

Framework for Extensible Application Testing

Agnieszka Zielińska

Computer Science Institute, AGH University of Science and technology, Kraków, Poland.
Email: agnieszka.zielinska@ubs.com

Received February 26th, 2012; revised March 27th, 2012; accepted April 29th, 2012

ABSTRACT

In recent years large corporations as well as smaller commercial enterprises have begun to devote increased attention to software testing and software quality. This paper introduces a novel tool—the Framework for Extensible Application Testing (FEAT), implemented by the author and applicable for automatic generation and execution of test cases. The paper discusses system requirements, design, architecture and modes of operation. It also contains a detailed comparison of the FEAT framework with existing test environments, focusing in particular on the STAF/STAX framework. The final section is devoted to experimental research into the applicability and efficiency of the presented tools in various projects and configurations, as reflected by quality metrics.

Keywords: Software Testing; Quality Assurance; Testing Automation; Framework for Extensible Application Testing (FEAT)

1. Introduction

In recent years, commercial entities dealing with component software development have shown increased concern with software testing and software quality issues. Tests need to be frequent, thorough and extensive; moreover they need to cover a wide range of use cases and potential applications of software components. Testing processes employ—with ever-increasing frequency—automated tools and frameworks, enabling developers to reduce the complexity of software solutions and ensure their optimal quality. Modern testing environments carry many benefits, yet they are not free of certain drawbacks and limitations such as language lock-in, poor support for inter-scenario dependencies, lack of parallel testing mechanisms for distributed architectures and lack of centralized test resource management tools.

This paper introduces a novel testing environment called FEAT (Framework for Extensible Application Testing) developed by the author and capable of automatically testing component software. The environment is readily adjustable and can be customized to match the requirements of various testing teams. It boasts many features which aren't found in popular testing suites currently available on the market.

The first section of the paper discusses basic issues relevant to software testing. This is followed by a discussion of testing principles and a general description of the testing process itself, where we distinguish specific modes of testing. Following this discussion, we present the key aims of the FEAT automatic testing environment

—the author's main contribution to the field. We also discuss the general premises on which the system is based as well as its architecture. An introduction to the development of test scenarios is presented, along with a description of test execution in the proposed environment [1].

Section 4 opens the experimental part of the paper, comparing the features of FEAT with those provided by popular testing frameworks such as JUnit and TestNG, as well as advanced environments, including STAF/STAX. We discuss the advantages and drawbacks of each solution and present the concepts which have been adopted and further elaborated by the author in the scope of FEAT.

Having presented a conceptual outline of the FEAT framework we follow through with an analysis of the tool's applicability to two specific IT projects, differing with respect to the size of development teams and budgets. The results of this study are presented in Section 5.

The paper ends with a recapitulation of the project's goals, conclusions drawn from the above-mentioned use cases and prospects for further development of the FEAT environment.

2. Basic Concepts Related to Software Testing

The primary goal of software testing is to identify any defects which may be present in the system being developed. A defect may occur during any phase of the software design, implementation and maintenance process. It

may result from coding errors (also called bugs), omissions or misinterpretations of the required functionality on the part of system developers [2].

Testing is a crucial element of software development since it guarantees that the application will perform as expected in all circumstances. Some testing methodologies actually extend this definition by checking whether the application doesn't do more than is expected of it. In either case, testing works to protect the user from software errors which may—depending on the circumstances—result in loss of time, financial resources or even human lives [3].

2.1. The Basic Principles of Testing

Testing typically follows the schema outlined in **Figure 1**. This standard is widely applied throughout the IT world, and consists of a set of rules defined by respected authorities in the field. For instance, Davis mentions the following principles of software testing [4]:

- All tests must reflect client requirements.
- Testing should be planned far in advance of its actual commencement.
- Testing should follow the Pareto principle (80/20): according to this observation 80% of all software errors are caused by 20% of software components. It is the goal of tests to identify these critical components and thoroughly verify their correctness.
- Testing should begin with general assumptions and then progress to specific criteria.
- Exhaustive testing is impractical.
- Tests yield better results when conducted by third parties.

2.2. Types of Tests

Two different approaches may be applied to software testing. The first is to conduct functional tests, *i.e.* view the system from the user's perspective and treat it as a "black box" which is expected to perform certain tasks. In this case the tester is not concerned with the inner

workings of the software being tested.

Not surprisingly, such tests are also referred to as black box testing [4]. The other approach bases on structural tests, where the tester has access to source code and may observe the behavior of individual components of an application (such as modules and libraries). This mode is sometimes called white box testing. A typical example of structural testing involves unit tests where the tester (or the programmer) develops additional code used solely to verify the operation of a specific portion of the application's production code [4].

Tests may be conducted manually, by the tester, with the use of the application's user interface and a checklist of actions, or automatically, where there is no need for a human actor.

While both of the presented divisions focus on the means of conducting tests, another distinction can be introduced by considering the scope of testing [5]:

- Unit tests;
- Integration tests;
- System tests.

There is also a division of tests with respect to their purpose, which we view as the most interesting of all testing hierarchies. It permits us to select tests suitable for a given goal. This division is outlined below [6]:

- Acceptance tests;
- Functional tests;
- Regression tests;
- Performance and stress tests;
- Installation and configuration tests;
- Alpha and beta version tests;
- Usability tests;
- Failure recovery tests.

It should be noted that each development phase of an IT project can be related to a specific testing phase. This relationship is outlined in **Figure 2**.

3. The FEAT Component Application Testing System

The initial requirements for our testing tool were gath-

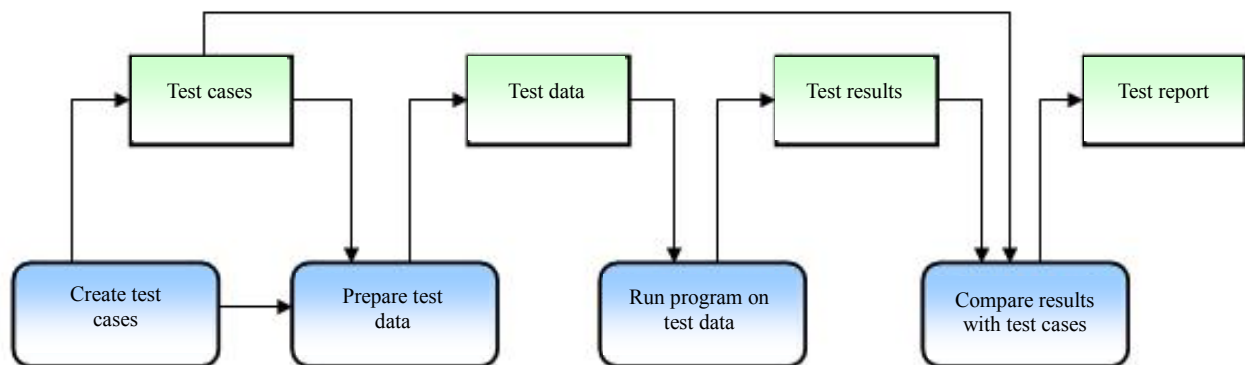


Figure 1. Software testing [4].

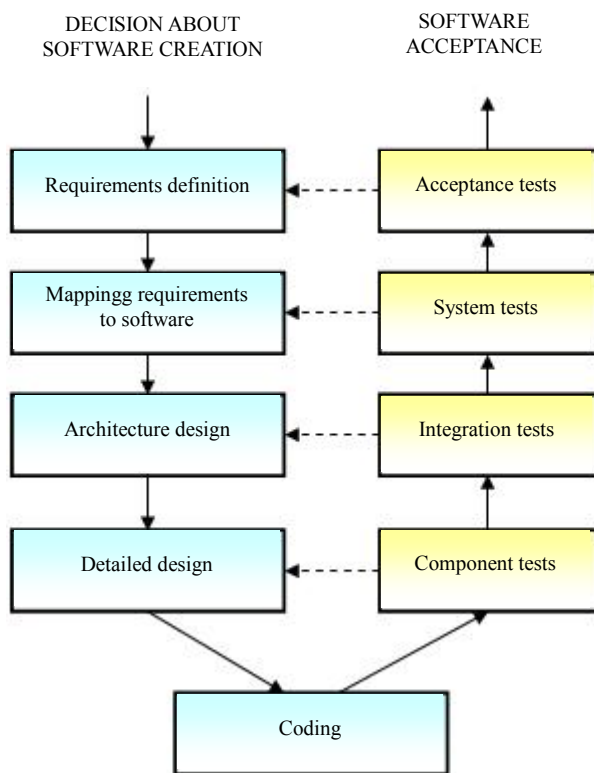


Figure 2. The relationship between project development and testing phases [4].

ered by studying the habits and opinions of teams developing component software in a major American corporation [7]. The presented environment can be applied by various corporate development teams to conduct automated software testing.

Following a thorough analysis of requirements we have formulated some basic assumptions which underpin the FEAT environment, presented in **Figure 3**:

- 1) The environment is to be written in Java;
- 2) The environment must enable easy, automated testing of distributed Java-based component applications with focus on simplicity and reusability of tests;
- 3) The environment must enable easy, automated testing of distributed Java-based component applications with focus on simplicity and reusability of tests;
- 4) The environment should focus on testing the APIs exposed by containers and components deployed in such containers. Supported containers should include J2EE and OSGI application servers as well as any container which provides a Java VM-based platform for pluggable software modules;
- 5) FEAT works by wrapping a test engine in a pluggable module so that it matches the container technology in use. Thus, each type of container should correspond to a specific wrapping;
- 6) The proposed environment does not approximate a

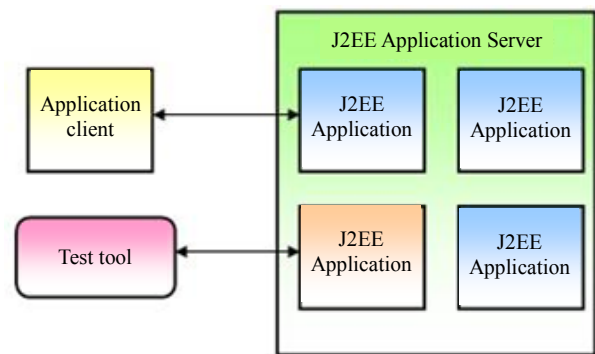


Figure 3. Generalized vision of the placement of the proposed FEAT testing environment [1].

JUnitEE-type system as unit tests are, by definition, simple and should not depend on external tools. Rather, the tests which can be conducted using FEAT can be arbitrarily complex and scalable;

7) The author focuses on developing universal components from which tests can be constructed and which can themselves be reused in various projects. Test developers must be prevented from creating implicit dependencies between tests;

8) The environment should support a flexible hierarchy of tests and permit results to be archived in ways appropriate for a given IT project and selected testing methodology;

9) When creating a specific test case, the developer should focus on abstract actions (e.g. install a product; verify the integrity of a database etc.) rather than on specific, platform-dependent implementations (taking into account the operating system, DBMS, application server etc.) In other words, the environment should conceal technical details from the user, relegating them to a separate (hidden) layer;

10) The system should ensure access to the test execution environment and facilitate interaction of tests with this environment (contrary to unit tests, which do not require access to such an environment);

11) The system should provide a selection of useful auxiliary features such as file downloads, logging, result storage etc.;

12) The system should feature advanced visualization of results and generation of testing reports;

13) The system should introduce a mechanism for automatic removal of test engines and/or other resources (e.g. databases), restoring the initial state of the tested environment;

14) The system should support pluggable installation and execution layers (such as SSH and STAF), responsible for deploying test engines on remote machines. The transport layers used for remote execution of tests should also be pluggable (e.g. RMI, servlets, custom protocols based on serializable Java objects etc.) [1].

3.1. System Overview

A general overview of the architecture of the FEAT testing environment is presented in **Figure 4**. Below we describe the basic principles of its operation [1]:

- The **Controller** receives test Scenarios;
- The **Scenerio Manager** scenarios (which may be written in an arbitrary programming language) and converts them to Java classes;
- Classes are forwarded to the **Test Execution Core** which executes the scenarios by invoking test fragments on appropriate test engines, deployed on remote hosts;
- The **Resource Manager** should gather information on the resources available for testing, including hosts, databases, application servers etc. The **Test Execution Core** should reserve selected resources (on demand) and assign them to a given scenario (by supplying its description). Resource access is query-based: for example, the user may request a machine with DB2 v9.1 and WebSphere v6.1 installed. Thus, test developers only indicate generic requirements instead of selecting specific machines for testing;
- The **Resource Manager** may be extended with plugins and/or interface with an external management system (for the purposes of reserving access to hosts);
- Test **Engines** are installed on remote machines. Once

tests are complete, these engines are automatically uninstalled to restore the local environment to its initial state;

- Test **Engines** execute test fragments and return results to the **Test Execution Core** from which they are dispatched to the **Results Manager**;
- The **Results Manager** may also be extended with plugins, for example to write results to files, custom databases etc.;
- The **Visualizer** be attached to the **Results Manager** in order to display graphical representations of test results.

The testing environment is configured in two phases. Phase one involves configuring the controller by determining which Resource Manager and which Result Manager it should interact with; which mode it should operate in and which services it should expose. This configuration does not directly influence tests. Phase two focuses on verifying the correctness of the test scenarios, checking their dependencies, determining the order in which tests should be executed etc.

3.2. System Architecture

One of the key aspects of FEAT is its layered architecture. Each layer is dedicated to a specific goal and may interact with other layers in pursuit of that goal. The lay-

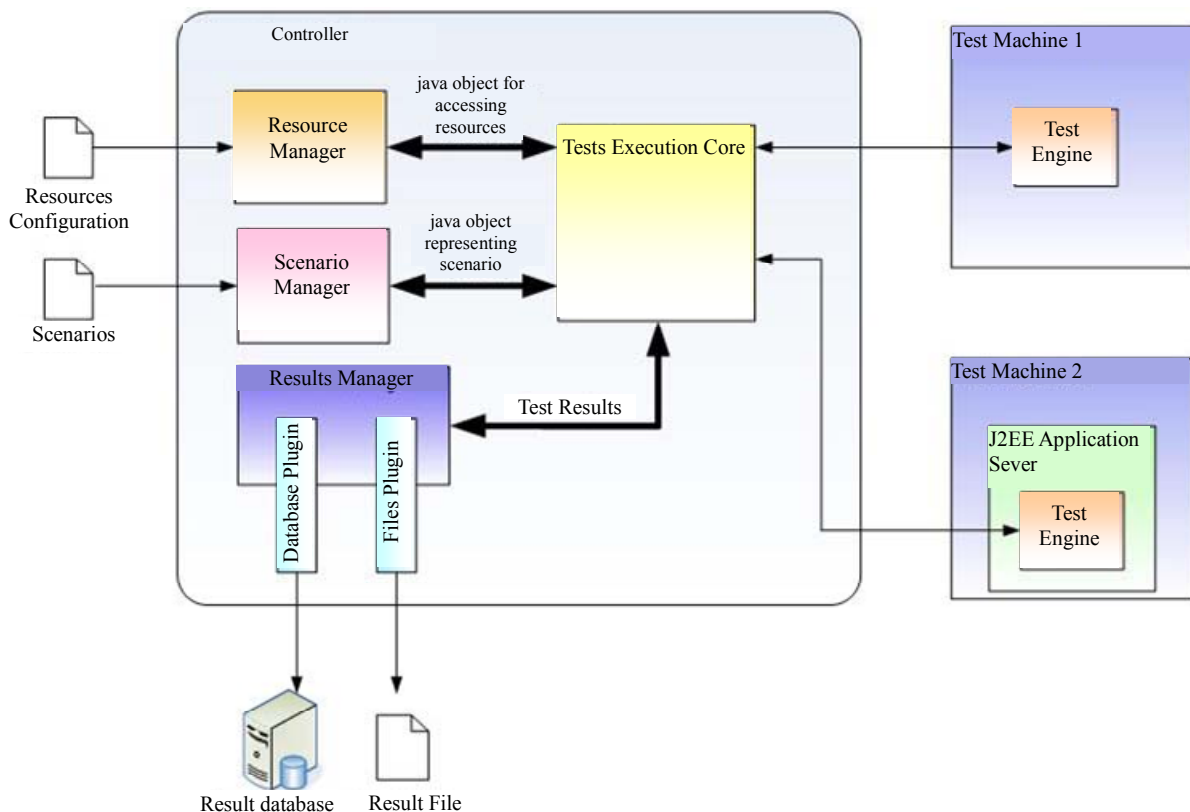


Figure 4. General architecture of the FEAT testing environment [1].

ers depicted in **Figure 5** are briefly discussed below:

- 1) **Transport Layer**: executing Java code on remote hosts;
- 2) **Deployment Layer**: installing test engines on remote hosts;
- 3) **Services**: services such as file download, file transfer etc.;
- 4) **Resources**: various resources, including test hosts, databases, application servers etc.;
- 5) **Test Engine API**: an API facilitating access to lower layers (Transport Layer, Deployment Layer, Services, Resources). It manages these layers and exposes their functionality to upper layers (Test fragments and installers and Test Scenarios);
- 6) **Test fragments and installers**: this layer executes test fragments (each test fragment is an atomic unit, independent of other fragments);
- 7) **Test Scenarios**: this layer executes entire test scenarios.

3.3. Development of Test Cases

At this point it seems appropriate to explain how test scenarios are created. Each test scenario corresponds to a specific test case and consists of the following elements

- Scenario identifier (ID);
- Scenario name;
- Textual scenario description;
- Dependent scenarios (which need to be executed prior to the given scenario);
- Programming language in which the scenario has been implemented;
- Name of class implementing the scenario;
- Resources required for the test scenario to execute (resource name, resource class and—optionally—other parameters such as version number).

A sample XML file describing two simple scenarios is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite>
```

```

    <scenario                                id="scen_20"
    name="Scenario1">
    <description>Test if database is cor-
    rectly installed</description>
    <code
    type="java">mytests.TestDBConn</code>
    <resource name="DB"
    class="feat.resources.Database"      re-
    quired="true" />
    </scenario>
    <scenario id="scen_21" name="Scenario2"
    depends="scen_20" >
    <description>This scenario tests
    creation of database
    schema</description>
    <code
    type="jython">dbscenario.py</code>
    <resource                        name="DB"
    class="feat.resources.Database"  re-
    quired="true">
    <property name="type">DB2</property>
    <property name="version">9.1</property>
    </resource>
    </scenario>
</testsuite>
```

The `<testsuite>` tag indicates a test suite and contains a separate `<scenario>` tag for each scenario which belongs to the given suite. The `<description>` tag contains a textual description of the scenario. Each scenario must also specify a list of required resources by supplying their abstract names. For instance,

```

    <resource name="DB"
    class="feat.resources.Database"
    required="true" />
```

means a database with the name DB2, managed by the `feat.resources.Database` class. This database is required for the test scenario to operate (`required="true"`).

In addition, each scenario follows a specific course of action which may be described using various languages

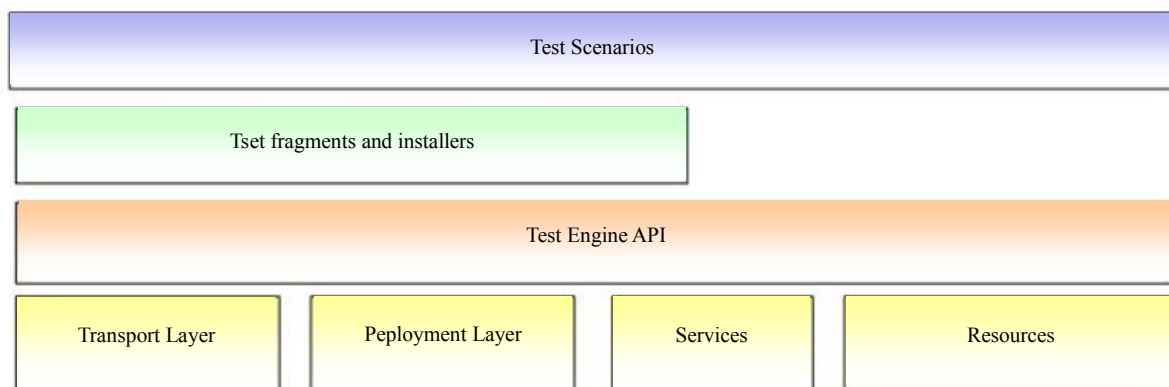


Figure 5. Layered architecture of the FEAT testing environment [1].

(for example, by a Java class or a Jython script). The required language and the name of the scenario file are given as:

```
<code>
type="jython">dbscenario.py</code>
```

Test scenarios may be bundled in test suites and may also specify internal dependencies (for instance, "if a given scenario fails, do not execute these dependent scenarios"). Such dependencies are expressed via the depends attribute in the scenario description. For example, the following description

```
<scenario id="scen_21"
name="Scenario2" depends="scen_20" >
```

3.4. Test Execution Process

Given the proposed architecture, automatic execution of tests in the FEAT environment proceeds according to the following plan [1]:

1. Controller initialization:

- 1.1. Loading resources.
- 1.2. Installation of test engines on remote hosts.
2. Scenario enactment:
 - 2.1. Loading scenario.
 - 2.2. Converting scenario to Java object.
 - 2.3. Enactment of the next waiting scenario:
 - 2.3.1. Allocating resources.
 - 2.3.2. Retrieving/creating test fragments.
 - 2.3.3. Executing test fragments on test engines.
 - 2.3.4. Retrieving results.
 - 2.3.5. Uninstalling test engines installed by the scenar.
 - 2.4. Collating scenario results.
3. Closing and uninstalling engines.

3.5. Testing Progress Report

While test scenarios and test fragments are executing on remote hosts, it is possible to monitor their progress and results via a Web-based interface. To this end, the Jetty

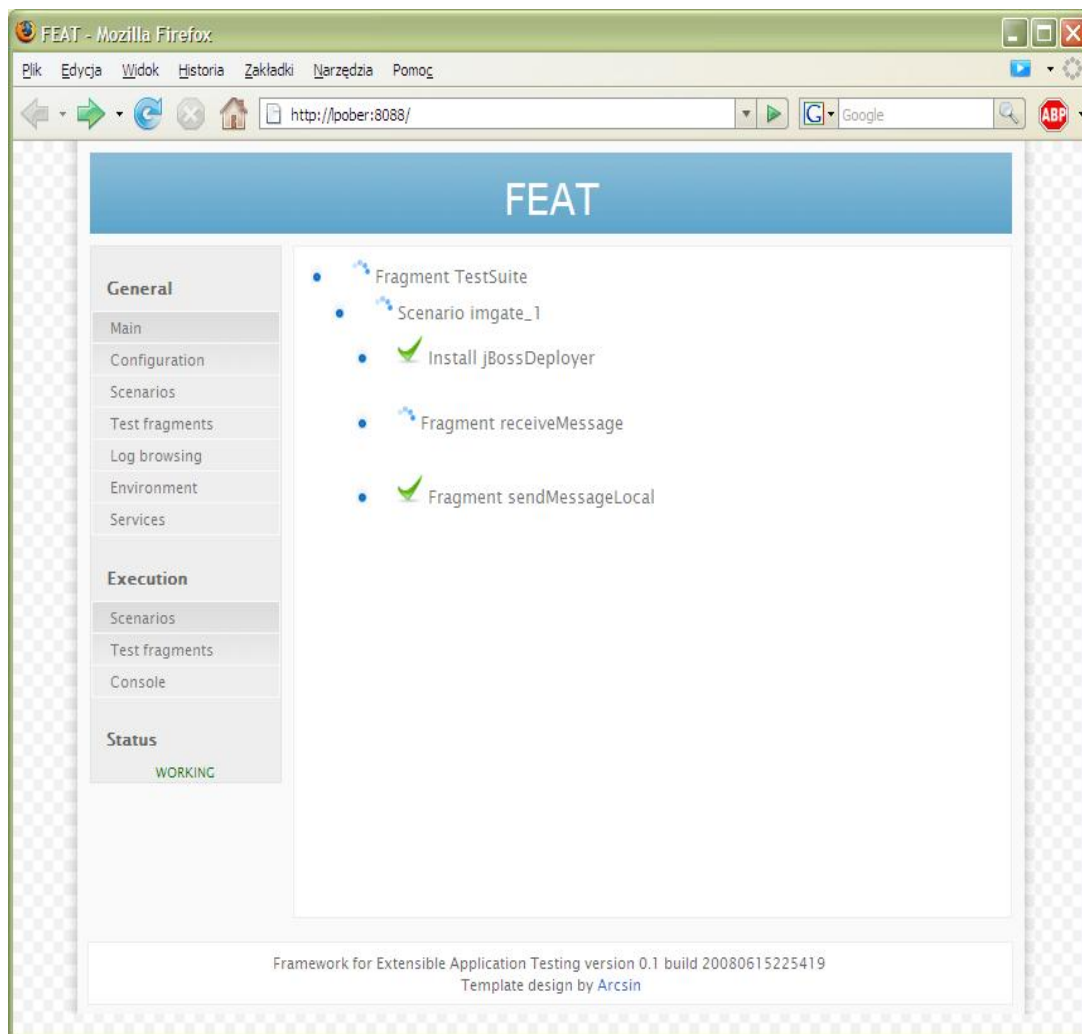


Figure 6. Testing progress report [1].

web server is used to serve a website at <http://localhost:8088>. Key configuration info is available under the General tab:

- **Main:** information about the system on which tests are being executed: version of the FEAT testing environment, build version of the tested application, elapsed testing time, ID and full name of the operating system, Java environment version, JAVA_HOME value, total RAM, free RAM and number of available processor cores;
- **Scenarios:** information on the executing test scenarios: scenario ID, programming language (Java, Python etc.), scenario name;
- **Test Fragments:** information on individual test fragments which make up the scenarios: fragment name, class, platform, resources utilized;
- **Execution:** information on the current state of the executing scenarios (Scenarios) and test fragments (Test Fragments). This menu also enables users to view execution logs (Log browsing) (Figure 6).

4. Comparison of JUnit, TestNG, STAF/STAX and FEAT Tools

Let's assume that we have been tasked with testing component software developed by a large corporation. The development team consists of experts from around the world, operating in various time zones. Let us furthermore assume that a test build of the software is created daily around 1:00 a.m. Tests are to be divided into the following phases [1]:

- **BVT** (Build Verification Tests)—performed each time a new build is deployed;
- **CVT** (Component Verification Tests)—if BVT pass successfully. CVT consist of regression tests conducted on a range of hosts with different configura-

tions;

Which tool—JUnit, TestNG, STAF/STAX or FEAT—would ensure the most efficient, most reliable and fastest testing under the proposed conditions? In order to answer this question, let's analyze the information presented in Figure 7, which lists individual testing stages and the associated benefits of each testing environment.

As can be seen, the highest efficiency—both for BVT and CVT—is provided by the FEAT environment. Tests can be performed automatically, with no human interaction, and may occur at any time (e.g. immediately upon the deployment of a new build, even late at night). When the testers come to work on the following day, they can peruse an automatically generated report detailing the conducted tests. It should be noted that the FEAT environment combines the benefits of the solutions presented in the previous sections. It derives its support for small test fragments from JUnit and inherits support for test suites from TestNG. In addition, it enables automatic execution of individual steps of the testing process; thus it shares many features with the STAF/STAX environments frequently used in large corporations [8].

It is the goal of FEAT to fully automatize testing, including preparation of the testing environment. This is why the author has extended the proposed tool with additional features such as automatic installation and execution, file transfers, automatic allocation of testing resources and centralized logging.

We can therefore conclude that the FEAT environment exhibits all the attributes of a self-contained automatic software testing solution and that—in terms of novel features—it compares favorably to the environments and tools currently available on the market. It can be expected that the concepts upon which FEAT bases will one day become standard practice in commercial and open-

	JUnit	TestNG	STAF/STAX	FEAT
Small, reusable test fragments	YES	YES	YES	YES
Development of complicated scenarios	NO	YES	YES	YES
Dependent tests	NO	YES	YES	YES
Complex test scenarios	NO	YES	YES	YES
Development of tests in multiple languages	NO	NO	YES	YES
Complex test execution	NO	YES	YES	YES
Assignment of test machines	manual	manual	manual	automatic
File transfer between machines	manual	manual	manual	automatic
Installation of new application version for tests	manual	manual	manual	automatic
Allocation of test resources	manual	manual	manual	automatic
Test execution	manual	manual	automatic	automatic
Test monitoring	manual	manual	automatic	automatic
Logs and result analysis	manual	manual	manual	automatic
Support for exchangeable modules	NO	NO	YES	YES

Figure 7. Comparison of JUnit, TestNG, STAF/STAX and FEAT [8,9].

source testing solutions, including those employed by large IT enterprises.

Comparison of Test Implementations in STAF/STAX and FEAT Environments

Compared to STAF, FEAT is better suited for Java-based environments since it does not require additional virtual machines to operate. As FEAT may be executed in any Java virtual machine, no additional code is required to perform mediation between the internal interface of the tested application and the testing environment itself.

STAF only supports communication via basic strings and lists. In contrast, FEAT enables transfer of configuration parameters, data and test results as serialized Java objects with arbitrary internal structure. During development of tests, syntactic verification occurs at the compilation stage, and not—as in STAF—at runtime [10].

When preparing tests in an IDE (such as Eclipse) the programmer may take advantage of the code autocompletion feature to find out which arguments are required for a given test fragment. This reduces the need to consult documentation and increases code reusability.

STAF is a highly generic environment, built upon the notion of agents and services deployed therein. Efficient development of tests requires users to deploy an additional layer of testing logic suited for a particular project. In contrast, FEAT aims to support creation of tests “out of the box”. As the test suite grows and matures, it may be extended by adding (or exchanging) plugins, installers or fragments, usually with no need to rewrite code [10].

FEAT defines a selection of interfaces and contracts which need to be implemented by the test code. This feature results in reusable modules which can be repurposed not only within the context of a specific project, but in the scope of an entire enterprise (or branch thereof).

Software testing often calls for examining the same functionality under various hardware and software configurations (different operating systems, application containers, databases etc.) FEAT supports this goal by separating the generic test scenario from implementation- and configuration-specific details. Test fragments may be differentiated into versions, depending on the engine in which they’re executed. Moreover, each scenario may be executed on various sets of resources, which enriches the set of test configurations without calling upon the user to develop additional scenarios.

Finally, FEAT supports central management of resources used in tests. The Resource Manager module may be swapped while the central database remains unchanged. This feature mitigates the problem of having several testers share a single machine and interfere with one another’s work.

5. Experimental Assessment of the Manual Software Testing Process with STAF/STAX as Compared to FEAT

In the following subsection of our work, we will discuss the software testing process on the basis of two distinct use cases [1]:

- a small enterprise employing up to 5 developers and testers with overlapping areas of responsibility, applying the MSF methodology.
- a large corporation with a well-defined organizational structure and clearly delineated developer/tester responsibilities, applying the RUP software development methodology. This use case is based on information acquired from a large real-world international corporation.

For each of the presented cases we will briefly describe the testing methodology, development team composition and software being produced. Following a preliminary analysis of the project team, we will assess the time required to manually perform tests using STAF/STAX and FEAT. This will enable us to compare the cost-effectiveness of test preparation and execution phases.

Our analysis will yield conclusions regarding the applicability and effectiveness of selected tools for different teams and software projects.

5.1. Quality Metrics

Published works [11,12] propose a great variety of metrics and quality assessment methods which may be applied by gauging the cost and expected duration of a project—e.g. COCOMO (COConstructive COSt MOdel) or the function points method. For the purposes of the presented work, we will only discuss some basic quality metrics and means to determine the percentage of undiscovered errors. Particular attention should be devoted to the quality of the testing process itself—its effectiveness and the reliability of the results it yields – as these values enable us to determine the quality of the software being tested.

The following metrics may be used when assessing the quality of testing [13]:

- **Requirements Coverage metric:** numerical assessment of the portion of requirements which have successfully passed tests. The following formula applies:

$$RC = \text{no. of requirements } (P, I, E, S) / \text{no. of requirements } (A, T).$$

This metric may be derived at many stages in the testing process, to determine the coverage of code by planned (P), implemented (I) or executed (E) tests. Moreover, it can be used to assess how many requirements have passed tests successfully (S). This value may be compared to the total number of requirements (A) or to the number of requirements which were subjected to testing (T). In the context of testing quality, the requirements

coverage metric can serve as the coefficient of tested requirements, namely:

$RC = \text{no. of tested requirements (E) / total no. of requirements (A)}$.

- **Code Coverage metric:** this metric enables us to determine the fraction of source code subjected to testing. It is given by the following formula:

$CC = \text{no. of executed code units / total no. of code units}$.

The number of code units is defined as the number of lines of code or, alternatively, as the number of potential control paths that can be traversed. Source code should be understood as an expression of the developer's intentions; however it must be noted that even complete code coverage and absence of errors do not warrant a good match between the resulting software and the end user's requirements.

- **Defect Removal Effectiveness metric**, usually given as:
 $DRE = \text{no. of errors identified during testing / total no. of identified errors}$.

The total number of identified errors is the sum of the number of errors identified during testing and the number of errors identified following testing (*i.e.* during deployment or during production runs involving the end user).

This metric indicates how efficient the testing process is in identifying errors; however in order to accurately derive it we must remain aware of how many errors are identified by end users. Thus, our assessment must be based on historical data and take into account the experiences of similarly competent testing teams working on similar projects with the use of similar testing methodologies.

The Defect Removal Effectiveness metric is applied by the CMM (Capability Maturity Model) to assess the maturity of an IT enterprise.

5.2. Case Study 1: Small Enterprise

5.2.1. Team Description

The small enterprise development team consists of 8 persons, one of whom plays the role of product manager, project manager and release manager. That person is responsible for planning, estimating the duration of project phases, assigning tasks, overall project management and release deployment. Four persons are employed as developers while the three remaining team members are testers who nevertheless actively participate in development (due to manpower limitations). Their assignments may change depending on the circumstances. Each team member handles clearly defined tasks. Team roles are mutually dependent and shared. The team applies the MSD (Microsoft Solution Framework) methodology.

5.2.2. Project Outline

The project is driven by client demands and has a spe-

cific timeframe for each implementation phase as well as a final deadline for software delivery. The client is particularly concerned with meeting this deadline while ensuring high quality of the delivered software.

Unfortunately, due to the lack of sufficient manpower, the team finds it difficult to meet all of the client's requirements.

5.2.3. Methodology

The team applies the MSF (Microsoft Solution Framework) methodology, developed by Microsoft Consulting Services and known throughout the world as an established standard for the development of IT projects by Microsoft and its partners.

In general, the project can be considered a success if it is realized within budget and in agreement with the predetermined deadlines.

5.2.4. Tool Effectiveness Analysis

How should this type of project be handled, given the specified methodology, requirements and team composition? The most significant problem seems to be the relatively small size of the team and, in particular, the need to intermittently engage testers in development of additional features. The question arises: how to optimally exploit the available resources? The following analysis begins with a phased implementation plan.

First, let us assume that the team does not perform any automatic testing. All tests need to be conducted manually. Whenever new features are introduced, regression tests must take place, given the strong dependencies between any new features and existing system components. While such tests are being conducted, testers cannot participate in further development, due to their relatively high workload. Two different scenarios may be applied at this point:

- Testers focus on manual tests run in parallel with implementation. This ensures better product quality but at the cost of missing deadlines (testers cannot engage in development and the team is reduced to four active developers—cf. column 1 in **Figure 8**).
- A single tester focuses on testing while the remaining two collaborate with developers on implementation of new features. The project proceeds as scheduled, however it may fail to meet client requirements due to the relatively low testing coverage (column 2 in **Figure 8**).
- Thus, given a manual testing regimen, it seems unlikely that all the goals of the project can be met. Either the project will be delayed or the quality of the delivered software will be compromised.

Let us now consider the potential advantages of applying automatic testing tools in the scope of this project:

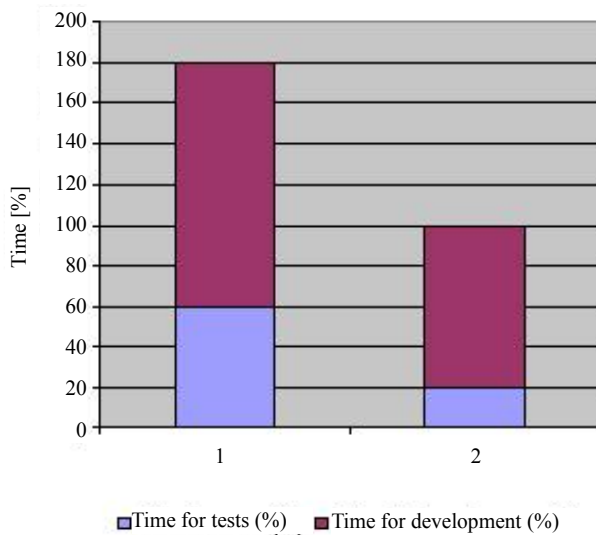


Figure 8. Comparison of project implementation time given manual (1) and automatic testing (2) [1].

- During the initial implementation phase all three testers design automatic tests of expected features, basing on the conceptual system design. Implementation proceeds slowly.
- Once the tests are implemented, testers may devote their attention to development of additional features while devoting only a small percentage of their workload to analysis of test results. This greatly speeds up further development.

Automatic tests can proceed in parallel with implementation throughout the duration of the project (column 2 in **Figure 9**). As testers are not required to personally execute and monitor tests, they can devote more of their time to implementation work.

Let us now analyze the project from the perspective of IT industry standards:

- Number of requirements: 150;
- Number of implemented classes: 80;
- Lines of code: 80 classes * ca. 1000 lines = 80000 lines;
- Lines of test code: 60 classes * ca. 1000 lines = 60000 lines;
- Under these conditions the implemented tests would cover approximately 90% of code, exposing approximately 94% of all errors and covering approximately 130 requirements;
- The quality metrics, as given by formulas presented in Section 5.1, would therefore equal:

$$RC = 130/150 = 0.86$$

$$CC = 72,000/80,000 = 0.9$$

$$DRE = 0.94$$

5.2.5. CONCLUSIONS

From the above analysis and the derived quality metrics

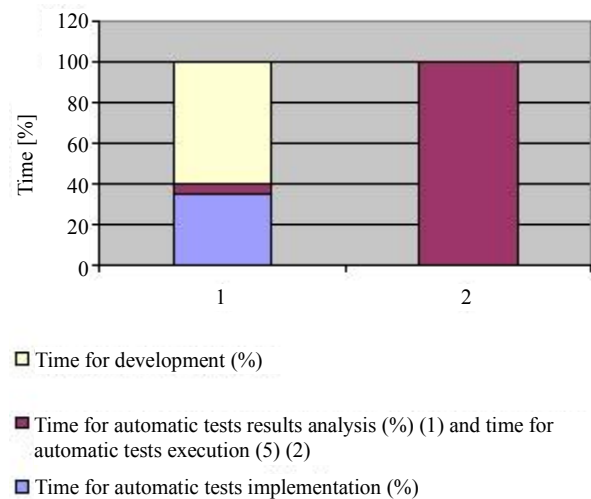


Figure 9. Manual tests: 1—deadline missed, acceptable software quality; 2—deadline met, insufficient software quality [1].

it follows that automatic testing would enable this project to be delivered on time, with no undue impact on software quality. Thus, automatic testing proves efficient and highly desirable from the business standpoint. No other testing environment seems to fulfill the stated requirements of this project, given the circumstances under which it is being implemented.

5.3. Case Study 2: Large Enterprise

5.3.1. Team Description

This use case bases upon data gathered while working with automatic testing tools in a large real-world corporation. The development team consists of developers, testers and a project manager. The role of each team member is explicitly defined. The team is geographically distributed; some members operate in Poland while others are based in the USA. The impact of the FEAT on business goals will be assessed based on data from two real-life IT development projects.

5.3.2. Project Outline

The goal of the project is to deliver a custom component to external clients. The team applies the RUP (Rational Unified Process) iterative methodology. Extensive testing of the product, covering as much of its functionality as possible, is critically important. Tests are to be divided into the following phases:

- BVT (Build Verification Tests)—performed each time a new build is deployed;
- CVT (Component Verification Tests)—performed if BVT pass successfully;
- IVT (Integration Verification Tests)—integration tests;
- SVT (System Verification Tests)—system-wide tests;

- RT (Regression Tests).

Proper testing coverage is a prerequisite of suitable QA (Quality Assurance).

5.3.3. Methodology

As mentioned above, the development team applies the RUP (Rational Unified Process) methodology, whose principles, advantages and drawbacks are outlined in Section 2 of this paper.

5.3.4. Tool Effectiveness Analysis

The analysis of the effectiveness of testing tools is based on real-world data acquired from a corporation:

- Test scenario: Selected scenario from the BVT;
- Condition: full automation of test scenario, including environment configuration, post-test cleanup and collation of test results for analysis;
- As depicted in **Figure 10**;
- Duration of a single test scenario:
 - ♦ when using SSH, JUnitEE and ant: 28 h—a significant portion of that time has to be devoted to proper implementation of the test, preparation of the testing environment and post-test cleanup;
 - ♦ when using the FEAT environment: 8.7 h;
- Time saved: 28 h - 8, 7 h = 19.3 h for a single test scenario;
- 50 requirements per functionality;
- 2 to 5 (average: 3.5) scenarios per requirement = 175 test scenarios;
- Number of implemented classes: 120;
- Lines of code: 120 classes * ca. 1000 lines = 120000 lines;
- Lines of test code: 200 classes * ca. 1000 lines = 200,000 lines;

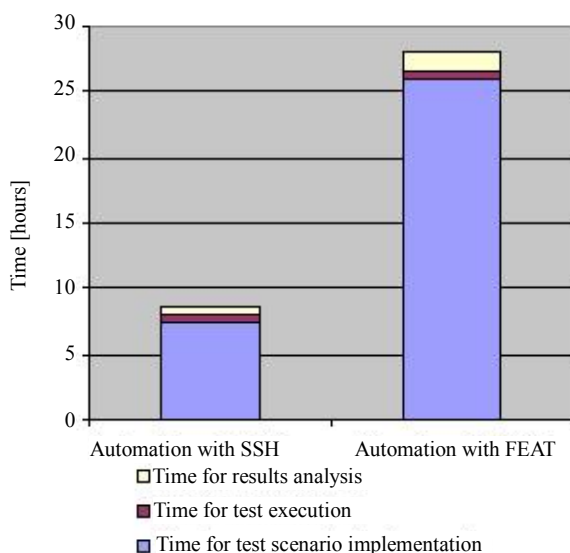


Figure 10. Manual tests with SSH and JUnit and with FEAT.

- Under these conditions the implemented tests would cover approximately 95% of code, exposing approximately 97% of all errors and covering approximately 48 requirements;

- The quality metrics, as given by formulas presented in Section 5.1, would therefore equal:

$$RC = 50/48 = 0.86$$

$$CC = 114,000/120,000 = 0.95$$

$$DRE = 0.97$$

- Business impact:

$$\blacklozenge 3377.5 \text{ h saved } (175 * 19.3 \text{ h});$$

$$\blacklozenge 422 \text{ PD (Project Days) or } 84 \text{ PW (Project Weeks) or } 21 \text{ PM (Project Months) or nearly } 1.8 \text{ PY (Project Years) saved};$$

$$\blacklozenge \text{ costs reduced by } 211\text{K USD (assuming } 120\text{KUSD/PY)}.$$

5.3.5. Conclusion

The above analysis suggests that FEAT-type automated testing environments carry advantages for large-scale IT projects as they ensure repeatability and increased quality of tests while saving time and reducing costs. This translates into significant improvements both in terms of quality metrics and business competitiveness.

6. Summary

The main goal of the presented work was to develop a custom environment called FEAT (Framework for Extensible Application Testing), which could be used to automatically test component software while ensuring high adaptability to the requirements of various testing teams. The presented environment can be applied to easily and automatically test distributed component applications written in Java, with focus on simplicity and reusability of tests. It focuses on tests of APIs provided by containers and components deployed therein. Supported technologies include J2EE and OSGI application servers and all other containers which provide a pluggable component-based system operating within the Java VM. The presented environment does not attempt to emulate JUnitEE as unit tests are, by definition, simple and should not call upon external dependencies. Tests conducted with the use of FEAT may be arbitrarily complex; however, we wish to stress the tester's ability to develop universal components which can be assembled into tests of various software systems. In addition, FEAT provides support for a flexible testing hierarchy where results can be logged and presented in accordance with the requirements of specific projects and specific testing methodologies. In developing a test case, the tester focuses on abstract actions (such as: "install a product" or "verify the integrity of a database") while omitting low-level implementation details, relating to given operating sys-

tems, database management systems or application servers. In other words, the proposed environment relegates all technical aspects to a separate layer which is concealed from the end user. FEAT provides a testing environment and ensures suitable interaction between this environment and the tests themselves (contrary to unit tests which do not require any such environment). Moreover, FEAT comes with a set of useful support services, including on-demand file transfer, logging, collating results etc. It also contains an advanced visualization subsystem which may be used to generate testing reports [1].

The FEAT environment combines the advantages of several other testing platforms mentioned in this paper. For instance, it derives the notion of small test fragments from JUnit and the ability to compose test suites from TestNG. In addition, it enables automatic execution of individual steps of the testing process thus it shares many concepts with the STAF/STAX environments frequently used by large corporations.

It is the goal of FEAT to fully automatize testing, including preparation of the testing environment. This is why the author has extended the proposed tool with additional features such as automatic installation and execution, file transfers, automatic allocation of testing resources and centralized logging.

The use case analysis presented in Section 6 indicates that automatic testing environments, including FEAT, can be beneficial both for medium-scale and large IT enterprises. Such environments facilitate the testing process, increasing its reproducibility and efficiency, while conserving the financial resources assigned to the project and thus improving the enterprise's competitiveness.

The FEAT environment exhibits all the attributes of a self-contained automatic software testing solution. In terms of novel features it compares favorably to the environments and tools currently available on the market. We can conclude that the concepts underpinning this environment will one day become standard practice in commercial and open-source testing solutions, including those applied in large IT enterprises.

The FEAT environment, in its current form (as presented in this paper), is a complex, feature-rich system. Even so, its modular architecture supports further extensions which can be implemented by developing additional modules. One example of such an extension would be to swap the application container for a Web browser. Under such circumstances, the testing plugin would be installed directly in a specific browser (Internet Explorer, Mozilla Firefox etc.) and verify that a given Web page has been correctly rendered by the browser. In order to perform this test it would take a screenshot of the browser window and then analyze its contents with the use of image recognition algorithms.

Another possible extension might involve a groupwise

resource manager, which would monitor (via a dedicated database) all the resources available to a group of testers, including test hosts, application servers, databases and external services. It could also support a booking system, enabling each tester to reserve access to certain resources for a specified amount of time. The resource manager would then ensure that no single resource is simultaneously assigned to two different testers. It might manage resources on a project-wide or company-wide basis while delivering reports and recommendations (for instance, if a given test host with a specific operating system proves to be a highly contested resource, it might issue a request for another such machine to be provided to the testing teams).

The FEAT testing environment could also provide virtualization capabilities, particularly important for large corporations. To this end it could be extended with a dedicated subsystem responsible for managing virtual system images and instantiating them on test hosts, as required by testers.

In the context of potential wide-scale adoption of FEAT, it might prove beneficial to create a database of tests and toolsets, facilitating rapid development of new tests and easy configuration of working environments. This feature could be supported by a custom module or by an existing software solution such as TPM (Tivoli Provisioning Manager).

Yet another possible extension would be to develop custom testing plugins for popular IDEs such as Eclipse, NetBeans or Visual Studio. Configuring tests would then be supported by a dedicated user-friendly GUI where the tester might select items from a menu and arrange them using a drag&drop mechanism. Tests could also be executed from within the IDE itself, in a way similar to JUnit integration in the Eclipse environment [1].

Clearly, the FEAT environment, as developed by the author and presented in this paper, could be extended in numerous ways, resulting in a highly complex and adaptable framework.

REFERENCES

- [1] L. Pobereźnik and A. Zielińska, "Automatyzacja Procesu Testowania Oprogramowania Komponentowego W Heterogenicznym Środowisku Produkcyjnym," AGH, Kraków, 2008.
- [2] G. J. Myers, C. Sandler, T. Badgett and T. M. Thomas, "Szuka Testowania Oprogramowania," Helion, Gliwice, 2005.
- [3] W. Hetzel, "The Complete Guide to Software Testing," John Wiley & Sons, New York, 1988.
- [4] I. Sommerville, "Inżynieria Oprogramowania," WNT, Warszawa, 2003.
- [5] R. Patton, "Software Testing," 2nd Edition, Sams, Indianapolis, 2005.

- [6] W. E. Perry, "Effective Methods for Software Testing," 3rd Edition, John Wiley & Sons, Indianapolis, 2006.
- [7] J. D. McGregor and D. A. Sykes, "A Practical Guide to Testing Object-Oriented Software," Addison-Wesley Professional, Upper Saddle River, 2001.
- [8] C. Beust and H. Suleiman "Next Generation Java Testing: Test NG and Advanced Concepts," Addison Wesley, San Francisco, 2007.
- [9] A. Hunt and D. Thomas, "JUnit. Pragmatyczne Testy Jednostkowe W Javie," Helion, Gliwice, 2006.
- [10] STAF/STAX documentation, 2009.
- [11] C. Y. Laporte, "An Overview of Software Quality Concepts and Management Issues," Hershey, 2005.
http://profs.logti.etsmtl.ca/claporte/Publications/Publications/Duggan_Chapter_SQA.pdf
- [12] L. Westfall, "Defect Removal Effectiveness," Austin, 1996.
http://www.westfallteam.com/Papers/defect_removal_effectiveness.pdf
- [13] S. H. Kan, "Metrics and Models in Software Quality Engineering," 2nd Edition, Addison-Wesley Professional, Boston, 2003.