

## Article

# Architecture-Aware Static Analysis and Violation Detection of C# Student Submissions

Bálint Dominik Orosz <sup>1</sup>, Judit Szücs <sup>2</sup> and Máté Cserép <sup>3,\*</sup>

<sup>1</sup> Department of Programming Languages and Compilers, Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter stny. 1/C, H-1117 Budapest, Hungary; balint.orosz@inf.elte.hu

<sup>2</sup> Department of Information Technology and Its Applications, Faculty of Information Technology, University of Pannonia, Gasparich M. utca 18/A, H-8900 Zalaegerszeg, Hungary; szucs.judit@zek.uni-pannon.hu

<sup>3</sup> Department of Software Technology and Methodology, Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter stny. 1/C, H-1117 Budapest, Hungary

\* Correspondence: mcserep@inf.elte.hu

## Abstract

Static analysis of student programming submissions has proven a useful supplement to manual evaluation in university courses, but existing approaches focus on local code-quality issues and rarely check higher-level design decisions such as architectural conformance. We propose an architecture-aware static-analysis methodology for student submissions written in C# and structured according to the Model-View (MV) or Model-View-ViewModel (MVVM) architectures. A deterministic clustering algorithm assigns user-defined types to architectural layers by combining heuristic rules derived from SDK conventions with course-specific information, and our 10 proposed violation checks—covering layer-dependency rules, encapsulation, event handling, and dependence on concretions—are evaluated on the recovered layer structure. We implemented the methodology as an open-source analyzer integrated with an automated submission-evaluation system used in a university course focused on event-driven applications, and evaluated it on 947 submissions containing 13,126 user-defined types from past semesters. The analyzer assigned more than 98% of types to their correct layer and surfaced more than 6000 architectural and design issues. The results show that architecture-aware static analysis is a viable complement to manual grading and produces actionable feedback for both students and lecturers.

**Keywords:** static code analysis; C#; MV; MVVM; architecture violation; design pattern violation; higher education



Academic Editor: Paolo Bellavista

Received: 21 May 2026

Revised: 17 June 2026

Accepted: 23 June 2026

Published: 25 June 2026

**Copyright:** © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

## 1. Introduction

The number of university students enrolling in computer science programs is rapidly increasing each year [1,2]. With this number, the number of student submissions to university course tasks also escalates. This results in the multiplication of lecturer workload related to the evaluation of these submissions, and the increased workload pressures the depth of feedback that lecturers can provide on each submission. Therefore, as a consequence of their increased workload, lecturers might tend to pass over small but important issues, so it is possible that students may not get relevant feedback for their solutions and believe that their knowledge in the topic is flawless, which in the future might present problems to enterprises and thus to industry itself.

In order to present a solution to the previously described problems, universities have started to apply static code analysis on student submissions in the hope of gaining

valuable information about regular flaws [3]. Indeed, automated assessment of programming exercises—with an increasing focus on the quality of the feedback delivered to students—has grown into an established and continuously expanding research area [4]. It was also used to discover the most common issues related to specific university courses. This method can help lecturers to identify programming problems where most students fail and also determine which areas would be most useful to focus on during practice.

While the applied static code analysis on student submissions in [3] has already proven to be a valuable approach during evaluation, we argue that it is not satisfactory enough, as it lacks support for the discovery of higher-level design decisions (such as architectural design) and their violations. Fortunately, student submission evaluator systems are already in use with built-in support for automated building and testing processes. It can be considered a foundation on which the development of higher-level design decision checkers and validators becomes possible, convenient and most reasonable.

Tools for the recovery of architectural layers of software and for violation detection have already been in existence, so the goal of this research was to examine how these approaches can be applied and/or fine-tuned regarding the assignments of the university course of Event-driven applications at Eötvös Loránd University Faculty of Informatics (ELTE FI). Therefore, the creation of a tool performing architectural analysis (and other kinds of checks related to the course also) was from the perspective of the research, and its validation with the evaluation of student submissions from past semesters.

The contributions of this paper are threefold: (i) a two-phase architecture-aware analysis workflow for .NET solutions that overcomes a limitation of the default Roslyn analyzer pipeline; (ii) a deterministic, education-oriented type-to-layer clustering method that combines SDK-derived heuristics with our proposed violation checks, and explicitly surfaces ambiguous cases to the user and (iii) an open-source implementation, integrated with an automated submission-evaluation system, and evaluated on 947 student submissions containing 13,126 user-defined types.

Section 2 presents related work of state-of-the-art architectural layer clustering and violation detection techniques, along with the approaches aiming to detect architectural and design pattern violations in software. The section also introduces the related university course, its targeted assignments and possible static analyzer options, determines the goals of research, and poses the research questions as well. Section 3 presents the proposed methodology in the form of an architectural analysis workflow and the clustering technique it uses, alongside the defined violation checks. Section 4 explains the evaluation of student submissions in detail, while the performance and accuracy of the proposed clustering method are also examined. Finally, Section 5 summarizes the achievements of the research, while discussing future research opportunities.

## 2. Background and Related Work

### 2.1. Structure of the Related Course

The intention of this paper is to examine the possibilities of checking the architectural conformance of student submissions to Model-View (MV) and Model-View-ViewModel (MVVM) architectures. The examined student solutions were submitted to assignments of the university course of Event-driven applications at Eötvös Loránd University Faculty of Informatics (ELTE FI).

As its name implies, the course is dedicated to teaching the methodologies of creating event-driven applications, mainly applications with a Graphical User Interface (GUI). The course is centered around the C# language, and during development, students get acquainted with the Windows Forms (WinForms) [5] and Windows Presentation Foundation

(WPF) [6] frameworks. C# is an effective and reasonable choice of language, as it supports the use of events on a language level without the use of any external libraries [7].

Assignments related to the course are usually simple games which can be played on game boards or game board-like structures. (Games that rely on a game board which can be projected onto a coordinate system.) Some examples are Sudoku, Minesweeper, Whac-A-Mole, Snake, Tron, Reversi, Nine men's morris, Asteroids and acceleration games in general (non-exhaustive list). Some of the assignments often require the use of file operations such as saving and loading game states; therefore, the existence of a *Persistence* layer.

Generally, assignments must follow the rules of architectural design presented during the course. As part of that, students are encouraged to separate C# projects in their solutions according to the architectural representation of the assignment. However, the use of the previous approach is not obligatory. Instead, students are warned to separate the architectural layers on the level of namespaces at least, and place the related types into the same namespace. However, the name of the namespace should still resemble the name of the corresponding architectural layer. Also, as the name of the course (Event-driven applications) implies, another demand of students is the use of events during the specification and implementation of the assignments.

## 2.2. Task Management System (TMS)

Task Management System (TMS) (<https://tms-elte.gitlab.io>, accessed on 20 May 2026) is an open-source, web-based assignment management, automated evaluator, plagiarism detection and examination software for computer science higher education developed at ELTE FI, where it is also used for managing tasks of several courses, including Event-driven applications. Comparable automated programming assessment systems are in widespread use at other universities—such as the Edgar system at the University of Zagreb [8]—which share a similar role of providing scalable, immediate feedback on student programming submissions.

Initially, one of the main advantages of TMS was that it provided an automated building and testing opportunity for the uploaded submissions. This was later further enhanced with the purpose of evaluating student submissions with static code analysis. Therefore, a workflow was implemented in TMS that supports the operation of various static analysis tools on student submissions [3].

## 2.3. Roslynator

Roslynator (<https://josefpihrt.github.io/docs/roslynator>, accessed on 20 May 2026) is a .NET analyzer tool developed as an open-source project. It contains a large number of Roslyn analyzers, code fixes and refactoring definitions. However, it also offers a tool that is able to run the static analysis of a given project or solution, while offering outstanding configurability. The reason why Roslynator is so useful is that it can be extended with third-party Roslyn analyzers at runtime. Therefore, its pure analyzing features can be leveraged.

The tool is already integrated with TMS, which offers a number of configurations to analyze submissions with Roslynator, using Roslyn analyzers contained within the Microsoft.CodeAnalysis.NetAnalyzers, SonarAnalyzer.CSharp and Roslynator.Analyzers NuGet-packages. The report converter tool of CodeChecker also supports the XML output of Roslynator [3].

## Roslyn(ator) Analysis Workflow

The default workflow of Roslynator uses Roslyn APIs, which means that the analysis will happen separately for compilation units and, therefore, for projects. However, when it comes to solutions, projects in a solution are analyzed one by one, based on the dependency graph of the analyzed projects. In this graph, vertices represent the projects of the solution,

and an edge  $(a, b)$  represents that project  $a$  depends on project  $b$ . After its creation, a topological sorting of the vertices of the graph can be traversed and the projects inside the solution can be analyzed in that order. It is crucial, considering that projects need to be compiled in a specific order, and that order is determined by their dependencies.

#### 2.4. Related Research

Static analysis is one of the major families of techniques used in automated code assessment for education, alongside dynamic analysis and hybrid approaches [9]. Within this broader landscape, architectural conformance checking forms a specialized line of work that targets higher-level design properties rather than purely local code quality.

In order to be able to state validation results regarding a software's architectural conformance, the validation methods ought to know the architectural setup of the software components beforehand. That is, the input of the rules checking for architectural violations must contain some information about the clustering of software components into pairwise disjoint sets, where these sets represent the layers of the examined architecture.

For this reason, clustering architectural layers can be considered the cornerstone and first step towards architectural violation detection. For quite some time, different approaches have been introduced that aim to solve this non-trivial problem. These approaches can be considered manual, automatic, or semiautomatic based on the amount of user interference needed [10]. However, detecting architectural violations has also been conducted by other means.

Wongtanuwat and Senivongse used a set of regular expressions to extract data from the source code of iOS applications written in Objective-C [11]. Their approach simply sorted classes into *Model*, *View* and *ViewModel* architectural layers based on their names and the inheritance information related to them. However, if the end user of their tool was not satisfied with the result of this clustering, the tool provided a way of manually setting the architectural layer of a class. They also categorized what they thought from their perspective as violations of the MVVM architecture and sorted these into three groups. The first group had only one rule, which required the presence of the three architectural layers; the second group concentrated on the data flow between layers and the individual tasks of each layer, such as the *Model* managing data or the *View* interacting with the user and the third group of rules focused on the relationship between layers and, more practically, the dependencies between them.

Aljamea and Alkandari introduced their validation model for iOS applications using MVC and MVVM architectures [12]. Their violation detection covered two checks. At first, they used regular expressions to validate whether a class in the Controller layer is massive or not—the decisive condition was based on a usual convention. After that, their method sorted types into layers based on their base class. The base classes anticipating architectural layers were determined from the special task-specific classes of the SDK. Some architectural layers can be specified with the previous approach; however, they argued that there is no definite way to assign types to the *Model* and *ViewModel* layers; therefore, they relied on additional user information regarding these layers. After the layer of all types was either identified or set, they used regular expressions once more to retrieve information about the dependencies between types. Finally, from the collected data, a dependency graph description was created with the DOT language [13] to enable visual representation. The created graph emphasized invalid dependencies that violate the checked architecture.

Hasan examined how the design pattern of an object-oriented Java application could be recovered using data extracted from source code [14]. They created a tool that outputs the visual representation of the relationships between classes in order to provide insight into the application structure. From that, they argued that the level of coupling between classes

can be discovered, which would help future development and design decisions. At first, they grouped the relationships between objects and classes into two categories: Operation-Operation interactions and Class-Class interactions. The former connects two classes if in the first class there is an operation that either has a parameter of the other class or has the return type of the other class. The latter connects two classes if an instance of the first class is declared inside the other class or the two classes are connected through inheritance.

Sarkar et al. describe so-called “Layering principles” that are considered good practices [15], these are the back-call principle, the skip-call principle and the cyclic dependency principle. The back-call principle states that an upper layer should depend on a lower layer, while the lower layer cannot depend on the upper layer. The skip-call principle describes that an upper layer should only depend on the lower layer immediately below it. The cyclic dependency principle states that violations of the back-call principle would result in cyclic dependency relations between layers, meaning that cycles would exist in the dependency graph. Of course, this should be avoided as such a set of connections would make the architecture monolithic. Although they mentioned Architectural Description Languages (ADLs) [16], they argued that these lack certain aspects of describing architectural layering principles, and instead they used a custom XML file describing architectural layers, the modules they contain, the ordering of layers and also some minor violations that might be useful to allow. In order to discover violations of the cyclic dependency principle, they created a strongly connected component graph from the originally recovered module dependency graph.

Cai et al. used design structure matrices (DSMs) [17] and design rule hierarchies (DRHs) [18] to examine the conformance of student submissions to the expected design patterns of university assignments [19]. Their method was to produce DSMs created from the connection of types in the student submissions. These matrices were then clustered into DRHs that represent a layered structure, which provides the dependency links between the sorted layers. These DRHs could then be used to analyze the conformance of design patterns present in submissions to the expected results (from which the same type of matrices were produced).

Scanniello et al. applied the Kleinberg algorithm [20] on industrial applications as well as on applications created by university students written in Java [10]. Their approach included the extraction of type dependencies from source code using static code analysis regarding the inheritance, aggregation, and association used between types. This ultimately resulted in the creation of a graph in which vertices represent types and edges represent the connection between those types. After that, the Kleinberg algorithm could be used to assign each node an authority and hub value that corresponds to the indegree and outdegree of the nodes. When these values are calculated and normalized, nodes of the graph can be sorted into categories. Types that are not used by other types belong to the top layer, and types that do not use other types belong to the bottom layer (If a type is not used by any other type and does not use any other type, then it does not have any connection with other types, and therefore, it is sorted into another layer). All other types belong to the middle layer. The advantage of this approach is that it may be applied multiple times for the middle layer as a subgraph of the previous graph if the architecture describes more than three layers.

Constantinou et al. used an almost purely graph-based approach to cluster types into architectural layers [21]. At first, they used the Classycle Analyzer tool (<https://classycle.sourceforge.net>, accessed on 20 May 2026) to determine the dependencies between classes and to build a graph consisting of strongly connected components. This graph then served as a base for the architectural layer partition of types—as it was turned into an acyclic directed graph of strongly connected components. As this first partition was created,

Chidamber and Kemerer metrics [22] were calculated for each layer. After that, boundaries of the previous layer partition were modified iteratively until the recalculated metrics seemed to improve.

Over the years, Dobrean and Dioşan performed a comprehensive study of the field [23], and came up with several approaches regarding the clustering of types and architectural layer recovery. Their main subject of study was to examine the architectural conformance of mobile applications that use the MVC architecture. In their first approach, while analyzing open-source and private (mainly iOS) mobile software, they have shown that it is possible to cluster the user-defined types to architectural layers mainly based on data collected from the basic setup of the used SDK and its design consequences. They represented layers as sets of types and applied several heuristics that were derived from the usage of types in order to categorize them. After this first deterministic approach, they applied machine learning techniques to the same problem to examine how it performs compared with the previous proposal [24]. However, the performance of this second approach was not as satisfactory as expected; therefore, they fused these two techniques into a hybrid solution, which achieved higher performance as it had properties of the first, more reliable deterministic approach, as well as adequate flexibility provided by the second approach [25,26]. It was also mentioned by Dobrean and Dioşan that detecting architectural violations in a codebase could also be important for educational purposes, as the result of the checking process can call attention to errors regarding fundamental design principles [25]. In that way, students might get first-hand experience and more guiding tools even during the early stages of their studies.

More recently, Komolov et al. approached the same problem from a purely machine-learning angle for Android applications written in Java, targeting the MVP (Model-View-Presenter) and MVVM architectural patterns [27]. They constructed a labeled dataset of 5973 data points collected from open-source GitHub projects and trained nine classifiers on source code metrics to predict the architectural pattern, reporting precision, recall, accuracy and F1 score of around 83%. This work demonstrates the feasibility of architectural pattern recognition with general-purpose classifiers on industrial codebases and provides a useful quantitative baseline against which deterministic, domain-tailored approaches can be compared.

The prior work shows that architectural layer recovery and conformance checking can be conducted with a range of techniques, but the cited papers differ in scope, platform, and level of automation. Wongtanuwat and Senivongse [11] detect MVVM violations in Objective-C iOS code using name- and inheritance-based rules with optional manual override, while Aljamea and Alkandari [12] combine regex-based checks with SDK base classes and user input for layers that cannot be inferred directly. Cai et al. [19] use DSMs and DRHs to compare intended and implemented modular structures in student submissions, and Scanniello et al. [10] apply Kleinberg-style hub/authority analysis to dependency graphs for layer recovery. Dobrean and Dioşan [23] develop heuristic and hybrid approaches for mobile MVC layer recovery, and Komolov et al. [27] use machine-learning classifiers to predict architectural patterns from source-code metrics. Against this background, the present work is distinct in being aimed at C# desktop student submissions, combining deterministic Roslyn-based heuristics, explicit ambiguity reporting, and a dependency-graph propagation step with explicit cycle handling. Table 1 summarizes these differences against the most closely related approaches.

**Table 1.** Comparison of the proposed approach with the most closely related prior work on architectural-layer recovery and conformance checking.

| Work                            | Target Arch.            | Platform/Language          | Layer-Recovery Technique                                                                                     | Reasons & Incons. | Educ. Eval. |
|---------------------------------|-------------------------|----------------------------|--------------------------------------------------------------------------------------------------------------|-------------------|-------------|
| Wongtanuwat and Senivongse [11] | MVVM                    | iOS/Objective-C            | Name- and inheritance-based rules, with manual override                                                      | No                | No          |
| Aljamea and Alkandari [12]      | MVC, MVVM               | iOS                        | Regex-based checks and SDK base classes; user input for some layers                                          | No                | No          |
| Cai et al. [19]                 | Layered design patterns | Student software, Java     | DSMs and DRHs to compare intended vs. implemented design                                                     | No                | Yes         |
| Scanniello et al. [10]          | Layered                 | Java                       | Kleinberg-style hub/authority analysis on dependency graphs                                                  | No                | Partly      |
| Doborean and Dioşan [23–25]     | MVC                     | Mobile (iOS)               | Deterministic heuristics, machine learning, and hybrid recovery                                              | Partly            | No          |
| Komolov et al. [27]             | MVP, MVVM               | Android/Java               | Machine-learning classifiers on source-code metrics                                                          | No                | No          |
| Our proposed method             | MV, MVVM                | Desktop (WinForms, WPF)/C# | Deterministic heuristics over the Roslyn semantic model with dependency-graph propagation and cycle handling | Yes               | Yes         |

### 2.5. Research Questions

Based on the previous information, the following research questions are posed:

**RQ1** How accurate is the layer identification process? What are its typical edge cases?

**RQ2** What are the most common issues in the analyzed submissions? What conclusions can be drawn from these?

**RQ3** How can the number of false positive cases be handled and minimized?

## 3. Methods

As an initial step, we have formalized the way architectures will be handled during our research. We described an architecture with its name (MV or MVVM), its possible layer types (*Model*, *View*, *ViewModel*, *Persistence*, *Test*, etc.), its required and optional layer types (which layers must and could be present in an implementation) and its set of layer dependency rules (in the form of a pair of layers *A* and *B* where *A* can depend on *B*).

### 3.1. Violations

Several violations were defined that later served as the basis for automated analysis. Some of these checks were derived from basic architectural rules, while others were derived from general development ideas and conventions usually taught during the related university course. Each of these violations was assigned a unique identifier (see in parentheses) according to the related Microsoft guidelines (<https://learn.microsoft.com/en-us/dotnet/>

[csharp/roslyn-sdk/choosing-diagnostic-ids](https://github.com/csharp/roslyn-sdk/choosing-diagnostic-ids), accessed on 20 May 2026). These rules were the following:

- Errors and inconsistencies during layer identification (ARCH1000–ARCH1001): an error is reported if it could not be determined which architectural layer a type belongs to. An inconsistency is reported if different architectural layers were assigned to a type and there was no way to choose the most suitable one.
- Architectural issues:
  - Missing architectural layer (ARCH1002): a major and required architectural layer is not present. Example: the *ViewModel* layer is missing in an MVVM architecture.
  - Invalid dependency between architectural layers (ARCH1003): there exists a dependency between two layers that is not part of the architecture description. Example: in an MVVM architecture, the *Model* layer uses elements of the *View* layer in some way.
  - Invalid method call between architectural layers (ARCH1004): An invalid method call is present that violates the architecture’s dependency rule set. Example: method calls between two unconnected layers by skipping some intermediate layers.
  - Event handling at inappropriate architectural layers (ARCH1005): handling of an event happens in an inappropriate layer. Example: an event is invoked in the *Model* layer and handled in the *View* layer instead of the *ViewModel* layer.
- Other issues:
  - Potential representation leaks and their usage (ARCH1006–ARCH1007): a type violating the principles of encapsulation by leaking its representation, and the usage of such objects in other types. Cases: a type has a public field or an immutable member field that is directly returned. (A type is considered immutable if it is a string, a record type (or a value type in general), or if it is part of the `System.Collection.Immutable` namespace. Only types in the *Persistence* and *Model* layers are analyzed.)
  - Class depending on concretions (ARCH1008): a class that depends on a concrete type when it could depend on an abstraction of that type. Example: the dependency type has an interface.
  - Usage of invalid event handlers (ARCH1009): use of event handlers of C#’s `Window` type instead of bindings offered by the WPF framework.

### Key Examples

This subsection provides minimalist examples for the defined violations. Listing 1 features multiple violations. Primarily, an inconsistency can be discovered during layer identification (ARCH1001): `MainWindow` inherits from `System.Windows.Window` (which usually implies the *View* layer), but the containing namespace indicates the *ViewModel* layer. The type `Something` is an empty class, outside of any namespaces; therefore, if no additional information is acquired through the solution, this will result in a layer identification error (ARCH1000). Since the class has an event handler, which in the context of an MVVM architecture submission should be handled in the *ViewModel* layer, an additional ARCH1009 violation is signaled.

In Listing 2, the type `MainViewModel` depends on a concretion instead of an abstraction (ARCH1008), as `FilePersistence` implements the `IPersistence` interface, which could be used as a dependency instead. However, since this dependency bypasses the *Model* layer, an invalid dependency between architectural layers is discovered (ARCH1003). Furthermore, on line 21, an invalid method invocation is present, causing an ARCH1004 violation. Line 22 also signals such an error, although the cases of method calls through

public property access get special attention during evaluation. It is notable that ARCH1003 and ARCH1004 violations are not always reported simultaneously, as method calls do not signal explicit dependencies.

In Listing 3, the type `GameBoard` has a public field, causing a representation leak (ARCH1006), while the type `GameModel` may leak its representation through its aggregated `GameBoard` (ARCH1007).

**Listing 1.** Example featuring ARCH1000, ARCH1001, and ARCH1009 violations, through a layer identification error, an inconsistency, and an incorrectly used event handler violating basic MVVM architectural principles.

```

1 namespace SampleSolution.ViewModel
2 {
3     public partial class MainWindow : System.Windows.Window
4     {
5         public MainWindow()
6         {
7             InitializeComponent();
8         }
9
10        private void Button_Click(object sender, RoutedEventArgs e)
11        {
12
13        }
14    }
15 }
16
17 public class Something { }
```

**Listing 2.** Example featuring ARCH1003, ARCH1004, and ARCH1008 violations, through an invalid layer dependency and method calls, and a dependency choice, neglecting a more abstract coupling solution.

```

1 namespace SampleSolution.Persistence
2 {
3     public interface IPersistence { ... }
4
5     public class FilePersistence : IPersistence { ... }
6 }
7
8 namespace SampleSolution.ViewModel
9 {
10    public class MainViewModel : ViewModelBase
11    {
12        private FilePersistence _persistence;
13
14        public MainViewModel(FilePersistence persistence)
15        {
16            _persistence = persistence;
17        }
18
19        public void Method()
20        {
21            _persistence.Load("path");
22            _persistence.Table.Get(3, 5);
23        }
24    }
25 }
```

**Listing 3.** Example featuring ARCH1006 and ARCH1007 violations, related to evident and potential representation leaks.

```

1 namespace SampleSolution.Model
2 {
3     public class GameBoard
4     {
5         public readonly int[] _gameTable;
6     }
7
8     public class GameModel
9     {
10         private GameBoard _board;
11     }
12 }

```

### 3.2. Need for a New Workflow

Since Roslynator is already integrated with TMS and it provides support for running any kind of Roslyn analyzers dynamically, it would be a reasonable requirement to create default Roslyn analyzers that perform architectural checks on the given solution/project and have Roslynator use these for architectural analysis. However, the analysis workflow Roslynator uses is not appropriate for architectural analysis.

Section Roslyn(ator) Analysis Workflow describes how the analysis of a solution happens when Roslynator is used. The problem with this approach is that for architectural checks to be applied, the whole solution (all the relevant projects) must be visited in advance. Otherwise, it would not be known by the time of the violation checks what the architectural layers and even the user-defined types are (as they were not visited before).

A solution could be the approach of merging all projects within a solution into a single project containing all the user-defined types needed before analysis. In theory, this seems like a solvable task if one thinks about the possibility of restructuring namespaces and their types into folders inside a project instead of implementing them in separate projects. However, this method clearly introduces a number of risks, for instance, the possibility of the created project not being buildable. Instead of mitigating such risks, a much cleaner and more transparent method will be proposed.

It would seem logical to report diagnostics after the analysis of all projects finished; however, this can only be conducted in analyzers during the built-in analysis process (this is a consequence of the Roslyn APIs that seek to use only immutable types to enhance performance and enable thread safety [28]). Based on the previous fact, the following idea could be the modification of the Roslynator analysis workflow, which is possible but would end in the violation of basic programming principles.

Since the output of Roslynator is already supported by TMS and it was mentioned that introducing architectural analysis to Roslynator should be avoided, it can be stated that for architectural analysis purposes, the introduction of a new tool and workflow is required. However, it will be an expectation of the new tool to produce an output similar to that of Roslynator (possibly a superset of that, extended with data collected during architectural analysis) in order to help the integration process into TMS, as that output schema is already supported.

### 3.3. Architectural Analysis Workflow

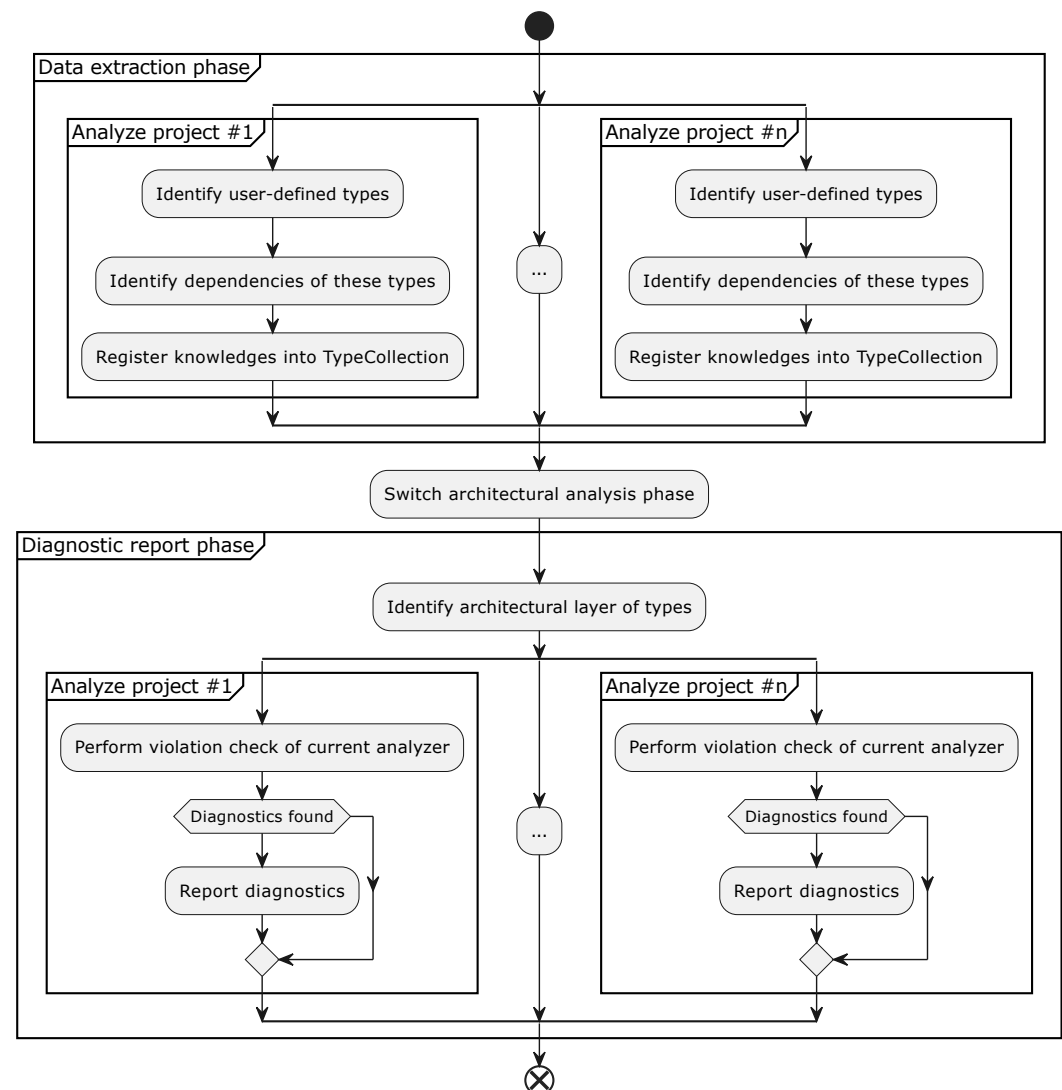
As described in Section 3.2, to perform architectural analysis on a given C# project or solution, the definition of a new kind of analysis workflow was required.

In order to perform architectural violation checks (or higher-level design decision checks in general), all necessary information must be present at the evaluation of the

checks, and therefore, this information must be gathered prior to the checks. That is, to examine architectural conformance and state architectural issues, architectural layers and their components must be recovered first from source code information.

Most of the defined violation checks rely on global knowledge regarding the structure of the examined implementation. Therefore, they cannot be evaluated “locally” in the context of a single C# source file (like other default analyzers would function). Rather, they can only be performed when information regarding all compilation units has been discovered.

During a C# compilation, only the analysis of a single project happens at a time, which means that projects would need to be compiled twice per violation check. Therefore, it is reasonable for performance enhancements to collect information just once, prior to the evaluation of violation checks. Consequently, architectural analysis must be performed using two phases: a data extraction phase and a diagnostic report phase (see Figure 1).



**Figure 1.** Activity diagram of the architectural analysis workflow from the perspective of a single analyzer. The project set is traversed twice: actions are grouped into a data extraction phase (first traversal) and a diagnostic report phase (second traversal).

During the first phase, the analyzers must collect all the necessary information from which they can later perform their violation checks. The main goal of this phase from the perspective of the analysis workflow is to discover all user-defined types in all analyzed projects, and to determine the dependencies between these types, while also collecting information about the kind of these connections (like their locations and expression types).

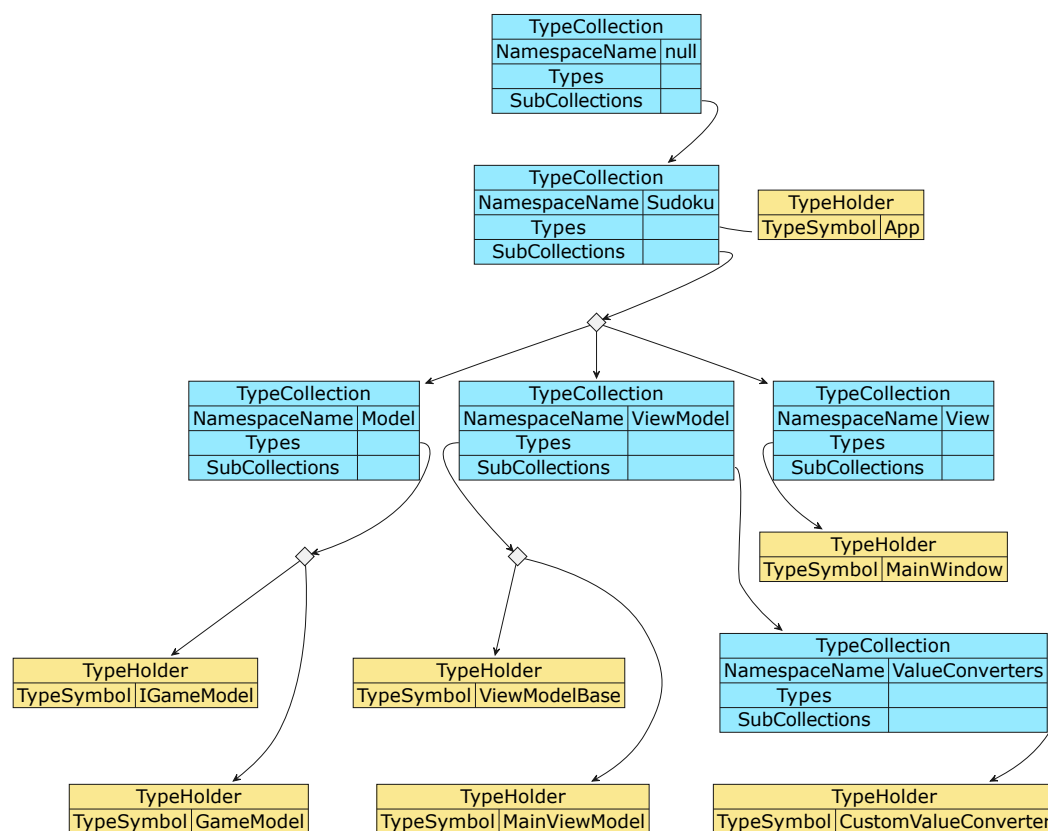
After that, the primary task of the diagnostic report phase is to report diagnostics related to the violation checks of the current analyzer. However, in most cases, the diagnostic report phase must start with the clustering of user-defined types based on the collected data of the first phase, prior to the violation checks.

However, during this kind of analysis, it must be ensured that the collected information from the first phase will be available in the second phase. Therefore, for logical and performance reasons, the use of a global repository is required.

This two-phase design does not impose a significant performance cost. A direct, like-for-like comparison with the default Roslyn analyzer pipeline is not applicable, since that pipeline does not perform cross-project architectural analysis in the first place; relative to a conventional single analysis pass, the only additional work is the second traversal of the project set. The runtime is, in any case, dominated by the underlying C# compilation, and therefore, scales with the size of the analyzed submission. The submissions in this study contained, on average, 1015 lines of C# code (excluding comments and blank lines), and for these, the analysis did not exceed a few seconds per submission on a commodity desktop machine, with the analyzer's built-in concurrent execution enabled.

### 3.4. Analysis Model

The defined analysis model ensures the connection between the two architectural analysis phases as a global repository for storing information. To best fit our objectives, the model itself is a tree hierarchy in which nodes represent namespaces discovered during analysis (see Figure 2). If there is an edge between two nodes *A* and *B*, then *B* is a subnamespace of *A* (meaning that *B* has a prefix of *A*).



**Figure 2.** Simplified representation of an analysis model. Blue TypeCollection objects store the namespaces and yellow TypeHolder objects store types (Any attached knowledge items are omitted in this example).

Also, each node stores the types that are directly part of that specific namespace. Therefore, it resembles the logical structure of the analyzed application, regardless of how types and namespaces were structured in different implementation files.

Besides, the containers of both namespaces and types might be extended with arbitrary pieces of knowledge during the analysis. This is to ensure that the model stays open for any kind of future extension.

### 3.5. Knowledge Items

Knowledge items are meant to be stored as additional information along with namespaces and types. For this reason, the introduction of a number of knowledge types was necessary.

- Knowledge: generic store for any kind of information to be stored.
- DependencyKnowledge: stores the dependency information between two types, the locations where these connections were discovered, and the types of expressions at those locations (Attribute, FieldDeclaration, VariableDeclaration, etc.). These are to be registered to the analysis model by a special syntax and semantic analyzer for all user-defined types during the data extraction phase.
- PotentialLayerKnowledge: stores information about the assumed containing architectural layer of a type or a namespace.
- ConnectedPotentialLayerKnowledge: creates a link to another type. This connection assumes that the architectural layer of the current type will be that of the other type. Used when, at a certain point of the analysis, it cannot be assumed which architectural layer the type belongs to, but it is possible to build such a link.

These last two are to be registered to the analysis model by a heuristic (see Section 3.6) during the layer identification process.

It is important to note that the set of existing knowledge items might be extended, so in case of the introduction of a new violation check (and its implementing analyzer) requiring some new kind of knowledge, the analysis model is prepared for extension.

### 3.6. Heuristics

Heuristics play an essential part in the layer identification process of types. The possibility of extending the set of heuristics used is also maintained. However, each of the currently defined heuristics uses some arbitrary rule to make a guess about the architectural layer of a type. These rules are often SDK-based.

A special property of the following heuristics is that, according to our educational purposes, upon assigning a specific knowledge item to a type or a namespace, a textual explanation is also attached, describing how the connection to an architectural layer was assumed.

- BaseTypeHeuristic: assigns a PotentialLayerKnowledge to a type if it has a base type that is known to belong to a specific architectural layer. Example: types implementing the INotifyPropertyChanged interface are most likely to belong to the *ViewModel* layer.
- AttributeHeuristic: assigns a PotentialLayerKnowledge to a type if the type or any of its members has a specific attribute that implies a specific layer. Example: a class with the *TestClass* attribute belongs to the *Test* layer.
- AppLayerHeuristic: assigns a PotentialLayerKnowledge to a type if it has a method that can be used as an entry point of the application. It will belong to the environmental App layer.
- ReferencedNamespaceHeuristic: assigns a PotentialLayerKnowledge to a type if it has a dependency connection to a type that is in a namespace which implies a specific

layer. Example: a class that uses the DbContext class (located in the System.Data.Entity namespace) is assumed to be in the *Persistence* layer.

- **NamespaceNameHeuristic:** assigns a PotentialLayerKnowledge to a namespace if the namespace has a name resembling the name of a specific layer. For the comparison of names, a special version of the Levenshtein distance [29] is used. Example: types in the *Persistence* namespace are assumed to be part of the *Persistence* layer.
- **TypeNameHeuristic:** assigns a PotentialLayerKnowledge to a type if its name resembles the name of a specific layer. For the comparison of names, a special version of the Levenshtein distance [29] is used. Example: a class named MainViewModel is assumed to belong to the *ViewModel* layer.
- **InterfaceHeuristic:** assigns a ConnectedPotentialLayerKnowledge to a type if it inherits from another user-defined type. The implementing class is assumed to be part of the same layer as its base type.
- **EventArgsHeuristic:** assigns a ConnectedPotentialLayerKnowledge to a type if it is used as an event argument at some point. The type using the event argument will have the layer of the event argument.
- **NestedClassHeuristic:** assigns a ConnectedPotentialLayerKnowledge to a type if it is a nested type of another user-defined type or vice versa.
- **ExceptionHeuristic:** assigns a ConnectedPotentialLayerKnowledge to a type if it is an exception that is thrown at some point.

### 3.7. Layer Identification

The user-defined types in the analyzed application must be sorted into architectural layers prior to evaluating the architectural violation checks. Therefore, at the start of the diagnostic phase of the report, the type layer identification process is performed.

Section 2.4 explores connected studies on architectural layer recovery. Although our approach aligns with these methods, it stands distinct: designed as an educational support tool, our methodology reflects this objective. Consequently, our approach not only provides results but also incorporates the perspective of a university lecturer assessing a submission.

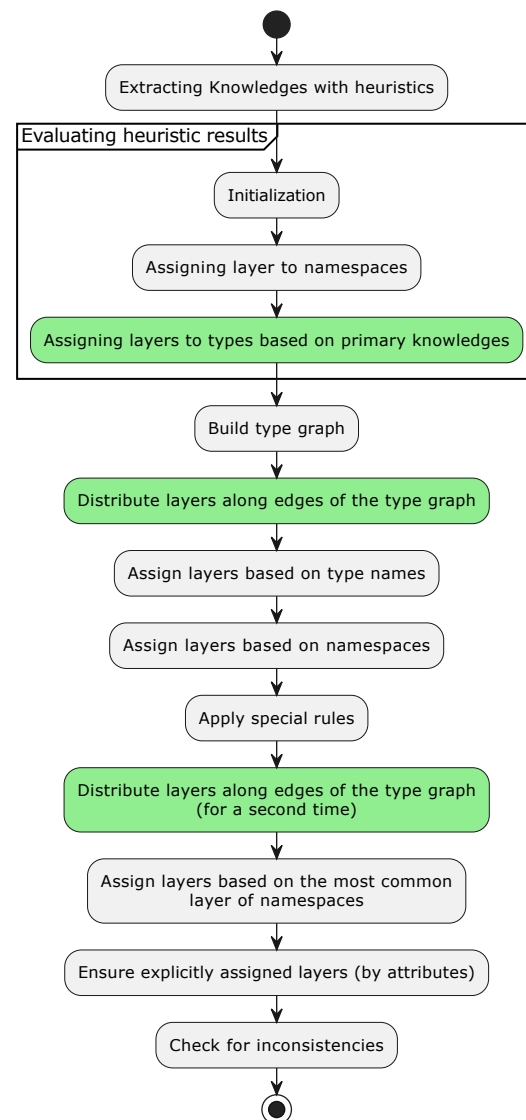
The result of the layer identification algorithm is the partition of user-defined types into pairwise disjoint sets representing the architectural layers. Generally, the algorithm is a deterministic algorithm that uses heuristics to guess the most probable architectural layer to which a type belongs. Through the process, a map of types and their corresponding assumed architectural layers is maintained.

At first, the special layer of CannotBeDetermined was introduced, which would contain types with no specific layer, meaning that after the layer identification process, those types could not be assigned to any layers.

The first part of the layer identification process is that all heuristics are run for each type present in the constructed analysis model, so that the knowledge items registered to the types are present to work with. These are considered the primary layer knowledge items, as these are direct consequences of the data extracted from the analyzed codebase.

The overview of the layer identification workflow is shown in Figure 3. After initialization, the primary knowledge items need to be evaluated (see Figure 4). Throughout this process, information for every type is reviewed. In the absence of any information, determining the layer becomes impossible. If a single piece of information is present, the type is assigned to its respective layer. However, if several pieces of potential layer information are recorded, it is necessary to verify if all these layers coincide. If they do, that particular layer is assigned; if not, a conflict within the heuristic rules is identified (though inconsistencies are not stored at this point, only if they are still present at the end of the

whole process). At this point, it is checked whether there is a dominant layer with the most occurrences among the assumed layers, and if there is exactly one, that layer is assigned.

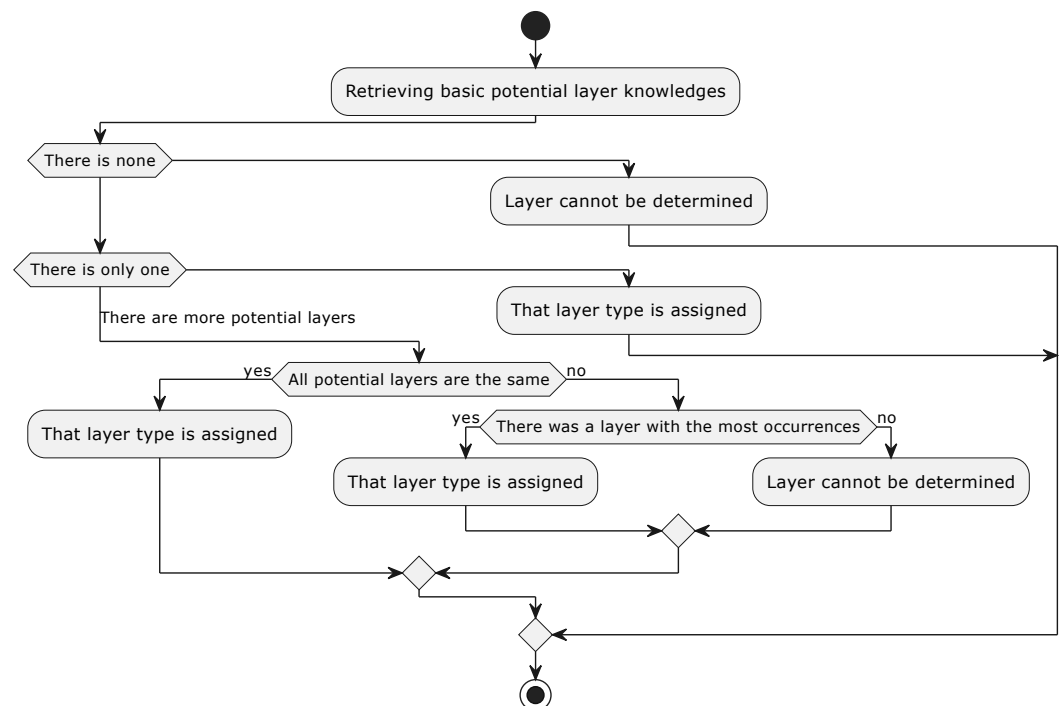


**Figure 3.** Activity diagram of the layer identification process of types. Actions colored green are detailed on Figures 4 and 5.

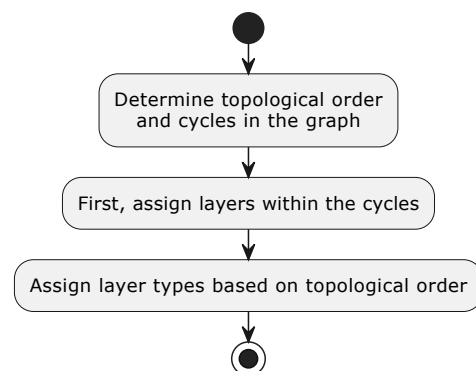
After that, some types are assigned layers, while others are not. In the next step, information about the connection of the layer of types will be used. For this, the construction of a graph is used, in which vertices contain types, and an edge  $(a, b)$  should represent that the layer of type  $a$  determines the layer of type  $b$ .

After such a graph is present, the assumed layer information between types can be distributed along its edges (see Figure 5). However, assigning layers should be conducted with the types (as vertices) in a topological order (to prevent a type from not having a layer when its layer should determine the layer of other types).

Since it is possible that this graph might contain cycles, we propose a modification of the Depth-First Search (DFS) algorithm, which determines a topological order [30] so that when a cycle is detected, the algorithm proceeds to calculate the topological order, while the cycles are collected separately for future use. It is crucial that, as nodes within the cycles can also be part of the topological order, the previously mentioned distribution of layers along the edges needs to be conducted in the cycles first (otherwise, layer information could be lost).



**Figure 4.** Activity diagram of the process of assigning layers to types based on primary knowledge items (the initial layer assumptions registered by the heuristics). A unique layer is assigned only when these assumptions agree or have a single most frequent value; otherwise, the layer of the type is left undetermined at this stage.



**Figure 5.** Activity diagram of the process of distributing layers along the edges of the type graph.

After these, it is true once more that the layers of some types are determined, while others cannot be determined. For these types at that time, a layer resembling the name of the type will be assigned (if one exists); otherwise, the assumed layer of the containing namespace will be assigned to them. Since these steps might modify the layer of some types, the previously described traversal of the type graph must be conducted once again so that layer updates might be redistributed.

Given the possibility that, after completing the layer identification procedure, certain types might not be distinctly allocated to a single layer—or might even be associated with multiple potential layers based on heuristic findings—it is essential to document these errors and inconsistencies. Such issues could potentially impact or entirely compromise the analysis process. For this reason, through the *AttributeHeuristic*, the right must be maintained for users to explicitly state the architectural layer of user-defined types.

The heuristics are not assigned explicit numeric weights. Their relative influence instead follows from the kind of knowledge they produce and from the stage at which it is consulted. The heuristics that yield basic potential-layer knowledge (those based on

base types, attributes, the application entry point and referenced namespaces) determine the primary layer assumptions that are evaluated first (as seen in Figure 4). When several of them suggest different layers for the same type, the conflict is resolved by a majority vote, and if no single layer dominates, the type is left undetermined and the disagreement is recorded as an inconsistency (ARCH1001). The remaining, name- and connection-based heuristics act as weaker evidence: as described above, they are applied only in the subsequent stages and only to types that are still unresolved, and an architectural layer stated explicitly through an attribute overrides the result. In this way, the more reliable, structure- and SDK-derived heuristics take precedence over name- and dependency-based ones without relying on a hand-tuned weighting scheme.

## 4. Results and Validation

In order to assess the proposed methods, a tool capable of performing architectural analysis was implemented and integrated with TMS. Using the tool, student submissions from previous semesters were evaluated. The evaluation of student submissions is two-sided. First, the results of the proposed clustering method had to be evaluated, and then the submissions themselves.

### 4.1. Evaluating the Clustering Method

The goal of this evaluation was to calculate the percentage of correctly clustered types in regard to all types present in the examined submissions. This was crucial as the completeness of the analysis largely depends on the performance of the layer identification process.

Thus, an initial step in this evaluation involved introducing a metric where accuracy was defined as the proportion of types correctly associated with their architectural layer, compared with the overall number of types. To determine the accuracy for a single submission or a collection of submissions, the layer clustering of all reviewed student submissions was manually provided to establish a baseline.

This manual determination of the architectural layer of user-defined types was conducted considering the lecturer's thought process of evaluating these submissions. Hence, ambiguous cases were decided with the examination of the submission as a whole, choosing the closest architectural layer possible. It was also important to be consistent with cases that resemble similar ambiguous cases. If there was no clear indicator that helped identify the layer of a type, the type was placed in a special layer (*CannotBeDetermined*), which contains types without a specific layer. The annotation was carried out by the first author, assigning each user-defined type by manually inspecting the source code of the submission in light of the architectural conventions taught in the course. A type was labeled according to its architectural role rather than its physical location, so that a type the student had placed in a non-standard but architecturally valid namespace or project was still assigned the layer corresponding to its responsibility. To limit the influence of subjectivity, the criteria described above—assigning the closest architecturally valid layer, treating similar cases consistently, and resorting to *CannotBeDetermined* whenever no clear indicator was present—were fixed in advance and applied uniformly to all submissions, and genuinely ambiguous types were left undetermined rather than forced into a layer.

During the evaluation, 947 student submissions were examined. The related C# solutions of these submissions contained a total number of 13,126 user-defined types. These types had to be clustered both manually and with the use of the implemented tool into the seven defined architectural layers (*Persistence*, *Model*, *ViewModel*, *View*, *App*, *Test* and *CannotBeDetermined*).

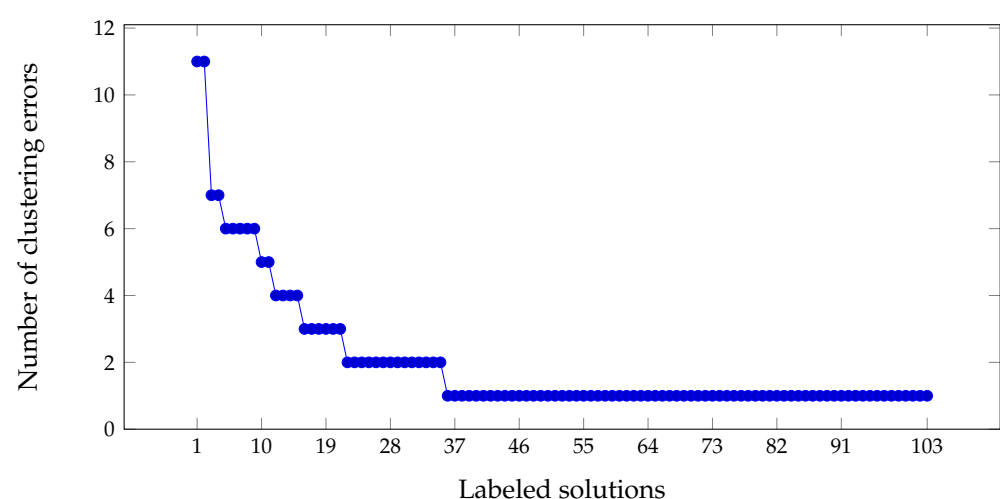
To assess the reliability of the manually established baseline, a second, independent annotation was carried out by a colleague of the second author who was not involved

in the development of the analyzer. This annotator independently assigned architectural layers to all user-defined types in a stratified subset of 100 submissions (10.56% of the full dataset), comprising 50 MV and 50 MVVM submissions and 1367 user-defined types in total. The two annotations agreed on 1349 of these types—that is, on 98.7% of the subset—and differed for only 18 types. The disagreements were not spread uniformly but were confined to architecturally ambiguous boundary cases: most concerned the Model–Persistence boundary (data-access types and types residing in a persistence namespace) and the Model–View boundary (domain entities or animation helpers with no direct user-interface role), while the remainder involved the *Test* layer (classes in the *Model* used only for testing purposes) and enumeration value types that one annotator left undetermined and the other assigned to the *Model* layer. This high level of agreement, together with the localized and explicable nature of the few disagreements, supports the validity of the manually established ground truth.

Inspecting the output of the architectural analyzer tool, the identified architectural layers of each type were collected and matched against the expected layers that were obtained from the manual clustering of types. This comparison showed that out of the total number of 13,126 user-defined types, the architectural layer of 12,920 types was identified correctly, while 206 were incorrectly identified. Therefore, accuracy of the layer identification process in the set of all student submissions is 0.9843 and the percentage of layer identification errors is 1.57%.

The comparison of expected and actual identified architectural layers also showed that of all submissions, 103 contained layer identification errors, that is 10.9% of all submissions. However, it is also important to note that this number contains submissions even if they contain just one type with an incorrectly identified layer.

Figure 6 shows the distribution of the 206 layer identification errors compared with the 103 submissions that contain these errors. It can be seen that in some cases the number of errors in a submission reached a number between 6 and 11; however, it happened only in a negligible number of cases. The average number of errors per solution is exactly 2. This is a noteworthy observation, meaning that there were some submissions which had a relatively bad structure, but in most of the cases, only a couple of layer identification issues emerged.



**Figure 6.** Distribution of clustering errors across the 103 submissions that contained at least one error. For each such submission, the number of incorrectly clustered types is shown; in most submissions, only one or two types were misclustered.

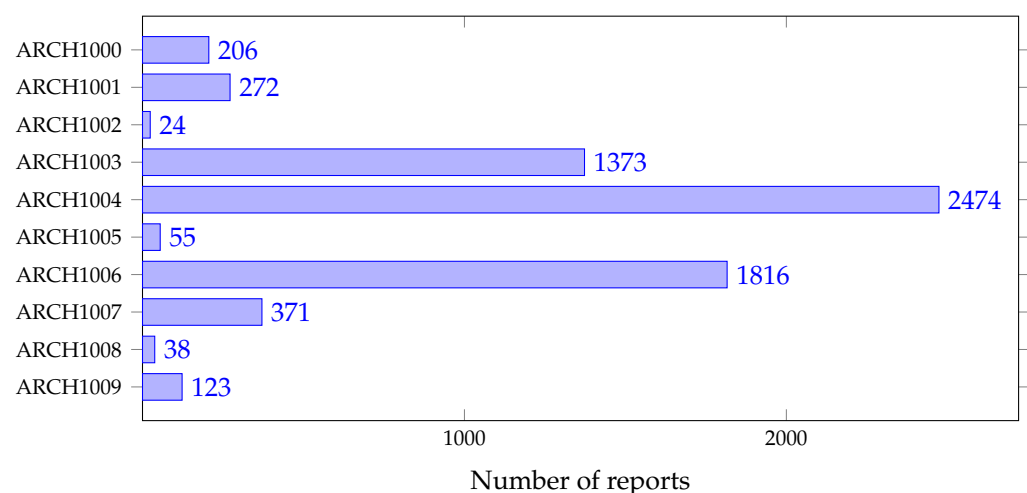
It is also important to mention that in the case of the 206 incorrectly clustered types, the layer of 183 types could not be determined, while in the case of a single type, inconsistencies

were indicated. Therefore, only 22 types were incorrectly clustered without providing any information to the user—which is less than 1% of all types.

Most misclassification cases were linked to an originally incorrect architectural layout of the analyzed submission. However, there were special recurring cases, which were typically sorted into the *CannotBeDetermined* category, like custom logger classes or utility classes. Other typical classification errors could be attributed to the blurred boundaries between architectural layers (for instance, at the complexity of the analyzed submissions, there can be ambiguous cases between the *Persistence* and *Model* layers, when sorting types into architectural layers).

#### 4.2. Evaluating Student Submissions

The second part of the evaluation was to evaluate the diagnostics collected during the architectural analysis of student submissions. The number of each diagnostic reported for all submissions is shown in Figure 7.

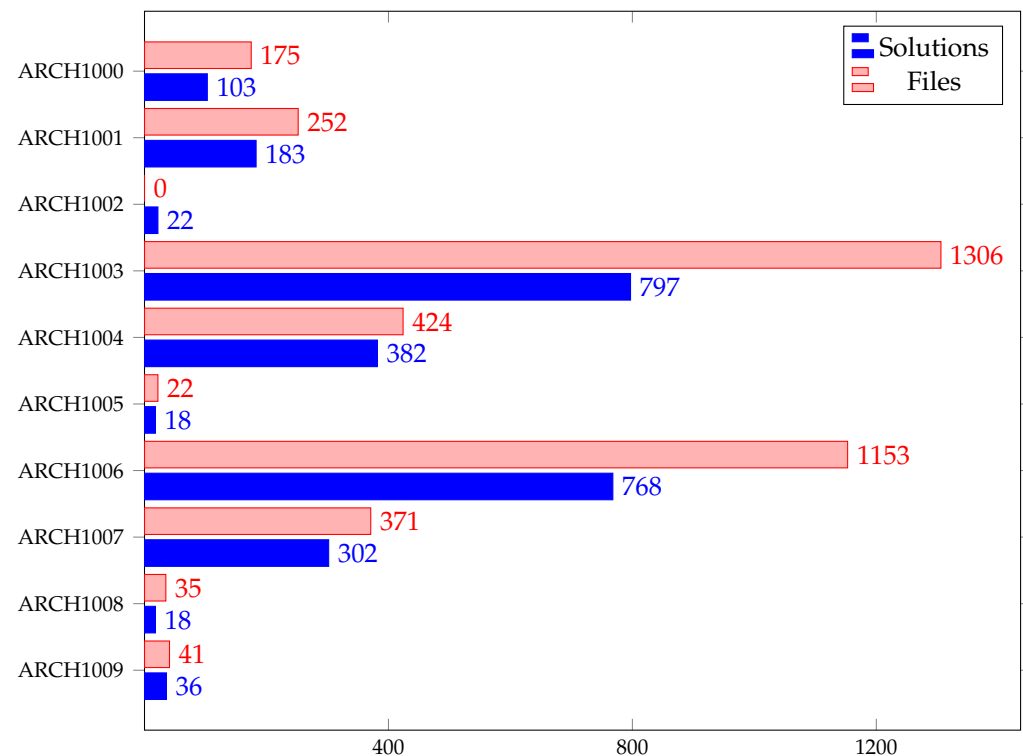


**Figure 7.** Total number of times each diagnostic (ARCH1000–ARCH1009) was reported across all 947 analyzed submissions.

It can be seen that the most commonly occurring diagnostic was ARCH1004 that reports method calls between unrelated layers. However, these diagnostics can be partitioned further, as students were not explicitly prohibited from calling methods two or more layers apart through public properties of an object contained within an adjacent layer. Therefore, if these cases are distinguished from all cases, this leaves ARCH1004 with a mere number of 908 reports, which is not that outstanding at all (nevertheless, assignments of future semesters are expected to require students to follow this rule).

After that, the second most common diagnostic can be related to the violation of encapsulation, concerning the representation leak of objects. This diagnostic was reported more than 1800 times, making it the second most common problem; however, these reported cases can also be grouped into two categories. First of all, out of all cases, there were 623 when public fields were defined within a type. The other cause of such a diagnostic could be a return statement that returns a mutable member field of a type, which happened in 1193 cases.

It is also important to examine how the reported diagnostics are distributed in regard to the submissions and their user-defined types; this is shown in Figure 8. Table 2 displays the same results, extended with the percentage of diagnostic reports compared with the number of all solutions and source files (since in multiple cases more than one type was placed into a single source file, the previously mentioned 13,126 types were distributed in 12,457 source files).



**Figure 8.** Number of submissions (solutions) and source files in which each diagnostic (ARCH1000–ARCH1009) occurred at least once.

**Table 2.** Number of reported diagnostics in the analyzed submissions and their source files, extended with their percentage compared with all solutions/files.

| Diagnostic Id | Solutions |            | Source Files |            |
|---------------|-----------|------------|--------------|------------|
|               | Number    | Percentage | Number       | Percentage |
| ARCH1000      | 103       | 10.88%     | 175          | 1.40%      |
| ARCH1001      | 183       | 19.32%     | 252          | 2.02%      |
| ARCH1002      | 22        | 2.32%      | -            | -          |
| ARCH1003      | 797       | 84.16%     | 1306         | 10.48%     |
| ARCH1004      | 382       | 40.34%     | 424          | 3.40%      |
| ARCH1005      | 18        | 1.90%      | 22           | 0.18%      |
| ARCH1006      | 768       | 81.10%     | 1153         | 9.26%      |
| ARCH1007      | 302       | 31.89%     | 371          | 2.98%      |
| ARCH1008      | 18        | 1.90%      | 35           | 0.28%      |
| ARCH1009      | 36        | 3.80%      | 41           | 0.33%      |

It can be seen that, despite being a diagnostic with only the third highest number of reports, ARCH1003 (invalid dependency between layers) was reported for the majority (84.16%) of all submissions, while affecting nearly 10% of all source files.

Another notable case is related to representation leak reports (ARCH1006). The issue corresponding to the diagnostic is shown to be present in the majority (81.10%) of all submissions, and is distributed among 9% of all source files.

Regarding the diagnostic reports ARCH1007 (possible representation leak), we can observe that its percentage was greater than 30%, and it was reported for exactly 371 source files. The latter is due to the fact that this diagnostic is reported for a type only once, so its

number being equal to the number of reports is completely reasonable. It is also notable that the frequency of ARCH1007 reports is dependent on the reported ARCH1006 cases, as the identified representation leak reports influence the reports of this diagnostic.

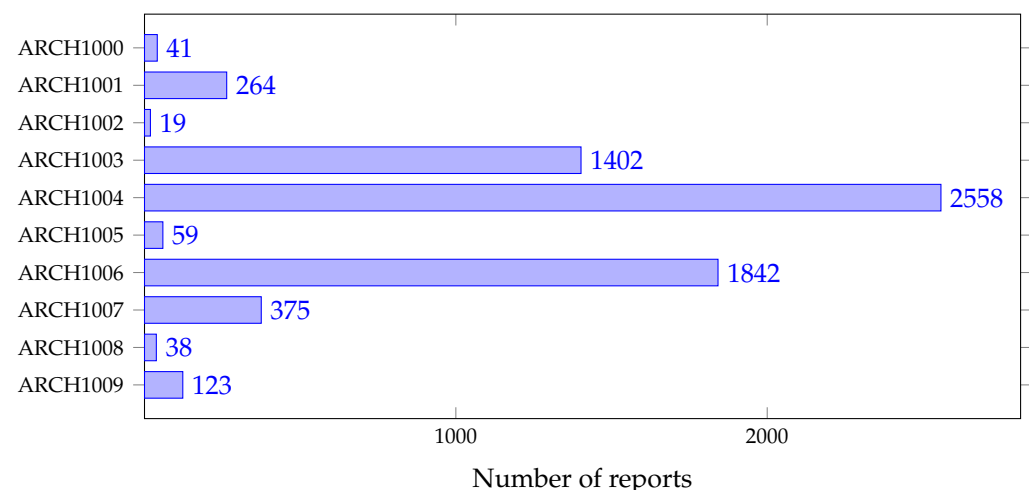
An interesting fact is that the diagnostic with the most reports (ARCH1004 — method call between unrelated layers) was only reported for 40% of all submissions, and is present in a mere number of 3.40% of all source files. From that, it can be concluded that these violations were not so common amongst students, although it means that a group of students made that mistake consistently.

As far as layer determination failures (ARCH1000) and layer identification inconsistencies (ARCH1001) are concerned, their number is significant as these diagnostics could report information about 89% of all layer identification errors (see Section 4.1).

ARCH1002 (missing required layer) is an exception as the diagnostic is reported at the submission level, without a type-specific location. All other diagnostics occurred in a relatively small number of submissions and source files.

### Comparison

It was a natural requirement that diagnostics should not be reported when related violations were related to a type for which its architectural layer could not be precisely determined (for example, if it had inconsistencies during the layer identification process). To assess this, student submissions were modified, and all user-defined types were explicitly applied to their expected architectural layer through the use of C# attributes. This was conducted programmatically using the manually created dataset that mapped types to their expected architectural layer—see Section 4.1. This explicit attribute-based annotation was applied by us solely to construct this idealized reference configuration; it is not part of the regular evaluation workflow and was not used by students. Consequently, it cannot be exploited to influence or suppress the diagnostics reported for an actual submission. These modified submissions were analyzed once more with the architectural analyzer tool. The results of this analysis are shown in Figure 9. These results mirror the expected number of diagnostics that would be reported in an ideal case when the layer identification of user-defined types is 100% accurate.



**Figure 9.** Number of times each diagnostic would be reported in the idealized case where every type is explicitly annotated with its correct architectural layer (i.e., 100% layer-identification accuracy). This serves as a baseline for comparison with the actual reports in Figure 7.

Comparing these results with those of the original analysis, the diagnostics fall into two groups. Diagnostics that report failures of the layer identification itself decrease when the correct layers are supplied explicitly, whereas diagnostics that detect violations on

top of the recovered layers increase, since a larger number of types with a known layer provides a broader basis on which violations can be reliably reported. This is the justified and expected outcome of supplying correct layer information at the time of the analysis. The behavior of the individual diagnostics is discussed below.

It can be seen that although the expected architectural layers of user-defined types were explicitly set before the analysis, there are a few cases when ARCH1000 is reported, which would mean that layers of those types could not be determined. However, it is important to remember that the layer of some types cannot be determined after human examination, and therefore, these cases arise from the fact that these types, in fact, could not be sorted into any architectural layer during manual inspection.

Beyond ARCH1000, the number of ARCH1002 (missing required layer) reports also decreased, as explicitly annotated layers make the presence of the required layers easier to recognize, while the number of ARCH1001 (inconsistency) reports remained essentially unchanged. In contrast, the reports of the layer-dependent violation checks (ARCH1003, ARCH1004, ARCH1005, ARCH1006 and ARCH1007) all increased, as the larger number of correctly identified layers provides a basis for potentially more diagnostics. Finally, ARCH1008 (class depends on concretion) and ARCH1009 (event handlers in .xaml.cs code-behind files) were unaffected, since these checks are structural and do not rely on the architectural layer of the involved types.

As mentioned before, the number of reported ARCH1004 diagnostics can be lower if cases related to special member access and method calls through more layers are distinguished. The same filter applied to the expected results yielded 929 diagnostics.

#### *4.3. Addressing the Research Questions*

RQ1. How accurate is the layer identification process? What are its typical edge cases?

Section 4.1 explained the details of the evaluation of the layer clustering method. It was shown that for the specific type of student submissions related to the course of Event-driven applications, a sufficient number of layer identification heuristics could be defined, which increased the efficiency of clustering. After the analysis of submissions from previous years, it was deduced that over 98% of user-defined types in the analyzed submissions were sorted into correct architectural layers. This number is even higher if we take into account that in almost 89% of the remaining cases, diagnostics were reported regarding layer determination failures and clustering inconsistencies.

However, there are cases that expose the limitations of the technique. The clustering relies on the regularities that well-structured submissions exhibit—a meaningful namespace hierarchy, conventional type and namespace names, and largely acyclic dependencies between layers—and its accuracy degrades when these regularities are absent. Specifically, the layer of a type may be missed or mistaken when a submission does not use namespaces to separate architectural layers, places types outside any namespace, gives types or namespaces names unrelated to their layer, or introduces circular dependencies between types. These situations are not merely adversarial: they arise naturally among novice students who have not yet internalized disciplined code structuring—precisely the population the tool is intended to support. Rather than failing silently, the method is designed to expose such cases: a type whose layer cannot be resolved is reported as ARCH1000, and contradictory evidence as ARCH1001, so that a breakdown of the analysis becomes actionable feedback rather than a hidden error. As reported in Section 4.1, a diagnostic of this kind accompanied the majority of the layer-identification errors observed in the evaluation.

RQ2. What are the most common issues in the analyzed submissions? What conclusions can be drawn from these?

The most common issues of the analyzed student submissions were related to invalid method calls between two unrelated layers, violations of the encapsulation principle (and, therefore, the presence of representation leaks), and invalid dependencies between architectural layers in that order.

It can be concluded that the architectural analysis of the student submissions gave valuable insight into the students' tendencies. First of all, a serious problem can be identified as there were cases in which the layer identification process failed to cluster types into architectural layers. Since the process mainly used well-defined approaches to group types into layers, its failure raises an important concern about the analyzed submission, questioning the fact that the submission is well-structured and its architectural setup is straightforward.

From the perspective of architectural violations, students tend to call methods from unrelated layers (although a significant amount of these occurred through properties of a middle layer). The other important architectural violation confirmed by the analysis was the presence of invalid dependencies between otherwise not connected architectural layers. Besides these, the high number of encapsulation principle violations also raises concern and indisputably requires further action on the part of students and lecturers as well.

The previous cases were highlighted as these were the most common issues during the analysis, which provided an insight into student habits; however, the other rules revealed noteworthy cases also. Finally, the main conclusion from the previous facts is that the importance of the defined architectural and auxiliary rules should be emphasized more during lectures.

RQ3. How can the number of false positive cases be handled and minimized?

During the specification of diagnostics and analyzers, a number of measures have been taken to reduce the number of false positive cases. Most importantly, analyzers do not report any diagnostics if any type related to that violation had inconsistencies during layer identification or the layer could not be determined at all. In this way, diagnostics cannot be reported for types if their architectural layer could not be determined with absolute certainty. The results of this safety measure can be seen when the analysis results are compared with the expected analysis results with explicitly set correct layers, as described in Section Comparison.

This approach makes the number of false positive cases solely depend on the layer clustering process. Therefore, when compared with the manually annotated submission set, besides the violation of rules related to layer identification (ARCH1000–ARCH1001), the originally analyzed submissions contain no false positive cases. This is justifiable, as if there were false positive cases, they would only indicate the incorrect implementation of violation rule checks. However, the comparison of the actually reported and expected violations yielded 143 false negative cases, each of which can be attributed to layer identification errors and inconsistencies. This further demonstrates the criticality and influence of the layer clustering process on violation reports.

## 5. Discussion and Conclusions

Prior to the research, the demand was identified to further increase the level of automatic analysis of student submissions related to specific university courses. Therefore, the possibilities of detecting architectural and design patterns of violation were explored. It was gathered how others approached these problems, and it was deduced that for such an

analysis, an architectural breakdown of the examined student submission must be available before the actual violation checks can be evaluated.

The default analysis workflow provided by the Roslyn APIs was shown to be not suitable for architectural analysis purposes, and therefore, a new workflow was introduced (along with an analyzer tool that uses it). This new model separated the analysis into two parts: a data extraction phase and a diagnostic report phase. The task of the first phase was to discover user-defined types, the dependencies between them, and the type of these dependencies. The task of the second phase was to sort the user-defined types into architectural layers, perform unique violation checks, and report the detected problems.

For the purpose of clustering user-defined types into architectural layers, a number of solutions were taken into account. However, since the proposed architectural analyzer process was meant to be used for educational purposes, it was a natural requirement that it should be able to provide as much information about the discovered errors as possible while mimicking the lecturer's thought process when analyzing a submission.

This was also true for the layer identification process of types: if the architectural layer of a type could not be certainly determined, the analyzer process was expected to indicate inconsistencies in the layer identification. That is, when a type could be assigned to more architectural layers, the fact of conflicting layers should be communicated to the user, along with the reasons why the inconsistency could occur in the first place.

For these reasons, the proposed clustering method uses a combined approach from previous research. The method is deterministic and automatic, which means it is able to cluster user-defined types without user interference. It mainly relies on heuristics that are able to state assumptions regarding the architectural layer of a type or a namespace. The rules of these heuristics usually came from the rules of the SDK used, the rules of the examined university course, and some were based on observations regarding the structure of student submissions.

The method is fully automatic; however, it is important to mention that it enables the explicit assignment of types to architectural layers, which we use only for evaluation and testing purposes, for example, to construct the idealized reference configuration of Section Comparison. This capability is not part of the assessment workflow and is not exposed to students.

After the recovery of information related to user-defined types, the discovery of the dependencies and connections between them, and the clustering of these types into architectural layers, all information was available for the violation checks to be performed and for the diagnostics to be reported.

Since the framework for architectural analysis had already been defined, rules regarding architectural violations were also defined, along with other rules related to the course of Event-driven applications, such as classes depending on concretions instead of abstractions, detection of objects leaking their representation, or declared event handlers in .xaml.cs code-behind files.

After the previous rules were defined, nearly a thousand student submissions from past semesters were analyzed with the created tool. The submissions were written in C# language using WinForms and WPF frameworks, and MV and MVVM architectures. This analysis revealed a large number of errors in the submissions, which could otherwise be overlooked during the manual evaluation. The results of the layer identification process and the diagnostics reported were evaluated and detailed in Section 4. After all, it was concluded that the architectural analysis of student submissions is truly a useful practice and should be used in future semesters. It was also suggested that the results from the analysis of previous student submissions should be taken into account during future

teaching activities. This way, lecturers of the related course could emphasize the most frequent issues and, by that, perhaps eliminate their large percentage.

The conducted research provided practical outputs also, in the form of the architectural analyzer tool implemented (<https://gitlab.com/tms-elte/analyzers/dotnet-architectural-analyzer>, accessed on 20 May 2026) the analyzers it uses and the integration of the tool with the introduced Task Management System (TMS).

#### *Future Work and Evaluation on Ongoing Courses*

In addition to the topics and scope of the conducted research, the following possibilities were identified for future research activities:

- extending methodologies and the set of analyzed architectures,
- extending testing scope to that of industrial applications, mainly for WPF and Avalonia applications using MVVM architecture,
- examining how this kind of architectural analysis could be used for other purposes,
- fine-tuning the layer identification process,
- extending the rules of immutability related to representation leak detection,
- extending methodologies and the analysis model to support refactoring actions.

**Author Contributions:** Conceptualization, M.C.; methodology, B.D.O.; software, B.D.O.; validation, B.D.O. and M.C.; investigation, B.D.O. and M.C.; resources, B.D.O. and M.C.; data curation, B.D.O. and M.C.; writing—original draft preparation, B.D.O.; writing—review and editing, M.C.; visualization, B.D.O.; supervision, M.C.; project administration, M.C. and J.S.; funding acquisition, M.C. and J.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** Supported by the EKÖP-25 University Research Scholarship Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, Hungary, grant number EKÖP-25-4-II-ELTE-1068.

**Data Availability Statement:** The developed architectural analyzer tool implemented as part of this research is released under the BSD-3 license at: <https://gitlab.com/tms-elte/analyzers/dotnet-architectural-analyzer>—accessed on 20 May 2026. The dataset used in this study consists of student programming submissions collected as part of university coursework. As these submissions may contain personally identifiable information, the raw data cannot be made publicly available to protect the privacy of the participants. Researchers who wish to access the data for verification or replication purposes may contact the corresponding author. Access may be granted upon reasonable request, subject to the approval of the relevant institutional ethics committee and the signing of a data use agreement ensuring confidentiality and non-disclosure.

**Acknowledgments:** The authors thank Tibor Guzsvinecz for independently re-annotating a subset of the student submissions, which enabled the assessment of inter-annotator agreement for the ground-truth dataset. This work builds upon and substantially extends the first author's master thesis [31], prepared under the supervision of the corresponding author.

**Conflicts of Interest:** The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

## References

1. Eötvös Loránd University. Student Enrollment Data. 2024. Available online: <https://neptun.elte.hu/VcShowReport/Index/341?skey=kEK61DN0NsNzQ5BjUmDpbFAn> (accessed on 20 May 2026).
2. Anderson, N. College Is Remade as Tech Majors Surge and Humanities Dwindle. 2023. Available online: <https://www.washingtonpost.com/education/2023/05/19/college-majors-computer-science-humanities/> (accessed on 20 May 2026).
3. Kaszab, P.; Cserép, M. Detecting Programming Flaws in Student Submissions with Static Source Code Analysis. *Stud. Univ. Babeş-Bolyai Inform.* **2023**, *68*, 37–54. [CrossRef]

4. Paiva, J.C.; Figueira, Á.; Leal, J.P. Bibliometric Analysis of Automated Assessment in Programming Education: A Deeper Insight into Feedback. *Electronics* **2023**, *12*, 2254. [CrossRef]
5. Microsoft. Desktop Guide for Windows Forms .NET. 2024. Available online: <https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-8.0> (accessed on 20 May 2026).
6. Microsoft. Desktop Guide for WPF .NET. 2024. Available online: <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0> (accessed on 20 May 2026).
7. Microsoft. C# Programming Guide: Events. 2025. Available online: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event> (accessed on 20 May 2026).
8. Mekterović, I.; Brkić, L.; Horvat, M. Scaling Automated Programming Assessment Systems. *Electronics* **2023**, *12*, 942. [CrossRef]
9. Combéfis, S. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* **2022**, *1*, 3–30. [CrossRef]
10. Scanniello, G.; D’Amico, A.; D’Amico, C.; D’Amico, T. An approach for architectural layer recovery. In *SAC ’10: Proceedings of the 2010 ACM Symposium on Applied Computing*; Association for Computing Machinery: New York, NY, USA, 2010; pp. 2198–2202. [CrossRef]
11. Wongtanuwat, W.; Senivongse, T. Detection of Violation of MVVM Design Pattern in Objective-C Programs. In *ICCCM ’20: Proceedings of the 8th International Conference on Computer and Communications Management*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 54–58. [CrossRef]
12. Aljamea, M.; Alkandari, M. MMVMi: A validation model for MVC and MVVM design patterns in iOS applications. *IAENG Int. J. Comput. Sci.* **2018**, *45*, 377–389.
13. Graphviz. *DOT Language*; Graphviz Documentation. 2024. Available online: <https://graphviz.org/doc/info/lang.html> (accessed on 20 May 2026).
14. Hasan, M. Finding the design pattern from the source code for developing reusable object oriented software. In *Proceedings of the Second International Conference on the Applications of Digital Information and Web Technologies*; IEEE: Piscataway, NJ, USA, 2009; pp. 157–162. [CrossRef]
15. Sarkar, S.; Rama, G.M.; Shubha, R. A Method for Detecting and Measuring Architectural Layering Violations in Source Code. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC’06)*; IEEE: Piscataway, NJ, USA, 2006; pp. 165–172. [CrossRef]
16. Sommerville, I. *Software Engineering*, 10th ed.; Pearson: London, UK, 2015.
17. Baldwin, C.Y.; Clark, K.B. *Design Rules: The Power of Modularity*; MIT Press: Cambridge, CA, USA, 2000. [CrossRef]
18. Wong, S.; Cai, Y.; Valetto, G.; Simeonov, G.; Sethi, K. Design Rule Hierarchies and Parallelism in Software Development Tasks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*; IEEE: Piscataway, NJ, USA, 2009; pp. 197–208. [CrossRef]
19. Cai, Y.; Iannuzzi, D.; Wong, S. Leveraging design structure matrices in software design education. In *Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*; IEEE: Piscataway, NJ, USA, 2011; pp. 179–188. [CrossRef]
20. Kleinberg, J.M. Authoritative sources in a hyperlinked environment. *J. ACM* **1999**, *46*, 604–632. [CrossRef]
21. Constantinou, E.; Kakarontzas, G.; Stamelos, I. Towards Open Source Software System Architecture Recovery Using Design Metrics. In *Proceedings of the 15th Panhellenic Conference on Informatics*; IEEE: Piscataway, NJ, USA, 2011; pp. 166–170. [CrossRef]
22. Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [CrossRef]
23. Dobrean, D.; Dioşan, L. Pathways for statically mining the Model-View-Controller software architecture on mobile applications. *Soft Comput.* **2022**, *26*, 10493–10511. [CrossRef]
24. Dobrean, D.; Dioşan, L. A Hybrid Approach to MVC Architectural Layers Analysis. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering—ENASE*; INSTICC; SciTePress: Setúbal, Portugal, 2021; pp. 36–46. [CrossRef]
25. Dobrean, D.; Dioşan, L. Validating HyDe: Intelligent Method for Inferring Software Architectures from Mobile Codebase. In *Proceedings of the Evaluation of Novel Approaches to Software Engineering*; Ali, R., Kaindl, H., Maciaszek, L.A., Eds.; Springer: Cham, Switzerland, 2022; pp. 3–28. [CrossRef]
26. Dobrean, D.; Dioşan, L. Intelligent Methods for Inferring Software Architectures from Mobile Applications Codebases. 2021. Available online: <https://teze.doctorat.ubbcluj.ro/doctorat/teza/fisier/6683> (accessed on 20 May 2026).
27. Komolov, S.; Dlamini, G.; Megha, S.; Mazzara, M. Towards Predicting Architectural Design Patterns: A Machine Learning Approach. *Computers* **2022**, *11*, 151. [CrossRef]
28. Dankovčiková, Z. Custom Roslyn Tool for Static Code Analysis. Master’s Thesis, Masaryk University, Brno, Czechia, 2017. Available online: <https://is.muni.cz/th/f6oc3/masterThesis.pdf> (accessed on 20 May 2026).
29. Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv.* **2001**, *33*, 31–88. [CrossRef]

30. Tarjan, R.E. Edge-disjoint spanning trees and depth-first search. *Acta Inform.* **1976**, *6*, 171–185. [[CrossRef](#)]
31. Orosz, B.D. Automated Validation of Design and Architectural Patterns on Student Assignments with Static Code Analysis. TDK Thesis/Master's Thesis, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2024. Available online: [https://tms-elte.gitlab.io/theses/orosz\\_balint\\_dominik\\_tdk.pdf](https://tms-elte.gitlab.io/theses/orosz_balint_dominik_tdk.pdf) (accessed on 20 May 2026).

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.