, XP stresses on this “test-first approach” in its development cycle.

Acceptance tests treat the software as a black box and do not need to know anything about the implementation except for some higher level interface, as the unit tests cover everything below. In combination these two test categories span the spectrum of white to black box tests allowing a practical and affordable approach for extensive and effective testing.

Tempto tries to release the co-operational potential of acceptance and unit tests without trying to unify them. As a consequence, while the handling and managing of unit and acceptance tests is implemented separately, the core framework remains the same, supporting the co-operation of both. This way, unit and acceptance tests keep their lightweight character and remain flexible.

Of course there are more advantages to be had from developing unit and acceptance tests than mere quality assurance. First of all, the automated unit tests give immediate feedback to the programmer on the code just written. This induces confidence and motivation: Confidence because the tests assure the correct behaviour of the object just created or modified; motivation as progress is quickly seen. Secondly, the collection of tests, especially the acceptance tests, in combination with some management tool helps keeping control over the current status of the development project and concentrating on the critical areas. Thirdly, the tests show how the implementation should be used. Getting run at least at every check-in, they are as much up to date as any documentation can ever be. This can be an invaluable resource of low level information for later extension or modification of the code. Finally there is an overall development speedup that comes from reduced need of debugging and from precise information on the point where a particular test fails, leading to reduced overall cost of development and maintenance.

Within large-scale and distributed enterprises the management of acceptance and unit tests is a major concern. In order to be able to access and retrieve the generated test

information in a structured manner, all test data should be made persistent within a data repository. This repository should store test information in a generic way using a standard data model. Thus, any kind of test application can use the proposed repository in a uniform fashion.

Furthermore, the distributed nature of large enterprises requires a scalable and extensible approach to the integration process of test applications and test tools. From an conceptual point of view, the participating components should be decoupled from each other as much as possible. In Tempto, lightweight and object-oriented middleware components offer extensible application interfaces and provide scalable communication strategies, such as a message-based notification mechanism which is based on the publish/subscribe communication paradigm. The key idea of the management infrastructure of Tempto is to provide facilities where each participating application can easily be interfaced to but does not necessarily rely on them. Therefore, the distributed management infrastructure can evolve and be extended fairly easily.

In Section 2, the core framework architecture of Tempto, which is based on JUnit, and our extensible management infrastructure is discussed. In Section 3, related research approaches are described. Section 4 concludes with a short summary of Tempto.

2 Architectural Overview

2.1 Design of the Test Framework

2.1.1. Overview

A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications. Frameworks provide a means to allow the reuse of design *and* the reuse of implementation [11, 13]. One classification of frameworks is the distinction of whitebox frameworks which are customized by using inheritance and blackbox frameworks which are customized by composition and specific initialization [11, 5]. Tempto has elements of both kinds of frameworks. Subclassing, which is typical for whitebox frameworks, for example is required to create tests. Many customization facilities however are provided by calling methods or composing objects differently, which is typical for blackbox frameworks.

Tempto is based on JUnit [2], a test framework for automated unit tests in Java [9], which is developed by Erich Gamma and Kent Beck. Tempto takes the same basic approach for mapping tests into code as JUnit. The usability and functionality is improved in many areas compared to JUnit, however. Main issues during the development of Tempto have been the support for acceptance tests and the

integration into a distributed test result management system, which is described in Section 2.2.

Tempto consists of four different packages. The `framework` package is the central package in which the core functionality is implemented. A graphical User interface (UI) in package `swingui` and a text based UI in package `textui` are enclosed, providing an easy and effective way to view and run the tests, check the result report and gain detailed information about detected failures and errors. The `tools` package contains generally helpful classes which are used by the UI packages.

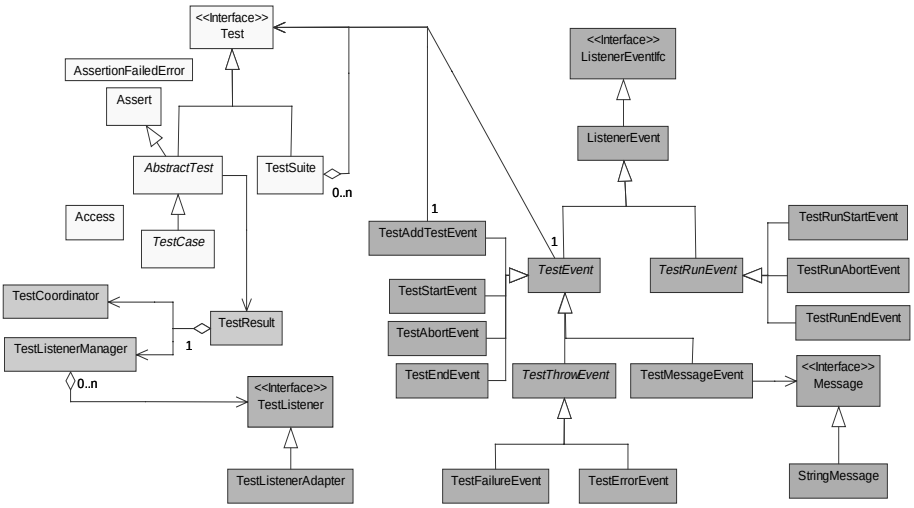


Figure 1 Framework Package Overview

2.1.2 The framework Package

The `framework` package, as already mentioned, is the central package of Tempto. There are no dependencies to any other package. This clear distinction not only helps to keep the costs low in case of changes not related to the core framework, but also is essential to the framework concept in general [5]. In consequence, the `framework` package can be used separately from all the other packages, for example to integrate it in some existing application or to provide a new UI.

An overview of the classes in the `framework` package are depicted in Figure 1. User defined tests are mapped as methods in classes derived from `TestCase`. From an abstract point of view, tests always have a basic execution sequence:

1. Setting up the test environment,
2. executing the test,

3. tearing down the test environment, and
4. comparing the result with the expected one.

The second step is reflected in the `Test` interface which is implemented by the classes `AbstractTest` and `TestSuite`. `AbstractTest` additionally provides methods which implement step one and three of the above list, and `Assert` supplies methods for verifying the results (step four). Additionally, the management of the tests is supported by methods within `TestCase` which uses `TestSuite` objects for grouping tests automatically or manually. The use of the Composite pattern [8] allows building flexible hierarchical test structures.

The flow of control in Tempto is quite intuitive. Having invoked the `run` method on the top `TestSuite`, the contained `TestSuite`s are automatically and recursively traversed and the enclosed `TestCase`s are executed. Hereby, the registered listeners get informed of all important events during the test run. In consequence, everything beside the initial start of the test run and the final evaluation of the detected errors and failures is automated. This leads to a minimal effort to do a complete test run, allowing frequent thoroughly testing.

The features explained so far are basically supported in JUnit. Tempto provides the following enhancements for test creation and management:

1. Test methods can have an arbitrary parameter signature which are not allowed in JUnit.
2. Tests can be run in parallel. By using the test suite as basic unit, some tests might run sequentially and others in parallel during the same test run. JUnit allows a test to execute in a thread but provides no execution control for it.
3. Dependencies between tests can be mapped by dynamically adding tests during a test run which is not possible in JUnit.
4. The events happening during a test run are buffered in order to decouple the test run from the listeners.
5. Each listener gets served by a separate event reader thread, protecting the listeners against misbehaving listeners.
6. Initialization and clean-up, which is performed once before and after execution of all tests within a test class, is supported additionally to the initialization and clean-up performed before and after every test.
7. User defined messages can be sent from within test methods to the listeners.
8. User defined events can be generated.
9. Access to private and protected object variables and methods is enabled by the class `Access`.

2.1.3 Test event Handling and the Listener Concept

While the test run is executed, the generated events have to be passed to the registered listeners. Hereby, it should be granted to achieve

- a fast execution of the test run, not interrupted by the event processing,
- support for execution of parallel running tests which may generate concurrent events and
- independent handling of the listeners, protecting them against each other.

In providing all these facilities, a maximal speed of both, the test run and the listener notification is achieved in Tempto.

To achieve a quick return from the event processing, all incoming events are stored in an event queue, as illustrated in Figure 2. This way, the caller is able to continue as soon as the event is stored and has not to wait until the listeners got informed about it. In synchronizing the access to the event queue, the loss of events from concurrently running tests is avoided.

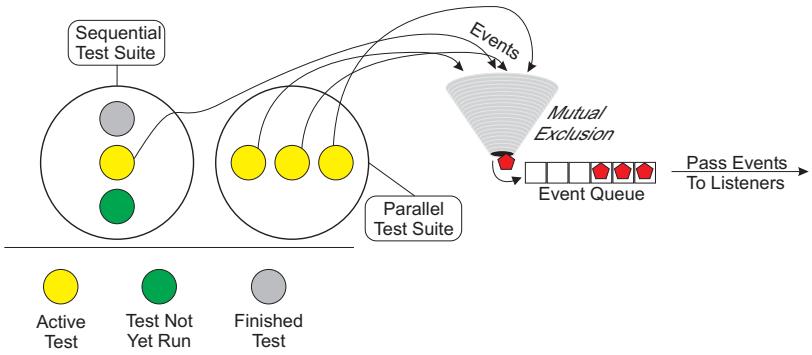


Figure 2 Event handling concept

The listeners are decoupled from each other by providing a separate event queue reader per listener, as illustrated in Figure 3. The event queue readers are created and destroyed automatically when the listeners register and de-register itself respectively. Besides protecting listeners from misbehaving listeners, this concept frees the listener to implement an event buffering. As neither the test run nor other listeners are dependent on quick event processing, even time-consuming event processing can be done directly on the listener's side.

2.1.4. Lightweight Approach

Tempto does not require the test developer to initialize a large system in order to start programming tests. Instead, by deriving a class from `TestCase` and directly implementing the test as methods therein, the programmer is already able to execute the tests with one of the included user interfaces. Given some implementation

examples of test classes, it is fairly easy for a developer to immediately start writing his own unit or acceptance tests and running them with Tempto. It is not required to know the whole framework concepts and its implementation or explore the accompanying packages in detail. This leads to a quick start and a flat learning curve, allowing to concentrate on the tests itself.

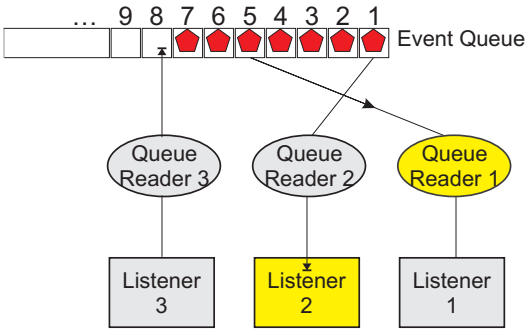


Figure 3 Listener handling concept

2.2 The XML-Based Test Management

We developed a test management infrastructure which aims to establish an object-oriented, extensible and distributed test environment. The key idea is to create a platform-independent test data repository which allows applications from other application domains to enrich the collected test data with additional information transforming ordinary test data into enhanced knowledge. For instance, a reasoner tool might figure out that particular failures of an application A have also occurred in the context of an application B. Although these two applications are anchored in different application domains, from an implementation point of view they might have some design- and architecture-related classes in common. Only the existence of a cross domain tool, such as the reasoner, offers the possibility to infer this kind of information or knowledge.

The management of test data includes the generation, processing, distribution, presentation and interaction of the information which is stored in a platform-independent repository. Test data is made persistent in a uniform way such that each piece of information is handled as a fine-grained XML-document [18]. This enables a structural retrieval mechanism of fine-grained test documents which follows the ideas and concepts of modern document, information and knowledge management systems.

The participating components, which are called parties, of such an XML-based test management are characterized by their possible heterogeneous application domains, application interfaces, and by their location within highly distributed infrastructures. Therefore each application which is integrated into Tempto (or tested by Tempto)

has to provide and generate test information in a platform and programming language independent manner because of interoperability issues.

```
<!-- DTD for events -->
<!ELEMENT events (preamble, (log | warn | error)*)>

<!ELEMENT preamble (version, timeZone?, datePattern?,
                    messageType?,  source?,  startDate?,
                    info?)>

<!ELEMENT version (#PCDATA)>
<!ELEMENT timeZone (#PCDATA)>
<!ELEMENT datePattern (#PCDATA)>
<!ELEMENT messageType (#PCDATA)>
<!ELEMENT startDate (#PCDATA)>
<!ELEMENT info (#PCDATA)>

<!ELEMENT log (source?, date, msg, param*)>
<!ELEMENT warn (source?, date, msg, param*)>
<!ELEMENT error (source?, date, msg, param*)>

<!ELEMENT source EMPTY>
<!ELEMENT date (#PCDATA)>
<!ELEMENT msg (#PCDATA)>
<!ELEMENT param EMPTY>

...
```

```
...
<!ATTLIST log     sev   CDATA   #IMPLIED
                  ID #IMPLIED
                  fatherID  IDREF #IMPLIED>

<!ATTLIST warn    sev             CDATA #IMPLIED
                  id              ID #IMPLIED
                  fatherID        IDREF #IMPLIED>

<!ATTLIST error   sev             CDATA #IMPLIED
                  id              ID #IMPLIED
                  fatherID        IDREF #IMPLIED>

<!ATTLIST source  app             CDATA #REQUIRED
                   appID          CDATA #IMPLIED
                   class          CDATA #IMPLIED
                   IP             CDATA #REQUIRED>

<!ATTLIST msg id                  CDATA #IMPLIED
                  para1           CDATA #IMPLIED
                  para2           CDATA #IMPLIED
                  para3           CDATA #IMPLIED
                  para4           CDATA #IMPLIED
                  para5           CDATA #IMPLIED>

<!ATTLIST param   type             CDATA #IMPLIED
                   name            CDATA #IMPLIED
                   value           CDATA #REQUIRED>
```

Figure 4 Document Type Definition (DTD) of test documents and test reports

The processing and distribution of test data within Tempto is performed by object-oriented middleware components which are implemented in Java:

- Respondeo [14] is a lightweight, message-oriented application server which integrates any kind of source system with information retrieval capabilities. It enables the generation of configurable collections of individual, heterogeneous source systems.

  Respondeo is used within the test management infrastructure of Tempto as a multithreaded server component for interfacing to the XML-documents in a multi-tier client/server fashion, as depicted in Figure 5. Each client (e.g. an offline project progress indicator) sets up a query which is wrapped into a generic message container. Then the message is sent to Respondeo which in turn dispatches the incoming message to the appropriate application object being in charge of handling test information of a particular application.

- Mitto, a message-oriented middleware, is in charge of distributing the generated test information in a scalable and transparent way. Mitto's primary communication paradigm is based on the publish/subscribe mechanism as

opposed to the traditional request/response protocol. Mitto is lightweight in that it is small in code size and only provides a subset of the functionality specified in the Java Message Service. It provides an object-oriented notification infrastructure for supporting recipients in the interpretation of the received message. For example, entire code fragments can be included in the event notification. This obviates any additional handshake with the sender of the event. In practice, every client can have its own dedicated message filter which is dynamically uploaded to Mitto. Additionally any type of message and event, i.e. messages containing XML-documents, can be transmitted as long as it is serializable in Java. In general, Mitto serves just as a lightweight execution environment for distributing events.

Each party and client respectively, which are subscribed to the particular information channel, are notified by Mitto as soon as the specified test information is generated and processed. For example, in Figure 5 a local Mitto Listener is in charge of publishing the test information to Mitto.

The level and granularity of test information (XML-documents) which reflect the party's interest, can be specified individually by applying the filter mechanism of Mitto.

- Specto [15] is a Web-enabled, scalable object-oriented infrastructure for the real-time and offline monitoring of applications in distributed systems. Specto introduces a common Document Type Definition (DTD) for managing state information of applications which might be used on various platforms and implemented in various programming languages.

  The Specto-DTD, as illustrated in Figure 4, forms the basis for capturing test documents in Tempto. Conceptually each test information in Tempto is an XML-document which conforms to the Specto-DTD. A Cerberus (e.g. Local Mitto Listener), as depicted in Figure 5, serves as a kind of watchdog for the XML-repository. Cerberus acts as a publisher by pushing the generated test information into Mitto's channel without knowing how many and what kind of parties are interested in its published test information. It is up to Mitto to propagate and transmit the published information to the respective subscribers or parties.

  The presentation of test documents is fulfilled by an object-oriented System Management Framework (SMF)[17]. The SMF provides graphical user interface components for querying and searching XML-based test documents, as well as so-called hitlists which are returned by Respondeo and Mitto. Each hitlist contains a list of test documents which are visualized within a table.

In the following sections, we describe the XML-based test repository and the integration of external management components into Tempto in more detail.

2.2.1 XML-Based Test Repository

Using an XML-based repository for storing and managing test information of various different applications aims at the vision of a general knowledge base for application-independent test information. The design of the knowledge base paves the ground for the support and integration of external components such as logic-oriented expert systems. This kind of inference systems refine and enrich the knowledge base by inserting additional, semantic information into it which is not necessarily directly related to the application domain of the producer of the test information. For instance, a statistical tool might provide some statistical information about the observed failures in a broader development context. Thus, in addition to the test reports which are based on the core test framework, as described above, the developer is able to retrieve a statistical report about how often a particular test case has been applied or which developers are actively using a particular test case.

As already mentioned in the previous section, each generated test information in Tempto is modeled as an XML-document based on the DTD introduced by Specto (see Figure 4). This DTD fulfills the requirements of generality and expressiveness of test information and test reports in Tempto. Thus it supports the extensibility of the overall infrastructure because it is not bound to a particular application domain and programming language, as long as there is an application programmer interface for generating XML-documents.

2.2.2 Integration of External Management Components into Tempto

The integration and interoperation of external components such as expert systems provide flexible and extensible management facilities in Tempto. These enhanced management capabilities serve as value-added components for Tempto although they are completely decoupled from the core framework components of Tempto. A value-added component is a component which is not directly related to the core framework of Tempto. However, it offers additional application facilities and services which are useful for the management of test documents and can be deployed within a distributed environment. From a system architectural point of view, such a value-added component is integrated into Tempto by interfacing to Mitto and Respondeo, respectively. These application middleware services decouple the value-added component from the rest of Tempto due to a uniform communication bus which is based on messages [16].

An example of such a value-added component is an expert system which actively supports the user (e.g. test administrator or framework developer) by analyzing the test reports. That is only possible due to the establishment of a test information repository which the expert system as an external component is able to interface and connect to and to deploy structural information retrieval mechanisms in a uniform way.

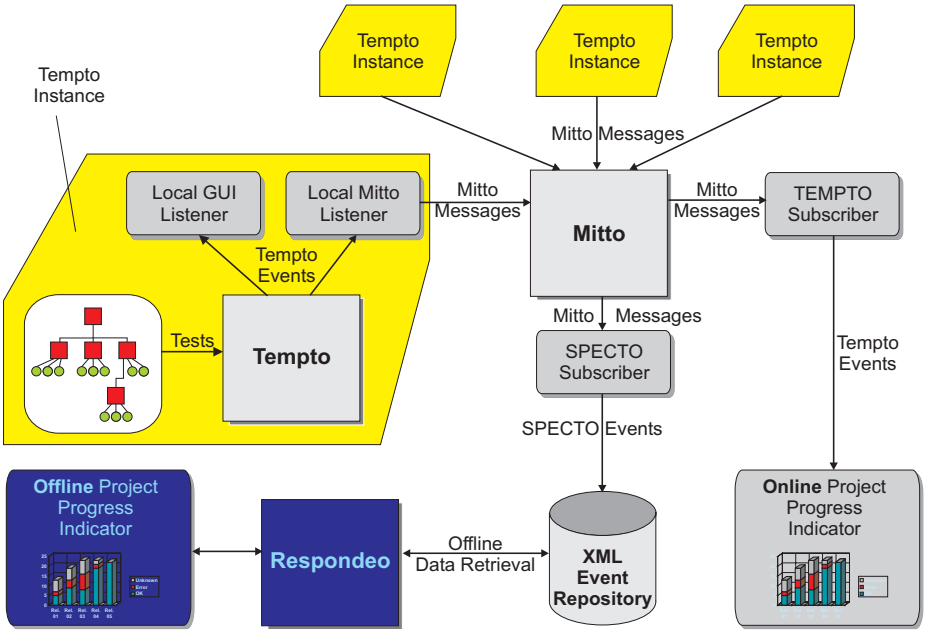


Figure 5 Test Result Management

3 Related Work

There has been a remarkable amount of research in the area of automated software testing. To derive the test data, often a formal specification or a description of the software as a finite state machine is used. For example the OzTEST test system [6] uses specifications in Object-Z [4] to derive test cases and to verify the results. From a theoretical point of view, as the formal specification generally allows automatic generation of test cases for the corresponding implementation as well as the generation of expected results, a very good level of overall automation can be achieved. However, the required formal specification is rarely provided for industrial software. More recently, research has been done to support the generation of test cases without a formal specification, based on knowledge of the source code and the intended behavior. Tempto is a framework that chose this approach. The same approach is chosen by Roast, a test tool developed at Monash University, Australia [3]. Roast supports the creation of Java unit tests, using a proprietary syntax for creating the "test drivers", which contain the control data which is used to check for errors and generate warnings. In contrast to Roast, Tempto (like JUnit) uses unit tests that are entirely written in Java, avoiding the limitation of any additional syntax.

Acceptance tests are treated differently by Tempto, as they are not closely related to the code but to the specification of the software. However Tempto will not rely on formal specifications in order to generate functional tests. At the functional test level,

it focuses on the result validation and test result management, leaving the generation of the acceptance test code to the tester.

As far as the management facilities of an integrated, distributed test environment are concerned, there are some approaches to store all kinds of test information, such as test cases and test programs, in a persistent repository. Whereas Xu et al. [19] use an object-relational knowledge base, Ohara et al. [12] build a integrated test and evaluation environment which is based on a relational database. Both of these approaches propose dedicated tools and components, such as a source code parser and analyzer, which are tailored to their particular test environment. In Tempto, we pursue a more interoperable and extensible approach. Tempto provides a platform- and system-independent XML-based repository to make all kind of test information persistent. In addition, by providing sophisticated object-oriented middleware components, the integration of external components into Tempto does not interfere with the rest of Tempto and of components which have already been integrated. Therefore, Tempto offers a scalable and extensible distributed infrastructure for building an integrated test and evaluation environment.

4 Summary

In this paper, we presented an object-oriented test framework that supports developing and managing of unit and acceptance tests in Java. Using JUnit as the starting point, we developed some enhancements in order to support parallel execution of tests and to speed up the test run even when run sequentially. Also, we improved the event handling management by separating the listeners from each other and adding user definable events to gain more detailed test reports. Various smaller changes have been made to improve the ease of use for the test developer.

Furthermore, we discussed a distributed test management infrastructure which transforms ordinary test data into well-structured XML-documents. Handling test data as fine-grained XML-documents supports the establishment of an extensible knowledge base for testing object-oriented applications. The distributed infrastructure consists of object-oriented middleware components which satisfy requirements within distributed systems such as scalability and heterogeneity of the participating components.

From an implementation point of view, the integration of the core test framework into the test management environment is ongoing work. The core test framework is currently used by the developer group of debis Systemhaus Industry DMS/PP in Germany.

Acknowledgements

We would like to acknowledge Erich Gamma and Kent Beck, who developed the JUnit framework on which Tempto is based, and made the sources freely available. We also wish to thank the members of the Java development team at debis Systemhaus Industry DMS/PP (Document Management Solutions) for their contribution in the practical evaluation of Tempto and their invaluable feedback.

References

1. K. Beck. eXtreme Programming explained. *Addison-Wesley, Reading, Massachusetts, 1999.*
2. K. Beck, E. Gamma. JUnit 3.2, online available at *http://www.xprogramming.com/software.htm*
3. N. Daley, D. Hoffman, P. Strooper. Unit Operations for Automated Class Testing. *Technical report 00-04*, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, January 2000.
4. R. Duke, G. Rose, G. Smith. Object-Z: A Specification Language Advocated for the Description of Standards. *Technical report 94-45*, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, December 1994.
5. M. Fayad, D. Schmidt, R. Johnson. Building Application Frameworks. *Wiley Computer Publishing, New York*, 1999.
6. R. Fletcher, A. Sajeev. A Framework for Testing Object Oriented Software Using Formal Specifications. In *Proceedings of the International Conference on Reliable Software Technologies: Ada-Europe*, Lecture Notes in Computer Science, June 1996.
7. M. Fowler. Refactoring - Improving the Design of Existing Code, *Addison-Wesley, Reading, Massachusetts*, 1999.
8. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. *Addison-Wesley, Reading, Mass.*, 1995
9. J. Gosling, K. Arnold. The Java Programming Language. In *The Java Series, Addison-Wesley, Reading, Mass.*, 1996
10. R. Jeffries. eXtreme Testing. In *Testing & Quality Engineering Magazine*, March/April 1999.
11. R. Johnson, B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, p. 22-35.
12. S. Ohara, l. Hu, M. Hui, T. Wang, P. Sheu. A Database Approach to Testing and Evaluating of Object-Oriented Programs. In *Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems*, November, 1999, Monterey, USA.

13. D. Roberts, R. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. Presented at PLoP 1996, online available at *http://st-www.cs.uiuc.edu/users/droberts/evolve.html*

14. R. Schimkat, S. Müller, W. Küchlin, R. Krautter. A Lightweight, Message-Oriented Application Server for the WWW. *In Proceedings of the ACM Symposium of Applied Computing*, March 2000, Como, Italy. Association of Computing Machinery.

15. R. Schimkat, M. Häußer, W. Küchlin, R. Krautter. Web Application Middleware to Support XML-Based Monitoring in Distributed Systems. In *Proceedings of the 13th Intl. Conference on Computer Applications in Industry and Engineering (CAINE-2000)*, November 2000, Hawaii, USA.

16. R. Schimkat, G. Nusser, D. Bühler. Scalability and Interoperability in Service-Centric Architectures for the Web. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA 2000)*, September 2000, London, UK. IEEE Computer Society Press.

17. S. Schroeder. Design of an extensible System Management Framework. *Master Thesis*, Univeristy of Tübingen, Germany.

18. World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-xml. Extensible Markup Language (XML) 1.0, 1998.

19. M. Xu, T. Wang, P. Sheu. Using an Object-Relational Knowledge Base for Software Testing. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, March, 1999, Richardson, Texas, USA.