

eFitNesse: A Java Testing Framework for Embedded OSGi Programming

Marcus Harringer

MicroDoc Computersysteme GmbH,
Elektrastrasse 6, D-81925 Munich
Marcus.Harringer@microdoc.de

January 4, 2008

Abstract

Existing Java testing methodologies for embedded systems suffer from weak tooling support for automated embedded test execution. As a result, automated test cases are often executed on the development machine instead of the actual target device. *eFitNesse* is a Java testing framework that supports execution of tests on the target hardware platform. Moreover, eFitNesse provides an easy to learn markup-language to define these tests. eFitNesse is based on the FitNesse acceptance testing framework [7]. eFitNesse extends FitNesse by means of remote test execution for OSGi applications on Java ME enabled target platforms. We have introduced eFitNesse within our customer projects, which we also present in this paper. We describe our practical experience with eFitNesse for testing software components of conditional access systems of a leading company.

General Terms: Testing Embedded Systems

Keywords: OSGi, Java Micro Edition, FitNesse, Fit, JUnit, Eclipse

1 Challenges in Embedded Software Development

One of the main challenges when developing embedded systems is the fact that development and execution take place on different platforms. Ideally, application code and corresponding automated tests are executed on the target execution platform. Common practice, however, is that automated tests are mostly performed on the development machine using simulators - tests on the target platform are often performed manually.

Another challenge in software development is customer acceptance. Developer understanding and customer expectation based on written requirement specifications often differ drastically. These misunderstandings can result in problems which are recognized fairly late in the development process, particularly when acceptance tests are performed with the final software version only.

Why not encourage the customer to express his requirements and expectations as acceptance tests in a human readable (while also executable) form ? If this method is adopted, developers are able to implement the software which fulfills the defined requirement tests, and acceptance procedures become a verification process of the test execution. FitNesse closes this gap by providing a wiki with an easy to learn markup language to define acceptance tests and an extensible feature set to execute and verify these tests.

In this paper we present our modified version of FitNesse, called eFitNesse, a bundle that enables OSGi applications to be tested via the FitNesse wiki interface. eFitNesse enables OSGi applications that can either run on the development machine or on an embedded Java ME-enabled target device. eFitNesse can be seamlessly integrated into the already available Eclipse plug-in for FitNesse [11].

2 Traditional Testing Methodologies

In this section we want to compare two projects: one that uses a traditional approach of testing, and one that uses eFitNesse. But before we start, we have to discuss the different levels of testing.

2.1 Levels of Testing

Testing software can be done at several levels of abstraction. We can define the following levels:

- Unit Isolation Testing
- Unit Integration Testing
- Acceptance Testing

Unit Isolation Testing describes the task of testing software at a very fine grained level. Usually, application components are tested in isolation even

if they are designed to run within a framework such as OSGi. The most successful unit testing framework for the Java programming language is doubtless JUnit by Kent Beck and Erich Gamma [4]. A common way to achieve isolation are Mock objects [5] [6].

Unit Integration Tests are used to test interaction of application components. In an OSGi environment, interaction describes the co-operation of different bundles or services. In the embedded space, integration testing also includes testing on the target execution environment. However, due to a lacking tool support for target integration tests, such tests are most often performed on the development environment by using JUnit on a device simulation.

Acceptance Tests are usually performed by customers in order to test if the application meets its requirements. Also developers perform some acceptance tests but these tests may be different to the customer tests. The FitNesse acceptance testing framework [7] can be used to define common (and executable) acceptance tests. Moreover, these acceptance tests can be automated. However, the current implementation of FitNesse is limited to non-embedded applications.

Figure 1 shows a traditional embedded testing scenario for OSGi applications. The figure shows an OSGi application with its corresponding tests that are executed on the development machine and on the target execution platform. Each bundle on the development site contains some unit testing code that is usually executed by JUnit. Unit integration tests will also be performed on the development machine by using JUnit. On the target, unit testing code is not available any more. Testing on the target platform is limited to acceptance tests that are performed manually in most cases.

2.2 Two Projects

We now want to compare two fictive projects: project A will follow a traditional approach of testing and project B will use eFitNesse. These projects will discuss common development problems and highlight the benefits when using eFitNesse. These example projects may be somehow overlooked, but we think this is a good way to bring out the advantages for the customer **and** the developer when using eFitNesse.

2.2.1 Project A

Project A starts with a requirements specification which serves as a manual for development. During development, developers perform unit and unit integration

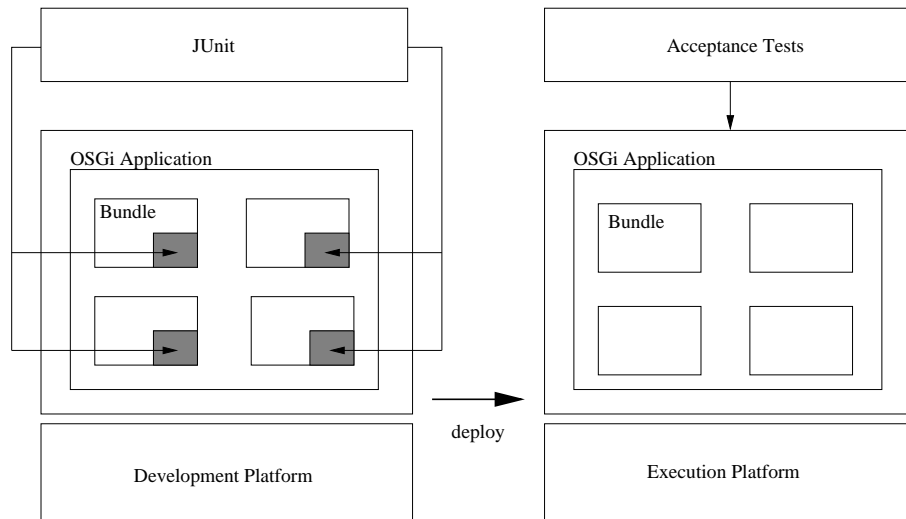


Figure 1: A Traditional Testing Scenario for Embedded OSGi Applications

tests. These tests are mostly executed on the development machine. Developers also perform some kind of acceptance tests on the device. These acceptance tests are performed manually and may be different to the acceptance tests from the customer. After a few months, the code is ready for delivery. On the customer side, the software will go through an acceptance testing cycle to verify its quality and if it meets its requirements. Because there is no tool to automate acceptance tests on the device, the customer has to perform his tests manually, which takes 4 weeks. After 3 weeks the customer notices that the software has some bugs, because the application has a different behaviour on the device than on the development machine. Moreover, the customer needs some change requests to be implemented. Developers have misinterpreted two chapters of the requirements specification. After fixing the bug and implementing the change requests, the customer starts again his 4 week testing cycle.

2.2.2 Project B

Project B also starts with a requirements specification. Additional to the requirements specification, the customer defines executable acceptance tests for eFitNesse. During development, developers perform unit tests on the development machine but also execute unit integration tests on the target device. Moreover, developers will continuously execute the customer's acceptance tests. After a few months, the code is ready for delivery. On the customer side, the software will go through an acceptance testing cycle too. The customer now tests his acceptance tests by using eFitNesse, which takes some hours including setup and testing. There may exist test cases that cannot be automated. However, the bigger part of the acceptance tests have been tested automatically. After

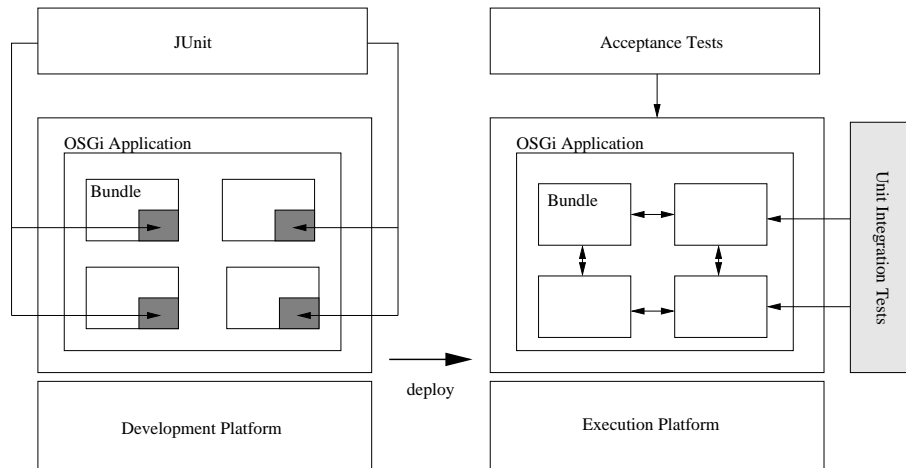


Figure 2: eFitNesse Testing Scenario for Embedded OSGi Applications

2 weeks, the customer has verified the software and is happy about that the software meets its requirements.

3 Improve Embedded Software Quality with eFitNesse

Testing embedded software requires sophisticated tool support for being accepted and successful. In this chapter we describe *eFitNesse*, a Java testing framework for embedded software. *eFitNesse* is directed towards the following goals:

- support for unit integration and acceptance tests
- execute tests on the target execution environment
- remote test control and evaluation
- debugging support for tests

A typical scenario of using *eFitNesse* is shown in Figure 2. Basically, eFitNesse adds support for unit integration tests on the target environment. The framework also covers acceptance tests in order to rely on only one tool for embedded testing. In an OSGi framework, unit integration tests verify bundles and its services.

3.1 eFitNesse Test Infrastructure

Basically, eFitNesse distributes test execution and verification on the development and execution platform. While verification will be performed on the

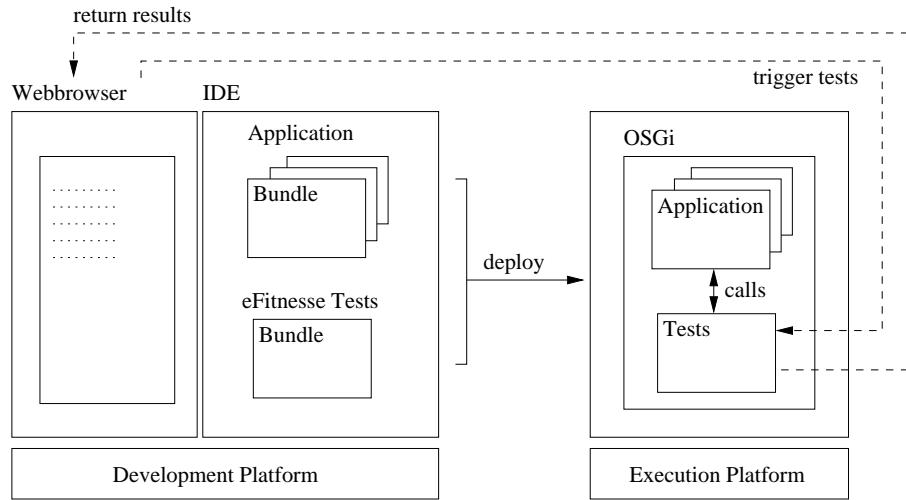


Figure 3: eFitNesse Architecture

development machine, test execution actually happens on the target execution platform. To be more precise, testing with eFitNesse comprises four activities:

1. write the code
2. write the test
3. deploy code and test
4. test execution and verification

Figure 3 shows the overall architecture. On the development site, you need an integrated development environment and a web browser. The first two activities cover writing the application code and the test code. Both parts will be OSGi bundles. Which part you write first, depends on the process model you follow. For example, the TDD approach decides to write the test first. The next step is to deploy both parts: the application code and the test code. In order to execute the tests, you have to start the OSGi framework with all application and test bundles on the target device first. Triggering a test can now be done via a web browser. All tests will be executed on the target device and test-results will be returned to the web browser when all tests are finished.

In order to get a feeling about the web interface for testing we refer to Figure 4. The figure shows a simple test case that tests if all required bundles are in state *ACTIVE*. The bundles are actually running on a remote embedded device.

In the following sections we will take a detailed look at the implementation of eFitNesse.

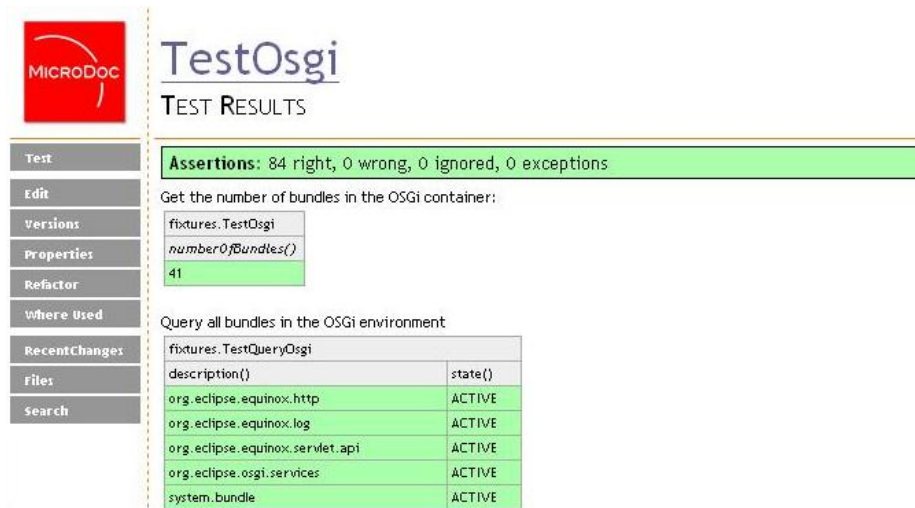


Figure 4: FitNesse Web Interface

3.2 FitNesse

The implementation of eFitNesse is based on the FitNesse acceptance testing framework [7]. FitNesse is a collaboration and testing tool for software development. The main characteristic of FitNesse is a wiki for creating test data and triggering tests. FitNesse also includes a web server that hosts the wiki.

Technically, FitNesse is build together by two parts:

- Web server and Wiki
- Framework for Integrated Testing (FIT) [9]

While FitNesse is used to describe and trigger test cases, FIT is the engine that actually executes the tests. FIT uses a tabular representation of test data in form of HTML, Word, or Excel. The framework executes the tests and generates a result document. This representation of test data can be seen as an executable requirements specification. FitNesse extends this tabular representation for being used within a wiki. The connection between the wiki and the FIT framework is established via so called *Fixtures*. A *Fixture* is a piece of Java code that can be called by the wiki. Figure 5 shows this architecture.

FitNesse is a great framework but has some limitations in the field of embedded and OSGi computing. Embedded Java virtual machines (VM) and its class libraries that implement the Java Micro Edition specification [8] differ

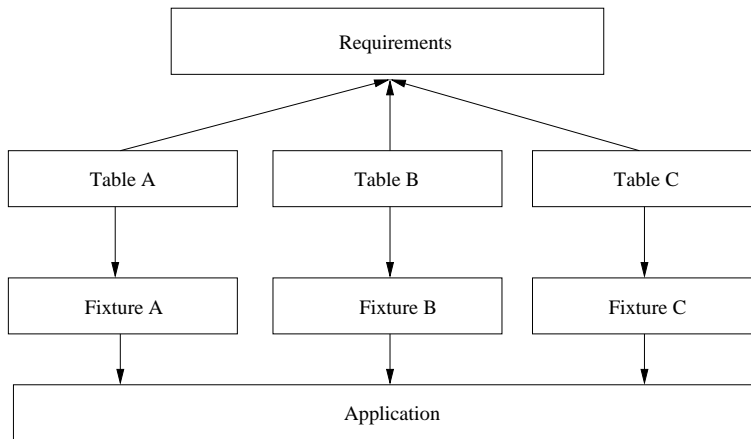


Figure 5: Architecture of FitNesse

from standard or enterprise Java VMs. The current implementation of FitNesse requires at least Java 5 Standard Edition, which makes it unsuitable for embedded Java VMs. Moreover, OSGi applications can hardly be tested with the framework.

3.3 eFitNesse Implementation

In order to enable FitNesse for embedded OSGi programming, eFitNesse implements two major modifications of the original source code:

1. The FitNesse code has been ported to Java Micro Edition CDC 1.1
2. FitNesse has been split up into two parts: the wiki and web server part and the actual FIT test runner.

This distributed design has been greatly inspired by patang [10]. Patang is a framework for testing server-side code. Figure 6 shows this separation.

While wiki and web server remain on the development machine, the FIT test runner is located on the target machine. Since FIT runs within a container like OSGi, access via the network, preferably over HTTP, is the only way to control the test runner. For this purpose, eFitNesse implements a special servlet, called the FitServlet, to manage this access. The OSGi HTTP service is used to publish the servlet.

This simple design has some useful implications. First, as we separated the wiki from the actual test runner, the embedded application can be tested *re-*

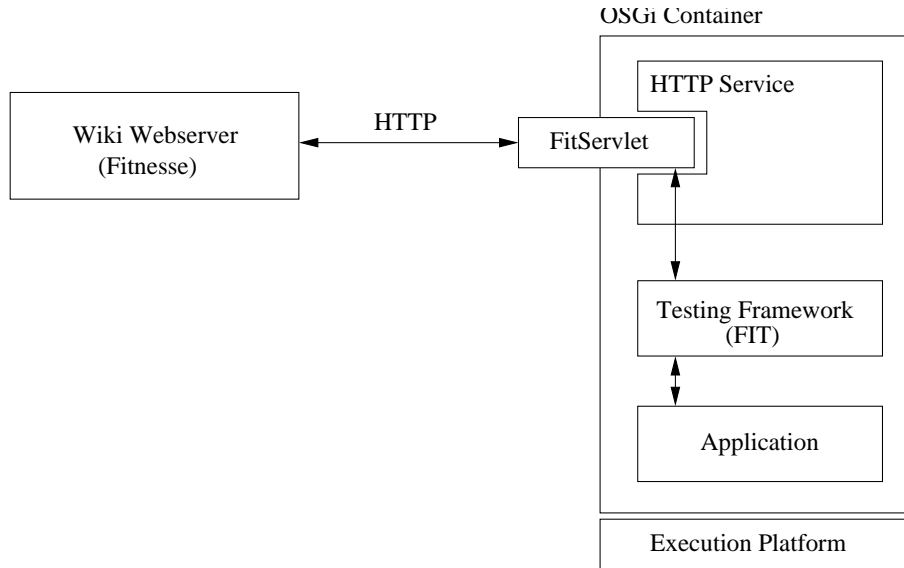


Figure 6: eFitNesse splits FitNesse into two parts: wiki and web server part and the actual FIT test runner.

motely. Moreover, we can start the application including the test runner in *debugging mode*. As a result, remote test cases can be debugged remotely.

3.4 Additional Features

During development we discovered some other useful features that can make development more comfortable.

One is the support for *system analysis*. In FitNesse, a test case is either true (green colored) or false (red colored). We have now added a special test case (that is neither true nor false), which can be used to analyse the current system. For example, the bundle runtime states or VM parameters could be queried by this special test case.

The second additional feature is the support for standard JUnit test cases. FitNesse uses wiki test tables in order to describe expected test data. However, there may exist test cases where expected data has to be calculated during runtime. Of course, this can be done by FitNesse as well, however doing so is not straight forward. JUnit offers all methods to quickly implement such a test case. Moreover, by supporting standard JUnit test cases, tests become more portable between different testing frameworks.

Both features have been implemented by special FitNesse Fixture classes.

3.5 Eclipse Integration

Using the wiki and web server on the development side requires some tasks like starting the web server and setting up classpaths. Sometimes development teams waste a lot of time in managing classpaths and setting up environments. In order to simplify starting wiki and web server and setting up classpaths, Band XI [11] provides an Eclipse FitNesse plugin that takes care of these error prone tasks. The plugin is available on their website.

4 Testing Conditional Access Systems with eFitNesse: A Success Story

This section describes a success story of using eFitNesse for testing SkiData's conditional access systems. SkiData is an international solution provider specialising in professional access management and ticketing [13]. SkiData offers a wide range of solutions and services for controlled access in the field of mountain destinations, shopping centers, airports, fair & exhibition centers, car park operators, arenas, amusement parks, and much more.

In 2006 MicroDoc GmbH [15] started a project with SkiData's firmware department. MicroDoc delivers first rate, high quality products, programming services, training and consulting for customers in the embedded and enterprise space. MicroDoc's Java expertise is used by companies from various industries like financial, airline, automation, automotive, logistics and IT. MicroDoc strongly believes in Open Source Software and is an active Eclipse and LBCN member.

The goal of the project was to develop a framework for writing control software for SkiData's custom hardware devices. The framework is based on OSGi and Java Micro Edition. The design of the framework was aimed for a clear separation of business logic and framework logic. This design preserves the distribution of domain knowledge competences. While SkiData has expert knowledge within the domain of access systems, MicroDoc's experience comprises OSGi and Java.

Access systems are long running, time-critical devices that have to deal with thousands of accesses each day. Software quality is a key success factor within the domain of SkiData. Moreover, SkiData has to deal with fast changing



Figure 7: SkiData's Freemotion Basic Gate

market requirements and keen competition. These are some of the facts that motivated us to follow a test-driven approach in developing the framework and its components. While performing detailed unit tests with JUnit, we introduced eFitNesse within the project. Not only MicroDoc has used eFitNesse for quality assurance, but also SkiData decided to integrate eFitNesse within their quality assurance department.

4.1 Freemotion Gate

To get an impression about SkiData devices, Figure 9 shows the Freemotion Basic Gate which has been used as a reference platform during development. The product family of Freemotion gates is used for access in mountain resorts. Freemotion is designed to deliver ultimate comfort and protection against fraud and ticket misuse [14].

The basic version of the Freemotion gate's devices include RFID and barcode scanners, sound, lights, the turnstile, light sensors, and a graphical display. Each of these devices will be controlled by a software module that runs within the framework.

4.2 Test Cases

The eFitNesse framework can be used for unit integration and acceptance testing. eFitNesse has been used to test MicroDoc's framework as well as SkiData's module implementations. In this section, we want to focus on implementations

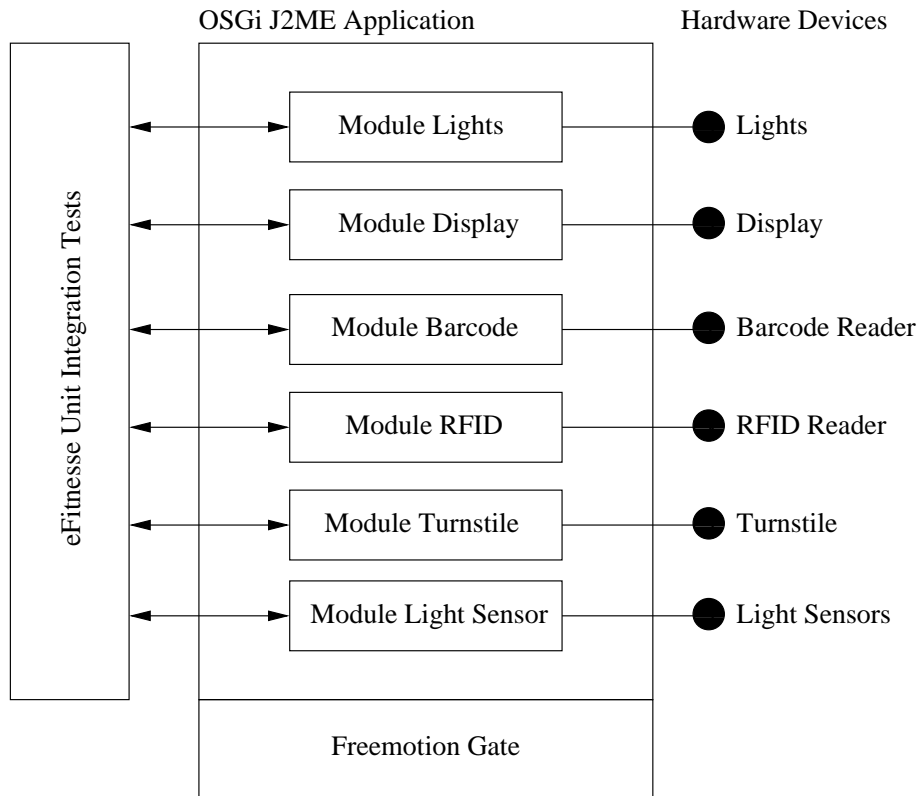


Figure 8: Unit Integration Tests for the Freemotion Gate.

by SkiData.

4.2.1 Unit Integration Tests

Unit integration tests verify bundles and services. Each hardware device of the Freemotion gate is controlled by one OSGi bundle. Figure 9 shows this scenario. Tests and bundles are actually executed on the target embedded device.

For example, a typical test case for the light module is to verify if all lights work correctly. Another test case is to verify if the display device draws GIF images correctly.

4.2.2 Acceptance Tests

Acceptance tests treat software as a black box. No detailed knowledge about internal modules is assumed. Ideally, acceptance tests are part of the requirements specification that are used to verify if the system meets its standards. When using acceptance tests, short iteration loops and early customer involvement are important. FitNesse provides a sophisticated framework that allows to

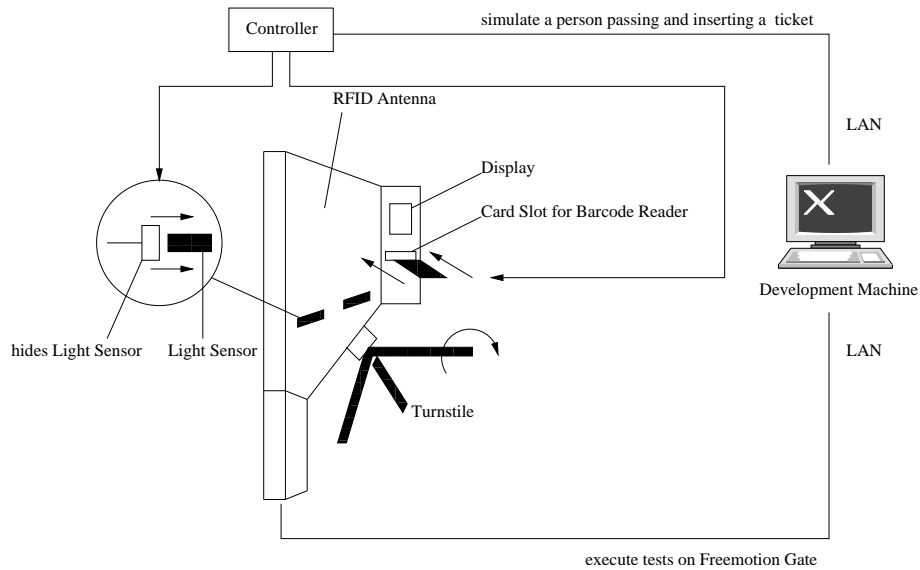


Figure 9: Acceptance Tests for the Freemotion Gate.

combine tests and descriptive information to be collected in one place. We have used eFitNesse, the embedded counterpart of FitNesse, in order to simplify the acceptance process of our delivered component framework. Moreover, SkiData has used eFitNesse internally for inter-departement delivery.

Testing embedded systems that require user input is a hard task to achieve. Especially, testing access systems always requires a volunteer to pass. To automate the process of people passing, SkiData invented a sophisticated approach. A complete chain of actions when passing an access station can be summarized by the following tasks:

- person arrives: this can be detected by light sensors
- person inserts ticket: the RFID or the barcode reader will be invoked
- if the ticket is valid, the turnstile opens
- person passes the turnstile: this can also be detected by light sensors.

SkiData has now installed hardware devices that can control the light sensors (hide/show) and insert tickets into the card slot. This makes it possible to simulate a real person passing. Figure 9 shows this scenario.

In order to implement and control this test case, SkiData used eFitNesse. Start and evaluation of the test case happen on the development machine. Moreover, the development machine also controls the simulation hardware on the device. This is necessary to correlate expected and actual results and actions.

5 Conclusions and Future Work

In this paper, we presented a framework for testing embedded Java OSGi applications. The framework is based on the acceptance testing framework FitNesse and implements the following features:

- Java Micro Edition (J2ME) compliant
- OSGi compliant
- Remote Debugging support
- Remoting support
- System analysis support
- JUnit test cases support

The value of tests strongly depends on the effort they require. If test setup and execution is very costly and has to be done manually, testing slows down the development process considerably. Ideally, tests should be integrated into a continuous build process. Currently, eFitNesse executes automated tests but has to be triggered manually. Future work will focus on integration of eFitNesse within an continuous build process like Cruise Control [12].

References

- [1] Java Programming Language: <http://java.sun.com>
- [2] Open Services Gateway Initiative: <http://www.osgi.org>
- [3] Eclipse Integrated Development Environment: <http://www.eclipse.org>
- [4] JUnit testing framework: <http://www.junit.org>
- [5] EasyMock: <http://www.easymock.org>
- [6] JMock: <http://www.jmock.org>
- [7] FitNesse acceptance testing framework: <http://www.fitnesse.org>
- [8] Java Micro Edition: <http://java.sun.com/javame/>
- [9] FIT testing framework: <http://fit.c2.com/>
- [10] Patang Framework <http://patang.sf.net/>
- [11] FitNesse Eclipse Plugin: <http://bandxi.com/>
- [12] Cruise Control: <http://cruisecontrol.sourceforge.net/>
- [13] SkiData AG: <http://www.skidata.com>
- [14] SkiData Freemotion Gate: <http://www.skidata.com/Freemotion-Gate.69.0.html?%L=1>
- [15] MicroDoc Computersysteme GmbH: <http://www.microdoc.com>