# TDK-thesis

Bálint Dominik Orosz

# Automated validation of design and architectural patterns on student assignments with static code analysis


EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF SOFTWARE TECHNOLOGY AND METHODOLOGY




*Author:*

Bálint Dominik Orosz

Computer Science MSc

II. grade

*Supervisor:*

Máté Cserép

Assistant Lecturer


Budapest, 2024

# Contents

## Abstract

Static code analysis can help to locate violations of common coding guidelines, most regularly used design patterns and it can be used to discover information related to the architectural structure of a software. It has also been shown that static analysis of student programming submissions of university courses is a useful practice. In this paper, a method is proposed for analyzing student submissions from past semesters from an architectural perspective. (The submissions are related to the course of Event-driven Applications at *ELTE FI*, and they were developed using either MV or MVVM architecture.) A deterministic approach addressing the problem of clustering user-defined types into architectural layers (in order to help recover the supposed architectural setup of a submission) was introduced based on previously existed ideas extended with domain specific data related to the targeted course. Along with basic architectural rules, special rules (e.g. for representation leak detection) were also defined corresponding to the rules of the course which are expected to be followed by students during development. In order to evaluate submissions using these rules, a prototype of an analyzer tool performing such kind of an architectural analysis was implemented and integrated with an already existing automated submission evaluator. Based on the results, it was concluded that the architectural-specific static analysis of student submissions can provide valuable insights during the evaluation of submissions, while also highlighting the most common flaws in them and therefore supplying feedback for lecturers.



***Keywords:*** static code analysis, C#, MV, MVVM, architecture violation, design pattern violation, student submission

# Chapter 1

# Introduction

## 1.1 Topic

The number of university students enrolling in computer science programmes is rapidly increasing each year [1, 2]. With this number, the amount of student submissions to university course tasks also escalates. This results in not only the multiplication of lecturer workload related to the evaluation of these submissions, but also raises other questions. Along with the previous problem, the amount of attention paid by a lecturer to evaluate an assignment is questionable. 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]. 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 practices.

While the applied static code analysis on student submissions in [4] has already proven to be a valuable approach during evaluation, we argue that it is not satisfactory enough, as it lacks to show support in the discovery of higher level design decisions (such as architectural design) and their violations. It is most fortunate that student submission evaluator systems are already in use with built-in support for automated building and testing pro-

cesses. 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 a 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 in the perspective of the research, and its validation with the evaluation of student submissions from past semesters.

In Chapter 2, related work of state-of-the-art architectural layer clustering and violation detection techniques are presented, along with the approaches aiming to detect architectural and design pattern violations in softwares. The chapter also introduces the related course, its targeted assignments and possible static analyzer options, determines the goals of research and poses the research questions as well. In Chapter 3, the proposed methodology in the form of an analyzer tool, its workflow and the clustering technique it uses are presented, alongside with the defined rules of the analysis. In Chapter 4, the evaluation of student submissions is explained in detail, while the performance and accuracy of the proposed clustering method are examined too. Finally, in Chapter 5, the achivements of the research are summarized, while future research opportunities are addressed.

# Chapter 2

# Background

In the following sections, the necessary theoretical and technical background of the topic is introduced that is required to gain a deeper understanding of the initial problem and later the proposed methods. This background includes the architectures that will serve as bases for the architectural analysis of student submission; the relevant parts of the .NET Compiler Platform SDK that is related to our cause; a description of the Task Management System (TMS) [4]; and the brief background of student submissions to be evaluated. After these, the related work will be presented, the goals of the research stated and the research questions posed.

## 2.1 Design principles

First of all, before the introduction of specific architectures and their setup, the following subsections describe some basic guiding design principles, namely that of separation of concerns and encapsulation.

### 2.1.1 Separation of concerns

The principle of separation of concerns states that during development, parts of the developed software should be separated according to the type of work these parts are supposed to do [5]. The advantage of following this design rule is that it enables components of a system to be loosely coupled, which shows its benefits when it comes to the testing of components or the further development activities related to them.

The default sample to this principle usually includes the separation of business logic components from other components responsible for the presentation of data and user interaction in general.

### 2.1.2 Encapsulation

According to Microsoft, classes should use encapsulation to isolate their inner state from outside consumers of the class [5]. It is also required that this state of an object should only be modifiable through well-defined methods from the outside. Also, they describe how the presence of a mutable global state completely differs from the ideas of encapsulation, as such a state might not be used reliably at different points of execution.

## 2.2 Architectures

The use of architectural patterns during software development have been essential for a long time. Garlan illustrates software architecture as a transfer layer between the requirements of a software and its implementing codebase [6]. It also determines a number of characteristics of software architectures that influence the development lifecycle. Some of which are understanding, reuse, construction, evolution, management, communication and analysis. We aim to investigate the latter within the scope of student assignments written in C# language using Model-View (MV) and Model-View-ViewModel (MVVM) architectures, therefore we introduce the details of these two architectures in the following subsections.

### 2.2.1 Monolithic architecture

Before we introduce MV and MVVM architectures, we must first mention monolithic architecture, which is the most simple architecture. It does not separate functionalities, like presentation, data handling and data persistence. This way, an application created using this architecture will be limited in regard to scalability, and will provide a slower development speed.

During the targeted course, students get acquainted with this kind of architecture at first, but posing as an anti-pattern instead, drawing attention to its problems in order to create necessity for the following architectures.

### 2.2.2 Model-View (MV) architecture

The Model-View architecture is the first step we reach if we start out from a monolithic architecture and take into account the principle of separation of concerns and apply it in order to split up the monolithic architecture into two parts. The first part being the Model layer, containing all the necessary objects which hold the business logic data using encapsulation and manage that data. The other part is the View layer, which is responsible for the presentation of model data and the handling of user interaction.

Model-View architecture is often viewed as the most simple architecture (after monolithic architectures of course). It only contains one layer of separation of concerns, separating the layer responsible for the implementation of business logic (Model layer) from the responsibility of display (View layer).

The user communicates with the View layer, which calls the methods of the Model layer. However, the Model layer is not dependent on the View layer, but provides events, which can offer communication in the other direction.

The architecture might be extended with an optional Persistence layer, responsible for persisting data. In this setup, the Model layer depends on the Persistence layer (not the other way around), calls its methods through its public interface and uses its results.

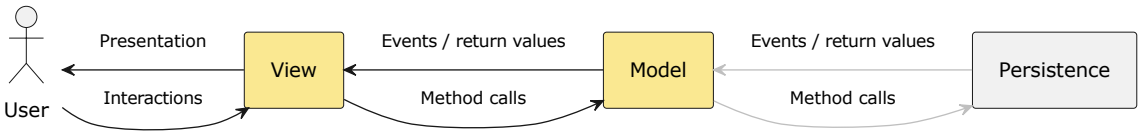


Figure 2.1: Model-View (MV) architectural layers and the flow of data between them.

### 2.2.3 Model-View-ViewModel (MVVM) architecture

MV architecture serves as a base for Model-View-ViewModel architecture, however it further separates responsibilities. It is a special type of Model-View-Controller (MVC) architectures [7, 8]. Instead of the previously seen View layer, it introduces an intermediate layer between the Model and View layers. This ViewModel layer will communicate with the Model layer and preprocess the data it provides for presentation. (This communication will happen through method calls, their return values and events.)

As a consequence of the introduction of the ViewModel layer, the View and Model layers will not be directly connected from now on: all communication will happen through the ViewModel layer. Similarly, the ViewModel layer will notify the View layer through

events, if a data to be presented changes. The communication in the other direction will be possible through data and command bindings.

Similarly to MV architecture, it may also be extended with a Persistence layer.

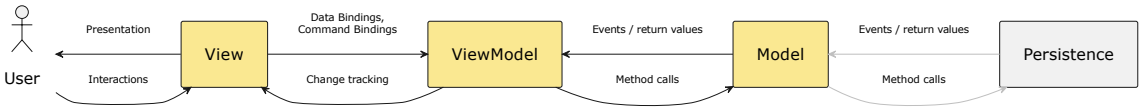


Figure 2.2: Model-View-ViewModel (MVV) architectural layers and the flow of data between them.

## 2.3 Structure of the related course

As it was already mentioned in Section 1.1, an intention of this paper is to examine the possibilities of checking the architectural conformance of student submissions to MV and MVVM architectures. These student solutions were submitted to assignments of the university course of Event-driven applications at Eötvös Loránd University Faculty of Informatics (*ELTE FI*). In this section, the course and its expectations towards students are introduced for the purpose of giving a comprehensive insight before delving any deeper into the theoretical background of the research.

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) [9] and Windows Presentation Foundation (WPF) [10] frameworks.

The course also has a third topic, as students get presented how their previous codebase can be adapted to mobile devices using the MAUI framework [11] (or previously using the Xamarin Forms framework [12]). However, submissions related to these are not analyzed, as we aim to examine MVVM architectural conformance through WPF submissions. (Although, the methodology can be extended to support these other frameworks too.)

During the course, the main objective is to teach students the best practices of specifying and implementing desktop based GUI applications using WinForms and WPF 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 [13]. Moreover, a high

number of industrial applications are developed using these frameworks (especially WPF [14]), so it is fair to say that the industrial demand supports these choices.

Although, at the end of the semester, students must take a 2-hour exam, besides that, multiple assignments are handed out to them randomly from a list of possible assignments. A student must implement the assignment first using WinForms, and then the same assignment using WPF. This way, students are not required to switch context between more problems multiple times, so they can focus on a single task the whole semester. Besides the previous fact, the intention behind this is that students might implement a more mature solution to the same problem, while also realizing first-hand how different approaches and frameworks can be used to solve a certain problem.

Assignments related to the course are usually simple games which can be played on game boards or game board-like structures[1]. 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. Besides that, students are encouraged to separate C# projects in their solutions according to the architectural representation of the assignment. For instance, placing all the business logic related types in a separate project called *'AssignmentName'.Model*, all the display related types in a project called *'AssignmentName'.View* and so on. An advantage of this design is that by using separate compilation units (projects) for the architectural layers, the compiler can detect potential circular dependencies, therefore prohibit this kind of invalid dependencies between architectural layers. Furthermore, with this structuring, each project might have its own set of dependencies, and projects will not have to depend on unrelated dependencies (e.q. the Model layer will have no UI-related dependencies). Besides these, the overall project structure becomes more transparent as it gives a feeling of separated layers.

Although, 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 is still ought to resemble the name of the corresponding architectural layer.

[1] Games that rely on a game board which can be projected onto a coordinate system.

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.

### 2.3.1 Overview of student submissions

The examined university course of Event-driven Applications is explained in details in Section 2.3. This subsection aims to give a general overview of the manual evaluation of student submissions as it can be considered a necessary prerequisite before defining the violation detection rules in Section 3.7.

After examining and evaluating nearly a thousand student submissions from the past two academic years, the following issues and common patterns were identified. Although, there may be special use cases that allow the use of some scenarios/practices listed below, during the related course these cases are considered to be indicators of design and/or development flaws.

- When defining constructors (in order to achieve a manual form of constructor injection of dependencies [15]), some students tend to define constructor parameters with the concrete types and not the interfaces they implement.
- Unfortunately, students tend to violate the principle of encapsulation, and leak the inner representation of objects by declaring public fields or generally by returning references of private fields through which the representation may be changed.
- Placement of types belonging to a specific architectural layer into a namespace which should contain types of another architectural layer.
- Types that do not belong to the Persistence layer using the *System.IO* namespace (or any other namespace that could potentially indicate that the type belongs to the Persistence layer).
- Placement of types into a hierarchically invalid folder or namespace structure (e.g. placing all types into the same namespace or placing a type outside all namespaces).
- Enabling and/or using invalid layer dependencies. For instance, while not using separate projects for each architectural layer, it is possible for a type in the Persistence layer to use a type from the Model layer to persist data (which is possible since

these layers are not placed in different projects, therefore the two using each other would not result in a circular dependency compiler error).

- In the ViewModel layer, re-initializing the whole of an *ObservableCollection* instead of updating its elements. It is a performance problem as the whole collection and its elements need to be rebuilt on the object level and on the user interface as well, instead of using the already existing objects.

- Defining event handlers in files with *xaml.cs* extension. These files contain the class definition for UI elements in a WPF application, and however it is easy to think of a use case when this is valid, the assignments of the course are usually formed so that they expect a solution using the binding features that the WPF framework provides [16].

- Method calls bypassing the below layer in the hierarchy. For instance, a method from a type in ViewModel layer calls a method located in Persistence layer. These two are so to say separated by the Model layer.

- Handled events when the position where the event was raised and the location of the event handler are located in layers that are not related.

- Spelling mistakes in the name of types and namespaces. Although, these issues are not design or development flaws, they will later play an important role during the layer clusterization method.

The previous list is obviously non-exhaustive (and it is not intended to be). Its main purpose is to highlight the problematic cases from the viewpoint of a lecturer during evaluating these kinds of submissions. However, the elements of this list will serve as bases when it comes to defining the rules of analysis.

## 2.4 .NET Compiler Platform SDK

This section introduces the .NET Compiler platform SDK (also known as Roslyn APIs) [17]. The usage of tools that support software development such as IDE features, static code analysis and code generators becomes more and more important. The common characteristic of these tools is that they need to access information that compilers generate. Therefore, the main purpose of Roslyn APIs is to provide a way for other applications to

reach the model which is used and updated by the compiler as it processes source code [17].

The SDK contains several API layers, some examples are the compiler, diagnostic and workspaces APIs [18].

The structure of the Compiler APIs resembles that of a normal compiler pipeline [18]. As the main objective is to offer an insight of the instantaneous state of the compiler to consumers during compilation, parts of the compiler API are designed so that they can be corresponded to a specific task of a normal compiler pipeline. As a compiler has a parsing, declaration, binding and emitting phase, the Compiler API has a Syntax Tree API, Symbol API, Binding and Flow Analysis API, and Emit API respectively. The task of these compiler APIs is to expose the object model related to their compilation phase. The object models provided by these APIs can serve as bases for custom code analyzers.

Diagnostic APIs are such parts of the Compiler APIs, which also make compilations extensible in a sense [18]. Through them, custom analyzers (user-defined or even third-party analyzers) might be used within the compilation process, along with compiler diagnostics.

Besides the previous APIs, the Workspace API is noteworthy as it provides types and access related to essential C# features such as workspaces, solutions, projects and source files. Upon these, code analysis and refactoring features can be implemented [18].

## 2.5 Task Management System (TMS)

In this section, an open-source Task Management System (TMS) [19] is described. The project is developed at *ELTE FI*, and it is also used there for managing tasks of several courses, including Event-driven applications too.

In the beginning, the main advantage of TMS was that it provided automatic building and testing opportunity for the uploaded submissions. However, for the purpose of evaluating student submissions from previous semesters with static code analysis, Kaszab implemented a workflow to TMS that supports the running of static analysis tools on student submissions [4, 3].

After correct configuration, the static analysis workflow consists of two steps. In the first step, the uploaded submission is transferred to a Docker container in which the running of a static code analyzer tool happens. (The workflow specifies a path where the output file containing the results of the analyzer tool must be.) After that, these results

are uploaded to another Docker container, where CodeChecker's Report Converter Tool converts these into JSON and HTML files that can be displayed on the UI afterwards [4].

Instructors are able to upload a Dockerfile or specify a Docker image that will serve as the evaluator container (which should contain the static code analyzer of their choice) and they can also upload a shell script (Bash or Windows Powershell) which contains the instructions of the analysis process. This way, the workflow enables instructors to fully configure the evaluator environment for each task of their courses.

However, a limitation of the evaluator workflow is noteworthy, that is, for CodeChecker to be able to perform the report conversion, the output of the static code analysis tool must be of a special format, which means that only CodeChecker-supported analyzers can be displayed by TMS.

## 2.6 Roslynator

Roslynator [20] 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 a solution, while offering outstanding configurability. The reason that makes Roslynator so useful is the fact that it can be extended with third-party Roslyn analyzers during 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 [4].

### 2.6.1 Roslyn(ator) analysis workflow

The default workflow of Roslynator uses Roslyn APIs which means that the analysis will happen separately for the compilation units, 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.

## 2.7 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.

To understand the problem, first it is important to acknowledge the structure of the student submissions which are subjects to analysis. It was described in Section 2.3, that students are encouraged to place the implementation of architectural layers into separate C# projects inside a solution, nevertheless it is not mandatory. Although, in normal cases this would be the preferable practice, but as far as the analysis workflow is considered, this only introduces problems.

Subsection 2.6.1 describes how the analysis of a solution will happen 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 check 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 inside a solution into a single project containing all the necessary user-defined types before analysis. In theory, this looks like a solvable task if one is to think 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 done in analyzers during the built-in analysis process. (This is a consequence of the Roslyn APIs that strive for using only immutable types to enhance performance and enable thread-safety [21].)

Based on the previous fact, the following idea could be the modification of the analysis workflow of Roslynator, which is possible, but would end in the violation of basic programming principles. A better solution is also possible but that would introduce a circular dependency in the solution, which is a byproduct of the architectural setup of Roslynator. Modifying this setup would also seem possible, although this should be avoided for several reasons. First of all, Roslynator and its set of analyzers are actively maintained and developed, therefore by using a fork of its repository for evaluation purposes in TMS, gaining the latest updates would require maintanance and the introduced changes to the analysis workflow might require the whole redesign of the process.

Since Roslynator is already supported by TMS and it was shown 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.

## 2.8 Related work

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 [22]. However, detecting architectural violations has also been done by other means.

In [23], Wongtanuwat and Senivongse used a set of regular expressions to extract data from source code of iOS applications written in Objective-C. Their approach simply sorted classes into *Model*, *View* and *ViewModel* architectural layers based on their names and the inheritance information related to them. Although, if the end user of their tool was not satisfied with the result of this clusterization, the tool provided a way of manually set-

ting 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 dataflow 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.

In [24], Aljamea and Alkandari introduced their validation model for iOS applications using MVC and MVVM architectures. 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 certainly 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 were 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 to enable visual representation. The created graph emphasized invalid dependencies that violate the checked architecture.

In [25], Hasan examined how the design pattern of an object-oriented Java application could be recovered using data extracted from source code. They created a tool that outputs the visual representation of the relationships between classes in order to provide an insight of 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 which either has a parameter with type 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.

In [26], Sarkar, Rama, and R. describe so-called "Layering principles" that are considered good practices, these are: back-call principle, skip-call principle and cyclic depen-

dency principle. Back-call principle states that an upper layer should depend on a lower layer, while the lower layer cannot depend on the upper layer. Skip-call principle describes that an upper layer should only depend on the lower layer immediately below it. 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) [8], 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.

In [27], Cai, Iannuzzi, and Wong used design structure matrices (DSMs) [28] and design rule hierarchies (DRHs) [29] to examine the conformance of student submissions to the expected design patterns of university assignments. 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 then could 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).

In [22], Scanniello et al. applied the Kleinberg algorithm [30] on industrial applications as well as on applications created by university students written in Java. 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 correspond 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.

In [31], Constantinou, Kakarontzas, and Stamelos used an almost purely graph-based approach to cluster types into architectural layers. At first, they used the Classycle Analyzer tool 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 [32] 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, and came up with several approaches regarding the clusterization of types and architectural layer recovery. Their main subject of study was to examine the architectural conformance of mobile applications which use MVC architecture.

In their first approach [33], while analyzing open source and private (mainly iOS) mobile softwares, 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 to the previous proposal [34]. 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 performace as it had properties of the first, more reliable deterministic approach, as well as adequate flexibility provided by the second approach [35, 36].

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 [35]. In that way, students might get first-hand experience and more guiding tools even during the early stages of their studies.

## 2.9 Goals

Section 2.8 introduces previous approaches to architectural layer clustering and design conformance checking. However, it is arguable that the described type clustering methods expect the codebase to show conformance to the examined architecture to some extent, otherwise their results do not reveal any meaningful information about the failures of categorization. That is, when some issue happens during the layer identification of types, it cannot be guaranteed that errors will be detected related to types with ambiguous architectural layers. Furthermore, those solutions do not wish to align these cases to the report level, so the reason of a type being assigned to a specific (or perhaps more) architectural layer(s) might not be detailed properly. Presumably, it is common during university studies for students to make incorrect design decisions that introduce similar problems. Therefore, a goal of this research is to provide a solution for these gaps.

This paper aims to examine how the type clustering method of Dobrean and Dioşan can be adapted to our cause, which is the automatic checking of architectural design conformance of student submissions related to a specific university course. We wish to investigate that by switching and extending their original scope of research from the analysis of mobile applications with MVC architecture to student submissions with MV and MVVM architectures, while also extending the number of heuristics used during the clustering method and modifying the basic clustering workflow. It is also an objective to define more rules for violation detection that are more descriptive, while also presenting more useful rules for our purposes.

The abstract objective of this paper is to examine how can a methodology be provided for the checking of higher level (architectural and design) decisions, meanwhile it also aims to describe methods for the assistance of university lecturers and students related to course assignments.

The provided methodology targets to mimic the lecturer's thought process of evaluating a student submission. This means that the analysis results are expected to provide as much information about the applied clusterization and the detected violations and their root causes as possible.

As it was previously shown in this chapter, in order to check for the violations of an architecture's ruleset, knowledge about the user-defined types must be known prior to that. Therefore, the application of a layer clustering method cannot be bypassed.

As it was indicated, tools for the recovery of architectural layers of a software have

already been in existence. However, since the scope of this paper is narrowed down to their applications in the course of university teaching and submission evaluation. This paper aims to examine an alternative way of layer recognition and data extraction from the codebase that provides lecturers and students the most insights regarding an inaccurate submission.

## 2.10 Research questions

Based on Section 2.9, the following research questions were posed:

RQ1. How does layer clustering based on heuristics perform on the dataset consisting of submissions from past semesters? How accurate is it?

RQ2. How error-prone is the proposed layer clustering method? What are the edge cases of it? Under what circumstances is it most likely to fail?

RQ3. What are the most common issues of the analyzed submissions?

RQ4. How can the number of false positive cases of the reported diagnostics be minimized?

RQ5. What conclusions can be drawn from the evaluation of student submissions from past semesters?

RQ6. How can the revealed issues be useful from an educational methodology perspective?

# Chapter 3

# Methods

This chapter introduces the proposal for new methods. This includes the proposal for the creation of a new tool that supports architectural analysis, describes its workflow; provides a model that the proposed analysis uses; presents a clustering method of user-defined types into architectural layers (and the heuristics it uses); defines the rules and their corresponding diagnostics for analysis and describes the integration of the proposed tool with an already existing evaluator system.

The proposed methods do not intend to provide methodology for the detection of all the issues listed in subsection 2.3.1, but it is intended to provide the description of an analysis tool and model that fits the goals of research (see Section 2.9).

## 3.1 Proposed tool

It was detailed in subsection 2.7, that as the consequence of the analysis workflow provided by Roslyn analyzers (see subsection 2.6.1), a new tool is needed for the architectural analysis of student submission, that supports a custom workflow. This section introduces the expectations of such a tool in the form of requirements.

### 3.1.1 Requirements

This subsection details the basic requirements defined before the specification of the proposed tool. Some of these requirements come from natural expectations of the tool, while others were formulated keeping in mind that the proposed tool is expected to be integrated relatively easily into TMS (see subsection 2.5).

1. The tool must be able to perform architectural analysis on a given C# solution. In order to do that, the tool must be able:
   (a) to provide an alternative workflow for the analysis of C# solutions[1]; and
   (b) to cluster user-defined types into architectural layers; and
   (c) to detect architectural violations[2].
2. The tool must be configurable in regard to:
   (a) the examined architecture; and
   (b) the auxiliary knowledges of layer identification[3].
3. The tool must be able to output an *xml* file containing the results of the architectural analysis.
   (a) Its path must also be configurable.
   (b) The schema used by the *xml* file must be a superset of the schema used by the *xml* output generated by Roslynator.

### 3.1.2 Configuration

This subsection introduces the supposed model for the configuration of the proposed tool. The configuration can be separated into two parts: options and arguments provided for executing the tool and the any other remaining configuration through a *json* file.

For an execution of the tool, a required argument (path of the solution to be analyzed), a required option (the architecture to check) and an optional option (path of the output file) should be provided.

All the other configuration values should be used to configure the dependency analyzer, the layer identifier and its heuristics. The supposed configuration values are:

**DependencyAnalysis:** contains information to configure the retrieval and extraction of type dependencies.

**IgnoreTypesWithAttribute:** list of attributes. Types marked with any of these attributes should be excluded from the analysis.

[1] That is different from the default usage of C# analyzers.
[2] From the extracted knowledge of the codebase and the result of layer clustering.
[3] Typical base types, attributes, referenced namespaces, etc.

**LayerIdentification:** contains information to configure the clusterization of user-defined types into architectural layers.

**TypicalBaseTypes:** map that assigns each architectural layer a list of type names. User-defined types inheriting from these types should be assumed as a part of the corresponding architectural layer.

**TypicalReferencedNamespaces:** map that assigns each architectural layer a list of namespace names. User-defined types referencing types from these namespaces should be assumed as a part of the corresponding architectural layer.

**TypicalAttributes:** map that assigns each architectural layer a list of type names (these types inherit from *System.Attribute*). User-defined types marked with these attributes should be assumed as a part of the corresponding architectural layer.

All items regarding type and namespace names in the configuration should be fully qualified names.

Appendix A contains a sample configuration for the tool.

## 3.2 Architectural analysis workflow

As it was shown in subsection 2.6.1, the analysis workflow provided by Microsoft (and therefore used by Roslynator too) lacks certain aspects necessary for our cause. It was also described how Roslynator traverses and analyzes projects in a C# solution by visiting projects in a topological order of the dependency graph of projects. It is important, as the architectural analysis workflow will also apply this particular feature.

As it was described previously, the main problem of the Roslynator workflow (and Roslyn analyzers in general) is that they do not make it possible to analyze a solution or its projects multiple times during the same analysis process. The reason this is essential for architectural analysis is that architectural violation checks can only be examined after all projects are processed; and if not provided, the possibility of reporting diagnostics becomes impossible.

Therefore, according to our necessities, we will adjust the analysis workflow by splitting it into two parts. The goal of the first part is to gather all the information necessary for architectural analysis and (mainly from that data) build the analysis model. After this

is done, in the second part, violation checks can be evaluated and then diagnostics can be reported. These parts will be called phases from now on, the former will be referred to as data extraction phase and the latter as diagnostic report phase. The structure and steps of this setup is shown on Figure 3.1.

The default execution of analysis is left unchanged, as the projects in the analyzed solution will be analyzed in a topological order computed from the dependency graph of projects just as before. However, after this first turn of analysis is finished, a main controller can switch between the two analysis phases and execute the default analysis process for a second time. The crucial point of this approach is that the analyzers from the first run must be reused since these objects collected all the necessary data for the diagnostic report phase. The 'switch' object can easily be represented as an instance of a configuration class and then be injected into each analyzer. With the use of such configuration, the initialization of analyzers (that gets executed during compilation) can contain a decision whether to register data extraction actions or diagnostic report actions into the main *AnalysisContext*.

Although, this approach can solve the problems introduced by the demand for architectural analysis, an implementation of the proposed analyzers could not be used in IDEs that collect diagnostics during compilation like *Visual Studio*, as these tools depend on the original analysis workflow.

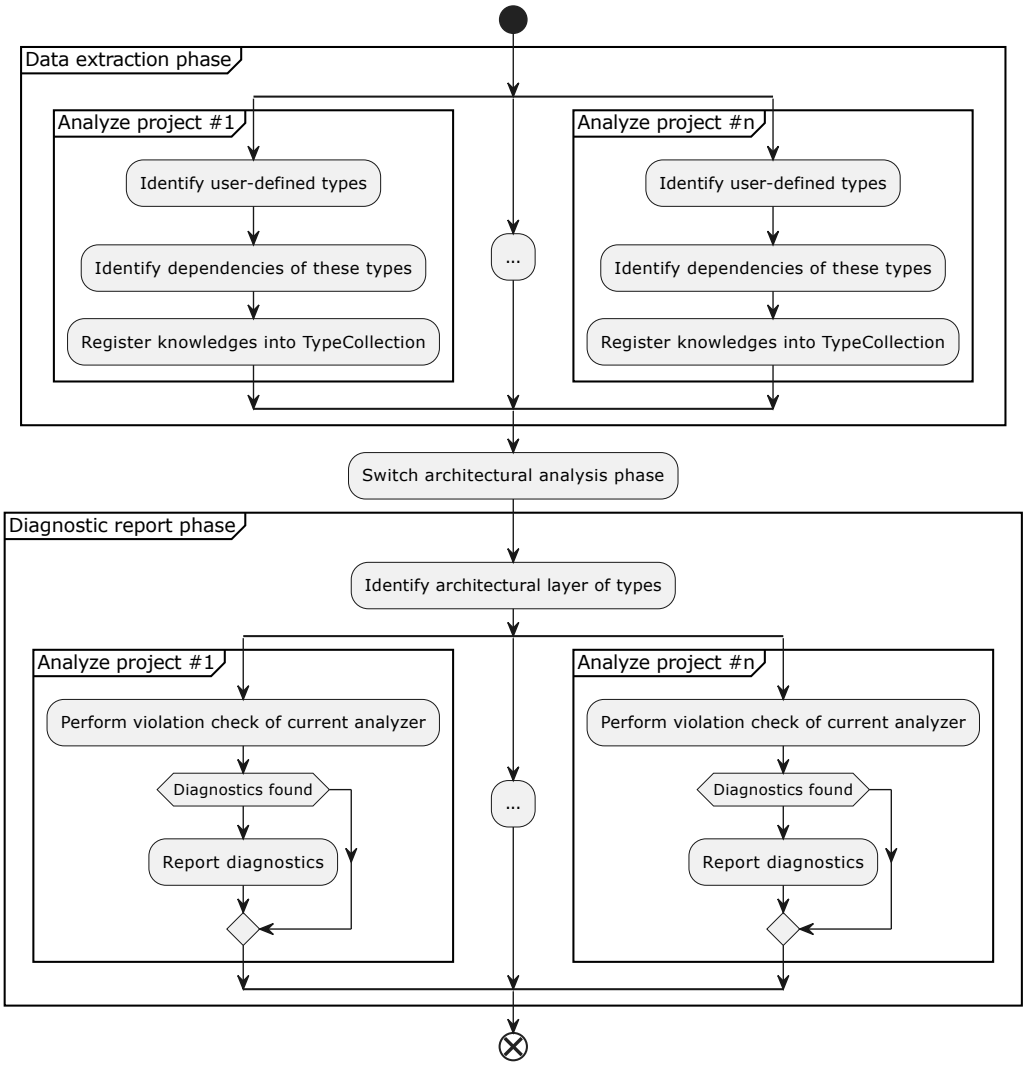


Figure 3.1: Activity diagram of the architectural analysis workflow from the perspective of a single analyzer. Actions are grouped into data extraction and diagnostic report phases.

### 3.2.1 Data extraction phase

During the data extraction phase, analyzers must collect all the necessary information from which they will be able to perform their violation checks later. 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 of these types, while also collecting information about the kind of these connections. For these purposes, two abstract analyzers will be introduced in subsections 3.4.1 and 3.4.2.

### 3.2.2 Diagnostic report phase

The primary task of the diagnostic report phase is to report diagnostics related to the violation checks of the current analyzer. However, the most used *LayerAnalyzer* (see subsection 3.4.3) firstly performs the clusterization of user-defined types based on the collected data prior to the violation checks.

Roslyn analyzers can only report those diagnostics that has their locations inside the currently analyzed project [37]. However, during architectural analysis, when projects are analyzed for a second time, all projects get analyzed and can try to report all diagnostics, so it is important to have a method which filters for the valid reported diagnostics. Therefore, the *BaseArchitecturalAnalyzer* class introduced in subsecion 3.4.1 should also provide a protected method that its derived classes can use to a report diagnostic.

## 3.3 Analysis model

It was realized that a special model for the whole process of architectural analysis is needed. The suggested structure should contain all the necessary data, but also it is ought to be specified keeping in mind that it should contain support for extensibility during future use. In this section, a model is specified that would enable to store information hierarchically in regard to user-defined types, while also being extendable for any kind of metadata extracted from the codebase or collected during the analysis workflow.

### 3.3.1 Knowledge-based approach

To ensure that the model should be extendable, a knowledge-based approach is used. An *IKnowledge* interface is introduced which is an empty interface that can represent some kind of a knowledge in regard to the analysis model later to be introduced (see Figure 3.2).

Another interface named *IKnowledgeHolder* is specified which is able to contain knowledges[4]. Besides that, an abstract class is introduced that can serve as a base for types that want to implement the *IKnowledgeHolder* interface. It contains methods for storing, removing and retrieving knowledge from their maintained collection. As it was mentioned previously, these methods should also enable concurrent usage to their users.

[4]The term *knowledges* is used to describe knowledge items registered to *IKnowledgeHolder*s.

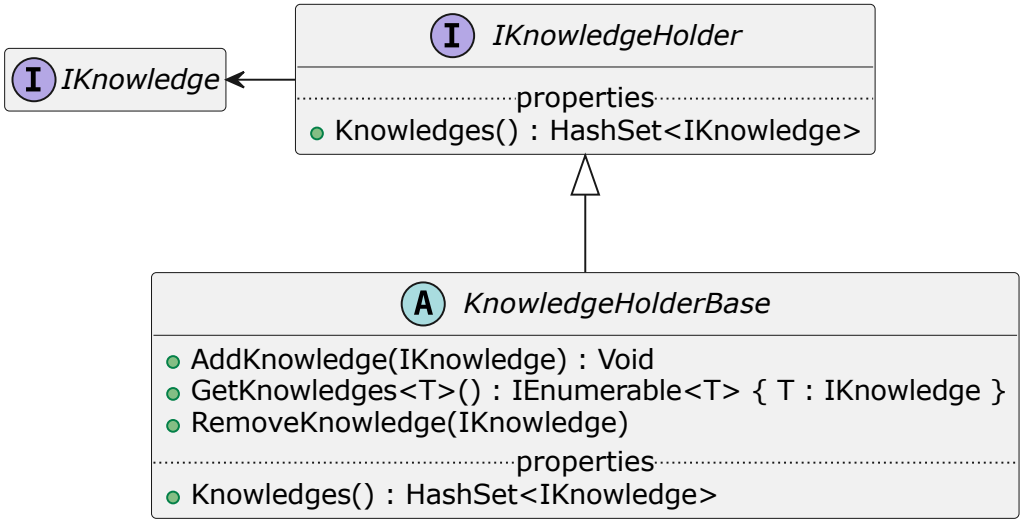


Figure 3.2: Class diagram of knowledge-related interfaces and *KnowledgeHolderBase* abstract class.

At this point, as the possibility of extending default analysis information gained from Roslyn APIs with any kind of knowledge is guaranteed, models that make up the core analysis model can be defined.

### 3.3.2 TypeHolder

At first, we start with *TypeHolder*. It inherits from *KnowledgeHolderBase*, and has a single property of type *ITypeSymbol* which is a product of the Roslyn semantic model and its corresponding API. Objects of *TypeHolder* type will be used to store and wrap all user-defined types during analysis.

As a descendant of *KnowledgeHolderBase*, a *TypeHolder* instance is able to contain knowledge about the type it holds, and that way, the support for extendibility is assured.

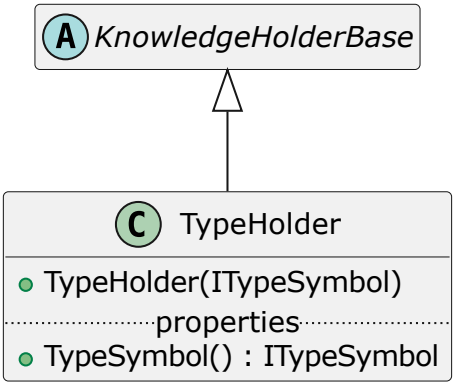


Figure 3.3: Class diagram of *TypeHolder* class.

### 3.3.3 TypeCollection

The other model class introduced is *TypeCollection* (Figure 3.4). A *TypeCollection* object is used to represent a collection of types that belong together in regard to a common namespace.

The structure of the class is recursive, an instance of it holds reference to its parent *TypeCollection* and also references to all of its subcollections. This means that *TypeCollection* objects form a tree that represent the hierarchical structure of namespaces and the types contained within them.

However, it is important to mention that an object of *TypeCollection* is aimed to represent only the last subnamespace of the full namespace of its contained types. Another key feature is that the root of a *TypeCollection* tree represents the global namespace of the solution - types declared outside all namespaces will belong there.

Its inheritance from *KnowledgeHolderBase* is suggested, in order to enable consumers the opportunity of registering knowledges specifically belonging to a collection of types or a namespace.

Besides all its other features, the class must provide functions to query whether a type had already been added to the collection (which work for either a *TypeHolder* or an *ITypeSymbol*). Moreover, the addition of types must also be guaranteed, in the form of methods that propagate down the tree to find the most appropriate namespace that the type to be added belongs to. The supposed class contains helper properties also that are defined to help its usage for consumers.

As far as the creation of *TypeHolder* objects contained within *TypeCollection*s are concerned, it should be provided through the addition method - but not limited to it, leaving the door open for future scenarios.

The importance of using tools supporting concurrent execution must be emphasized once more; an implementation of this class must also take actions to provide thread-safety in order to fit into the analysis workflow.

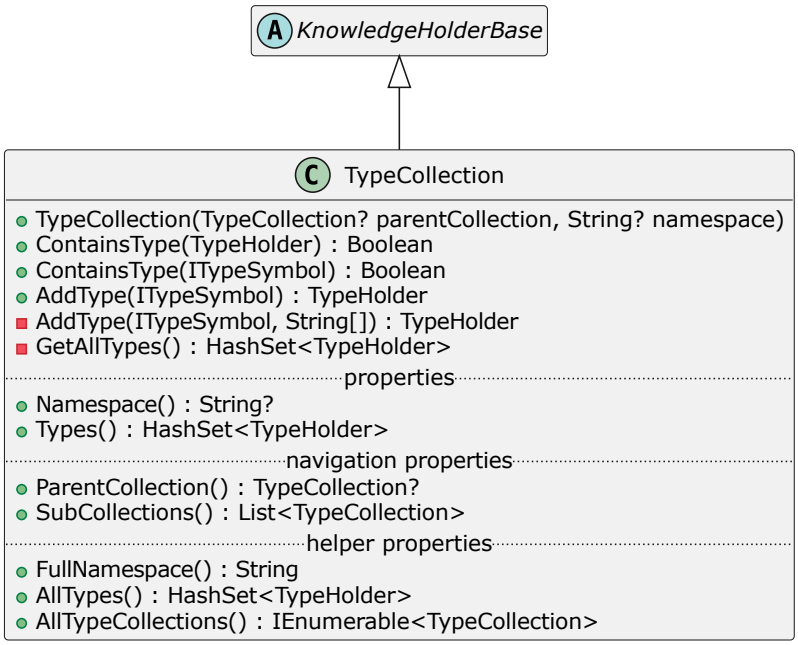


Figure 3.4: Class diagram of *TypeCollection* class.

### 3.3.4 Example for TypeCollection

This subsection aims to provide an example for the *TypeCollection* that is ought to be built during the analysis of a solution. The UML class diagram of the supposed solution is shown on Figure 3.5. During the analysis of this solution, the built *TypeCollection* tree must be equivalent to what is shown on Figure 3.6. It can be seen how *TypeCollection* and *TypeHolder* objects are used to build a tree which represents the hierarchical structure of namespaces, while wrapping the initial underlying analysis data. For the sake of simplicity and understandability, most properties (*Knowledges*, *ParentCollection*, etc.) of the objects were omitted.

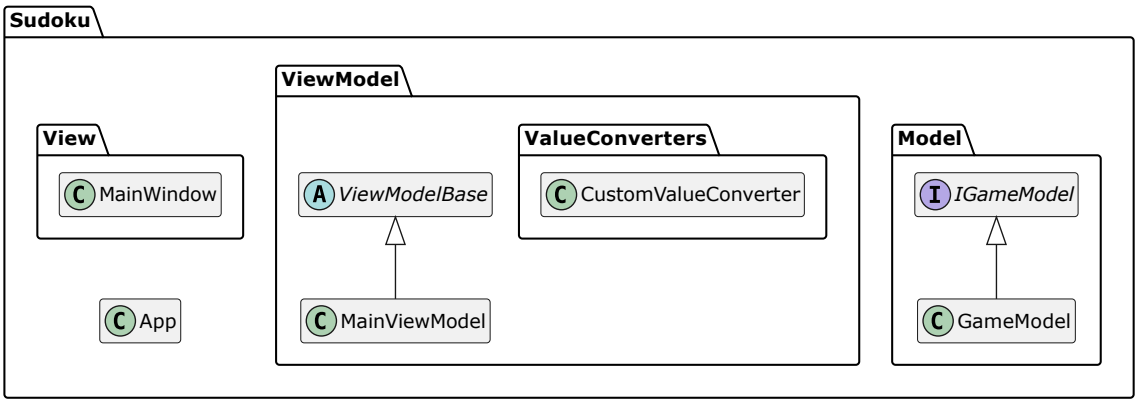


Figure 3.5: Class diagram of an imaginary/abstract Sudoku game.

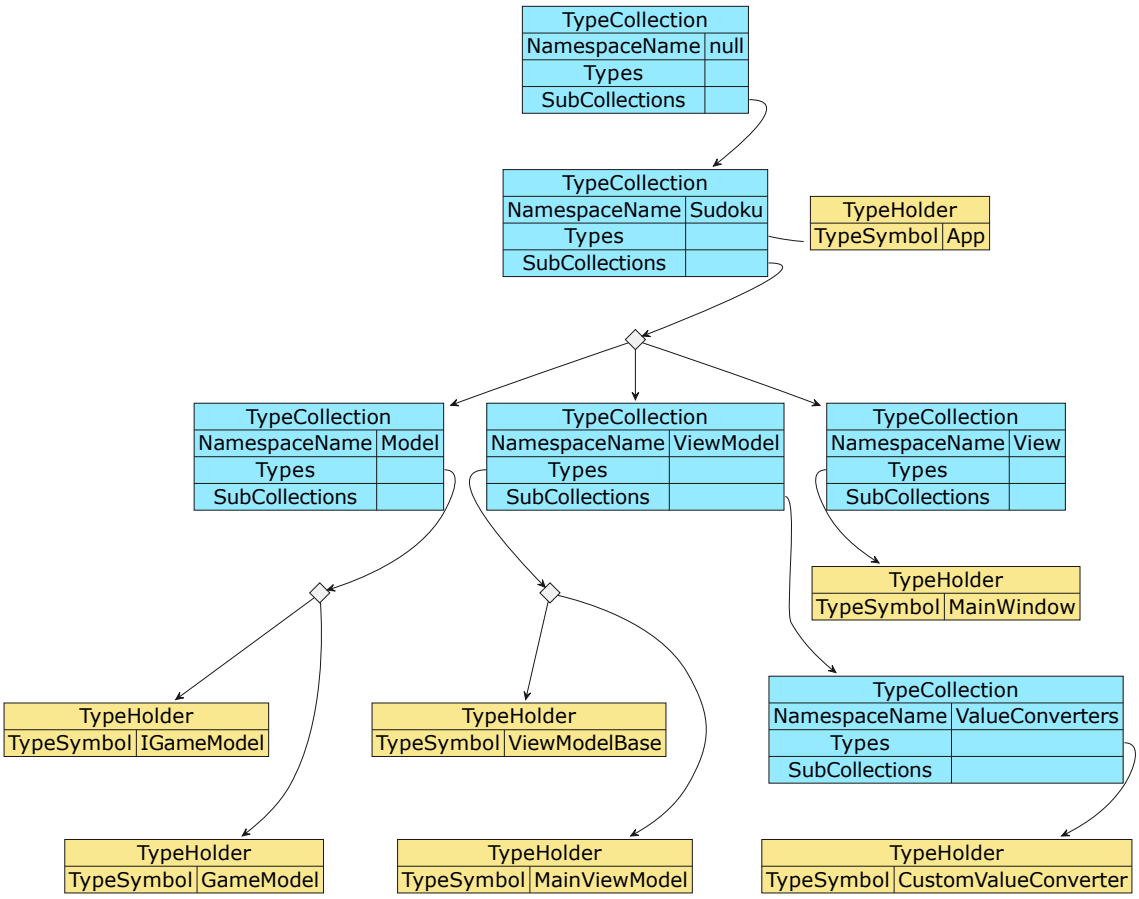


Figure 3.6: Simplified representation of the built tree made from *TypeCollection* and *TypeHolder* objects. Instances of *TypeCollection* are colored blue, while instances of *TypeHolder* are colored yellow.

### 3.3.5 Equality comparers

In C#, the *IEqualityComparer<T>* interface provides methods for the comparison of objects [38]. Its generic type parameter corresponds to a type over which methods must be implemented regarding the comparison of instances of that type. These methods are *Equals* and *GetHashCode*. The main difference between declaring these two methods on the class level (that could be invoked through instances) and implementing these methods in a separate class, is that by the latter option multiple implementations regarding object equality can be provided. It can be advantageous as implementations of *IEqualityComparer* may be used alongside with special *IEnumerable* implementations, notably with *Dictionary* (for comparing keys) and with *HashSet* (for comparing values).

In the proposed analysis model, two equality comparers are implemented: *TypeSymbolNameComparer* and *TypeHolderComparer*. The former is able to compare object instances derived from *ITypeSymbol*, the latter is similar but with

*TypeHolder* instances. The implementation of both classes is relatively simple, *TypeSymbolNameComparer* compares types based on their fully qualified name extended with the name of the containing assembly (through the use of an extension method defined over *ITypeSymbol*), while the methods of *TypeHolderComparer* call an instance of *TypeSymbolNameComparer* using the *TypeSymbol* property of *TypeHolder* parameters. We suggest implementing both classes using singleton design pattern.

### 3.3.6 Architectural rulesets

Since the rules of the examined architectures should be present during analysis, methodology should provide a way of formalising the rules of architectures that can later be used to be matched against. For this reason, an interface named *IArchitecturalRuleset* is introduced (see Figure 3.7). It suggests that architectures should have a way of describing their set of required and optional layers (and with that, the set of all their possible layers) and the dependency rules between these layers. They should also have a method for checking whether a connection between two given layers violate the rules of the examined architecture.

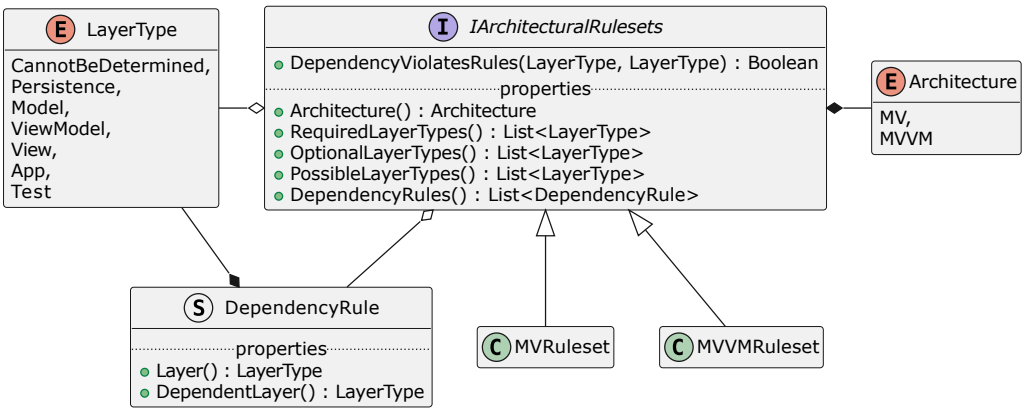


Figure 3.7: Class diagram of the architectural rulesets.

Based on these, MV architecture can be modelled as an architectural ruleset, that requires the presence of the *App*, *View*, *Model*, but optionally can contain *Persistence* and *Test* layers. Its dependency rules should contain the basic architectural rules, rules regarding the testing layer (it may use any other layer), and reflexive rules also (so that invalid dependencies may not arise from a layer depending on itself). Therefore, between these layers, the allowed cases are the following:

- *App* can depend on *App*, *View*, *Model* and *Persistence* layers,
- *View* can depend on *View*, *Model* layers,

- *Model* can depend on *Model*, *Persistence* layers,
- *Persistence* can depend on *Persistence* layer,
- *Test* can depend on *Test*, *App*, *View*, *Model* and *Persistence* layers.

The formalization of MVVM architecture can be done similarly, with requiring the presence of the *ViewModel* layer, letting it depend on itself and the *Model* layer, and making *View* depend on it instead of the *Model* layer. The *App* and *Test* layers may depend on *ViewModel* of course.

## 3.4 Abstract base analyzers

### 3.4.1 BaseArchitecturalAnalyzer

In order to set ground for the analysis setup, an abstract analyzer base class is introduced. It resembles the separation of the two phases, and has a way of switching between them. This class intends to be a base class for all analyzers performing checks related to architectural structure.

*BaseArchitecturalAnalyzer* inherits from *DiagnosticAnalyzer* which is an abstract analyzer provided by Roslyn APIs. Its main features are the property of *SupportedDiagnostics* containing all diagnostics that can be reported from the current analyzer, and the *Initialize* method which is used to setup the analyzer by registering actions into its *AnalysisContext* parameter. Its class diagram is shown on Figure 3.8.

Besides these, *BaseArchitecturalAnalyzer* gets an instance of *ArchitecturalAnalysisOptions* (later to be introduced) in its constructor, that can be used to configure its behavior from the outside. From the perspective of this base analyzer, the important point is that this options class has a property of enumeration type *ArchitecturalAnalysisPhase* that represents the two previously described phases.

Furthermore, the class needs to provide two virtual methods with no default implementation. These methods correspond to the two phases of data extraction and diagnostic report. The logic behind this structure is that if the analysis is in the data extraction phase, the *Initialize* method will call the *InitializeContext* method with the given context, otherwise the *ReportDiagnostics* method will be registered into the context as a compilation end action [39], so it will run at the end of the analysis. In either cases, according to the best practices, the *Initialize* method invokes *EnableConcurrentExecution* on the analy-

sis context for enhanced performance. As a consequence of this, all model classes and analyzers are expected to function well in a multithread environment.

It is also notable, that *BaseArchitecturalAnalyzer* class has a static property of type *TypeCollection*. This instance serves as the top level *TypeCollection* and represents the main analysis model that all analyzers should work on.

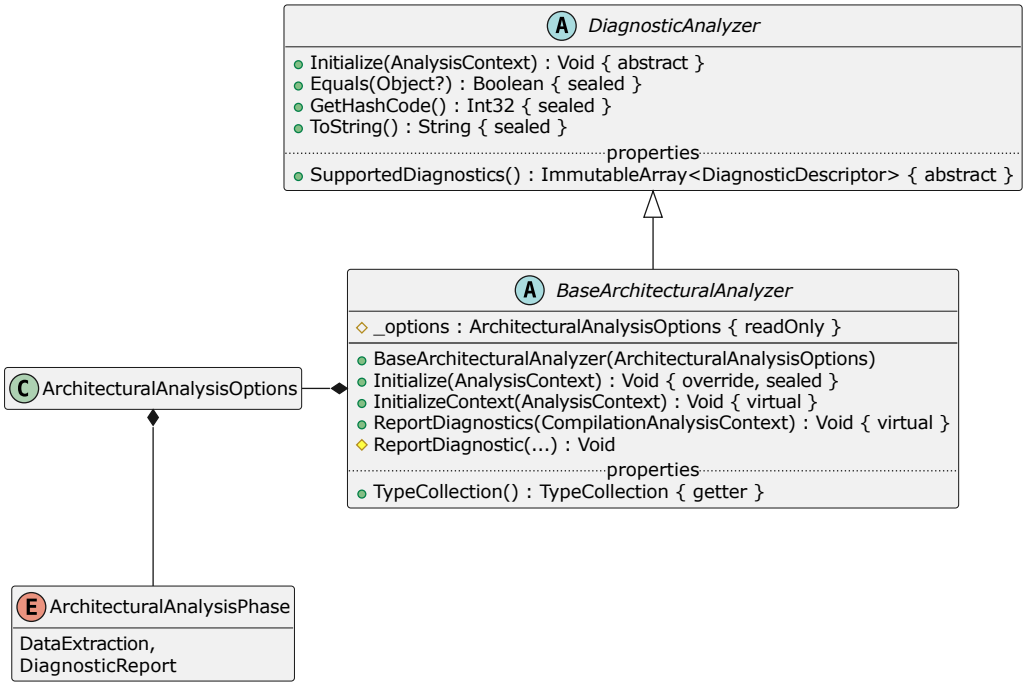


Figure 3.8: Class diagram of *BaseArchitecturalAnalyzer* abstract class and related types.

### 3.4.2 DependencyAnalyzer

*DependencyAnalyzer* is another abstract analyzer class which concrete future analyzers can inherit from. It is a derived class of *BaseArchitecturalAnalyzer* that overrides the latter's *InitializeContext* method. It registers a method into the *AnalysisContext* that can process *SyntaxNode*s - this method can and will be executed upon analyzing type definitions, such as *class*, *struct*, *interface*, *record struct*, *record* in general or an *enum*.

Upon seeing a new type that has not been added to the main *TypeCollection* of *BaseArchitecturalAnalyzer* before, the type is added to it. After this, the other main feature of *DependencyAnalyzer* analyzer must be implemented which is the task of collecting the dependency types of the currently found type. This process is explained in detail in Section 3.5.

### 3.4.3 LayerAnalyzer

The last abstract analyzer specified is *LayerAnalyzer* which derives from *DependencyAnalyzer*. While *DependencyAnalyzer* may be seen as the coordinator of data extraction phase, *LayerAnalyzer* might be its counterpart in diagnostic report phase. It expects an *ILayerIdentifier* implementation in its constructor (which is responsible for the clusterization of types into architectural layers; see Section 3.6), and provides a *LayerIdentificationResult* property that will contain the results of clusterization.

While *DependencyAnalyzer* has overridden *InitializeContext* of *BaseArchitecturalAnalyzer*, the supposed *LayerAnalyzer* class should override *ReportDiagnostics* with including a call to its *ILayerIdentifier*. More on the layer identification process is described in Section 3.6.

## 3.5 Discovering dependencies

For the detection of dependencies of a type, *ClassReferencesSyntaxWalker* is introduced. It is derived from *CSharpSyntaxWalker* provided by *Microsoft* [40]. It is capable of visiting the syntax tree using depth-first search. Its main feature is that it provides virtual methods for visiting the different types of syntax nodes in the abstract syntax tree (AST), for instace attributes, field, property and variable declarations, object creation expressions, etc. Basically, it provides methods for descendants of *SyntaxNode*.

For the implementation of such a class, we recommend overriding the methods for most (if not each) syntax node types that its base class provides, and through the *SemanticModel* corresponding to the visited tree, each method could get the type information of the syntax nodes and the retrieved types can be viewed as dependencies of the current type.

In order to be more specific in regard to dependencies, we suppose the use of a categorisation. We introduce *DependencyKnowledge* to represent the dependency of a type, which can be registered into a *TypeHolder*, as it is an implementation of *IKnowledgeHolder*. A *DependencyKnowledge* instance should contain the two *ITypeSymbol*s involved in the connection, and a collection of locations where the dependency was discovered. For such a location, the introduction of a *DependencyLocation* struct becomes reasonable, which can contain the *Location* (provided by Roslyn APIs) and the source of the dependency which corresponds to the type of the syntax node where

the dependency was found. For the latter, we suppose the use of a new enumeration type (*DependencySource*) as the built-in type that would be appropriate (*SyntaxKind* enumeration type) uses an incremental correspondence to its values and *ushort* numbers, but it would be more advantageous for our case if the corresponding numbers of enum values would be the powers of two, as they could be combined later.

## 3.6 Layer identification of types

In this section, a clustering method for user-defined types into architectural layers is presented. As a part of that, its clustering workflow is explained in detail, as well as the heuristics it uses, alongside with the special rules applied in order to gain higher accuracy.

### 3.6.1 Proposed clustering workflow

The sole purpose of this clusterization method is to get a partition of the set of user-defined types, so that the subsets representing each architectural layer are disjoint sets pairwise. The proposed method is a deterministic approach of identifying layers, which means that it is expected to produce the same output each time for the same input data.

Before defining the process itself, the possible architectural layer values are defined in the form of an enumeration type (*LayerType*). It contains the special value of *CannotBeDetermined* which is used to represent the failure of the clustering method to sort the type into any architectural layer. (This failure is not meant to represent the failure of the layer identification method, but the lack of information about the type.)

As the first part of layer identification, all heuristics (see subsection 3.6.3) are run for each type in the main *TypeCollection*. These heuristics register *PotentialLayerKnowledge*s to *TypeHolder*s (and in special cases to *TypeCollection*s also - see subsection 3.6.3). These are considered the primary layer knowledges as these are direct consequences of the data extracted from the analyzed codebase.

The identification process maintains a map of types and their corresponding output layers. At first, all types are assigned the layer type of *CannotBeDetermined*. Layer types are also assigned to namespaces (and therefore *TypeCollection*s). After initialization, the main *TypeCollection* is traversed and all namespaces are assigned a layer type. The assigned layer of a namespace is determined by whether the related *TypeCollection* has a registered *NameBasedPotentialLayerKnowledge*. If it has, the namespace is assigned the

layer contained in the knowledge, otherwise the namespace is assigned the layer of its closest ancestor (whose layer could be determined).

After initialization, as results of the heuristics are present, these primary knowledges need to be evaluated (see Figure 3.10). All types in the *TypeCollection* and their registered *BasicPotentialLayerKnowledge*s need to be examined. If there is none of that kind, the layer of the type cannot be determined. If there is exactly one knowledge, its layer is assigned to the type. However, if there were multiple potential layer knowledges registered by the heuristics, it must be checked if all these registered layers are the same. If that is the case, the type has a specific layer to be assigned, otherwise a conflict of heuristic rules is discovered. (This clustering method aims to provide information about such conflicts and inconsistencies, but at this point, this particular situation is not stored.) At this point, the algorithm checks whether there is a layer with the most occurrences in the set of registered layers, and if there is one, that layer is assigned, otherwise layer information about the examined type cannot be certainly determined.

After the previous section, some types are assigned a specific layer, while the layer of other types could not be determined so far. In the next step, information about the connection of the layer of types will be used. At first, we suppose the creation of a graph, which should have edges containing types. An edge of this graph $(a, b)$ should represent that the layer of type $a$ determines the layer of type $b$. The nodes of the graph should be a subset of all user-defined types that are present on either side of a connection represented by a *ConnectedPotentialLayerKnowledge*.

After the previously described graph is present, layer information between types can be distributed along its edges. However, assigning layers should be done with the types (as nodes) in topological order (to prevent a type not having a layer when their layer should determine the layer of other types).

Since it is possible that this type graph might contain cycles, these cycles must also be handled. We suppose a modification of the Depth-First Search (DFS) algorithm that determines a topological order [41] 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 done in the cycles first (otherwise layer information could be lost).

Edges in the graph that are consequences of knowledges determined by *InterfaceHeuristic* are ignored in case of types belonging to the *Test* layer during the

traversal of nodes as they usually represent mock implementations of interfaces (and most reasonably they should not be considered as belonging to the same layer).

After these steps, it is true once more, that the layer of some types are determined, while others could not be determined. Although, *NameBasedPotentialLayerKnowledge*s can be applied to specify the layer of more types. For types that do not have a specific layer at that time, a layer resembling their name will be assigned (if one exists - see 3.6.3). If there are any types without a layer after the previous step, the layer of their containing namespace will be assigned to them.

Although, the previous process might seem rational, there are special rules that may need to be applied in order to avoid the miscategorization of types. One such rule is that if a type references the *System.IO* namespace, but based on its name or the name of its namespace, it could be assigned to another layer (besides Persistence), it should be assumed that the type is part of that other layer. Also, if the application of this rule would change the layer of a type, then it should also change the layer of types connected through *NestedClassHeuristic* - as otherwise these changes would be invalidated during the next graph visit.

After the traversal of nodes in the type graph, layer of many types possibly got determined. Therefore, it has also become possible to determine the layers of their connected types. For this reason, the nodes of the previous graph must be traversed once more to distribute these newly discovered layers along the edges of the type graph.

Since it is imaginable that not all types were assigned layers besides *CannotBeDetermined*, these types will be assigned the most occurring layer in their containing namespace (if that is unambiguous), presuming that namespaces were formed to represent the structure of architectural layers.

If a type has more connected layers, and these would indicate that the type should belong to different layers, the type will be assigned the layer with most occurrences (although, this ambiguity will be discovered as an inconsistency).

As a last step, if the layer of a type was explicitly provided in the codebase (via an attribute), it is possible that this layer was overridden by the previously detailed layer identification process. However, we want to maintain the possibility of explicitly stating a type's layer, therefore after the process, a method should ensure these cases.

Furthermore, a method should search for all inconsistencies of the process. Such a method would have to identify cases when knowledges registered to a type and its namespace describe conflicting layers. (App layer knowledges are ought to be excluded as usu-

ally its related types are not placed in separate namespaces.)

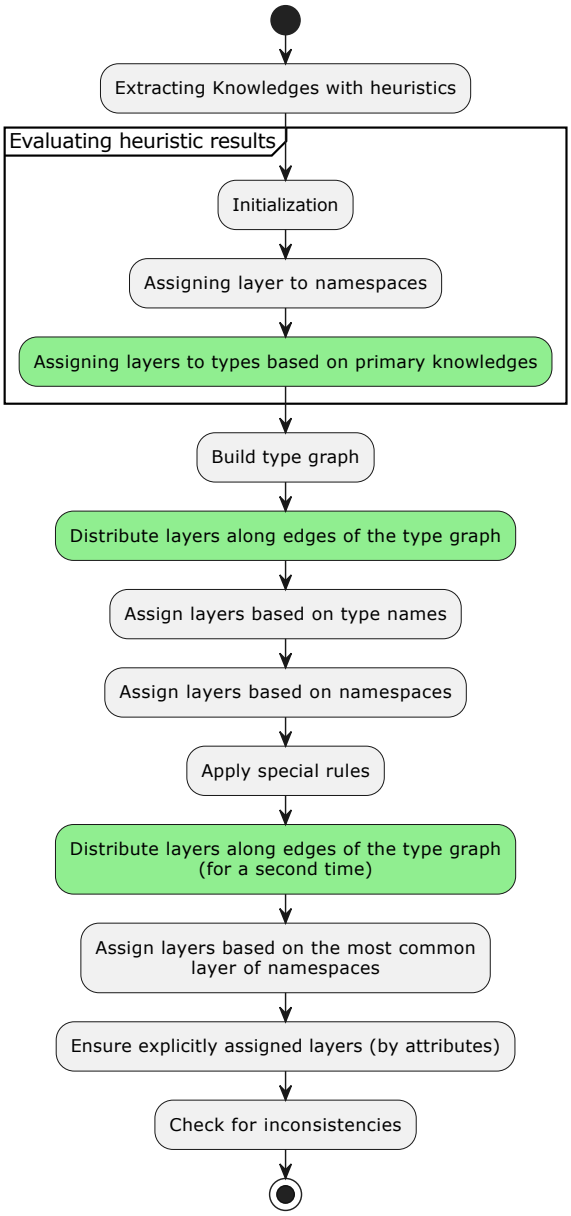


Figure 3.9: Activity diagram of the layer identification process of types. Actions colored green are detailed on Figure 3.10 and Figure 3.11.

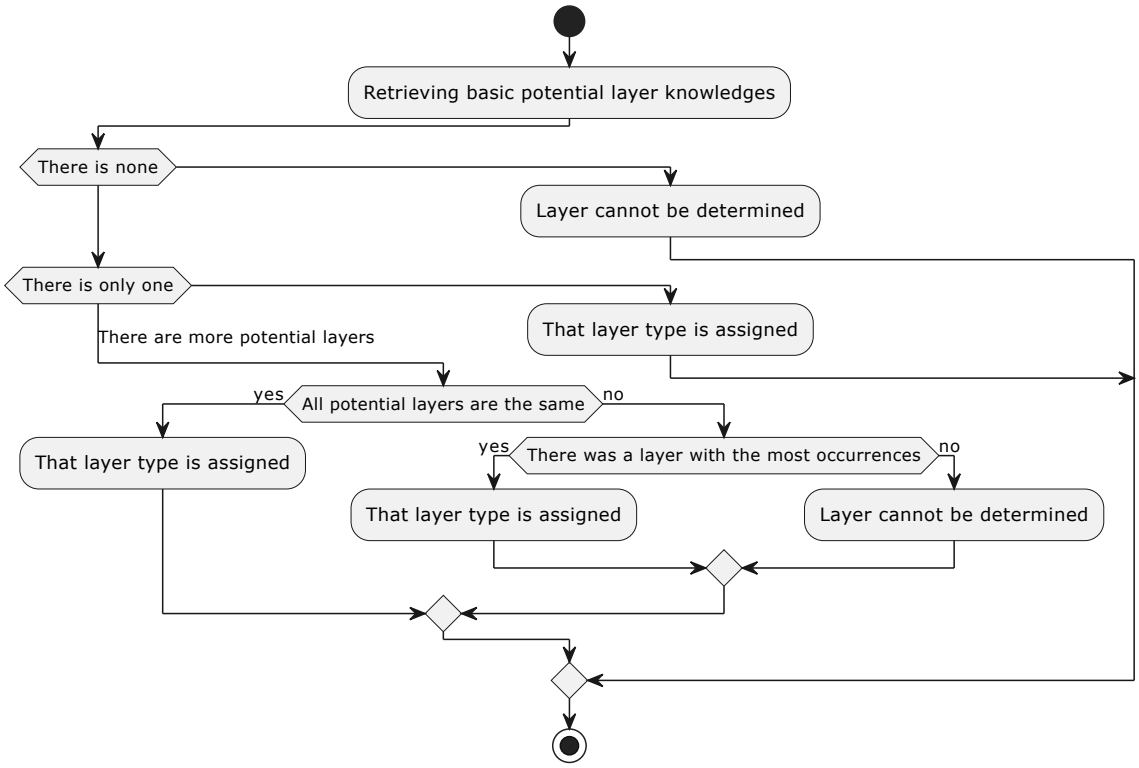


Figure 3.10: Activity diagram of the process of assigning layers to types based on primary knowledges.

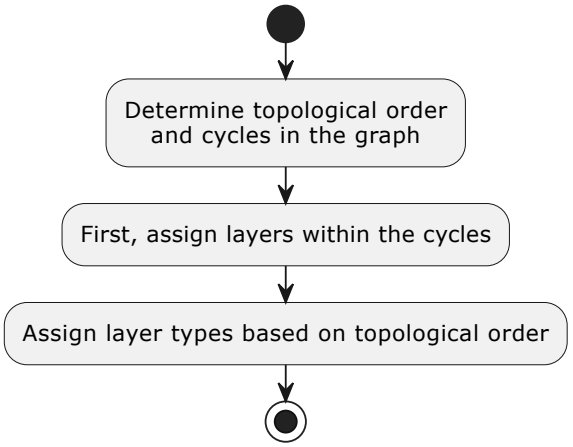


Figure 3.11: Activity diagram of the process of distributing layers along the edges of the type graph.

### 3.6.2 Related knowledges

In order to provide a technique of identifying architectural layers of types, that fits into the proposed analysis model, a particular knowledge that holds information about the potential layer of a type is introduced, so that later, these knowledges can be registered into *IKnowledgeHolder* implementations such as *TypeHolder* and even *TypeCollection*.

To be more specific, a number of types implementing the *IKnowledge* interface are defined. These will have a common abstract base class of *PotentialLayerKnowledge* which holds information about the used heuristic that produced this assumption (of a type belonging to a specific layer). Moreover, it was a targeted goal in Section 2.9 to give students more insight regarding the thought process of a lecturer while evaluating a student submission. In order to ensure that, a *Reason* string property will be present at each *PotentialLayerKnowledge*.

Besides this, another abstract type in the form of *SpecificPotentialLayerKnowledge* is defined that is derived from the previous and holds the specific *LayerType* that the knowledge refers to. It has two implementations which are not abstract: *BasicPotentialLayerKnowledge* and *NameBasedPotentialLayerKnowledge*. (They contain the same members, but are not meant to represent the same kind of knowledge.)

Derived classes of *PotentialLayerKnowledge* also include *ConnectedPotentialLayerKnowledge* that is used to represent the case when a certain type's architectural layer depends on the layer of another type. This cannot have information about the layer of a type, but can only contain a reference to the connected type as member, as the layer of that type might not be known at the time this connection is discovered.

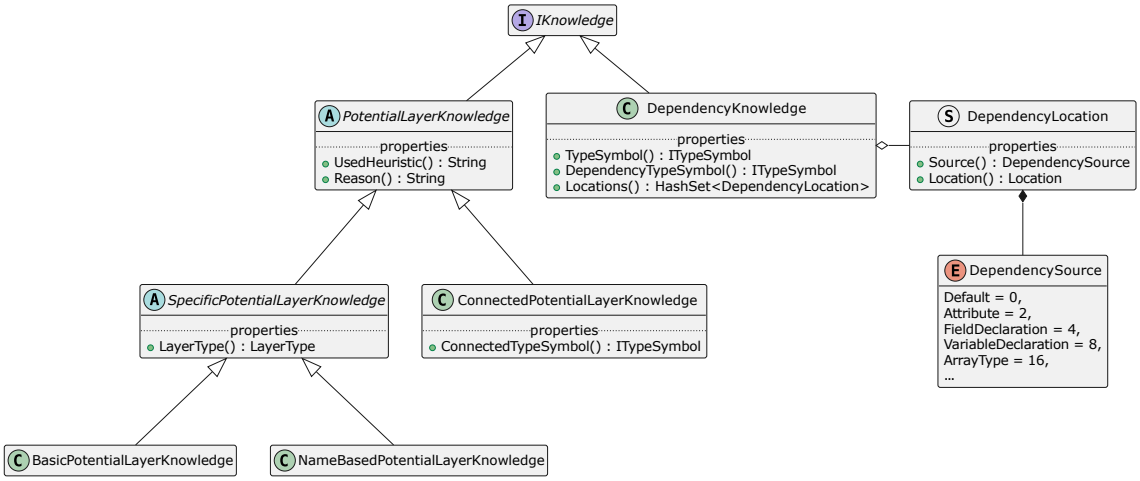


Figure 3.12: Class diagram of *IKnowledge* related types.

### 3.6.3 Heuristics

In order to be able to specify and categorize heuristics, a number of heuristic interfaces were introduced. Figure 3.13 shows these interfaces. A common interface of *IHeuristic* was identified with the sole purpose of registering knowledges

to the types of a *TypeCollection* containing the potential architectural layer that the type belongs to (and the reason of this assumption). Besides this, two more kinds of heuristic interfaces were defined to categorize specific heuristics in the future. These two interfaces are *INameBasedHeuristic* and *ITypeConnectionHeuristics*. The specialty of these two is that they register special knowledges to the *TypeHolder*s. The former *NameBasedPotentialLayerKnowledge*, representing the potential layer of a type determined by a name (name of a type or name of a namespace); the latter *ConnectedPotentialLayerKnowledge*, representing the dependency relation in regard to the potential architectural layer of another type.

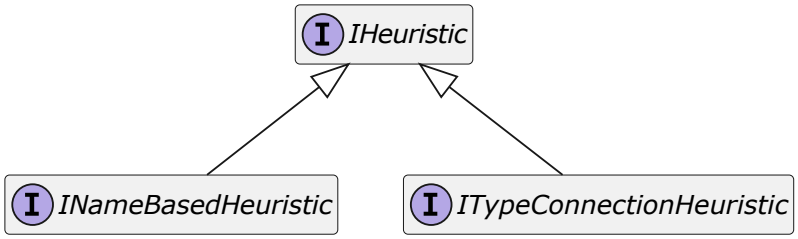


Figure 3.13: Heuristic interfaces and the relationships between them.

#### BaseTypeHeuristic

This heuristic derives from *IHeuristic*. Its purpose is to register *BasicPotentialLayerKnowledge*s into *TypeHolder*s if their type inherits from a base type that implies a specific layer. The map of layers and their connected base types are listed in the configuration of the tool (see subsection 3.1.2).

#### AttributeHeuristic

This heuristic derives from *IHeuristic*. Its purpose is to register *BasicPotentialLayerKnowledge*s into *TypeHolder*s if their type or any of its members have a specific attribute that implies a specific layer. The map of layers and their connected attributes are listed in the configuration of the tool (see subsection 3.1.2).

#### AppLayerHeuristic

This heuristic derives from *IHeuristic*. Its purpose is to register *BasicPotentialLayerKnowledge*s into *TypeHolder*s if their type has a method that can be used as an entry point of the application, that is, the method is static, its name

is *Main*, it is contained in a type that is a *class* or a *struct*, it does not have any type parameters, and either has a string array parameter or no parameters at all.

**ReferencedNamespaceHeuristic**

This heuristic derives from *IHeuristic*. Its purpose is to register *BasicPotentialLayerKnowledge*s into *TypeHolder*s if their type has a dependency connection to a type that is in a namespace that implies a specific layer. The map of layers and their referenced namespaces are listed in the configuration of the tool (see subsection 3.1.2).

**NamespaceNameHeuristic**

This heuristic derives from *INameBasedHeuristic*. Its purpose is to register *NameBasedPotentialLayerKnowledge*s into *TypeCollection*s if their namespace resembles the name of a specific layer. This match is determined by *TypeNameHelper* (see subsection 3.6.3).

**TypeNameHeuristic**

This heuristic derives from *INameBasedHeuristic*. Its purpose is to register *NameBasedPotentialLayerKnowledge*s into *TypeHolder*s if their type name resembles the name of a specific layer. This match is determined by *TypeNameHelper* (see subsection 3.6.3).

This heuristic also introduces a special rule: if the name of a type starts or ends with the word *Mock*, it is assumed that it belongs to the *Test* layer.

**InterfaceHeuristic**

This heuristic derives from *ITypeConnectionHeuristic*. Its purpose is to register *ConnectedPotentialLayerKnowledge*s into *TypeHolder*s if their type inherits from another user-defined type. (That way, layer of the first type will become dependent on the type of its interface or base class.)

**EventArgsHeuristic**

This heuristic derives from *ITypeConnectionHeuristic*. Its purpose is to register *ConnectedPotentialLayerKnowledge*s into *TypeHolder*s if their type is used as event

argument at some point. This heuristic connects the layer of event arguments to the layer of types that have an event with that event argument.

#### NestedClassHeuristic

This heuristic derives from *ITypeConnectionHeuristic*. Its purpose is to register *ConnectedPotentialLayerKnowledge*s into *TypeHolder*s if their type is a nested type of another user-defined type or vice versa. This heuristic connects the layer of nested types to the layer of their containing types and vice versa.

#### ExceptionHeuristic

This heuristic derives from *ITypeConnectionHeuristic*. Its purpose is to register *ConnectedPotentialLayerKnowledge*s into *TypeHolder*s if their type is an exception and it is thrown at some point. This heuristic connects the layer of exception types to the layer of types that have a throw expression related to those exceptions.

#### TypeNameHelper

*TypeNameHelper* is not a heuristics, but a helper class used by some heuristics, which contains a method that can assign a layer to the given name of a type or a namespace.[5]

The method should first check if the given parameter equals to the name of any architectural layer. If there is a match, that layer should be returned. After that, it should be checked whether the given parameter contains the name of any layers. If it contains exactly one, it should be returned. If after the previous steps there were no results to be returned, the given parameter should be checked against the name of each layer using some kind of a distance metric, for instance the Levenshtein distance [42]. The purpose of that is to make sure that any spelling mistakes would not influence this kind of layer identification (see subsection 2.3.1). If there is a minimal distance to a layer, that should be returned. If all the previous steps fail to return a layer, then *CannotBeDetermined* should be returned.

Although, the use of a distance metric can be useful during layer identification, if the minimal distance to a layer's name is large enough, the identification strategy can be quite inefficient. Therefore, we suppose the use of an upper limit, such as two.

[5]Not fully qualified name.

During this process, layers of *CannotBeDetermined* and *App* should be excluded as the former is not a practical architectural layer, and the latter would highly influence the results (for instance type names ending with the word *Map* would most certainly end up sorted into the wrong - *App* - layer). Also, while comparing strings, ignoring case seems justified.

### 3.6.4 Example for layer identification

This subsection provides an example regarding the proposed layer identification process. The abstract Sudoku game from a previous example shown on Figure 3.5 will serve as a base for this example as well.

As a first step, the rules of the registered heuristics are evaluated. Based on typical base types and used attributes, the following knowledges could be determined:

1. *MainWindow* belongs to the *View* layer, as it inherits from *System.Windows.Window*,
2. *ViewModelBase* belongs to the *ViewModel* layer, as it inherits from *System.ComponentModel.INotifyPropertyChanged*,
3. *MainViewModel* belongs to the *ViewModel* layer, as it inherits from *System.ComponentModel.INotifyPropertyChanged*,
4. *CustomValueConverter* belongs to the *ViewModel* layer, as it is annotated with the attribute *System.Windows.Data.ValueConversionAttribute*,
5. the *TypeCollection* representing the *ValueConverters* folder is assigned the *ViewModel* layer, as its closest ancestor has that as well,
6. other *TypeCollection*s are assigned the layer corresponding to the name of their represented folders,
7. *ViewModelBase* belongs to the *ViewModel* layer, as its name contains "ViewModel" and does not contain the name of any other layer[6],
8. *MainViewModel* belongs to the *ViewModel* layer, as its name contains "ViewModel" and does not contain the name of any other layer,
9. *IGameModel* belongs to the *Model* layer, as its name contains "Model" and does not contain the name of any other layer,
10. *GameModel* belongs to the *Model* layer, as its name contains "Model" and does not contain the name of any other layer,

[6]Layer names containing each other are excluded in these cases (e.q. View and ViewModel).

11. there is a connection between the layers of *ViewModelBase* and *MainViewModel*,
12. there is a connection between the layers of *IGameModel* and *GameModel*.

The next step is the evaluation of the results of the heuristics. At first, all types are assigned *CannotBeDetermined*. Then, the *TypeCollection* representing the *ValueConverters* folder is assigned the *ViewModel* layer, as its closest ancestor has that as well, while all other *TypeCollection*s are assigned the layer corresponding to the name of their represented folders. After these, based on 1-4, layers of the four related types can be assigned obviously. At this point, the type graph is built, and layers are tried to be distributed along its edges (with no effect in this case). Since after the previous steps there are two types without a specific layer (*IGameModel* and *GameModel*), these types are assigned layers based on their names (that would be the *Model* layer in both cases).

As of this point, the layer of all types has been identified, further actions (even those of special rules) will have no effect on this example set. Finally, it should be mentioned that there were no inconsistencies regarding this example.

## 3.7 Defined rules

Based on the previous sections and subsection 2.3.1, the following cases are expected to be recognized during architectural analysis:

- when the architectural layer of a user-defined type could not be determined during the layer identification process,
- when inconsistencies were found during the layer identification process,
- when the submission lacks required layers of the examined architecture,
- when invalid dependencies were found between layers,
- when method calls were found between unrelated layers,
- when the handling of an event was not done in appropriate layers,
- when an object leaks its representation,
- when an object has a member that might leak its representation,
- when a class depends on a concretion (but could depend on an abstraction instead),
- when event handlers are defined in files with *xaml.cs* extension.

## 3.8 Reported diagnostics

In this section, the defined rules are introduced in the form of their related diagnostics, their details are explained and examples regarding them are given, along with counterexamples if possible.

Table 3.1 lists the defined diagnostics with be their identifiers and titles. According to Microsoft, identifiers of diagnostics should be of form <*PREFIX*><*number*> where the prefix should be longer in order to avoid conflicting identifiers [43]. Therefore, identifiers of the defined diagnostics will have the prefix of *ARCH* and their numbers will start from 1000 and then increase.

The first group of diagnostics (*ARCH1000-ARCH1004*) are connected to architectural layer clustering and basic architectural rules; the second group (*ARCH1006-ARCH1007*) is associated with representation leak detection, while the remaining diagnostics come from bad practices that are considered relatively smaller violations during the related university course. The following subsections will detail these diagnostics.

| **Diagnostic id** | **Diagnostic title** |
|---|---|
| *ARCH1000* | Layer cannot be determined |
| *ARCH1001* | Inconsistency during layer identification |
| *ARCH1002* | Missing required layer |
| *ARCH1003* | Invalid dependency between layers |
| *ARCH1004* | Method call between unrelated layers |
| *ARCH1005* | Events should be handled in appropriate layers |
| *ARCH1006* | Representation leak |
| *ARCH1007* | Possible representation leak |
| *ARCH1008* | Class depends on concretion |
| *ARCH1009* | Don't define event handlers in xaml.cs files |

Table 3.1: Reported diagnostics by their identifiers and titles.

### 3.8.1 ARCH1000 - Layer cannot be determined

In Section 3.6, the layer clustering technique of user-defined types to architectural layers states and ensures that if the corresponding architectural layer of a type cannot be determined, it will be assigned a special layer type (*CannotBeDetermined*). It is considered important to notify students of such cases as these types might look like that they are not part of the current solution, and their placement in the actual C# solution next to other types becomes questionable.

Furthermore, these types can pose threats in the flow of checking architectural violations that is yet to come (right after layer clustering). It is possible that valuable diagnostics cannot be reported as the layer of the type could not be determined, which is a problem.

In order to prevent the possibility of such cases, the extended part of layer clustering was made configurable, and that basic configuration gives students the chance to annotate their types with the built-in C# *DescriptionAttribute* (its parameter must be the name of the desired layer). That way, if students explicitly express their intention to place a type into a certain layer, the efficiency and accuracy of the reported diagnostics can be maximized.

### 3.8.2 ARCH1001 - Inconsistency during layer identification

It is possible that the clustering algorithm was not able to classify a type as a part of any specific architectural layer, as there were conflicts from the extracted data regarding which layer should contain that type. It is considered important to notify students of such cases as not being sure of the architectural layer of a user-defined type can seriously influence the future flow of architectural violation checking. Therefore, cases when such inconsistencies are found regarding a type, a diagnostic should be reported at the location of the type definition, while the description of the diagnostic should contain all the potential layers and the reasons why it is assumed that the type belongs to those layers.

#### Example

A class named *GameViewModel* which implements the *INotifyPropertyChanged* interface has a method that performs file operations that require the use of the *System.IO* namespace. This class should be sorted into the *ViewModel* layer based on its name and based on the interface it implements, however, it could also be sorted into the *Persistence* layer since it uses *System.IO*. This conflict should be reported in the form of a diagnostic at the location where *GameViewModel* is defined.

### 3.8.3 ARCH1002 - Missing required layer

Subsection 3.3.6 describes that architectures may have required layers that must be present in a submission, therefore, in cases when the analyzed submission lacks such a required architectural layer, a diagnostic should be reported. This diagnostic should

be reported without a specific location in source files as it describes a property of the analyzed submission itself and is not related to any specific part of the implementation, but to the solution as a whole.

**Example**

A submission that is being analyzed against MVVM architecture does not contain any types that are part of the *ViewModel* layer, therefore, it lacks the complete *ViewModel* layer which is considered as an architectural violation.

### 3.8.4 ARCH1003 - Invalid dependency between layers

This diagnostic should be reported in case of invalid dependencies between layers, that is, when the determined dependencies between user-defined types imply that these dependencies violate the dependency rules of the examined architecture. This diagnostic is related to types, and therefore should be reported at type definitions, and their description should contain the types that cause the invalid dependencies.

### 3.8.5 ARCH1004 - Method call between unrelated layers

If a method call is present in a user-defined type which is part of the architectural layer of $A$, and the called method is in a type which is part of the architectural layer of $B$, such that $A$ and $B$ violates the dependency rules of the analyzed architecture (see subsection 3.3.6), that method call is between unrelated layers, and therefore should be reported. ($A$ and $B$ comes from the set of possible layers of the analyzed architecture.) The diagnostic should be reported at the location of the method call and its description should contain the two disconnected architectural layers.

### 3.8.6 ARCH1005 - Events should be handled in appropriate layers

This diagnostic should be reported when the raising and handling of an event happens in two disconnected architectural layers, at the location of the event handlers.

### 3.8.7 ARCH1006 - Representation leak

This diagnostic should be reported when from the type definition, it is obvious that instances of that type will be suspect to representation leaks. The following cases certainly imply the possibility of a leaking representation and therefore should be handled:

- if the type has a public field[7],
- if a member field is directly returned, while its type is not immutable.

Some remarks regarding the previous cases:

- Using these rules, only types in the *Persistence* and *Model* layers should be analyzed, as in the case of other layers, encapsulation might be slightly differently interpreted.
- Static and const fields should be ignored, as either these cannot be considered as part of the representation of an instance or their value cannot be changed.
- During analysis, a type can be considered immutable if it is a string [45], a record type [46] or a value type [47] in general. (Types in namespace *System.Collections.Immutable* could also be listed, although those types are not especially usual in the analyzed submissions.)

### 3.8.8 ARCH1007 - Possible representation leak

This diagnostic should be reported for types that have a type member, for which *ARCH1006* was reported, because their representation is leaked, and therefore their usage could result in the containing type leaking its own representation, hence using them should be avoided.

### 3.8.9 ARCH1008 - Class depends on concretion

This diagnostic should be reported when a user-defined type has a constructor with a parameter that has the type of another user-defined type. If the type of that parameter is not abstract (not an abstract class or an interface) and has an abstract base class or interface, the original type depends on a concretion, when it could depend on an abstraction. If reported, the location of the diagnostic should be the constructor, so that this diagnostic could be reported for more constructors of the same class.

[7]This case is already reported by the *S1104* diagnostic of Sonar.Analyzers [44], however *ARCH1006* wishes to identify representation leaks even without the use of external analyzers.

### 3.8.10 ARCH1009 - Don't define event handlers in xaml.cs files

This diagnostic should be reported in case of event handlers defined in files with *xaml.cs* extension and contained within a class representing a Window (inheriting from *System.Windows.Window*), at the location of the event handlers.

## 3.9 Proposed analyzers

Based on the previously defined custom diagnostics, a number of analyzers are proposed with the description of their tasks. Simply, the basic task of these analyzers is to register actions that collect data during the data extraction phase, and actions that perform violation checks on the collected data and report related diagnostics in the diagnostic report phase. Analyzers are usually defined in relation to a specific diagnostic. The overall structure of the proposed analyzers is shown on Figure 3.14.

It is emphasized once more, that during the implementation of these analyzers, the use of multi-threaded tools is suggested to ensure thread-safety during concurrent execution of the analysis.

Also, most diagnostics require certain knowledge regarding the architectural layer of the related types when they are reported. Therefore, diagnostics should not be reported if the related types had inconsistencies during the layer identification process.

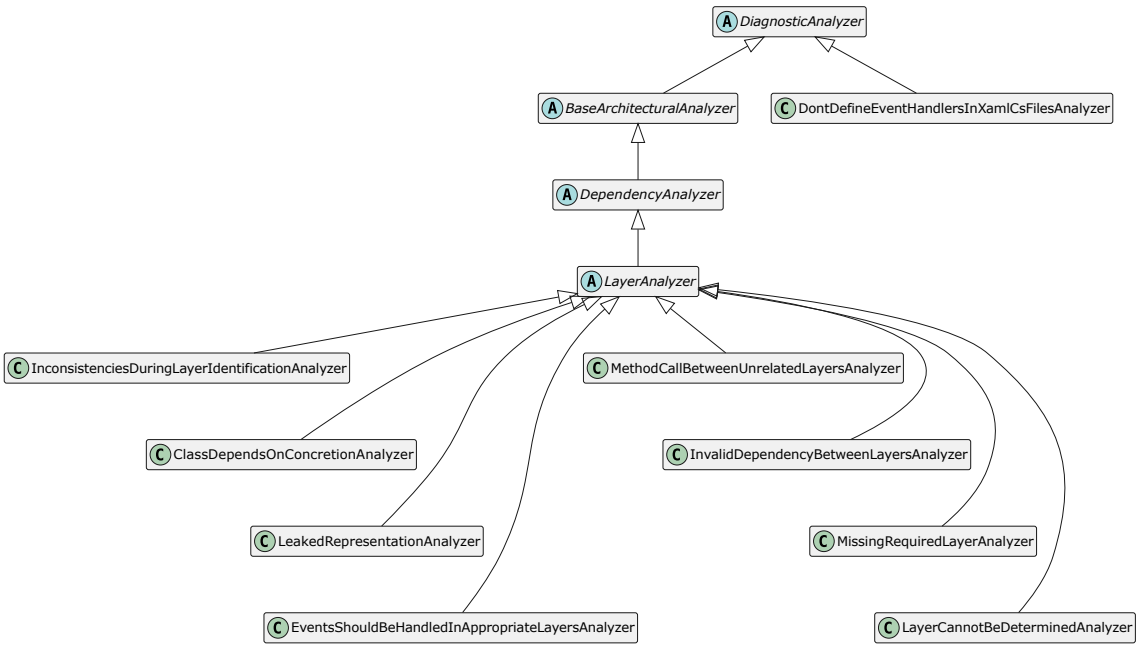


Figure 3.14: Class diagram of analyzer classes.

### 3.9.1 InconsistenciesDuringLayerIdentificationAnalyzer

After the layer identification process, this analyzer should collect inconsistencies of the process, and report them at the location of type declarations, mentioning the potential layers in the diagnostic's description, while also including the reason of layer assumptions.

### 3.9.2 LayerCannotBeDeterminedAnalyzer

After the layer identification process, this analyzer should collect the cases when the architectural layer of a type could not be determined, and report them.

### 3.9.3 MissingRequiredLayerAnalyzer

After the layer identification process, this analyzer should filter for cases when required architectural layers of the examined architecture are missing, and report them. However, these cases should only be reported once for a solution, without a specific location in source.

### 3.9.4 InvalidDependencyBetweenLayersAnalyzer

After the layer identification process, as not only the results of clusterization, but also the dependency connections between architectural layers are known, this analyzer should filter for the dependencies that violate the ruleset of the examined architecture. The related diagnostics should be reported for type declarations, containing that type's dependencies grouped by their containing layer.

### 3.9.5 MethodCallBetweenUnrelatedLayersAnalyzer

This analyzer should visit invocations in an analyzed type's AST. From these invocations (and from information provided by the related semantic model), type of the caller and type containing the called method can be recovered. After the layer identification process, the architectural layer of these recovered types can be queried, and if these layers violate the ruleset of the examined architecture, diagnostics should be reported at the location of the invocation.

### 3.9.6 EventsShouldBeHandledInAppropriateLayersAnalyzer

This analyzer should visit assignment expressions in an analyzed type's AST. From these assignment expressions (and from information provided by the related semantic model), type of the assignment expression and type of the right side of the assignment can be recovered. From these cases, assignments must be filtered: the kind of the assignments must be *MethodKind.EventAdd*, which corresponds to event handler assigments. During the diagnostic report phase, layers connected to the remaining event handler assignments should be queried for the layer of the type containing the event symbol and for the layer of the type containing the event handler itself. If these two violate the ruleset of the examined architecture, diagnostics should be reported at the location of the event handler assignment.

### 3.9.7 LeakedRepresentationAnalyzer

During the data extraction phase, this analyzer must visit return statements, and mark those of them which return a non-static and non-const member field of the analyzed type. (Although, it should be checked whether there is another variable with the same name declared within the same scope.) Also, non-static and non-const fields must be collected.

During the diagnostic report phase, the marked return statements should be reported if their containing type is part of the Persistence or Model layer. Besides these, the collected non-static and non-const fields should also be reported.

Furthermore, this analyzer is unique in a sense, as unlike the others it can report multiple diagnostics. The previous paragraphs described the report process for *ARCH1006* diagnostics, however the rule of *ARCH1007* requires knowledge about types leaking their representation, therefore it seems practical to let this analyzer report these diagnostics as well.

Accordingly, during the diagnostic report phase, when the set of types leaking their representation is already known, the set of all user-defined types must be traversed and if any of them has a member with a representation leaker type, an *ARCH1007* diagnostic should be reported at the original type's declaration. The description of the diagnostic in such a case is ought contain the names of the related potentially representation leaking fields.

### 3.9.8 ClassDependsOnConcretionAnalyzer

During the diagnostic report phase, this analyzer should determine the set of concrete types which have an abstract type or an interface they implement. After then, constructors of all user-defined types must be observed. If a constructor has a parameter with a concrete type, a diagnostic should be reported at the location of the constructor.

### 3.9.9 DontDefineEventHandlersInXamlCsFilesAnalyzer

This analyzer is special, as it does not inherit from any special analyzer base class, just the default *DiagnosticAnalyzer* abstract class. It analyzes method symbols: if a method is an event handler, with a location within a file with *xaml.cs* extension and its containing type inherits from *System.Windows.Window*, a diagnostic should be reported at its location.

The characteristics of an event handler method are the following: it returns *void*, has two parameters and its first parameter is *object*.

## 3.10 Integration with evaluator system

Section 2.5 introduces and describes *TMS*, its workflow and mentions that it is configurable. Subsection 3.1.1 lists the requirements of the proposed tool. These requirements include that the output of the proposed tool should be compatible with the *xml* output generated by Roslynator. This is to make its integration with TMS easier (as it already supports the output of Roslynator).

In order to integrate the proposed tool, TMS can easily be configured. Only a Docker image needs to be created, along with a script that runs the analysis. The Dockerfile used to build the image should use the already existing evaluator image that is used to analyze solutions with Roslynator. On top of that, the new image must contain the implementation of the architectural analyzer tool (supposedly through an installation from a NuGet package registry) and the shell script running the analysis itself.

The mentioned shell script must first run the original static code analysis using Roslynator, since the results of it and that of the architectural analyzer should be displayed alongside each other at the end. (The *xml* result of this step must be saved.) After then, the architectural analyzer tool must be invoked on the submission, generating an *xml* output of its results.

Since the results of both analyzers resemble the same structure, they can and should be merged into a single *xml* file. This ultimate *xml* file then contains the results of both analyses and since the original output files shared a common schema, this merged version will also be valid by that schema. Therefore, its structure is appropriate and can be processed by CodeChecker to display data to the users of TMS. The script or application that is able to merge the output *xml* files must also be included in the previously described Docker image.

# Chapter 4

# Results and validation

In order to assess the proposed methods, an implementation of the proposed tool performing architectural analysis was created and was integrated with TMS. Using the tool, student submissions from past semesters were evaluated.[1] In this chapter, results of this evaluation are presented, and in the end, the initial research questions are addressed.

The evaluation of student submissions is two-sided. First, results of the proposed clustering method should be evaluated, and then the submissions themselves.

## 4.1 Evaluating the clustering method

In this section, we aim to present the results of evaluating the type clustering method through the analysis of student submissions. The goal of this evaluation is to calculate the percentage of the correctly clustered types in regard to all types present in the examined submissions.

Accordingly, a metric needs to be introduced, therefore, as a first step of this evaluation, *accuracy* is defined as the ratio of the number of types that were assigned their correct layer and the total number of types. In order to calculate the *accuracy* of a submission or the set of all submissions, the layer clustering of all examined student submissions were created manually to serve as a ground truth.

This manual determination of the architectural layer of user-defined types were done considering the lecturer 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 consequent regarding cases

[1] Along with this thesis, a sample submission is provided to demonstrate most of the reported diagnostics through an example.

resembling similar ambiguous cases. If there was no clear indicator helping the identification of a type's layer, the type was placed into a special layer (*CannotBeDetermined*), which contains types without a specific layer.

During evaluation, 947 student submissions were examined. The related C# solutions of these submissions contained a total number of 13126 user-defined types. These types had to be clustered both manually and with the use of the created tool into the seven defined architectural layers (*Persistence*, *Model*, *ViewModel*, *View*, *App*, *Test* and *CannotBeDetermined*).

Inspecting the extended 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 clusterization of types. This comparison showed that out of the total number of 13126 user-defined types, the architectural layer of 12900 types were identified correctly, while 226 incorrectly. Therefore, the *accuracy* of the layer identification process on the set of all student submissions is 0.9828 and the percentage of layer identification errors is 1.72%.

The comparison of the expected and actual identified architectural layers also showed that out of all submissions, 108 contained layer identification errors, that is 11.4% of all submissions. However, it is also important to note, that this number contains submissions even if they contain just one type with incorrectly identified layer.

Figure 4.1 shows the distribution of the 226 layer identification errors in regard to the 108 submissions containing these errors. It can be seen that in some cases the number of errors in a submission reached a number between 5 and 15, however it happened only in a negligible number of cases. The average number of errors per solution was around 2.

It is also important to mention, that in the case of the 226 incorrectly clustered types, the layer of 65 types could not be determined, while in the case of 63 other types, inconsistencies were indicated. Therefore, only 98 types were clustered incorrectly without providing any information to the user - which is less than 1% of all types.

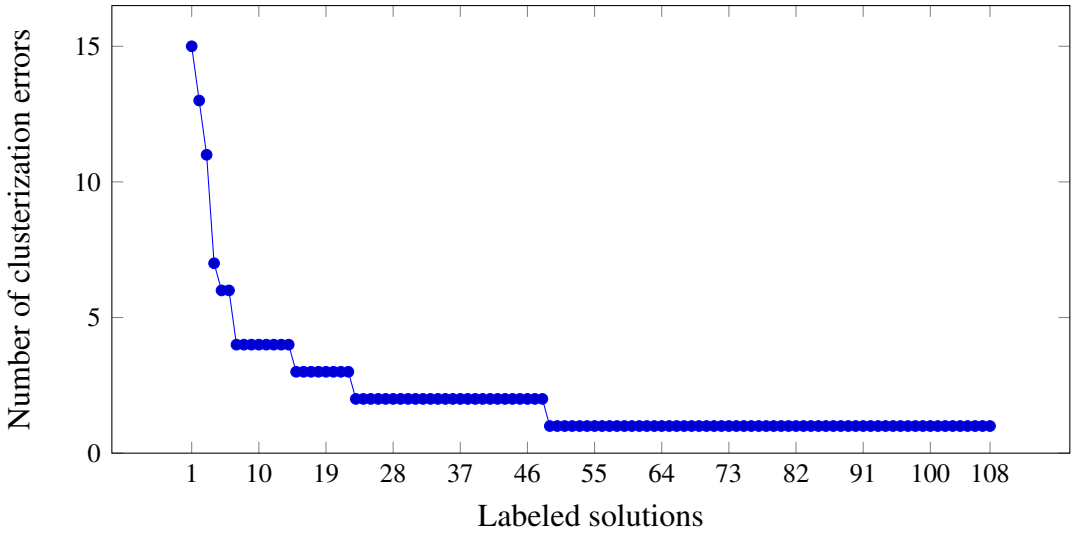


Figure 4.1: Distribution of clusterization errors in submissions.

## 4.2 Evaluating student submissions

The second part of the evaluation is evaluating the diagnostics collected during the architectural analysis of student submissions. The number of each diagnostic reported for all submissions is shown on Figure 4.2.

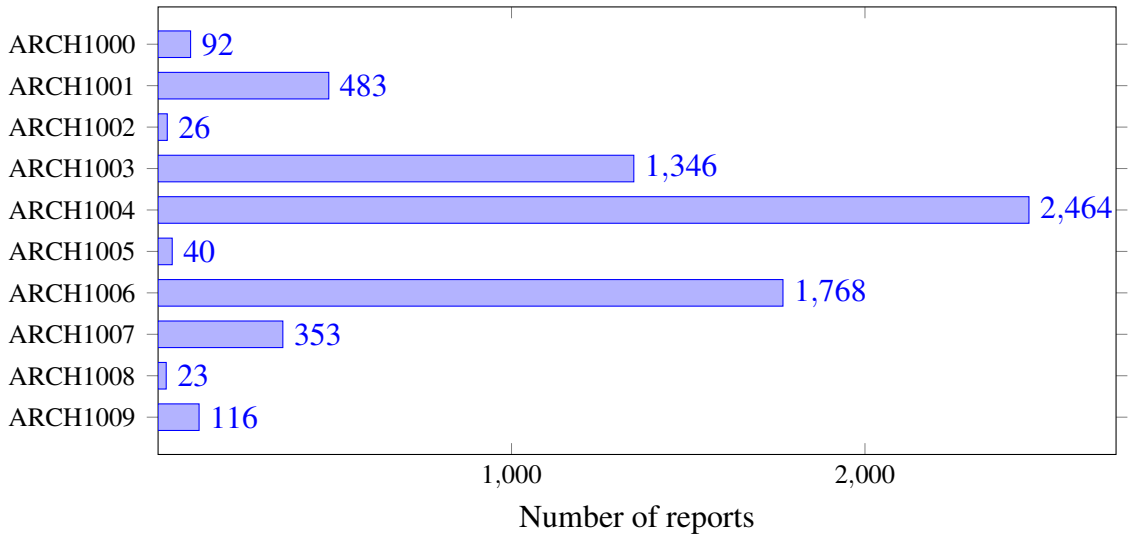


Figure 4.2: Diagnostics with the number of reported cases.

It can be seen that the most occuring diagnostic was *ARCH1004* which reports method calls between unrelated layers. However, these diagnostics can be partitioned further as students were not explicitly prohibited to call 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 904 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 occuring diagnostic can be related to the violation of encapsulation, concerning the representation leak of objects. This diagnostic was reported more than 1700 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 608, when public fields were defined within a type. The other cause of such a diagnostic could be a return statement returning a mutable member field of a type, which happened in 1160 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 on Figure 4.3. Table 4.1 displays the same results, extended with the percentage of diagnostic reports compared to the number of all solutions and source files[2].

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 more than half (53,75%) of all submissions, while affecting more than 10% of all source files.

Another noteworthy case is connected to representation leak reports (*ARCH1006*). It is shown that the issue corresponding to the diagnostic is present in almost half (48,36%) of all submissions, and is distributed among 9% of all source files.

The case of *ARCH1007* (possible representation leak) diagnostic reports is remarkable in a sense, as its percentage was over 20%, and it was reported for exactly 353 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 in nearly over the quarter of all submissions (27,03%), and is present in a mere number of 3,34% 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 consequently.

As far as layer determination failures (*ARCH1000*) and layer identification inconsis-

[2]Since in multiple cases more than one type was placed into a single source file, the previously mentioned 13126 types are distributed in 12457 source files.

tencies (*ARCH1001*) are concerned, their number is significant as these diagnostics could report information about 57% of all layer identification errors (see Section 4.1).

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

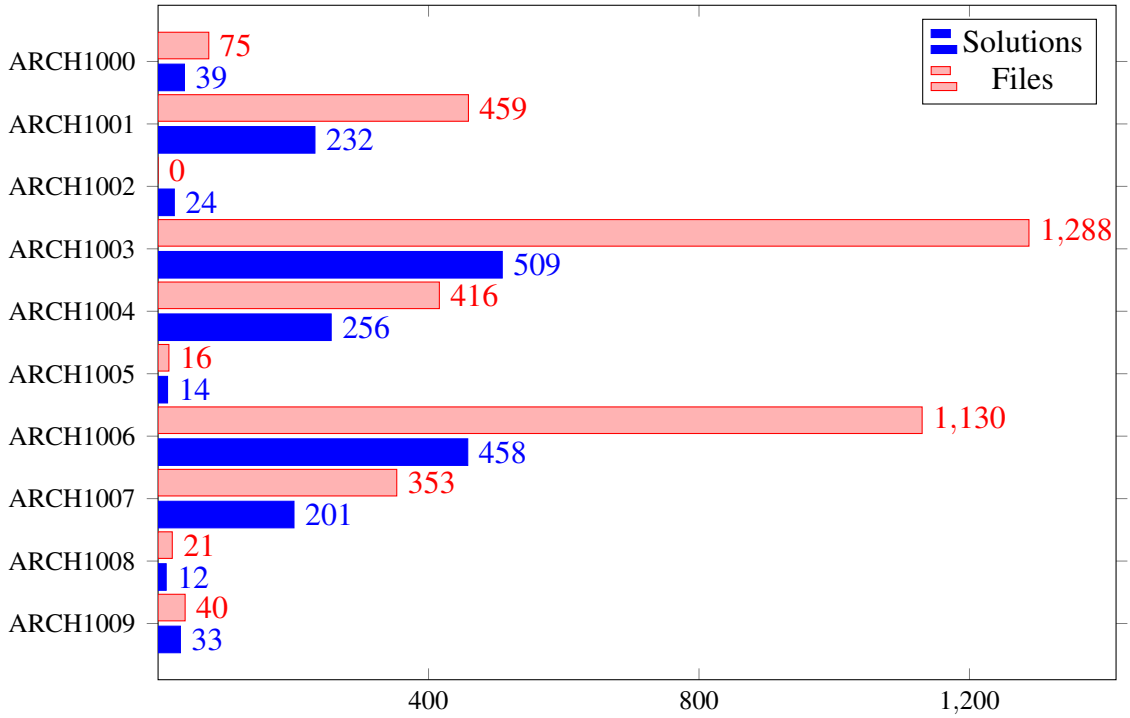


Figure 4.3: Diagnostics with the number of occurrences in solutions and source files.

| **Diagnostic id** | **Solutions** | | **Source files** | |
|---|---|---|---|---|
| | Number | Percentage | Number | Percentage |
| *ARCH1000* | 39 | 4,12 % | 75 | 0,6 % |
| *ARCH1001* | 232 | 24,5 % | 459 | 3,68 % |
| *ARCH1002* | 24 | 2,53 % | - | - |
| *ARCH1003* | 509 | 53,75 % | 1288 | 10,34 % |
| *ARCH1004* | 256 | 27,03 % | 416 | 3,34 % |
| *ARCH1005* | 14 | 1,48 % | 16 | 0,13 % |
| *ARCH1006* | 458 | 48,36 % | 1130 | 9,07 % |
| *ARCH1007* | 201 | 21,22 % | 353 | 2,83 % |
| *ARCH1008* | 12 | 1,27 % | 21 | 0,17 % |
| *ARCH1009* | 33 | 3,48 % | 40 | 0,32 % |

Table 4.1: Number of reported diagnostics in the analyzed submissions and their source files, extended with their percentage compared to all solutions/files.

### 4.2.1 Comparison

It was a natural requirement that diagnostics should not be reported when the related violations were in connection with a type for which its architectural layer could not be determined precisely (for instance if it had inconsistencies during the layer identification process). In order to assess this, student submissions were modified, and all user-defined types were explicitly applied their expected architectural layer through the use of C# attributes. (This was done programatically using the manually created dataset that mapped types to their expected architectural layer - see Section 4.1.) These modified submissions were analyzed once more with the architectural analyzer tool. The results of this analysis are shown on Figure 4.4. 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.

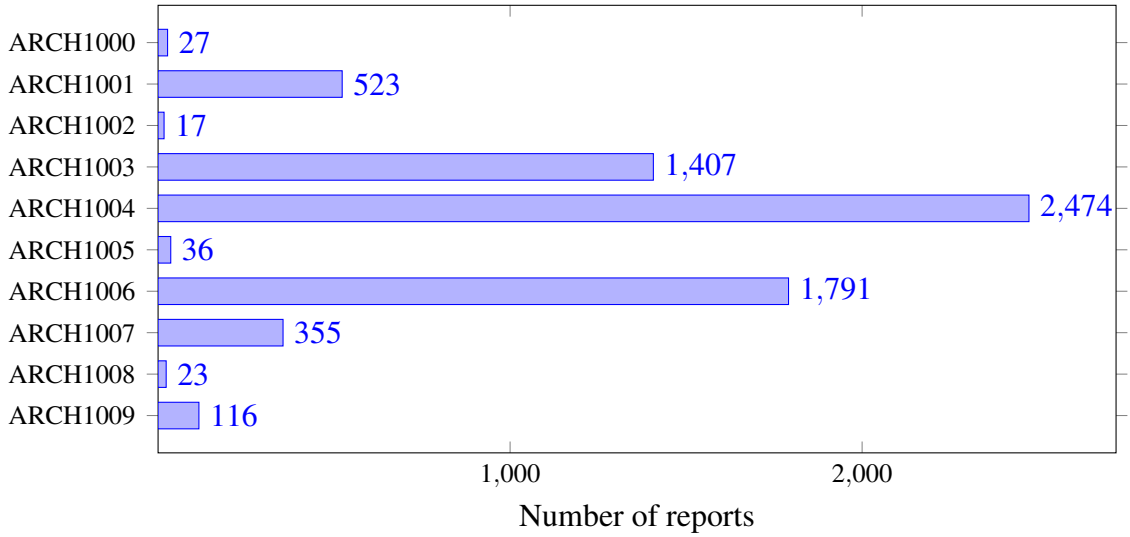


Figure 4.4: Diagnostics with the number of expected reported cases.

Comparing these results with the results of the previous analysis, it can be seen that most of the expected numbers are higher than the related number of actual reported diagnostics. In fact, this is the justified and expected outcome, as more diagnostics can be reported reliably when the architectural layer of the related types are certainly known at the time of the analysis. This explanation is appropriate for some diagnostics, while others require further clarification.

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.

Also, the number of *ARCH1001* diagnostic reports increased during the second analysis. This is due to the fact that more inconsistencies may arise as the explicit layers are provided alongside with the other information extracted from the source code. Similarly, the number of expected *ARCH1003* and *ARCH1004* reports are higher, as the higher number of correctly identified layers provides a basis for potentially more diagnostics. However, the number of *ARCH1002*, *ARCH1005* and *ARCH1006* reports decreased because of the previous explanation.

As it was 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 filtering applied to the expected results yielded 929 diagnostics.

## 4.3 Addressing the research questions

This section sumarizes the evaluation results, while addressing the research questions.

**RQ1. How does layer clustering based on heuristics perform on the dataset consisting of submissions from past semesters? How accurate is it?**

Section 4.1 explained the details of the evaluation of the layer clusterization 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 clusterization. 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 57% of the remaining cases, diagnostics were reported regarding layer determination failures and clusterization inconsistencies.

**RQ2. How error-prone is the proposed layer clustering method? What are the edge cases of it? Under what circumstances is it most likely to fail?**

It was shown that the layer clustering method performs well on the user-defined types in the analyzed student submissions, because of the heuristics it uses. However, there are

some cases which show the limitation of the used technique. For instance, errors could be provoked by avoiding the use of namespaces for the architectural layers; by placing types outside all namespaces; by giving types and namespaces unrelated names; by intentionally causing layer identification inconsistencies; by making types depend on each other causing circular dependencies, and so on. Generally, by trying to sabotage each step of the layer identification process may cause issues.

**RQ3. What are the most common issues of the analyzed submissions?**

It was concluded that the most common issues of the analyzed student submissions are 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.

**RQ4. How can the number of false positive cases of the reported diagnostics be minimized?**

During the specification of diagnostics and analyzers, a number of measures have been taken to reduce the number of false positive cases. For instance, 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. This way, diagnostics cannot be reported for types if their architectural layer could not be determined with absolute certainty. Results of this safety measure can be seen when the analysis results are compared to expected analysis results with explicitly set correct layers, as it was described in subsection 4.2.1.

**RQ5. What conclusions can be drawn from the evaluation of student submissions from past semesters?**

It can be concluded that the architectural analysis of student submissions gave valuable insights regarding student 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 happened 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 actions from 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 to 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 must be emphasized more during lectures.

**RQ6. How can the revealed issues be useful from an educational methodology perspective?**

The revealed issues could and should be drawn to the attention of the lecturers of the related course, who should then estimate how the most common issues could be highlighted and emphasized during future semesters. Also, experiences of this analysis could be taken into account when it comes to the modification of the currently existing student assignments and the creation of new ones.

# Chapter 5

# Conclusion and discussion

This chapter summarizes the achievements of the conducted research and identifies connected areas for potential future research.

First of all, before the research, the demand was identified to further increase the level of automatic analysis of student submissions related to the course of Event-driven applications with analysis from an architectural viewpoint. Therefore, the possibilities of architectural and design pattern violation detection were explored. It was gathered how others approached these problems, and it was deduced that for such kind of an analysis, an architectural breakdown of the examined student submission must be available before the actual violation checks can be evaluated.

Moreover, it was also shown that the default analysis workflow provided by the Roslyn APIs was not suitable for the purposes of architectural analysis, 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 tool 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 analysing 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 tool was expected to indicate inconsistencies of the layer identification. That is, when a type could be assigned to more layers, the fact of conflicting layers should be communicated to the user, along with the reasons why the inconsistency could happen in the first place.

For these reasons, the proposed clustering method uses a combined approach from previous researches. The method is deterministic and automatic, meaning that 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. The rules of these heuristics usually came from the rules of the used SDK, the rules of the examined university course and some were based on observations regarding the structure of student submissions.

The method is automatic, however, it is important to mention that it enables the explicit assignment of types to architectural layers, mainly for testing purposes, but this feature can even be leveraged by users to ensure the required layer resolution of types during analysis.

After the recovery of information related to user-defined types, the discovery of the dependencies and connections between them, and the clusterization 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* 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 otherwise could be overlooked during manual evaluation. The results of the layer identification process and reported diagnostics were evaluated and detailed in Chapter 4. After all, it was concluded that the architectural analysis of student submissions is truly a useful practice, which should be used for the analysis of future student assignments. 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 occurring issues, and by that, perhaps eliminate their large percentage.

Last but not least, the conducted research provided practical outputs also, in the form of the implemented architectural analyzer tool, the analyzers it uses and the tool's integration with the introduced Task Management System (TMS).

## 5.1 Future work

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

- extending functionality and the set of supported and analyzed architectures,
- extending testing scope to that of industrial applications, mainly for WPF applications using MVVM architecture,
- examining how this kind of architectural analysis could be used for other university courses,
- improving the layer identification process of user-defined types and/or extending the current configuration to do so,
- extending the rules of immutability related to representation leak detection,
- extending the representation leak detection by monitoring most (if not all) member accesses,
- extending the analyzer tool to output a visual representation that could be displayed by TMS. This way, during the evaluation of submissions, lecturers could easily check if the structure of the implemented and submitted solution conforms to the specification of the solution (as a document describing the specification of the submission is required alongside the submission).

# Acknowledgements

The related research of this paper was supported by a student research scholarship at Eötvös Loránd University Faculty of Informatics.

# Appendix A

# Tool configuration

This appendix contains a sample configuration for the proposed architectural analyzer tool.

```
{
    "DependencyAnalysis": {
        "IgnoreTypesWithAttribute": [
            "System.Runtime.CompilerServices.CompilerGeneratedAttribute",
            "Microsoft.VisualStudio.TestPlatform.TestSDKAutoGeneratedCode",
            "System.CodeDom.Compiler.GeneratedCodeAttribute",
            "System.Diagnostics.DebuggerNonUserCodeAttribute",
            "System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverageAttribute",
            "System.ComponentModel.EditorBrowsableAttribute"
        ]
    },
    "LayerIdentification": {
        "TypicalBaseTypes": {
          "Persistence": [],
          "Model": [],
          "ViewModel": [
              "System.ComponentModel.INotifyPropertyChanged",
              "System.Windows.Input.ICommand"
          ],
          "View": [
              "System.Windows.Window",
              "System.Windows.Forms.Form",
              "System.Windows.Forms.Control"
          ],
          "App": [
              "System.Windows.Application"
          ],
          "Test": [
              "Xunit.Sdk.DataAttribute"
          ]
        },
        "TypicalReferencedNamespaces": {
```

```
            "Persistence": [
                "System.IO",
                "System.Data.Entity"
            ],
            "Model": [],
            "ViewModel": [],
            "View": [],
            "App": [],
            "Test": [
                "Microsoft.VisualStudio.TestTools.UnitTesting",
                "Moq",
                "Xunit",
                "Xunit.Sdk",
                "NUnit.Framework"
            ]
        },
        "TypicalAttributes": {
            "Persistence": [
                "System.ComponentModel.DescriptionAttribute(\"Persistence\")"
            ],
            "Model": [
                "System.ComponentModel.DescriptionAttribute(\"Model\")"
            ],
            "ViewModel": [
                "System.ComponentModel.DescriptionAttribute(\"ViewModel\")",
                "System.Windows.Data.ValueConversionAttribute"
            ],
            "View": [
                "System.ComponentModel.DescriptionAttribute(\"View\")"
            ],
            "App": [
                "System.ComponentModel.DescriptionAttribute(\"App\")"
            ],
            "Test": [
                "System.ComponentModel.DescriptionAttribute(\"Test\")",
                "Microsoft.VisualStudio.TestTools.UnitTesting.TestClassAttribute",
                "Microsoft.VisualStudio.TestTools.UnitTesting.TestMethodAttribute",
                "Xunit.Fact",
                "NUnit.Framework.Test"
            ]
        }
    }
}
```

Code A.1: Sample configuration of the proposed architectural analyzer tool.

# Bibliography


[1] ELTE. *Hallgatói létszám adatok*. https://neptun.elte.hu/VcShowReport/Index/341?skey=kEK61DN0NsNzQ5BjUmDpbFAn. Accessed: 2024.05.02.

[2] Nick Anderson. "College is remade as tech majors surge and humanities dwindle". In: *The Washington Post* (May 19, 2023). URL: https://www.washingtonpost.com/education/2023/05/19/college-majors-computer-science-humanities (visited on 05/02/2024).

[3] Péter Kaszab and Máté Cserép. "Detecting Programming Flaws in Student Submissions with Static Source Code Analysis". In: *Studia Universitatis Babeş-Bolyai Informatica* 68.1 (2023), pp. 37–54. ISSN: 2065-9601. DOI: 10.24193/subbi.2023.1.03. URL: https://www.cs.ubbcluj.ro/~studia-i/journal/journal/article/view/87.

[4] Péter Kaszab. *Automated evaluation of programming assignments with static code analysis*. https://tms-elte.gitlab.io/theses/kaszab_peter_tdk.pdf. Accessed: 2024.05.02.

[5] Microsoft. *Architectural principles*. https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles. Accessed: 2024.05.02.

[6] David Garlan. "Software architecture". In: (2008).

[7] Varusai Mohamed, Shamsudeen Abubucker, and Abubucker Shaffi. "A study on Model-View-View Model (MVVM) Design Pattern". In: *International Journal of Emerging Technology and Advanced Engineering* [VOLUME 6] (July 2016), p. 264.

[8] Ian Sommerville. *Software Engineering*. 10th. Pearson, 2015. ISBN: 0133943038.

[9] Microsoft. *Desktop Guide (Windows Forms .NET)*. https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-8.0. Accessed: 2024.05.02.

[10] Microsoft. *Desktop Guide (WPF .NET)*. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0. Accessed: 2024.05.02.

[11] Microsoft. *What is .NET MAUI?* https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-8.0. Accessed: 2024.05.02.

[12] Microsoft. *What is Xamarin.Forms?* https://learn.microsoft.com/en-us/previous-versions/xamarin/get-started/what-is-xamarin-forms. Accessed: 2024.05.02.

[13] Microsoft. *Events (C# Programming Guide)*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event. Accessed: 2024.05.02.

[14] Andrew Burnett-Thompson. *Is WPF Dead? The Data Says Anything But, here's why*. https://www.scichart.com/blog/is-wpf-dead-whats-the-future-of-wpf. Accessed: 2024.05.02.

[15] Microsoft. *.NET dependency injection*. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. Accessed: 2024.05.02.

[16] Microsoft. *Data binding overview (WPF .NET)*. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-8.0. Accessed: 2024.05.02.

[17] Microsoft. *The .NET Compiler Platform SDK*. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk. Accessed: 2024.05.02.

[18] Microsoft. *Understand the .NET Compiler Platform SDK model*. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model. Accessed: 2024.05.02.

[19] Máté Cserép and Péter Kaszab. *Task Management System*. https://tms-elte.gitlab.io. Accessed: 2024.05.02.

[20] Josef Pihrt. *Roslynator*. https://josefpihrt.github.io/docs/roslynator. Accessed: 2024.05.02.

[21] Zuzana Dankovčíková. *Custom Roslyn Tool for Static Code Analysis*. https://is.muni.cz/th/f6oc3/masterThesis.pdf. Accessed: 2024.05.02.

[22] Giuseppe Scanniello et al. "An approach for architectural layer recovery". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 2198–2202. ISBN: 9781605586397. DOI: 10.1145/1774088.1774551. URL: https://doi.org/10.1145/1774088.1774551.

[23] Wat Wongtanuwat and Twittie Senivongse. "Detection of Violation of MVVM Design Pattern in Objective-C Programs". In: *Proceedings of the 8th International Conference on Computer and Communications Management*. ICCCM '20. Singapore, Singapore: Association for Computing Machinery, 2020, pp. 54–58. ISBN: 9781450387668. DOI: 10.1145/3411174.3411193. URL: https://doi.org/10.1145/3411174.3411193.

[24] M. Aljamea and Mohammad Alkandari. "MMVMi: A validation model for MVC and MVVM design patterns in iOS applications". In: *IAENG International Journal of Computer Science* 45 (Aug. 2018), pp. 377–389.

[25] Mohammad Hasan. "Finding the design pattern from the source code for developing reusable object oriented software". In: *2009 Second International Conference on the Applications of Digital Information and Web Technologies*. Aug. 2009, pp. 157–162. DOI: 10.1109/ICADIWT.2009.5273947.

[26] Santonu Sarkar, Girish Maskeri Rama, and Shubha R. "A Method for Detecting and Measuring Architectural Layering Violations in Source Code". In: *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. Dec. 2006, pp. 165–172. DOI: 10.1109/APSEC.2006.7.

[27] Yuanfang Cai, Daniel Iannuzzi, and Sunny Wong. "Leveraging design structure matrices in software design education". In: *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. May 2011, pp. 179–188. DOI: 10.1109/CSEET.2011.5876085.

[28] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*. The MIT Press, Mar. 2000. ISBN: 9780262267649. DOI: 10.7551/mitpress/2366.001.0001. URL: https://doi.org/10.7551/mitpress/2366.001.0001.

[29] Sunny Wong et al. “Design Rule Hierarchies and Parallelism in Software Development Tasks”. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. Nov. 2009, pp. 197–208. DOI: 10.1109/ASE.2009.53.

[30] Jon M. Kleinberg. “Authoritative sources in a hyperlinked environment”. In: *J. ACM* 46.5 (Sept. 1999), pp. 604–632. ISSN: 0004-5411. DOI: 10.1145/324133.324140. URL: https://doi.org/10.1145/324133.324140.

[31] Eleni Constantinou, George Kakarontzas, and Ioannis Stamelos. “Towards Open Source Software System Architecture Recovery Using Design Metrics”. In: *2011 15th Panhellenic Conference on Informatics*. Sept. 2011, pp. 166–170. DOI: 10.1109/PCI.2011.36.

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

[33] Dragoş Dobrean and Laura Dioşan. “Pathways for statically mining the Model-View-Controller software architecture on mobile applications”. In: *Soft Computing* 26.19 (Oct. 2022), pp. 10493–10511. ISSN: 1433-7479. DOI: 10.1007/s00500-022-06908-0. URL: https://doi.org/10.1007/s00500-022-06908-0.

[34] Dragoş Dobrean and Laura Dioşan. “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, 2021, pp. 36–46. ISBN: 978-989-758-508-1. DOI: 10.5220/0010326700360046.

[35] Dragoş Dobrean and Laura Dioşan. “Validating HyDe: Intelligent Method for Inferring Software Architectures from Mobile Codebase”. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek. Cham: Springer International Publishing, 2022, pp. 3–28. ISBN: 978-3-030-96648-5.

[36] Dragoş Dobrean and Laura Dioşan. *Intelligent Methods for Inferring Software Architectures from Mobile Applications Codebases*. https://teze.doctorat.ubbcluj.ro/doctorat/teza/fisier/6683. Accessed: 2024.05.02.

[37] Microsoft. *SourceProductionContext.ReportDiagnostic(Diagnostic) Method*. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.sourceproductioncontext.reportdiagnostic?view=roslyn-dotnet-4.7.0. Accessed: 2024.05.02.

[38] Microsoft. *IEqualityComparer<T> Interface*. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.iequalitycomparer-1?view=net-8.0. Accessed: 2024.05.02.

[39] Microsoft. *CompilationStartAnalysisContext.RegisterCompilationEndAction Method*. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.diagnostics.compilationstartanalysiscontext.registercompilationendaction?view=roslyn-dotnet-4.7.0. Accessed: 2024.05.02.

[40] Microsoft. *CSharpSyntaxWalker Class*. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.csharpsyntaxwalker?view=roslyn-dotnet-4.7.0. Accessed: 2024.05.02.

[41] Tibor Ásványi. *Algorithms and Data Structures II. Lecture Notes: Elementary graph algorithms*. http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/AlgDs2graphs1.pdf. Accessed: 2024.05.02.

[42] Gonzalo Navarro. "A guided tour to approximate string matching". In: *ACM Comput. Surv.* 33.1 (Mar. 2001), pp. 31–88. ISSN: 0360-0300. DOI: 10.1145/375360.375365. URL: https://doi.org/10.1145/375360.375365.

[43] Microsoft. *Choose diagnostic IDs*. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/choosing-diagnostic-ids. Accessed: 2024.05.02.

[44] Sonar. *S1104 - Fields should not have public accessibility*. https://sonarsource.github.io/rspec/#/rspec/S1104/csharp. Accessed: 2024.05.02.

[45] Microsoft. *Strings and string literals*. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings. Accessed: 2024.05.02.

[46] Microsoft. *Records (C# reference)*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record. Accessed: 2024.05.02.

[47] Microsoft. *Value types (C# reference)*. `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types`. Accessed: 2024.05.02.

# List of Figures