IPS Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

Fábio Miguel Rodrigues de Jesus

CodeGen

Improving software development through code generation

A project submitted in fulfillment of the requirements for the degree of **master's in software engineering**

Jury

Presidente (Prof. Dr, Cláudio Miguel Garcia Loureiro dos Santos Sapateiro, Instituto Politécnico de Setúbal – Escola Superior de Tecnologia)

Orientador (Professor, José António Sena Pereira, Instituto Politécnico de Setúbal – Escola Superior de Tecnologia)

Vogal (Prof. Dr, José Faustino Fragoso Femenin dos Santos, Instituto Superior Técnico de Lisboa)

November, 2019

To Inês Oliveira,

Whose struggle has motivated me overcome limits I hadn't acknowledged so that I could be the person I am today.

In memoriam.

Acknowledgments

Writing a master's thesis engages the author in a self-reflection process where each action taken, from learning to read, and doing basic math, to the ever so needed critical though that we use on daily basis, result in something that we would never thought it could be made by them. While passion and hard work are the main ingredients for a successful result, it's the people that surrounds us, be they family, friends, or even the ones that antagonize us, that often cause the greatest change in this result.

I would like to express my deep gratitude to both my parents, Maria and Miguel Jesus, for all the hard work and sacrifice made so that I could achieve my goals, and my brother, Diogo Jesus who had to endure my programming antics.

To my thesis supervisor, Professor José Pereira, who accompanied my education from the first day of college to the last day of this thesis, my thanks for his support and constant guidance throughout this research.

I would like to acknowledge Ricardo Pinto, the director of agap2IT's Research & Development department who provided me this challenge, and as also been accompanying this process from the start, providing any assistance required. This acknowledgement extends to the members of the department that acted as "rubber duckies".

Lastly to Agap2IT, for allowing me to experience several developing environments and technologies, some of them being pivotal for the decisions taken during the research, my thanks.

Abstract

Developing software applications requires time and experience that developers often lack. Additionally, development is more about the problem's domain and not about the coding process itself, making the automatization of the process quite challenging and engaging, unlike other successfully automated processes. To further reduce the developer's engagement in corporal developing standards such as following specific patterns or rules, CodeGen presents itself as a code generating tool that, while limited as a prototype, is intended to build and test applications in a set of languages and patterns. In order to do so, an exploratory research on the topics of code generation, architectural and design patterns, and programming languages is required, in order to evaluate what can be done with the current technology and knowledge available. Supported by this research, a prototype is developed as a proof of concept for a Visual Studio Extension that generates web applications in .NET MVC (Model-View-Controller). Since Visual Studio can't compile Java and the user is not restricted to the choice of development environment, the current research also analyses the possibility of having more than one user interface.

Keywords: Application-Model, Code-generation, MDSD, Roslyn, VSIX,

Table of Contents

List of Figures

List of Tables

List of references

Chapter 1.
Introduction

The IT (Information Technologies) field is one of the most influential markets of the modern era, introducing new ways to communicate and improve the efficiency on other fields such as education, healthcare and industry. As the backbone of many services in these activities, most of its investment is made to improve and maintain already existing solutions, unless a complete overhaul is justified.

As such, it has an ever-expanding workforce demand, and offer, that, according to the Market Research Institute, exhibits an estimated Compound Annual Growth Rate of 11.73% between 2019 and 2022 (Market Research Future, 2019).

Based on these values and considering how difficult it is to find ideal software engineers due to the time and experience required to master this set of skills, it is one of the hardest to fill jobs (Experis Engineering, 2017). It gets even worse when the amount of different technology available is taken into consideration.

With that in mind, the current research analyses the most commonly used technological stacks at agap2IT, the company where this project was planned, so that a prototype code generator/boilerplate template can be developed, with the goal of producing standard web applications with a reduced effort. In the current chapter, the reader is presented an overview of the motivation behind the project, along with an outline of the research's structure and methodology.

1.1. Context

Developing software is a complex task. It requires experience from each of its participants, and time to undergo a series of phases, depending on the methodology used. Being efficient at it, while assuring a certain amount of quality requirements is accomplished is even harder.

Fitzgerald (Fitzgerald, 2019) states that, according to the Association of Information Technology Professionals, the demand for developers increased by 32%, just in the United States. Konicek (Konicek, 2018) justifies such an excessive demand based on the role that IT plays for many of the services used as an alternative for physical systems like calendars, phone and fax calls, and written documents. This is evidence of a dependency on software solutions to simplify routine activities, which reinforces the need to innovate and develop better applications.

Consequently, the use of software solutions (e-commerce, productivity, logistics, etc…) available to the end-user has radically increased productivity and attends to diverse niches that couldn't find enough workforce for tasks such as improving customer experience with out-of-office hours, customized assistance and executing menial and repetitive tasks. While these process optimizations are not visible at the software technology field, they occur as frameworks and pattern standardization. Any further improvement would rely on complex automation that could lead to a loss of quality or become too specific, which is out of question.

However, due a ferociously competitive market and high customer expectations to deliver a quality project on a short period, developers lack the time to acquire new skills required for other assignments. Therefore, considering requirements can easily change, a tool that can generate an application based on a specification should not only ease the process, but reduce the number of errors produced and time expended on workarounds.

1.2. Research Setting

Both research and development began on a code academy at agap2IT's headquarters, from June to August of 2018. Subsequently, the process continues to be accompanied as the researcher interacts with several assignments, each with its own team, methodology and technological stack, allowing an exposure to various environments, providing experience in fields that could take longer on a single project, be it by acquiring know-how or by integrating concepts that would be otherwise difficult to understand by textual-source materials.

1.3. Research Methodology

This research is divided into research and development activities, and as such it is segmented into four phases, as seen below in figure 1.

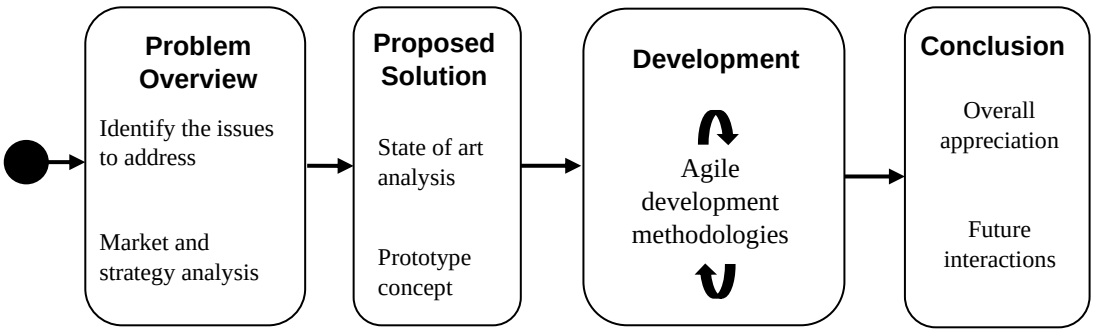


Figure 1. Research methodology workflow

During the first phase, several issues that will compose the motivation for the project are identified, allowing the reader to have an overview of the problem to solve. In addition, by observing the main lead in the code generation field a market and strategy analysis is performed to identify how the proposed solution could "fill in the gaps".

After delineating the strengths and weaknesses of the prototype, during the second phase, a set of functionalities is defined to represent a base for the prototype's concept. These work according to a study on the state of art about code generation, mainly through concepts of MDD (Model-driven Development).

The third phase focuses on software analysis and development, organized through one of the several methodologies available. This methodology provides a set of tools, metrics and benchmarks that vary according to the needs of the project or by sociological reasons. They have, however, several concepts in common such has specifications, reusability factors, Software Quality Assurance and progress analysis. (Jones, 2017)

Finally, an analysis on the progress is performed, reflecting on what could be made to improve what was developed, and what could be implemented in future instances.

The development methodology should be adequate for one analyst/developer, whose disposition allows short spans of irregular development phases. Therefore, practices like the Waterfall should be avoided. Pattern-based methodologies consider legacy software as the main development artifact, which, due to the project's dimension, would hardly be the case. XP (Extreme Programming) is best used on small teams that work on a single project, and Scrum and Kanban are organized, but since the project will suffer from time gaps between features, a daily board and the change cost might not represent the reality (Lamelas, 2018). With that in mind, it is assumable that an agile process is the best option.

RAD (Rapid Application Development) is a process designed by James Martin that, according to Rod Stephens and represented in figure 2, satisfies to the a few of the twelve agile principles, being the most important (Stephens, 2015):

- Continuous delivery and integration
- Possible changes of requirements
- Working software is the primary measure of progress
- Reflection post-execution

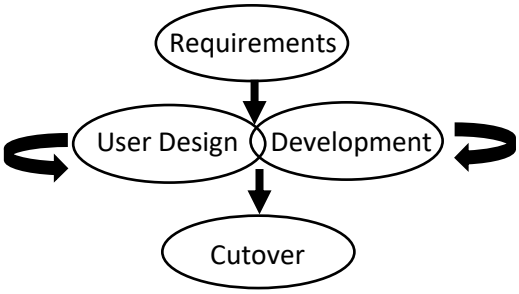


Figure 2. James Martin's RAD perspective adapted (Stephens, 2015)

1.4. Goals

Due to its nature, this research establishes two primary goals. The first one is result-oriented and is based on the development of a prototype tool that can, partially or fully, create a .NET MVC (Model-View-Controller) web application from a specification. The second main goal is to achieve the highest maintainability, readability and reusability possible. This implies that any feature implemented may be as modular as needed, so that other projects can make use of it as well, and the code must be documented and tested to support future implementations.

As an example, observed in the following figure, a developer would need to create a solution, four projects (one for each layer on a Layered Architecture), and the code required for a simple application, the prototype would be able to generate the same content based on a set of data provided by the user.

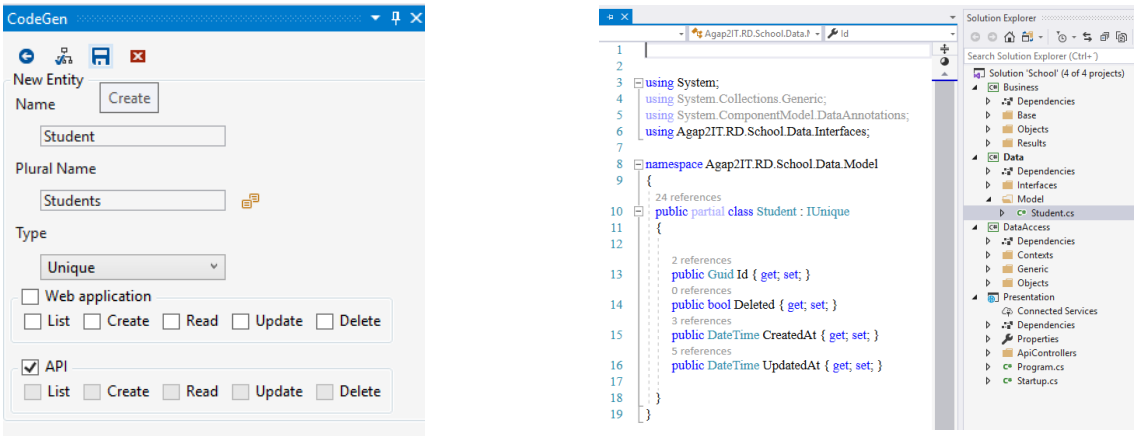


Figure 3. The data inserted in the left window can create the solution and code in the right window

Secondary goals for this research include an investment in the possibility of generating solutions in other programming languages and architectural patterns, being Java the principal interest since together with C# it's one of the most used in the industry. (TIOBE, 2019) Besides the typical monolithic application, there are many architectures available, and one like it may represent the ideal candidate for investigation.

Additionally, since IDE (Integrated Development Environment) plugins have become fundamental to the a customized experience, it could be possible to develop a VSIX (Visual Studio Integration Extension) as an alternative to CLI (Command Line Interface) and GUI (Graphical User Interface), the two common user interfaces. Due to the release of Microsoft's Visual Studio 2019, further evaluation into it may be required.

1.5. Practical and scientific relevance

As with any other field, a tool that decreases the number of mundane tasks the user needs to execute will, most likely, decrease on the predicted time the project would take to finish. The gained extra time can be employed on tasks that have higher specificity and, as such, may increase its value.

Since this research involves theoretical concepts and generation-assisting tools, it may prove useful to encourage researchers and developers alike to develop a more complete tool than the one projected.

1.6. Research Questions

This research attends to several topics on code generation, that must be addressed before beginning development. Thus, the following questions must be resolved through an exploratory research based on the available literature.

1. What can be generated by the application?
2. Can MDD (Model-Driven Development) be applied to a generator?
3. How should the generator be presented to the end user?

1.7. Document structure and reading guide

The current document is structured with the same order as the research's phases. All the questions posed are answered during one or more chapters. The following table relates each chapter with a research phase and the answered questions.

As observable in the following figure (figure 3) and tables (table 1 and table 2) presented in this section, that the first chapter introduces the research, motivating it. In the second chapter, the reader is introduced to each project's background, issues, and state of art. The theoretical analysis presents each of the main research topics for the projects.

Through the information acquired, the projects are planned and developed during the fourth chapter, with the fifth one serving as the research's conclusion by presenting a reflection about the work done and the questions posed during the first chapter and prompting the possibility of subsequent research.

Table 1. Research traceability matrix

Chapter	Phase	Questions
1. Introduction	-	-
2. Project Overview	Overview	
2.1.Code Generation	Overview	1
2.2.Market Research	Overview	1
2.3.CodeGen	Overview	3
3. State of Art	Proposed Solution	
3.1.Code Generation	Proposed Solution	1
3.2.Model-Driven Development	Proposed Solution	2
3.3.Applying MDD to code generation	Proposed Solution	2
3.4.Programming Languages	Proposed Solution	1, 2, 3
3.5.Architectural and Design Patterns	Proposed Solution	1, 2, 3
3.6.User Interfaces	Proposed Solution	3
3.7..NET Compiler Platform	Proposed Solution	1, 2
4. Prototype	Development	
4.1.Objectives and Obstacles	Development	2
4.2.Common Libraries	Development	3
4.3.Application Model	Development	1, 2, 3
4.4.Templates	Development	1, 2, 3
4.5.Engine	Development	3
4.6.User Interfaces	Development	3
5. Conclusion	Conclusion	All

Table 2. Chapter / Question distribution matrix

		Chapters																				
		1	2	2.1	2.2	2.3	3	3.1	3.2	3.3	3.4	3.5	3.6	3.7	4	4.1	4.2	4.3	4.4	4.5	4.6	5
Questions	1			x	x			x			x	x		x			x	x				x
	2								x	x	x	x		x			x	x				x
	3					x					x	x	x			x	x	x	x	x	x	x

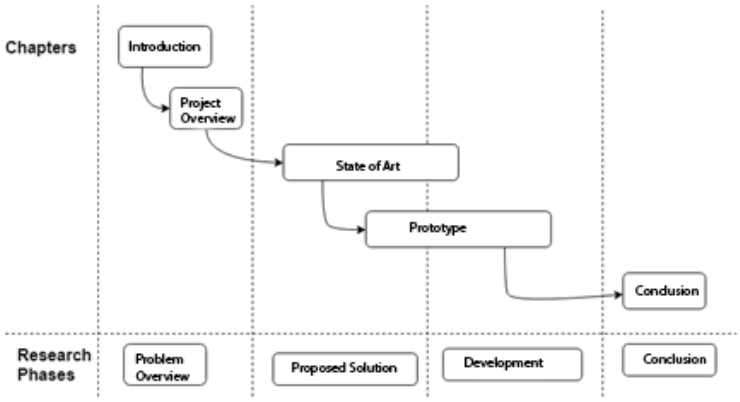


Figure 4. Chapter / Research Phase distribution matrix

1.8. Languages and Notation

Software development tends to be a highly complex process that requires a detailed documentation to ease later involvements in the project. Written documentation takes longer to read and write, so a notation that allows most processes to be described as a diagram reduce the level of formality and the time spent on it. To this end, two modeling notations are used, UML (Unified Modeling Language) and BPMN (Business process Modeling and Notation).

1.8.1. UML (Unified Modeling Language)

UML is the *". . . standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system."* Currently, it provides the user with thirteen templates like use cases, class, package and sequence diagrams. (Booch, Rumbaugh, & Jacobson, 2005) A description of the UML elements used can be observed in appendix A1.

1.8.2. BPMN (Business Process Modeling and Notation)

BPMN is a standard developed by the OMG (Object Management Group) with the purpose of providing a "*... notation that is readily understandable by all business users, from the analysts that create the initial drafts of the procedures, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will perform those processes, and finally, to the business people who will manage and monitor those processes."* (OMG, 2011) Its components description can be observed in appendix A2.

Chapter 2.
Project Overview

As stated in the previous chapter, the main goal of this research is the development of a prototype tool that can generate an application from a specification. There are several solutions available in the market, whose lead may be pivotal to acknowledge what is being done, what could be done and how it could be done.

Therefore, before engaging with any sort of theoretical analysis or product development, it is essential to understand the concept of code generation, and its place in software development, presented in the first section.

In the second section, taking into consideration how the market perceives code generation, a market research is performed to address OutSystems, the leading tool, followed by a strategy analysis in order to determine if it is a potential competitor or belongs on another niche.

The third and last section presents CodeGen as agap2IT's most challenging project describing how it could be a tool that may assist its developers into increasing their productivity.

2.1. Code Generation

Although code generation wasn't an explicitly visible process up until a few years ago, nowadays it is considered a staple in many software houses and consulting companies.

Application generators, or Low-Code platforms, reduce the time it takes to produce a semantically precise code, allowing young developers with a lack of solid coding knowledge to build applications by drawing flowcharts with UML elements or even by connecting "puzzle pieces". (MIT, 2019)

These platforms commonly attend to the three following development sectors, observed in figure 5.

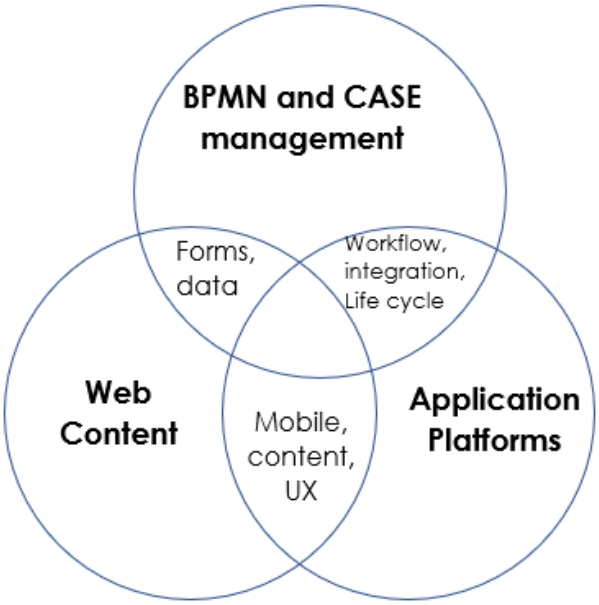


Figure 5. Software development sectors (Richardson & Rymer, 2014)

Development platforms adversely affect experienced programmers due to several factors such as implementation limitations, the time it takes to build a simple function in a platform and the lack of post-build maintenance. The later refers to a trend among most generation applications that create and deploy an application without allowing the developer any interaction whet the pre-compiled result, thus leading to a constant dependency of the tool. As such, developers tend to mistrust the algorithms' efficiency since it's illegible, even if the user decompiles the result.

2.2. Market Research

As a high-ranking member of Gartner's 2018 "Enterprise High Productivity Application Platform as a System" and Forrester's Q1 2019 "Low-Code Development Platforms for AD&D Professionals, as seen in figure 6, OutSystems is one of the most distinguished application platforms available. (Richardson & Rymer, 2014)

Figure 6. Forrester's Q1 2019 LC Development Platforms for AD&D Professionals (Outsystems, 2019)

It provides a simple interface that allows the user to model the application from a database perspective and express business rules through flowcharts. Although the interface is akin to a professional developer, dealing with flowcharts instead of pure code and patterns can take longer to develop and reduce the readability of the business flow due to the dimension of the flowchart. The result of an application can be either published or just released to a folder. However, since it is only available after compilation, changes must be made through the OutSystems application, unless the user decompiles the DLL (Dynamic-link Library) produced. (OutSystems, 2018)

2.3. CodeGen

In 2016, agap2IT's research and development department took on a new challenge, developing a tool that could reduce the developer's workload without reducing the maintainability of the product. At the time, Microsoft technology was the mainstream and, as such, a code boilerplate would substantially reduce the amount of errors and time taken to develop a project. This would also normalize any solution to a set of standards, such as increasing readability and standardizing the architecture.

Nowadays, since there's a bit of variety in the client's needs, there's a need to expand this concept to other architectures, programming languages and any other relevant element. Since Visual Studio allows developers to create their own extensions that are integrated into the IDE's shell, a CodeGen extension would also be desirable.

Table 3. CodeGen's strategic analysis

Strengths	**Weaknesses**
Solid base knowledge about the field of study. **T**he developer is experienced in code generation and boilerplating.	**L**ack of workforce and available time. **C**ertain application nodes may not be generated.
Opportunities	**Threats**
Existence of libraries that assist code analysis. **N**o available applications in the market like the proposed solution.	**S**urging technologies that require additional study to be implemented. **C**hange on the competitor's approach to the problem domain.

Chapter 3.
State of Art

Code generators are commonly used over frameworks due to certain restrains that force the application to conform to a set of standards. While it is possible to create software without their assistance, frameworks provide means to reduce the required interaction with several technologies. Since these are used at most projects at agap2IT, CodeGen is an application that generates code based on a model, which is a structure for the expected project. This requires knowledge about concepts such as modeling, software engineering patterns, and user interfaces that can be acquired throughout the available literature, being pivotal for the success of the fourth chapter.

The first section presents code generation by establishing it in the timeline, which allows a better understanding of its motivation and how it should be engaged. Next, in the second section, the reader is introduced to MDSE (Model-Driven Software Engineering), the paradigm used to create the model used by the generator. These concepts are then bridged with the project through a reflection on the design of the components of the generator, exposed during the third section, through a brief presentation on each of architectural or design patterns.

With a clear understanding on what must compose the generator, the reader can delve into their dynamic aspects. These concepts attend to syntax and semantic elements of programming languages, and the structures of architectural, or software design patterns. Both programming languages and architectures are addressed in the fourth and fifth sections. The generator can then be interacted with through more than one user interfaces. These are described in the sixth section.

In the seventh, and last chapter, the reader is presented to the .NET Compiler Platform, the cornerstone of the code generator due to its capacity to build and execute code at runtime, exposing the compiling process.

3.1. Code Generation

As a process, code generation has existed since the beginnings of programming. It is constantly evolving, through stimulations caused by paradigm and technology shifts. Still, up to today, its objective remains the same.

3.1.1. Historical Rundown

The concept of code generation, before it was known as such, was the act of automatizing a programming process that was done manually. One of the earliest references, associated to automatic programming, dates to the early 1940s as the optimization of the tape punching process. (Gorn, 1940)

A decade later, punching tapes were replaced by mathematical symbols that, when joined as a series of expressions, formed autocodes. Scientists later recognized certain natural occurrences that could be standardized and repurposed. These patterns would be called universal autocodes, or as they are currently recognized, programming languages. (Redish, 1959) During the mid-eighties, Parnas describes code automation as a euphemism for *"... programming with a higher-level language that the ones currently available*". (Parnas, 1985) Solomon reinforces this statement by formalizing the definition of the compiler, a tool that converts high-level languages, such as FORTRAN and ALGOL, to lower-level ones. (Solomon, 1993)

Currently, the concept remains mostly the same, with Novak describing its Artificial Intelligence aspect as *"... the generation of programs by computer, usually based on specifications that are higher-level and easier for humans to specify than ordinary programming languages*." (Novak Jr., 2006)

3.1.2. Motivation

Considering the presented chronology, it is observable that the main intent behind this process is the reduction of complex procedures by achieving a normalized language that resembles written human expressions. This process of automatization, or encapsulation, allows developers to invest on other tasks such as research and development, which lead to concepts in the fields virtual and augmented reality, sensor tracking applications, amongst others.

From a developer's perspective, one of code generation's most attractive features is reverse engineering, allowing the creation of a database or file based on code, and vice-versa. (Tomassetti, 2018)

Nowadays, with the technological boom, several other components and tasks are connected to development, such as unit and integration testing, project management and documentation. Most of these need to be informatized but, unlike user experience tests and high-level analysis, can somewhat be automatized to a certain point where the developer only needs to fill some gaps.

3.1.3. Forethought

There are several advantages to the use of code generation such as error reduction, artifact reusability, development time reduction, and since the generated code is already optimized, the application tends to perform better. Depending on the approach it can also protect the programmer's intellectual property, increase flexibility during development and deployment, and attend to specific characteristics. (Cabot, Brambilla, & Manuel, 2017)

There are, however, some considerations to take, mainly when dealing with maintenance due to the dependency on the generator, which must evolve along with the developers' needs. If the problem domain becomes too complex, it is likely that the code becomes so complex that the generated one needs to be changed to the point of taking longer than it would take to write it from an empty solution. This may be a deciding factor on whether to avoid a code generation tool. (Tomassetti, 2018)

3.2. Model-driven Development

MDD (Model-Driven Development) is an abstraction-level improvement approach that uses a model as the key concept for an application, instead of code. A higher level of abstraction is justified by the increased expressiveness through natural speech instead of machine language. (Milicev, 2009)

3.2.1. The many layers of Model-based Engineering

MDD is but one of the many layers that compose the MBE (Model-based Engineering), as displayed in figure 7.

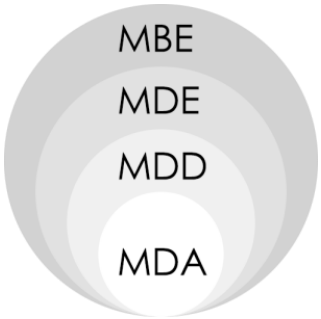


Figure 7. MBE's Layers (Cabot, Brambilla, & Manuel, 2017)

Though the model may not represent a key element to development, MBE (Model-Based Engineering), as a process, defines it as an important element that drives most, if not all, engineering activities. Bergenthal defines MBE as "*an approach to engineering that employs models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle*". (Bergenthal, 2011)

MDE (Model-driven Engineering) is a subset of MBE that makes significant structures explicit, while maintaining a high level of abstraction. Being the production of software through the junction of models and transformations its key concept, architects and engineers can have a thorough overview of the system, allowing them to develop earlier. (Cabot, Brambilla, & Manuel, 2017) Thus, it can be defined as a "systematic use of models as a primary engineering artifact through its lifecycle". (Punter, Voeten, & Huang, 2009)

MDD (Model-driven Development) acts as the focal point for this research since it's the paradigm that "uses models as the primary artifact of the development process", typically by being subjected to a semiautomatic implementation process. MDD is intended to increase the level of abstraction and automation of the application development process. By doing so, developers can focus on solving domain problems instead of implementation ones. (Cabot, Brambilla, & Manuel, 2017)

MDA (Model-driven Architecture) is a specific vision of MDD proposed by the OMG (Object Management Group) which, according to their standards, intended to encapsulate business and application logic from technological evolution, defining the model as the main artifact during development. It's composed by several elements, such as a system, a model, viewpoints, view, and transformations, following four principles based on MDE. (Cabot, Brambilla, & Manuel, 2017)

- Models must be strictly expressed by a concise and explicit notation.
- Models must comply with metamodels.
- System specifications must be organized around a set of models and associated transformations with relationships and mappings.
- It should also increase acceptance and adoption to the MDE approach.

3.2.2. Model and Metamodel

Since analysts describe a system through diagrams, it is deductible that a formalized abstract language could be used to describe the system's model. This model can be described as "… an abstract representation of a system's structure, function or behavior." Therefore, if the model is an abstraction of something in a reality, another level of abstraction could be applied, producing a metamodel.

Metamodeling has several practices such as defining new modeling programming languages or metadata that describes existing data. Since compilers don't recognize the model as a source of data to produce the application, metamodels are used to parse and validate the model. As seen below, figure 8 represents the relationship between a model and a metamodel, based on Jordi's definition. (Cabot, Brambilla, & Manuel, 2017)

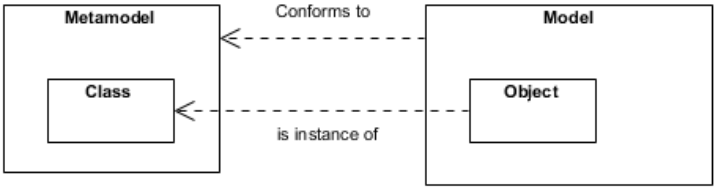


Figure 8. Relationship between a model and a metamodel (Cabot, Brambilla, & Manuel, 2017)

There are three levels of abstraction connected by model transformations, observable in figure 9, based on Truyen's description (Truyen, 2006). These range from business (CIM), to domain viewpoints (PIM), to implementation (PSM) viewpoints.

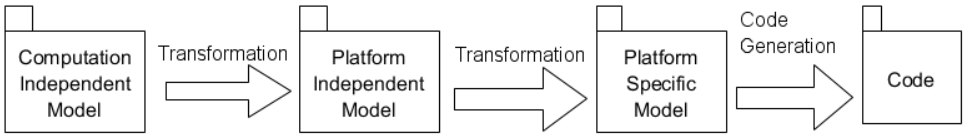


Figure 9. Transformations applied to MDA viewpoints (Cabot, Brambilla, & Manuel, 2017)

The CIM (Computation-independent Model) uses the appropriate vocabulary for domain experts, presenting what is expected from the system without any technological specification.

The PIM (Platform-independent Model) can be mapped to one or more platforms and describes the application's behavior and structure without any technical details, and is the only part of the CIM that can be solved by a software-based solution.

The PSM (Platform-specific Model) can be created by joining the PIM's specification with the details from a specific platform.

3.2.3. Transformations

As previously mentioned, a model alone is not enough to build an application. There must be a way to validate and parse it. This process, named transformation, is a set of actions taken on a model to map its content to another kind of artifact.

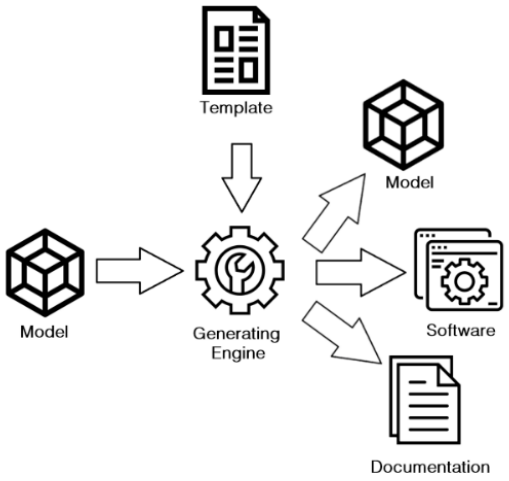


Figure 10. Possible results of a generating engine adapted from the description provided by Jordi et al. at page 39 (Cabot, Brambilla, & Manuel, 2017)

This is commonly achieved by the use of templates, that are, as Brambilla et al. describe them, a *"... kind of blueprint which defines static text elements shared by all artifacts, as well as dynamic parts which have to be filled with information specific to each particular case*". (Cabot, Brambilla, & Manuel, 2017) Transformations can be classified as M2M (Model-to-Model), M2T (Model-to-Text), and T2M (Text-to-Model).

M2M transformations convert one or more models into other ones, being exogenous if the language is different, or endogenous if it's the same language, as observable in figure 11.

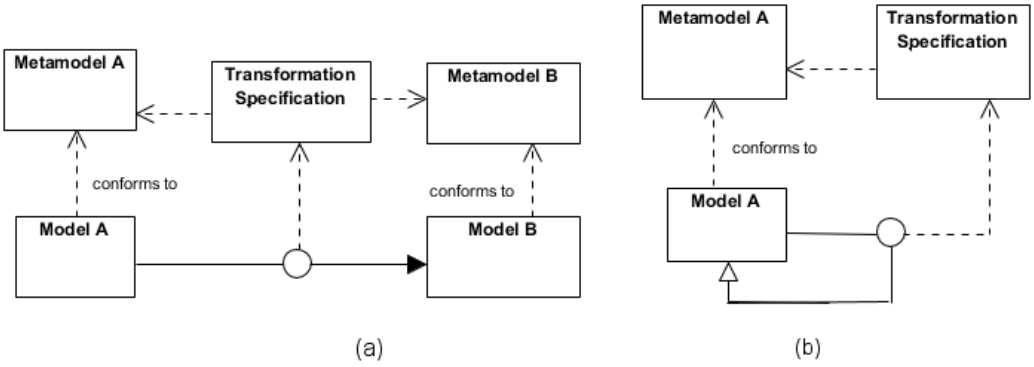


Figure 11. Exogenous (a) and endogenous (b) transformations (Cabot, Brambilla, & Manuel, 2017)

M2T transformations use one or more models as input, generating code that can be compiled. While separating the static code, at the template, from the dynamic code, generated using a declarative query language with meta-markers, this transformation allows the explicit representation of the expected output. (Cabot, Brambilla, & Manuel, 2017) The following figure represents an abstract view of how, through an M2T engine that contains a template can transform a model into executable code.

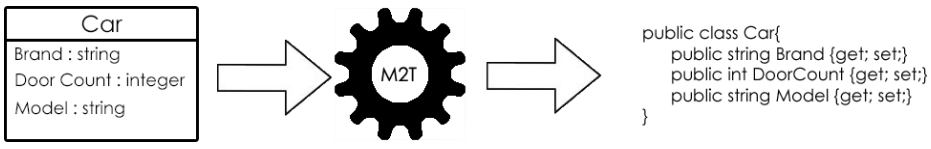


Figure 12. M2T transformation example

T2M transformations convert inputted text into a model. It is commonly used in reverse engineering. The text must be parsed, validated and then, through pattern recognition, filtered for static and dynamic elements, and building a model from that information.

3.3. Applying MDD to code generation

The process of MDSE produces applications or features from the highest-level specification possible, through continuous applications of M2M transformation to each of the main steps of development, and eventually a M2T transformation, to acquire code, files and any other artifact.

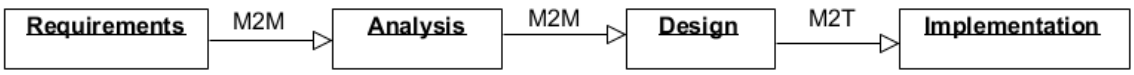


Figure 13. MDSE process through development

As previously stated, code generation has some areas of concern. These may be just simple decisions to take based on how the generator should act, or pitfalls that the developer should be aware before implementing this kind of solution.

3.3.1. Full generation vs. Partial generation

It is quite evident that when a class is created, if anything that invokes it does not input the necessary arguments, it won't work. The same happens when the generating process occurs over an incomplete, or corrupt, model. Without any knowledge base to fill the missing data, if there's no proper model validation, the code will be incomplete. Still, it is possible to overcome this obstacle by executing a partial code generation instead of the one presented, which could be called "full generation".

Partial code generation expects developers to fill in the missing code, be it by post-generation where the file is changed, or by code injection during the generation process. As expected, this process tends to be dangerous since it can cause compilation or execution errors. Therefore, it is highly advisable to use this approach as a last resource, if the user knows its risks and there's a way to either validate this code or create protected areas where the manually inputted code is added, so that it does not affect the generated one. (Cabot, Brambilla, & Manuel, 2017)

3.3.2. Designing the application model

Software applications contain both static and dynamic content. Static content is commonly provided as media and, if it is the case, styles that format said media or text according to a set of specifications.

The dynamic content must be filled with the application model, that provides the structure for the application. This model needs to identify the application and its components, while taking in consideration a certain level of abstraction in order to be used without any, or at least the minimal, amount of programming language constraints.

According to Cabot et al, the developing a metamodel is a three-step process that is both iterative and incremental. The first step, that ties in with the notions presented earlier in this section is the analysis of the modeling domain based on its purpose, realization and the programming language's content. This can be achieved by taking in several reference examples to define the model's requirements. (Cabot, Brambilla, & Manuel, 2017)

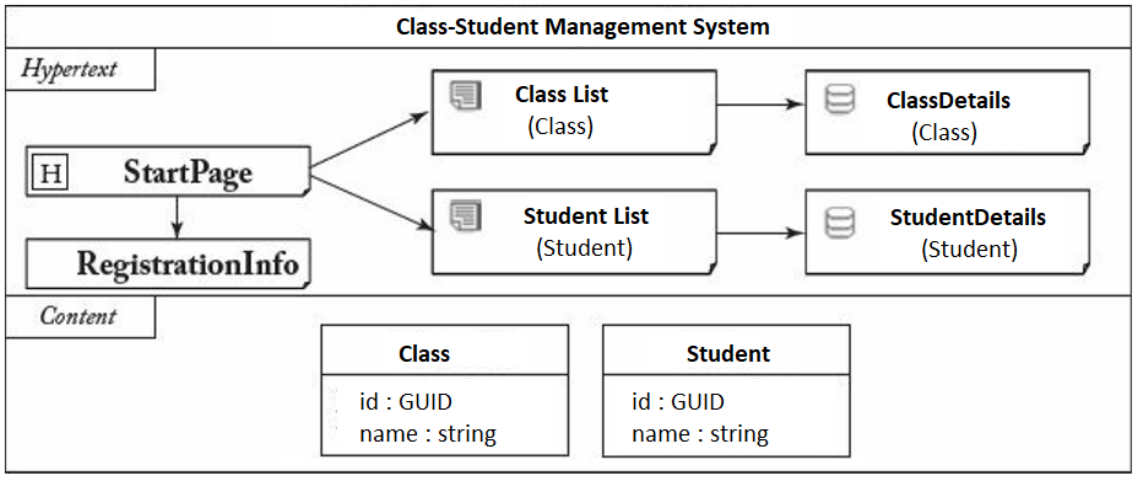


Figure 14. Example of an application that can be modeled (adapted) (Cabot, Brambilla, & Manuel, 2017)

The next step consists on designing the modeling language. While the modeling concepts are formalized by "… modeling the abstract syntax of the language…", and the constraints should be formalized by an OCL (Object Constraint Language). As it is visible, in the previous figure, there are several concepts that can be addressed as part of the model such as the layers, classes and other elements, exposed in table 4.

Table 4. Modeling concept table for the adapted example sketch, based on the concept table provided by Cabot et al. (Cabot, Brambilla, & Manuel, 2017).

Concept	Intrinsic Properties	Extrinsic Properties
Web Model	name : string	a Content Layer and a Hypertext Layer
Content Layer		Any number of classes
Class	name : string	Any number of attributes
Attribute	name : string type :Type	
Hypertext Layer		Any number of pages and one page as homepage
Static Page	Name : string	Any number of non-contextual links
Index Page	Name : string Size : integer	Any number of non-contextual links and contextual links, and a displayed class
Details Page	Name : string	Any number of non-contextual links and contextual links, and a displayed class
Non-Contextual Links		One target page
Contextual Link		One target page

With an established notion of the metamodel components, it is now possible to create the metamodel representation, displayed in figure 15.

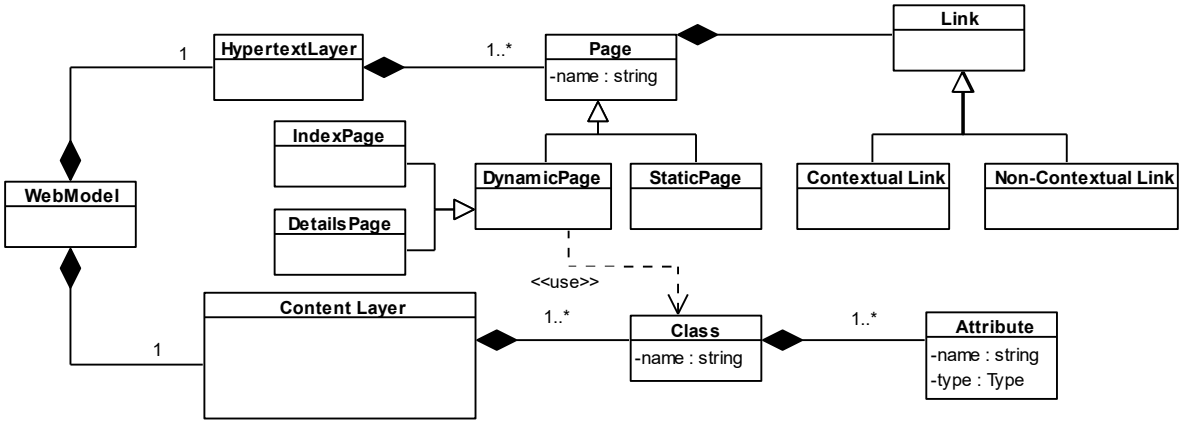


Figure 15. Metamodel based on the one provided (Cabot, Brambilla, & Manuel, 2017)

In this example, the authors also add a few constraints such as the uniqueness of a class' name, and the restriction of pages reference by links, amongst other guidelines specific for the metamodel at hand.

Finally, the model must by validated for its completeness (the ability to achieve its purpose) and correctness (the ability to avoid producing applications without errors). Object diagrams can be used for this purpose since objects describe classes, values are

for attributes and links for associations between classes.

> "… other general principles of language design such as simplicity, consistency, scalability, and readability also have to be considered." – Cabot et al (2017)

3.3.3. Designing templates

Templates are one of the components used in a M2T transformation that defines static content and dynamic spaces that can be filled with the model's data or with logic that builds static code. As such, analogous to HTML (Hyper Text Markup Language), they are composed by text artifacts that may have a syntactic meaning, and metamarkers for dynamic data that must be interpreted by a template engine. (Python Software Foundation, 2016)

The use of templates allows the explicit representation of the output's structure. This leads to a better readability and understandability of specifications and eases the development by providing code that is repeated and optimized. (Cabot, Brambilla, & Manuel, 2017) This, however, does not imply that a template will be a easily maintained or readable artifact since the mixture of static and dynamic data may cause unwanted characters such as extra spaces.

3.3.4. Designing the generator

Depending on the user's needs, the generator may range from a simple application through a GPL (General-purpose Language), that produces the code as a string of characters by joining the template with its respective model elements, to a fully DSML (Domain-specific Modeling Language) design tool that generates the code based on a series of syntax nodes that, when parsed, act as a model. This has several conditionings such as the benefit / cost ratio to implement the solution, how the user will interact with it, its maintainability, and so on.

While GPLs are easier to interact with, since they are commonly used languages such as C# or Java, according to Fowler, DSLs (Domain-specific Language) are only focused in solving a particular aspect of a system which, by consequence, has limited expressiveness. (Fowler, Domain Specific Languages, 2011) A DSL can be external, where it has a custom syntax and is separated from the application's main language to the point of needing parsing tools to be understood (e.g. SQL, XML), or internal, whose

syntax is valid in the GPL, and are seen, as Jordi describes them, as an extension of the actual host language. (Cabot, Brambilla, & Manuel, 2017)

3.4. Programming Languages

Mitchel describes programming languages as sets of expressions with a specific syntax that *"... provide abstractions, organizing principles and control structures that programmers use to write good programs*." Unlike the DSLs presented earlier in the previous section, the programming languages that developers use are GPLs.

According to a survey from Stack Overflow (Stack Overflow, 2019), amongst the top ten "most loved" languages, C# (67%) and Java (58.3%) stand out. Since frameworks are an important aspect of this research, the data on these is also relevant. On the same survey, the most loved frameworks include Spring (65.6%) for Java, and ASP.NET (64.9%) for C#. Adding to this data, on TIOBE's index for September 2019, Java ranks first, followed by C# at 5th place. (TIOBE, 2019)

These two languages, along with their most commonly used web frameworks, are the subject of investigation for this section, and will be addressed as the basis for the templates and any element of the model that may by unique to them.

3.4.1. C# and .Net Core

According to Microsoft, C# is a *"type-safe object-oriented language that enables developers to build a variety of secure and robust applications ..."*. (Microsoft, 2015) With a highly expressive syntax, it shares many similarities such as brackets, class and function naming, and keywords, with other programming languages.

According to the main source of documentation, its main advantages are (Microsoft, 2015):

- LINQ (Language-Integrated Query) that provides built-in query across various data sources, properties that serve as accessors to variables, and other sorts of syntactic sugar.
- Support for encapsulation (delegates), inheritance, polymorphism
- Inline XML documentation
- Statically typed
- Operator and conversion Overloading

- Concise syntax
- Highly efficient garbage collector and memory management
- Attributes that provide declarative metadata

```
namespace ProjectName
{
    0 references
    public class ClassName
    {
        0 references
        public int Property { get; set; }
    }
}
```

Figure 16. Example of a C# class with get/set accessors

Since it runs on .NET Framework, one of Windows main components, the code is compiled into an IL (Intermediate Language) that conforms to the CLI (common Language Infrastructure). When the program is ready to be executed, the instructions in IL are compiled into native machine language. This process can be observed in figure 17. (Microsoft, 2015)

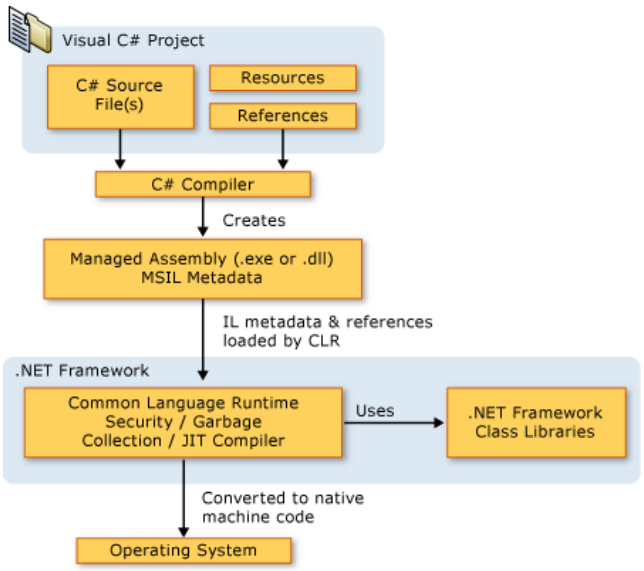


Figure 17. C# Compilation process (Microsoft, 2015)

.NET Core is the latest version of .NET Framework. It's cross-platform, can run code on several architectures, is retro compatible, and open source. Amongst these, the projects that are typically connected to Web development are Class Libraries that can contain a set of classes and features, and ASP.NET Web applications that can be used to create such as Web APIs (Application Programming Interface), WebServices and MVC applications (Microsoft, 2018).

3.4.2. Java and Spring

According to Oracle's documentation on the language, Java is, much like C#, a strongly typed, object-oriented GPL that is compiled to a format defined in the JVM (Java Virtual Machine) specification. For developers to produce source code, Oracle provides the JDK. This toolkit contains several tools used for development, documentation and compilation (Oracle, 2014). Its main advantages are:

- Supports strict static types
- Cost friendly
- Backwards and cross-platform compatibility

```
package ProjectName;
public class ClassName {
    private int _property;

    public int GetProperty() {
        return _property;
    }

    public void SetProperty(int value) {
        _property = value;
    }
}
```

Figure 18. Example of a Java class

As seen on figure 19, Java compilation is different from C# because it requires the developer to install both the JDK (Java Development Kit) and the JRE (Java Runtime Environment). The source code is compiled to bytecode and the JRE converts it to machine code.

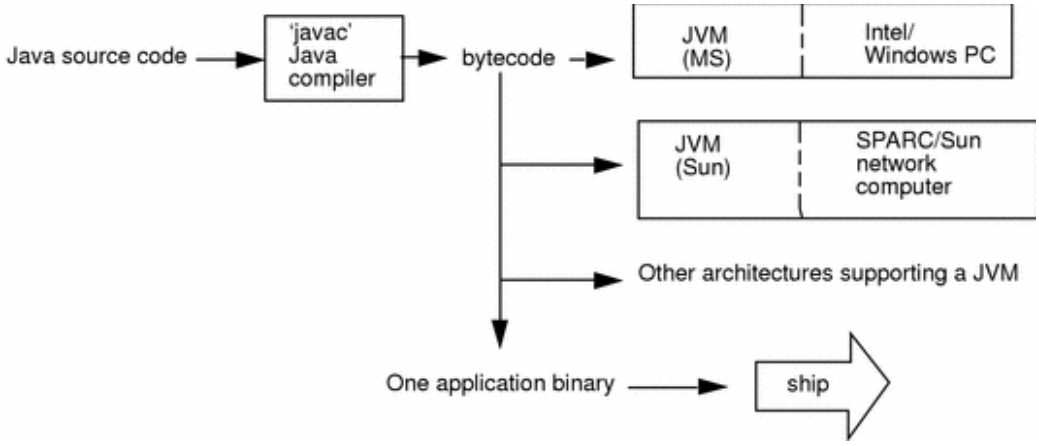


Figure 19. Java compilation process (Oracle, 2001)

In order to develop web applications with ease, developers can use the Spring framework, which, according to Pivotal, *"... provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform*." Spring features core technologies such as dependency injection, testing, Data access transactions, and an MVC(Model-View-Controller) web framework. (Pivotal, 2010)

3.4.3. Syntactic Deviation

The two presented languages are indeed similar, mainly because they are the standard languages used in the industry. There are however slight differences such as certain components that are present on only one of them, which would incur in a change to the application model and syntactic nodes that can cause changes to the templates. With the acquired knowledge from this section, it is now possible to focus on a set of languages, and frameworks, in order to build at least the model in the most reusable way possible.

One of the main components that Spring lacks, compared to .NET is the notion of attribute. These pieces of metadata allow, as previously mentioned, the addition of content that may be used for validation (web applications) (Microsoft, 2016)

```
[CreditCard]
0 references
public string CreditCard { get; set; }
```

Figure 20. Example of the credit card attribute

Spring does, however, provide validators through Hibernate which allow the use of similar components as the attributes. (Stalla, 2019) Since these may be different, or have different properties.

```
@CreditCardNumber(ignoreNonDigitCharacters = true)
private String lenientCreditCardNumber;
```

Figure 21. Example of the credit card annotation

On a language level, there are several differences on how a class is written. There are several data types that differ, but a middle term could be used to represent both the C# and the Java types (e.g. C# has GUID and Java has UUID). There are other nuances as well, such as namespaces for C#, that encase the classes into curly brackets, and packages in Java that are a one-line declaration.

3.5. Architectural and Design Patterns

While building an application doesn't require any sort of structure, if the files are correctly linked, software patterns can ease the process by providing best practices that target certain issues solvable using rules or algorithms. Among these patters, the architectural ones are responsible for defining the application's structure, by defining the basic characteristics and behaviors.

There are several architectural patterns, each with its specific purpose. In agreement with Richards, there are five essential patterns, layers (1), event-driven (2), microkernel (3), microservices (4) and space-based (5). (Richards, 2015)

The layered architecture is the most commonly used pattern for object-oriented applications, resembling the traditional IT (Information Technologies) communication structure, to the point where each layer is closed to outside requests, at the exception of the layer right below it, which implies that for data to reach the presentation layer, it needs to flow between each layer.

By building highly decoupled, single-purpose event processing components that communicate asynchronously, the event-driven architecture allows developers to create small to large complex applications. It consists of two topologies. The mediator, that is used to execute several steps in an event, and the broker, used when the developer wants to chain multiple events together without a mediator.

The microkernel architecture pattern is used to produce product-based applications that are packaged and made available as a third-party product. Some examples of its use are internal business applications that can be extended by integrating pluggable features, being useful in situations where there's a core application that must be separated, and isolated from these additional features.

The microservices architecture has become one of the most used architectural patterns as of late since it is a viable alternative to monolithic applications when justifiable. It's composed of separate deployable units that increase the degree of component decoupling. Since it's still a concept in development, it should be used with caution.

The space-based architecture uses the concept of tuple space to increase a higher scalability. Most web-based applications use this pattern because of the linear request

composed by a request from the browser to the web server, followed by the application server and finally to the database server. With this pattern it is easier to decrease bottleneck issues.

For the purpose of this research, both the layered and microservice architectures are the main study cases. Since the microservice architecture requires additional concepts and skills such as DevOps, although it's a study case, this research will focus on the concept of a single microservice.

3.5.1. Layered Architectural Pattern

As previously stated, the layered architecture, or multi-tier architecture, is the most commonly used architecture. Components are disposed as horizontal layers, each having a specific role. As a standard, applications have layers for presentation, business, persistence and database, as seen in figure 22. (Richards, 2015)

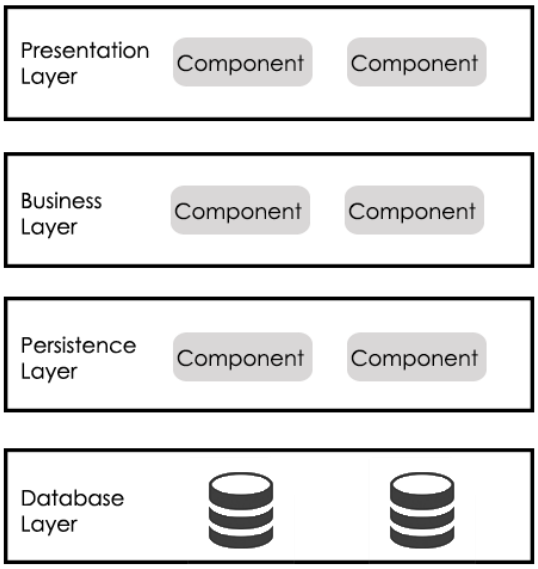


Figure 22. Layered architecture pattern (Richards, 2015)

Nonetheless, there's no set number of layers, so it is expected that some developers use only three by pairing the persistence and business layers. If the application is complex enough, it might be separated into five or more layers. agap2IT uses this pattern frequently, with the four layers presented, together with the MVC (Model-View-Controller) pattern to produce web applications. Additionally, as Vaugh describes it, this architecture supports n-tier systems and, as such, can be used for most kinds of applications, entwining with the notion of Domain-driven Development. (Vernon, 2013)

Database Layer

Depending on the available literature, this layer may be known as “Data Layer”, “Database Layer” or “Domain Layer”. From a domain-driven development perspective, this layer contains concepts based on entities (tables) that contain attributes (columns),and are associated with other tables (foreign keys). (Vernon, 2013)

When observed together with the MVC pattern, each of these entities are the same as the ones on the model, that is described by Fowler, et al, as an object, or set of objects that represent some information about the domain (Fowler, et al., 2002)

Choosing the type of key to use on an entity can be based on how the user treats the records. A numerical index is faster, easier to remember and keeps a chronological order whereas the data type range may become a limitation. A UUID). (Universally Unique Identifier) is generated so that it is impossible to know the previous record and it’s is rarely repeated because it can have 2128 combinations. It may, however, consume an exaggerated amount of resources since it takes 16 bytes to store instead of the 4 used by integers. For consistency, classes identified by a number are called enumerables and the ones identified by a UUID) are unique entities. (Loritsch, 2017)

```
public partial class User
{

    0 references
    public System.Guid Id { get; set; }
    0 references
    public string Username { get; set; }
    0 references
    public string Password { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Identification { get; set; }
    0 references
    public string Contact { get; set; }
    0 references
    public string Email { get; set; }
}
```

Figure 23. Example of a unique entity

Through use of these identifiers it is possible to form relationships that can be one-to-many, one-to one and many-to-many, being the first one the most common (see figure 24). Other properties that are commonly integrated in entities keep track of status changes such as the timestamp when it was created or updated, and status such as whether it is deleted or not.

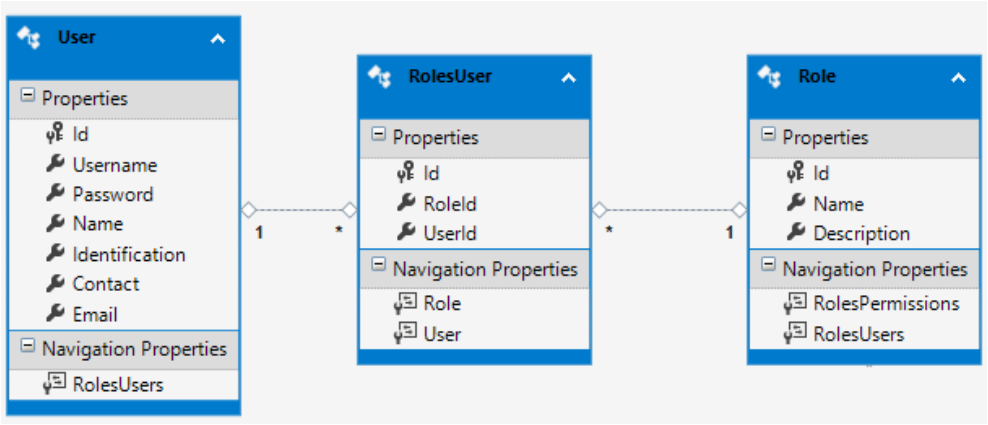


Figure 24. Example of relationships between three entities

It is noteworthy that, while all the layers are closed to communication, when there's a data abstraction layer, it still can be used by any of the high-tier layers, as seen in figures 26, 28 and 29.

Data Access Layer (Persistence)

The data access layer is created when, according to, the main intent is to "encapsulate data access and manipulation in a separate layer." This layer is composed by DAOs (Data Access Object), a component that implements data access mechanisms to access and manipulate data known as CRUD (Create, Read, Update, Delete) operations, as seen in figure 25. (Alur, Crupi, & Malks, 2003)

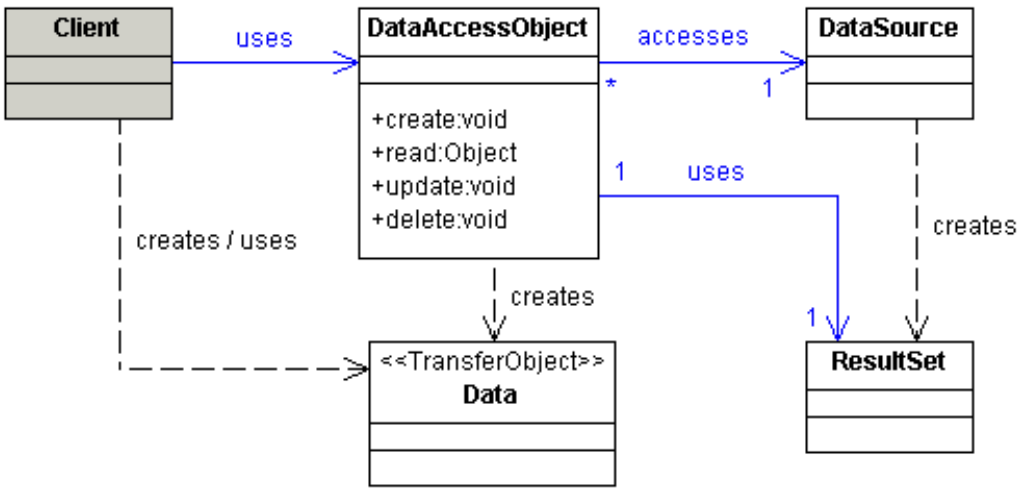


Figure 25. Class diagram for a data access object example (Alur, Crupi, & Malks, 2003)

The main advantages for this layer are a reduced migration difficulty and code complexity, and an organized and transparent structure. However, it adds an extra layer,

increases the complexity on object-oriented design and requires a class hierarchy design (Alur, Crupi, & Malks, 2003).

Based on the description provided by Richards, since all the layers are closed to communication, this tier interacts directly with the database layer and the business layer, as seen on figure 26.

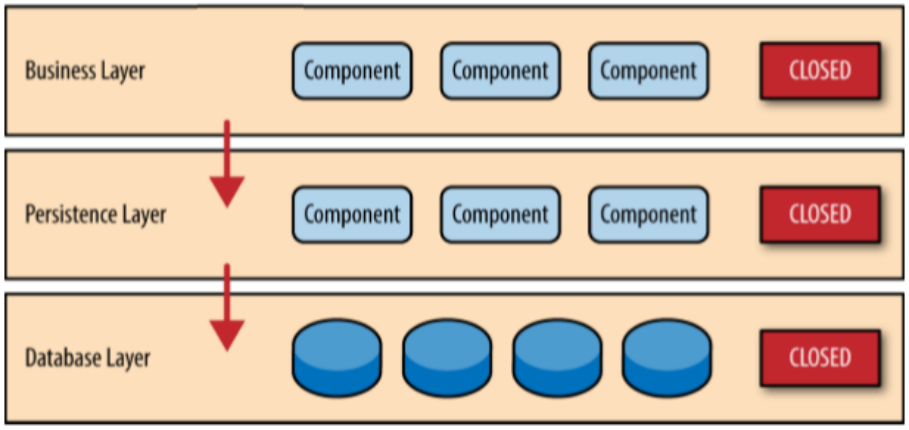


Figure 26. Excerpt of the layer architecture request flow on the persistence layer (Richards, 2015)

Depending on the programming language, or framework, used, it is also possible to turn these operations asynchronous.

Business Layer

According to Oracle, the business layer is composed by components (Business Objects) that provide functionalities related to the business domain to the application (Oracle, 2010). Alur, et al, advise the use of business objects when there's a conceptual domain model with business logic and relationship. It's motivated by an increased reusability of business logic and a need to centralize this logic, as seen in figure 27. (Alur, Crupi, & Malks, 2003)

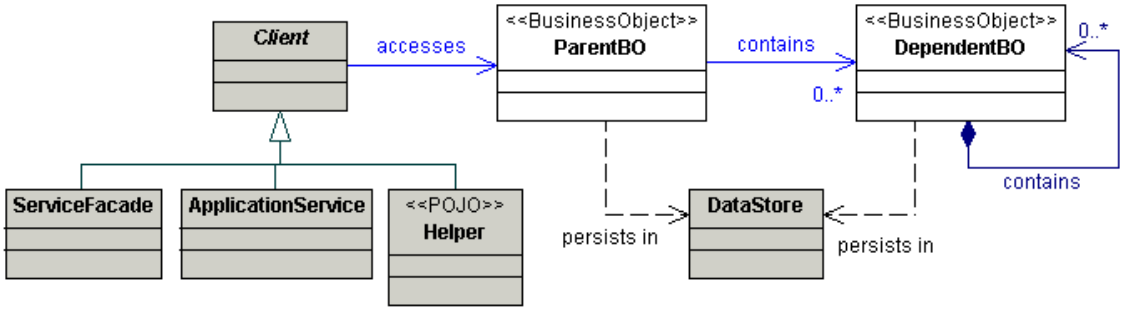


Figure 27. Class diagram for a business object example (Alur, Crupi, & Malks, 2003)

The use of this layer promotes an object-oriented approach to the business model, centralizing any business behavior and state. It may, however, cause stale data and bloated objects. (Alur, Crupi, & Malks, 2003).

As it was described on the persistence layer, all the layers surrounding the business layer are closed, so that it communicates directly with the persistence and presentation layers (Richards, 2015). This can be achieved by having a business operation, that is contained in the business object, invoke one or more data access operations. Since data access operations can be asynchronous, or not, the business operation must abide the same synchronous mechanism as well.

It is also on this layer that error containment mechanisms can be implemented, in order to return a custom error to the user, instead on a less legible one. (Microsoft, 2010)

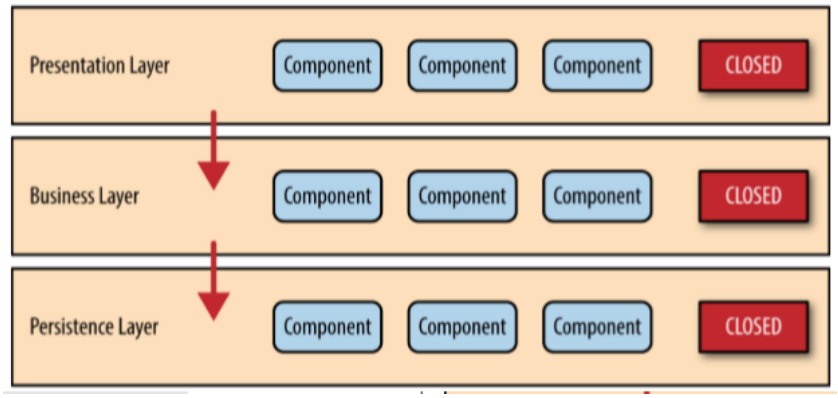


Figure 28. Excerpt of the layer architecture request flow on the business layer (Richards, 2015)

Presentation Layer

The presentation layer, or Web layer as Oracle designates it, is a tier that contains components used to dynamically generate content to the client, receive input and process it, control the flow of pages, and perform some of the logic. (Oracle, 2006) It is also commonly known as the frontend for an application, being built on web technologies such as HTML, JavaScript, CSS and their popular frameworks, communicating through a reference (ASP .NET) or Web API (Application Programming Interface) calls.

As stated by Richards, this layer's responsibilities are the handling of user interface and browser communication logic without having any need to know how to fetch data or process it. Like any of the other layers, cannot communicate with any other besides the business layer, and the requests received, as see in figure 29 (Richards, 2015).

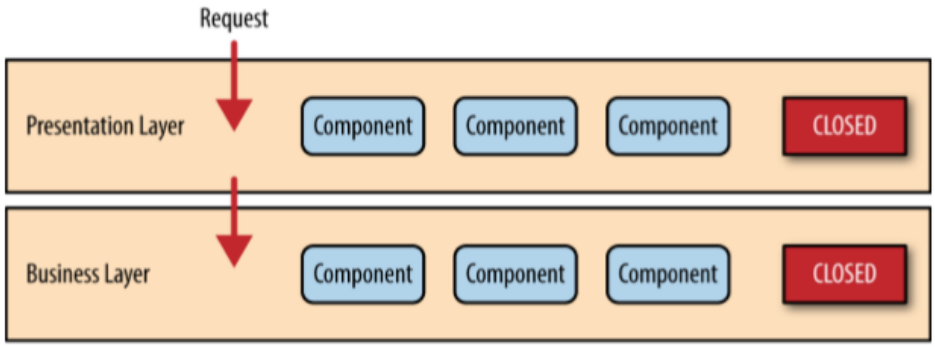


Figure 29. Excerpt of the layer architecture request flow on the presentation layer

While the concept of this layer is broad enough to allow developers some freedom, it may become harder to understand what is expected of a correctly built application. In the following figure (30), the author of the tutorial separates the presentation layer into the client layer, that acts as the interface to the user, and the client presenter layer, that contains the logic for the client application which includes templates and controllers. (Hu, 2012)

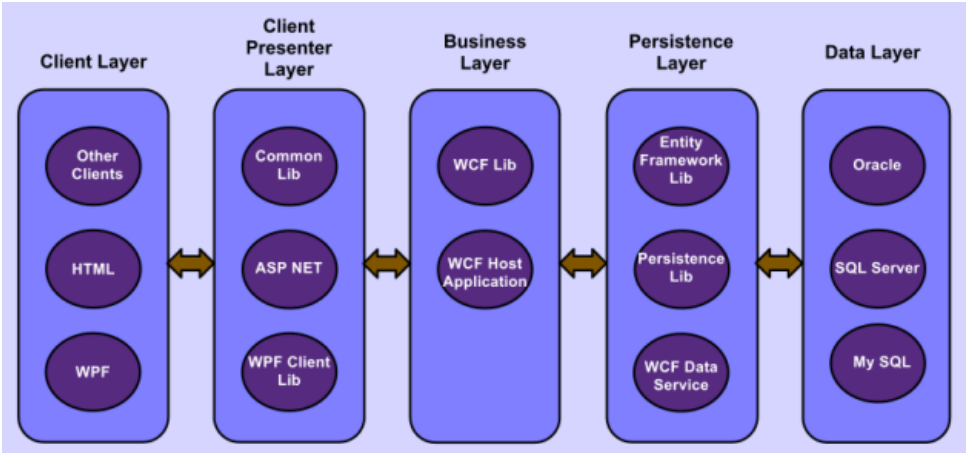


Figure 30. Hu's approach to a layered architecture (Hu, 2012)

3.5.2. Microservices

There are several approaches based on this architecture, each filling several needs the developers have. However, according to Richards, the concept of separately deployed units remains the same. Each unit is developed as a service component, that can vary its granularity level if it's a single purpose module. Communication between services is made through a remote access protocol such as REST (Representational State Transfer) or SOAP (Simple Object Access Protocol). (Richards, 2015)

While it should not be considered in any way as a replace for a monolithic application, as claimed by Richardson, this architecture is used when a system must be highly maintainable and testable, loosely coupled, independently deployable and organized around business capabilities. Otherwise, the developer may be faced with a series of anti-patterns such as:

- Believing that using microservices will solve all the development problems
- Adopting this strategy without knowing the basic development techniques
- Focusing on technology instead of key issues such as service decomposition

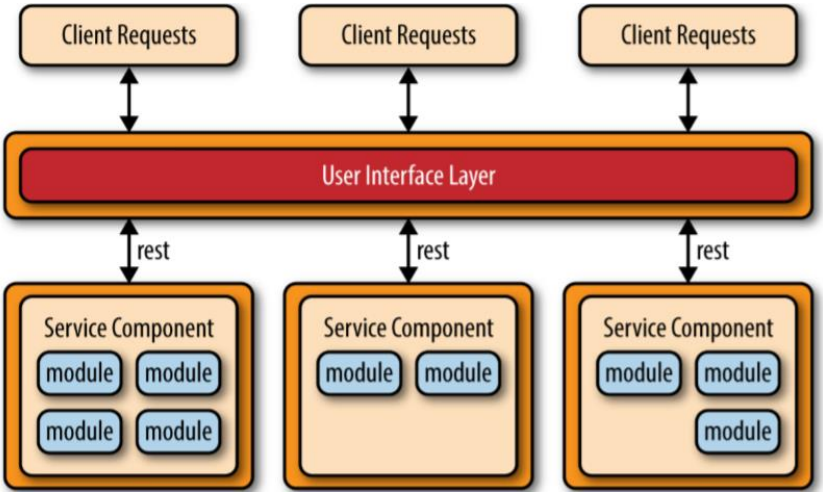


Figure 31. Application REST-based topology (Richards, 2015)

By analyzing Microsoft's guide to developing a CRUD microservice, it is observable that, aside from adding environment variables, it's the same as developing a monolithic application without any layer restrictions and a high granularity level. (Microsoft, 2019)

3.5.3. MVC (Model-View-Controller)

According to Fowler et al, the MVC design pattern is "… one of the most quoted (and most misquoted) patterns around." It's a pattern that separates concerns into three elements, as seen in figure 32, and figure 33. (Fowler, et al., 2002)

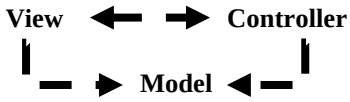


Figure 32. Representation of the MVC pattern (Fowler, et al., 2002)

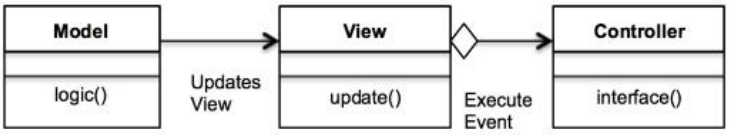


Figure 33. Class diagram representative of the MVC pattern (Giridhar, 2019)

The model, is an object that exists in a certain domain (e.g. books in a bookstore), containing information about it. Gamma, et al, describe the model as the "application object". As previously stated, a model is also known as an entity, as is defined by Vernon.

The view is any visual interface presented to the used that displays the data contained in the model and provides input to the user. Depending on the framework used, a view can be created from a mix of HTML, CSS and JavaScript, or through templates on a DSL (Domain Specific Language) such as Razor, Java Server Pages or one of the many DSLs available online.

The controller contains logic related to how the user input causes a change on the view. It controls the data flow on the model, updating the view when needed, and separates the view from the model. (Vernon, 2013) The following class diagram (figure 34) represents the MVC's structure.

```
public class HomeController : Controller
{
    0 references
    public ActionResult Index()
    {
        var viewModel = new IndexViewModel();
        viewModel.Title = "The Best Title";
        return View(viewModel);
    }
}
```

Figure 34. Example of a controller

3.5.4. Web API (Application Programming Interface)

According to apigee, a web API is an HTTP request and response pattern used by arbitrary software instead of websites. The user can perform a series of operations, executed by a URI (an URL that has variables), based on the available HTTP verbs. The content sent or received through REST or SOAP , usually formatted as a JSON or XML object. (apigee, 2018)

As seen in figure 35, an API is composed by one or more endpoints, each being represented by an HTTP verb, a routing address, and optional input and output values. It's noteworthy that both the input and output types must be serializable (object that can be persisted as a sequence of bits).

```
[Route( template: "api/[controller]")]
[ApiController]
0 references | Fabio, 11 days ago | 1 author, 1 change
public class DefaultController : ControllerBase
{
    // POST: api/Default
    [HttpPost]
    0 references | Fabio, 11 days ago | 1 author, 1 change | 0 requests | 0 exceptions
    public void Post([FromBody] LogOperationResult value)
    {
        OperationResultState.Instance.OperationResults.Add(value);
    }
}
```

Figure 35. An example of an API with one endpoint

There are some guidelines that can be used to develop better APIs.

- The developer should use the correct verbs to execute the operations. POST, GET, PUT and DELETE should be analogous the operations in the acronym CRUD (Create, Read, Update, Delete).
- The developer should avoid using verbs when building the URI, opting by nouns instead (see figure 36).
- Independent of the result, any response to a request should include the HTTP response status code.
- If possible, there should be options to sort, filter, search and paginate.
- If there's more than one version of the API , this should be explicit on the URI.

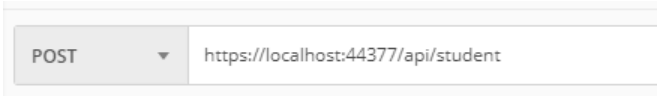


Figure 36. Example of an URI for a POST request

3.6. User Interfaces

An interface is a set of commands or menus through which a user communicates with a program. According to Mátrai, a well-designed interface "... promotes users to complete their everyday tasks in a great extent, particularly users with special needs". (Mátrai, 2010) There are several approaches to a user interface, being the CLI (Command-Line Interface) and GUI (Graphical User Interface) the most prominent ones.

In this section these types of user interface will be addressed in order to understand what can be done for any of the interfaces implemented in the next chapter. Additionally, since it is possible to extend Visual Studio, VSIX (Visual Studio Integration eXtension) will also be subject to research.

3.6.1. CLI (Command-line Interface)

A command-line interface is a form of user interaction based on textual command input. While the common user may find that this kind of interface, Grant justifies it used based on the possibility of easing repeatable tasks through copy/paste actions or script automatization, their ease of development and testing. (Grant, 2016)

Nielsen and Gentner describe command-line interfaces as suited for a community that had expertise and interest in understanding how the system works. These are, however, very rigid and won't tolerate syntax or semantic errors (Nielsen & Gentner, 1996).

There are several success cases of tools whose interfaces are textual, such as git, node package manager and NuGet.

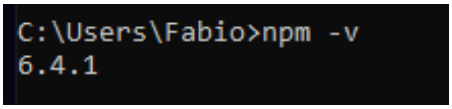


Figure 37. Command-line interface command example

In order to develop a good CLI, according to Kowalski, the developer must follow three principles. First, the user must never be stuck in an operation without knowing what to do next. Second, it should be simple and support power users. Lastly, it should allow the user to do everything possible. (Kowalski, 2017)

3.6.2. GUI (Graphical User Interface)

As defined by Encyclopædia Britannica, a GUI is "… a computer program that enables a person to communicate with a computer through the use of symbols, visual metaphors, and pointing devices" (Levi, 2001). Its use is justified by the reduction of the software learning slope that new users faced in Command-line Interfaces, by sacrificing a bit of control that CLI users have (Nielsen & Gentner, 1996). As such, as users adhere to the use of software, these applications must evolve to cater their needs. Creating a GUI requires developers to follow several principles. Marcus indicates a few of them (Marcus, 1995):

- **Organize**. The application must provide the user with a clear and consistent conceptual structure. This is done to avoid disorder and maintain consistency between the layout, relationships and navigability.
- **Economize**. The application must maximize the effectiveness of a minimal set of cues. The fewer controls, the easier it is to find them. They should not be ambiguous thus making the page easier to understand.
- **Communicate**. The page should match the presentation to the capabilities of the user, being legible and taking into consideration the user's readability, typography, symbolism, and color schemes.

Later on, the design principles would be reinforced with new guidelines for UX / UI that provide tips to build applications that users that functional and cognitive disabilities, or even physical ones. These are known as web accessibility standard guidelines (W3, 2018)

There are many examples of successful GUIs, being the operating systems Windows and MacOS the most notable ones, since their development brings features such as speech recognition, user history and many other interactive applications.

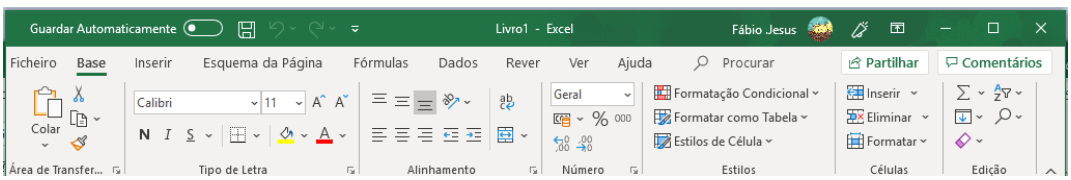


Figure 38. Example of a GUI

3.6.3. VSIX (Visual Studio Integration eXtension)

According to Microsoft, extensions add new features or existing tools to Visual Studio, with the intent of increasing productivity. (Microsoft, 2019)

There are several types of projects such as code analyzers and refactoring tools, that allow the IDE to detect, advise and correct syntax or semantic errors, provide item templates, that can be used to create files or projects according to a specification, and tool windows, similar to the solution explorer, or the output window. (Microsoft, 2017)

Microsoft's take on extensions is based on the fact that the IDE is just an empty shell, as seen on figure 39, and each component the user accesses is an extension, allowing developers to customize their windows with any tools they want, in any location.

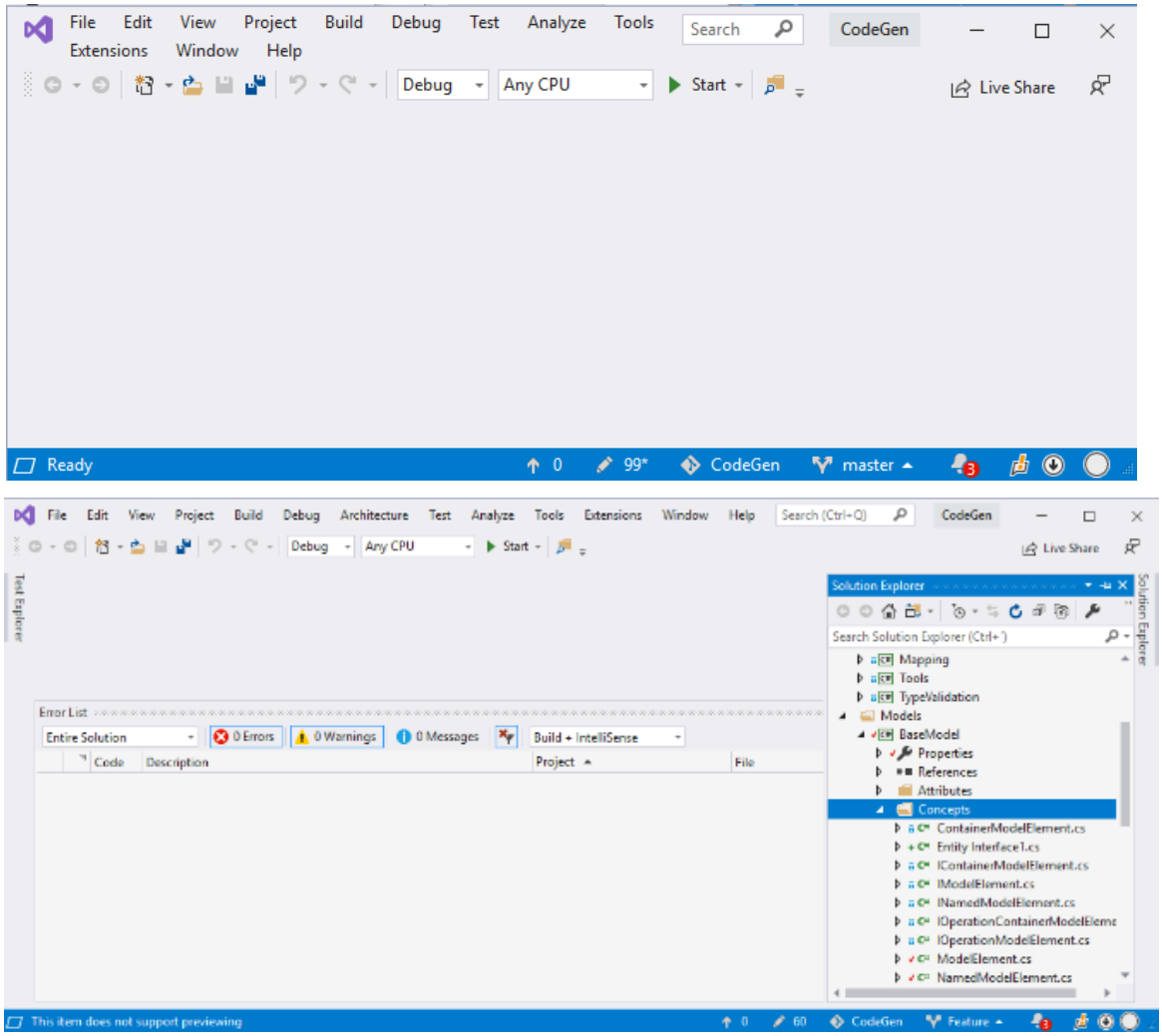


Figure 39. An empty visual studio Shell (top) and a Visual Studio instance with two tools anchored (bottom)

3.7. .NET (Dot NET) Compiler Platform

.NET Compiler Platform, or Roslyn as it was previously known, is a tool that, according to Microsoft, "… creates many opportunities for innovation in areas such as metaprogramming, code generation, …, and embedding of C# and VB in domain specific languages." It allows developers to create code analyzers and fixers, through a set of APIs used to execute the same functions as the standard compiler, but in runtime (Microsoft, 2017).

Together with VSIX, presented in subsection 3.6.3, it's part of the extensibility pack, and as such, it is possible to pack all the created functionalities into a NuGet and share it with other developers so that it can be used to enforce team coding standards or just provide general code guidance. This platform is composed of two main layers, the compiler APIs and workspace APIs, presented on figure 40.

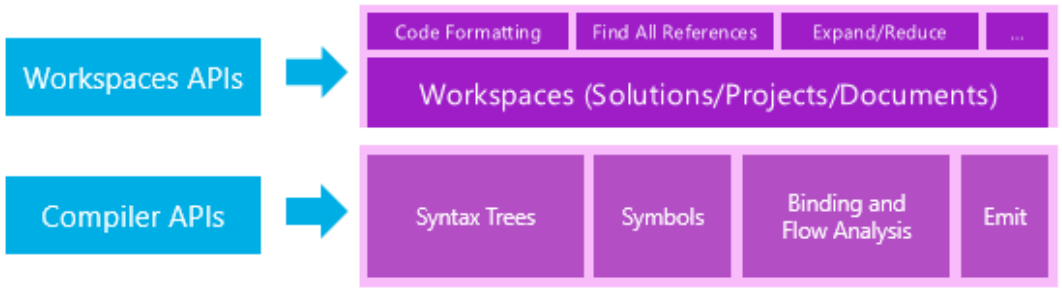


Figure 40. .NET Compiler Platform layers adapted from the source (Microsoft, 2017)

For this research, however, the main focus are the Compiler APIs, that allow code parsing and compilation in runtime, created by joining the model with the templates, bringing the focus to compiler pipeline that, as seen in figure 41, is composed by a parser, symbols and metadata import, a binder and the IL (Intermediate Language) Emitter. Each of these elements is a step on the compiling process, from reading and parsing the code written by the developer to the emission of a DLL containing all the classes and other code artifacts.

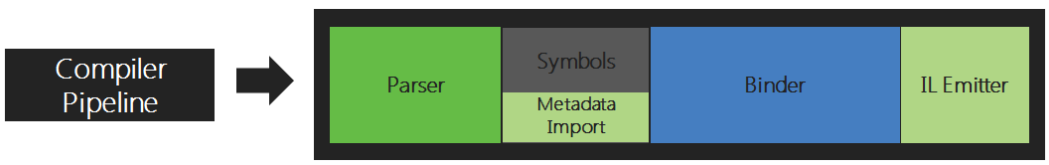


Figure 41. C# and VB compilation pipeline (Microsoft, 2017)

Microsoft states that the compiler API "… contains the object models that correspond to the information exposed at each phase of the compiler pipeline, both syntactic and semantic." This means that there's a way to parse text in order to produce code that can be compiled, import any references needed to execute that code, compile it, and if it fails, get a full report, or create DLL if it succeeds. These APIs can be observed in figure 42.

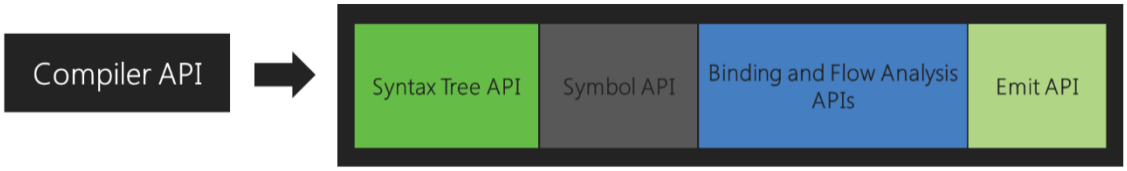


Figure 42. .NET Compiler API (Microsoft, 2017)

Syntax Tree API

Code syntax comes as text. From that text, on the Syntax Tree API, it is possible to build a Syntax tree, that is a syntactic and lexical structure described by three key attributes. First, the syntax tree contains every single lexical token from white spaces to preprocessing directives. Second, a syntax tree resulting from the parser can reproduce the exact content from the source, being possible to create a sub-tree from a node. The third, and last attribute is immutability. When built, syntax trees are a snapshot of the current code (Microsoft, 2017).

Some examples of its use are the colorizing, outlining and formatting features on Visual Studio, and as displayed in the following figure (43), a syntax tree analyzer.

Figure 43. Example of a syntax tree based on code

Symbol API

Based on the syntax trees previously parsed, this API focuses on external assemblies, such as custom DLLs or the ones provided with NuGet packages that were previously installed, and all the symbols available, from a top-down perspective, allowing developers to observe types and members of the code that is being compiled, as if they were accessed through reflection, resulting on a semantic model. It is commonly used for object browsing and navigation (Microsoft, 2017).

Binding and Flow Analysis API

This API is responsible for mapping symbols to source code (Microsoft, 2017). Harrison adds that it also exposes the result of the compiler's semantic analysis so that developers can better understand code logic and dependencies (Harrison, 2017). Some of the uses for this API are autocompletion, "go to definition", rename, and find.

Workspaces API

This API is essential for solution analysis and refactoring (Microsoft, 2017). One of its main features, according to Harrison, is the creation of the context used to manage solutions and projects. (Harrison, 2017).

Emit API

As the last API to be addressed, in the compilation process it is also the last one to be executed. Its uses include diagnostic results, in-memory assembly loading, and file emission. (Microsoft, 2017)

Code generation applications

As observed on the previously presented APIs, this development kit has several APIs with functionalities based on code analysis. While not directly connected generating source code, the use of the Syntax-Tree, Binding and Emit APIs allows developers to compile the code generated before publishing it. For CodeGen developers, its use also reduces the amount of errors produced from T4 templates to the point of producing reliable code without too much maintenance.

Chapter 4.
Prototype Development

While the research is fundamental to delineating what can a code generator do, developing a prototype is the best way to design and concept proof a model based on the knowledge acquired in the previous chapter. Therefore, in the present chapter, each section focuses on a segment of the code generator, from the model to the user interface.

The first section sets objectives and delineates previously observed obstacles, making it possible to discern a viable prototype. Since reusability is one of the most important aspects of this project, the developed features are separated in modules sorted be usefulness in other projects (tools, generic, and specific features).

The second section presents all the tools and algorithms developed to assist the generator and that can be reused throughout other chapters.

The third section uses concepts described in the third chapter and applies them in building an application model. Since there are several elements that can be used in more than one model, this section addresses abstract, generic and specific model elements.

The fourth section presents the work done on every type of template produced, from the ones available as items in Visual Studio to simple file templates for code generation, that can then be joined with the model in the engine, presented in the fifth section to generate code.

Lastly, in the sixth section, each of the user interfaces developed is presented to the user, along with a demonstration (through screenshots) on how the user currently can interact with these interfaces.

4.1. Objectives and Obstacles

As previously stated, the use of code generators has a mixed reception among developers. This occurs, mainly, due to their inflexibility at adding custom code to any component. As such, instead of solely providing the DLL (Dynamic-link Library), it's proposed that the generated code is provided as well.

Another issue, stated by Harrison (Harrison, 2017), is found in implementing a business rule, considering that it is typically a set of constraints related to the domain problem scope. There are several ways to describe these rules such as lookup tables, as Harrison mentions, through a DSL (Domain Specific Language) like OutSystems does, or by parsing text on a natural language through text analysis. For the current prototype, on account of the generated code being provided to the user, these can be provided later.

Implementation-wise, the following situations must be addressed to improve maintainability, and reusability.

- **Value validation** – properties can have constraints to guarantee that the model does not contain invalid content.
- **Model consistency** – each element must have a flag so that the application notifies the user, during import, that the model was changed externally.
- **History** – each element must contain information about its creation and last update.
- **Event Record** – the application should be able to "rollback" changes if an error occurs.
- **Model element reusability** – there are several elements that share several characteristics. These elements can be developed using inheritance and generics to not only reduce the amount of code produced but the number of elements as well.
- **Scalability** – There are certain features that can be reused throughout other applications. The prototype developed should be as modular as possible.

The concept for the application is available at annex 2 as a component diagram.

4.2. Common Libraries

There are several features that are used throughout the project and can be used in other ones as well. The main objective for the common libraries is the development of a series of features, sorted by the possibility of being reused. These modules can then be provided as NuGets. As an example, while the file manager, the cloner and the dependency manager are both tools, the odds of handling files or cloning an object are greater than the ones for the dependencies. Therefore, the dependency manager belongs in a different module. Additional information about the common libraries is available at annex 3.

4.2.1. Base

Composed by elements common to all solutions, this module's main objective is the extension of the features available at pre-existing types and the representation of an all-purpose return object for all operations.

4.2.2. Tools

This module contains operations that can be employed throughout other projects, motivated by the reduction of code written and feature centralization. Each of these methods must return an Operation Result that contains the return value and a message. The following tools can be used by including this library in a project:

- **Cloning** – Copies the object by value. This cloning process can be executed if the object has a serializable attribute or implements the Serializable interface.
- **Hasher** –This tool creates a hash value based on a character string (typically the stringified object) and the algorithm of choice, which, in this case is the MD5 algorithm a hexadecimal value. It is fundamental to make sure that an imported application model has not been changed outside the application.
- **File Input / Output** – set of methods to create, read, delete, check and write files, wrapped in a disposable context so that it can be deleted when it's not needed.
- **Idiomatics** – a simple library used to add features connected to linguistics and text processing. Its main use is an algorithm that, according to a set of rules(Grammarly, 2017), provides the plural of a singular word.

- **Processes** – Executes a set of commands with Powershell, returning the result of the command call. While it eases the process of creating projects, it is not the most viable approach since the process execution may not be terminated "freezing" the application.

- **Settings Management** – There are instances where certain data depends on user's preferences. This is an element that can be attached to an application, and work as a customizable object, much like the resources used in visual studio projects. In the Visual Studio extension, this is replaced by the Visual Studio Options.

4.2.3. Dependency Manager

It is quite common for a model to have cyclical redundancies. This occurs because there are elements that can be related to two other elements related to each other (e.g. an entity exists in a data layer, and a data access object depends on the entity, thus depending on the data layer as well). Another hindrance caused by this sort of relationships is the production of excessive content on an exported model, as the one displayed in figure 44, where an object contains two other objects (A and B), each containing the same object (C).

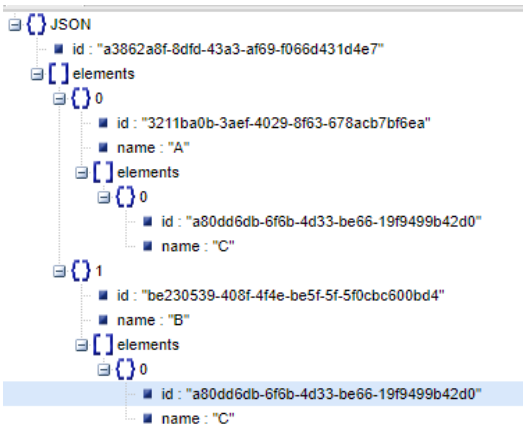


Figure 44. JSON object exploration. The object contains A and B, that each contain C.

To address these two issues, the Dependency Manager the concepts of "Depender" and "Dependee". These can be used to create a relationship between two elements without referencing them directly. Based on the previous example, the entity would be a "Dependee" to the data layer and a "Depender" to the data access object.

These two concepts along with the manager can provide a dependency tree for the application model that can be used, as described further into this chapter, to ease import and export operations. The expected result from a model exportation with this approach, as displayed on figure 45, is a list of elements disconnected from each other and a list of relationships between elements, identified by both the dependee and depender's ids.

Figure 45. The same object with the objects related as dependencies

4.2.4. State Manager

No application is safe from errors that may cause unsaved changes to be lost. To protect the user against these situations, a state manager is implemented, in a way that a change stores the current "state" of the model. This state can be used so that the user can reload the model if the application crashes or is closed. Aside from this feature, it is also possible to undo and redo actions. It is also possible to "pick a state" if the undo operation is executed twice, as seen in the following figure (figure 46), where the node named "C" can be restored to either the node "E" or the node "F".

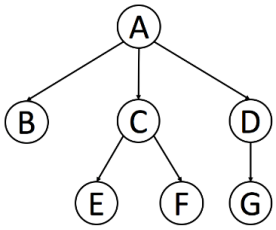


Figure 46. A representation of an n-ary tree

4.2.5. Type Validation

Properties and model elements may need custom validation. The following type validators were developed, based on the concepts used on form elements, such as credit cards and postal codes. Any validator checks if the value is set.

- **Data validation** – compares the property's value as a date to an inputted date. If there's no value to compare, the algorithm checks if the value is not the same as the minimum or maximum possible value.
- **Id Validation** - If the id's a UUID the validator checks if the value is the same as an empty id. Otherwise, the algorithm checks if the value is greater than 0 (zero).
- **Interval validation** - checks if the property's value is within a range of two real numbers. The range can include or exclude the extremes.
- **Regex validation** - matches the property´s value with an inputted regex pattern.

4.2.6. Mapping

While the dependency manager reduces the amount of repeated content on a model exportation, there are properties that do not need to be exported. Additionally, since conventional instance creators such as Microsoft's Activator will use the class' constructor instantiate an imported model, not only must the properties be in the right argument order, but also with the right type. To solve this issue, a mapper capable of instantiating objects through an attribute-based specification was developed. To be mappable, a class must be serializable, and the amount of arguments in the specified constructor must match a set of properties with the correct type and name, as seen in the following figure, that uses attributes to describe properties and constructors.

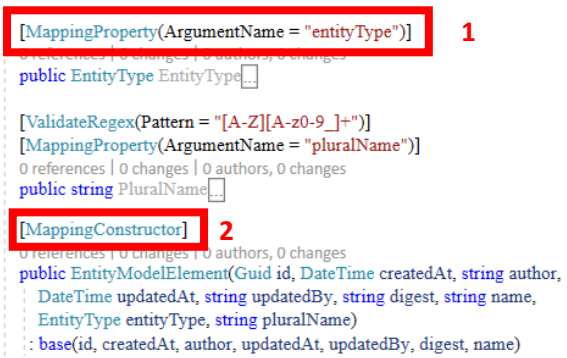


Figure 47. Mapping association where 1 is a reference mapping and 2 is the mapping constructor

4.2.7. Compilation

The compilation library contains a set of features that allow the developer to build the whole solution into memory, and the execution of .NET commands, such as project and solution generation. With the .Net Compiler platform, it is also possible to test, to a certain point, the generated code before producing any artifact, if the templates contain C# code. For this purpose, this library contains a set of features that allow the execution of .NET commands, manage solutions, projects and compile code in runtime.

To simplify runtime compilation, from the developer's perspective, the process was segmented into three services, the syntax tree service that parses the rendered template, the metareference service that manages dependencies and third-party NuGets, and the compilation service that generates the assemble and stores it as a reference.

Syntax tree Service

As previously presented on chapter 3 (three), section 7 (seven), syntax trees are containers for every lexical node in the code. The first step of the compilation process is the creation of syntax trees. It is at this point that the model is joined with the template to produce text. Through the syntax tree service, this text is parsed, with the assistance of the .NET Compiler Platform, to produce said container (see figure 43).

Metareference Service

In order for a project to use the features of another, there's a need to link, or "reference", them. Through the traditional means, this is achieved by either installing a NuGet package or reference a project (see figure 48).

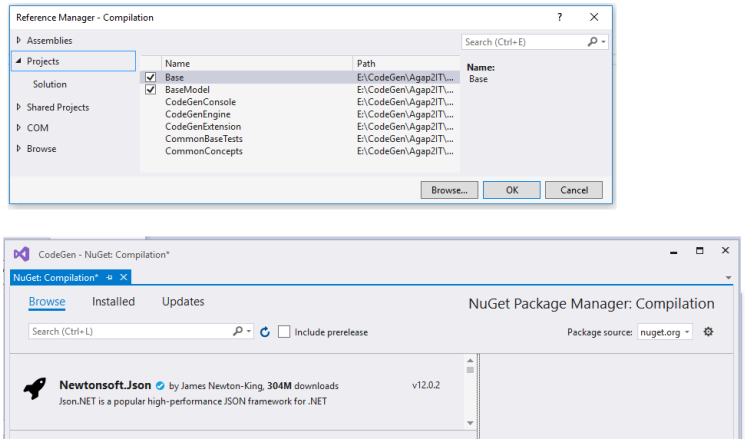


Figure 48. Project reference via Visual Studio (Top) and Nuget manager (Bottom)

During code generation and in-memory execution, however, this is not possible because the user needs to manually search and install NuGets or identify the projects that need to be referred.

This, however, can be tackled with the .NET Compiler Platform by using a series of Metareference related features that allows the generation of a metadata reference from a file or an assembly. Since it is possible to create assemblies from generated code in runtime, this will also be the main component used to link generated projects (layers).

One of the main validations required is duplicates, whose imports could cause an inner uncontrollable failure. To solve this issue, once again due to the lack of documentation, further study was required to find a way to differentiate assemblies, being the Display property essential to solve this issue.

Compilation Service

The final piece for runtime code execution is the compilation service. By adding the syntax trees and the metadata references to a specification of a compilation instance, it is possible to generate an assembly that can then be built. During the build process, a series of evaluations is performed to check if any reference is missing, or if any syntactic node is invalid. These validations can also be exported to a log object that can be stored in a file.

4.3. Application Model

Each application model follows an architecture pattern composed por other elements that may obey to a certain design pattern and result in a component that could be developed through a set of well-defined steps (e.g. creating solutions and projects).

Due to the nature of the tool, an effort is made to keep the application model as abstract of target language and architectural concepts as possible. Such attempt implies that even with a significant amount of model elements, many can be reused in different application models, such as the example provided below (figure 49) of a property with annotation and one without.

Figure 49. A property with an annotation (top) and a property without annotation (bottom)

This section focuses on the application model, the most arduous concept in this project. To reduce the amount of work that it takes to create a model element, a set of abstract, generic and concrete elements were created. Since there are also certain elements that can be reused in more than one application model, these are distributed in two modules, the "abstract model elements" that keeps most abstract, generic and reusable elements, and the "layered model elements", that has model elements related to the layered architecture. Additional information on the application model is available in annex 4.

4.3.1. Abstract model

A model's complexity is often found on the lack of property analysis engagement, which results in entities with the same properties. While it does increase the number of classes in the project, these tend to be small and easy to maintain and, as such, is the best strategy to adopt in this case.

Model Element

Any model element needs to be controlled in a way that the developer knows who created it or changed it, and when. There are also a few operations that all elements can execute. These are indicatives of a core abstraction, dubbed "Model Element", that contains the Id, creation and update timestamp and username, and a hash value. Additionally, as previously stated, it also contains operations that registers a change (update the update username and timestamp), converts the element to a JSON object and two others that validate the element (see subsubsection 4.3.4). In order to maintain a purist view, and increase maintenance on the notion of model, these operations are implemented in a support class.

Named Model Element

Besides the common properties, there are several elements that have a name, such as entities and properties. A named model element is valid as the name is valid, according to a regular expression that only accepts single words composed by characters on the alphabet. It inherits the Model Element.

Project Base Model Element

There are certain elements that generate containers such as solutions and projects. All elements of this variety have namespaces, workspaces (stores temporary DLLs) and publishing paths (stores the final product). The namespace is valid if it is composed by

terms with alphanumerical characters separated by dots (.), and the work path and publish path are valid Windows paths. It inherits the Named Model Element.

Generic Model Element

There are several model elements that have a limited amount of elements of a certain type, such as operations that match the CRUD (Create, Read, Update, Delete) operations commonly used. A generic element has a flag that, when set, build one instance for each subelement available by default.

Operation Model Element

The operation Container Model Element uses generics to avoid resorting to the nuclear "object" type. This model element type is the base for the data access, business and any other future operation model elements. It inherits the Named Model Element.

4.3.2. Conceptual Model

There are as many model elements as the ones needed to build a specific application in any of the available architectures and design patterns. For the purpose of this prototype, as previously described, the focus is the development of layered applications with a Web application and/or an API. The following figure (51) presents the proposed structure for the layered application model.

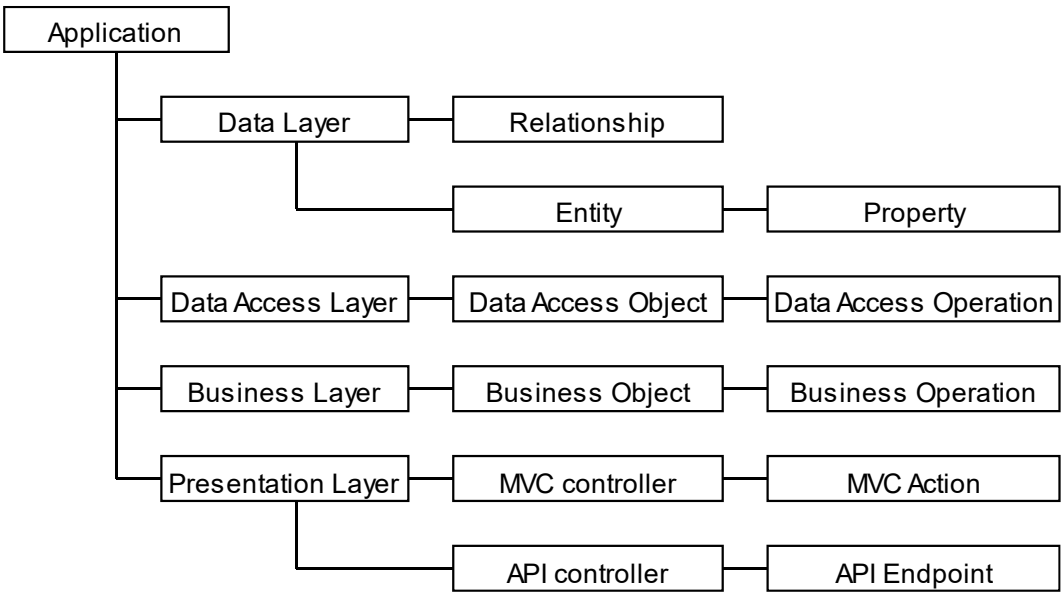


Figure 50. Hierarchical model element structure

API Controller Model Element

An aggregate of endpoints used to create an API. It's associated to an entity and can be generic, provided that the business and data access objects are as well. It's valid when it is associated to a valid entity and business object.

API Endpoint Model Element

An operation executed by the user. It is valid if it has a valid path, HTTP verb and business operation.

Solution Model Element

The root element for all applications. It is composed by one of each layer type and carries settings that are common amongst them. A solution is valid when all its layers are associated to a non-null object and the name, namespace and workspace are valid text.

Business Layer Model Element

A layer that aggregates business objects. It can be generic if the data access layer is generic as well. A business layer is valid if it has a valid data layer associated, and valid name, namespace and workspace.

Business Object Model Element

An aggregate of operations, that operate over an associated entity. It can be generic if the data access object is generic as well.

Business Operation Model Element

Business operations are executed by a controller's method on an entity by executing a set of well-defined business rules. A business operation is valid if it has valid name and operation type and has a data access operation of the same type.

Data Access Layer Model Element

A layer that aggregates data access objects. It can be generic, providing persistency operations to any entity. It's valid if it has a valid name, namespace and workspace.

Data Access Object Model Element

An aggregate of persistency operations, that operate over an associated entity. It can be generic, serving list, create, read, update and delete operations to an entity. It's valid if it has an entity associated.

Data Access Operation Model Element

Actions taken over an entity. These operations are executed by a business operation and act over the persistence of the object. A data access operation is valid if it has a valid name and operation type.

Data Layer Model Element

The layer that contains the model and its relationships. It aggregates relationships and entities. The data layer is valid if it has at least one entity.

Entity Model Element

An entity is a class that aggregates a set of properties that represent, to a certain level of abstraction, something in a dimension (e.g. a person). It's valid if it has a valid name, plural name and entity type.

MVC Action Model Element

Server-side actions executed when the user interacts with the application, usually by clicking on links or buttons. An action is valid if it has a valid name and has a business operation associated to it.

MVC Controller Model Element

An aggregate of actions used to create an MVC application. It is associated to an entity and a business object. The controller can be generic, granted that the business layer is generic as well.

Presentation Layer Model Element

The layer that contains API controllers and MVC controllers. This is the one of the most complicated layers to manage because there's no strict rules to it, and the controllers can range from none to more than the double amount of entities (one MVC controller and one API controller for each entity). The presentation layer can also be generic, granted that all layers below, except for the data layer, are generic as well. In this case, the user should determine whether there should be only MVC controllers, API controllers, or both.

Property Model Element

A property represents a characteristic of something in a dimension. (e.g. a person's name). A property is valid if it has a valid name and a valid data type.

4.3.3. Model Container

As previously stated in the Dependency Manager and Mapping sections, importing and exporting has several issues caused by cyclical redundancies and repeated content. Element instances (e.g. a property model element named “Name”) can exist in several other elements, which implies that, during an exportation, if two elements had the same property, it would be exported twice.

Since all elements are associated with, at least, one other element, the model container abstracts from that relationship and keeps each element on a hash map, whose key is the value provided in a Model Member attribute, as seen in the following figure.

```
[ModelMember(Key = "547c19a4-b978-44d8-92b2-d34766fef100")]
21 references
public class PropertyModelElement : NamedModelElement
[ModelMember(Key = "938a7198-ccb9-428e-afc2-f9d74f8ed0b8")]
70 references
public class EntityModelElement : NamedModelElement
```

Figure 51.Two model elements, with different model member keys

The result of having, for example, two properties and one entity instantiated in the element would result in the Model Container having two keys, one for the property model elements, associated to a list with the two properties, and another one for the entity model elements, associated to a list with the entity.

4.3.4. Model Support

With several features such as validation, change triggers available to every element, the developer would need to implement each of these into all the new elements created. If these methods are implemented in the abstract model element, when executed, will lose all the properties that the concrete element has.

Therefore, using reflection and generic techniques, it is possible to create methods that can be executed by the abstract model element by upcasting it to the concrete one.

Validate data

Data validation is a process based the use of specific data type attributes, presented in the Type Validation library (subsection 4.2.5), associated with each property, ranging regular expressions to numerical limits. The validation is performed, resulting in an operation result that indicates which properties have an invalid value and why.

Register a change

Registering a change updates the information about the last change to the model element. This way, it's always visible who had the last interaction with it and when. Since the element itself may be related to another (as a dependee), each depender will be updated as well, and so forth, as seen in the following figure (52).

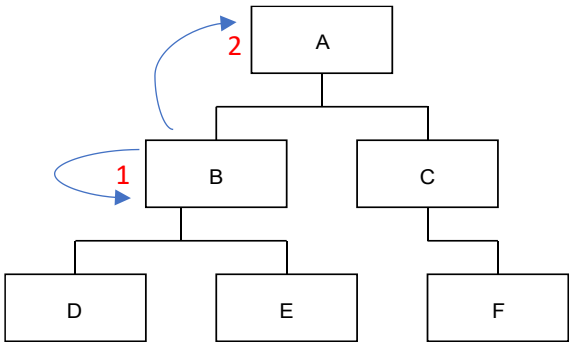


Figure 52. Model element change registration with depender update

Generate hash value

Hash values are a way to guarantee that external changes to the model are noticeable. Since the value is based on the hash value of the one at each dependee, when it's generated, each of the dependees' hash values should be generated as well, as seen in the following figure (53).

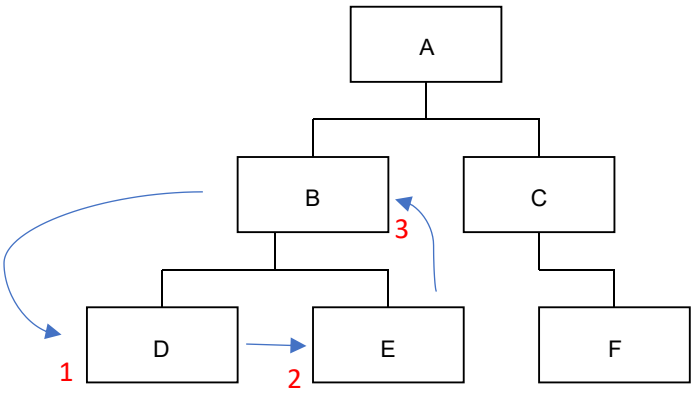


Figure 53. Model element hash value generation process

4.3.5. Model Controller

While there are several mechanisms to handle the model elements, these features end up scattered amongst the common libraries and the model container. Reducing the amount of interactions needed to manage the application model is the next step to homogenize the application model, resulting in the model controller.

This controller is responsible for model element management operations such adding, fetching, updating and removing elements, by using the model container and the dependency manager. Besides these operations, other generic ones such as validation, hashing and change registration are also delegated to the controller to reduce the amount of code in any model element, which in turn are executed by the Model Support. To ease the description of model element management, relations are described as parent and child elements.

Handling a root element

The root element is the main element on a solution. Only one element of its type can exist in the application model. The model controller can fetch and set it in the container, granted that there's no root set with a different id.

Registering a model element

A non-root model element is registered as a composition or as an aggregate. Registering a composition requires the existence of the child elements to the model container. The parent element is only added at the end of the composition.

If the registration is an aggregation, the parent must exist, and the children are added to the container at the end of the aggregation. As an example, the data layer aggregates entities and relationships, and the relationship is composed by two entities.

Listing model elements

If the parent aggregates children, then the children can be listed. This can be achieved by fetching all the parent's dependencies related to the property that returns the list. The resulting list is filtered to return only objects of type "B".

Fetching a model element

Fetching elements also depend on the relationship between the two types. If the child is aggregated to the parent, an id is required to filter the result. Otherwise, if the relationship is by composition, the id should be provided by the composed object (a relationship has two private properties with each of the entities' id.

Updating a model element

Updates and registrations are very similar, having the same validation with the exception that the updated element can only have the same values as another element of the same type and key if the id is the same. The container replaces the object with the new one.

Removing a model element

A model element cannot be removed if it has dependees (e.g. a relationship can be removed from the data layer without restrictions, but an entity can only be removed if it does not have relationships or properties. It could be possible to perform a cascade delete.

4.4. Templates

As previously observed in the third chapter, templates are mostly used to wrap a model element with syntax from a language. It's a straightforward process where the developer engages in T4 template writing, a preprocessing toolkit used to generate said templates. Additional information on the templates is available in annex 5.

4.4.1. Developing a template

The process of developing a template acts as a reflection over the model element(s) used to generate the code, because it will show the developer whether the chosen properties while abstracting from the result to the model are enough. This can be easily proved in an entity template, which is easier if the entity has a property model element accessor. It is also in the template that the complexity of the code produced and the choice of template engine are evident because a minor property may involve a large amount of code or logical operations applied to the template, that may be undesirable, as is the case of the backing field on a property as seen on the following figure (figure 54).

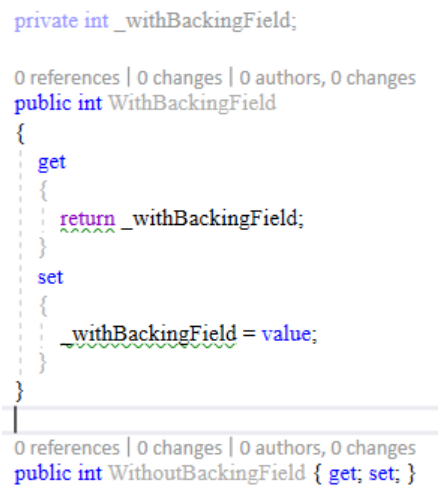


Figure 54. Comparison of code produced by a property with a backing field (top) and another without (bottom)

Preferably, there are as many templates as there are model elements, which would be indicative of a higher maintainability index due to template composition (e.g. an entity template is composed by property and relationship templates). This can't be easily achieved because these templates some restrictions and its confusing syntax may make it harder for developers to manage indentation and line orientation. Nonetheless, in order to reduce the amount of properties requested to the user, several other features are required, such as the capacity to format the name of a property to the one advised for private variables (with the "_" prefix and the first character lower case) or the one for properties (capitalized name).

To that intent, besides the templates, a template base was created, so that besides these features, it could also be possible to implement a template controller that could generate the template as many times as needed while cleaning the result (an instance of a template is built to generate content only once). Such action cannot be easily achieved, requiring the creation of complementary code that already existed in other templates. This allows the generated code to be easily indented (since spaces in the template count as produced spaces) by using an indentation method, and by generating template content inside other templates (properties inside entities).

4.4.2. Templates in CodeGen

For this prototype, while it is possible to use the template syntax to produce any language as long the user knows how to do so, only C# code is addressed. This is mainly due to it being the step before executing code in runtime, which is something available

for C# at the moment of this document's version. The following elements are generated through templates:

- **Entity** – Based on the "data layer" element for the namespace, the "entity" element for every property, the "property" element for every property, and the "relationship" element for a list of the other entity if it is one-to-many or many to many, or the reference to the other entity if it is one-to-one or many-to-one.
- **Database Context** – Using components of the solution (name), data access layer (namespace) and entities at the data layer, a list of datasets and the database connection is generation for a local database.
- **Business Object** – Based on the "business layer" for the namespace, the "entity" for its name, the "business object" for its properties, and the "business operation" for the operation's information.
- **Data Access Object** – Based on the "data access layer" for the namespace, the "entity" for its name, the "data access object" for its properties, and the "data access operation" for the operation's information.
- **API Controller** – Based on the "presentation layer" for the namespace, "entity" for its name, the "API controller" for its properties, and the "API endpoint" for the endpoints' information.
- **MVC Controller** – Based on the "presentation layer" for the namespace, "entity" for its name, the "MVC controller" for its properties, and the "MVC action" for the actions' information. Since the MVC is supposed to be a web application, there are also templates for listing and detail views.
- **Solution –** To wrap all the layers in a single container, a solution is required. It contains information about folders, projects and global settings.
- **Project** – Any layer available in the solution is a project that can be either an application (console or web), or a library (dll). These are composed by the framework used, dependencies, files and folders, and debug / compilation options.

While the same result could be achieved with literal templates, it could require further development and testing, and would display a less attractive syntax.

Besides code templates, Visual Studio Extensions allow developers to produce item and project templates, that, while are not as advanced as the ones produces with CodeGen, can also be used on any project, as seen below on the following figures where, in figure 54 Visual Studio presents the available Visual C# items, one of which is the Entity Interface that, when added, produces the code presented on figure 55.

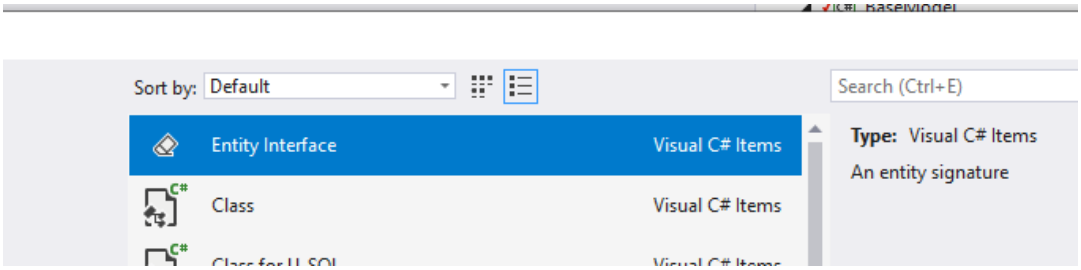


Figure 55. A list of available Visual C# items

```
using System;

namespace Agap2IT.RD.CodeGen.Models.BaseModel.Concepts
{
    0 references | 0 changes | 0 authors, 0 changes
    public interface Entity_Interface1
    {
        0 references | 0 changes | 0 authors, 0 changes
        Guid Id { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        bool Deleted { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        DateTime CreatedAt { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        DateTime UpdatedAt { get; set; }
    }
}
```

Figure 56. The code produced in the Entity Interface Visual Studio Item

These templates are better suited for Generic objects such as data access and business object due to the little to no control of what can be added dynamically. The following templates were created:

- **Generic Data Access Object** – A data access object that has sync and async methods commonly used in any other objects.
- **Generic Business Object** – A business object that has sync and async methods commonly used in any other objects.

- **Unique Entity Interface** – An interface that describes unique entities
- **Enumerable Entity Interface** – An interface that describes enumerable entities.
- **Unique Entity** – An entity base composed by a unique identifier and created, and updated, dates.
- **Enumerable Entity** – An entity base composed by an integer identifier.
- **Data Layer** – A project that already contains some of the elements that are available at most, if not all, data layers.
- **Data Access Layer** – A project that already contains some of the elements that are available at most, if not all, data access layers.

4.5. Generation Engine

Unlike any of the components built up to this point, whose main objective is to be as generic or abstract as possible, the engine is the complete opposite. There should be one for each specification, with the possibility to shift between programming languages.

The engine's main objective is the reduction of interactions required for the user to build an application. It should also be possible for the engine to be a possible user interface for developers by using it as a library. Nonetheless, due to the strongly typed nature of C#, file generation and emission is still required because the user won't have access to the properties and method of the runtime code execution due to the type being generated in runtime as well.

It is also in the generator that other components developed are joined together, with the application model, the compilation library and the templates being the central piece. Therefore, the generator, as a library that can be used by a developer as a user interface, should be able to perform the following actions:

- Create a new model
- Import an existing model
- Export a model
- Manage elements in a way that the complexity of the container and dependency manager are not visible to the developer.

- Perform tests
- Instantiate object of a generated type

Together with the layered model elements, templates, and additional features, the layered application engine is the core for the prototype.

4.6. User interfaces

The final step to developing a user-friendly application, is the interface. As previously observed in the third chapter, there are several different interfaces available.

While there are several potential user interfaces that can be built, besides the library, only the Command-line Interface and the extension where produced to showcase relevant features to the prototype.

At the moment of the prototype's development, Microsoft's technological stack as changed from Visual Studio 2017 to 2019, and with that, the way extensions work also changed. While the development started as a 2017's extension, what already had been created wouldn't work for 2019, and it was an error-prone process that was discarded due to the need to acquire additional know-how about VSIX development. Nonetheless, trials for a 2019's version eventually started, to access if it was a viable opportunity. Additional information on the user interfaces is available in annex 6.

4.6.1. Command-line Interface

Developing a command line interface is an arduous task because it needs to be easy to use, but it doesn't generally maintain a state or allows any interaction other than the typical keyboard input. While building a set of commands is important, the inclusion of flags and options, as observed in the previous chapter, can reduce the number of queries posed to the user.

Still, the common user interface is displeasing to this type of projects, and as such, the following types of queries were developed to increase the dynamism of the application:

- Line reader – receives the user input, key by key, until the "escape" or "enter" keys are inputted. The escape key terminates the input without a successful result.

- Text request – continuously presents the query to the user until the inputted value matches the requested pattern.

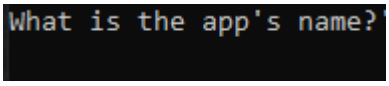


Figure 57. An example of a text request

- Number request – similar to the text request, itcontinuously requests the query to the user until a number in a range is inputted.
- Pick – the user is presented to a series of options from which one can be selected by using the up and down arrows

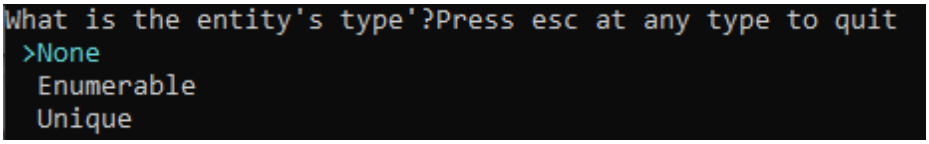


Figure 58. An example of a pick

- Multi-pick – The same as the pick, but the user may choose none or more options.
- YN request – The request is presented until the user either presses the Y or N key.

Along with the command parser developed the CLI, while not being near perfect, is slightly better than a series of console output and input operations.

4.6.2. Visual Studio Integration Extension

The Visual Studio extension is one of the most challenging components to develop since that, while it does resemble a Windows Presentation Forms application, there's no window class to modify. Therefore, it was required to readapt the Model-View-ViewModel pattern to provide a sort of navigability in this kind of projects.

Although there's still little to no documentation available for developers about this kind of projects, there were some significant changes, that changed the expected result when the research began, such as the lack possibility to organize menus, asynchronous tools and the "hot reload" mode that allows non-significant changes to be made to the extension during execution. One of the most interesting, and troublesome features were the styles, that while it was possible to synchronize colors with the Visual Studio instance,

there's no information on what elements to use, and the application tends to crash often for an unknown reason in certain machines. Nonetheless, a viable product was developed to display its capabilities, as seen in the following figure.

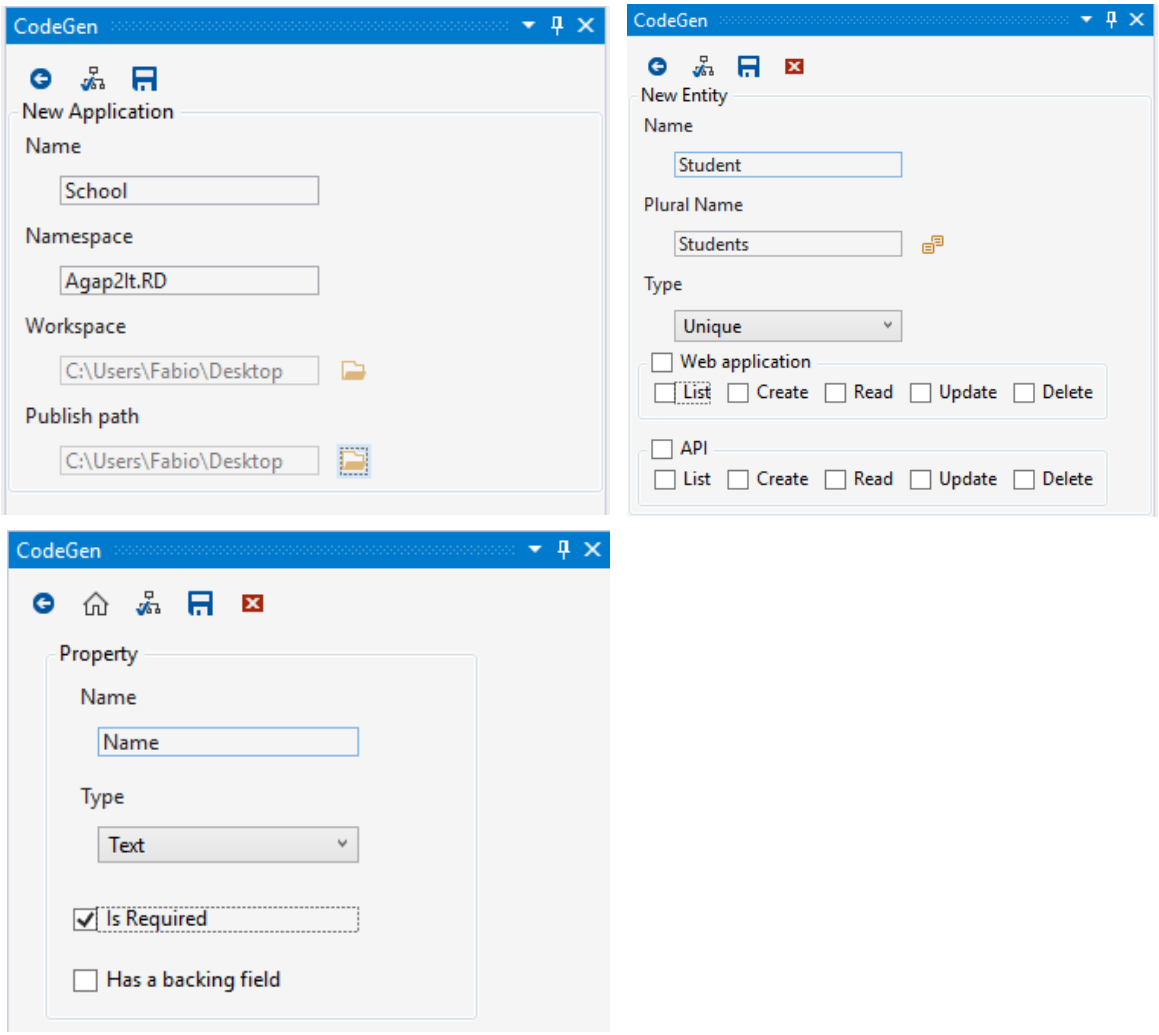


Figure 59. Creating an application, entity and property in the Visual Studio extension

Chapter 5.
Conclusion

As the last chapter of a research, the conclusion reviews both the research process and the prototype's development. With this review it is possible to draw some conclusions about code generation, the effectiveness of the common libraries, the design of the application model and the correct use of templates.

The first section of this chapter is based on what was observed during the research and developed in the prototype to pinpoint the current situation, slightly describing the wrongs and rights of the process.

Based on the results obtained of the tests performed, and the result of the prototype's execution, an overall appreciation can be performed, in a way that research questions posed in the first chapter can be answered.

Lastly, in the third section, based on the other sections of this chapter, a path is traced defining what can be made during the next interaction with the project.

5.1. Point of situation

As of this delivery, the proposed research was performed, acquiring knowledge on the generation of code for Java and C# in different architectures. While the prototype has been developed, being able to generate simple ASP .NET applications in a Layered Architecture, the lack of documentation on the .Net Compiler Platform and Visual Studio extensions hinders further implementation. Both the emission and the runtime code execution module, perform as expected, requiring further development to improve the resulting libraries' customization.

5.2. Overall appreciation

During the research, it is observed that code generation is a process that has both negative and positive connotations, depending on who does it, and with what intent. These observations were important to determine the path taken.

Currently, the best way to describe an application is the use of models, as observed in sections 2.1 and 3.1, where the use of a model driven approach allows the development of a "form" per se, that can be filled with any kind of data provided by the end user. Therefore, the model's complexity can be adjusted to either ease development of most simple applications, by being lowered, or to allow developers to build all sorts of applications. Since there's already some modularity to the model itself, instead of solidifying this complexity straight to the application, different kinds of models could be distributed, along with the respective templates, engines and interface modules.

While templates must, still, be developed through a difficult process, due to the lack of any Intellisense, having the possibility to inherit and compose templates eases development and maintenance for new templates. As an alternative to using the generator, the Visual Studio templates allow the user to add most of the code that can be generated in an already existing solution.

As for the generator itself, as described in section 4.5, is a specific piece of software that can only be used with the layered architecture as it acts as the middle man for the the user interface and a combination of the Model Controller, Templates and other support features. Therefore, when a new type of project is proposed, such as a microservice, a new engine should be developed as well.

Finally, on the user interfaces, the first trials for a Visual Studio extension were not successful, and as such, only the CLI was taken in consideration. The first version of the CLI was designed in a way that the user could select items answer to binary response questions and validate input through custom commands. Since this kind of interfaces still rely too much on interaction through the means of questions to fill the model, flags and switches were added in order reduce the number of questions needed to execute a command. Eventually, as stated in section 3.6.3, Microsoft's changed their technological stack, so a new attempt on the Visual Studio Extension was performed, performing adequately for a prototype.

5.3. Future work

As a project that will be used in a professional setting, a series of heavy-duty tests need to be made in order to fix any issue that wasn't previously detected. This project's development will continue with several interactions of analysis and production of new components, depending on the users' needs, including changes to any of the interfaces. The models can be updated, and with a surge of new programming languages, frameworks and architectural options each day, future interactions can include support for these options.

The process of reverse engineering is also a proposed interaction since the .Net Compiler Platform not only compiles code, but also allows developers to observe a syntax node tree that, when properly analyzed, can be used to build a model. This process, however, can only, currently, be achieved for C#. Other development possibilities include the generation of code based on a text paragraph, through automatic text extraction techniques.

Maintenance-wise, there's a need to increate template development productivity, which could be achieved by switching to a text generator with a more "user-friendly" syntax. Additionally, since the prototype only has one type of application model, the generic model element features should be validated with other types of architectural and design patterns.

Since the IDE extension was not completed, further development is required. The lack of documentation and support for newer features increase the learning slope. Such hassle makes it impractical when facing the available time for a delivery. Nonetheless, with the

proper investment and practice, this difficult and often frustrating type of project can be used to create a better interface for CodeGen that can be incorporated in the IDE instead of relying on a separate application.

References

Alur, D., Crupi, J., & Malks, D. (2003). Core J2EE Patterns : Best practices and design strategies. Prentice Hall / Sun Microsystems Press.

apigee. (2018). *Web API Design: The Missing Link.* Google.

Bergenthal, J. (2011). *Final Report, Model Based Engineering (MBE).* Arlington: NIDA.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide.* Addison-Wesley.

Cabot, J., Brambilla, M., & Manuel, W. (2017). *Model-Driven Software Engineering in Practice.* San Rafael, CA: Morgan & Claypool Publishers.

Crockford, D. (2003, March 1). *Introducing JSON*. Retrieved from Json: json.org

Experis Engineering. (2017, 02 25). *Focus on Engineering.* Retrieved from experisjobs: https://www.experisjobs.us/Website-File-Pile/Whitepapers/Experis/engineering-whitepaper.pdf

Fitzgerald, L. (2019, 09 15). *IT IN DEMAND: SOFTWARE DEVELOPER JOB POSTINGS CONTINUE TO INCREASE.* Retrieved 01 31, 2019, from https://www.aitp.org/blog/aitp-blog/2018/09/25/it-in-demand-software-development-job-postings-continue-to-increase/

Fowler, M. (2011). *Domain Specific Languages.* Boston: Pearson Education, Inc.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2002). *Patterns of enterprise application architecture.* San Francisco: Addison Wesley.

Giridhar, C. (2019). *Learning Python Design Patterns - Second Edition.* Packt Publishing.

Gorn, S. (1940). *Is Automatic Programming Feasible?*

Grammarly. (2017, January 07). *Plural Nouns: Rules and Examples*. Retrieved from Grammarly blog: https://www.grammarly.com/blog/plural-nouns/

Grant, T. (2016, November 30). *Why are terminal consoles still used?* Retrieved from StackExchange: https://ux.stackexchange.com/questions/101990/why-are-terminal-consoles-still-used

Harrison, N. (2017). *Code Generation with Roslyn.* Lexington: Apress.

Hu, W. (2012, September 11). *A N-Tier Architecture Sample with ASP.NET MVC3, WCF, and Entity Framework*. Retrieved from Code Project: https://www.codeproject.com/Articles/434282/A-N-Tier-Architecture-Sample-with-ASP-NET-MVC3-WCF

Jones, C. (2017). Software methodologies : a quantitative guide. Boca Raton: CRC Press.

Konicek, M. (2018, October 11). *Why demand of software engineers is going to stay high*. Retrieved September 20, 2019, from Medium: https://medium.com/swlh/why-demand-for-software-engineers-is-going-to-stay-high-5acb789c5015

Kowalski, R. (2017). *The CLI Book: Writing successful command line interfaces with Node.js.* Apress.

Lamelas, A. (2018, 10 11). *Top 5 main Agile methodologies: advantages and disadvantages.* Retrieved Janeiro 9, 2019, from https://www.xpand-it.com/2018/10/11/top-5-agile-methodologies/ Last access 9/1/2019

Levi, S. (2001, February 1). *Graphical user interface.* Retrieved from ENCYCLOPÆDIA BRITANNICA: https://www.britannica.com/technology/graphical-user-interface

Loritsch, B. (2017, August 2). *Software Engineering - Using a GUID as a Primary Key.* Retrieved from StackExchange: https://softwareengineering.stackexchange.com/questions/354977/using-a-guid-as-a-primary-key

Marcus, A. (1995). Principles of Effective Visual Communication for Graphical User Interface Design. In W. Buxton, J. Grudin, & S. Greenberg, *Readings in Human-Computer Interaction Towards the Year 2000* (pp. 425-441). Elsevier Inc.

Market Research Future. (2019, August 19). *Software Engineering Market Research Report - Global Forecast to 2022.* Retrieved Setembro 15, 2019, from https://www.marketresearchfuture.com/reports/software-engineering-market-2180

Mátrai, R. (2010). *User Interfaces.* Vukovar, Croatia: Intech.

Meixner, D. (2017, June 13). *How to get Visual Studio 217 version number and edition.* Retrieved from Microsot Developer: https://blogs.msdn.microsoft.com/dmx/2017/06/13/how-to-get-visual-studio-2017-version-number-and-edition/

Microsoft. (2010, March 31). *Creating a business logic layer - Responding to Validation Errors in the Presentation Tier*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/web-forms/overview/data-access/introduction/creating-a-business-logic-layer-cs#responding-to-validation-errors-in-the-presentation-tier

Microsoft. (2015, July 19). *Introduction to the C# Language and the .NET Framework.* Retrieved from Microsoft Docs: https://docs.microsoft.com/enus/dotnet/csharp/getting-started/introduction-to-the-csharp-languageand-the-net-framework

Microsoft. (2016, April 26). *Attributes (C#)*. Retrieved from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/

Microsoft. (2017, September 18). *Starting to Develop Visual Studio Extensions*. Retrieved from Microsoft - Visual Studio Docs: https://docs.microsoft.com/en-us/visualstudio/extensibility/starting-to-develop-visual-studio-extensions?view=vs-2019

Microsoft. (2017, October 10). *The .NET Compiler Platform SDK.* Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

Microsoft. (2017, September 9). *Understand the .NET Compiler Platform SDK model*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model

Microsoft. (2018, 01 08). *About .NET Core.* Retrieved October 19, 2019, from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/core/about

Microsoft. (2019, July 1). *Creating a simple data-driven CRUD microservice*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/data-driven-crud-microservice

Microsoft. (2019, August 19). *Extend Visual Studio IDE*. Retrieved from Microsoft - Visual Studio: https://visualstudio.microsoft.com/vs/features/extend/

Microsoft. (2019, April 18). *How to: Determine which .NET Framework versions are installed.* Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/how-to-determine-which-versions-are-installed

Milicev, D. (2009). *Model-Driven Development with Executable UML.* Indianapolis: Wiley Publishing.

MIT. (2019, May 31). *MIT App Inventor*. Retrieved July 5, 2019, from https://appinventor.mit.edu/explore/

Nielsen, J., & Gentner, D. (1996, August). The Anti-Mac interface. *Communications of the ACM*, pp. 70-82.

Novak Jr., G. (2006, January). *CS 394P: Automatic Programming*. Retrieved January 9, 2019, from https://www.cs.utexas.edu/users/novak/cs394p.html

OMG. (2011, January). *Business process model and notation*. Retrieved January 09, 2019, from https://www.omg.org/spec/BPMN/2.0/PDF

Oracle. (2001, February 01). *Java Programming Environment and the Java Runtime Environment (JRE).* Retrieved from Oracle Documentation: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html

Oracle. (2006, June 21). *The web Tier*. Retrieved from Oracle Docs: https://docs.oracle.com/cd/E19226-01/820-7759/gcrnl/index.html

Oracle. (2010, December 2). *The Business Tier*. Retrieved from Oracle Docs: https://docs.oracle.com/cd/E19226-01/820-7759/gcrls/index.html

Oracle. (2014, March 19). *Java Programming Language.* Retrieved from Oracle Java Documentation: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html

OutSystems. (2018, February). *View generated code*. Retrieved January 9, 2019, from https://www.outsystems.com/forums/discussion/13350/view-generated-code/

Outsystems. (2019, March 06). *The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q1 2019.* Retrieved from Outsystems: https://www.outsystems.com/1/low-code-development-platforms-wave/

Parnas, D. (1985). SOFTWARE ASPECTS OF STRATEGIC DEFENSE SYSTEMS. *Association for Computing Machinery*.

Pivotal. (2010, May 01). *Spring Framework*. Retrieved from Spring: https://spring.io/projects/spring-framework

Punter, T., Voeten, J., & Huang, J. (2009). Integrating Quality Assurance. In *Model-Driven Software Development* (pp. 40-43). Hershey, PA: Information Science Reference.

Python Software Foundation. (2016, December 05). *Templating in Python.* Retrieved from Python Wiki: https://wiki.python.org/moin/Templating

Redish, K. (1959). Some Problems of a universal autocode. In *Annual Review in Automatic Programming* (pp. 16-22). Pergamon Press.

Richards, M. (2015). *Software Architecture Patterns.* California: O'Reilly.

Richardson, C., & Rymer, J. R. (2014). *New Development Platforms Emerge For Customer-Facing Applications.* Forrester.

Solomon, D. (1993). *Assemblers and Loaders.* West Sussex: Ellis Horwood Ltd.

Stack Overflow. (2019, April 9). *Developer Survey Results 2019.* Retrieved from StackOverflow Insights: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

Stalla, A. (2019, June 13). *Hibernate Validator Specific Constraints*. Retrieved from Baeldung: https://www.baeldung.com/hibernate-validator-constraints

Stephens, R. (2015). Beginning Software Engineering. Indianapolis: John Wiley & Sons, Inc.

TIOBE. (2019, September). *TIOBE Index for September 2019.* Retrieved from TIOBE: https://www.tiobe.com/tiobe-index/

Tomassetti, G. (2018, May 9). *A Guide to Code Generation.* Retrieved February 4, 2019, from https://tomassetti.me/code-generation/

Truyen, F. (2006, January). *The Fast Guide to Model Driven Architecture.* Retrieved January 31, 2019, from https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

Vernon, V. (2013). *Domain-driven Development.* San Francisco: Addison-Wesley.

W3. (2018, May 25). *Web Content Accessibility Guidelines (WCAG) Overview*. Retrieved from W3C: https://www.w3.org/WAI/standards-guidelines/wcag/

Annex 1. Notation element tables

The UML (Unified Model Language) is a modeling language used to produce artifacts that identify structures and algorithms in a way that it is less extent than natural text but easier to read than code. On the other hand, BPMN (Business Process Modeling Notation) is a modeling language used to produce workflow diagrams that supplement the ones produced with UML. The following tables display, respectively the UML and BPMN elements used during development.

Table 5. UML notation symbols (Booch, Rumbaugh, & Jacobson, 2005)

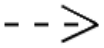
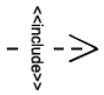
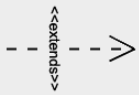
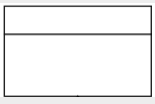
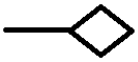
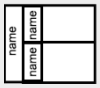
	Name	Description
	System	The container system for the use cases
	Use Case	An activity that the actor can execute in the system
	Actor	Some entity that intervenes with the use cases
	Dependency	A relationship between two model elements
	Package	Groups elements
	Includes	Defines a behavior to a model element
	Extends	Defines a relationship between two elements when one is a variant of another
	Generalization	A relationship where two elements share characteristics
	Note	A simple note element
	Class	An element that represents an entity on the system
	Association	Relationship that represents an element that is composed by other elements
	Aggregation	Relationship that groups elements
	Ternary relationship	Relationship that involves more than one class
	Component	Modular part of a system that encapsulates its contents
	Node	Resource from which the artifacts may be deployed

Table 6. BPMN notation symbols (OMG, 2011)

	Name	Description
	Annotation	Provides additional information
	Data	Represents data objects that can be input or output
	End event	Indicates the process' end
	Event-based Gateway	Can start a new instance of the process with basis on an event and an evaluation
	Exclusive Gateway	Exclusive decision and merging
	Group	Groups several elements
	Inclusive Gateway	Executes multiple paths based on an evaluation
	Intermediate event	Occurs between the start and the end of the process but won't start or end it.
	Lane	Sub partition of a pool within a process, being used to categorize segments.
	Parallel event Gateway	Can start a new instance of the process with basis on a parallel event without evaluation
	Parallel Gateway	Executes multiple paths without evaluation
	Pool	Represents a participant in a process
	Start Event	Dictates when the event starts
	Sub-process Activity	A compound activity that is included in a process, so that it can be broken down to a finer level
	Task Activity	An atomic task
	Transaction Activity	A task that must be considered whole, failing if any of its tasks fail

Annex 2. CodeGen Component Diagram

A component diagram is typically used to visualize the relationship between physical components in the solution. In this case the components are libraries that can be reused in other projects.

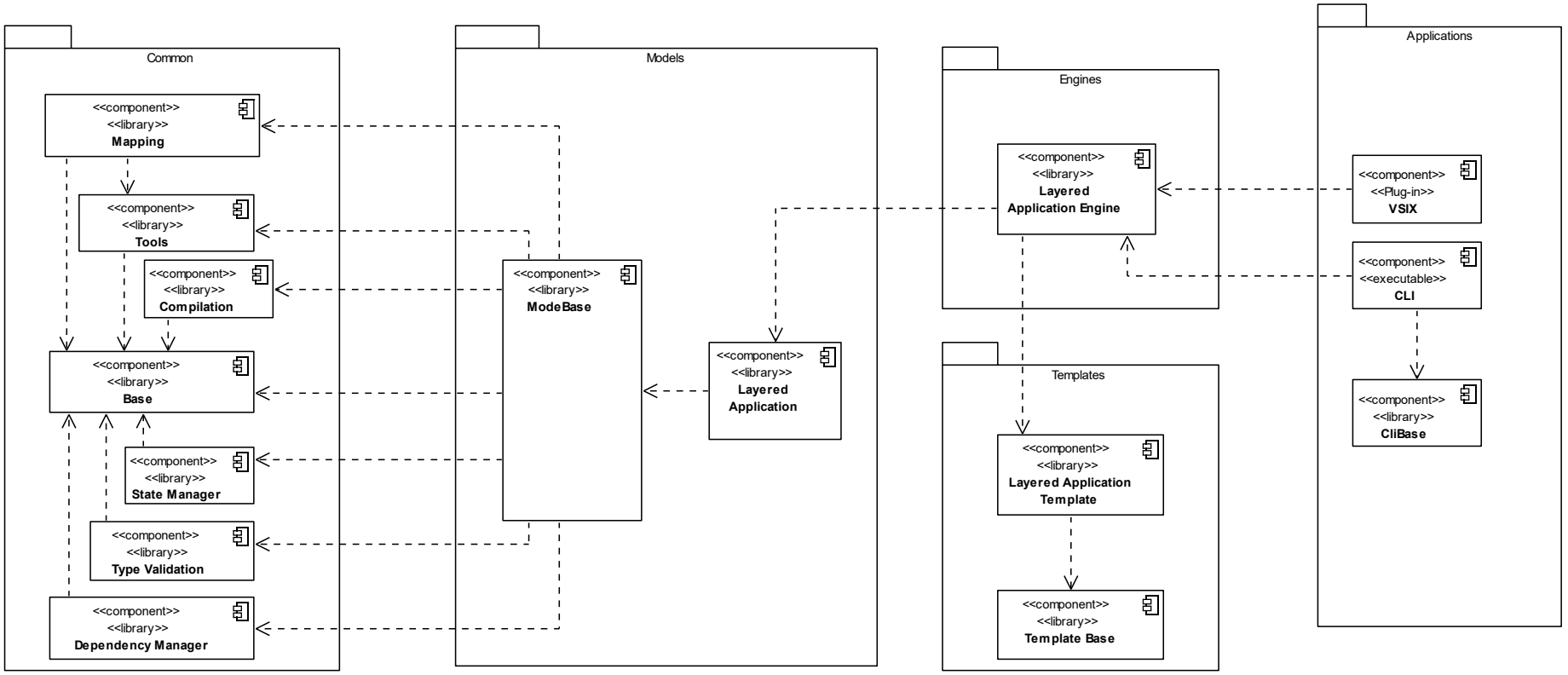


Figure 60. CodeGen's component diagram

Annex 3. Common Libraries' documentation

The common library is fundamental for the maintenance and reusability of the code. As such, it is composed by several small units of code that can be easily described and tested. This annex is composed by class diagrams, code metric results and test lists for these libraries.

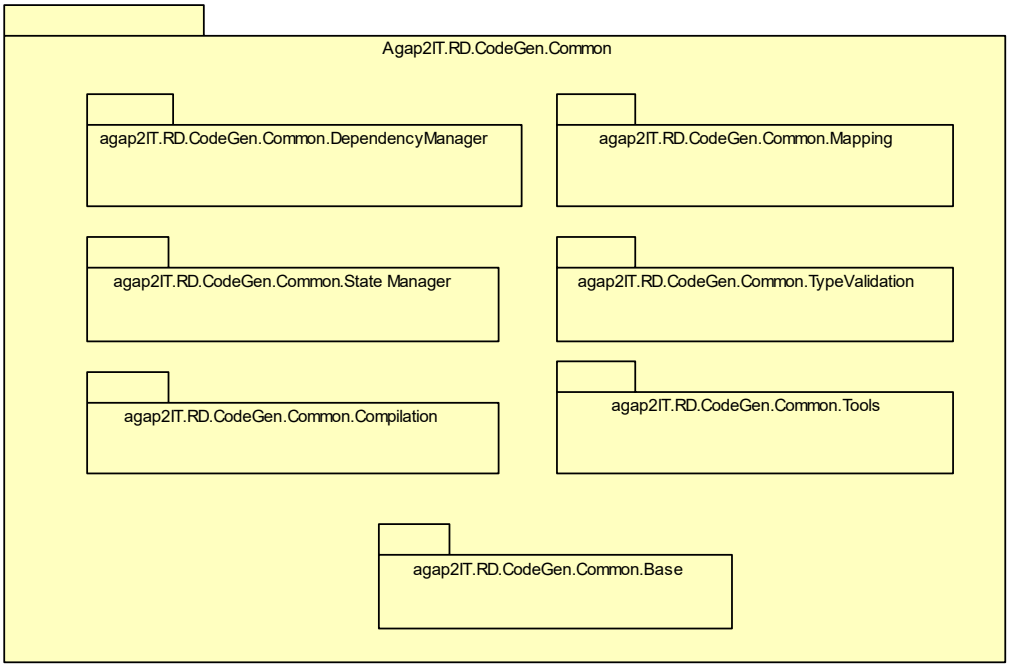


Figure 61. Common package diagram

Base library

Class diagram

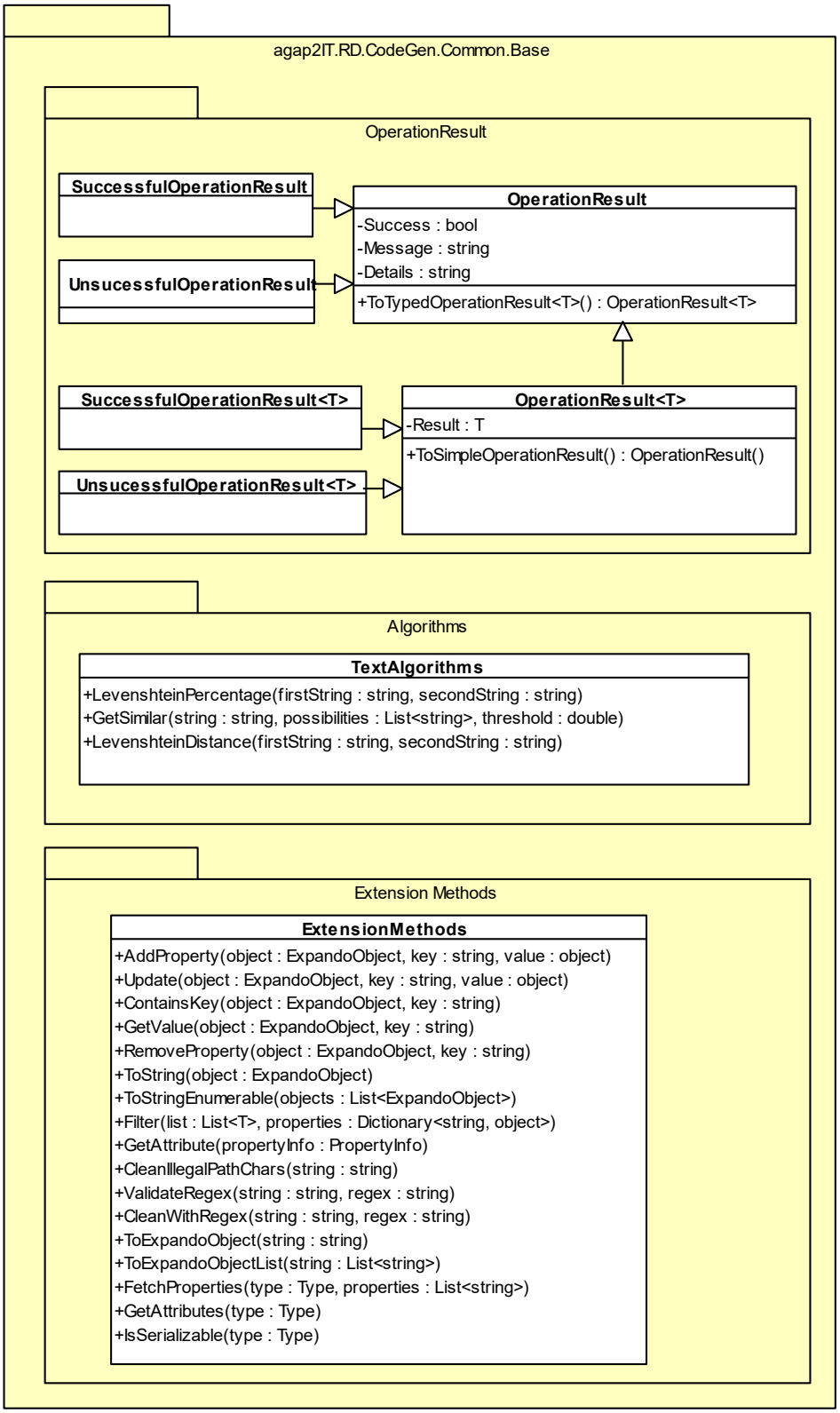


Figure 62. Support library class diagram

Code metric results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Common\Base (Debug)	84	64	3	42	539
Agap2IT.RD.CodeGen.Common.Base.Algorithms	62	13	1	3	61
TextAlgorithms	62	13	1	3	57
Agap2IT.RD.CodeGen.Common.Base.ExtensionMethods	79	31	1	32	273
ExtensionMethods	79	31	1	32	257
Agap2IT.RD.CodeGen.Common.Base.OperationResults	89	20	3	8	205
OperationResult	91	7	1	3	54
OperationResult<T>	83	5	2	3	44
SuccessfulOperationResult	94	2	2	1	19
SuccessfulOperationResult<T>	86	2	3	1	28
UnsuccessfulOperationResult	94	2	2	1	23
UnsuccessfulOperationResult<T>	86	2	3	1	25

Figure 63. Code metric results for the base library

Test list

Table 7. List of unit tests performed to the support library

Code	Description	Result
TST001	Adds a property to an expando object and checks if it exists	**Success**
TST002	Updates a property on an expando object	**Success**
TST003	Removes a property from an expando object	**Success**
TST004	Converts an expando object to a string and vice-versa	**Success**
TST005	Checks if the object has a propert	**Success**
TST006	Serializes an object	**Success**
TST007	Checks if the object is primitive	**Success**

Tools Library

Class diagram

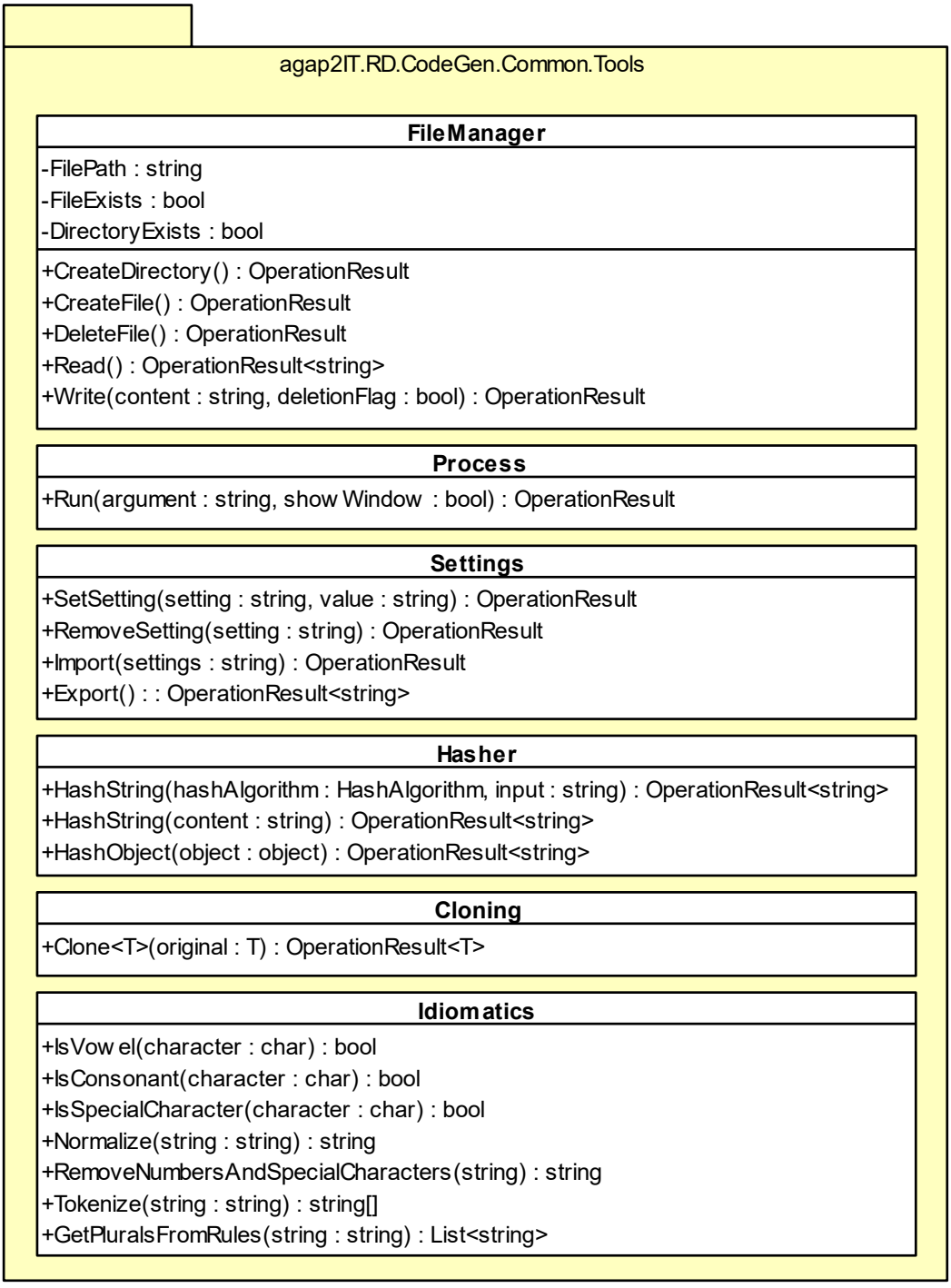


Figure 64. Tools Library class diagram

Code metric results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Common\Tools (Debug)	74	91	2	60	613
Agap2IT.RD.CodeGen.Common.Tools.Properties	88	23	1	10	225
Resources	88	23	1	10	212
Agap2IT.RD.CodeGen.Common.Tools.Attributes	100	1	2	3	11
IgnoreInDigestAttribute	100	1	2	3	7
Agap2IT.RD.CodeGen.Common.Tools	67	67	1	48	377
Process	61	3	1	6	28
Idiomatics	73	26	1	8	82
Hasher	67	8	1	21	66
FileInputOutput	75	26	1	16	147
Cloning	60	4	1	10	34

Figure 65. Code metric results for the tools library

Test list

Table 8. List of unit tests performed to the tool library

Code	Feature	Description	Result
TST008	Cloning	Deep clones a serializable object	Success
TST009	Cloning	Deep clones a non-serializable object	Success
TST010	FileManager	Creates, writes, reads and deletes a file	Success
TST011	Hashing	Creates a hash and compares it to the expected value	Success

Business Process Diagrams

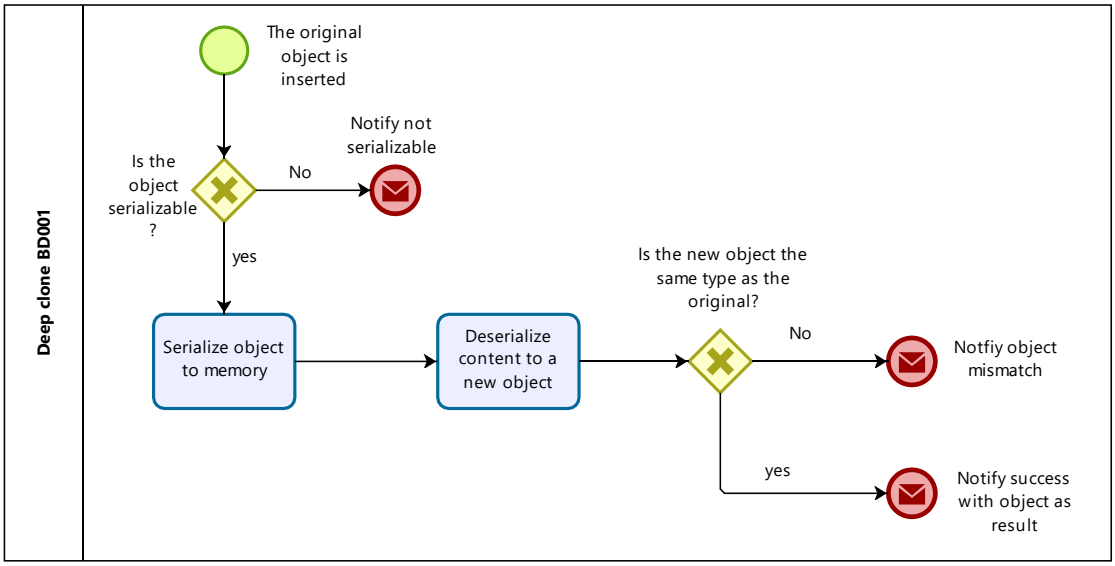


Figure 66. Tools library - Deep clone business process diagram

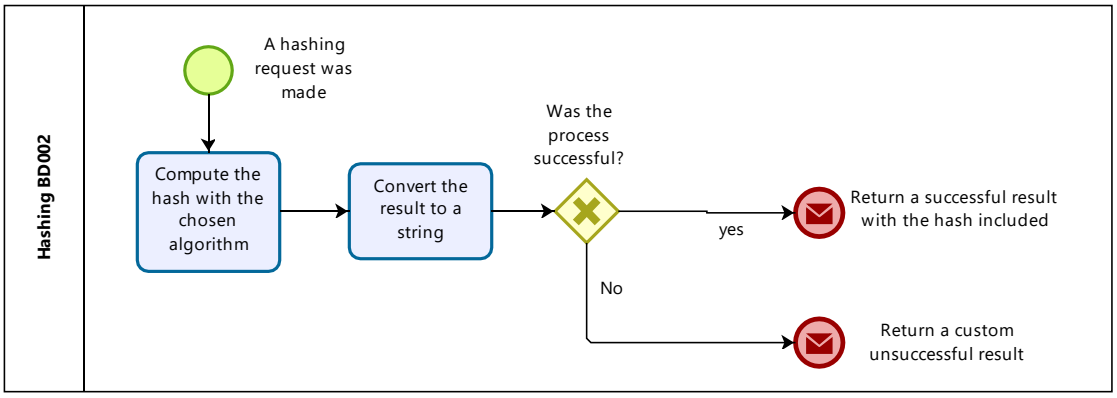


Figure 67. Tools library - String hashing business process diagram

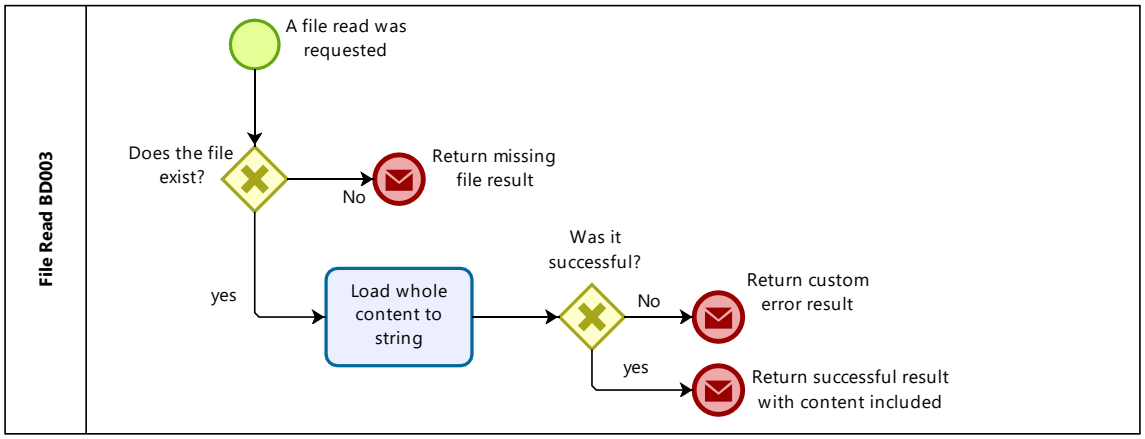


Figure 68. Tools library - File read business process diagram

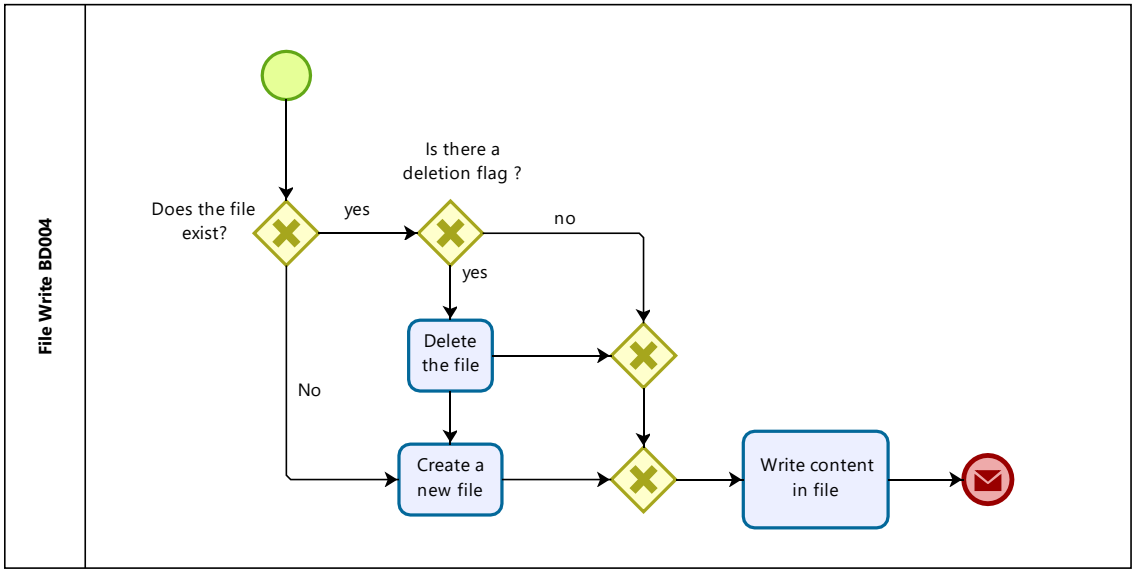


Figure 69. Tools library - File write business process diagram

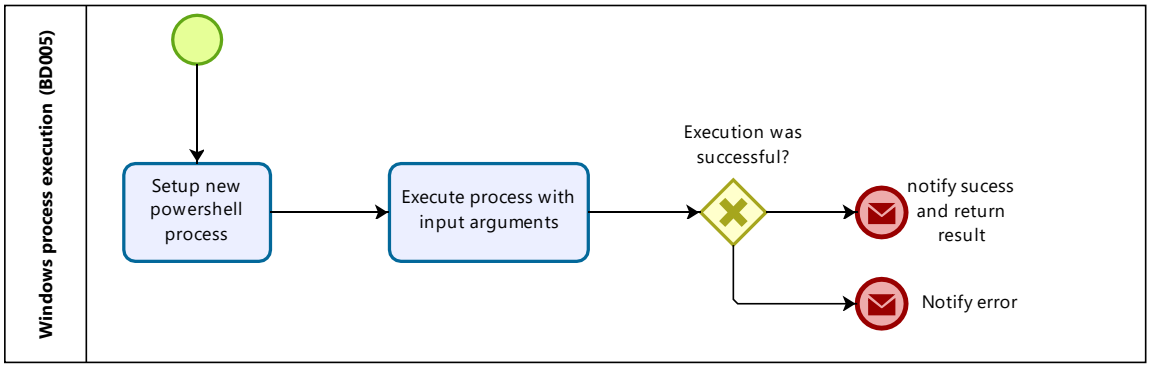


Figure 70. Tools library - Process execution business process diagram

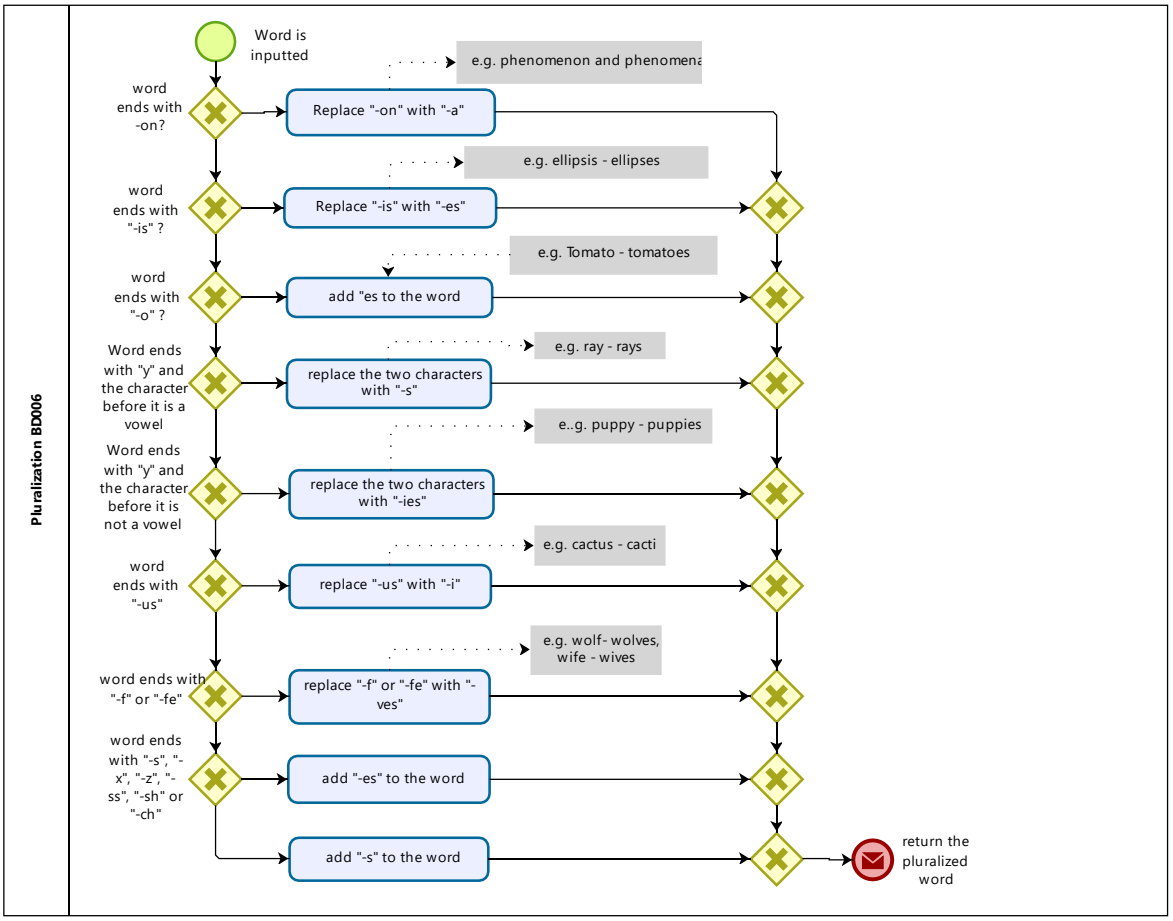


Figure 71. Tools library - String pluralization business process diagram

Dependency Manager Library

Class diagram

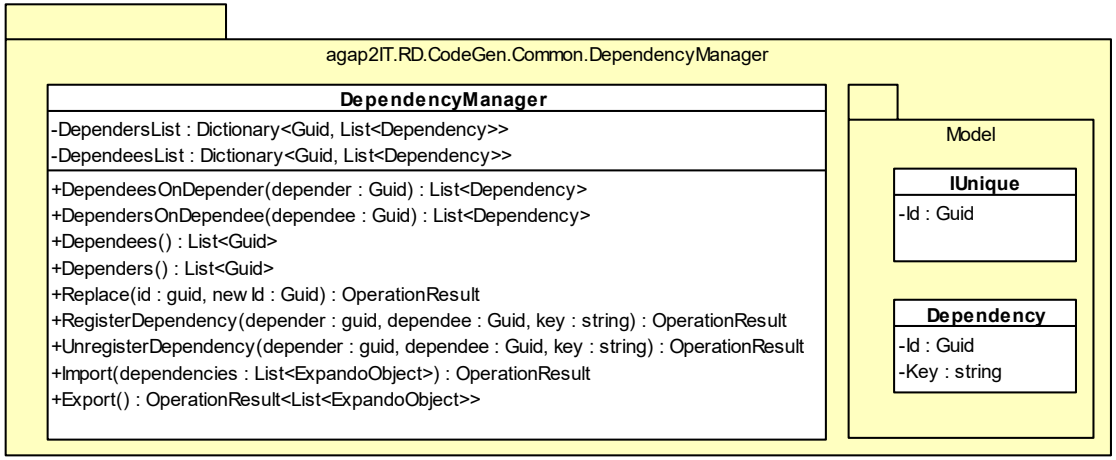


Figure 72. Dependency manager class diagram

Code metric results

Common\DependencyManager (Debug)		87	64	1	37	333
Agap2IT.RD.CodeGen.Common.DependencyManager.Pr		88	16	1	10	162
Resources		88	16	1	10	149
Agap2IT.RD.CodeGen.Common.DependencyManager.M		97	4	1	2	43
IUnique		100	1	0	1	10
Dependency		95	3	1	2	25
Agap2IT.RD.CodeGen.Common.DependencyManager		68	44	1	27	128
DependencyManager		68	44	1	27	125

Figure 73. Dependency manager code metrics

Test list

Table 9. List of unit tests performed to the tool library

Code	Feature	Result
TST012	Register a dependency	Success
TST013	Register an existing dependency	Failure
TST014	Register a reverse dependency	Failure
TST015	Register self dependency	Failure
TST016	Unregister a dependency	Success
TST017	Unregister a non-existing dependency	Failure
TST018	Unregister a reverse dependency	Success
TST019	List dependencies of a depender	Success
TST020	List dependencies of a dependee	Success
TST021	List all dependees and dependers	Success
TST022	Import dependencies	Success
TST023	Export dependencies	Success

Business Process Diagrams

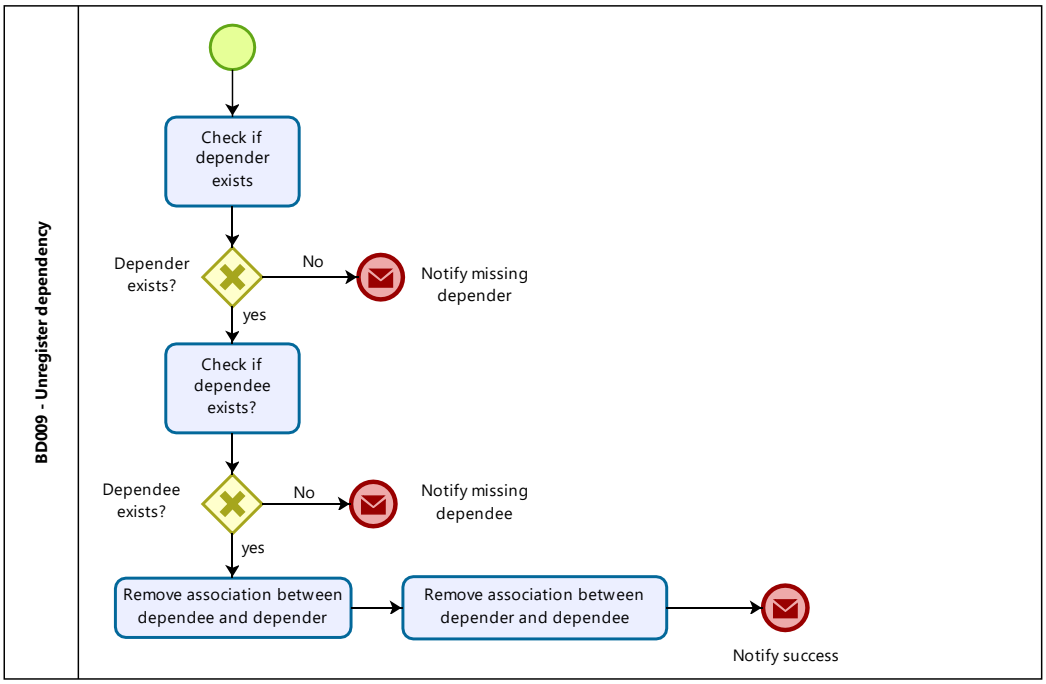


Figure 74. Dependency Manager library - Dependency removal

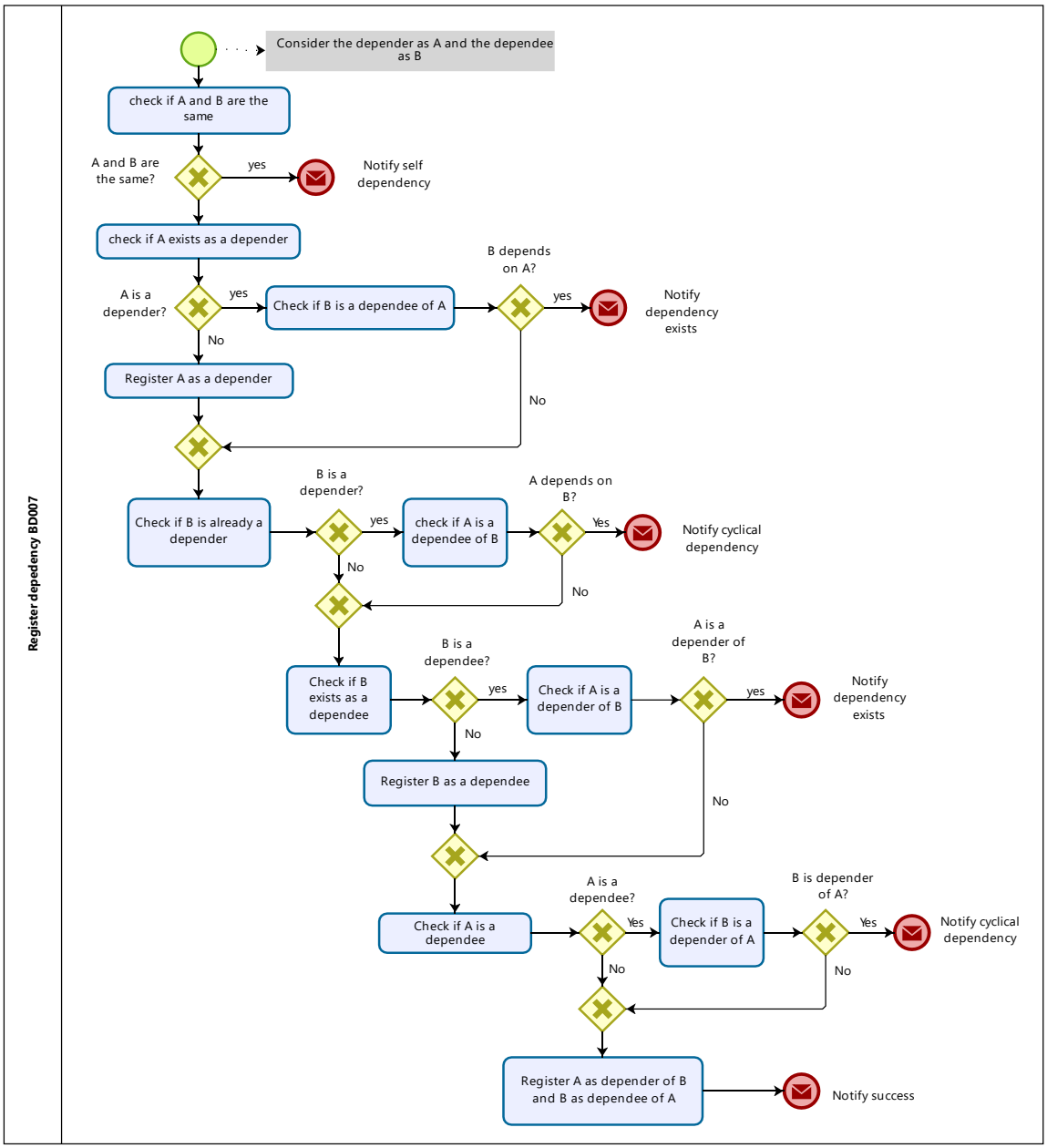


Figure 75. Dependency Manager library - Dependency registration

State Manager Library

Class diagram

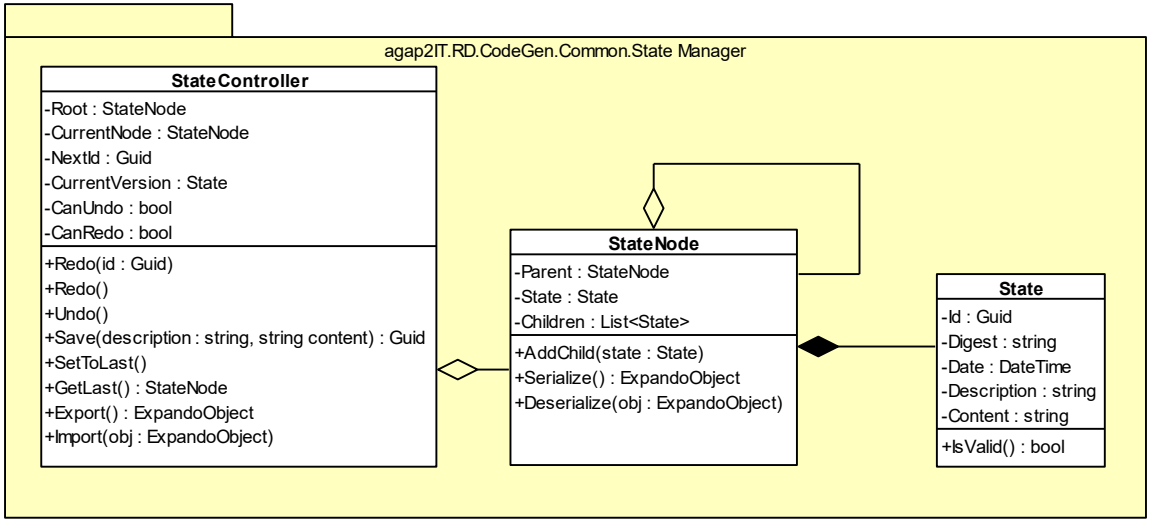


Figure 76. State Controller class diagram

Code metrics

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Common\StateManager (Debug)	86	55	1	24	198
StateManager	86	55	1	24	198
State	90	16	1	8	39
StateController	88	26	1	10	94
StateNode	81	13	1	16	53

Figure 77. State Controller code metrics

Mapping library

Class diagram

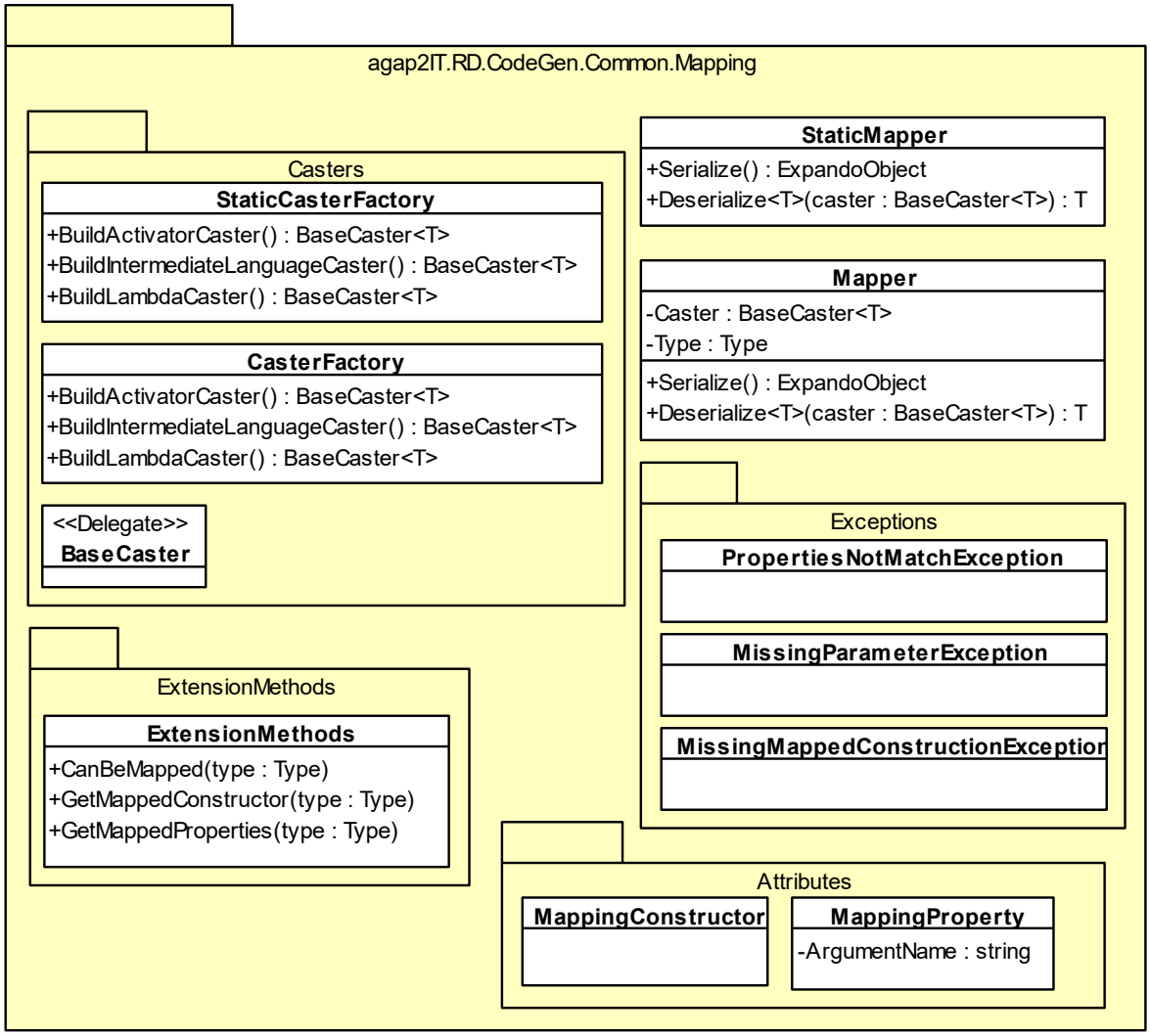


Figure 78. Mapping library class diagram

Code Metric results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Common\Mapper (Debug)	85	99	2	82	831
Agap2IT.RD.CodeGen.Common.Mapper.Properties	88	11	1	10	117
Resources	88	11	1	10	104
Agap2IT.RD.CodeGen.Common.Mapper.ExtensionMethods	81	15	1	15	114
ExtensionMethods	81	15	1	15	102
Agap2IT.RD.CodeGen.Common.Mapper.Exceptions	88	3	2	4	49
MissingParameterException	88	1	2	4	13
MissingMappedConstructorException	89	1	2	4	14
MappedConstructorDoesNotMatchMappedPropertiesException	89	1	2	4	11
Agap2IT.RD.CodeGen.Common.Mapper.Casters	84	16	1	20	181
StaticCasterFactory<T>	62	5	1	19	69
CasterFactory<T>	75	10	1	19	88
BaseCaster.Caster<T1>	100	1	1	0	7
BaseCaster	100	0	1	0	13
Agap2IT.RD.CodeGen.Common.Mapper.Attributes	100	2	2	3	24
MappingPropertyAttribute	100	1	2	3	11
MappingConstructorAttribute	100	1	2	3	5
Agap2IT.RD.CodeGen.Common.Mapper	68	52	1	45	346
StaticMapper	66	18	1	28	98
Mapper<T>	71	34	1	39	240

Figure 79. Map library code metric results

Test List

Table 10.List of unit tests performed 10^5 times to check the performance of assisted mapping with Microsoft's Activator

Code	Description	Avg. Result (s)
TST024	Dynamically deserialize five strings with dynamic activator	
TST025	Dynamically deserialize five strings with static activator	
TST026	Statically deserialize five strings with dynamic activator	
TST027	Statically deserialize five strings with static activator	
TST028	Dynamically deserialize ten strings with dynamic activator	
TST029	Dynamically deserialize ten strings with static activator	
TST030	Statically deserialize ten strings with dynamic activator	
TST031	Statically deserialize ten strings with static activator	
TST032	Dynamically deserialize an object with dynamic activator	
TST033	Dynamically deserialize an object with static activator	
TST034	Statically deserialize an object with dynamic activator	
TST035	Statically deserialize an object with static activator	

Table 11. List of unit tests performed 105 times to check the performance of assisted mapping with a lambda expression

Code	Description	Avg. Result (s)
TST036	Dynamically deserialize five strings with dynamic a lambda expression	
TST037	Dynamically deserialize five strings with static a lambda expression	
TST038	Statically deserialize five strings with dynamic a lambda expression	
TST039	Statically deserialize five strings with static a lambda expression	
TST040	Dynamically deserialize ten strings with dynamic a lambda expression	
TST041	Dynamically deserialize ten strings with static a lambda expression	
TST042	Statically deserialize ten strings with dynamic a lambda expression	
TST043	Statically deserialize ten strings with static a lambda expression	
TST044	Dynamically deserialize an object with dynamic a lambda expression	
TST045	Dynamically deserialize an object with static a lambda expression	
TST046	Statically deserialize an object with dynamic a lambda expression	
TST047	Statically deserialize an object with static a lambda expression	

Table 12. List of unit tests performed 105 times to check the performance of assisted mapping with intermediate language

Code	Description	Avg. Result (s)
TST048	Dynamically deserialize five strings with dynamic intermediate language	
TST049	Dynamically deserialize five strings with static intermediate language	
TST050	Statically deserialize five strings with dynamic intermediate language	
TST051	Statically deserialize five strings with static intermediate language	
TST052	Dynamically deserialize ten strings with dynamic intermediate language	
TST053	Dynamically deserialize ten strings with static intermediate language	
TST054	Statically deserialize ten strings with dynamic intermediate language	
TST055	Statically deserialize ten strings with static intermediate language	
TST056	Dynamically deserialize an object with dynamic intermediate language	
TST057	Dynamically deserialize an object with static intermediate language	
TST058	Statically deserialize an object with dynamic intermediate language	
TST059	Statically deserialize an object with static intermediate language	

Business process diagrams

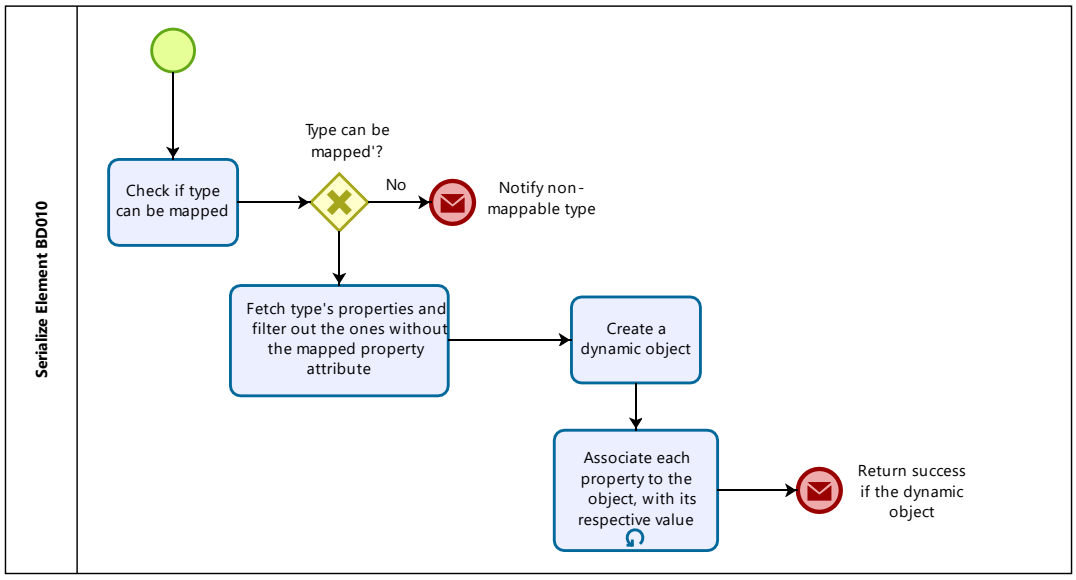


Figure 80. Element serialization business process diagram

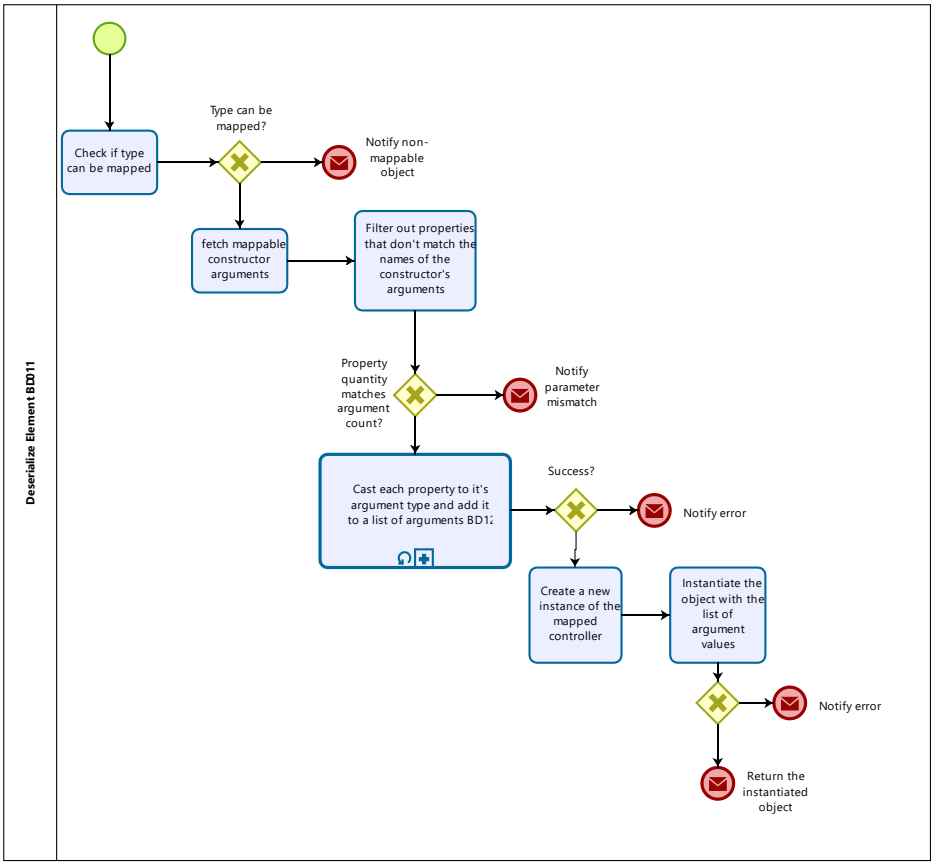


Figure 81. Element deserialization business process diagram

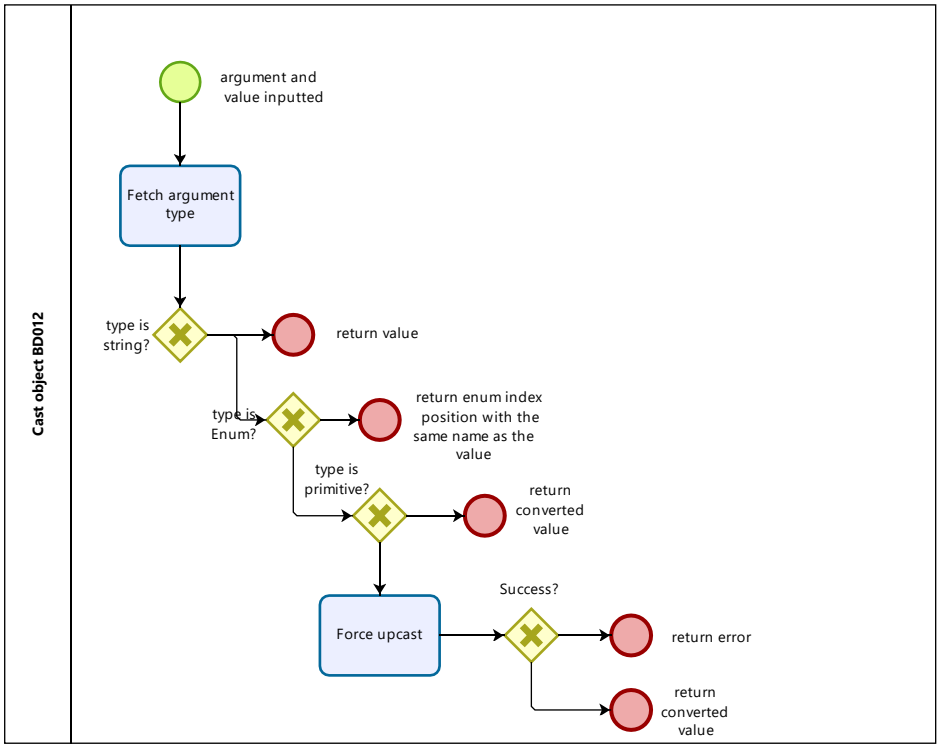


Figure 82. Unknown value casting business process diagram

Type validation library

Test List

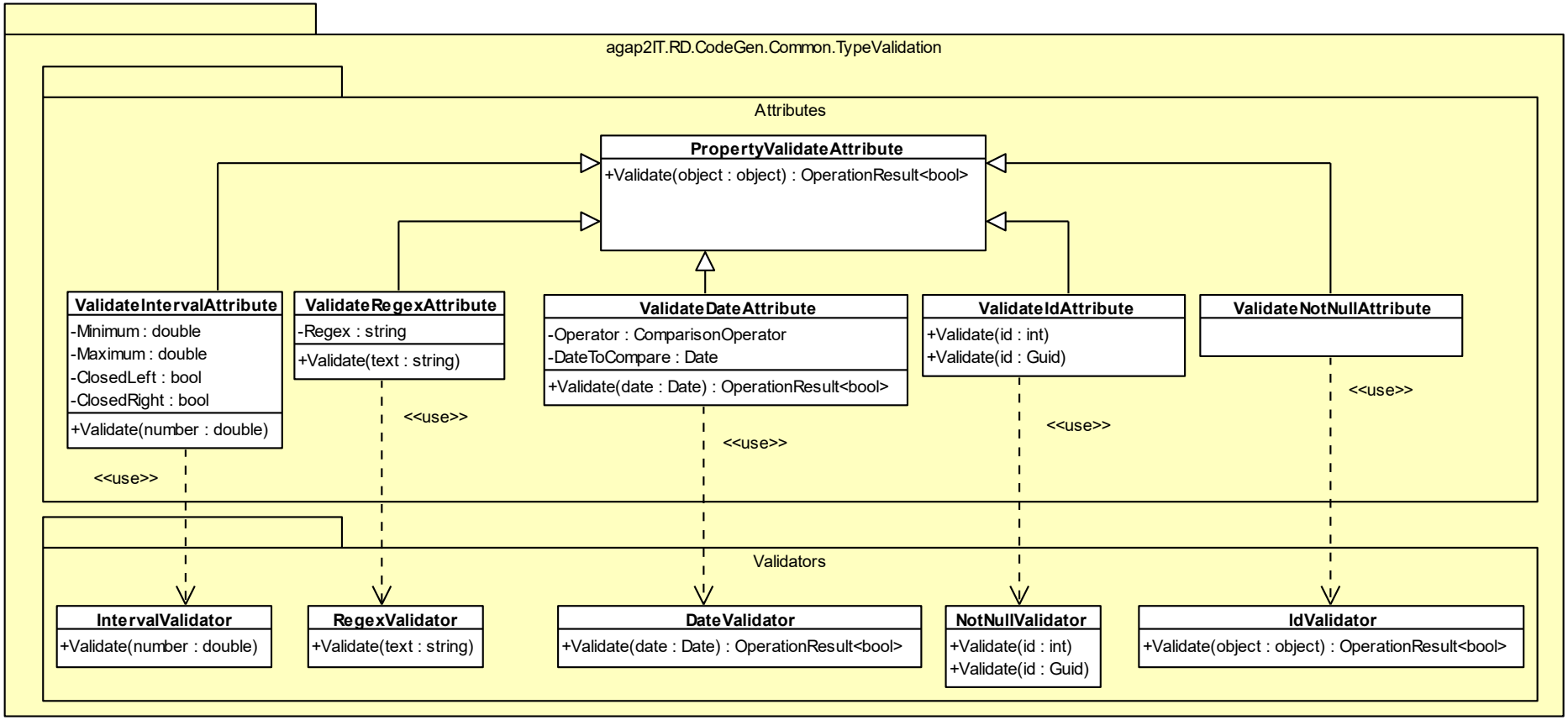


Figure 83. Type validation class diagram

Code metrics results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Common\TypeValidation (Debug)	85	70	3	42	520
Agap2IT.RD.CodeGen.Common.TypeValidation.Attributes	88	29	3	23	178
PropertyValidationAttribute	63	5	2	13	16
ValidateDateAttribute	89	7	3	9	28
ValidateIdAttribute	94	2	3	5	25
ValidateIntervalAttribute	94	9	3	3	35
ValidateNotNullAttribute	94	3	3	4	30
ValidateRegexAttribute	95	3	3	4	20
Agap2IT.RD.CodeGen.Common.TypeValidation.Enumerators	100	1	1	0	16
ComparisonOperatorEnum	100	1	1	0	13
Agap2IT.RD.CodeGen.Common.TypeValidation.Properties	88	17	1	10	171
Resources	88	17	1	10	158
Agap2IT.RD.CodeGen.Common.TypeValidation.Validators	79	23	1	18	155
DateValidator	75	9	1	5	36
IdValidator	90	2	1	5	26
IntervalValidator	69	5	1	3	23
NotNullValidator	74	6	1	12	34
RegexValidator	91	1	1	4	16

Figure 84. Code metrics for the Type Validation library

Business Process diagrams

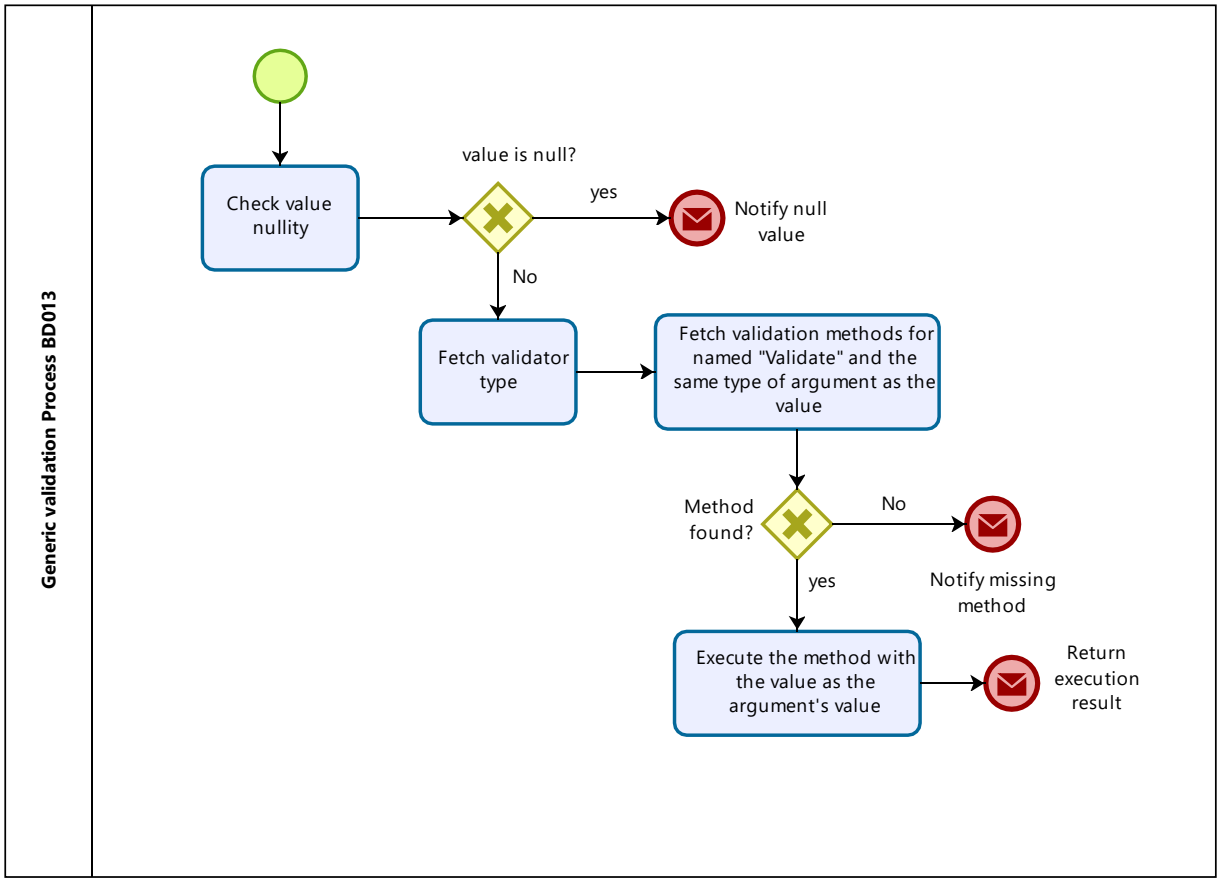


Figure 85. Validation process for any validation property

Compilation library

Class diagram

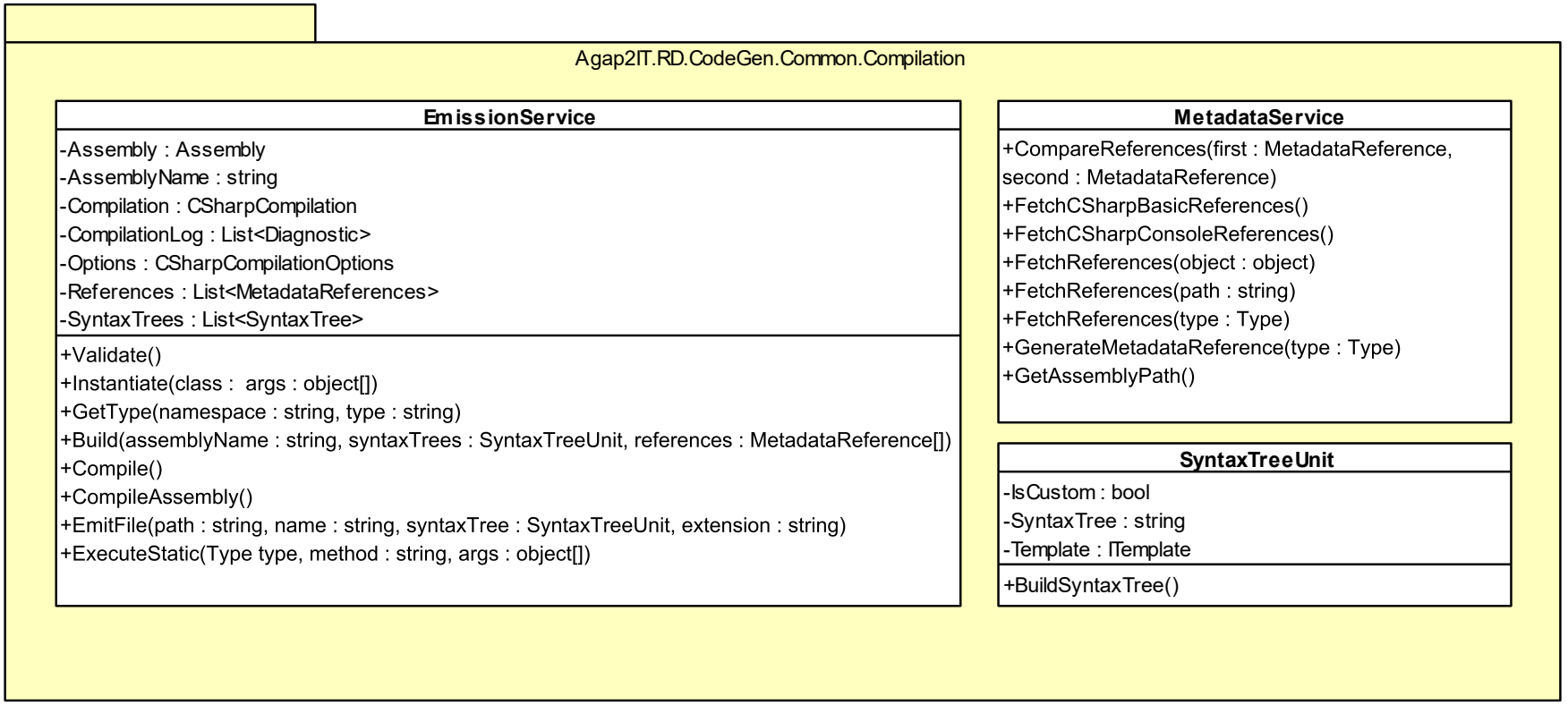


Figure 86. Compilation Class Diagram

Code metrics results

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability I...	Cyclomatic Comp...	Depth of Inherit...	Class Cou...	Lines of Code
Common\Compilation (Debug)	86	81	1	81	526
Agap2IT.RD.CodeGen.Common.Compilation	82	65	1	73	366
EmissionService	78	45	1	67	225
MetadataService	88	9	1	8	51
SyntaxTreeUnit	81	11	1	7	78
Agap2IT.RD.CodeGen.Common.Compilation.Interfaces	100	1	0	0	7
ITemplate	100	1	0	0	4
Agap2IT.RD.CodeGen.Common.Compilation.Properties	88	15	1	10	153
Resources	88	15	1	10	140

Figure 87. Code metrics for the compilation library

Business Process diagrams

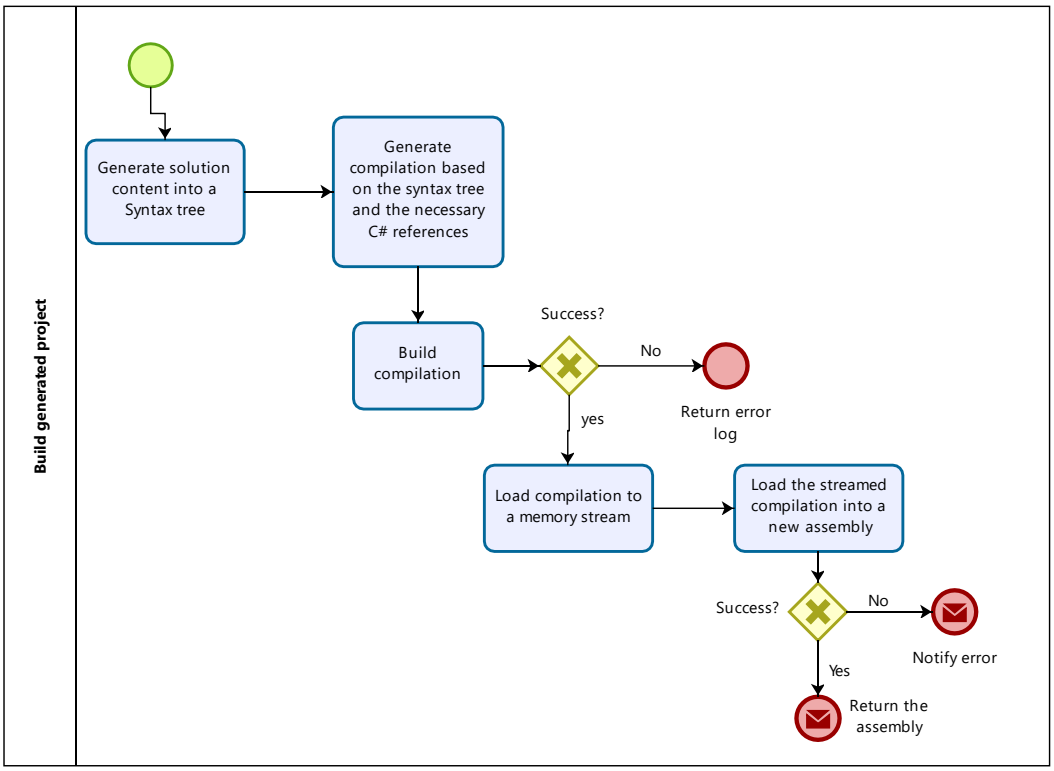


Figure 88. Runtime build workflow

Annex 4. Model Libraries' documentation

As presented in the third section of the fourth chapter, the application model is the representation used by the code generator to produce the result. This model is filled by interactions between the user and an interface. The current document presents class additional information about the model elements available, such as class diagrams, significant business process diagrams and the code metric result for each library presented in the following package diagram (figure 89)

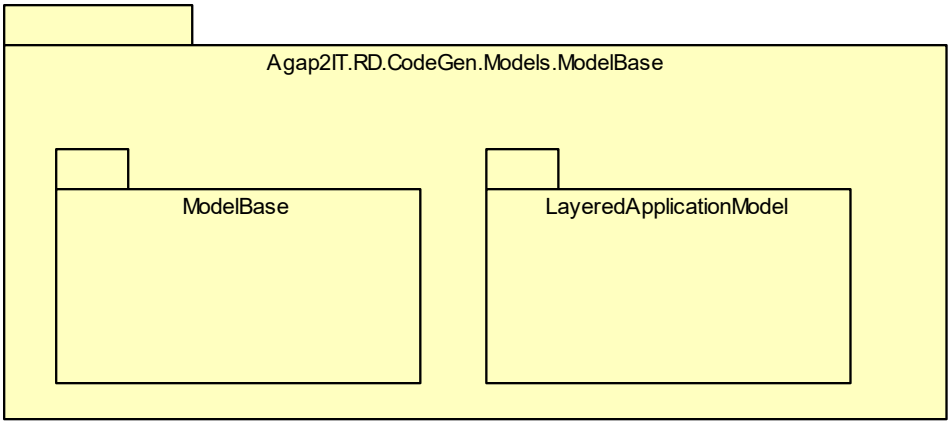


Figure 89. Models package diagram

Model Base library

Class diagram

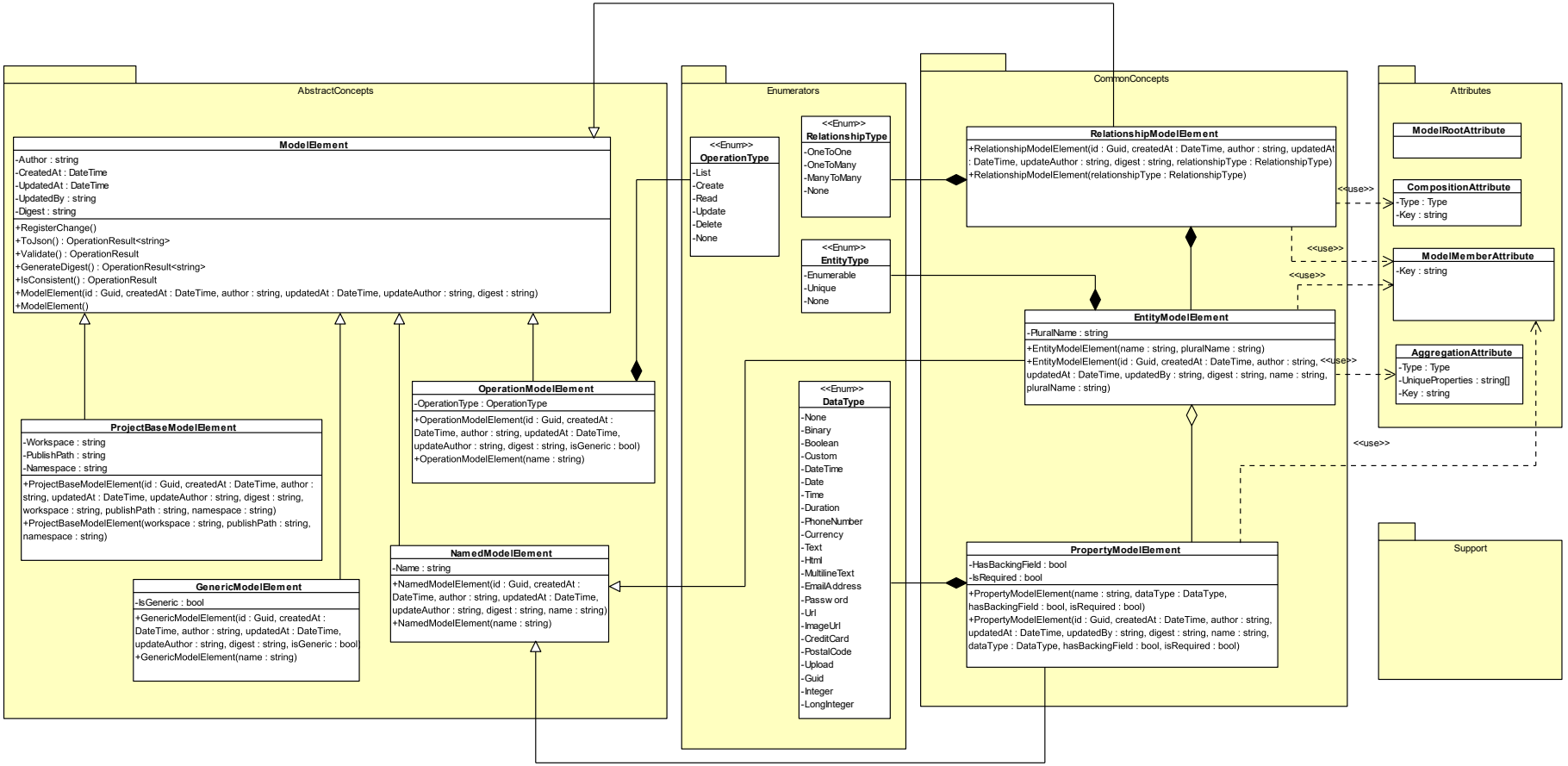


Figure 90. Class diagram for the model base library (1/2)

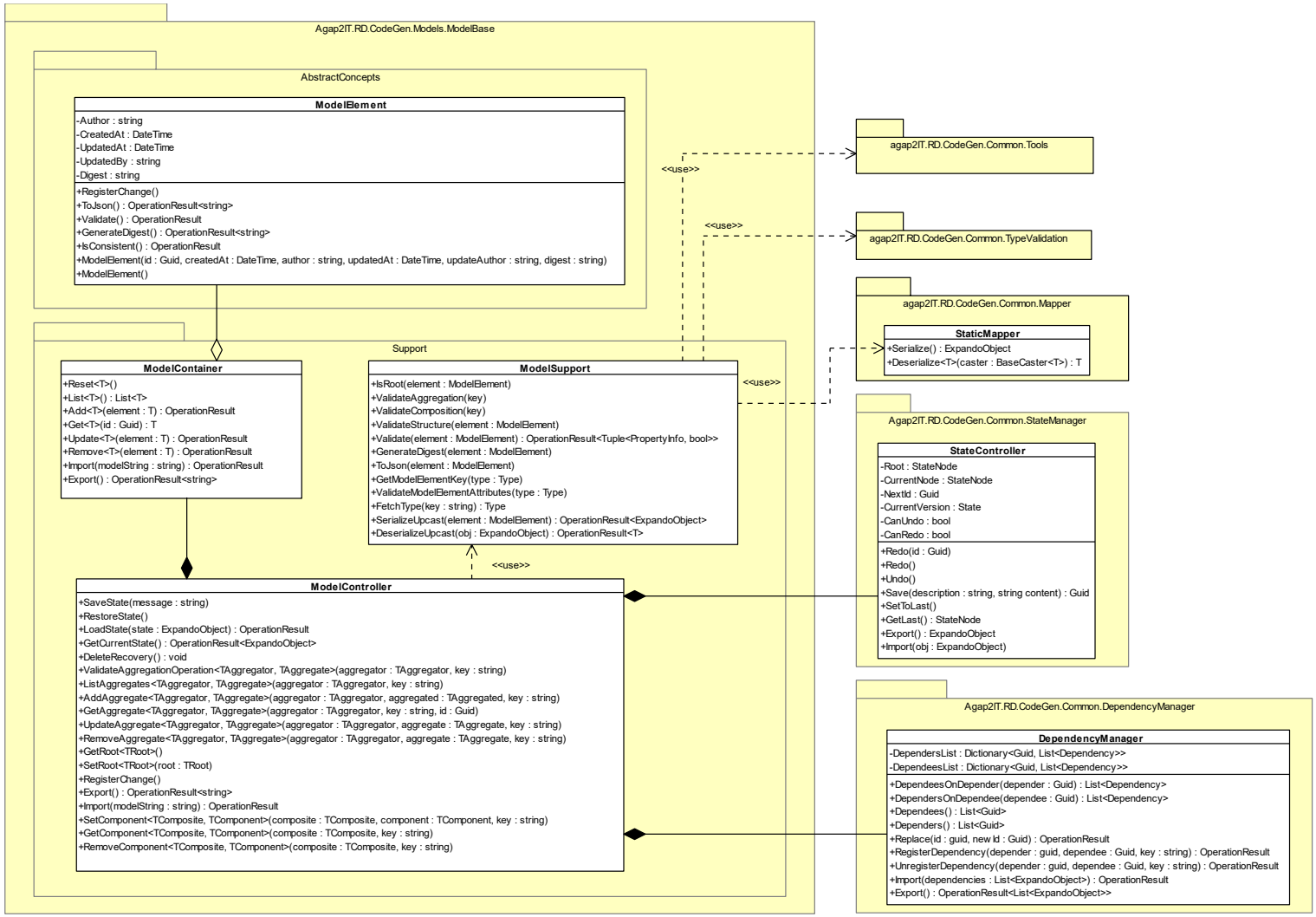


Figure 91. Class diagram for the model base library (2/2)

Code metric results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Models\ModelBase (Debug)	85	276	3	142	2,014
Agap2IT.RD.CodeGen.Models.ModelBase.Support	65	158	1	109	717
ModelSupport	65	47	1	55	142
ModelController	66	67	1	42	367
ModelContainer	66	44	1	38	196
Agap2IT.RD.CodeGen.Models.ModelBase.Properties	88	34	1	10	324
Resources	88	34	1	10	311
Agap2IT.RD.CodeGen.Models.ModelBase.Enumerators	100	4	1	0	61
RelationshipType	100	1	1	0	7
OperationType	100	1	1	0	9
EntityType	100	1	1	0	6
DataType	100	1	1	0	27
Agap2IT.RD.CodeGen.Models.ModelBase.CommonConcepts	80	21	3	19	285
RelationshipModelElement	87	6	2	12	102
PropertyModelElement	76	8	3	8	113
EntityModelElement	79	7	3	13	58
Agap2IT.RD.CodeGen.Models.ModelBase.Builders	64	4	1	9	24
RelationshipFactory<T>	64	4	1	9	21
Agap2IT.RD.CodeGen.Models.ModelBase.Attributes	97	13	2	4	62
RootAttribute	100	1	2	1	3
ModelMemberAttribute	100	2	2	3	12
CompositionAttribute	93	4	2	4	16
AggregationAttribute	95	6	2	4	15
Agap2IT.RD.CodeGen.Models.ModelBase.AbstractConcepts	84	42	2	19	541
ProjectBaseModelElement	76	8	2	7	116
OperationModelElement	89	4	2	7	69
NamedModelElement	88	4	2	7	70
ModelElement	80	22	1	16	195
GenericModelElement	89	4	2	6	69

Figure 92. Code metrics for the model base library

Model Container test list

Table 13. Test list for the model container

Code	Description	Result
TST048	Add an entity to the container	Success
TST049	Add a property to the container	Success
TST050	Add a repeated entity to the container	Failure
TST051	Add a repeated property to the container	Failure
TST052	List elements of a type	Success
TST053	List elements of a different type	Success
TST054	Fetch an existing element	Success
TST055	Fetch a non-existing element	Failure
TST056	Update an element	Success
TST057	Update a non-existing element	Failure
TST058	Remove an element	Success
TST059	Remove a non-existing element	Failure
TST060	Change a value on an element without updating the list	Failure

Model Container business process diagrams

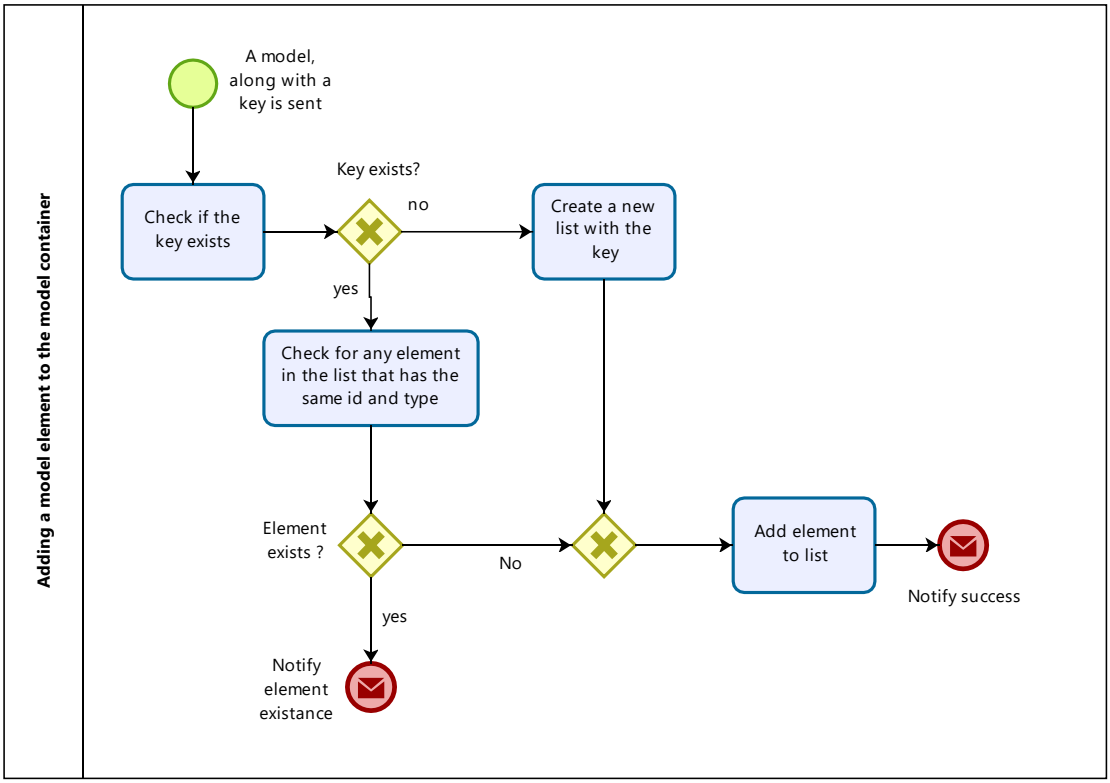


Figure 93 Adding an element to the model container

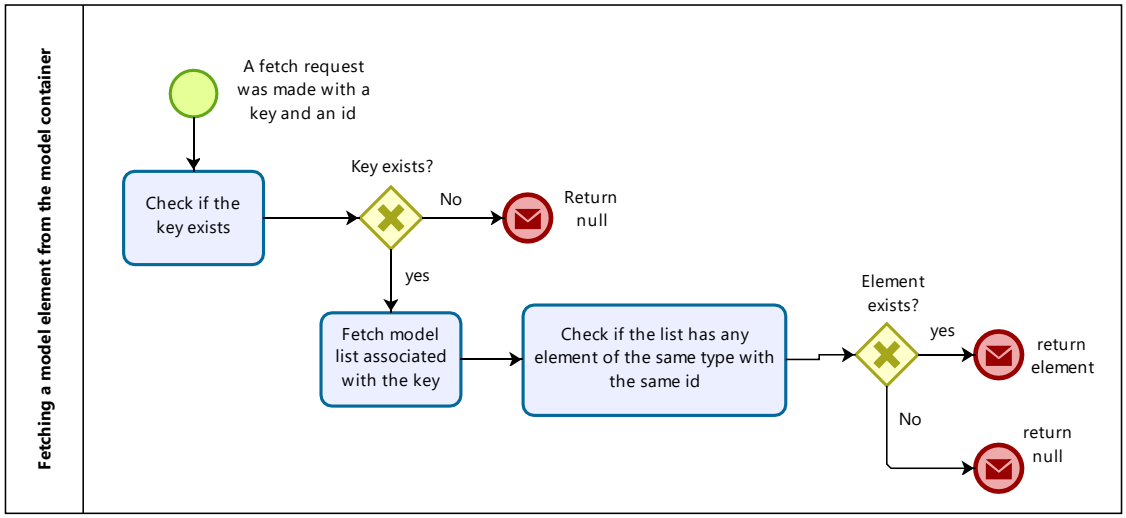


Figure 94. Fetching an element from the container

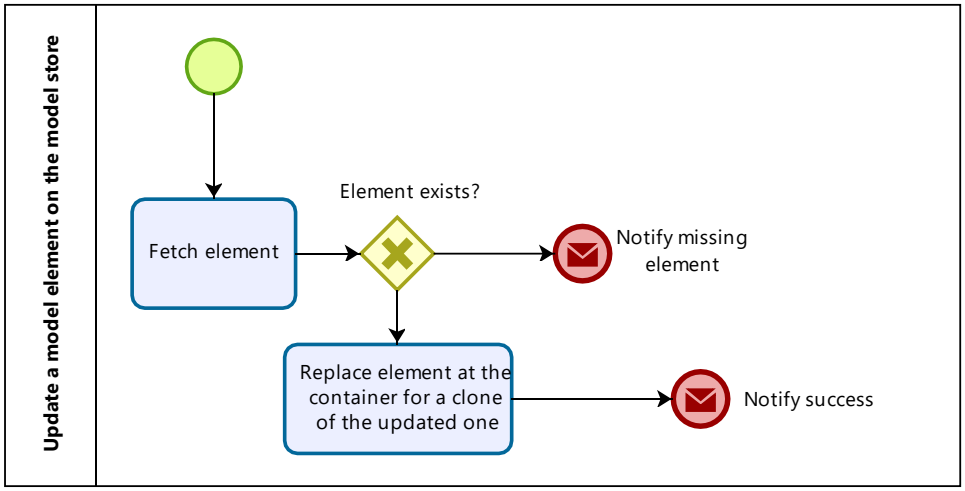


Figure 95. Updating an element on the container

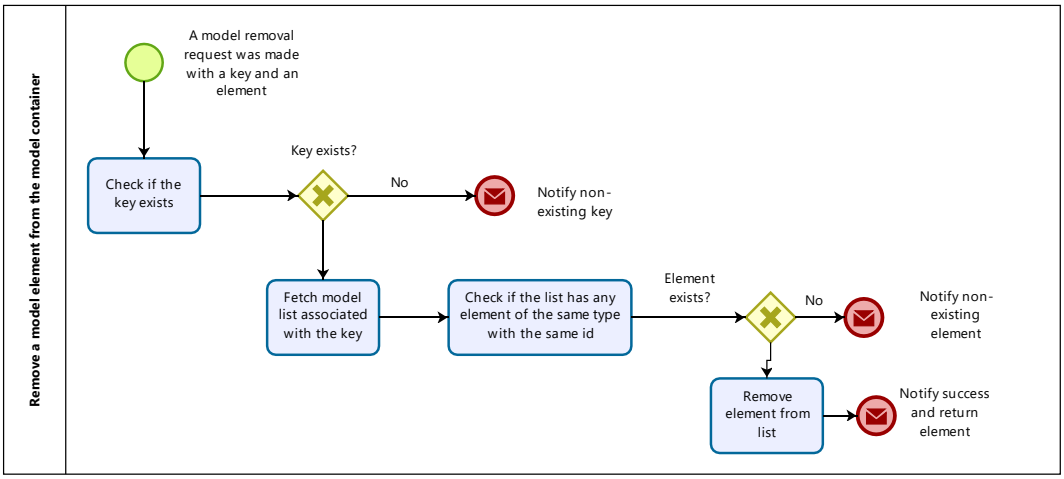


Figure 96. Remove element from container

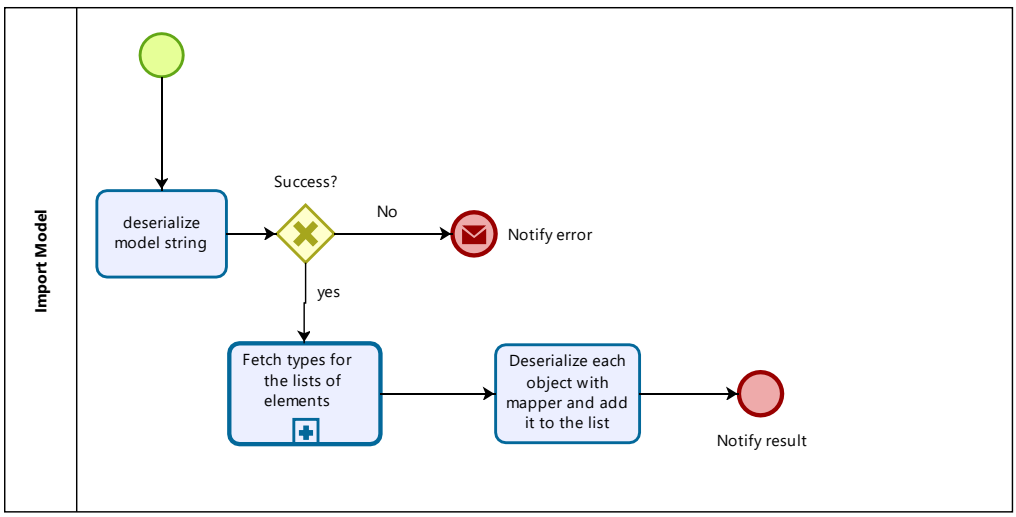


Figure 97. Import container

Model Support test list

Code	Description	Result
TST061	Validate valid model element	Success
TST062	Validate invalid model element	Success
TST063	Convert model element to json	Failure
TST064	Validate valid model element composition	Success
TST064	Validate invalid model element composition	Failure
TST065	Validate valid model element aggregation	Success
TST066	Validate invalid model element aggregation	Failure
TST067	Validate valid element’s structure (attributes)	Success
TST067	Validate invalid element’s structure (attributes)	Success
TST065	Generate the element’s digest	Success
TST066	Fetch model element types in all the loaded assemblies	Success

Model Support business diagrams

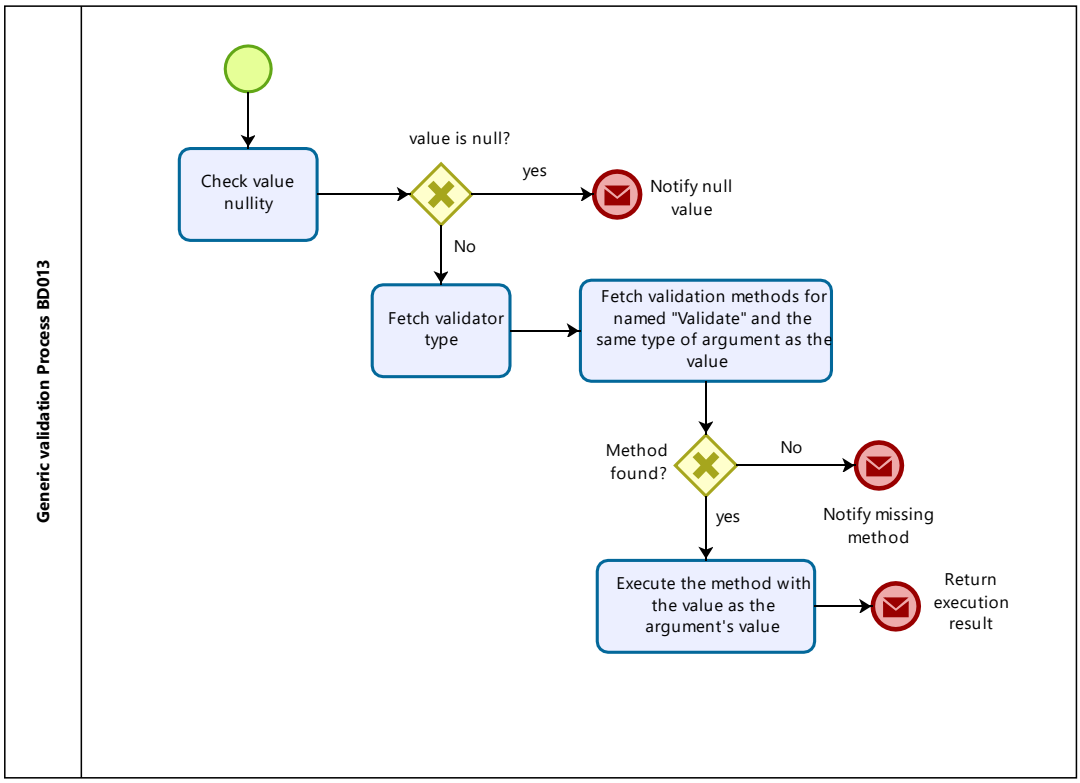


Figure 98. Generic validation

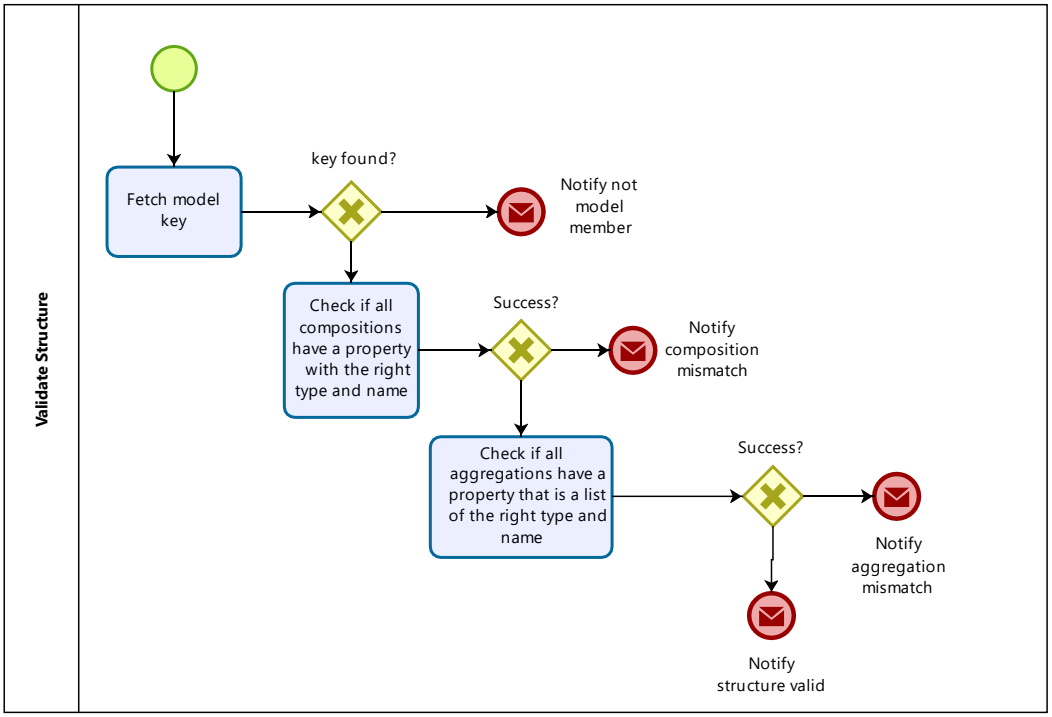


Figure 99. Model structure validation

Model Controller test list

Code	Description	Result
TST067	Add a property to an entity	Success
TST068	Validate invalid model element	Success
TST069	Convert model element to json	Failure
TST070	Validate valid model element composition	Success
TST071	Validate invalid model element composition	Failure
TST072	Validate valid model element aggregation	Success
TST073	Validate invalid model element aggregationwd	Failure
TST074	Validate valid element’s structure (attributes)	Success
TST075	Validate invalid element’s structure (attributes)	Success
TST076	Generate the element’s digest	Success
TST077	Fetch model element types in all the loaded assemblies	Success

Model Controller business diagrams

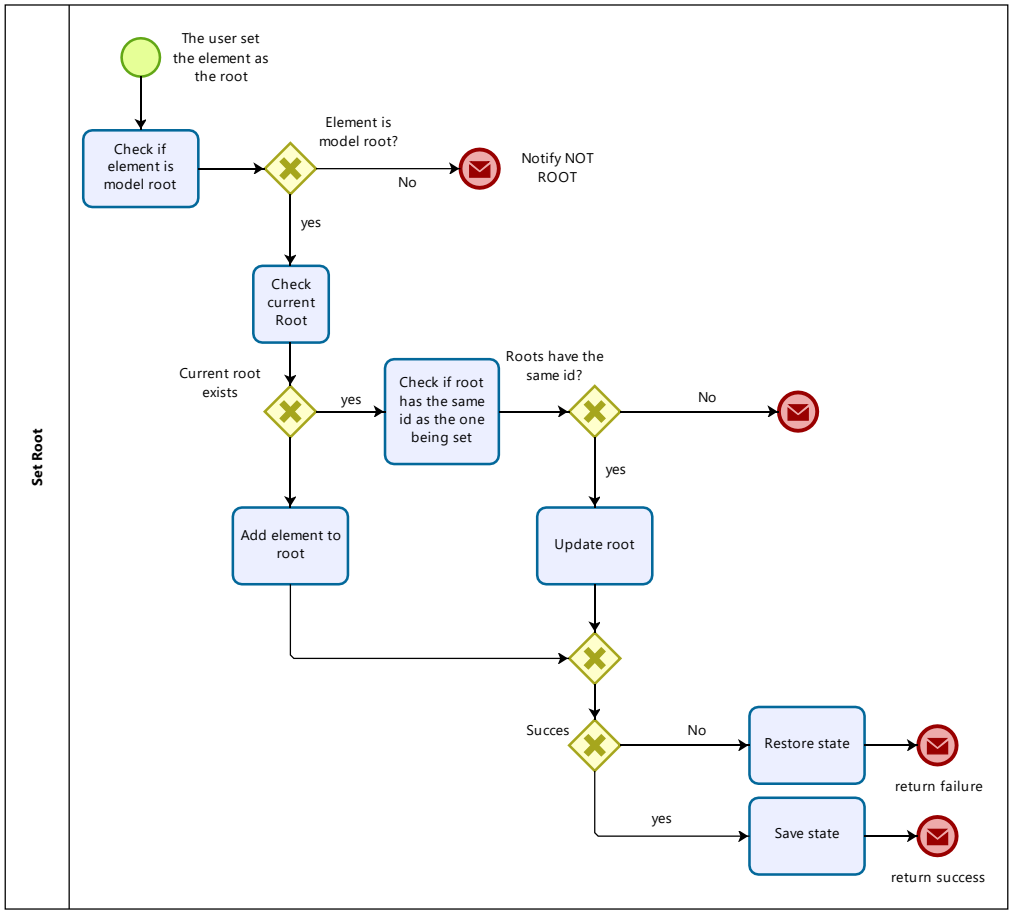


Figure 100. Setting the root model element (solution) in the model

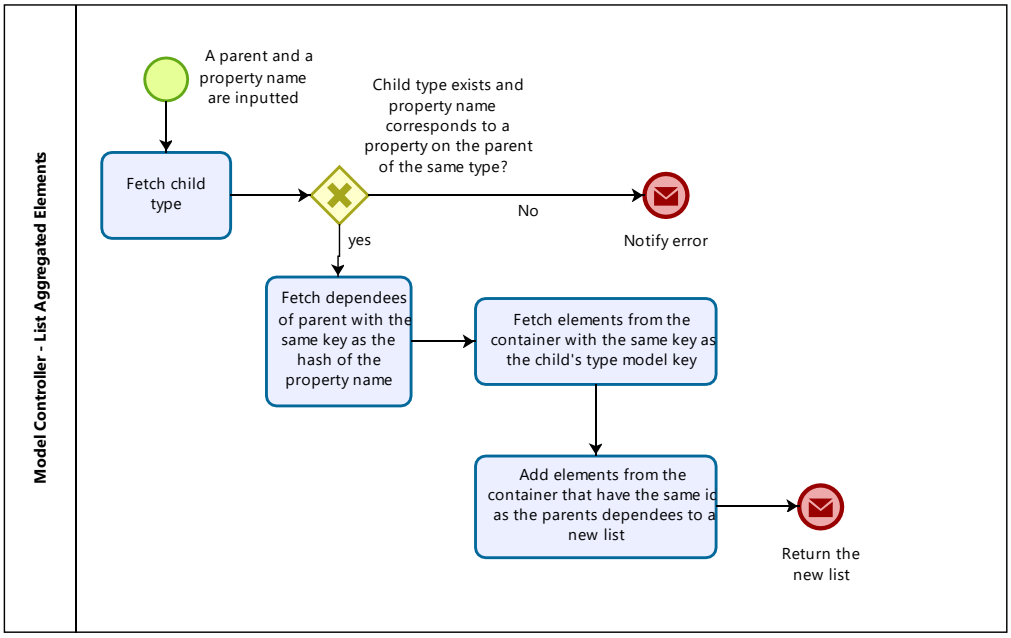


Figure 101. Listing aggregated model elements from an element

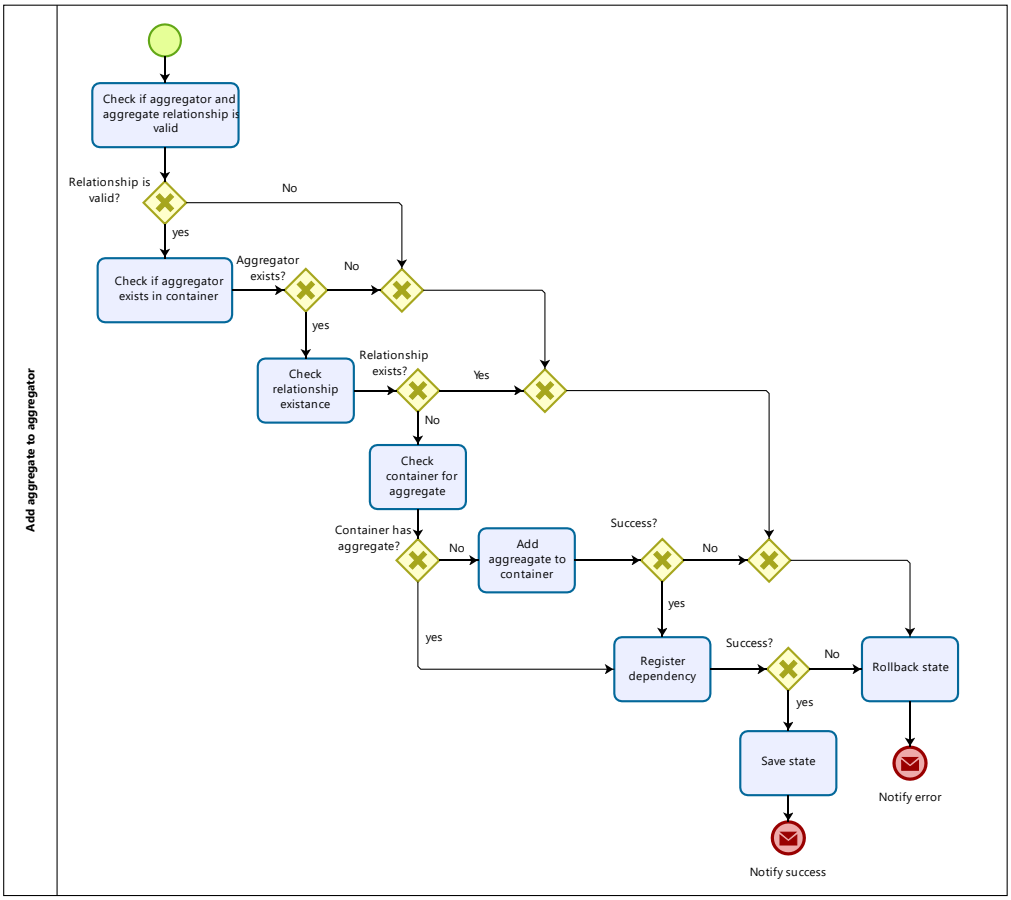


Figure 102. Aggregating a model element to another one

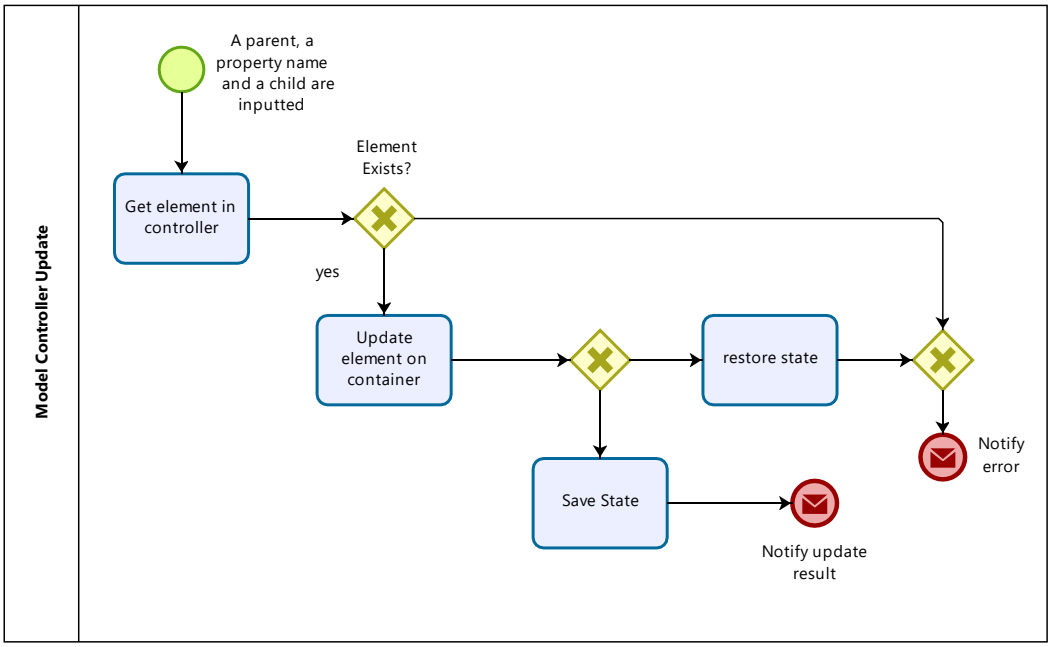


Figure 103. Updating an aggregated model element

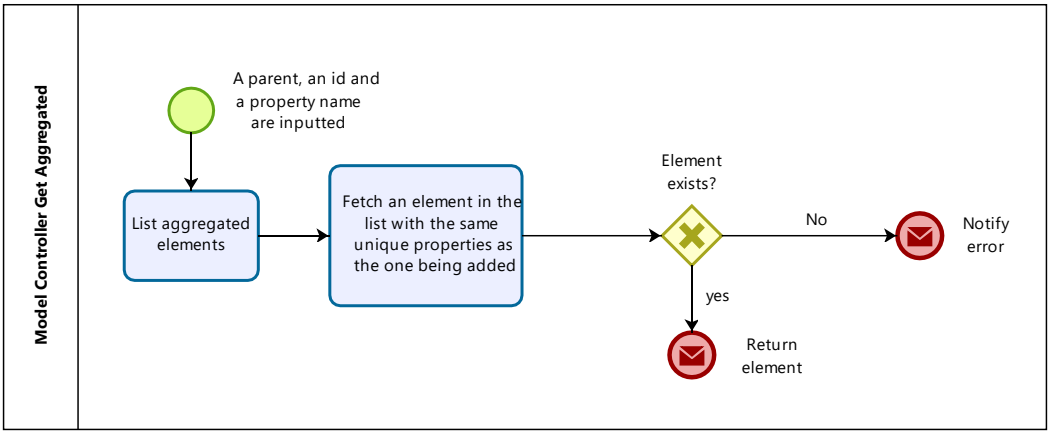


Figure 104. Fetching an aggregated model element

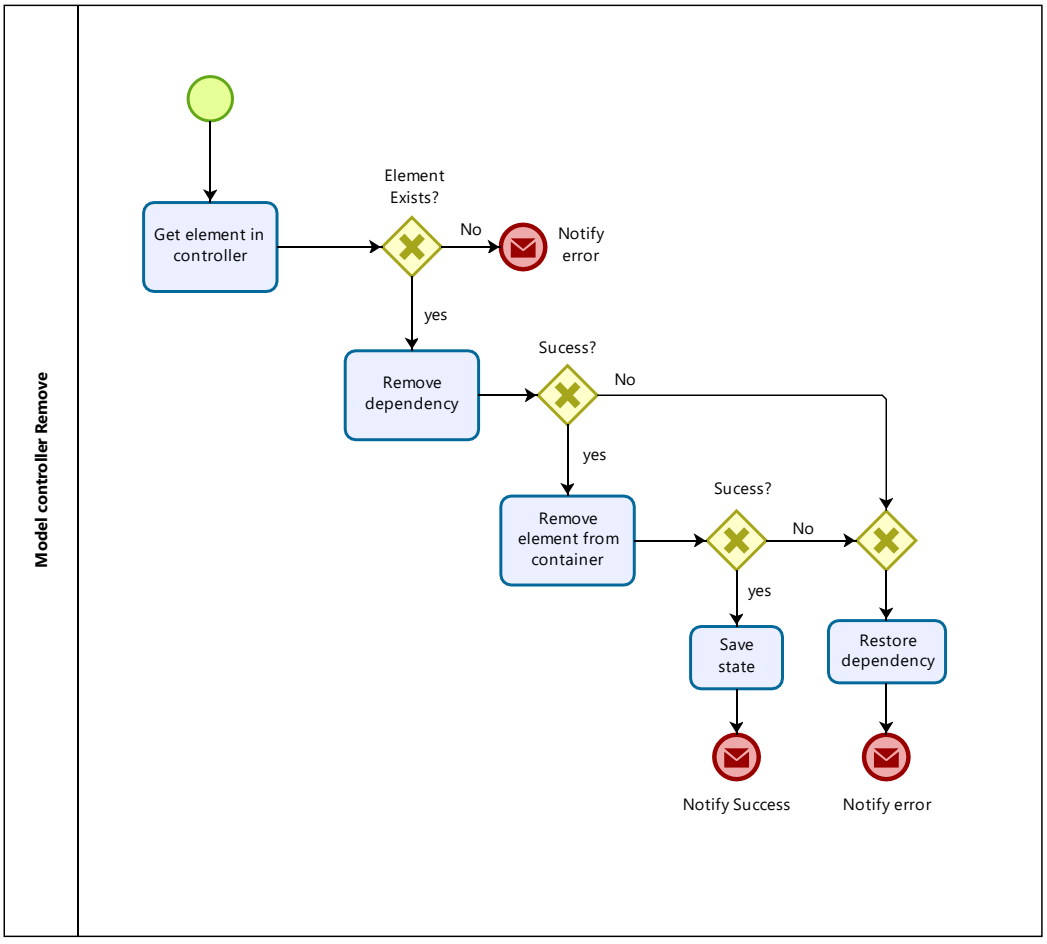


Figure 105. Aggregating a model element to another one

Layered Application Model library

Class diagrams

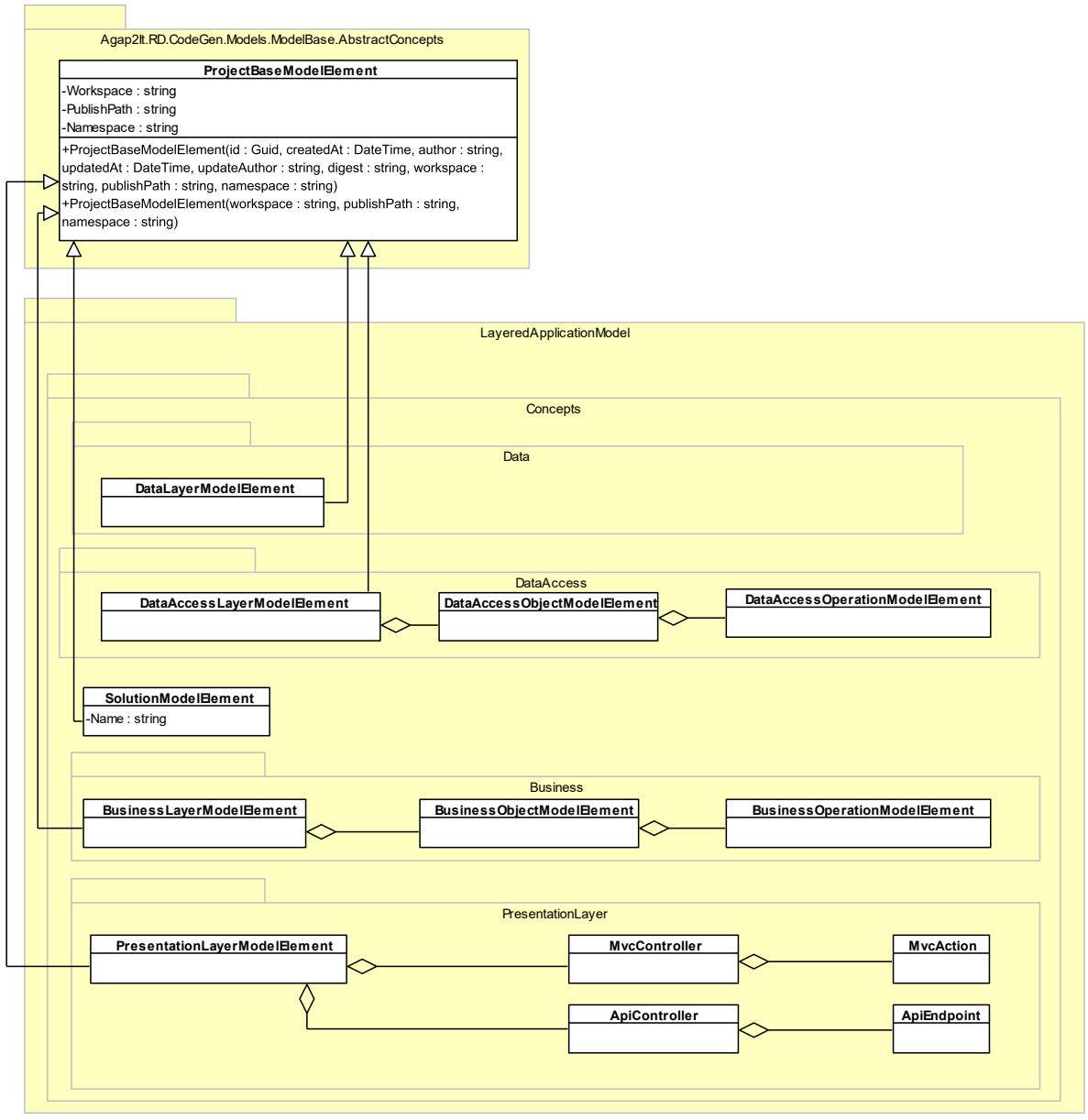


Figure 106. Layered application model class diagram focused in the aggregations between model elements

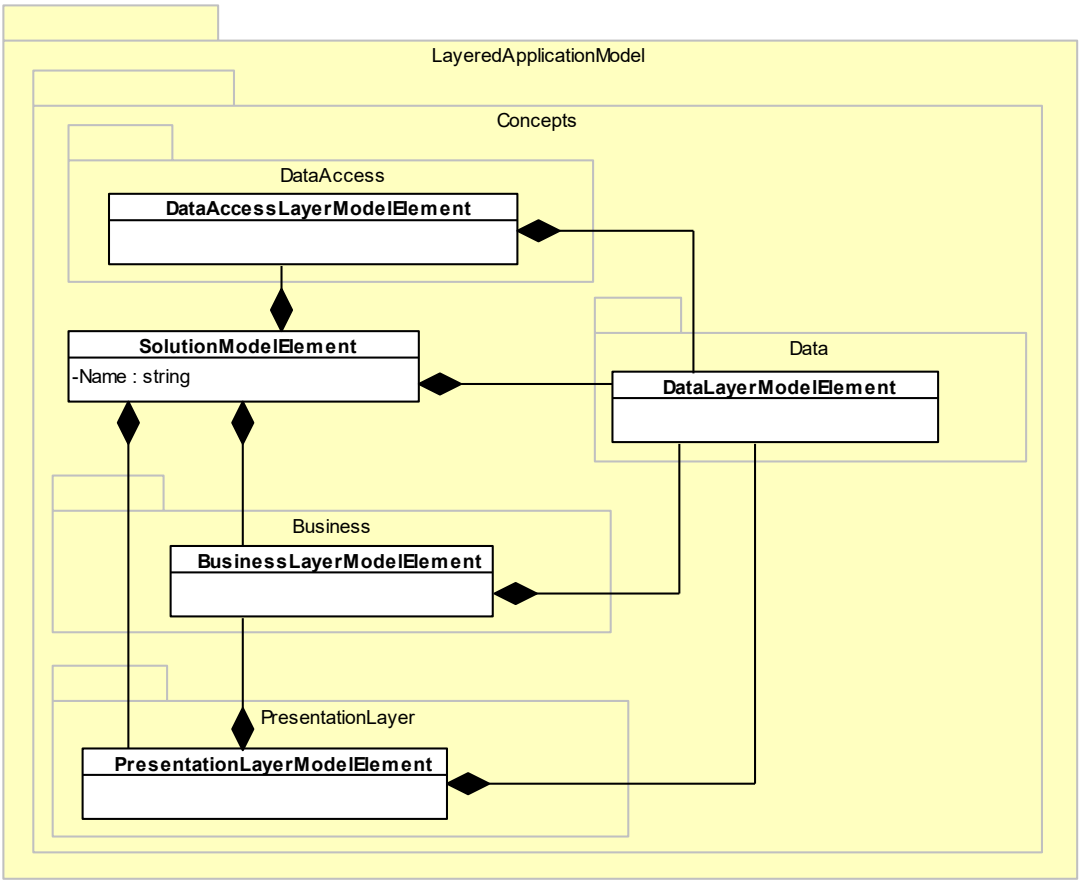
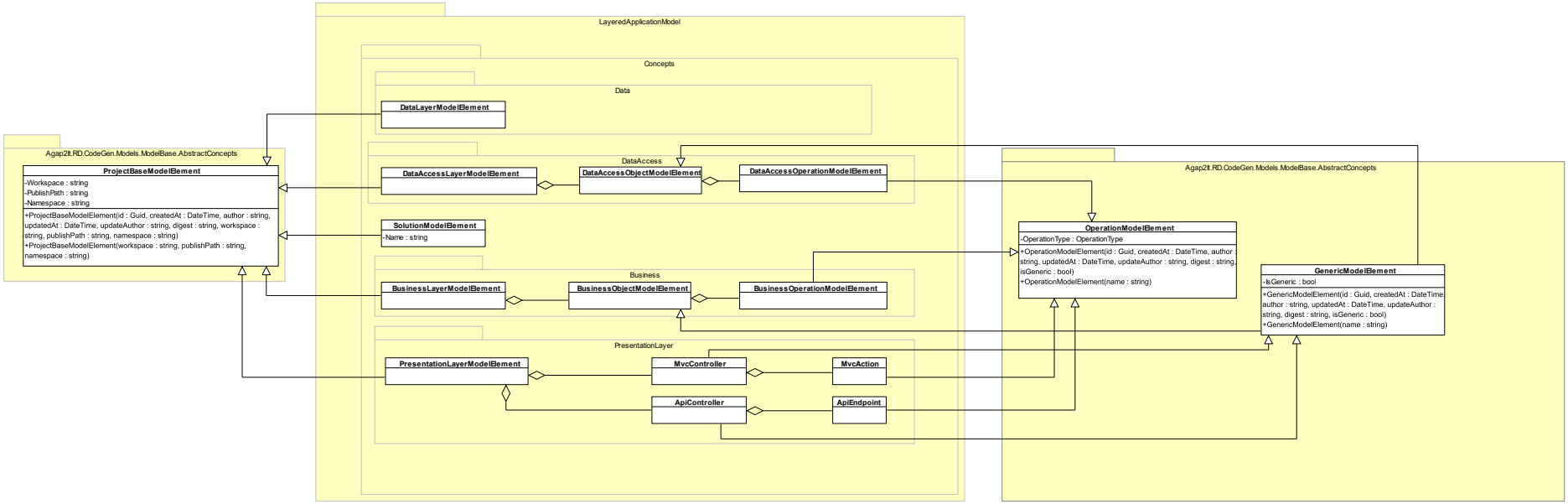


Figure 107. Layered application model class diagram focused in the compositions between project model elements


LayeredApplicationModel
Concepts
Data
DataLayerModelElement
DataAccess
DataAccessLayerModelElement
DataAccessObjectModelElement
DataAccessOperationModelElement
SolutionModelElement
-Name : string
Business
BusinessLayerModelElement
BusinessObjectModelElement
BusinessOperationModelElement
PresentationLayer
PresentationLayerModelElement
MvcController
MvcAction
ApiController
ApiEndpoint
Agap2lt.RD.CodeGen.Models.ModelBase.AbstractConcepts
ProjectBaseModelElement
-Workspace : string
-PublishPath : string
-Namespace : string
+ProjectBaseModelElement(id : Guid, createdAt : DateTime, author : string, updatedAt : DateTime, updateAuthor : string, digest : string, workspace : string, publishPath : string, namespace : string)
+ProjectBaseModelElement(workspace : string, publishPath : string, namespace : string)
Agap2lt.RD.CodeGen.Models.ModelBase.AbstractConcepts
OperationModelElement
-OperationType : OperationType
+OperationModelElement(id : Guid, createdAt : DateTime, author : string, updatedAt : DateTime, updateAuthor : string, digest : string, isGeneric : bool)
+OperationModelElement(name : string)
GenericModelElement
-IsGeneric : bool
+GenericModelElement(id : Guid, createdAt : DateTime, author : string, updatedAt : DateTime, updateAuthor : string, digest : string, isGeneric : bool)
+GenericModelElement(name : string)

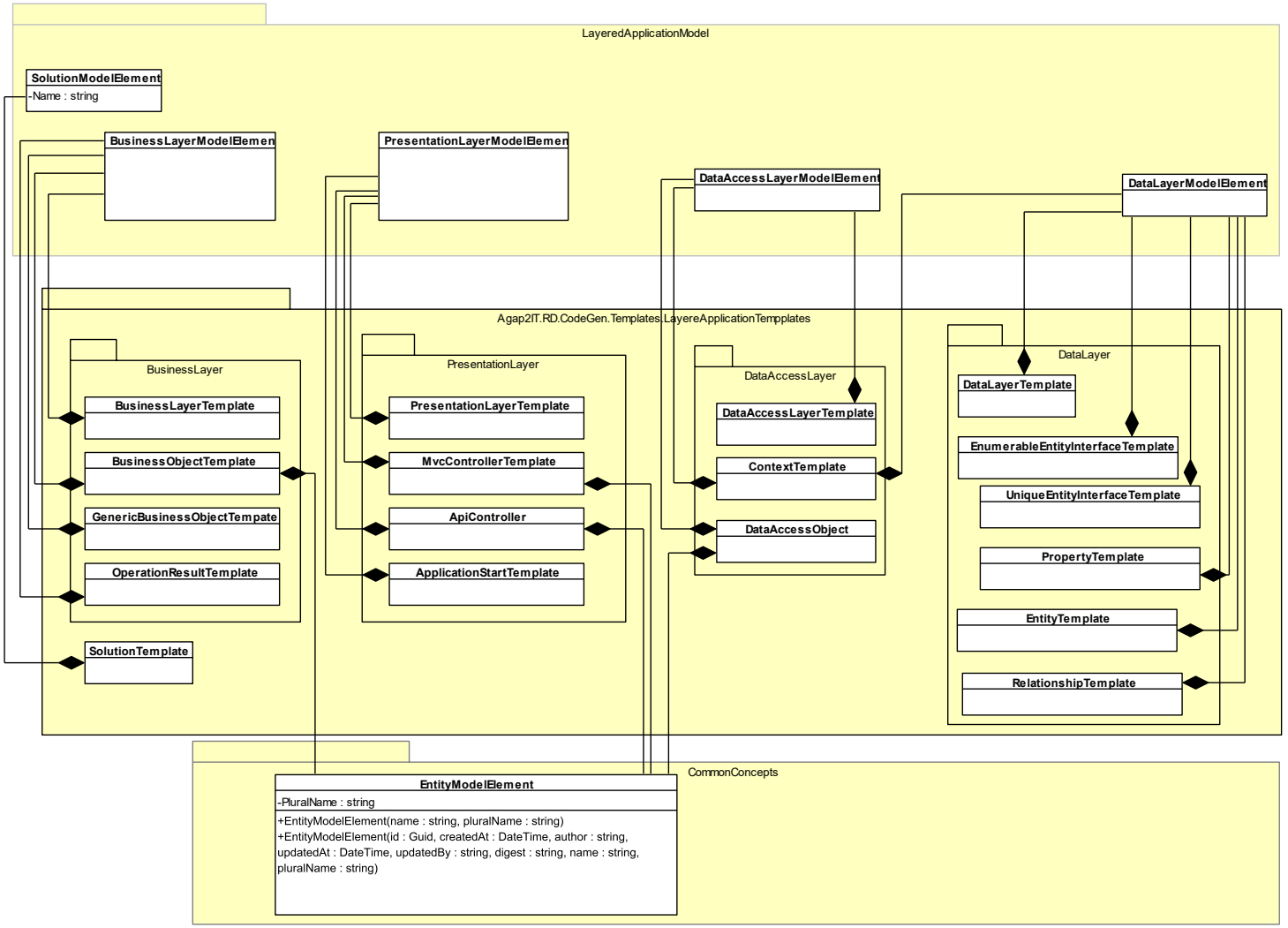


Figure 108. Layered application model class diagram focused in the compositions between model elements

Code metric results

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Models\LayeredApplicationModel (Debug)	82	107	3	67	903
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Builders	50	23	1	13	99
DataAccessObjectBuilder	46	11	1	8	34
SolutionBuilder	54	12	1	12	57
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Concepts	86	13	3	15	47
SolutionModelElement	86	13	3	15	43
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Concepts.Business	86	18	3	24	195
BusinessLayerModelElement	85	8	3	15	67
BusinessObjectModelElement	85	8	3	16	75
BusinessOperationModelElement	90	2	3	7	41
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Concepts.Data	85	6	3	15	23
DataLayerModelElement	85	6	3	15	19
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Concepts.DataAccess	87	14	3	21	169
DataAccessLayerModelElement	86	6	3	14	60
DataAccessObjectModelElement	86	6	3	14	56
DataAccessOperationModelElement	90	2	3	7	41
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Concepts.Presentation	87	26	3	28	289
ApiControllerModelElement	86	6	3	14	55
ApiEndpointModelElement	90	2	3	7	41
MvcActionModelElement	90	2	3	7	42
MvcControllerModelElement	85	6	3	15	57
PresentationLayerModelElement	85	10	3	18	74
{ } Agap2IT.RD.CodeGen.Models.LayeredApplicationModel.Properties	88	7	1	10	81
Resources	88	7	1	10	68

Figure 109. Code metrics for the layered application model library

Annex 5. Templates Libraries' documentation

As presented in the fourth section of the fourth chapter, templates are static content that has the dynamic content added to them to produce, in this case, code. The current document presents class additional information about the templates produced.

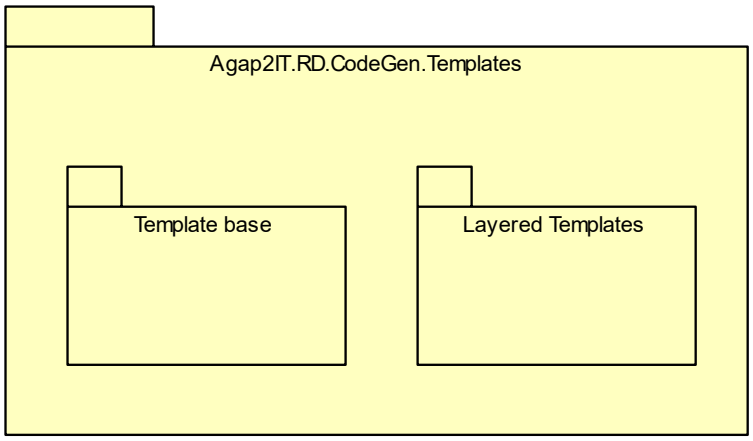


Figure 110. Templates package diagram

Template base library

Class diagram

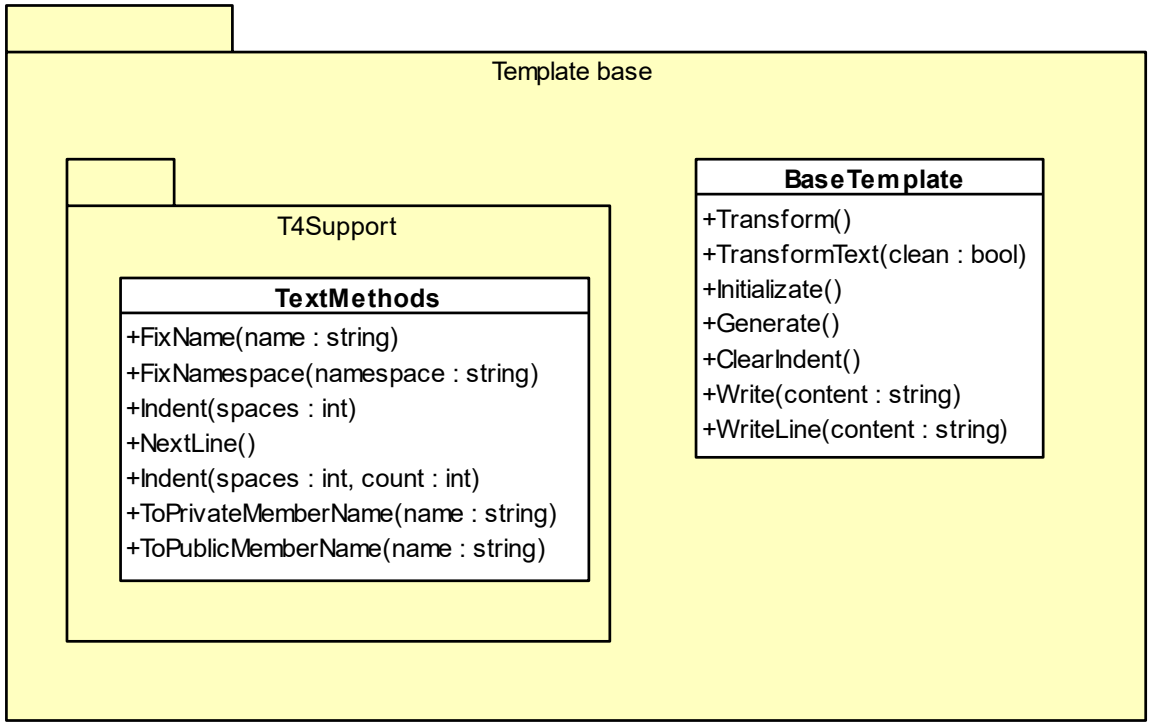


Figure 111. Template base class diagram

Code metric results

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Templates\TemplateBase (Debug)	80	62	1	26	357
Agap2IT.RD.CodeGen.Templates.TemplateBase.Templates	78	42	1	16	236
RuntimeTextTemplateBase.ToStringInstanceHelper	76	6	1	5	38
RuntimeTextTemplateBase	80	36	1	14	232
Agap2IT.RD.CodeGen.Templates.TemplateBase.T4Support	79	14	1	3	49
TextMethods	79	14	1	3	45
Agap2IT.RD.CodeGen.Templates.TemplateBase.Properties	88	6	1	10	72
Resources	88	6	1	10	59

Figure 112. Code metric results for the template base library

Layered Model Template library

Class diagram

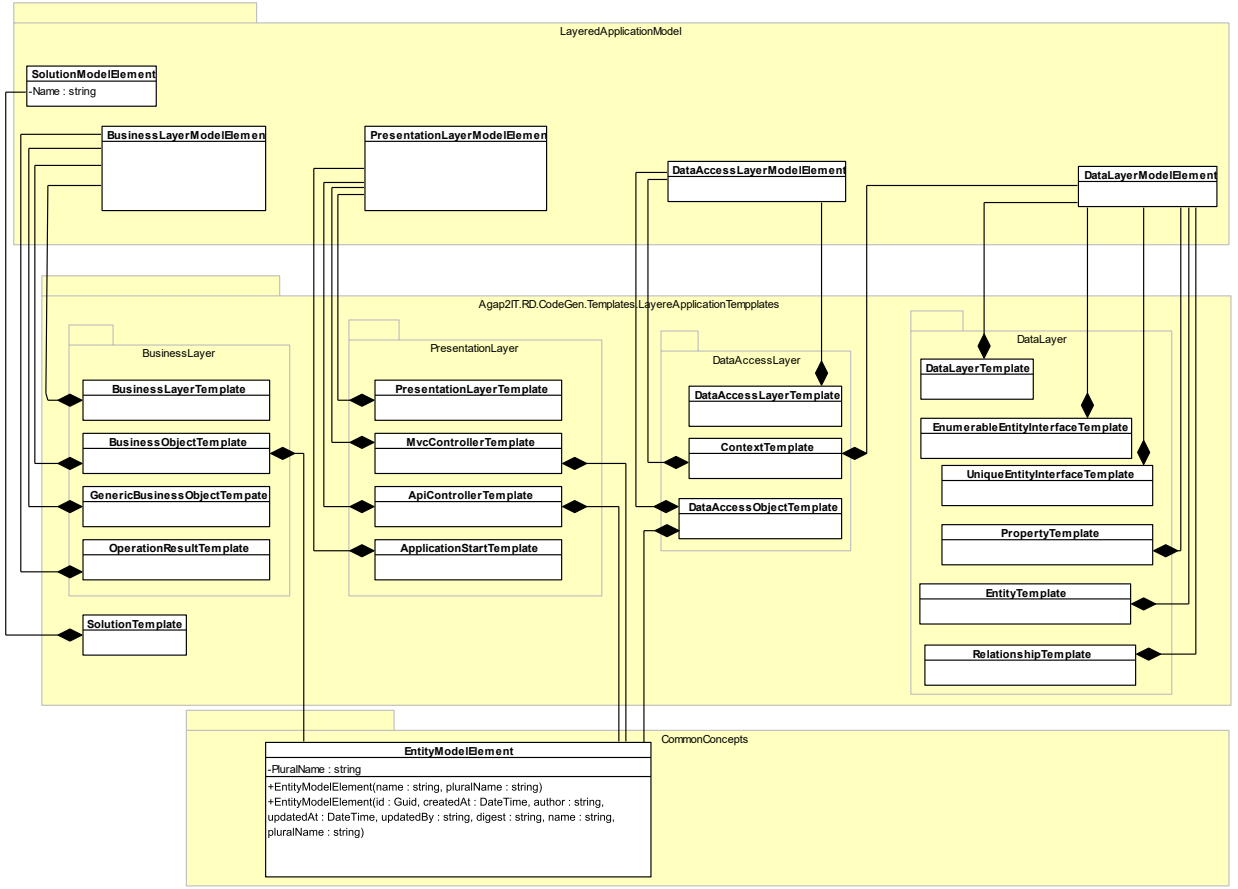


Figure 113. Class diagram for the layered application template library

Code Metrics

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Templates\LayeredApplicationTemplates (Debug)	66	208	2	46	3 212
Agap2IT.RD.CodeGen.Templates.LayeredApplicationTemplate	73	42	2	17	381
OperationResult	70	7	2	8	90
OperationResultBase	75	29	1	12	274
OperationResultBase.ToStringInstanceHelper	74	6	1	5	48
Agap2IT.RD.CodeGen.Templates.LayeredApplicationTemplate	57	64	2	24	1 178
DotNetContextTemplate	62	17	2	13	223
DotNetDataAccessObjectTemplate	59	28	2	23	391
DotNetGenericDataAccessObjectTemplate	52	19	2	10	492
Agap2IT.RD.CodeGen.Templates.LayeredApplicationTemplate	59	46	2	22	739
DotNetEntityTemplate	60	16	2	17	218
DotNetPropertyTemplate	56	12	2	12	257
DotNetRelationshipTemplate	62	18	2	12	189
Agap2IT.RD.CodeGen.Templates.LayeredApplicationTemplate	73	16	2	8	211
DotNetEnumerableTemplate	73	8	2	8	82
DotNetUniqueTemplate	73	8	2	8	83
Agap2IT.RD.CodeGen.Templates.LayeredApplicationTemplate	69	40	2	13	703
CoreBusinessLayerTemplate	72	8	2	9	86
CoreDataAccessLayerTemplate	72	8	2	9	85
CoreDataLayerTemplate	73	8	2	9	82
CorePresentationLayerTemplate	76	8	2	7	105
CoreSolutionTemplate	56	8	2	13	231

Figure 114. Code metric results for the layered templates library

Annex 6. User Interface documentation

As presented in the sixth section of the fourth chapter, there are several user interfaces that can be used to provide access to the application. Based on the research, and the interface types requested, besides the library, both the command-line interface and the visual studio interface were developed. The current annex presents the proposed package diagram, the extension's navigation diagram and the mockups produced.

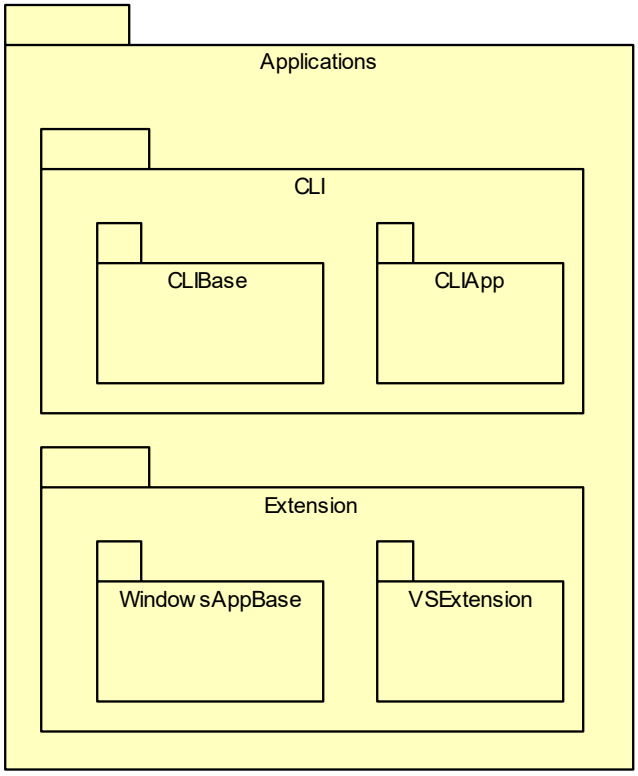


Figure 115. Applications' package diagram

Visual Studio Extension

Navigation Diagram

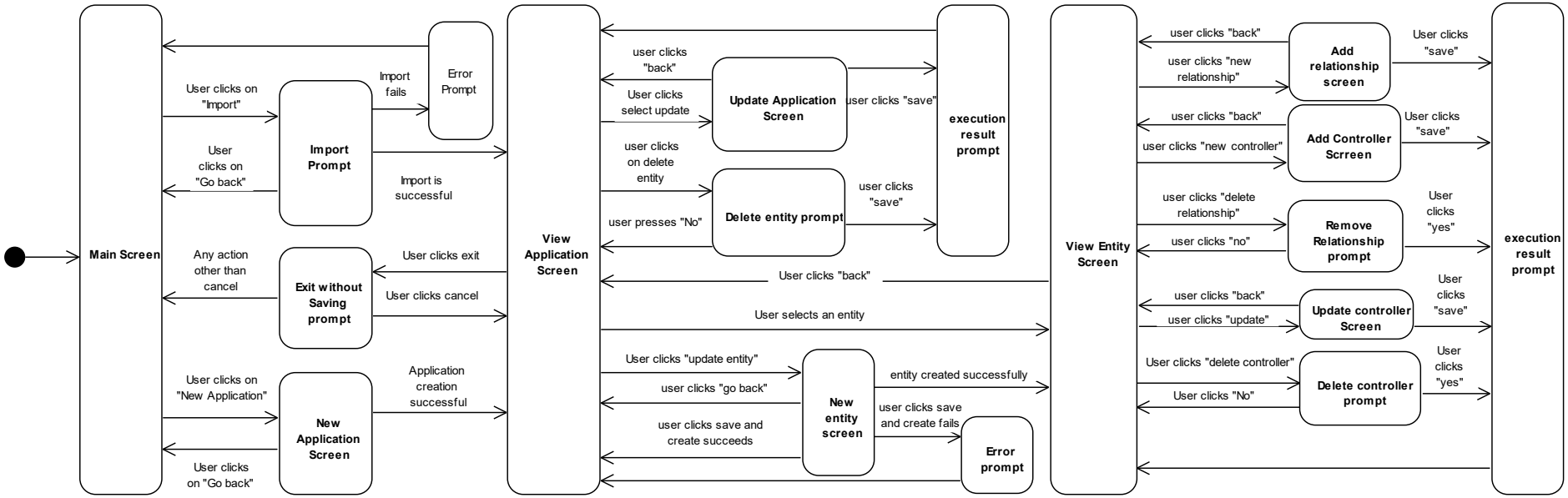


Figure 116. CodeGen extension navigation diagram

Mockups

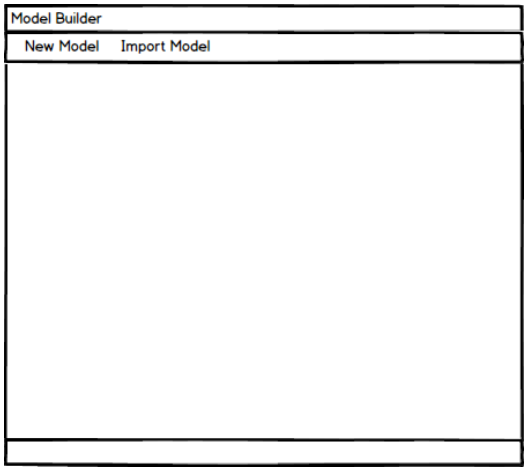


Figure 117. Main Page mockup

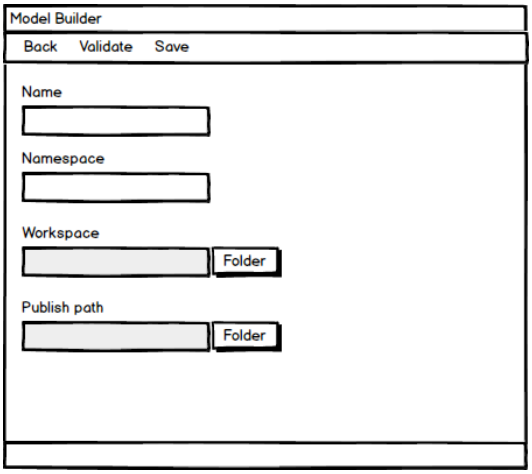


Figure 118. "Create new application" page mockup

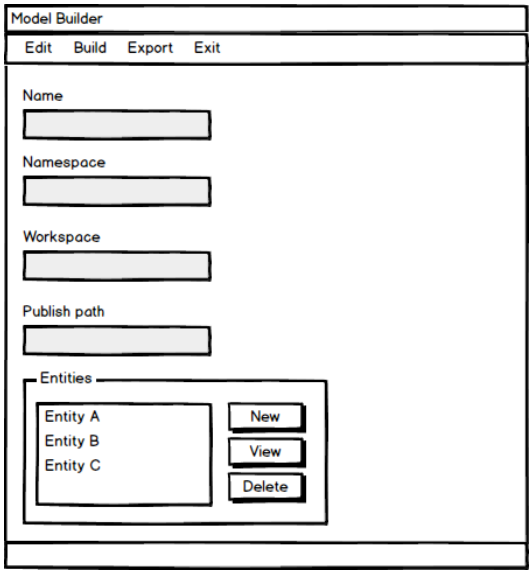


Figure 119. "View application" page mockup

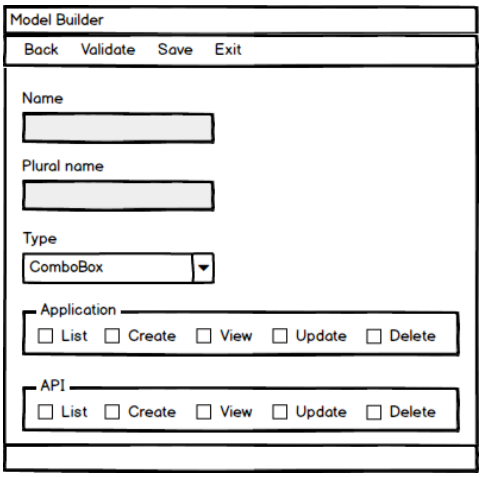


Figure 120. “Create new entity" page mockup

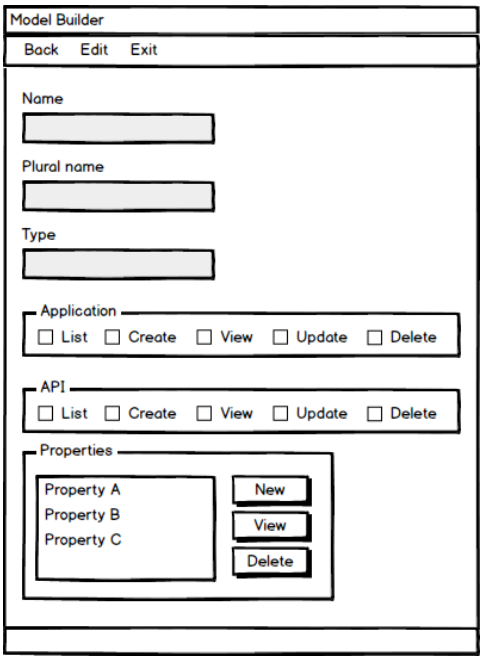


Figure 121. "View Entity" page mockup

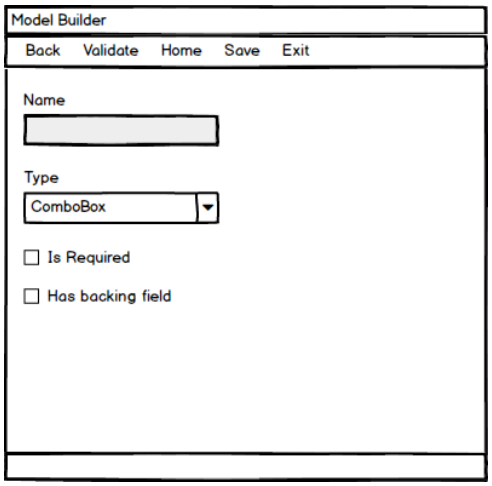


Figure 122. "Create property" page mockup

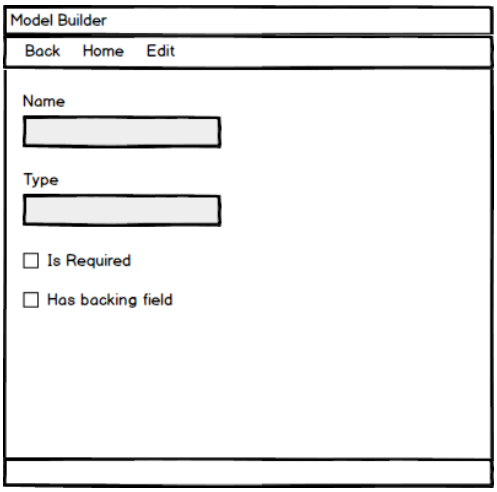


Figure 123. "View property" page mockup

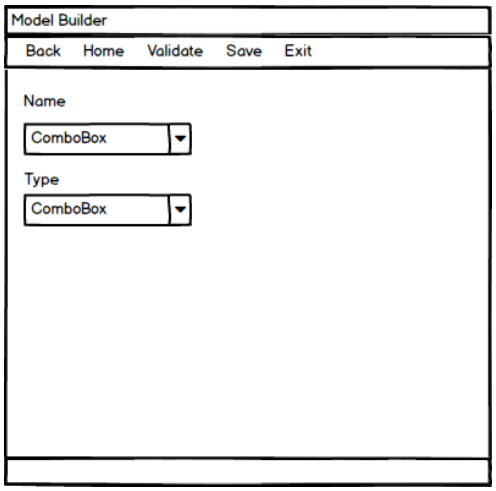


Figure 124. "New Relationship" page mockup