13th International Conference on Applied Informatics
Eszterházy Károly Catholic University
Eger, Hungary, February 19–21, 2026

Language for Specifying Applications of Design and Architectural Patterns

Bálint Dominik Orosz[a], **Tamás Kozsik**[a]

[a]ELTE Eötvös Loránd University, Budapest, Hungary
balint.orosz@inf.elte.hu, ORCID: 0009-0006-8420-6576
tamas.kozsik@elte.hu, ORCID: 0000-0003-4484-9172

Abstract

With its ever-changing requirements, modifications are inevitable during a software's lifecycle. Studies have identified the structural issues of the implementation as the main cause for the majority of future software changes [2, 4]. For this reason, several automated refactoring approaches have been developed to transform code bases in a structural point of view, so that they would apply possible design and architectural patterns, in hopes of increasing code quality [1, 5–8].

However, we argue that the outputs of these refactorings do not provide satisfactory guarantees for future modifications, as there is still a possibility that developers reintroduce faulty design decisions. In this paper, we propose an approach enabling developers to use design and architectural patterns natively, as parts of the underlying programming language. This way, such patterns can appear more naturally and robustly on the level of language constructs.

The language defined in this paper is domain-specific in the sense that it aims to provide a way to describe software tools used in pipelines or desktop environments (with an optional graphical user interface). However, with minimal generalization it might also be used for the specification of any software or individual code components.

Our approach aims to allow automated refactorings in the long run with a multi-step refactoring process. This study focuses purely on the description and validation of the desired output of this refactoring process in the form of a domain-specific language. Therefore, we introduce the language constructs featuring formal logic and native syntax elements for the description of design and architectural pattern applications. This is followed by the presentation of a case study, showcasing

how an industrial software tool with originally faulty design decisions could be defined alternatively using our methods, avoiding future manual refactorings. To validate our methods, functionality-preservance is ensured by system tests, while an increase in overall code quality is assessed through widespread metric suites [3].

Domain-specific language

The language introduced here provides a way to describe systems and components, the connections between them and their functionalities in formal logic, and – in certain cases – with code written in a programming language. If a functionality cannot be, or is relatively hard to be, formalized, our language offers a way to use incorporated programming language (i.e., C#) codes on its lowest levels, integrating these with the formal descriptions.

Our language offers native language constructs for describing the application of design and architectural patterns. The syntax ensures that components and subcomponents may define, and even force, such patterns within their context, forming a hierarchy of architectural decisions. This provides the opportunity for design and architectural patterns to be combined with, and embedded into, each other. Although the main intention is the architectural description of the business logic layers of an application, the language makes it possible to define a graphical user interface as well, if the applied architectural patterns require so.

Along with the language constructs, automated code generators compile the application specifications into implementations, with enhanced analyzers supporting the compilation process to ensure an output with an increased code quality. These automated analysis rules not only assist the compilation but also notify of design and architectural conformance errors in design time.

Case study

In order to showcase the expressive power and effectiveness of the defined domain-specific language, we present a case study featuring a selected C# software tool with a graphical user interface. The tool was created to be used in an industrial environment and regarding its functionalities, it has been used to modify the inner structure of C# projects and solutions.

However, it was crucial that the tool should be able to handle relatively large code bases as inputs. Although, it was working satisfactorily for smaller inputs, when the size of the input was scaled, the performance of the application deteriorated, which became noticeable at runtime, implying failure to meet certain non-functional requirements. Our manual examinations revealed deeper structural issues.

Nevertheless, it was supposed that the issues of the original code base could be solved with a refactoring process. But instead of applying manual or semi-automatic refactorings, we specified the tool with our domain-specific language, maintaining the original intentions of the software, while laying new structural foundations with explicitly described design and architectural pattern usages.

The validation of our methodologies happened with previously set system tests aiming to ensure functionality-preservance, while the overall code quality has been assessed through widespread metric suites [3]. Our results indicated that the specified functionalities of the tool was satisfactorily maintained, while the overall code quality has shown an increase.

Our goal with this case study was to demonstrate that with the defined domain-specific language underlying structural issues could be eliminated. Also, a correct specification guarantees that the generated codes are well-parameterizable and reusable in other environments as well.

Acknowledgement

SUPPORTED BY THE EKÖP-KDP-24 UNIVERSITY EXCELLENCE SCHOLARSHIP PROGRAM COOPERATIVE DOCTORAL PROGRAM OF THE MINISTRY FOR CULTURE AND INNOVATION FROM THE SOURCE OF THE NATIONAL RESEARCH, DEVELOPMENT AND INNOVATION FUND.

T.K. was supported by project no. TKP2021-NVA-29 under the Ministry of Innovation and Technology of Hungary from the National Research, Development, and Innovation Fund and financed under the TKP2021-NVA funding scheme.

References

[1] A. A. B. Baqais, M. Alshayeb: *Automatic software refactoring: a systematic literature review*, Software Quality Journal 28.2 (2020), pp. 459–502, DOI: 10.1007/s11219-019-09477-y.

[2] K. H. Bennett, V. T. Rajlich: *Software maintenance and evolution: a roadmap*, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, Limerick, Ireland: Association for Computing Machinery, 2000, pp. 73–87, ISBN: 1581132530, DOI: 10.1145/336512.336534.

[3] S. Chidamber, C. Kemerer: *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering 20.6 (June 1994), pp. 476–493, ISSN: 1939-3520, DOI: 10.1109/32.295895.

[4] C. Jaktman, J. Leaney, M. Liu: *Structural Analysis of the Software Architecture — A Maintenance Assessment Case Study*, in: 1999, pp. 455–470, ISBN: 978-1-4757-6536-6, DOI: 10.1007/978-0-387-35563-4_26.

[5] S.-U. Jeon, J.-S. Lee, D.-H. Bae: *An automated refactoring approach to design pattern-based program transformations in Java programs*, in: Ninth Asia-Pacific Software Engineering Conference, 2002. Dec. 2002, pp. 337–345, DOI: 10.1109/APSEC.2002.1183003.

[6] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, E. Arvonio: *Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review*, in: 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), May 2020, pp. 125–136, DOI: 10.1145/3379597.3387475.

[7] H. Y. Shahir, E. Kouroshfar, R. Ramsin: *Using Design Patterns for Refactoring Real-World Models*, in: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, Aug. 2009, pp. 436–441, DOI: 10.1109/SEAA.2009.56.

[8] L. Tokuda, D. Batory: *Evolving Object-Oriented Designs with Refactorings*, Automated Software Engineering 8.1 (2001), pp. 89–120, DOI: 10.1023/A:1008715808855.