Test-Driven Migration Towards a Hardware-Abstracted Platform

Wolfgang Raschke[1], Massimiliano Zilli[1], Johannes Loinig[2], Reinhold Weiss[1], Christian Steger[1] and Christian Kreiner[1]

[1]*Institute for Technical Informatics, Graz University of Technology, Inffeldgasse 16/I, Graz, Austria*

[2]*Business Unit Identification, NXP Semiconductors Austria GmbH, Gratkorn, Austria*

Keywords: Software Reusability, Test-Driven Development.

Abstract: Platform-based development is one of the most successful paradigms in software engineering. In embedded systems, the reuse of software on several processor families is often abandoned due to the multitude of compilers, processor architectures and instruction sets. In practice, we experienced that a lack of hardware abstraction leads to non-reusable test cases. We will demonstrate a re-engineering process that follows test-driven development practices which fits perfectly for migration activities. Moreover, we will introduce a process that provides trust for the test cases on a new hardware.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Engineering in the field of Smart Card development faces several challenges, such as the demand for a high level of security (Mostowski and Poll, 2008), low memory footprint, power consumption and runtime performance (Rankl and Effing, 2003). All these requirements are interrelated and in fact, the multitude of dependencies hinders Smart Card suppliers and issuers from deploying a great deal of diversified customizable products. Rather, there are only a few standard products available on the market. These do not meet the needs of today's customers who are increasingly demanding tailor-made products.

Principles of platform-based development and Software Product Line Engineering (SPLE) (Pohl et al., 2005)(Clements and Northrop, 2002) are a successful paradigm in software engineering. SPLE aims at systematic reuse where possible and provides a conceptual framework for the diversification of products. In the *domain engineering* process the purpose is not to develop single products but to develop a base of related systems in respect to the product family. Dedicated rules of composition are defined. In the *application engineering* process, the engineers assemble a product out of the product familythat corresponds to these rules of composition.

Products are based on several processor families (PF) which have different implications regarding compilers, byte endianness and architecture. Test cases are not platform-independent *per se*, even if they are written in Junit[1], a Java based unit test framework.

The migration to a product line has to be accomplished during operation: a Smart Card system is under construction on PF A. The plan to transition towards a hardware-abstracted software is to track the development of the Smart Card system on a second PF B as a proof of concept. The way of working is as follows: first, take existing test cases and abstract them from PF A. Second, build confidence for platform-abstracted test cases. Third, use test cases on PF B in order to port the software in a test-driven development process.

2 REQUIREMENTS

2.1 Requirements in the Industrial Context

It is intended to port as many software components as possible to several PF. Initially, it was deemed appropriate to elicit a set of coding guidelines in order to keep the source code platform-abstracted. Once a pilot project had been conducted to validate our approach, it became apparent that the tests, in particular, create a bottleneck. Porting the source code has worked without much refactoring of the code. Unfortunately, it turned out that most of the

[1]http://junit.org/

Raschke W., Zilli M., Loinig J., Weiss R., Steger C. and Kreiner C..
Test-Driven Migration Towards a Hardware-Abstracted Platform.
DOI: 10.5220/0005216302610267
In *Proceedings of the 5th International Conference on Pervasive and Embedded Computing and Communication Systems* (PECCS-2015), pages 261-267
ISBN: 978-989-758-084-0
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

test cases needed to be refactored manually. This was not acceptable for the following reasons: first, the Software Product Line is intended to run on several PF. Industrial embedded systems are usually tested by several thousands of test cases. So, porting test cases manually is not economically feasible in the long run.
Second, manual porting activities are a source of possible defects. The test cases inhibit a high amount of memory dumps. Manually processing this data is error-prone to some degree. Thus, the test cases introduce an additional risk for each migration to a new hardware.
Third, in order to achieve platform-abstraction, we decided to allow *no code change* for different PF, *neither* in source code *nor* in test code. A code change for a specific PF would significantly decrease the reusability of the code.

2.2 Requirements Due to Variability Drivers

The Java Card (Oracle, 2011b)(Oracle, 2011a) operating system under analysis is basically built up as depicted in Figure 1. At the bottom of the system, the variability stems from the utilization of the different PF A or B. Both of them introduce several facets of variability: first, each PF enforces the utilization of a separate tool chain which usually includes compiler, linker and simulator. This facet of variability propagates to the hardware abstraction layer. A considerable portion of it has to be written in assembler and code which is not ANSI-C compliant.
Second, each PF may have a different byte endianness and pointer size. For instance, a 32 bit pointer on processor A corresponds to a 16 bit pointer on processor B. We experienced these factors as the major impediments for test case reuse over several PF. These drivers affect the process in all layers except for the Java Card application layer.

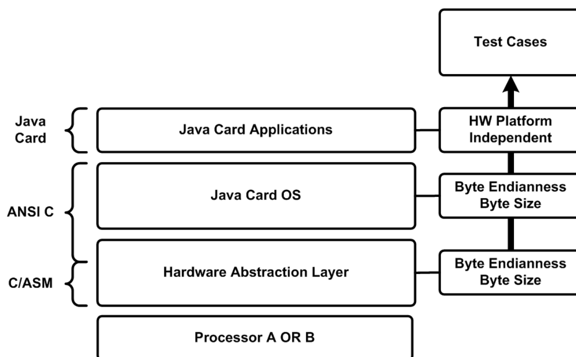

Figure 1: Variability Drivers for Test Cases.

2.3 Platform Lifecycle: Product 1 - Reengineering - Platform, Product 2

The major constraint of the refactoring process is that the industrial product development for PF A may not be disturbed. The continuing industrial development is shown as phase 1 in Figure 2. The pilot study is intended to demonstrate and prove the feasibility of a platform-abstracted SPL. If the study is a success we will transform the development to a platform. In order to fullfill the dedicated requirements, the process is structured as follows:
Phase 1 is the ongoing industrial product development process which may not be disturbed.
Phase 2 is intended to refactor the tests to be hardware-abstracted.
Phase 3 is a dedicated phase where confidence of the tests on all PF must be demonstrated.
Phase 4 uses the abstracted tests to refactor the code base. Daily test runs keep the feedback cycle accurate for early detection of defects. This will help to mitigate the influence of the hardware abstraction activities on the industrial product development process.

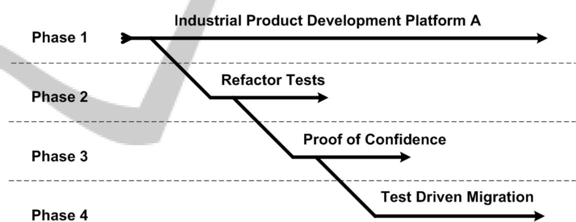

Figure 2: Test Refactoring and Proof of Confidence are the Precondition for Test-Driven Migration.

3 TECHNICAL BACKGROUND

The existing test infrastructure is basically split up into two parts: *off-card* and *on-card*. Off-card, Junit is used as a test framework.
Junit launches a test case which then has the responsibility to serialize the test data within a *transmit buffer*. This buffer is then transmitted via packets to the on-card side.

On-card, the In System Test Framework (ISTF) stores the test data within a *receive buffer*. The dispatcher then analyzes the address which denotes the intended caller stub. Then, the *caller stub* is launched. It has access to the receive buffer. The test data has to be de-serialized, which means that it is retrieved from the buffer and stored in variables. These variables are used as parameters for calling functions of the *Module Under Test* (MUT). The response is collected by the stub module and propagated to the ISTF and then

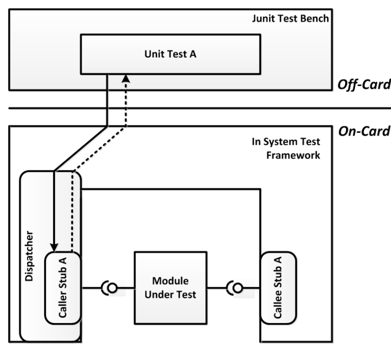

Figure 3: The Test System is Contains Two Parts: The *Junit Test Bench* and the *In System Test Framework (ISTF)*.

to the Junit test bench. At that point, the test response is evaluated and a corresponding test report is generated. The *callee stub* is not connected directly to the Junit test bench. It substitutes the other modules the MUT usually calls but which are not present in unit testing. This stub has to provide the MUT with the appropriate responses.

4 METHOD

The method of handling hardware dependencies within tests is similar to that of Model-Based Testing (MBT) (Pretschner and Philipps, 2005). The latter methodology aims to abstract the system to a model in order to generate abstract tests and test specifications. Nevertheless, the goal of MBT is different to Test-Driven Development (TDD). In TDD, tests basically represent pure functional requirements.
A reasonable synthesis between MBT and TDD is to raise the level of *functional* test cases to an abstraction where the following conditions are fullfilled: first, developers are easily able to formulate the test cases without the need of a formal model. Second, the tests need to abstract all hardware specifics that impede portability.

4.1 Abstraction Method

Abstract test cases need to abstract two issues: first, the endianness of the target PF has to be abstracted. This is accomplished by defining big endianness as a rule for implementing abstract test cases. Fortunately, the Java byte endianness is big endian, by definition. Second, the data (usually a memory dump) has to be abstracted in order to be reusable for several PF. Complementing the data with data types is a reasonable abstraction of a binary representation and meets the previously stated requirements.
The basic methodology of test case abstraction is shown in Figure 4. The methodology constitutes of 4 states and 3 transitions.

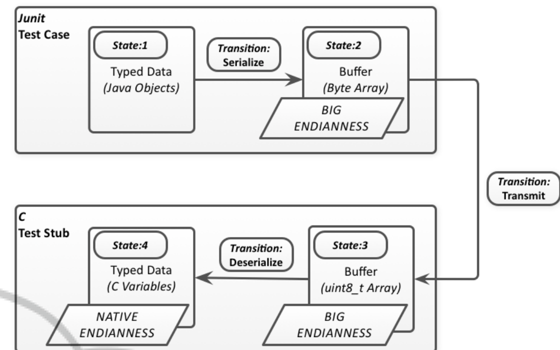

Figure 4: The Method of Handling Tests on Different Processors Contains 4 *States* and 3 *Transitions*.

The process starts at *State 1*, where the data is bound to types. These types are Java classes. When they are instantiated by objects they are initialized with the test data. In order to send these objects to the Java Card, they have to be serialized which is shown in Figure 4 as a transition. Afterwards, the data is stored without any type information in a Java Byte Array in *State 2*. The serialized data is organized in the buffer with *big endianness*. In the next transition the data is transmitted from the Junit framework to the Java Card. Thereafter, in *State 3* the data is available in a buffer. Here, the byte order is still *big endian*, regardless of the processor. In the following transition the data has to be de-serialized. Finally, in *State 4* the endianness is resolved and the data is stored in variables. These variables are usually parameters for calling the MUT. The test case is now determined and executable.

4.2 Traditional TDD Process

The traditional *red-green-refactor process* (Beck, 2002) for TDD is shown in Figure 5. Basically, a test is first written which covers a certain functional feature. In the *red* step, this test is executed without the implementation of this feature. This then results in the test failing, which shall be demonstrated in this step. If the test does not fail, it indicates that there is something wrong. Then follows the *green* step where code is written in order to allow the test to pass. If this is achieved, the next phase is *refactoring*. Here the code is rewritten until it also meets the defined non-functional requirements, such as maintainability, reliability and the like.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

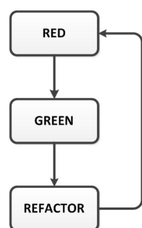

Figure 5: Red-Green-Refactor Process of Test-Driven Development (Beck, 2002).

4.3 Inverted U Process

Due to the high number of test cases, it is not economically feasible to review each and every test case. Thus, we developed a dedicated process (Figure 6) for providing confidence in the abstracted test cases.

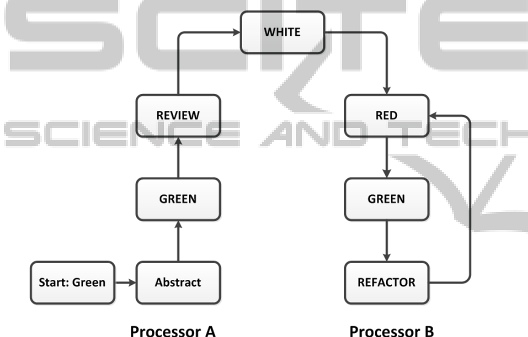

Figure 6: Inverted U Process to Provide Confidence for Abstract Test Cases on PF A *and* B.

The inverted U process is used to port the tests and provide confidence in them. Because this process embraces two processors, it has a right and a left branch. The process has a dedicated starting point on the left hand side. A green (passed) test case on processor A is the starting condition. If it has passed once, it can be abstracted, as described previously. After the abstraction, the test case needs to pass again to show that no flaws have been introduced. There is now evidence that the test case is correct on processor A. Still there needs to be assurance that it also works on processor B. Thus, the next step is a *review*, where the abstracted test case is investigated for potential pitfalls, such as the utilization of pointers.
In the *white* step the test case passes on processor A and there is some confidence for it on processor B. Nevertheless, the level of confidence here is not high enough, so the process continues on the right hand side.
Here, the test first enters the *red* phase. Until the test passes, there remains uncertainty about its correctness. So, in the *green* step, the code is ported to run on processor B. If the test did not go green before, consideration is given to investigating and rewriting the test. After the test has turned green for the first time, there is confidence that the test makes sense on processor B. Nevertheless, it should always be checked that the test is compatible with other hardware platforms.
From now on, on the right branch the process is in the *traditional* TDD refactoring loop, as described previously.

5 IMPLEMENTATION

In the following, we will explain the implementation of our method with a sample test case at hand. This test case sets the Java Card program counter *pc* to a certain value. For the explanantion, we will follow the abstraction reference process which we defined in Section 4.1 and in Figure 4.

5.1 Implementation of the Test Case and the Test Stub

The previously mentioned reference process starts in the *Junit* test case implementation which is shown in Figure 7 and continues in the *C* test stub (see Figure 8). The states and transitions of the reference process are indicated within the comments. In the following, we will describe the implementation of the states and transitions.

State 1: In State 1 (see line 3-8 in Figure 7) first, a new object of the type *pointer* is created. In line 5 the pointer address is set to this object. In the next step, the buffer is allocated with the correct length (see line 7).

Transition - Serialize: In line 10-11 the object is serialized and stored within the buffer. Each class which can be serialized has to implement its own *serialize* function.

State 2: After the serialization, the objects data is now stored within the buffer in big endianness. This state is indicated by comments in line 13-15.

Transition - Transmit: In line 18 the *send* function transmits the buffer to the Java Card.

State 3: In State 3 (see line 3-5 in Figure 8), the In System Test Framework has already stored the transmitted data in a buffer, which is located on-card.

```
01 public void setPC(address addr)
02 {
03 //State 1
04 pointer pc = new pointer();
05 pc.setAddress(addr);
07 ByteBuffer txBuffer = ByteBuffer
08 .allocate(pc.getLength());
09
10 //Transition: Serialize
11 txBuffer.put(pc.serialize());
12
13 //State 2
14 //serialized data are now stored
15 //off-card in the buffer
16
17 //Transition: Transmit
18 send(txBuffer.array());
19 ...
20 }
```

Figure 7: *Junit* Test Case: It Sets the Java Card Program Counter (pc) to a Value.

```
00 static ErrorCode StubJvmSetPc(void)
01 {
02 ...
03 //State 3
04 //serialized data are now stored
05 //on-card in the buffer
06
07 //Transition: De-Serialize
08 pc = GET_INT_PTR
09
10 //State 4
11 //Data are now stored
12 // in the variable pc
13
14 //increment buffer index
15 // by pointer length
16 idx += PTR_INT_LEN;
17
18 // call module function
19 returnCode = setJvmPC(pc);
20 ...
21 }
```

Figure 8: *C* Test Stub: It Sets the Java Card Program Counter (pc) to a Value.

Transition - De-Serialize: In line 8 a de-serializion macro is used to retrieve the pointer and store it in a variable. The de-serialization method will be explained in more detail in Section 5.2.

State 4: In State 4, the buffer index *idx* is incremented by the pointer size (line 16). Finally the function of the MUT is called with the currently calculated pointer. The MUT returns a value which can be used to evaluate the test response.

5.2 Implementation of the De-serialization Macro

The de-serialization macro is called in line 8 in Figure 8. For each PF, there is a separate implementation of it. Figure 9 is a variant for an architecture with 3 byte integer pointers and big endianness. First, in line 0, the constant PRT_INT_LEN is set to 3 according to the pointer size. The pointer is reconstructed by shifting and concatenating the bytes with an *or* in the appropriate order. It can be seen that the first byte is not shifted. So, the first byte is the *smallest* which is the case for little endian byte order. The following bytes are then shifted by increments of 8. Finally, the resulting value has to be casted to the relevant pointer.

```
00 #define  PTR_INT_LEN 3
01
02 //Get Little Endian Pointer
03 #define GET_INT_PTR (uint8_t *)(
04 (uint32_t)(rxBuffer[0 + idx])  |
05 (uint32_t)(rxBuffer[1 + idx]) << 8 |
06 (uint32_t)(rxBuffer[2 + idx]) << 16);
```

Figure 9: De-Serialization Macro for Resolving 3 Byte Pointers From the Buffer.

In Figure 10 the same principle is applied for a processor with 4 byte integer pointers and big endianness. The principle is the same but in contrast, in line 0 the PRT_INT_LEN is set to 4. Regarding the reconstruction of the pointer, the lowest byte is shifted by 24 bits. Thus, the first byte is the *highest* which is true for big endian byte order. The next bytes are shifted by increments of -8.

```
00 #define  PTR_INT_LEN 4
01
02 //Get Big Endian Pointer
03 #define GET_INT_PTR (uint8_t *)(
04 (uint32_t)(rxBuffer[0 + idx]) << 24 |
05 (uint32_t)(rxBuffer[1 + idx]) << 16 |
06 (uint32_t)(rxBuffer[2 + idx]) << 8 |
07 (uint32_t)(rxBuffer[3 + idx]));
```

Figure 10: De-Serialization Macro for Resolving 4 Byte Pointers From the Buffer.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

6 RESULTS

6.1 Identification of Variability Within Junit Tests

When we started the porting of the software, we were not aware that Junit tests incorporate that high a degree of variability. Most of the tests have not been written with portability in mind which, in the beginning, made our approach difficult.

6.2 Initiative Came from a Programmer

The initiative for changing the legacy testing system came from a programmer who was involved in the porting of the software. For those who were involved in these activities it was initially almost impossible to keep up with the porting of the test cases. The guidelines for writing test cases which we developed helped a lot.

6.3 There is a Need for Training

We experienced that the appropriate coding of test cases requires the training of programmers. Otherwise they are not aware of the problems and do not create platform-independent tests. We created a set of guidelines and provided training to the programmers. The resulting awareness mitigated many problems during the porting.

6.4 Low Overhead

The overhead of the methodology is low, if it is adjusted at the start of a project. The overhead is then limited to training programmers and keeping to the coding guidelines. If the methodology is introduced at a later stage, the additional work is higher because all tests have to be refactored and it usually takes some time for people to become familiar with it.

6.5 High Benefit

If the embedded software is used on more than one hardware platform, the benefit of the methodology is high. In embedded systems, there are usually several hundred or thousand tests that could be reused. With the number of supported PF the benefit also increases with comparably little overhead.

7 RELATED WORK

Software Product Line Engineering (Pohl et al., 2005)(Clements and Northrop, 2002) aims at systematically reusing software. For this purpose, a base of reusable software components, the *product family* is maintained. Rules of aggregation are explicitly defined for code and tests.

Platform-based methodologies for embedded systems are given in (Sangiovanni-Vincentelli and Martin, 2001). The authors discuss the specific reuqirements of a reuse strategy for embedded systems, including hardware and software. They provide a vision and a conceptual framework for platform-based software which starts with a high-level system description. This description is then refined incrementally.

TDD is part of agile development practices (Cockburn, 2006) which are lightweight processes that make use of feedback methodologies. Greene (Greene, 2004) gives insight to the application and requirements of agile practices on embedded systems development. He discusses several facets of XP and Scrum and their adoption of embedded systems design. He concludes that there is a positive effect of most of the applied practices.

Grenning (Grenning, 2007) describes special challenges of TDD in embedded systems. These challenges are addressed with the *embedded TDD* cycle that embraces several stages of testing which are applied with different frequency. The five stages range from testing on a workstation to manual testing in the target. This approach has the benefit that the most simple testing approaches are applied most frequently. Finally, Greening discusses issues regarding compiler compatibility and hardware dependencies and possible solutions regarding TDD.

Karlesky et al. (Karlesky et al., 2007) present the so-called *Model-Conductor-Hardware* design pattern in order to facilitate testing in hardware-dependent software. This design pattern is adopted from the Model-View-Presenter (MVP) and the Model-View-Controller (MVC) patterns. Both patterns address issues regarding the development and interaction with Graphical User Interfaces (GUI). A GUI has similar challenges for programming and testing as hardware (event-handling, asynchronous communication and accessibility). Furthermore, a four-tier testing strategy is presented which deals with issues in automation, hardware and communication testing.

In (Bohnet and Meszaros, 2005) a case study of porting software using TDD is presented. The legacy application is a business software which was ported to adapt to a new database system. In order to port the system, the test cases served as a template and

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

specification for the required functionality. It turned out that the application of TDD resulted in less code on the target platform because unused code was not ported. Moreover, the authors showed that defect test cases are a severe problem because the process suggests searching for problems within the code and not within the tests.

8 CONCLUSION AND FUTURE WORK

We challenged the problem of porting a legacy system to a new hardware platform. In order to do this economically, a high number of tests had to be rewritten to be platform-independent. We experienced that this is possible with a relatively low overhead. A major problem of the proposed test-driven migration process is that the correctness of the tests on the new hardware needs to be shown. A dedicated process helps to establish the necesary confidence. The challenges can only be addressed successfully, if the technical realization of the porting parallels the proposed process. Additional training and guidelines for programmers are neccessary.

For further work, it would be interesting to add type-specific information to the serialized data which can be reused during the de-serialization. Going further, a domain-specific language for testing would allow the generation of both, the Junit tests and the C test stubs from one description of a test case.

ACKNOWLEDGEMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

REFERENCES

Beck, K. (2002). *Test-driven Development*. Addison-Wesley Professional.

Bohnet, R. and Meszaros, G. (2005). Test-Driven Porting. In *AGILE*, pages 259–266. IEEE Computer Society.

Clements, P. C. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

Cockburn, A. (2006). *Agile Software Development*. Pearson Education.

Greene, B. (2004). Agile methods applied to embedded firmware development. In *Agile Development Conference*, pages 71–77.

Grenning, J. (2007). Applying test driven development to embedded software. *Instrumentation & Measurement Magazine, IEEE*, 10(6):20–25.

Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. (2007). Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA*.

Mostowski, W. and Poll, E. (2008). Malicious Code on Java Card Smartcards: Attacks and Countermeasures. pages 1–16. Springer.

Oracle (2011a). *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition.

Oracle (2011b). *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

Pretschner, A. and Philipps, J. (2005). 10 Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291. Springer.

Rankl, W. and Effing, W. (2003). *Smart Card Handbook*. John Wiley & Sons, Inc., 3 edition.

Sangiovanni-Vincentelli, A. and Martin, G. (2001). Platform-based design and software design methodology for embedded systems. *Design Test of Computers, IEEE*, 18(6):23–33.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS