STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LI**, Number 2, 2006

NOTES ON THE ROLE OF THE INCREMENTALITY IN SOFTWARE ENGINEERING

LADISLAV SAMUELIS AND CSABA SZABÓ

Abstract. The incrementality principle appears in various contexts in the relatively short history of software engineering. This fact seems natural because the processes like software comprehension, design, refinement and implementation are done incrementally in practice. Due to this common fact the incrementality principle is interpreted often superficially in the software engineering literature. The aim of this paper is to highlight the ubiquity of the incrementality utility in software engineering literature and to clarify the concepts behind its role in various contexts.

1. Background and motivation

The notion of incrementality is being applied simultaneously both in the field of artificial intelligence (e.g. in the field of machine learning) and software engineering.

While reading the software engineering literature we observe that the incrementality principle is mentioned and utilized frequently in various contexts in its relatively short history (see in the next paragraph). This ubiquitous presence of the incrementality principle, almost in every software development paradigm, motivated the investigation towards unrevealing the similarities and differences between its interpretations in various contexts. Another source of the motivation is the challenge to condense the knowledge about the incrementality principle used in contemporary experimental software engineering practices. The aim of this paper is to pave the route towards the understanding and reuse of this notion in the future contexts. In order to achieve this goal, we will investigate the utilization of the incrementality utility in several selected software development approaches.

Received by the editors: September 13, 2006.

2000 *Mathematics Subject Classification.* 68N19, 68N30.

1998 *CR Categories and Descriptors.* D.1.2 [**Software**]: Programming Techniques – *Automatic Programming*; D.2.2 [**Software**]: Software Engineering – *Design Tools and Techniques*.

2. The ubiquity of incremental tasks

The following selective sample of works (not exhaustive) shows the wide range of the usage of the incrementality utility. A brief historical overview of the *"incremental and iterative development"* is presented in the work of C. Larman and V. Basili [11]. This work summarizes the role of the iterative and incremental software development through significant software projects since the mid-1950s. It focuses on the incrementality utility, applied in the software engineering processes, from the managerial point of view. Describes the driving thoughts and misbeliefs, which were behind the practices applied in the past decades in the field of software engineering.

The incrementality concept is stressed and dealt also in the field of program comprehension, which is already a matured topic with standalone conferences [8]. It is not a trivial task to understand the architecture of object-oriented programs [22]. It is obvious, that to enhance the functionality or to add new features requires to comprehend the task in more detail. As software engineering topics evolved, the incremental comprehension of programs came into focus. In other words it means that without the thorough analysis it is impossible to make effective reform or re-engineering. Remarkable is the statement of K. Nygaard [5] who said: "to program is to understand". We accept in general that comprehension is also a continuous iterative and incremental process. The fact that problem solving does not progress in a linear manner from one activity to the next is highlighted as the conjecture: "empirically based models mature from understanding to explaining and predicting capability". This conjecture is explained in the handbook of authors A. Endres and D. Rombach [6, chap. 12], which is devoted to the empirical aspects of software engineering.

The recent work of C. Larman [10] stresses the role of the incrementality against the waterfall model in the software development. In addition, the work stresses its crucial role in the history of the software engineering and considers it as a fundamental revolutionary change against the waterfall model of the software development. Authors R. E. Fairley and M. J. Wilshire [7] exhaustively identify the nature of the iterative rework. They point to the fuzziness between the avoidable and unavoidable rework and the incrementality issue is dealt from this point of view.

The field of software design also uses the notion of iteration and incrementality, e.g. Arlow and Neustadt characterize iterations, in association with the Unified Process [3], as "mini projects, which are easier to manage and complete than the original large SW development project".

Machine learning differentiates between nonincremental and incremental learning (e.g. in [14]). The same terms are marked in [17] as revolutionary and evolutionary learning strategies. In essence, the difference between these two definitions

lies in the fact that the non-incremental (revolutionary) approach is based on one-shot experience and the incremental (evolutionary) learning allows the learning process to take place over time in a continuous and progressive way, taking into consideration also the history of the training sets at building the inferred rules.

Incremental change plays important role in practical software engineering. At the present time the incremental change in object-oriented programs are in the focus (see for instance [20]). These activities investigate the impact of adding new functionalities into the code and finding the relevant program dependencies. Incrementality is of importance to software visualization too [9]. The aim is to get a better comprehension of the software behavior by representing complex structures graphically.

We may conclude the above mentioned remarks by the statement of E. W. Dijkstra: "the only available technique for effective ordering of one's thoughts is by separation of concerns". This approach leads to the observation of new facts and in ths way to improve incrementally the previous knowledge. In summary, the incrementality utility is a broad term and is in the focus of software engineers from various aspects. In the next sections we narrow the focus towards revealing the principles behind these experiences and endeavours.

3. Iteration and incrementality

The objective of the software development is to model a certain aspect or abstraction of the reality [16]. Software engineering, as every engineering discipline, is characterized by trials and errors, which are necessary steps for clarifying the comprehension of the requirements, design and implementation. It consists of many small steps and it is necessary to take into account plenty of details. Software systems are becoming more and more complex over time. They are changing and modified; the complexity arises and we need more time to comprehend them. The maintenance of the final code and other software artifacts consumes more time too.

As noted in Section 2, 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 [2]. This approach to problem solving is also called generalization and will be explained in more detail in Section 4. 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" [19].

The empirical evidence from the real-world software suggests that learning or incremental program development is possible only when the data are presented incrementally. For instance programming languages dispose with constructs, which help to postpone solving some issues. As an example is the exception mechanism in the object-oriented programming. This process makes, of course, the software more complex and drifts away from the original design. These facts may lower the quality of the software but it is the task of the validation and verification to ensure the formal quality software.

To sum up, the incrementality principle is ubiquitous in the literature devoted to the software engineering. After every step we discover new requirements, analyze them, plan, implement and test. Every iteration adds new insights and the system grows in this way, or logically clarifies. In other words, software programs are too complex to try to get the details of any one artifact entirely correct without some amount of experimentation. Software developers' ideas are evolving as they work and the steps are associated with the progress, this is evidence.

4. Incrementality in the software development

In the next sections we will focus on the application of the incrementality principle in software engineering paradigms, which are aimed at program synthesis. The selected paradigms are as follows:

(1) Programming by examples,
(2) Automatic program synthesis from specifications,
(3) Test driven programming,
(4) Programming by sketching.

4.1. **Programming by examples.** Programming by examples (positive or negative) seems popular and recurrent topic to the software research community, but often is neglected the fact that this approach has very limited usage actually and it is not generally applicable [21]. Let us analyze it in more detail. The principle of

Programming By Example (PBE), or Programming By Demonstration (PBD) or doing by watching, was investigated intensively around the eighties and a survey is available in the work of H. Lieberman [12].

We have to be aware that the paradigm of programming by example has theoretical background in the theory of inductive inference, as we mentioned in Section 3. Thought provoking is again the work of G. Polya [19], who declares the fundamental role of the mathematical induction in the problem-solving domain.

First attempts to synthesize programs by examples were done in the field of the automata theory. The task was to synthesize finite automata from a set of examples. The examples were defined by set of pairs (state and transition). The work of A. W. Biermann [4], e.g., describes practical results from experiences with implemented incremental algorithms.

Programming by example is recently also mentioned as an exciting new technology in the work of H. Lieberman and C. Fry: Will Software Ever Work? [13]. But in reality this approach is old and characterized in the work of A. Endres and D. Rombach: A Handbook of Software and Systems Engineering. They state that in fact the code generated from test cases will satisfy all test cases but we will always need one more test case because the generalization delivers a model which need not cover all test cases [6, p. 89].

What is the definition of the incremental algorithm? An algorithm is incremental if, for any given training example $e_1 \dots e_n$, it produces a sequence of hypotheses $h_0, h_1, \dots, h_n$, such that h_{i+1} depends only on h_i and the current example e_i. Now we may substitute various software artifacts, e.g. components, for h_i, which may appear in various contexts of the software development cycle (see Figure 1).

In other words it means that the hypothesis built by inductive inference will be compatible only with the current set of the proposed examples and nothing more. E.g., when we construct a cycle, we widen the scope of the algorithm in inductive way. This is the inherent feature of the inductive inference that newly generated hypothesis need not follow the intended functionality.

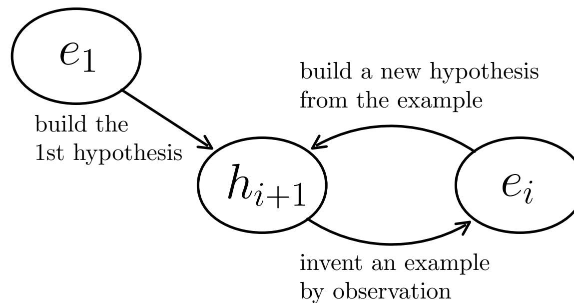

FIGURE 1. The principle of the incremental algorithm.

4.2. **Automatic program synthesis from specifications.** Synthesis of programs from specifications is a methodology, which allows to construct the program code automatically from the specification. The essence of these methods lies in the transformations, which could be used for the modification of the specifications and in order to reach the final code. The final code is constructed by the application of transformation rules to the specifications. Such a program is verified against the specification and that is why it is not necessary to prove the correctness of the program additionally. Recent review can be found in [18]. The problem with this approach is that it is an illusion to expect that perfect requirements can be formulated ahead of time. Both developers and users may need some feedback. They require a *(learning cycle)*. This is cited again from the Handbook of Software and Systems Engineering [6, p. 15]. The role of the incrementality is hidden behind the phrase learning cycle. In other words, we cannot predict the correctness of our abstraction with the reality. This approach is suitable only for trivial tasks.

4.3. **Test driven programming.** Incrementality principle may be observed both in the specification and in the implementation phases of the software development. The following simple figure illustrates the idea.

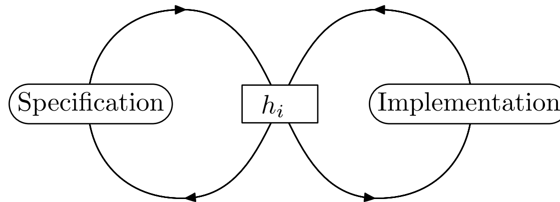

FIGURE 2. Location of the model h_i.

On the specification (left hand) side we may consider the examples e_i as test cases t_i against which the requirements are clarified and in this way h_i is continuously modified. If we consider h_i as a fixed artifact (e.g. program) and we intend to test the program (approaching the picture from the right hand side) then we may consider the tests t_i as examples e_i, which have to be accepted or ejected by the h_i. Of course, testing shows only the presence of bugs and not the absence of errors (Dijkstra's law [5]). This simple picture shows the relativity of the concepts as *"example"* and *"test"*. Both concepts' interpretations depends on the context. To sum up, the interpretation of the examples and the tests depends only on the actual agreement whether h_i is fixed or not. We stress again, that the principle behind building h_i is the incremental principle as mentioned in Section 4.1. If a new example has to be embedded into the actual model then the incremental algorithm has to modify (including the regression) the already prepared model. In each case the final model has to comply with the new set of examples.

4.4. **Programming by sketching.** Programming by sketching is motivated by the fact that programmers may acquire powerful help during the software development. In fact this approach helps in the software development with the visualization of available components. The group led by R. Bodik [15] recently experiments with this approach in practice. This approach argues that the final work of coding has to be done by computer and also the computer has to ease developers' work by provision of efficient support, or data retrieval, in order to create the final code more efficiently. Programming by sketching relies on the fact that programmers are sketching actually the skeletons (e.g. applying patterns or components off the shelf) of programs and the coding is done automatically. This approach is also part of the agile programming [1]. Following our definition of the incremental algorithm in Section 4.1, we may observe that in this case the artifact of the incrementality paradigm is the code segment or component.

5. Conclusion

The aim of this paper was to show and discuss the role of the incrementality principle in selected software development paradigms, which do not fit together at the first sight and in this way to provoke thinking. We have tried to synthesize the scattered fragments of the applied incrementality principle in the software engineering literature in order to create a more condensed foresight. We know that we did not invent a new solution to an existing problem, we dug out rather old ideas and observed them in new contexts. There is no doubt that new paradigms will emerge in the future. It is the task of the informatics to show the role of the principles in the emerging fashionable ideas.

References

[1] Scott W. Ambler. *The Object Primer 3rd Edition, Agile Model Driven Development with UML 2*. Cambridge University Press, 2002. ISBN: 0-521-54018-6.
[2] D. Angluin and C. H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):238–269, September 1983.
[3] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley, second edition, June 2005. ISBN: 0-321-32127-8.
[4] Alan W. Biermann. Automatic programming. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, January 1992.
[5] L. Böszörményi, S. Podlipnig (with contributions of Manfred Broy, Tove Dahl, and Marius Nygaard). *People behind Informatics (In memory of Ole-Johann Dahl, Edsger W. Dijkstra, Kristen Nygaard)*. Institute of Information Technology, University of Klagenfurt, 2003.
[6] A. Endres and S. Rombach. *A Handbook of Software and Systems Engineering; Empirical observations, laws and theories*. Pearson, Addison Wesley, May 2003.
[7] Richard E. Fairley and Mary Jane Wilshire. Iterative rework: The good, the bad and the ugly. *IEEE Computer*, 38(9):34–41, September 2005.
[8] `http://www.ieee-iwpc.org`, International Conferences on Program Comprehension.

[9] C. Knight and M. Munro. *Visual Information: Amplifying and Foraging, Proceedings of SPIE, San Jose, USA*, volume 4032. International Society for Optical Engineering, January 2001. ISBN: 0-8194-3980-0.
[10] C. Larman. History and evidence of evolutionary versus waterfall methods. `http://it.sun.com/eventi/jc05/pdf/02_Larman.pdf`. Java Conference 05, 22–23 June 2005, Milan, Italy.
[11] C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):46–57, June 2003.
[12] H. Liebermann, editor. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann, San Francisco, February 2001.
[13] H. Liebermann and C. Fry. Will software ever work? *Communications of the ACM*, 44(3):122–124, March 2001.
[14] K. Machová. *Machine learning.* Faculty of electrical engineering and informatics, Technical University of Košice, elfa, 2002.
[15] David Mandelin, Lin Xu, and Rastislav Bodík. Jungloid Mining: Helping to Navigate the API Jungle. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI05)*, pages 48–61, 2005. ISSN: 0362-1340.
[16] B. Meyer. Reality: A cousin twice removed. *IEEE Computer*, 29(7):96–97, July 1996.
[17] R. Michalski. Knowledge repair mechanisms: Evolution vs. revolution. Technical report, Department of Computer Science, University of Illinois, February 1985. Reports of the Intelligent Systems Group, ISG 85-11, UIUCDCS-F-85-941.
[18] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.
[19] G. Polya. *How to solve it: A New Aspect of Mathematical Method.* Princeton University Press, 2nd edition, 1957.
[20] V. Rajlich. Incremental change in object-oriented programming. *IEEE Software*, 21(2):62–69, July/August 2004. ISSN:.
[21] Ladislav Samuelis. Synthesis of programs by examples. Technical report, Budapest University of Technology, May 1990. PhD thesis,(in Hungarian).
[22] N. Wilde and B. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

Acknowledgements

The research was supported by the following grants:

- Mathematical Theory of Programming and its Application in the Methods of Stochastic Programming. Scientific grant agency project (VEGA) No. 1/2181/05
- Technologies for Agent-based and Component-based Distributed Systems Lifecycle Support. Scientific grant agency project (VEGA) No. 1/2176/05
- Evaluation of operational parameters in broadband communicational infrastructures: research of supporting platforms. Scientific grant agency project (VEGA) No. 1/2175/05

Department of Computers and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovakia
E-mail address: `Ladislav.Samuelis@tuke.sk, Csaba.Szabo@tuke.sk`