Preface

The 2020 Turkish National Software Engineering Symposium (UYMS) is the fourteenth in the UYMS series. The first UYMS was realized in 2003, was held bi-annually until 2011, and has been continuing annually since then.

The UYMS Steering Committee has established the fundamental aim of UYMS as bringing academicians and industrial practitioners together to share innovations, developments, experiences and results in software engineering. With the ubiquity of software technology, the relevance and appeal of UYMS have increased to encompass all realms, not only of industry, but almost of all of our daily lives.

2020 has brought a completely unforeseen challenge at a global scale: the Covid-19 pandemic. This year, while some important scientific and technological conferences were simply canceled or postponed, we decided, in June 2020, hold UYMS online, at the scheduled dates, over the O'Learn facility of Istanbul Okan University, provided on the Blackboard ® distance learning platform. In hindsight, we can state that the decision was a right one, and we did not face any technical difficulties, thanks to the competence of the support staff and the maturity of the provided facilities.

Unlike the previous 13 UYMS meetings, this year we did not organize the software tools fair, but the technical program and the graduate theses sessions were held, with fewer contributions in comparison to earlier years, but with consistently high quality of contents, as a perusal in the present volume will prove.

Out of a total of 64 submissions, each blind-reviewed by at least three referees, 39 were selected for presentation, and the 38 papers published here, were actually presented. Two broad categories under which the papers may be placed were software engineering technologies, *per se*, and their applications in various domains. There were significant innovative works on software development processes and their adaptation to newer families of artifacts such as microservices, or on applications of established approaches such as machine learning to novel areas such as scientific thinking, or intelligent contracts and cashflow planning. Improving test effectiveness, a traditionally popular topic of software engineering, was, again, not neglected this year. Software engineering applications in such diverse areas as remote application control and on-board satellite computers or game software development were among the elaborated topics.

The Invited Speaker, Prof. Mehmet Akşit, from Twente University, set the stage for UYMS with his plenary talk on the dichotomy of leading to paradigm shifts as opposed to conducting research confined to established problematics. His focus on fundamental software technology issues such as (1) object-oriented vs aspect-oriented programming, (2) language- vs model-driven engineering or (3) problem domain- vs solution domain-driven design was truly enlightening.

The panel discussion in the last session, organized and led by YASAD, the Software Industrialist Association, was devoted to strategic and critical technologies. Professors Albert Levi from Sabancı University and Pınar Karagöz from METU were joined by Hakan Yüksel, AI Manager in ETIYA and Mevlüt Serdar, Founding Partner in KORA Analytics. They all

discussed their work responding to and forming current and future trends in software technologies, with particular emphasis on disruptive innovations on machine learning, big data and information security.

The organizing committee thanks the steering committee for starting and maintaining the respectable UYMS sequence since 2003. We thank the members of all committees who have contributed to the success of this year's UYMS. We are grateful to Prof. Aksit for his visionary talk and to YASAD for the exciting panel discussion. Last but not least, we thank all authors, presenters and participants of UYMS, without whom we would simply have yielded to the pandemic which seems to be the number one adversary of all mankind in 2020.

Conference Co-Chairs

Prof. Dr. Semih Bilgen & Prof. Dr. Bekir Tevfik Akgün

KEYNOTE

Examples of Paradigm Shifts in Software Engineering: Our Research Experience

Mehmet Akşit

TOBB Economics and Technology University

Abstract— Based on our research experience, we analyze three examples of paradigm shifts in software engineering. We consider paradigm shifts as important disruptive changes in the evolution of software technology. Understanding the need for such changes can help the researchers to create more effective and novel solutions. Practitioners can be more conscious about the obstacles in software development and as such prepare themselves accordingly.

Keywords—paradigm shift, software engineering

I. NEED FOR PARADIGM SHIFTS

To understand the need for a paradigm change, there are three indicators: (a) Features = problems: When the features of a paradigm are identified as problems: To address the problems, when workarounds are introduced as good programming practices; (b) Lack of expression power: When certain programming requirements cannot be expressed easily and workarounds are necessary; (c) Anomaly: When the characteristics of language constructs work against the promises;

II. OBJECT-ORIENTED PROGRAMMING (OOP) VS. ASPECT-ORIENTED PROGRAMMING (AOP)

Consider the following claimed features of OOP [1]: An object has a well-defined interface and encapsulates its implementation; concerns can be better represented as objects; message-passing is a proper way of representing interactions among objects; hierarchical organization of objects (classes) using inheritance and aggregation is an effective way of structuring and reusing code.

Several researchers, however, have published that the features of OOP causes problems in certain kinds of applications [2]: Namely, object interfaces may evolve in a dynamic way, crosscutting concerns cannot be directly represented as objects, hierarchical organization is not suitable for crosscutting objects, patterns of message exchanges cannot be conveniently abstracted as objects, reuse of certain code such as synchronization and real-time code cannot be effectively realized using inheritance. The overcome these problems, various AOP languages have been introduced [3].

In the beginning, the AOP paradigm has pulled the attention of a large community. After 2010, however, interest in AOP has gradually declined, due to one or more of the following reasons: (a) Almost all proposals were defined as extensions of an existing language; this increased the language contracts to learn; (b) One language has dominated the area. This brought some risk in practice; (c) Writing an aspect code requires specialization; (d) Many programming environments have started to offer some aspect-oriented support; (e) Opportunistic researchers

have moved to new areas; (f) Programming-in-the-small issues have become less relevant due to large size of code to be written; (g) Although there were some attempts for example in the modularization of emergent behavior [4], new modularization needs in the programming-in-the-large context have not been studied in detail.

III. DOMAIN-SPECIFIC LANGUAGE (DSL) ENGINEERING VS. MODEL-DRIVEN ENGINEERING (MDE)

Specialization of companies in certain product categories has motivated them to develop DSLs [5]. DSL engineering has offered compiler generators, application generators and language design environments. The challenges of DSL engineering are: (a) Every abstraction (or model) may require its own DSL. Coupling mechanisms among DSLs appear to be DSLs too. Handling and coupling multiple DSLs can become problematic; (b) Fixed composition operators defined in the grammar of DSLs (inheritance, aggregation, message passing, etc.) are not expressive for all needs unless extensible languages are defined [6]; (c) Sometimes high-level abstractions required to be decorated with low-level code.

MDE has proposed models and model transformations as the basic features. Models can be considered as DSL's with a more emphasis on graphical representation. Transformations are defined as models as well which can be used to define arbitrary composition operators. Although this provides expressivity, there is a burden of defining transformations. Unless transformations are reused in multiple projects, the extra effort needed cannot be justified. MDE is an active research area with the following challenges: (a) Models are tedious to define. Combining MDE with data-driven engineering to infer models from project data is difficult due to unreliable and incomplete project repositories of companies. Especially concerns (requirements) and design rationale are largely missing; (b) To reduce anomalies, both in DSL engineering and MDE, so-called abstractness and standardization constraints must be well-understood and managed [7]; (c) Not all the aspects of models may be known precisely in their definition time and as such imperfection must be managed [8]; (d) Modelling certain concerns such as emergent behavior can be problematic; (e) To design, assure and trade-off quality attributes of models such as correctness, availability, security, timeliness, etc. are not trivial. Finding novel answers to these challenges may create a new paradigm shift both in DSL engineering and MDE.

IV. PROBLEM-DOMAIN (PD) DRIVEN VS. SOLUTION-DOMAIN (SD) DRIVEN DESIGN

PD and SD are the two important sources of information in any design process. We define PD as the domain of the user of the software system to be designed whereas SD is the domain of the technology where the system is realized. From the early years of software engineering, functional decomposition has been the dominating paradigm in software design.

Functional decomposition has been generally interpreted as defining user requirements as a set of functions, and decomposing functions into smaller functions until each sub-function is well-understood and can be represented as manageable program code. In use-case driven object-oriented methods [9], for example, usage scenarios are taken as the main deriving sources. These approaches can be considered as PD driven methods, where the following obstacles can be observed: (a) PD is defined at the level of "end-user", whereas software is logically realized

at the level of SD. These are in general two separate domains; (b) PD hardly incorporates software realization knowledge. As such its expression power is limited.

An evolution to this approach is to derive software from SD, where the requirements are first decomposed into the problems to be solved, and each problem is then mapped onto the corresponding SD. Software is designed as a synthesis of the selected instances of the SD knowledge [10].

Applying computer-aided-design techniques especially in quality trade-off considerations [11] is becoming a promising approach. Also, semantically integrating computable design rationale with self-adaptive models (i.e. hermeneutics [13]) offers new perspectives. A more comprehensive analysis of the future of software engineering can be found in [13].

REFERENCES

[1] A. Goldberg, and D. Robson, "Smalltalk-80: The Language and its Implementation," Addison-Wesley, 1983.

[2] M. Akşit and L. Bergmans, "Obstacles in Object-oriented Software Development," ACM SIGPLAN Notices, 27 (10), 1992, pp. 341-358.

[3] R. Filman, T. Elrad, S. Clarke and M. Akşit, "Aspect-Oriented Software Development," Addison-Wesley, 2004.

[4] S. Malakuti and M. Akşit, "Emergent Gummy Modules: Modular Representation of Emergent Behavior, "Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE), 2014, pp 15-24.

[5] M. Mernik, J. Heering and A. M. Sloane, "When and How to Develop Domain-Specific Languages," ACM Computing Surveys, 37(4), , 2005, pp. 316–344.

[6] L. Bergmans, W. Havinga and M. Akşit, "First-class Compositions--Defining and Composing Object and Aspect Compositions with First-Cass Pperators, Transactions on Aspect-Oriented Software Development, IX., 2012, pp. 216-267.

[7] M. Akşit, "The 7 C's for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering," Turkish Journal of Electrical Engineering & Computer Sciences, 12 (2), 2004, pp. 61-95.

[8] J. Noppen, P. van den Broek and M. Akşit, "Software Development with Imperfect Information," Soft Computing - A Fusion of Foundations, Methodologies and Applications, 12 (1), 2008, pp. 3-28.

[9] B. Tekinerdogan and M. Akşit, "Classifying and Evaluating Architecture Design Methods, In: Software Architectures and Component Technology. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Dordrecht, 2002, pp. 3-28.

[10] B. Tekinerdogan and M. Akşit, "Synthesis-Based Software Architecture Design," In: Software Architectures and Component Technology. International Series in Engineering and Computer Science. Kluwer Academic Publishers, Dordrecht, 2002, pp. 143-174.

[11] A. de Roo, H. Sözer, L. Bergmans and M. Akşit,"MOO: An Architectural Framework for Runtime Optimization of Multiple System Objectives in Embedded Control Software," Journal of Systems and Software, 86 (10), 2013, pp. 2502-2519.

[12] M. Akşit and S. Malakuti, "Hermeneutics Framework: Integration of Design Rationale and Optimizing Software Modules, "Proceedings of the 14th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS), 2015, pp. 58-62.

[13] M. Akşit, "The Role of Computer Science and Software Technology in Organizing Universities for Industry 4.0 and Beyond," in Federated Conference on Computer Science and Information Systems (FedCSIS), 2018, pp. 5-11.