Annales Mathematicae et Informaticae
63 (2026) pp. 107–121
DOI: 10.33039/ami.2026.07.003
URL: https://ami.uni-eszterhazy.hu

Case study: Refactoring with design and architectural patterns

Bálint Dominik Orosz, Tamás Kozsik

ELTE Eötvös Loránd University, Budapest, Hungary
balint.orosz@inf.elte.hu, tamas.kozsik@elte.hu

Abstract. Many software quality issues can be attributed to the fact that developers fail to adhere to architectural decisions, or that during the life cycle of the software the original design decisions fade away, and violations of the architecture emerge. In this paper, we present a case study on refactoring with design and architectural patterns. For describing the architecture of software tools, we propose a methodology, which relies on a language that allows software architects to natively express, and enforce the application of design and architectural patterns. The subject of the case study is an industrial software tool, which have been refactored by respecifying it using the introduced language. From this new specification, the transpiler of the language was able to generate a code base with correctly implemented design and architectural patterns, which could be compared to the original. Our methods were validated through expert reviews, and the Chidamber and Kemerer metrics suite. These showed that with our approach the quality of the refactored software tool can be increased, while preserving functionalities, and also satisfying originally neglected non-functional requirements.

Keywords: architecture, design pattern, refactoring, domain-specific language

2020 *Mathematics Subject Classification:* Primary 68N15, 68N19, 68N20; Secondary 68M99

1. Introduction

With its ever-changing requirements, modifications are inevitable during a software's lifecycle. Many studies have identified the structural issues of the implementation as the main cause for the majority of future software changes [2, 10]. For this reason, several automated refactoring approaches have been developed to

Submitted: March 31, 2026
Accepted: July 6, 2026
Published online: July 17, 2026

transform code bases from a structural point of view, so that they would apply possible design and architectural patterns, in hopes of increasing code quality [1, 11, 16, 18, 21]. However, we argue that the outputs of these refactorings do not necessarily provide satisfactory guarantees for the avoidance of future modifications, as there is still a possibility that developers reintroduce faulty design decisions.

In this paper, we present a case study, along with a language supporting explicit design and architectural pattern applications, to present an approach which could be used to improve the quality of industrial code bases. The language allows software architects to better express design decisions, and provides support for advanced code generation with forced application of selected architectural and design patterns. The approach manifests itself in a language combining architecture specification, first-order formal logic based description of functionality, and concrete programming language code. To make our approach more focused and more effective, we restrict ourselves to a subset of potential applications: the approach is designed to support the construction and restructuring of software development tools, ones which are often used in Continuous Integration and Continuous Delivery (CI/CD) pipelines, or desktop applications used by software developers. This is also in line with our primary industrial motivation, namely the improvement of a substantial code base used daily by a software development company.

During our study, since the targeted industrial code base has been implemented in C#, the language proposed in this paper can be seen as a direct extension of C#: an extension which raises C# to the end user level to that of a software architect. However, the provided approach in this paper is independent of any languages, it is fit for most modern object-oriented programming languages also (for instance Java).

The main contribution of this paper is a case study, during which we demonstrate that the use of a specific language describing pattern applications would lead to a robust design and refactoring framework. This would allow, albeit in a restricted domain, software companies to develop better quality code by allowing architects to make architectural and design pattern applications explicit on a high abstraction level, and force implementations to follow requirements imposed by architectural decisions. The applicability of the approach is demonstrated on a case study, evaluated using a set of standard code quality metrics.

Although, the supporting language and its code generation process can be seen as essential accessories of the presented case study, their formal specification is outside the scope of this paper. Rather, its higher level constructs are presented along with key examples, providing a better understanding at this level.

The paper is structured as follows: Section 2 reviews related work, Section 3 presents the titular language, its main concepts, and its layered structure through key examples. Subsection 3.1 explains the supporting code generation process of the language. Section 4 presents the case study on the refactoring of an industrial software tool, the results of which are analyzed and evaluated in Subsection 4.3. Section 5 summarizes the achievements, addresses the limitations of the proposed approach, and discusses future work opportunities.

2. Related work

In the past decades, there have been quite a few studies related to design patterns, their applications and possibilities, and automated code refactoring attempts. Recent systematic literature reviews on these topics [1, 22] have revealed useful insights.

Baqais, Alshayeb published a systematic literature review on automatic software refactorings [1]. They analyzed and compared 41 papers from several aspects. They concluded that 78% of the papers applied refactorings on the code level, while 10% of them on the model level. 5% and 5% used the levels of package and UML, while 2% applied transformations to a graph representing the original software version. Also, for these refactorings, only 29% of the published papers provided some tool for supporting their methodologies, while the majority of them only proposed their techniques. As far as the applied techniques are concerned, most of them used some genetic algorithms or their variants, while applications of hill climbing, simulated annealing, and particle swarm optimizations were also used partially. 54% of these techniques were especially created for automation purposes. 56% of the papers used statistics, as their algorithms are usually non-deterministic.

Tokuda, Batory studied automated code refactorings [21]. Their studies were closely related and narrowed down to object-oriented designs, as they believed that code changes might be automatically applied through visual modifications done on a Graphical User Interface (GUI). This way, architects could visually modify software design and specification through types and their connections.

Jeon, Lee, Bae defined an automated refactoring approach with design pattern-based code transformations for Java applications [11]. Their approach primarily converted Java applications to Prolog facts based on specific rules, and matched these with pre-defined Prolog rules corresponding to each input design pattern. Through these Prolog queries, a number of candidate spots could be identified as potential design pattern application areas in the examined code base.

Shahir, Kouroshfar, Ramsin also investigated the opportunities of applying design patterns to refactor real-world models. In [18], they showcased a case study, in which they refactored an already existing system from their previous studies. It is notable that they already applied design patterns during the initial requirements engineering and analysis phases of the software development lifecycle. Their results showed an increased code cohesion and reduced coupling properties, while the output model became more comprehensible for stakeholders.

Elish, Alshayeb attempted to classify *refactoring to pattern* techniques as their main argument was that choosing between these techniques is not as straightforward as they should be [6]. When it came to the priorization of the techniques, their approach was to measure through different metrics how the quality attributes of the modified system changes when design patterns are introduced. They use the Chidamber and Kemerer metrics suite [3] for internal quality attributes, and the ISO/IEC 9126 standard for external ones. Although, their approach was useful overall, their methodology presumed that design patterns are only introduced to

the system with low-level refactorings. Therefore, their methodology is to be tested against higher level code bases and also higher level refactorings.

Mahouachi, Kessentini, Ghedira noted that in previous works, the detection of design-level problems and definition of possible solutions solving these problems were sharply separated. Therefore, they experimented with handling these two steps as one, and accordingly, they specified a genetic algorithm [13]. The input of this algorithm was a set of examples, containing faulty code segments and manually created possible solutions for their corrections, and also the set of possible refactoring rules was provided as input. The algorithm then applied the available refactoring rules to the examples, and then the output was compared to the expected manual solutions by a fitness function.

Zeinab et al. investigated how design patterns and anti-patterns mutate with the ever-changing development requirements, and how the fault-proneness properties of these designs change with modifications [23]. They specifically examined whether the presumed mutations across design patterns and anti-patterns are possible, and also the possibility of these mutations (for this reason, they used Markov models).

A noteworthy example is the use of aspect-oriented programming through the AspectJ extension of the Java language in the works of Hannemann, Kiczales. In [8], they used aspects to extract the repetitive implementation of certain design patterns from the source code. This way, design pattern implementations were separated from their specific applications.

Another notable example can be found in the works of Dévai et al., in which they introduce a system for executable UML specifications [4]. Their system introduces special description extensions to the UML set of tools, which later can be used to specify applications exclusively on the design level. This approach tries to address and tackle the scalability issues of UML, which makes its use in an industrial environment difficult.

In our previous works [15], we have already experimented with error detection related to design and architectural patterns, and the conformance of code bases to them.

Based on the previous related works, a similarity is that our approach also aims to eliminate boilerplate code to reduce the possibility of errors, by using patterns as templates, with automated code generation.

A notable difference is that most works have a clear distinction between design and architectural patterns, whereas we handle architectural patterns as design patterns applied to higher level design constructs. Also, most works mainly focus on specific examples of design patterns, rather than covering the whole spectrum of design and architectural patterns. Furthermore, most works use large open-source code bases for validation, while our primary target is a specific software tool, an independent subsystem of a larger industrial code base.

Therefore, during our studies, we focused on these deficiencies, and integrated these insights into our approach. Regarding the validation techniques, along with expert reviews, we also use the Chidamber and Kemerer metrics suite, similarly to other works.

3. LADY: Language for architectural design

LADY is a language for describing software architecture, in particular the architecture of software tools. Hence, the language is domain-specific in the sense that it is best suited to formulate the design of command-line applications, which can be used in CI/CD pipelines, or desktop applications with a graphical user interface – although it can be used to describe general standalone software components as well.

LADY especially targets software architects (with a more senior understanding regarding design decisions) as end users, and enables them to specify software tools efficiently, with a clear description of design decisions. It allows architects to natively express applications of patterns relevant in this domain, such as Model-View-ViewModel, or pipe-and-filter. Furthermore, LADY prohibits certain language features if they interfere with the application of the applied patterns, significantly complicating developers to accidentally deviate from the original design, although it remains technically possible to weaken architectural guarantees through plugin codes.

According to these principles, LADY has two sets of language constructs, the first being the general programming constructs suitable to define software tools, the other being the set of pattern application constructs. Naturally, elements of the former set are used to determine the functionalities of the specified application, while elements of the latter set are going to serve as higher level structuring guidelines, by influencing the architectural setup of the specified application.

In this regard, LADY should not be considered merely as an architecture description language, as its main purpose is not only to describe and govern architectural setups (it also handles lower-level constructs, such as design patterns). Rather it is closer to being a model-driven code generation language, as it also aims to guide the whole design process. Although, it is notable that the language is planned to be part of an automatic refactoring framework in future works.

3.1. Code generation process

Instead of a traditional compiler, LADY is accompanied with a code-to-code compiler, a transpiler, which outputs C# code (instead of executable binaries or some intermediary virtual machine code). This is especially useful when the language is used in refactoring (this being one of the primary motivations behind it), because in this way it becomes possible to observe and compare the refactored output code to the original. Naturally, from this setup, the compilation of executable binaries does not require further modification of the methodology.

The code generation in the transpiler ensures that the output code meets the general requirements for the application of the supported patterns. Moreover, when the currently used patterns do not prevent it (or other factors do not interfere with it), the transpiler will generate concurrently executing code by default. For this purpose, certain static code analysis rules (like the immutability check of user-defined types or the inference-check of sequential expressions) are applied by

the transpiler to collect necessary information, which influences the transpilation process and its output.

Design patterns are implemented by the transpiler as expected [9], the only notable exception being the *singleton* design pattern, which is often regarded as an anti-pattern [17]. In LADY, if a type is marked singleton, then the type is not going to know of itself whether it is a singleton type, rather, a dependency injection framework is used in the context of the whole application, and the annotated type is registered into it as a singleton implementation. This results in the type effectively being a singleton throughout the whole life-cycle of the application, while still maintaining code quality standards and best practice principles.

3.2. Constructs for describing software tools

At the architecture design level, the main emphasis is on the specification of functionalities, and the placement of these functionalities in special structures (as provided by the first set of constructs of LADY), which are going to be guided and safeguarded by the other set of language constructs: the architectural and design patterns.

The structural constructs of LADY are designed with a layered approach. The bottom layer defines the layer of object types. LADY provides two kinds of such types: *entities* and *records*. The former is mainly used for persistence purposes (like data transfer objects—DTOs), while the latter is closer to the traditional concept of objects (i.e., types with functionalities).

The next layer is concerned with the specification of functionalities in so-called *actions*. Ideally, these *actions* are described with formal logic. For practical reasons, if something cannot be, or is too complex to be, formalized, C# code can also be used.

One layer higher, types and *actions* are encapsulated by *components*. These are the main guiding structures within the language, and an application is basically built on their collaboration.

In the top layer, the structure of the main *application* needs to be defined. An application can either be a console or a desktop application. Within these, the possible *commands* of the *application* need to be placed, consisting of their input parameters and validators (according to the usual best practices). Also, each *command* has to be connected to some functionality through an *action* of a *component*.

Key examples

Listing 1. A main entry point for a console application with a specific command.

```
1 ConsoleApplication
2 {
3     commands: [
4         {
```

```
            name: "analyzer",
            input: {
                scopePath: {
                    type: String,
                    validators: [REQUIRED]
                }
            }
            action: NamespaceAnomalyAnalyzer.Main(scopePath)
        }
    ]
}
```

Listing 1 shows an example for a possible main entry point of a console application with a specific command. This code will ensure that during the transpilation process, a C# console application will be generated, with only one possible invokable command, with the name *analyzer*, which has a required *String* input parameter. Besides, the action of this command is referring to the *Main* action of the *NamespaceAnomalyAnalyzer* component. This example gives an abstract specification of this top-level application description, which is free of any concrete implementation details.

Listing 2. A component description containing an action.

```
component NamespaceAnomalyAnalyzer
{
    record AnalyzerRecommendation
    {
        Path: String;
        Namespace: String;
        SupposedPath: String;
        SupposedNamespace: String;
    }

    action Main
    {
        input: { scopePath: String; }
        output: { result: Sequence<AnalyzerRecommendation>; }

        logic: {
            projectFiles := <code lang="csharp">...</code>
            AND
            ...
            AND
            result := map ...
        }
    }
}
```

Listing 2 depicts a *component* and an *action* specification. Notable cases include the possibility to define *records* within the context of a *component* (and therefore making it component-specific), the abandonment of concrete collection types and the use of an abstract *Sequence* type (to make sure that the most suitable alternative is used according to the specific example), and the possibility to use mixed specification styles (formal and plugin code) during the specification of an *action*'s functionality.

3.3. Pattern-related language constructs

LADY provides a way to describe the application of architectural and design patterns natively. These patterns are present as higher organizing principles, and they ensure a more robust architectural setup within the specification.

Regarding the pattern-related language constructs, the following architectural patterns [20] have been selected to be part of the language: the *Model-View* (MV), the *Model-View-ViewModel* (MVVM), and the *pipe-and-filter* architectures. Besides these, LADY supports the application of a number of design patterns [7], namely: *factory*, *builder*, *singleton*, *proxy*, *composite*, *iterator*, *mediator*, *command*, and *chain-of-responsibility*.

Pattern applications in the language work with an underlying *context* definition, which is mostly not noted explicitly in the source code. That is, pattern applications have a limited and well-defined boundary, in which their applications take effect and ensure their purposes. For instance, an MV or MVVM architecture might determine the whole setup of an application specification, while they might also be used within the context of a single component of the application. Notably, the *pipe-and-filter* architecture might be used with multiple contexts, as its use on the level of higher-level commands (for a whole command) and actions (for a smaller set of logical steps) also seem rational and justified.

Key examples

Listing 3. An action with a pipe-and-filter modifier causing the action's functionality to execute its steps in a sequential manner.

```
pipe-and-filter action Main
{
    input: { scopePath: String; }
    output: { result: Sequence<AnalyzerRecommendation>; }

    logic: {
        projectFiles :=
            <code lang="csharp">
                Directory.GetFiles( scopePath, "*.csproj",
                                    SearchOption.AllDirectories )
            </code>
        AND
        projects := map projectFiles (projectFile) ->
                      create ExtendedProject(scopePath, projectFile)
        AND
        sourceFiles := flatten projects (project) ->
                         project.SourceFiles
        AND
        result := forall sf in sourceFiles:
                    ((sf.Namespace != sf.SupposedNamespace)
                    OR
                    (sf.Path != sf.SupposedPath))
    }
}
```

Listing 3 gives an example of an *action* with the *pipe-and-filter* modifier, forcing the underlying steps to be evaluated sequentially, one after another, excluding any concurrent execution opportunities. Moreover, according to the rules of this architectural pattern, the steps (separated by the conjunctions) are going to be limited to use only the output result of the immediately preceding step, instead of any results of any previous steps. This example also illustrates that the definition of an action can be provided as a mixture of logical descriptions of functionalities and C# code fragments.

Listing 4. *Composite* design pattern application for representing the setup of a C# solution.

```
entity Solution { Path: String; }

entity Project { Path: String; }

entity SourceFile
{
    Path: String;

    lazy Namespace: String
    {
        <code lang="csharp">...</code>
    }
}

composite
{
    parts: [Solution, Project, SourceFile],
    type: 1-N
}
```

Listing 4 showcases the use of the *composite* design pattern. The example follows the abstract description of a C# solution, which contains projects, with their contained source files. This is the classical setup for a composite design pattern application with a one-to-many type, causing the transpiler to generate the required navigational properties to the entities. The example also features the use of the *lazy* keyword, which delays the evaluation of the field, while it can also be seen that the functionality of a field might be specified through the use of C# plugin codes as well.

Listing 5. An example for the *factory* design pattern application.

```
factory component for Project
{
    action Load
    {
        input: { scope: String; path: String; }
        logic: { ... }
    }
}
```

Listing 5 features an example for the use of the *factory* design pattern. The *factory* keyword introducing this declaration will ensure that within the context of

the whole application, the *Project* type can only be instantiated through actions of this component.

4. Case study

To demonstrate the capabilities and expressive power of LADY, we summarize the lessons learned during the specification and recreation of an already existing software tool, a desktop application which is in use in an industrial environment. Regarding its functionalities, it aims to modify the structure of C# projects and solutions, adjusting the namespaces of C# source files to the name of their containing folders – and vice versa.

Although, other software tools and plugins performing the same operations had already existed (except for the manual restructuring of projects based on custom rulesets) [5, 12, 14], the creation of this tool was justified, as the sheer size of the input C# solutions requires special processing algorithms not present in other, commonly used tools and plugins. However, the refactoring of the tool became necessary finally, since its original implementation was not capable of handling sufficiently large input data – the tool was violating critical non-functional requirements related to input size, runtime, and memory-handling.

4.1. Research questions

As part of the preparation for the case study, the following research questions have been formulated:

RQ1 Is LADY expressive enough to describe the architecture of an industrial software tool?

RQ2 Can quality improvements be seen after the respecification process?

RQ3 What is the state of non-functional requirements after the respecification process?

4.2. Respecification process

Using LADY, we specified, and – relying on the transpiler – refactored the selected industrial application. The new specification includes a more precise, formal logic based description of the tool's functionalities, as well as the enforcement of the desired design and architectural patterns. After the process, the new implementation of the tool, which was generated from the improved specification was able to fulfill all functional and non-functional requirements.

During the respecification process, most of the previous specification was omitted and rewritten in the form of formal logic rules. The original specification was reverse engineered from the original code base, and requirements were collected from preceding design documents.

Although, special C# codes were necessarily added in specific cases, like OS-level file and directory handling, and unique file modification operations. However, this only happened in a negligible number of cases. Naturally, these plugin C# codes were explicitly made part of the output solution, while the remaining specification was transpiled.

4.3. Results

The output of our case study, the refactored code base was validated through three steps.

At first, system tests were run to ensure the preservation of functionalities, as naturally required for any sound refactoring process. In this case, the use of unit and component tests were not an option, as during the specification and code generation processes, the signature of the methods are likely to change, which invalidates any previous unit tests.

Testing was followed by a code review performed by software architects, the owners of the original code base. The new code base was especially reviewed for clean code measurements, and it was inspected whether the implementation conforms to the applied design and architectural patterns.

Finally, the Chidamber and Kemerer metrics suite was selected as a ground to compare the original and refactored code bases on a metric-level.

Table 1. Chidamber and Kemerer metrics suite results for the original and the refactored code bases.

Metric	Original code base	Refactored code base
WMC	5.875	4.074
DIT	1	1.142
NOC	0	0
CBO	1.75	0.481
RFC	9.75	7.962
LCOM_1	-	0.62

Table 1 shows the results of the selected metrics. Regarding this kind of validation, a clear improvement can be seen in half of the metrics. Namely, the WMC (Weighted Methods per Class), CBO (Coupling between Object Classes), and RFC (Response for a Class) metrics have shown a decrease in their values, which indicates a positive change.

Regarding the remaining metrics, the following observations can be made. The NOC (Number of Children) metric shows the number of immediate subclasses of a class. Since neither the original, nor the refactored code base had any subclass behavior besides the built-in inheritance from the *Object* class, this metric is not relevant in our case.

As far as the DIT (Depth of Inheritance Tree) metric is concerned, an increase can be seen in the measured value. This is due to the fact that each of the generated service classes received an extra implemented interface during our transpilation process, but these corresponding interfaces are not present in the original code base. The DIT value for the original code base is 1, indicating that all the classes have the *Object* class as their base class, while the DIT value for the new code base is slightly increased, as some of the classes now possess interface base-types as well. Since from an object-oriented perspective this can be regarded as a sign of increased code quality, the slight increase in DIT is not just acceptable, but even desirable.

A similar reasoning applies for the last metric. It can be seen that the LCOM (Lack of Cohesion of Methods) metric could not be measured in the original code base, due to its overuse of static classes and methods. LCOM became meaningful in the new code base, and again, the measured value is justified and accepted, since from OOP point of view the transition from static methods to instance methods can be observed as quality improvement.

Finally, it is worth considering threshold values for the metrics published in previous works, such as the ones available for CBO, RFC, and WMC [19]. We can conclude that our related values for all of those metrics are well in the desired ranges.

4.4. Non-functional requirements

Furthermore, the case of non-functional requirements must also be examined. Before the respecification process, the runtime of the original tool could be measured in hours, while after changing the underlying architectural setup, this runtime on the same inputs could be measured in just a couple of minutes, which indicates a large performance optimization. Similarly, the originally present memory-handling problems disappeared after the process.

The former can be attributed to the analysis related to immutable types and the concurrent code generation, while the latter to the specific control flow inherited from the use of the *pipe-and-filter* architecture, which results in the dropping of unnecessary objects of previous steps from memory.

5. Conclusion and discussion

The results presented in this paper contribute to the analysis and refactoring of large-scale industrial code bases. As our initial examinations revealed, many problems in code bases are related to some structural issues which might be solved by introducing some rigorous implementations of architectural and design patterns. In the existing literature, we have seen that many works are focusing on the violation detection of pattern implementations. Some other relevant works propose annotations to eliminate the repetitive, boilerplate code of the patterns. We have concluded that the root cause of our problems can be traced back to the fact

that programming languages often do not provide sufficient support for the native description of patterns at the location of their applications in source codes.

Therefore, we have defined a domain-specific language, characterised by the fact that it allows architectural and design pattern applications on the level of native language elements. Also, through a well-controlled code generation process, we are able to ensure that the code output by the transpiler of the language contains a correct and efficient implementation of these patterns.

To validate our methods, we have performed a case study, during which we have specified and refactored an already existing industrial software tool. As a result, code quality issues of the original implementation were eliminated, and the problem of not meeting non-functional requirements was also solved. Comparing the original and refactored code bases through the Chidamber and Kemerer metrics suite, our method proved to produce a clear improvement in half of the metrics. Furthermore, even when improvements were not shown directly by the metric values, enhancements on the code quality were justified. This shows that our language is suitable for describing software architecture, and can also serve as a fundamental tool for a large-scale automated refactoring process.

It is notable, however, that the mentioned metrics suite is mainly for capturing object-oriented and structural properties, while architecture-based improvements were mainly assessed through expert reviews by code architects. Moreover, the improvement of the architectural properties can be indirectly discovered by the satisfaction of non-functional requirements, which were previously neglected.

To address the research questions, LADY has proven to be expressive enough to describe the design structures of the respecified industrial software tool (*RQ1*). As explained above, clear code quality improvements could be seen after the respecification process: improvements on structure (*RQ2*) and on the side of non-functional requirements also (*RQ3*).

Regarding future research possibilities, we plan to extend the scope of our language from specifying software tools to supporting the architectural description of general software applications. Ideally, by being able to handle general software applications, future research must focus on strengthening the results of this paper with a broader set of open-source applications. Besides, the formal specification of the language is planned to be published in a standalone paper as well.

Another research direction can improve the scalability of the approach by the addition of "outer components": this would allow code bases to be refactored by smaller units (by components or a set of components), while keeping existing component connections intact with outer components not being refactored. Also, the mentioned immutability checks for user-defined types, and the inference checks for sequential expressions will be discussed in another paper.

Additionally, as we agree that the plugin C# codes may weaken currently existing architectural guarantees, we are planning on extending the analysis capabilities of the transpiler architecture to pre-analyse these codes during the compilation process.

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] G. Dévai, T. Gregorics, M. Tóth, D. Asztalos, D. J. Németh, G. F. Kovács, B. Németh, Z. Gera, A. Dobreff, B. Gregorics, A. Nagy, M. Budai, Z. Kulik, K. Kanyó: *txtUML*, in: Proceedings of the MoDELS 2016 Demo and Poster Sessions (MoDELS 2016), Saint-Malo, France, 2016, pp. 8–15.

[5] DevExpress: *Rename Namespace to Match Folder Structure*, https://docs.devexpress.com/CodeRushForRoslyn/117359/refactoring-assistance/rename-namespace-to-match-folder-structure, 2021.

[6] K. Elish, M. Alshayeb: *Using Software Quality Attributes to Classify Refactoring to Patterns*, Journal of Software 7 (Feb. 2012), DOI: 10.4304/jsw.7.2.408-419.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns: elements of reusable object-oriented software*, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0201633612.

[8] J. Hannemann, G. Kiczales: *Design pattern implementation in Java and aspectJ*, ACM SIGPLAN Notices 37.11 (Nov. 2002), pp. 161–173, ISSN: 0362-1340, DOI: 10.1145/583854.582436, URL: https://doi.org/10.1145/583854.582436.

[9] J. Hunt: *Gang of Four Design Patterns*, in: Scala Design Patterns: Patterns for Practical Reuse and Design, Cham: Springer International Publishing, 2013, pp. 135–, ISBN: 978-3-319-02192-8, DOI: 10.1007/978-3-319-02192-8_16, URL: https://doi.org/10.1007/978-3-319-02192-8_16.

[10] C. B. Jaktman, J. Leaney, M. Liu: *Structural Analysis of the Software Architecture — A Maintenance Assessment Case Study*, in: Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22–24 February 1999, San Antonio, Texas, USA, ed. by P. Donohoe, Boston, MA: Springer US, 1999, pp. 455–470, ISBN: 978-0-387-35563-4, DOI: 10.1007/978-0-387-35563-4_26, URL: https://doi.org/10.1007/978-0-387-35563-4_26.

[11] 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.

[12] JetBrains: *Adjust Namespaces*, https://www.jetbrains.com/help/resharper/Refactorings__Adjust_Namespaces.html, 2025.

[13] R. Mahouachi, M. Kessentini, K. Ghedira: *A New Design Defects Classification: Marrying Detection and Correction*, in: Fundamental Approaches to Software Engineering, ed. by J. de Lara, A. Zisman, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 455–470, isbn: 978-3-642-28872-2.

[14] Microsoft: *Sync Namespace and Folder Name*, https://learn.microsoft.com/en-us/visualstudio/ide/reference/sync-namespace-and-folder-name, 2024.

[15] B. D. Orosz, J. Szücs, M. Cserép: *Architecture-Aware Static Analysis and Violation Detection of C# Student Submissions*, Computers 15.7 (2026), issn: 2073-431X, doi: 10.3390/computers15070404, url: https://www.mdpi.com/2073-431X/15/7/404.

[16] 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.

[17] M. Radford: *SINGLETON – the Anti-Pattern!*, ACCU Overload 57 (2003), url: https://accu.org/journals/overload/11/57/radford_337.

[18] 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.

[19] R. Shatnawi: *A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems*, IEEE Trans. Softw. Eng. 36.2 (Mar. 2010), pp. 216–225.

[20] I. Sommerville: *Software Engineering*, 10th, Pearson, 2015, isbn: 0133943038.

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

[22] F. Wedyan, S. Abufakher: *Impact of design patterns on software quality: a systematic literature review*, IET Software 14.1 (2020), pp. 1–17, doi: https://doi.org/10.1049/iet-sen.2018.5446.

[23] Zeinab, Kermansaravi, M. S. Rahman, F. Khomh, F. Jaafar, Y.-G. Gueheneuc: *Investigating Design Anti-pattern and Design Pattern Mutations and Their Change- and Fault-proneness*, 2021, arXiv: 2104.00058 [cs.SE], url: https://arxiv.org/abs/2104.00058.