NOTES ON THE SOFTWARE EVOLUTION WITHIN TEST PLANS

*Csaba SZABÓ, *Ladislav SAMUELIS
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, tel. 055/602 4120, 4313,
E-mail: csaba.szabo@tuke.sk, ladislav.samuelis@tuke.sk

ABSTRACT

Nowadays, software engineering research community pays much attention to the development software design methods. We observe relatively much less attention in the field program test research. Generally, a good test plan enhances not only the reliability but also increases the reusability of the created test plan in the regression testing phase that will follow after implementing the changes into the system.

After describing the motivations and the brief theoretical background we introduce the A-shaped model of the Software Life Cycle (SWLC) with emphasis on the change propagation across the design and test plans. This model consolidates the system design and test planning phases across the SWLC.

We define relations between tests and tested elements in a form of existence dependencies: all design must have at least one test, and tests without an attached design element may signalize errors. Further we introduce the specification of a system dependence graph for modeling and visualization of dependencies between the elements. This feature is important during the visualization of the change propagation. The results are demonstrated on an example and finally discussion follows devoted to the similar models.

Keywords: *A-shaped model, change propagation, incrementality, software design, software life cycle, system dependence graph, test plan*

1. INTRODUCTION

Decades long goal has been to find repeatable, predictable methodologies or processes that improve productivity and quality of the software development. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to this task.

The idea of this paper stems from the needs to speed up the inclusion of a new functionality into the system by the incremental change and to avoid time losses due to regression test selection and/or creation.

During software development, we often have to modify arbitrary parts of the system or just to extend it by a new component. In case when the modified component is used (or referenced) by any other component of the system, it is necessary to change the influenced components too. This process is called change propagation [11]. After implementing the required changes, the regression testing of the system follows in order to ensure that the functionalities remained the same.

In order to shorten (minimize) the development time of software projects, it is very useful to design a model for speeding up the test preparation for later regression testing. This paper introduces the A-shaped model for decreasing program development time by consolidating the design and test planning phases using the incremental change approach.

2. THEORETICAL BACKGROUND

There exist two approaches to facing the SW crisis:

- to develop a general methodology for the development of all kind of SW systems, or
- to develop special methods that will be devoted to the special goals of the project.

2.1. Methods and models

Lots of SWLC models and related methods are discussed in [3, 9, 15]. Some of them are more universal; others prefer early release to universality. Next, we present (not exhaustively) some examples of SWLC methods:

1) Waterfall [3, 9, 16]
2) Staged [13]
3) Spiral [1, 16]
4) Incremental (also in [14])
5) Unified Process (UP) [1]
6) Extreme Programming (XP) [4]
7) Feature Driven Development (FDD) [21]
8) Test Driven Development (TDD) [2]
9) Dynamic System Development Method (DSDM) [20]
10) Agile Model Driven Development (AMDD) [20]

The first six methods listed above are discussed as core methods at large, the second half of the list includes methods that stam from XP emphasizing selected aspects of development. Some of them support good maintenance by partially less effective development, others support fast development with less abilities in maintenance area. From the viewpoint of testing, a method with good support for both mentioned procedures is most acceptable.

We agree with the authors of [3, 15] and [20] that there will be always requirements in the middle of interest, but these are processed and taken into consideration in different ways in each SWLC method. Different is the level of abstraction in the views on the

system being constructed and the used programming technique too (e. g. object-oriented – OO).

2.2. Top-down versus bottom-up development

There are two main approaches in SW development that are applied in the methods mentioned above: top-down and bottom-up development. Brief description follows.

Top-down development (TD).

This is called also as "model driven", where the *requirements* (expressed in model) are transformed into the target code.

Bottom-up development (BU).

This process reuses patterns of the target code (or Components Of The Shelf - COTS) in order to fulfill the *requirements* of the specifications. This process may involve unwanted functionalities into the final product.

In practice the SW industry blends these two approaches.

2.3. Iterations and incrementality

As noted above, there are many approaches that present some aspects of the incrementality utility and are used under names like incremental learning, evolutionary and revolutionary rework, program synthesis and incremental building.

Therefore, a clear definition of the *"iteration"* and the *"incrementality"* turns out to be vital. Here are the definitions:

- We define that *"iteration"* refers to repeating an activity, e. g. phases, in the software development process. Iteration is applied e. g. in refactoring when developers perform semantics-preserving structural transformations usually in small steps. Motivation for the improvement may be focused towards the enhancement of the efficiency of the code with respect to the time or space complexity or towards the improvement the structure so that developers can more easily understand, modify, evolve and test it. The research domain that addresses this problem is referred also as restructuring.

- On the other hand "*incrementality*" refers to the process of adding new functionalities through successive implementations. This is a significant and essential difference to the iteration and deserves much more attention. First of all the incrementality principle has its mathematical roots and is explained in the theory of inductive inference [17]. This approach to problem solving is also called generalization. Incremental software development is sometimes called build a little, test a little. We may observe the similarity between building concepts and models in software engineering and building hypotheses in mathematics. This process is very clearly highlighted in Polya's classic work, "How to Solve It" [18].

2.4. The place of testing

Each SWLC model defines a sequence of phases. One of them is testing. The role of testing is clear: to ensure the product offers the required features. The placement of the test differs in the sequence of phases.

In the case of UP, each phase may contain a number of workflows and testing is one of the core processes. The tested products may be different [1].

Extreme programming stipulates a set of best practices that collectively encourage core values such as feedback and simplicity. The feedback occurs in the form of tests, by delivering products in short iterations, and by the simple expedient talking between the developer and customer. Rapid development is achieved by rapid refactoring [4].

TDD is not about testing, it is a development method that uses tests in its steps [2]. The point of TDD is to drive out the functionality the software actually needs, rather than what the programmer thinks it probably ought to have. The way it does this seems at first counter-intuitive, if not downright silly, but it not only makes sense, it also quickly becomes a natural and elegant way to develop software.

We can conclude this section by the statement that all SW development methods deal with the problem of testing and define a stable place for it in their workflows. Fig. 1 visualizes this statement, where: R denotes requirements, D represents design and P stands for the test plan. The triangle denotes the relationships between these three key subjects.

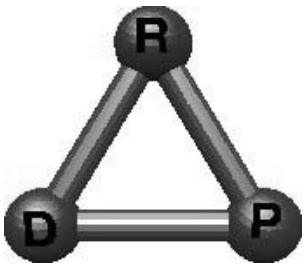

Fig. 1 The triangular relationship between R, D and P

2.5. Testing and test design

Software testing is the process that helps identifying the correctness, completeness, security, and quality of the software under development. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. Testing phase is part of

each SWLC either TD or BU, either less or more so-called agile.

Testing is important *before* any release. This is the claim of all release-oriented methods. Others claim that testing may be independent from the release phase and it has to be executed as often as possible. In all cases, testing is very relevant to the final product and documentation. However, we agree with the Dijkstra's statement that "test could reveal bugs but do not prove the correctness of the program".

To run tests effectively, we have to have a test plan. The more revised plan we have, the better. To get one, we need to design it and that is why it is a part of the SWLC.

2.6. Test plan design

Test plan design is a phase and a model too. From the point of view of the development process, it is a group of activities around test preparation: planning, generating or developing tests. On the other hand, it is a model that may have its sophisticated structure according to the selected standards and it may be documented as well.

Test plan preparation answers for the questions: *what? when? why (not)? how? who?* These are classical questions of planning.

Test plan design in its closer meaning includes all tasks according to the creation of the test elements - concretization of the *what-how-when* question triangle.

There is another way of test plan preparation: partial replacing of the design by generation. Test generation [8] is the automated way of test creation that can be made from:

- application model, or
- source code.

Test structure and documentation are the other side of test design. This point of view declares a model.

The use of standards in the testing phase is important from the maintainability point of view. Common standards or categories used more or less are as follows:

- nothing or intuition,
- (internal) company standard,
- programming standard (e. g. JUnit [2, 5]),
- TTCN/UML TP [7],
- IEEE 829–1998 [6].

These standards describe the wanted structure as well as the way of documenting the testing process. Their combination enhances the reliability of the final product.

All above-mentioned standards recommend a structure that can be implemented in a specific form of a system dependence graph (SDG) investigated by Yu and Rajlich in [11] and by many others from the incremental SW development community.

2.7. Regression testing

Regression testing is the execution of all tests on the system during its development and/or before a major or minor release of the system being developed. It includes all unit, integration, functional and system tests.

For each kind of tests there must exist an execution plan and a set of expected results [12]. This plan results from requirements analysis and from the expected programme results.

Regression testing is an integrated part of extreme programming (XP) [4]. In this methodology, the design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage of the SWLC.

The core regression testing method may include all tests for the system. This claim is more precise in the way that regression testing ensures that all old functionalities remain in the system and work further properly.

In the case of very large systems, the execution of all tests may take a longer while, therefore new techniques are needed to decrease the amount of the tests by selecting only the ones which are in some way related to the change executed during the system evolution. This task can be fulfilled either using a table of test-code coverage or by consideration of the relations between tests and the elements of the design at higher level of abstraction.

The better-known method of test-code coverage needs an initial execution of the tests on the older system (before the changes are applied) that creates the table (records all relations at the level of lines of source code). The significant differences in the lines than signalize which tests have to be re-run during regression testing.

3. THE A-SHAPED MODEL

This model (shown on Fig. 2) copies the classical sequence of actions from the waterfall model for both the application and its test development. It stems from the requirements and branches into two processes, which end with implementation of the application resp. implementation of the test plans. We introduce here and emphasize the mutual influence between the development and test planning phases based on observation we state that some activities may be executed in parallel.

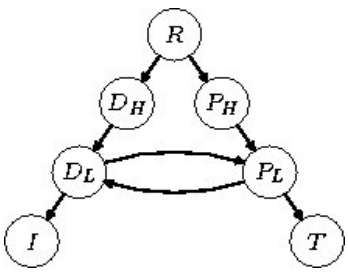

Fig. 2 The A-shaped Model

We describe the activities within this model as follows:

Requirements engineering (R).

In this phase the requirements [1, 10, 15] are gathered and preprocessed in a way of their separation into two sets. One of these sets is the basis for the application development, the another one is the basis for designing the tests.

Design – high level (D_H).

This phase includes the implementation of the functionalities at the highest level of abstraction.

Design – low level (D_L).

In this phase we refine the ideas from the higher levels iteratively and incrementally. In separated cases, the test planning process may produce tests due to those some selected elements of the design may change i. e. test-driven development of the functionality [2, 4].

Planning – high level (P_H).

This phase is about functional test planning. Test plans are prepared for testing functionality and are decomposed (hierarchically) at this level.

Planning – low level (P_L).

The detailed planning of tests and dependency analysis follows the decomposition. After the development of the structure of functionality and design-oriented tests, a new functionality can be introduced via inclusion of a new functionality oriented test or via a new design element at the selected level of abstraction. In other words, inclusion (or deletion) of functionalities may be executed in both test driven [2, 4] and classical (in the design process [3]) way.

Implementation (I).

This is the final phase of refinement (design) where all the coding takes place.

Test implementation (T).

The test plans are implemented in the form of a program, or other testing code.

3.1. Description of the Phases and their Results

Analogously to the waterfall model, we distinguish between R- (from Requirement), H- (from High level), L-phases (from Low level) and implementations in this model.

R-phase

The first phase belongs to the requirement engineering [1, 10]. The outputs are two subsets from the perspective of their usefulness in the design and testing. During this process the requirements are decomposed and categorized. Categories serve for better requirement tracing and separation.

Two phases follow after the R-phase in parallel: higher level design and planning of testing.

H-phases

At the higher level, we can see two phases executing in parallel. Both of them are based on functional and structural decomposition as refinement activities. Higher level design interprets architectural ideas of the system being developed, higher level planning outputs functional testing concepts at a very high level of abstraction – the basic structure of the upcoming tests on both architectural and behavioral base.

Both phases result into their lower level correspondents.

L-phases

The core of the SWLC model is built up from these phases, D_L and P_L (the L-phases). It is the point where the parallel threads are synchronized. These two phases may be executed in parallel but there is a significant influence between them that makes the core of the method incremental.

A detailed model of the system is designed in the D_L phase that includes the full architectural and behavioral specification of the system in the modeling language selected by the designer. The whole model is built from the results of the higher level design using similar refinement steps. The only difference is, that all data are specified here with the design of the operations with them.

The P_L phase results into the model of the tests, the behavioral and architectural specification of tests, test contexts and data. The test cases are refined to the crisp values and dependency definitions (e. g. which design element is tested by which test case). These results come from the stepwise refinement of higher level test specifications.

Implementation phases

Implementations (the final system and the implemented tests) are generated during the whole development many times as prototypes. These implementations are the outputs of the actual models at the L-phases. The line between the design and the implementation is clear: the point of applying a concrete, programming language specific aspect. The mentioned border is that between the portable and special architecture.

Further, the implementation of the system is stressed against the corresponding tests in the testing procedure.

The features of A-shaped model (as always from a certain aspect) can be divided into two groups [3], e. g. advantageous and disadvantageous ones. All these already known features are discussed in [10].

3.2. Remarks on the evolution in the test plan

The model of the tests includes records about the used (tested) design elements, which are the traces for the change propagation or just for the dependency monitoring. These records allow defining change propagation across both models.

Changes caused by design activities propagate changes in the test model in the form of a changing requirement what and/or how to test. In this way, changing the initial requirements for the affected tests does involve change propagation. The adaptation process (to the change) is de facto started as introduction of a new requirement or a modification into the requirement set. Looking at it from a wider perspective, there is an evolution inside the SWLC model. This idea works vice versa for the design thread.

3.3. Remarks on the parallelism

Considering the two outputs from the R-phase (no influence between L-phases) we can split the activities into two groups. The design thread is than the same as in the waterfall model. The test development represents as a separated development of the testing application.

The parallel threads allow an independent design of tests and the application, but without the joining of them we loose the ability to design complete tests, e. g. tests specialized (passed) to the designed components of the system. On the other hand, joining of the test planning thread with the application design one provides the ability of test driven development [2] of some parts of the system.

Considering both test evolution and parallelism in A-shaped model we can animate these processes as the work around an ever-changing hypothesis (h_i) as we show in [9].

4. SYSTEM DEPENDENCE GRAPHS – A SOLUTION?

The models used in the different stages of the SW development differ and there is no universal notation thus far that could be used for the requirement, application and test modeling with considering the weak and strong relations between them.

Using system dependence graphs (SDGs) is a well-understood method for modeling relationships between non-homogeneous elements (elements that are internally implemented in different ways and may in general belong to different modeling aspects) as requirements, tests and implementation details of the application are. Considering the relations shown on fig. 1 there are three types of so-called sub-models representing the set of requirements as R, the application model elements as D (the design), and the set of tests as a whole as P (the test plan).

4.1. The R sub-model

The first sub-model according to the A-shaped SWLC model phases is the model of the requirements. It represents the relations between the requirements as their category and priority too. The relations may be of type *„is part of“*, *„depends on“* etc. The categories are project specific and could represent the belonging to the bigger parts of the system as requirement groups. Prioritization of the requirements may be taken from the DSDM [20] method: MoSCoW [19], e. g. *m*ust, *s*hould, *c*ould, won't have but *w*ould like in the future.

4.2. The D sub-model

Design representation is a standard SDG with a level of abstraction chosen by the designers. It is not important to have the slices at instruction-level. This sub-model just must make possible the change propagation when a new element is introduced, an existing removed or changed [11].

4.3. The P sub-model

The P sub-model follows the tests' structure and as it is a SDG the relationships between the tests too. The relations may be hierarchical (*„part of“*) or defined by the correspondence to the categories of the related requirements. The granularity of the elements of this model depends on the chosen way of representation. In the case of textual descriptions [6], the elements are those documents. In the case of object-oriented modeling of the tests, the elements are as defined in the modeling language of the chosen framework, e. g. UML Testing Profile [7].

4.4. Putting the sub-models together

Now, we know about the sub-models and the main ideas of their relationships (fig. 1). Following that, we define a new kind of dependence at project-level: *inter-part dependence*.

Inter-part dependence (IPD).

IPD interprets relations between elements in different sub-models that are processed in the SW project. Those can be between elements from the R and D sub-models denoting implementation of required feature, ones from the R and P sub-models denoting functional testing aspects, or ones from D and P sub-models denoting unit, integration and/or system testing aspects [1, 3, 5, 15, 16].

The creation of a complex SDG for the whole project including all sub-models and IPDs gives a tool for dealing with changes in any sub-model as with a more complex change that can influence the product and its evaluation as well. E. g. the change of the requirement may cause a change in the SW product and its testing procedures too.

4.5. The web service example

Creating web services (WS) is a typical TD process that allows only narrow changes to the requirements during both high and low level design stages without touching a huge group of these requirements. Therefore, we can divide the set of the tests into two sections according to the implementation-specific and the implementation-independent (those that evaluate fulfilling of basic requirements) tests. This is a typical use case where the A-model is applicable, because the functional tests can be developed parallel to the high level design of the WS. Any modification to the implemented WS are easier to made due to the already existing SDG that was built during the development of the basic implementation.

Due to the limitations on the length of this paper, we do not include all details about this example.

Step one: execute the R-phase.

We collect and organize requirements to the WS. In our example, these are related to the task of reporting activities (e. g. new posts) within a web forum of a specific portal. The requirement categories will be as follows: user-interface (UI), portal-specific (PS), other. We send all directly implementable requirements to the next design phase and all functionality related ones to the high level test planning phase. The relations between these requirements define a basic SDG for them.

Step two: parallel processing of the requirement-subsets by the designers (T1) and test developers (T2).

Both teams describe the required interface in a language they can work with in the next periods, e. g. WSDL [22]. T1 members create the higher level structure of the WS implementation, T2 members the higher level structure of the general test plan including basic communication aspects. Both teams extend the parts of the SDG related to their issues.

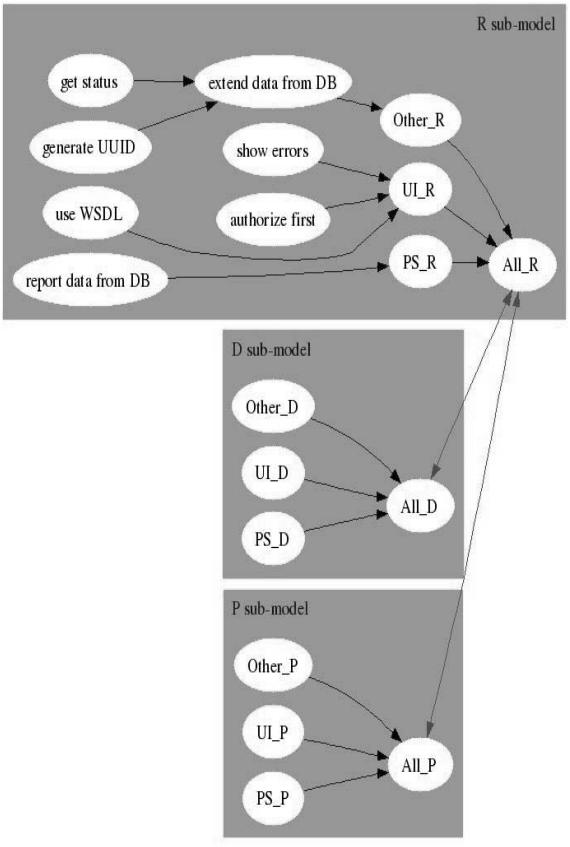

Fig. 3 The highest level view on the SDG being built

Fig. 3 shows the highest level view on the WS, the only known information is that requirement are related to the design and to the test plans. In this phase we do not have any knowledge about the relations between tests and the design (e. g. the application being developed).

Step three: getting synchronized.

First, the teams join their partial SDGs on the basis of correspondence to the requirements, e. g. we get a more complex SDG. After that, both teams work independent to each other, but they follow the indications got from change propagation across the SDG. It means, that a change in the lower level design of the WS may indicate the need to update some related tests (or the absence of them). The main issue of the A-model is there: tests change in time with the application as they were maintained in an evolutionary development process. Using SDG with bidirectional edges representing IPD, change propagation can arise by accessing the test plans first as well.

Step four: implementing the WS and the test (separately).

Members of T1 and T2 compile their implementations.

Step five: deploying the WS and running the test application.

T1 deploys the WS and sends required informations to T2 (e. g. WS location URL), then T2 members run their tests to evaluate the WS, write their reports etc.

Maintenance

If any failures are found in the WS, the already existing SDG indicates the parts of the system related with the failed test(s).

WS development is not an area, where the evolutionary approach is commonly used due the facts listed above, but the hard maintenance can be made easier using the A-model methodology.

Figure 4 shows an example evolution indicating situation after a change in a function inside the D sub-model. Colors on the figures have the meanings as follows:

1. green areas represent the sub-models such as R, D, P;
2. white ellipses are the nodes of the SDG;
3. text in the ellipse is the (unique) name of the concrete element;
4. black oriented edges represent relations within a sub-model;
5. red bidirectional edges represent IPDs;
6. red highlighted ellipsis is the change location;
7. yellow highlighted ellipses are the places where the change could be propagated.

At the first sight all related elements of the project are to be updated, therefore there is a need to evaluate the strength of the change to avoid the indication of irrelevant (but related) elements.

That needs an evaluation function or better a prediction mechanism.

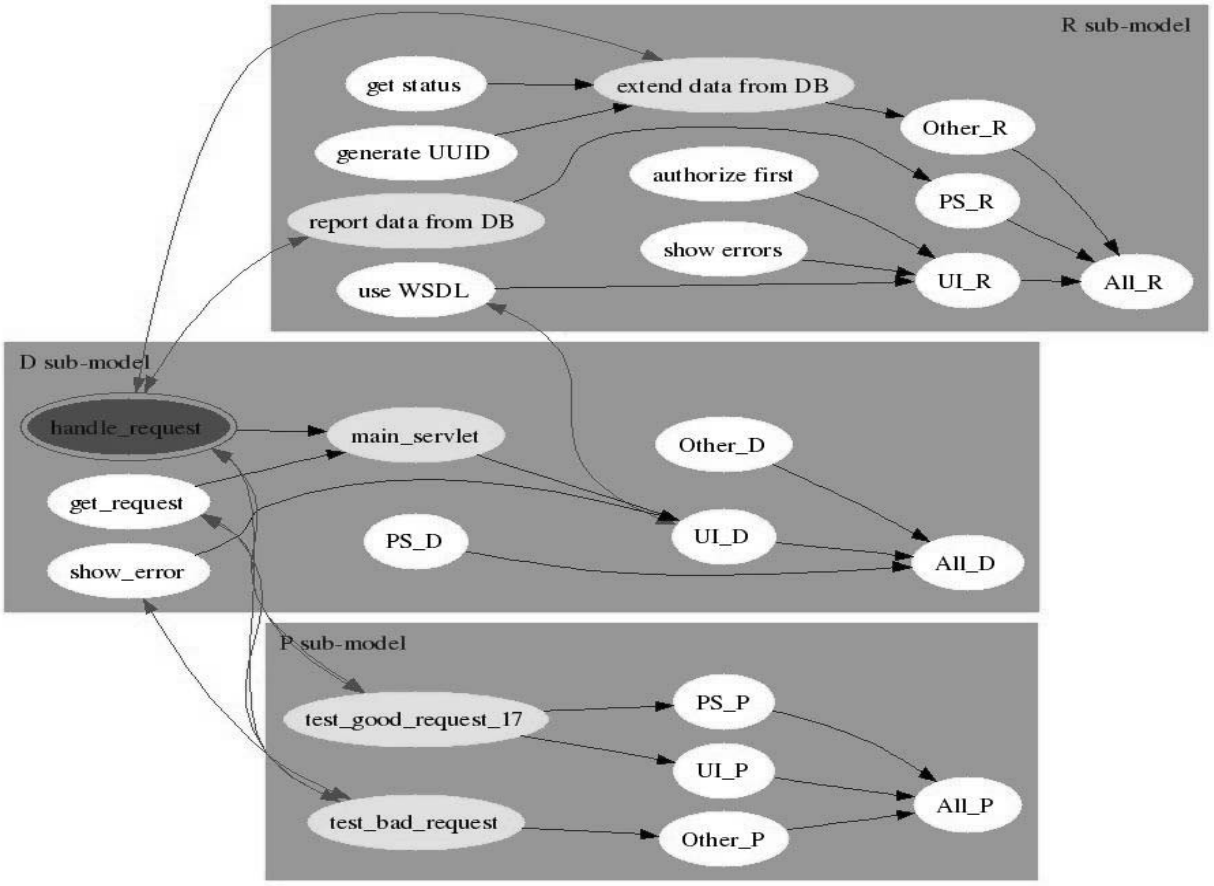

Fig. 4 Evolution indicated after changes in method *handle_request*

5. CONCLUSION

We showed selected number of methods of SW development and the placement of testing and/or test planning within them. Our result is that the majority of methods do not fully cover the problem of test design. They only specify the placeholders for the *what-how-when* question triangle. Other methods such as TDD rely on the test model even if it is represented in a form of lists and does not include any relationships between the tests.

We have introduced the A-shaped SWLC model and its pros and cons. This model covers the test planning phases of the SW development; it shows the location of these phases and the dependencies between them.

The model covers the evolution of the tests via considering the design as the extension of the set of the requirements for test design and planning. It may include the possibility to generate test cases to the design [8], but with the extension to map the relations between these test plans and tested design elements.

We showed a shortened example of a use case of WS development task, where classical methods do not fully support maintenance or fail as a core TDD may due the lack on interface description.

The next step may refer to the extension of the system dependence graph [11] by the test plans and the requirement hierarchy and putting it to a higher level of abstraction (considering not only classes as elements mostly within the D-model's SDG) to allow the usage of the model with other than OO methods.

ACKNOWLEDGMENTS

The research was supported by the grant *Technologies for Agent-based and Component-based Distributed Systems Life-cycle Support*, Scientific grant agency project (VEGA) No. 1/2176/05.

REFERENCES

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

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

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

[4] Chromatic. *Extreme Programming Pocket Guide*. O'Reilly Media, Inc., 1st edition, July 2003.

[5] E. Gamma and K. Beck. JUnit, Testing Resources for Extreme Programming.

http://junit.org/index.htm, 24 November 2005.

[6] IEEE Standard for Software Test Documentation, IEEE 829–1998, 1998.

[7] Object Management Group. UML Testing Profile, Version 1.0, July 2005.

[8] J. Offutt, Sh. Liu, A. Abdurazik, and P. Ammann. Generating Test Data From State-based Specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[9] L. Samuelis and Cs. Szabó. Notes on the role of the incrementality in software engineering. *Studia Universitatis Babes-Bolyai Informatica*, 51(2):11–18, 2006.

[10] Cs. Szabó and L. Samuelis. The A-Shaped Model of Software Life Cycle. In *Proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, Poprad, 25-26 January 2007, pages 129-135, 2007.

[11] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proc. International Workshop on Program Comprehension*, pages 293–299. IEEE Computer Society Press, 2001.

[12] W. D. Woodruff and R. Pisechko. Efficient Test Planning and Tracking. *Software Quality Professional*, Volume 5, Issue 2, March, 2003.

[13] V. Rajlich and K. Bennet. A Staged Model for the Software Life Cycle. *IEEE Computer*, 33(7):66-71, July 2000

[14] V. Rajlich. Incremental change in object-oriented programming. *IEEE Software*, 21(2):62-69, July/August 2004

[15] I. Sommerville. *Software Engineering*. Addison-Wesley Publishers Ltd., Pearson Education Ltd., Boston, MA, USA, 7th edition, 2004.

[16] Cs. Szabó. The V-shaped model from the testings point of view. In *Proceeding from the 6th PhD Student Conference and Scientific Competition of Students of Faculty of Electrical Engineering and Informatics, Technical University of Košice*, pages 127–128, Košice, Slovakia, 2006. elfa, s.r.o.

[17] D. Angluin and C. H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):238–269, September 1983.

[18] G. Polya. *How to solve it: A New Aspect of Mathematical Method*. Princeton University Press, 2nd edition, 1957.

[19] MoSCoW Prioritisation. URL: http://www.protoolkits.com/Analysisandrequirements/Analysistechniques/moscowprioritisation.html

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

[21] Feature Driven Development (FDD),

http://www.featuredrivendevelopment.com/

[22] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

Received Jun 18, 2007, accepted November 12, 2007

BIOGRAPHIES

Csaba Szabó was born in 1979. In 2003 he graduated (MSc.) with distinction at the department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University in Košice. Currently, he is a PhD. student in the field of software engineering and information systems; his thesis title will be "The A-model: Emphasizing the evolution within the tests during software design". Since December 2006 he is working as a research assistant with the Department of Computers and Informatics. His scientific research is focusing on object oriented programming and test design. In addition, he also investigates questions related with the quality and maintenance of programme systems.

Ladislav Samuelis, Assistant Prof.: Obtained MSc. in Electrical Engineering at Prague Technical University (1975), and PhD. in Informatics at Budapest University of Technology (1990). Has been engaged in research into the automatic program synthesis at the Institute of Computer Technology at the Technical University of Košice, Slovakia. Since 1998 affiliated with the Dept. of Computers and Informatics, Faculty of Electrical Engineering and Informatics, taught Operating systems, Database systems, Computer Networks and Java. Currently is involved in software engineering metrics and in the principles of software evolution.