Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 63–70

Observations on incrementality principle within the test preparation process

Csaba Szabó
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: `Csaba.Szabo@tuke.sk`

Ladislav Samuelis
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: `Ladislav.Samuelis@tuke.sk`

Abstract. This paper deals with the abilities of consolidating the system design and test planning phases of the software life cycle (SWLC). We present our observations on the presence of evolution-like features inside test plans during design and development of the application. We focus on the role of incrementality principle within the test preparation process. The presence of this principle is not evoked by the planning process itself, but is inherent in the development stream that incrementally interferes the software design. We analyze this feature and give explanation. Finally, we discuss the impacts of the incrementality principle on management and improvement of software processes.

1 Motivation

Nowadays, a lot of software development methods is available for use in SWLC implementations to achieve the best fit to users' requirements [1]. The main task of each one of them is to deliver faultless software.

When software reliability, safety, stability and overall applicability are questioned, software engineers use testing methods to satisfy and prove these features.

AMS 2000 subject classifications: 68N19
CR Categories and Descriptors: D.2.5. [Testing and Debugging]: Subtopic - Tracing.
Key words and phrases: incrementality principle, test preparation, software evolution, test evolution

Test method selection and execution (i.e. the realization of proofs) strictly depend on the characteristics of the used development method and on the overall SWLC management strategy. Even if no strategy is present, there is no strictly meant freedom in testing process selection. No classical testing is needed in case of mathematically proven algorithm implementation [2] if there is used an also proven implementation technique. (S-type software according to Lehman's laws [3].)

The connection between development and testing does not end by strategy selection and SWLC principles definition. The final verification method of the software depends on the software itself, on its architecture and aim. The most of SWLC models propose test preparation as a separate process that might be executed parallel to or immediately after the design/implementation processes.

On other hand, any interconnection between two models indicates dependencies between them and this is the basis of change propagation across these models [4, 5, 6]. This propagation preserves the actual state of each one according to the development stage and requirements. The mentioned interconnection might be tight or loose depending on the SWLC, but will be present, due to the main principles of software development and testing.

In our paper, we present three observations that highlight change propagation across the whole SWLC, especially on test preparation. Section 2 deals with so-called old school SWLCs based on top-down or bottom-up development strategies. Section 3 is denoted to agile processes of SWLC. In Section 4, we show an example development method called cowboy coding, where might be neither SWLC nor development practices considered. Section 5 concludes our observations and points out more issues to take into consideration in the future.

2 Observation one: old school techniques

Old school methodologies include the waterfall, spiral, staged, iterative, incremental strategies [1]. We classify classical model-driven development (MDD) [7], component based development and the Unified Process (UP) [8] as belonging to this group as well.

Within these methodologies, test preparation is a separate process running parallel divided into smaller activities or sequentially after implementation of software is done.

Having a parallel process implementation, development and test preparation processes operate the same requirements, functions and interfaces. These techniques principle is to prepare test right after a component interface and/or implementation is finished. Any incremental change of those components having tests already prepared is propagated across these tests as well as across all other affected interfaces/implementations within the same design model. Figure 1 shows a typical example of such a SWLC, where the software reached a specific development level (i.e. can be tested), but there are still parts of it that are not implemented yet.

```
Running make test
Prepending /root/.cpan/build/ExtUtils-MakeMaker-6.46/blib/arch
/root/.cpan/build/ExtUtils-MakeMaker-6.46/blib/lib to PERL5LIB.
PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM"
"-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/00-load..............1/1 # Testing Test::Pod 1.26, Perl 5.008
008, /usr/local/bin/perl
t/00-load..............ok
t/all_pod_files........ok
t/cut-outside-block....ok
t/good.................ok
t/item-ordering........ok
t/load.................ok
t/missing-file.........ok
t/pod..................ok
t/selftest.............ok
t/spaced-directives....skipped: Not written yet
t/unknown-directive....ok
All tests successful.
Files=11, Tests=19,  1 wallclock secs ( 0.11 usr  0.05 sys +  0
.84 cusr  0.18 csys =  1.17 CPU)
Result: PASS
```

Figure 1: Example test execution upon a module still being developed

Sequential processes do not share any resources during their execution, therefore change propagation might not affect the other process' results. Incrementality is observable e.g. in MDD [7], where test case skeletons are created first, then these skeletons are processed mostly separately. This process includes more detailed specification of test cases. Using independent test cases, the incrementality principle appears only in the phase of their specification. In the case of hierarchical test structure, the dependencies are active through the whole preparation and execution process.

Non-incremental software processes use always sequential execution that induces a less complicated first-time test preparation but a more complicated regression testing during maintenance. Maintenance is the life-cycle phase when the regression test selection and test review and modification appear to be incremental. Figures 1 and 2 present practical examples of test grouping to ease test selection and regression testing.

```
Running make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/00-load.........1/3 # Testing LaTeX::Parser 1.00, Perl 5.0080
06, /usr/local/bin/perl
t/00-load.........ok
t/01-html.........ok
t/pod-coverage....ok
t/pod.............ok
All tests successful.
Files=4, Tests=20,  0 wallclock secs ( 0.05 usr  0.03 sys +  0.
23 cusr  0.14 csys =  0.45 CPU)
Result: PASS
```

Figure 2: Example test execution upon a final version of a program that was developed using processes of the waterfall SWLC

3 Observation two: agile processes

Agile software development methods use best practices of software development to improve old school methods in selected areas. Extreme programming focuses on rapid delivery and high frequency of iterations with customers involved into the development process, feature and behavior driven programming emphasize on requirement tracing and fulfillment [9]. The test driven development (TDD) [10] approach uses tests as requirement representation and the logical interconnection between tests and program code is very tight. Therefore, changes are propagated across all code not only in the case of intended intervention, but due to the test refactoring as well.

Figure 3 shows a test case reflecting functionality and architecture of the designed implementation. The test case says *Address should contain Street, Number, City, PostalCode attributes; Street should be changeable.* After the implementation of a program that satisfies this requirement, the test case will be extended by other requirements on *Address* one-by-one that incrementally changes it. After any change within the tests, these tests are run to check

```
public class AddressTest extends TestCase {
 public void testStreet() {
  Address address = new Address("Letna","9","Kosice","04200");
  assertTrue(address.getStreet().compareTo("Letna") == 0);

  address.setStreet("Main");
  assertTrue(address.getStreet().compareTo("Letna") == 0);
 }

 protected void runTest() throws Throwable {
  testStreet();
 }
}
```

Figure 3: Example code snippet of JUnit test for TDD

implementation consistency. I.e. changes are propagated from tests to implementation. Considering the next requirement as *requesting Street must not return bad value*, after extending the test case, probably no change within the implementation will be needed.

4 Observation three: cowboy coding

Cowboy coding technique is an approach, where the emphasis is on writing code. It is a term used to describe software development where the developers have autonomy over the development process. This includes control of the project's schedule, algorithms, tools, and coding style. A cowboy coder can be a lone developer or part of a group of developers with either no external management or management that controls only non-development aspects of the project, such as its nature, scope, and feature set. (The *what*, but not the *how*.) An example is the .NET environment where source code in C#, Visual Basic and other languages are used together in programming task solutions. These sources are mostly translated into Common Intermediate Language (CIL) that allows besides bytecode execution a good analysis of the program [11] due to the meta-data that are stored within that bytecode.

In such a case, testing is covered within the project goals as satisfying the requirement of producing a "deliverable" product, or final product evaluation.

For the evaluation phase, acceptance tests are weighted before any other kind of tests. Acceptance test failures are reflected into functional test creation. Functional test failures indicate coding failures that should be detected by writing and running test case code. Figure 4 reflects relations between the mentioned activities. Incrementality within tests occurs in unit testing phase where old tests are revised and modified.

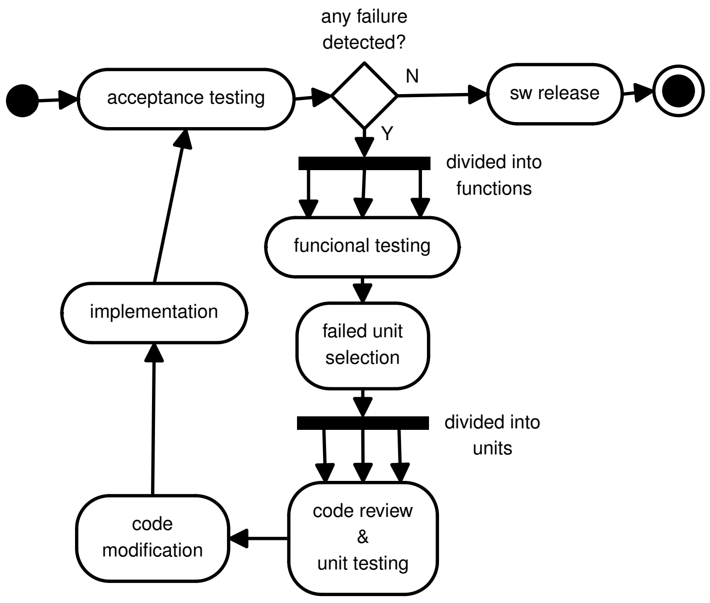

Figure 4: Example testing activities of cowboy coding

5 Conclusion

We highlighted the presence of incrementality principle in software projects and proved that it is a not replaceable feature of test preparation.

Summarizing observations gained with old school techniques, test preparation is distributed across more phases and runs in parallel to other streams of development. Communication between these streams is one-way: changes in the design and/or implementation indicate changes within test plan and/or test cases.It happens because of the main top-down development principle that introduces changes in the way of refinements at current or next (lower) levels of abstraction. On the other hand, the bottom-up strategy needs more independent tests first at implementation level of components. The upper (more abstract) level tests are a kind of an incomplete set of integration tests considering all possible integrations and all reachable interfaces of components. Change propagation can be observed in cases, when changes are introduced at bottom (or any other low) level.

The second group of observations tells us about the incrementality principle within projects being solved agile. Extreme programming focuses on rapid delivery of software products, iterations incrementally extend the software solution that implicitly starts change propagation. This is propagation in both ways, test might influence the design (the main principle of test driven development), and design/implementation tasks might result into changes in test plan structure or test case behavior.

The third case studies cowboy coding and points to the presence of incrementality principle even in this strategy. Test preparation is based on requirements, these requirements build the main bridge for change propagation. At the highest level, incrementality shows up in the way changed requirements effect the implementation and test cases. At lower level, when a change on the code is made, tests are created/changed only in the case of failure of tests from higher level. Cowboy coding is a fast development technique omitting the most things that are not strictly needed for successful delivery of the software product. Even testing is intended to be as independent from the application implementation as possible, but there are observations presenting some dependencies defined at top level through requirements and at lower levels by failure removal. Cowboy coding was the only methodology allowing less incrementality within our observations, but even there are maintenance and development tasks that indicate change propagation in the indirect way – by making tests fail.

The future of this research might be focused on selection and specification of common parameters to create meta-level descriptions of these dependencies that, in further, might lead to rule discovery by abstraction on these dependencies.

Recently, we run a few projects of quite simple programming tasks being different in the type of used SWLC and being common in the goal to collect test preparation incrementality characteristics. These characteristics will provide attributes and numerical values for further analysis and metrics definition [11].

Acknowledgments

This work was supported by project VEGA No. 1/0350/08 "Knowledge-Based Software Life Cycle and Architectures."

References

[1] D. Bell, I. Morrey, J. Pugh, *The Essence of Program Design*, Prentice Hall Europe, 1st edition, 1997, Hungarian translation: *Programtervezés*, Kiskapu Kft., 2003.

[2] Z. Juhász, Á. Sipos, Implementation of a finite state machine with active libraries in C++, *Proc. of the 7th International Conference on Applied Informatics*, Eger, Hungary, Vol. 2, 247–255, 2007.

[3] M. M. Lehman, J. F. Ramil, Towards a Theory of Software Evolution – And its Practical Impact, *International Symposium on Principles of Software Evolution,* ISPSE (2000) pp. 2.

[4] L. Samuelis, Cs. Szabó, Notes on the role of the incrementality in software engineering, *Studia Univ. Babeş-Bolyai Inform.*, **51**, 2 (2006) 11–18.

[5] Cs. Szabó, L. Samuelis, Notes on the software evolution within test plans, *Acta Electrotechnica et Informatica*, **8**, 2 (2008) 56–63.

[6] Cs. Szabó, L. Samuelis, Software evolution within test plans – how, when and why? *Egyptian Computer Science Journal*, **29**, 2 (2007) 1–10.

[7] B. Hailpern, P. Tarr, Model-driven development: The good, the bad, and the ugly, *IBM Systems Journal*, **45**, 3 (2006) 451–461.

[8] J. Arlow, I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley, 2nd edition, 2005.

[9] P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, *Agile software development methods – Review and analysis*, Otamedia Oy, Espoo, VTT Publications, 2002.

[10] K. Beck, *Test Driven Development: By Example*, The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[11] Z. Porkoláb et al., Application of OO metrics to estimate .NET project software size, *Proc. of the 7th International Conference on Applied Informatics*, Eger, Hungary, Vol. 2, 293–299, 2007.

Received: October 13, 2008