Université de Montréal

Towards using intelligent techniques to assist software specialists in their tasks

par

Oussama Ben Sghaier

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maîtrise sciences (M.Sc.)
en Intelligence Artificielle

November 30, 2020

© Oussama Ben Sghaier, 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Towards using intelligent techniques to assist software specialists in their tasks

présenté par

Oussama Ben Sghaier

a été évalué par un jury composé des personnes suivantes :

Eugene Syriani
(président-rapporteur)

Houari Sahraoui
(directeur de recherche)

Michalis Famelis
(membre du jury)

Sommaire

L'automatisation et l'intelligence constituent des préoccupations majeures dans le domaine de l'Informatique. Avec l'évolution accrue de l'Intelligence Artificielle, les chercheurs et l'industrie se sont orientés vers l'utilisation des modèles d'apprentissage automatique et d'apprentissage profond pour optimiser les tâches, automatiser les pipelines et construire des systèmes intelligents. Les grandes capacités de l'Intelligence Artificielle ont rendu possible d'imiter et même surpasser l'intelligence humaine dans certains cas aussi bien que d'automatiser les tâches manuelles tout en augmentant la précision, la qualité et l'efficacité. En fait, l'accomplissement de tâches informatiques nécessite des connaissances, une expertise et des compétences bien spécifiques au domaine. Grâce aux puissantes capacités de l'intelligence artificielle, nous pouvons déduire ces connaissances en utilisant des techniques d'apprentissage automatique et profond appliquées à des données historiques représentant des expériences antérieures. Ceci permettra, éventuellement, d'alléger le fardeau des spécialistes logiciel et de débrider toute la puissance de l'intelligence humaine. Par conséquent, libérer les spécialistes de la corvée et des tâches ordinaires leurs permettra, certainement, de consacrer plus du temps à des activités plus précieuses.

En particulier, l'Ingénierie dirigée par les modèles est un sous-domaine de l'informatique qui vise à élever le niveau d'abstraction des langages, d'automatiser la production des applications et de se concentrer davantage sur les spécificités du domaine. Ceci permet de déplacer l'effort mis sur l'implémentation vers un niveau plus élevé axé sur la conception, la prise de décision. Ainsi, ceci permet d'augmenter la qualité, l'efficacité et productivité de la création des applications.

La conception des métamodèles est une tâche primordiale dans l'ingénierie dirigée par les modèles. Par conséquent, il est important de maintenir une bonne qualité des métamodèles étant donné qu'ils constituent un artéfact primaire et fondamental. Les mauvais choix de conception, ainsi que les changements conceptuels répétitifs dus à l'évolution permanente des exigences, pourraient dégrader la qualité du métamodèle. En effet, l'accumulation de mauvais choix de conception et la dégradation de la qualité pourraient entraîner des résultats négatifs sur le long terme. Ainsi, la restructuration des métamodèles est une tâche importante qui vise

à améliorer et à maintenir une bonne qualité des métamodèles en terme de maintenabilité, réutilisabilité et extensibilité, etc.

De plus, la tâche de restructuration des métamodèles est délicate et compliquée, notamment, lorsqu'il s'agit de grands modèles. De là, automatiser ou encore assister les architectes dans cette tâche est très bénéfique et avantageux. Par conséquent, les architectes de métamodèles pourraient se concentrer sur des tâches plus précieuses qui nécessitent de la créativité, de l'intuition et de l'intelligence humaine.

Dans ce mémoire, nous proposons une cartographie des tâches qui pourraient être automatisées ou bien améliorées moyennant des techniques d'intelligence artificielle. Ensuite, nous sélectionnons la tâche de métamodélisation et nous essayons d'automatiser le processus de refactoring des métamodèles. A cet égard, nous proposons deux approches différentes: une première approche qui consiste à utiliser un algorithme génétique pour optimiser des critères de qualité et recommander des solutions de refactoring, et une seconde approche qui consiste à définir une spécification d'un métamodèle en entrée, encoder les attributs de qualité et l'absence des design smells comme un ensemble de contraintes et les satisfaire en utilisant Alloy.

Mots clés: Intelligence Artificielle, Ingénierie dirigée par les modèles, Génie Logiciel, Apprentissage Automatique, Optimisation Multi-objectif.

Summary

Automation and intelligence constitute a major preoccupation in the field of software engineering. With the great evolution of Artificial Intelligence, researchers and industry were steered to the use of Machine Learning and Deep Learning models to optimize tasks, automate pipelines, and build intelligent systems. The big capabilities of Artificial Intelligence make it possible to imitate and even outperform human intelligence in some cases as well as to automate manual tasks while rising accuracy, quality, and efficiency.

In fact, accomplishing software-related tasks requires specific knowledge and skills. Thanks to the powerful capabilities of Artificial Intelligence, we could infer that expertise from historical experience using machine learning techniques. This would alleviate the burden on software specialists and allow them to focus on valuable tasks.

In particular, Model-Driven Engineering is an evolving field that aims to raise the abstraction level of languages and to focus more on domain specificities. This allows shifting the effort put on the implementation and low-level programming to a higher point of view focused on design, architecture, and decision making. Thereby, this will increase the efficiency and productivity of creating applications.

For its part, the design of metamodels is a substantial task in Model-Driven Engineering. Accordingly, it is important to maintain a high-level quality of metamodels because they constitute a primary and fundamental artifact. However, the bad design choices as well as the repetitive design modifications, due to the evolution of requirements, could deteriorate the quality of the metamodel. The accumulation of bad design choices and quality degradation could imply negative outcomes in the long term. Thus, refactoring metamodels is a very important task. It aims to improve and maintain good quality characteristics of metamodels such as maintainability, reusability, extendibility, etc.

Moreover, the refactoring task of metamodels is complex, especially, when dealing with large designs. Therefore, automating and assisting architects in this task is advantageous since they could focus on more valuable tasks that require human intuition.

In this thesis, we propose a cartography of the potential tasks that we could either automate or improve using Artificial Intelligence techniques. Then, we select the metamodeling

task and we tackle the problem of metamodel refactoring. We suggest two different approaches: A first approach that consists of using a genetic algorithm to optimize set quality attributes and recommend candidate metamodel refactoring solutions. A second approach based on mathematical logic that consists of defining the specification of an input metamodel, encoding the quality attributes and the absence of smells as a set of constraints and finally satisfying these constraints using Alloy.

Keywords: Artificial Intelligence, Model-Driven Engineering, Software Engineering, Machine Learning, Multi-Objective Optimization.

Contents

List of tables

List of figures

Acronyms & Abbreviations

1. AI	Artificial Intelligence
2. ML	Machine Learning
3. DL	Deep Learning
4. NLP	Natural Language Processing
5. MDE	Model-Driven Engineering
6. GA	Genetic Algorithm
7. MOO	Multi-Objective Optimization
8. NSGA-II	Non-Dominated Sorting Genetic Algorithm II
9. UML	Unified Modeling Language
10. OCL	Object Constraint Language

11. API Application Programming Interface

12. DSL Domain-Specific language

13. DSM Domain-Specific Modeling

14. OMG Object Management Group

15. MOF Meta-Object Facility

16. RS Random Search

17. CI/CD Continuous Integration & Continuous Delivery

Dedications

I am most thankful to my God,
the most beneficent and the most merciful.

I dedicate this work to my family and special friends.

In the memory of my grandmothers and grandfathers.

A special feeling of gratitude to my loving parents,
who waited patiently for the fruits of their good education,
and whose words of encouragement and pushes for tenacity ring in my ears.
This work is the fruit of the sacrifices you have made for my education and my well-being.

To my sweetheart brother and sister,
In testimony of my deep love,
and to whom I wish all success and happiness.
To the flower of my life,
the one that makes my heart beat,
I offer you all the beautiful expressions of the language.
May God unite our paths for a long serene common.

To my friends,
because there is no better capital than that of lasting and sincere friendship.

To all my family,
May God protect you and keep us always together and in solidarity

To everyone else who contributed to this humble work,
To all the people who are dear to me,
and all those who want to share my happiness,
I dedicate this modest work,
symbol of my deep gratitude and crowning of their assistance.

I am eternally grateful to all of you.
Thank you ..
For anything!

Acknowledgements

> Develop an attitude of gratitude, and
> give thanks for everything that
> happens to you,
> knowing that every step forward is a
> step toward achieving something bigger
> and better than your current situation.
>
> *Brian Tracy*

It is a pleasant duty to perform the recognition accumulated throughout his curriculum.

I would like to express my greatest gratitude to both the Tunisian Ministry of Higher Education and Scientific Research and the University of Montreal for co-funding this research work through many excellent scholarships.

I also address myself to the members of the honorable jury, who I thank for agreeing to consider this thesis and to kindly take the time to evaluate this humble work, hoping that they will find in it the qualities of clarity and rigor as expected.

First and foremost, I would like to express my immense gratitude and respect to my great and indescribable supervisor Professor. Houari Sahraoui for his unwavering guidance, relentless support, and astute advice. I also would like to express my deepest appreciation for his colossal kindness, tireless diplomacy, and his unblemished geniality. I am super proud to be one of his students and glad to continue working with him during my Ph.D. One simply could not wish for a better supervisor

Then, I would like to extend my heartiest thanks to all my colleagues in the GEODES laboratory for sharing knowledge and a good work atmosphere.

Last but not least, I wish to thank the professors and the administrators of UdeM. Hats off to you for your hard work, for keeping moving forward and for maintaining good conditions,

and a nice environment for education, teaching, and research, even in the difficult situation of COVID-19.

Finally, I would like to thank all those who, from far and near, contributed to the realization of this work.

Chapter 1

Introduction

1.1. Research context

The quality of software pipelines and tasks is very important for the sake of delivering high software quality and in order to maintain an elevated level of software practitioner's efficiency [**61, 78, 77, 62**]. Currently, IT companies are granting prominent importance towards the refinement and optimization of their IT pipelines and tasks to increase their efficiency in producing deliverables (Machine Learning models, software, services, etc.) with better quality [**61, 18, 90**]. Furthermore, many tools have been designed and implemented to assist software specialists (data scientists, software developers, data engineers, etc.) in their tasks [**104, 112, 15, 122, 111**]. These tools enable software specialists' assistance aiming to increase their efficiency and effectiveness, decrease error-proneness, and maintain a lofty level of productivity.

In this regard, many researchers were oriented towards the improvement of software pipelines and deliverables' quality [**121**]. Most of the researches were software engineering-oriented and focused mainly on development and testing enhancement [**122, 42, 128, 104, 30, 41**]. Accordingly, a lot of work, that has been done, was related to models and source code refactoring (bad smells detection and correction), testing, IntelliSense or code completion, etc. [**15, 8, 109**] Yet, few research contributions were related to the optimization of some fields such as Model-Driven Engineering (MDE) or Artificial Intelligence tasks due to the recent growth of these fields that are still not mature enough. Indeed, these fields are witnessing an increased evolution in the last decade. Although there is still a lot to be done in the mature fields, the growing fields should be tackled more intensively since there are tremendous related research directions that could be addressed to contribute to the success and achievements of these fields.

On the other side, the field of Artificial Intelligence (AI) has witnessed an outstanding evolution recently. The incommensurable potential of AI, especially its sub-fields machine learning and deep learning, made it one of the most successful fields nowadays. It provides

powerful capabilities to imitate human-intelligence and build smart systems; This exceptional potential of AI made it an appealing target in high-demand and broadly used in other domains. Thus, AI represents today, the backbone of many smart systems [**91, 13, 83**], the key solution for many challenging problems, and the fuel of innovation and creativity [**31, 86**].

Consequently, AI is being used to solve challenging and complex problems. Machine learning and Deep learning models have a potent capacity to infer knowledge from raw data and to learn complex and hidden patterns. This forms a major asset allowing to imitate and even outperform human intelligence in some cases [**56, 120**]. If integrated wisely and properly, AI could be a great asset to other fields by injecting intelligence, smoothing and accelerating workflows, optimizing tasks, improving artifacts' quality, assisting engineers, and raising their effectiveness, etc.

One of the most interesting fields where we could use AI is Software Engineering [**34**]. So far, there are some AI-based tools have been designed and developed, some research work has been done to benefit from AI to assist software specialists in their tasks, to optimize pipelines, and improve software quality. However, there is still a lot to be done. As it is aforesaid, the research field was narrow and limited to some specific tasks (development, testing, security, etc.) and it has investigated less the opportunities of using AI to promote other types of tasks such as modeling, monitoring of both resources and ML models, CI/CD, designing and developing ML models, ML evaluation and security, etc.

Artificial Intelligence could be employed in different forms: automation, recommendation, optimization, detection, analysis, prediction, forecast, etc. We should take advantage of these benefits to carry up the fields of software engineering, MDE, etc. to an upper-level replete with automation and intelligence. Along with putting efforts into improving AI techniques, a new research direction should be adopted to investigate new improvement opportunities within the software development process.

Indeed, integrating AI into the development processes can help to address several major challenges that software specialists are facing for several years, including the difficulty of automating many activities with a high propensity to introduce errors. Furthermore, the capacity of Machine Learning to detect hidden patterns, to provide accurate predictions and recommendations, to learn from past experiences, and to infer knowledge; constitutes a considerable opportunity to alleviate the burden on the different software actors in the accomplishment and optimization of their tasks.

In fact, one important task in the software generation workflow is the design phase in which some metamodels are produced. Metamodels are a central and fundamental pillar in Model-Driven Engineering (MDE), allowing to formalize domain concepts and build domain-specific languages. The key idea behind this paradigm is that dealing with domain models is

significantly easier for stakeholders than directly dealing with low-level implementation artifacts as it allows to abstract and automates the mechanism of domain applications generation and transformation. The major benefits of MDE are the increase in productivity, efficiency, quality, and automation. Besides, MDE allows empowering domain experts, bridge the gap between domain experts and IT and it allows them also to focus on business problems instead of the technical details. On the other hand, MDE decreases cost, time-to-market, and error-proneness. Hence, this paradigm has widely spread and became increasingly popular and more used by dint of its enormous advantages.

Metamodels, the central artifact in MDE, are subject to changes, evolution, and maintenance to adapt to the dynamic environment, the varying domain, and to the requirements' evolution. Thus, these artifacts should be maintained carefully, and they should be of good quality. Faced with the evolution and changes in the requirements, we should always ensure a good metamodels quality since the majority of the other artifacts depend on them. Here, AI could be an interesting and appealing candidate for this task. We could use some machine learning techniques to help model architects design and maintain robust and high-quality metamodels. This could be through recommending quality improvement solutions (e.g. refactoring) or by detecting potential defects or bad design decisions.

To sum up, the powerful capabilities of AI could be exploited to improve pipelines, optimize tasks, assist, and help software specialists in their work. If used properly, AI could provide great support for the Software Engineering and MDE industry.

1.2. Problem statement

AI is being used broadly in different fields to solve challenging and complex problems, to inject a kind of intelligence to build smart systems, etc. The inherent idea behind this is the capacity of AI to capture knowledge that is difficult to elicit by humans. Even though human characters like intelligence, intuition, and creativity are unique, AI shows many capabilities in imitating human intelligence in many cases or, at least, to exploit it more efficiently for some applications.

In this thesis, we are interested in two problematics: A high-level problem where we investigate AI-based opportunities to assist software specialists and optimize software-related tasks; A low-level problem where we are interested in a specific task which is the design of metamodels. Maintaining a high level of metamodels' quality is a real challenge that should be addressed by the research community while considering the modeler's preferences to guide the refactoring task.

1.2.1. Problem 1: How can AI assist software specialists in their tasks?

As previously mentioned, AI has outstanding capabilities and could be used either to assist humans or to optimize tasks. However, what matters most is knowing when and where to use AI. Thus, it is very important to identify tasks or their parts in development pipelines where we could use AI to improve the effectiveness and promote productivity and quality.

In addition, Software specialists, including scientists and engineers need deep knowledge and wide expertise to accomplish their tasks properly and perfectly. Nevertheless, they may encounter problems and hurdles when facing difficult and complex tasks. AI allows to learn from past experiences, detect hidden patterns and infer knowledge, thus, it could be used in a way to assist the different software actors in their tasks using either automation, detection, recommendation, or guidance, etc. To get the most out of AI, it is highly critical to detect the proper manner of using it to promote the corresponding task. This should be done by identifying the appropriate AI techniques to use, the right data to learn from, and correctly employing the results to improve this task.

Furthermore, to generate software products (software, website, machine learning models, etc.), companies set up pipelines that allow to automate the deployment of those artifacts and to connect different tasks. For example, Continuous Integration/Continuous Delivery (CI/CD) is a common pipeline in IT companies as it allows to automate building, testing, and deployment of applications. Despite the high level of automation, there are still some manual tasks in these pipelines (i.e. code review). AI could be a great opportunity to increase the automation level of these pipelines, to refine,, automate and optimize them, and to inject a kind of intelligence.

Despite a large number of improvement opportunities, it is important to identify the right spots where AI could be useful to adduce profit and added value.

1.2.2. Problem 2: How to recommend relevant metamodel refactoring solutions?

One important task in the paradigm of model-driven engineering is the design of metamodels. To maintain a good quality of metamodels, we should continuously identify bad design choices and to refactor them. Many research axes related to metamodels refactoring were investigated. Some of them were based on bad smells detection and correction. Nevertheless, removing all the bad smells blindly is not necessarily the best option since correcting one bad smell can improve one quality attribute but it can also worsen another one [**5**]. Therefore, Bad smells should be removed with respect to some objective criteria which could

be the quality attributes. In fact, architects don't refactor metamodels aimlessly. However, they do it to improve some quality factors such as the understandability and reusability of the metamodel. In this context, a real challenge consists of not only recommending refactoring solutions but rather providing meaningful ones with respect to some relevant criteria.

Takeaways

We sum up the aforementioned problems in a form of short takeaways:

- **Problem 1:** It is very important to have a global overview of the potential AI-based opportunities to improve software-related tasks and to assist software specialists in their work. To the best of our knowledge, few research works have been done in this context.
- **Problem 2:** Metamodel refactoring was previously tackled by the MDE research community. However, none of those previous works has addressed the recommendation of refactoring based on the optimization of some quality criteria.

1.3. Main contributions

In this section, we present our solutions to address the aforementioned problems. These solutions are organized into three main contributions: First, we present a macro-contribution that depicts the AI-based opportunities to improve software-related tasks.

Then, we present our two other contributions focusing on a specific software-related task which is the design of metamodels. In these contributions, we propose different approaches to refactor metamodels. The refactoring is guided by some quality criteria. We use two different AI techniques: constraint solving and multi-objective optimization.

1.3.1. Contribution 1: Cartography of potential AI-based improvements for software-related tasks

In Chapter 3, we present our contribution that addresses the first problem. In this contribution, we analyze the pipelines and tasks in an industrial setting. This will give us an overview of the expertise and knowledge required to accomplish each task. Then, we depict a cartography of the improvement opportunities of these tasks and pipelines where we could infer this expertise from experience and historical data in order to automate parts of these tasks and to assist software practitioners in their work. This contribution is divided into several steps that consist of: (1) identify the tasks and pipelines done inside the AI team, (2) link each task with the corresponding knowledge and expertise required to realize it, and (3) recommend machine learning-based opportunities allowing to infer this knowledge from past experience which will allow to optimize and automate parts of these tasks.

1.3.2. Contribution 2: Quality-driven multi-objective optimization approach for metamodel refactoring

We address the second problem in two different ways. Our second contribution in chapter 4 tackles the problem of improving the quality of metamodels that is the fundamental piece in the paradigm of Model-Driven Engineering. Metamodels reside in a changing environment where requirements are evolvable. The goal of this contribution is to maintain a high-quality of metamodels by detecting and correcting metamodels' smells, not purposelessly, but, rather improve some chosen quality factors. We use a search-based approach to improve simultaneously some quality attributes in the input metamodel. Then, we recommend the resulted solutions to the modeler to select the adequate candidate solution to be automatically applied to generate the refactored metamodel.

1.3.3. Contribution 3: Recommending metamodel refactorings using constraint solving

We present our third contribution in Chapter 5 in which we try to address differently the problem of improving metamodels quality. Rather than using a meta-heuristic with multi-objective optimization, we rely on a constraint solving approach. Indeed, we use Alloy as a specification language to express the quality criteria and the absence of metamodel smells as constraints. Then, we try to satisfy these constraints using alloy as a constraint solver. The solutions that satisfy the specified constraints are the refactoring solutions that we recommend to improve the quality of the input metamodel. To sum up, in this contribution we provide a novel, robust, and formally-sound approach to refactor metamodels while satisfying quality constraints.

Takeaways

We summarize our main contributions as follows:

- **Contribution 1:** A cartography of tasks and pipelines that could be automated or improved by inferring the required expertise and knowledge to accomplish them from past experiences and historical data using AI techniques.
- **Contribution 2:** Recommend metamodel refactoring solutions using a multi-objective optimization approach of quality attributes.
- **Contribution 3:** We use Alloy as a formal specification language to encode the absence of smells and quality criteria as logical constraints. Then, we use Alloy to satisfy these constraints and recommend candidate metamodel refactoring solutions that correct bad smells while satisfying quality constraints.

1.4. Thesis structure

The remainder of this thesis is organized as follows. Chapter 2 provides a literature review on previous related works that are relevant to the main contributions of this thesis. In chapter 3, we portray a cartography of AI-based improvement opportunities identified in a real-life study of pipelines and tasks of the AI team of an industrial organization. Chapter 4 reports our contribution for recommending refactoring solutions for a given metamodel using a search-based multi-objective optimization approach with quality attributes. Chapter 5 presents another robust approach for quality-based metamodel refactoring recommendation using constraints solving. Finally, we conclude our thesis in chapter 6 by summarizing our contributions, underlining their limitations, and outlining future research directions.

Chapter 2

State of the art

Introduction

This chapter provides a thorough overview of the work related to the main contributions stated in this thesis. First, we provide the essential background needed to understand the key concepts. Then, we review the literature with regard to the main themes of this research work: (1) Analysis of AI-based improvement opportunities (2) Metamodel refactoring recommendation and design smells detection. Afterward, we identify the limitations that are addressed by our contributions.

2.1. Background

2.1.1. Model-driven engineering (MDE)

Models played a subordinate role in the software development process and were used for simple documentation purposes [**129**]. The advent of MDE has given greater priority to models by considering them as the main and most important artifact in the development process. In fact, MDE is a software development approach that relies on the design, creation, and exploitation of domain models. Moreover, MDE aims to substitute the task of writing source code with the design of models. The abstraction of domain knowledge, problem representation, and all activities that guide the development of domain applications is done by shifting the main effort from the technical to the domain level, allowing stakeholders and domain experts to become more involved in the development process without worrying about the technical and IT aspects. Hence, MDE has made it easier for stakeholders to represent domain problems and develop domain-specific applications [**22**] by encouraging the use of domains, shifting the focus from the technical level to the domain level by using models instead of source code and automating the generation of domain applications [**102**].

2.1.1.1. Domain-specific language.

A General-Purpose Language (GPL) is a computer language applicable to a large set of problems and across domains as it allows a fine-grained development and specification of applications.

In contrast to a GPL, a Domain-Specific Language (DSL) is a modeling language related to a particular application domain. A domain refers either to an application context (e.g. a web application, database, etc.) or to a business sector (e.g., banking, healthcare, manufacturing, etc.). A DSL is specialized to a particular application domain. Nevertheless, it can facilitate, abstract, and automate the generation of domain applications. MDE strongly fosters the use of DSL to create high-quality domain applications quickly and easily. The use of a DSL abstracts the technical and computational details and complexity such as the programming language, memory management, pointers, threads, data structures, etc. Also, it involves more stakeholders in the creation of domain-specific applications. As an example, HTML is a domain-specific language dedicated to web pages, CSS is a DSL that describes the presentation of an HTML document, and Latex is also a DSL used for document preparation. A DSL can be textual or graphical. For instance, SQL is a textual DSL that is used to manage the data contained in a relational database (defining how to create, modify, read, or delete data from a relational database). Automate[1] is an example of a graphical DSL that allows Android smartphone users to automate various tasks by building flowcharts (i.e. to change settings such as Wi-Fi, Bluetooth, etc. or to perform actions such as sending emails or SMS messages, etc. based on other event triggers such as location, time, etc.).

Listing 2.1 provides an example of an SQL query and figure 2.1 shows a flowchart created with Automate.

```
SELECT *
FROM Students
WHERE grade>80
```

Listing 2.1. Example of SQL query

A DSL has many fruitful outcomes such as having a little language with minimal maintenance effort [**39**]. Empirically, the use of a DSL increases performance, flexibility, reliability, usability, maintainability, and productivity [**4, 79, 14**]. Besides, using a DSL remarkably reduces costs and speeds up the software development process, which is attracting interest in domain-specific languages [**75, 33, 82**].

As illustrated in figure 2.2, a modeling language is composed mainly of 3 parts:

- **Syntax**: It is composed of two parts:

[1] `https://llamalab.com/automate/doc/index.html`

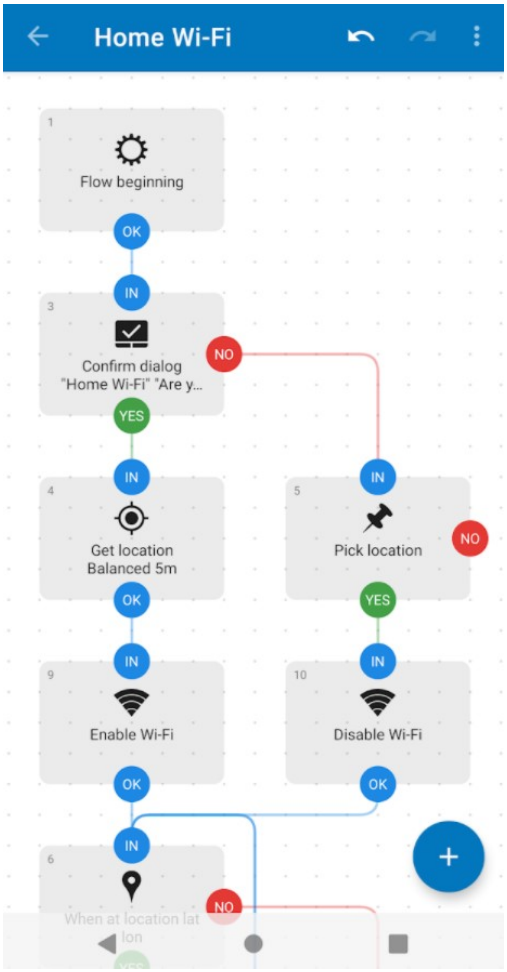


Fig. 2.1. An example of flowchart created with Automate
source: `https://llamalab.com/automate/`

  - *Abstract Syntax:* defines the actual syntax of the language that represents the underlying information and all the main concepts needed to build the language, e.g. the concepts, the elements to be used in the language, and the relationship between them.
  - *Concrete Syntax:* defines what the user actually sees. It maps the elements and relationships of the abstract syntax to the concrete notations the user should use. The concrete syntax can be either graphical (i.e. shapes) or textual (i.e. keywords).
- **Semantics:** defines the meaning of the modeling language. It represents the semantic domain.
- **Pragmatics:** defines recommended guidelines and practices, patterns and anti-patterns for the proper use of a DSML.

The syntax mapping plays a double role: a rendering engine (concrete to abstract syntax) and an analyzer (abstract to concrete syntax). The syntactic mapping function assigns each

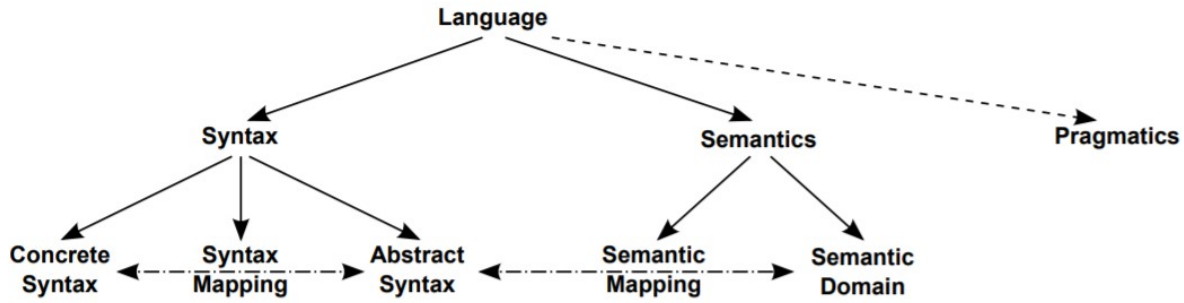


Fig. 2.2. Components of a modeling language
source: Syriani et al. [**125**]

element of the abstract syntax to concrete notations (textual or graphical). The semantic mapping function associates each element of the abstract syntax to elements of the semantic domain [**125**]. The abstract syntax is defined by the metamodel and the static semantics. We discuss the concept of meta-modeling in the next section.

2.1.1.2. Metamodeling.

In MDE, metamodels are the means to specify the abstract syntax of modeling languages [**24**]. Metamodels delineate the permitted structure to which we must adhere to build a valid model [**127**].

A Meta-model represents the structure of valid models by defining a tiny core of concepts, elements, and relations in a particular domain. Therefore, it restrains the valid list of models that form part of a DSML. Valid models are instances that conform to the meta-model specification. Thus, a meta-model describes with a higher level of abstraction the list of valid models.

The Object Management Group (OMG) has defined a standard for Model-Driven Architecture (MDA) that consists of four levels of abstractions called: Meta-Object Facility (MOF)[2]. As revealed by figure 2.3, MOF is composed of four multi-layers.

The metamodel layer at the top level is called the M3 layer. MOF is an M3-linguistic model. It defines a language to specify metamodels belonging to the M2 layer such as UML. The latter provides a language for specifying M1-models (i.e. the model of a problem domain) which, in turn, describes a real-world system (M0 layer).

[2]https://www.omg.org/mof/

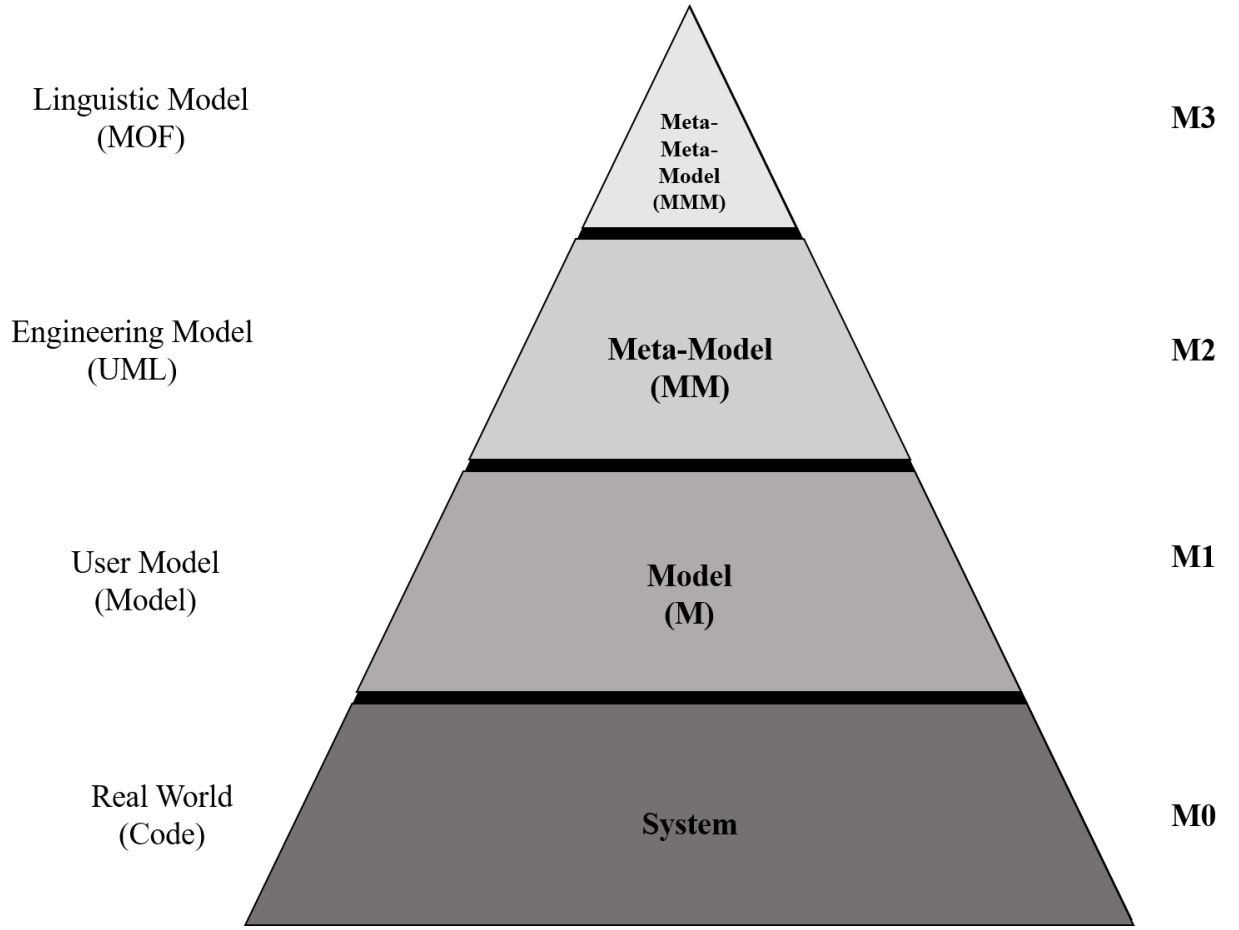


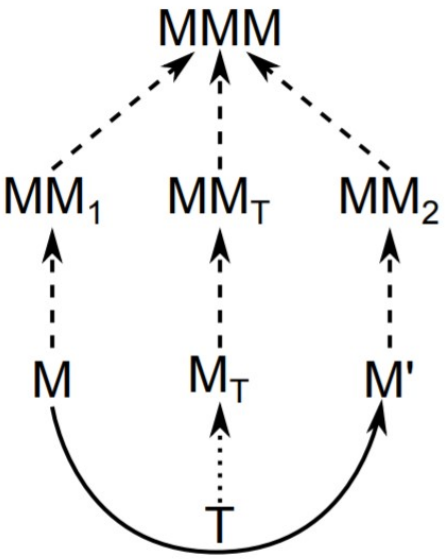
Fig. 2.3. The MDA meta-layers

2.1.1.3. Model transformation.

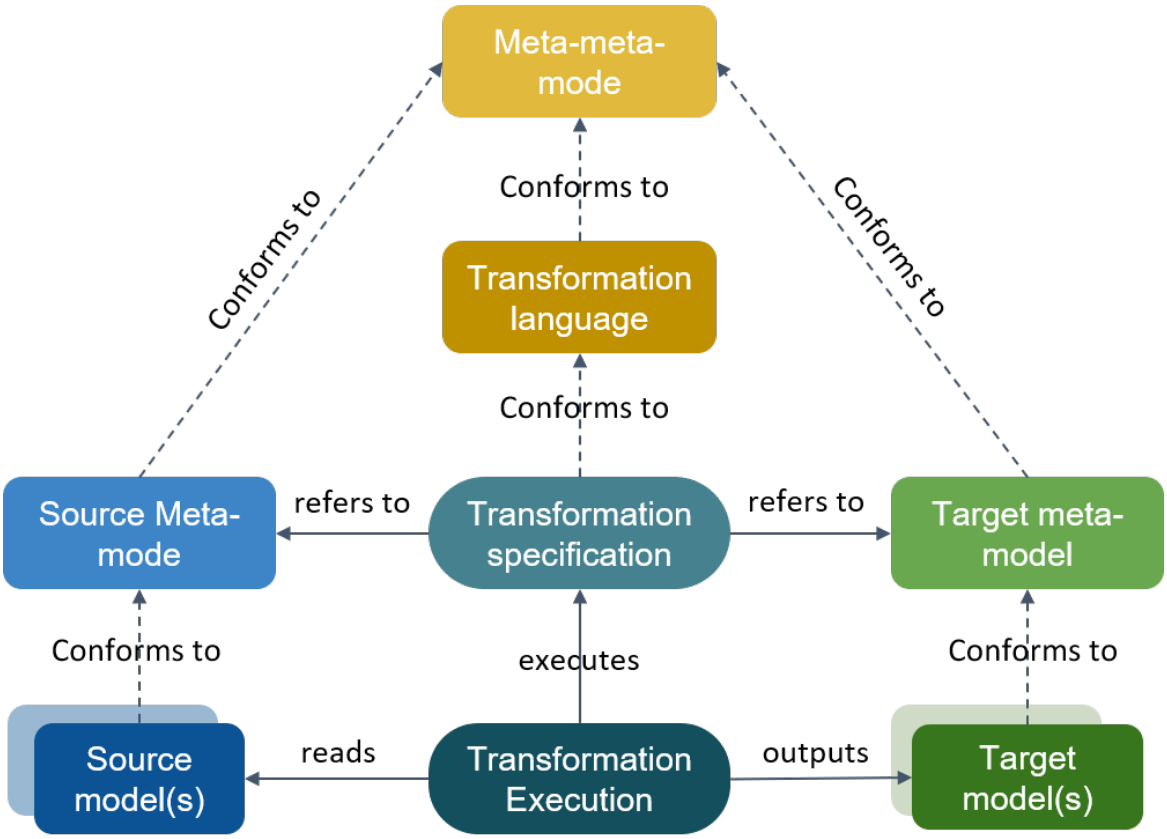
Model transformation is considered a key and fundamental concept in MDE. In [**119**], Sendall et al. described model transformations as the heart and soul of MDE because of the great role it plays.

Model Transformations are an automated way of creating and modifying models and are used for many purposes: model refinement, synthesis such as code generation, simulation, translation, model refactoring, etc. [**87**].

Figure 2.4 shows that a model transformation takes a source model as input and executes a transformation to generate a target model. Both models conform to their respective metamodels. The model transformation is defined at the metamodel level and is executed at the model level. There are two types of model transformations: an endogenous transformation if the source and the target have the same metamodel and an exogenous transformation if the source metamodel is different from the target metamodel.



((a)) Meta-modeling of model transformations



((b)) Terminology of model transformation

Fig. 2.4. Model transformation
source: Syriani et al. [**125**]

As illustrated in figure 2.4, a model transformation is defined at the metamodel level to be executed automatically on any input model that conforms to the source metamodel. We notice here that a model transformation can have several source/target models.

2.1.1.4. Metamodel quality assurance.

In MDE, metamodels must have a good, precise, and well-structured design that holds up and lasts against a dynamic and changing environment.

Metamodels are considered a central asset and an essential foundation in the field of MDE as they allow the analysis and formal modeling of a domain to then generate domain-specific languages, perform transformations that can be used for multiple purposes such as code generation, simulation, refinement, etc. Obviously, in MDE, building a metamodel is the outset for any application. Thus, this step should be performed attentively and accurately since it is the foundation for any MDE application, and must therefore be very accurate and of good quality.

2.1.1.5. Metamodel quality evaluation.

Metamodels are a fundamental pillar of MDE, as they are the foundation of any project. Let's take as an example, the construction of a domain-specific language that requires abstract and concrete syntax and semantics to define this language. The definition of an abstract syntax is done through the construction of a metamodel. Then, this metamodel can be useful for many applications such as model-to-model transformations or code generation. Regarding this crucial impact and the significant importance of metamodels, we should devote the required effort and time to build accurate and efficient metamodels in order to reap the benefits later on. This is one of the fundamental concepts and aspects of MDE, which costs in terms of resources at the beginning, but then becomes much more profitable than the classical software development process. The quality of the metamodel could be evaluated using different measures. Their composition can lead to the development of more abstract measures that are not intuitive and easy to calculate but are easier to interpret and understand. Metamodel quality assessment allows the modeler to evaluate the metamodel based on defined measures and to compare it with an advanced version (e.g. refactored version).

According to [**11**], model quality assurance processes are typically based on three main steps: model analysis, identification of model smells and resolution of model smells.

Mohagheghi et al. [**100**] reviewed the literature on model quality and identified six classes of model quality goals (6c model): *correctness, completeness, consistency, comprehensibility, containment, changeability, goals*. This quality model [**100**] means that other quality goals introduced in the literature can be met if the 6C goals are in place.

2.1.1.6. Metamodel smells.

Metamodel smells are indicators of potential problems with the metamodel [**58**]. The introduction of bad smells in metamodels can affect its quality, which can have a negative impact in the long term.

Bad smells are a symptom of a design or system code problem. Many bad smells are defined in the literature, but detecting them is a challenging task that is far from being trivial [**46**].

To resolve a bad smell, we can apply one or a set of refactorings and thus improve the metamodel quality.

An example of bad smells in metamodels could be a dead metaclass, i.e. a metaclass that is completely disconnected from the rest of the metamodel. This bad smell is similar to the dead code [**37**] which is a common code smell that is easy to correct (by removing it). It is a code that is never called or reached. As reported by Strittmatter, Misha, et al in [**124**], this bad smell harms the comprehensibility of the metamodel.

Bad smells may have a negative impact on the artifacts defined within the metamodel, including models, transformations, etc. Moreover, refactoring the metamodel to remove these smells does not come without a cost, as explained in [**21**], because the evolution of the metamodel will invalidate the artifact based on it. Here, arises the term co-evolution of different artifacts. This problem has been tackled by numerous researches, even based on the detection of differences between versions of the metamodel, as is the case for many approaches(e.g. [**60, 63, 97, 98, 130**]), or as formulated by [**76**] where the problem of metamodel/model co-evolution was tackled as an optimization process.

2.1.1.7. Metamodel refactoring.

Metamodel bad smells may be a trigger for refactorings. Refactorings are applied to resolve these bad smells and thus improve the metamodel quality. Refactorings have been created in the field of software development to improve the quality of the code to make it more reusable, more understandable, easier to maintain, etc. It has been defined by Fowler [**47**] as the process of improving the design structure of software while preserving its overall behavior and functionality. And then, this concept of refactoring as well as for bad smells has been elevated to the field of modeling. An example of refactoring could be *extract super class* which allows factorizing common features between multiple meta-classes into a super-class they inherit. In analogy to code refactoring, metamodel refactoring is defined as the process of restructuring an existing metamodel while preserving its observable behavior [**113**]. The process of metamodel refactoring can lead to the invalidity of the resulting models and this problem is being addressed by research on the co-evolution of different artifacts. Hence, we

would be able to co-evolve metamodels, models, transformations that conform to them, and the different artifacts that depend on them.

Thus, the MDE field is inspired by research done in the field of software engineering by trying to adapt and evolve these approaches. The general and global concepts of quality, bad smells, and rework are the same but require some adaptation.

There has been much work done on various techniques and tools for refactoring either for software engineering [**47, 106, 101, 103**] or MDE [**21, 114, 124, 10, 48**]. According to Alizadeh et al. [**5**], these approaches could be classified into three main categories: *manual, semi-automated and fully-automated* approaches. For manual refactoring, the modeler refactors without any tool support [**5**], for semi-automated approaches the modeler uses a tool to perform the refactorings he deems necessary with the ability to make some customization and modifications on the proposed solutions and to take his preferences into account. By automated refactoring approach and by analogy to its definition in [**5**], this means that the refactoring process is fully automated and that there is no interaction with the modeler. It is important to keep in mind that the application of refactoring can improve quality on the one hand but deteriorate it on the other. For example, the application of refactoring to improve the maintainability and reusability of a metamodel can affect its complexity and comprehensibility if, for example, many new metaclasses are introduced.

In the following sections, we provide an overview of the Evolutionary Multi-Objective Optimization (EMO) paradigm and focus on the Multi-objective Genetic Algorithm (MOGA), in particular, the Non-dominated Sorting Genetic Algorithm II (NSGA-II).

2.1.2. Artificial intelligence

Artificial Intelligence (AI) is an interdisciplinary science which is a sub-field of computer science. The aim is to build intelligent machines capable of performing tasks that generally require human intelligence. The term AI is often used to describe machines that imitate human intelligence and their ability to solve problems.

AI is an interdisciplinary science with multiple approaches, but advances in machine learning and deep learning are creating a paradigm shift in virtually every sector of the technology industry.

Among the most used techniques in AI are: Machine Learning, Evolutionary Computing, Fuzzy logic.

- Unlike Boolean Logic which is based on two truth values (0 and 1, or, FALSE and TRUE), ***Fuzzy logic*** is a many-valued logic concept where variables have many degrees of truth. A degree of truth is a real number between 0 and 1 both inclusive [**105**]. This approach is used to model partial truth. From a probabilistic point of view, it represents the degree or probability of a statement to be true.

Many AI techniques rely on Fuzzy Logic, i.e. Fuzzy clustering, which is a soft clustering technique where each element has a probability of belonging to each cluster [**131**], as opposed to classical clustering where each element belongs to a single cluster.

- ***Evolutionary computation*** is a sub-field of AI. It is a family of biologically inspired meta-heuristics that are used for optimization purposes. These problem-solvers are based on a stochastic process (i.e. trial and error). Genetic algorithms are the most eminent techniques among Evolutionary algorithms (See section 2.1.2.5) [**44, 43**].
- *Machine Learning* is the most prominent and prosperous sub-field of AI. In the next section, we discuss its definition, concepts, and techniques in more detail.

2.1.2.1. Machine learning.

Machine learning is a subset of Artificial Intelligence (AI) that provides systems with the ability to auto-improve themselves through experience [**53, 99**]. In other terms, it is a specific application of AI that gives systems the ability to automatically learn and improve from experience without being explicitly programmed [**50**].

The learning process consists of automatically learning patterns from data and making better decisions from experiences without human guidance or intervention. Machine learning algorithms can adjust their actions, decisions, and predictions based on historical data to improve performance and deliver accurate results.

Machine learning is widely used in a variety of applications such as computer vision, prediction, recommendation, self-driving cars, speech recognition, machine translation for natural language processing to translate text from one language to another, etc. It is used to solve challenging and complex problems that are difficult or impossible to solve using classical algorithms and techniques.

As reflected in figure 2.5, machine learning is categorized into three main sub-types: Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

(1) ***Supervised Learning:*** It is a subset of Machine Learning algorithms in which the data set used is labeled. In other words, we have the input data and the desired output. The goal is to learn from this data to generate the output according to the input. This means the output variable is known for old examples and we use other input features to learn and predict the value of this target variable for new observations. In supervised learning, we apply the algorithm repeatedly to a data set and compare the correct value with the predicted one to estimate the error, and then refine our model accordingly to improve its performance. Here are some of the most common techniques used in supervised learning include:

- Classification: It consists of predicting a class label for each input element. Data labels are discrete. For instance, spam filtering is a classification problem where

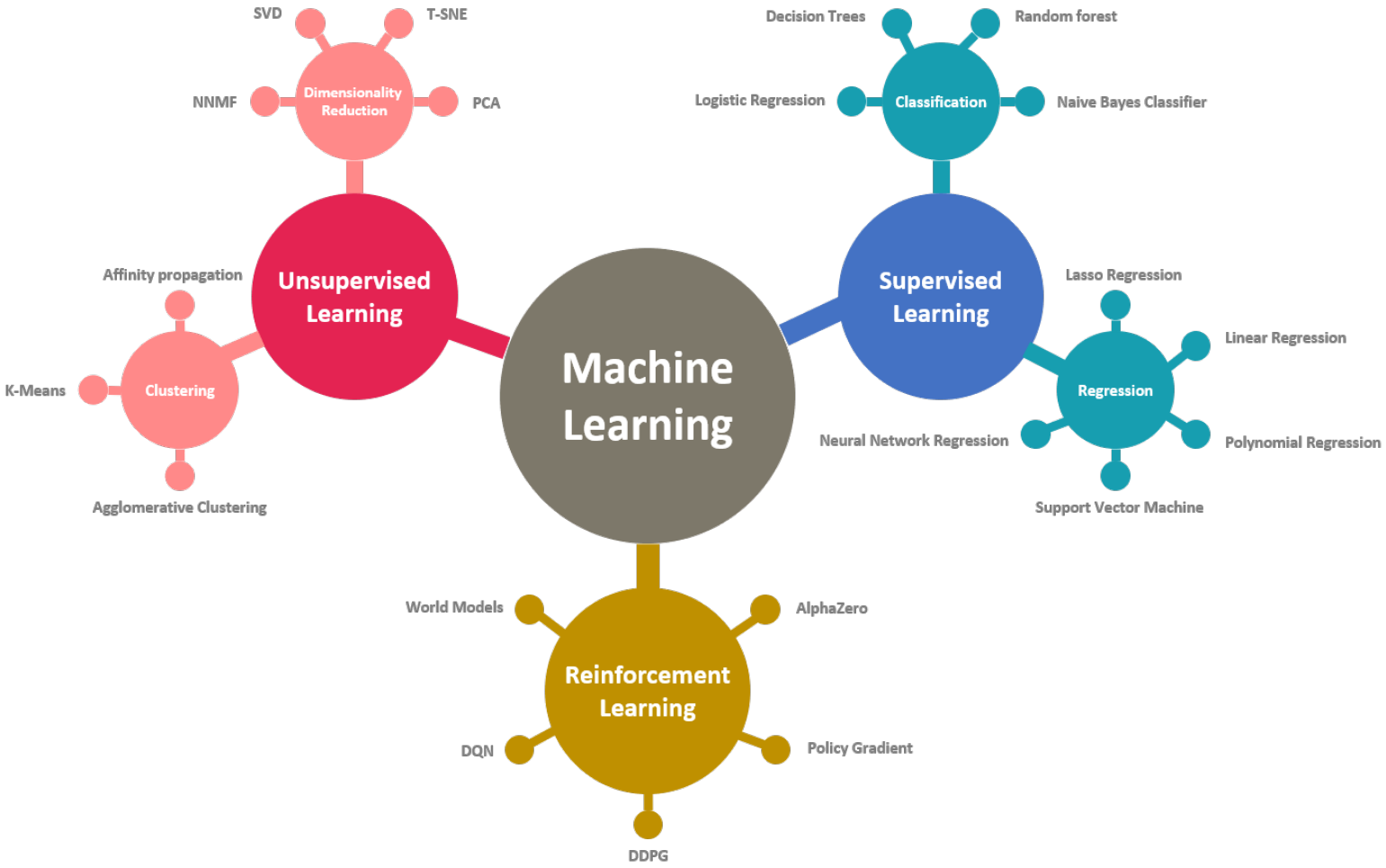


Fig. 2.5. Types of machine learning

spam and not-spam are the two classes or labels. The goal is to learn a function that associates the input email with one or the other of the classes. Figure 2.5 shows some of the best-known classification algorithms, e.g. Decision Trees, Random Forest, Logistic Regression, etc.

- Regression: it is similar to classification with the difference the data labels are continuous in regression problems. A regression model predicts a continuous quantity output given an input. For instance, estimating the price of houses given their features (e.g. area, number of rooms, location, number of floors, etc.) is a regression problem. Linear Regression, Polynomial Regression, Support Vector Machine and Artificial Neural Networks are among the most commonly used algorithms in regression problems.

(2) ***Unsupervised Learning:*** In contrast to Supervised Learning, Unsupervised Learning is another type of Machine learning where the used data set is not labeled. Unsupervised Learning algorithms aim to automatically infer hidden structures from

unlabeled data. Otherwise stated, we have just the input data and we don't know in advance the expected output or labels. The Unsupervised learning algorithm explores the input data and tries to figure out hidden patterns and structures in this unlabeled data. Some of the most common algorithms used in unsupervised learning include:

- Clustering: The process of splitting the data set into groups to maximize similarities inside the same group and maximize dissimilarities between groups. For example, we could use clustering to segment clients based on many characteristics (e.g. purchase history, interests, age, etc.), which would help the company target specific customers' groups for specific campaigns. Among the most common clustering techniques, we mention K-means, Agglomerative Clustering, Affinity Propagation, etc.
- Dimensionality Reduction: It aims to reduce the dimensional space of data. It transforms data from a high-dimensional space to a lower-dimensional representation while retaining the maximum amount of information and properties of the original data. This technique can be used for visualization purposes or to automatically reduce the number of features before the model training in order to eliminate correlated features that could be expressed in terms of other features. Principal Component Analysis (PCA), Non-negative Matrix Factorization (NNMF), and Singular Value Decomposition are considered among the most commonly used techniques for dimensionality reduction.

(3) ***Semi-supervised Learning:*** A hybrid category that mixes both Supervised and Unsupervised Learning is called Semi-Supervised learning. A Semi-supervised Learning algorithm relies on the use of a large amount of unlabeled data and a small amount of labeled data. It typically uses the labeled data to partially train a ML model and then uses this model to label the unlabeled data: this process is called **pseudo-labeling**. Once all the data labeled, we train the model on the whole resulting labeled data set. These techniques are used to deal with the lack of huge labeled data sets. It is quite difficult to always find large amounts of labeled data, sometimes it is costly and challenging to adopt a fully labeling process of data. Semi-Supervised Learning is useful if we are faced with these kinds of problems.

(4) ***Reinforcement Learning:*** Reinforcement learning is another sub-category of machine learning that is often used for robotics, text mining, gaming, navigation, and recommendation systems, etc. A Reinforcement Learning model interacts with the environment by taking actions and discovering errors or rewards. This discovery is made through trial and error while trying to maximize rewards and minimize errors [**73**]. As shown in figure 2.6, a reinforcement learning system is characterized by two primary components. The agent which is the decision maker or learner, the

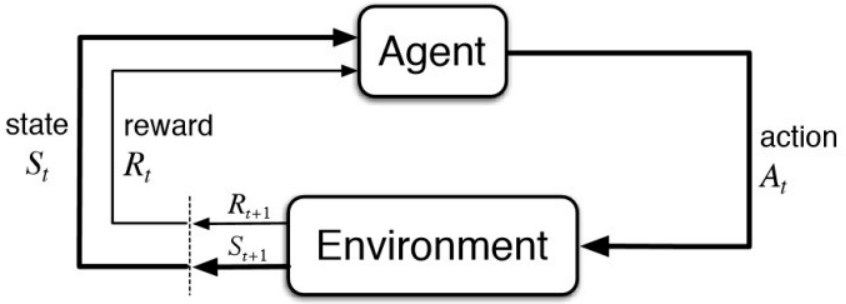


Fig. 2.6. Reinforcement learning

environment that contains everything the agent interacts with [**73**]. The learning process is agent-centered. The agent has a set of actions it can perform. In each state, the agent performs an action A_i in the environment to move to another target state. According to the selected action, the agent gets a reward. The goal of the learning phase is to train the agent on selecting a good policy to reach its goal of choosing the right actions to maximize the cumulative rewards.

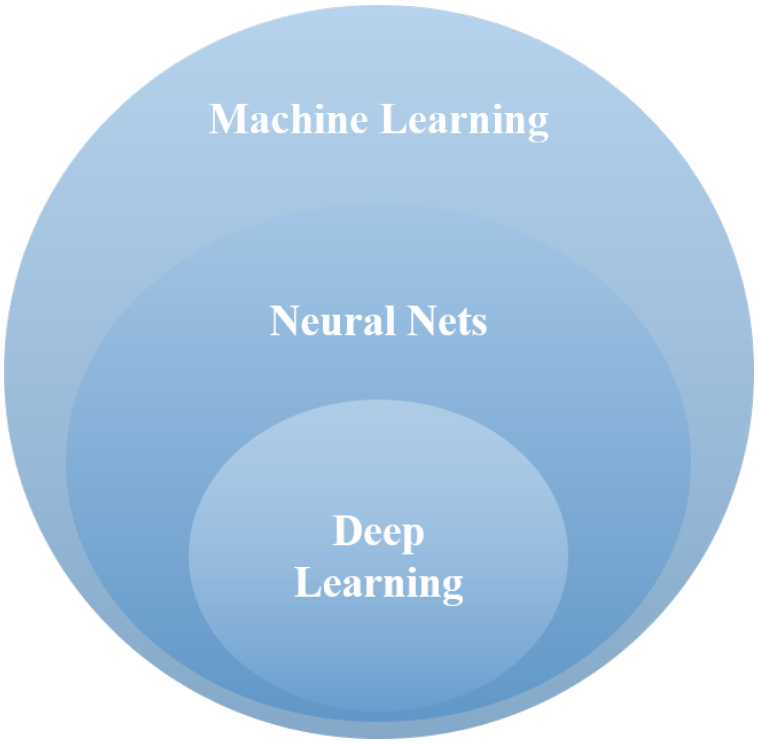


Fig. 2.7. Machine learning taxonomy

Another perspective on the categorization of machine learning is illustrated in figure 2.7. We detail it in the next sections.

e

2.1.2.2. Artificial neural networks.

Artificial Neural Networks (ANNs) are a sub-set of Machine learning that is inspired by the biological brain [**64**]. As revealed by figure 2.8, an ANN is composed of a set of successive and connected layers.

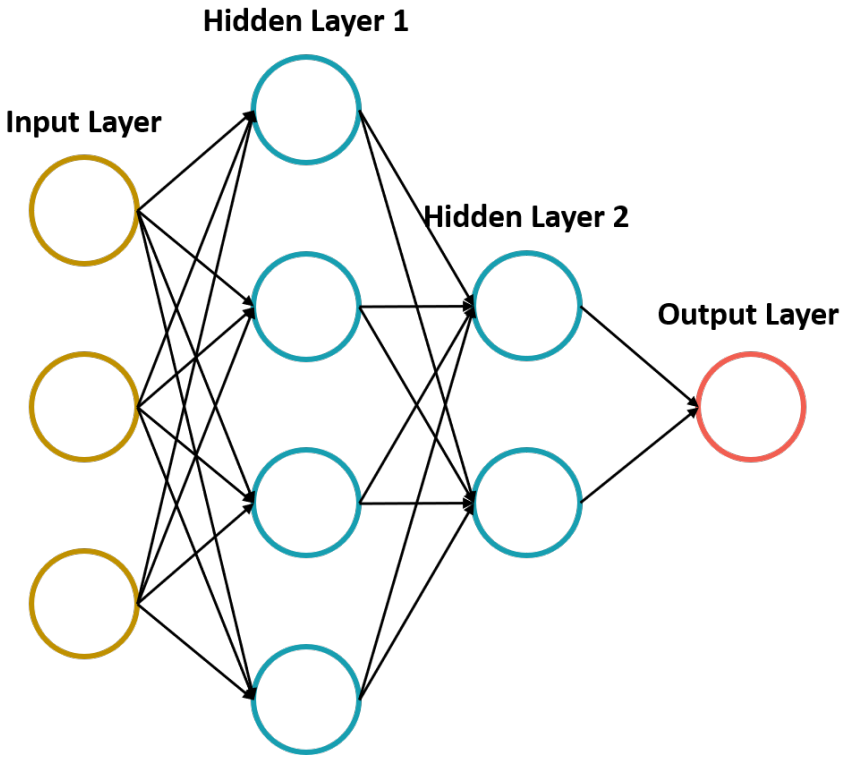


Fig. 2.8. Structure of artificial neural networks

Each layer consists of a collection of connected units or nodes called neurons. Each neuron receives an input signal and processes it to produce an output signal.

An ANN consists of an input layer that represents our input data, a set of hidden layers, and an output layer that contains our desired value. The layers are interconnected and each layer transforms an input signal into an output signal readable by the next layer.

Each neural has some weights that represent the contribution of each input feature to the output value. The node applies an activation function on the output value. Activation functions allow to introduce a non-linearity behavior to detect complex patterns. For instance, Sigmoid (Figure 2.9) is an activation function that has an output between 0 and 1. It is often used in binary classification.

With the arrival of back-propagation [**28**], ANNs were able to adjust their hidden layers so as to optimize an objective function [**85**]. Since then, ANNs have spread and evolved, giving rise to deep learning.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Fig. 2.9. Sigmoid activation function

The ANN defines a cost function that allows to measure the performance of the network on the training data. This cost function drives the learning process. With back-propagation, the weights are updated according to a learning rate to improve the defined cost function. The learning rate controls the learning process by defining the update amount of the weights.

ANNs were a revolution in the domain of AI and they led to the appearance of Deep Learning.

2.1.2.3. Deep learning.

Deep Learning is Machine Learning based on ANNs. It is a more advanced technique compared to classical Machine learning algorithms. It is a technique based on deep ANNs that is formed of multiple hidden layers (typically more than two) and tries to mimic the brain's functioning in order to learn precise patterns and make accurate decisions [**117**]. Data, such as images, videos, text, etc., are transmitted to the deep ANN to automatically learn the hidden patterns and structures and then perform accurate predictions [**32**]. Deep Learning has shown powerful capabilities and has notably surpassed classical Machine Learning techniques [**17**] that allowed machines to solve complex problems even when using a very diverse, unstructured, and interconnected set of data.

2.1.2.4. Multi-objective optimization.

The process of systematically and simultaneously optimizing a set of objective functions is called multi-objective optimization (MOO) or vector optimization [**93**]. It refers to any optimization problem that involves more than one conflicting objective than what needs to be optimized simultaneously. Therefore, there is not a single solution that optimizes all the objectives because of their divergence which prevents the simultaneous optimization of the objectives. On the contrary, we can have a set of Pareto-optimal solutions (the number can tend towards infinity).

A solution is called *not-dominated* or *Pareto to optimal* if none of the objective functions can be improved in value without degrading some of the other objective values. All non-dominated solutions are considered equally good and we cannot order them completely unless we use additional information about subjective preferences.

Therefore, the goal of multi-objective optimization problems is to find a representative set of Pareto to optimal solutions that constitute a trade-off between predefined objectives

and we then select the most appropriate solution according to the preferences of a human decision-maker [**45**].

In [**5**], it is assumed that the goal of a preference-based EMO algorithm is to assign different importance levels to the problem's objectives to guide the search towards the Region of Interest that is the portion of the Pareto Front that best matches the user preferences because, usually, the user is not interested with the whole Pareto front and thus he/she is searching only for his/her ROI from which the problem's final solution will be selected. In [**5**], it is assumed that the goal of a preference-based EMO algorithm is to assign different levels of importance to the objectives of the problem. The goal is to guide the search to the region of interest that is the portion of the Pareto Front best matching the user's preferences. This is because the user is usually not interested in the whole Pareto Front and therefore only looks for its return on investment from which the final solution of the problem will be selected. This is an a priori method [**93**] where the decision maker's preferences are taken into account before proceeding with the optimization, and then the Pareto to optimal solution that satisfies these preferences is produced. Among these, one used method [**69**] consists of scalarizing a multi-objective optimization problem, by transforming it into a single-objective optimization problem such that optimal solutions to the single-objective optimization problem are Pareto optimal solutions to the multi-objective optimization problem. In a posteriori methods [**93**], the Pareto front constituted by the non-dominated solutions is first found and then the decision-maker can choose the solution that best fits his preferences.

Evolutionary algorithms, such as Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [**36**], are considered as common and standard approaches to solve multi-objective optimization problems.

2.1.2.5. Genetic algorithms.

Genetic algorithms are metaheuristics algorithms that are particularly well-suited for this class of problems.

The concept of GA was developed by Holland et al. in the 1960s and 1970s [**116**]. Genetic algorithms are biology-inspired.

In nature, weak species that are unsuited to their environment are faced with extinction by natural selection. However, stronger species are more likely to pass on their genes to future generations through reproduction. In the long term, species carrying the right combination of genes become dominant in their population [**81**]. In genetic algorithms (GA) terminology, several new specific terms are defined [**81**]:

- *Chromosome/Individual:* In GA, a solution is called chromosome or individual. A chromosome corresponds to a unique solution.
- *Gene:* Chromosomes are composed of a set of genes. i.e. A solution vector is called a chromosome and each element in that vector is called a gene.

- *Population:* GA works on a set of chromosomes called population. At first, it is randomly initialized and then it evolves through the steps of the GA to include fitter solutions.

To apply GA to a problem, we should formulate and adapt it by encoding it to make it work.

To generate new solutions, GA use two operators which are *crossover* and *mutation* that are defined as follows:

- *Crossover* [**81**]*:* It consists of combining two chromosomes, called *parents*, to form new chromosomes, called *offspring*. The parents are selected from the existing population with a preference for fitness, so the offspring are expected to inherit good genes that make them more fit.

  By applying the crossover operator iteratively, the genes of good chromosomes are expected to appear more frequently in the population, implying more adapted child chromosomes, which will eventually lead to convergence towards a good overall solution.

  There are different types of crossover. Among them, we cite, *one-point crossing* and *two-point crossing*. The single-point crossover consists of picking up a random position and combining the left part of the first parent and the right part of the second parent (relative to the selected point) to form the new offspring. A two-point crossover consists of picking-up two positions randomly and forming a new child chromosome composed of the left and right sides of the first parent and the central part of the second parent. These two types of crossover are well-depicted in figure 2.10 and figure 2.11.

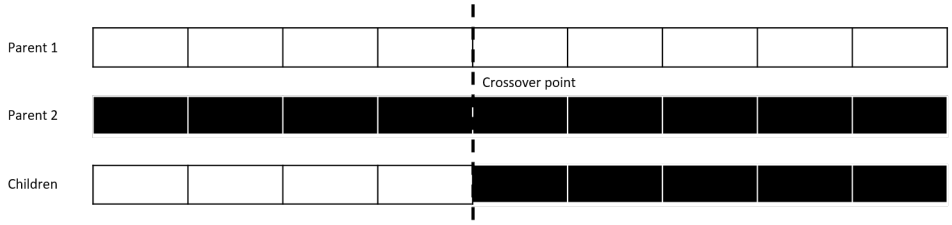


Fig. 2.10. Single-point crossover example

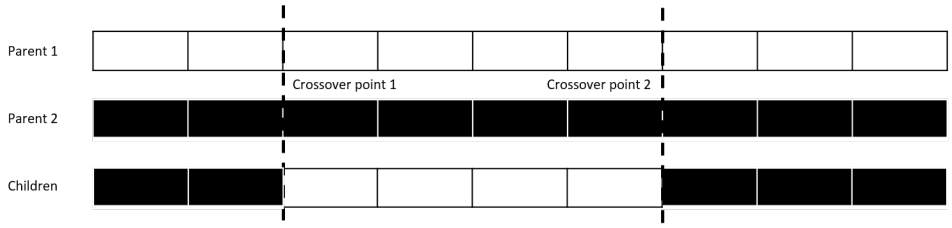


Fig. 2.11. Two-point crossover example

- *Mutation* [**81**]*:* It consists of introducing random changes in a single chromosome. It is generally applied at the gene level. Each gene has a probability of being altered. The new chromosome produced by the mutation is not very different from the original one. Figure 2.12 shows an application example of the mutation operator. The mutation plays a critical role in GA. Unlike crossover which leads the population to converge by making the chromosomes of the population alike, mutation reintroduces genetic diversity back to the population and assists the search escape from local optima.

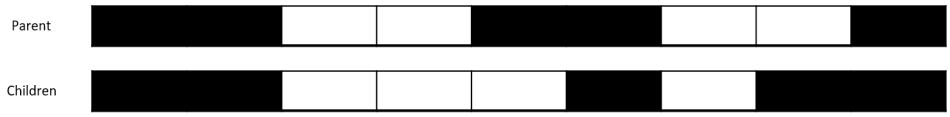


Fig. 2.12. Mutation example

In addition to the genetic operators, population reproduction also involves the selection of chromosomes for the next generation, which is determined according to the aptitude of each individual. This selection differs from one genetic algorithm to another as well as for tournament selection (consists of selecting two individuals for a crossover) and for ranking (ranking the population into several fronts where the first front is composed of the Pareto-optimal solutions, the second front contains the individuals that are not dominated by any other solution than the first front, etc.). A GA has a set of hyperparameters (e.g. population size, mutation rate, number of iterations, etc.) that are chosen manually before the execution. The results depend on the values of hyperparameters that are often chosen based on experience, using a trial and error approach, or using a search technique to find the optimal set of hyperparameters that lead to the best solutions.

Listing 2.2 describes the generic steps of GA.

2.1.2.6. Non-dominated Sorting Genetic Algorithm II.

NSGA-II is a Multi-objective Optimization Evolutionary Algorithm (MOEA) that is computationally fast and elitist. NSGA-II is based on a nondominated sorting approach [**36**]. This genetic algorithm is based on sorting the population on multiple fronts using a non-dominated rapid sorting approach. It also uses crowding distance to sort individuals belonging to the same front to ensure diversity. As illustrated in figure 2.13, the crowding distance value is defined as the perimeter of the cuboid formed by the nearest neighbors in the objective space.

NSGA-II asserts diversity and selects diversified solutions (in the last front) using the crowding distance. If we select members with high crowding distance, we ensure diversity since we eliminate the crowded points and favor distant individuals to cover a large space [**36**]. This is basically what characterizes the NSGA-II algorithm that is used in our approach to find a compromise between our conflicting objectives.

```
1/ Initialization
t = 1
Generate N individuals to form the initial population, P_1.
Evaluate the fitness of solutions (value of objectives) in P_1.

2/ Crossover
Generate an offspring population Qt using crossover operator by repeating
   ↪ these two steps:
    - Tournament selection: choose two solutions x and y from P_t based on
   ↪ the fitness values.
    - Use a crossover operator to generate offspring and add them to Q_t.

3/ Mutation
Mutate each solution x ∈ Q_t with a predefined mutation probability.

4/ Fitness assignment
Evaluate each solution x ∈ Q_t by assigning a fitness value (computing
   ↪ objectives values for each solution).

5/ Selection
Select N solutions from Q_t based on their fitness values to form P_{t+1}

6/ Termination
If (the termination criterion is satisfied)
then
    Stop the algorithm and return the current population
else,
    t = t+1
    go to  2/
```

Listing 2.2. Genetic algorithm structure

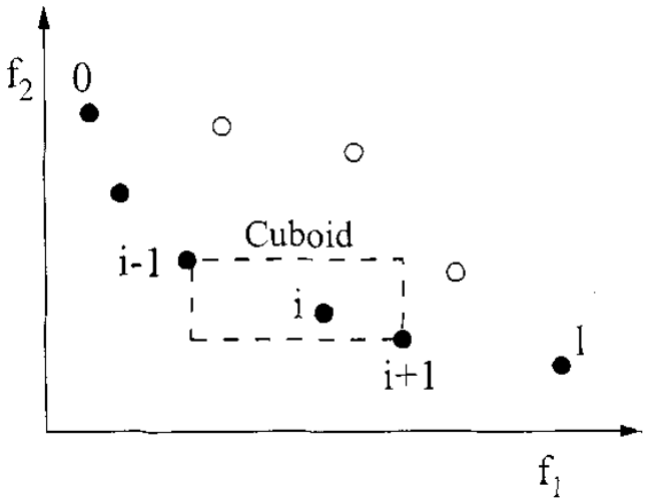


Fig. 2.13. Crowding distance calculation
source: Deb et al. [**36**]

2.1.2.7. Constraint solving.

Constraint solving is a sub-field of Artificial Intelligence that consists on declaratively state constraints and use solvers to satisfy them. In constraint solving, we do not need to specify the required steps to find a solution, however, we need to declare constraints and properties of the solution. We use constraint solvers such as Alloy, Z3 or Prolog to look for the solution that satisfies these constraints.

A constraint is a relation between multiple variables. It restricts the number of possible values that it can take. Searching for a solution that satisfies a set of provided constraints is called Constraint Satisfaction Problem (CSP). If the constraint solver does not find a solution for a CSP, then, it is unsatisfiable. This means, there is no solution that satisfies all the constraints. CSP on finite domains are solved using some search techniques such as constraint propagation, backtracking and local search. Depending on the number of clauses and variables, the CSP resolution could be very complex.

A technique that has been found very useful for verifying properties on a system is model-checking. It consists of modeling the system as a structure, formulating the properties to be checked as constraints, then, determining whether a constraint is true in the structure, otherwise, it generates a counter-example. Composing linear number of structures could yield to an exponential growth of the system, therefore the verification of certain properties may become impossible for big structures. Some constraint solvers overcome this problem using bounded verification. The solver performs a bounded scope analysis by checking the encoded specification over a finite number of instances.

For instance, Alloy is a formal constraint specification language based on first-order logic. It allows to formally express the structural properties and behavior of a system. Alloy provides a lightweight modeling tool to express and check system properties [**71**]. It is designed to perform a bounded scope analysis by checking the encoded specification over a finite number of instances.

2.1.3. Scrum: an agile software development methodology

Scrum is a lightweight yet agile and powerful framework for productive software development. It involves the development, delivery, and sustainment of complex products. Moreover, it allows to smoothly manage a team and to promote effective collaboration on complex projects [**118**]. It is an iterative method for project management that provides a set of values, principles, and practices. This framework is characterized by the division of tasks into short phases of work called *Sprints* and by frequent and early deliverables called *Increments*. A Sprint is a short period where a scrum team works to complete a set of tasks called *Sprint Backlog* taken from the product backlog. An increment is a sprint deliverable.

Figure 2.14 illustrates the team roles, artifacts, events, and their interactions in scrum.

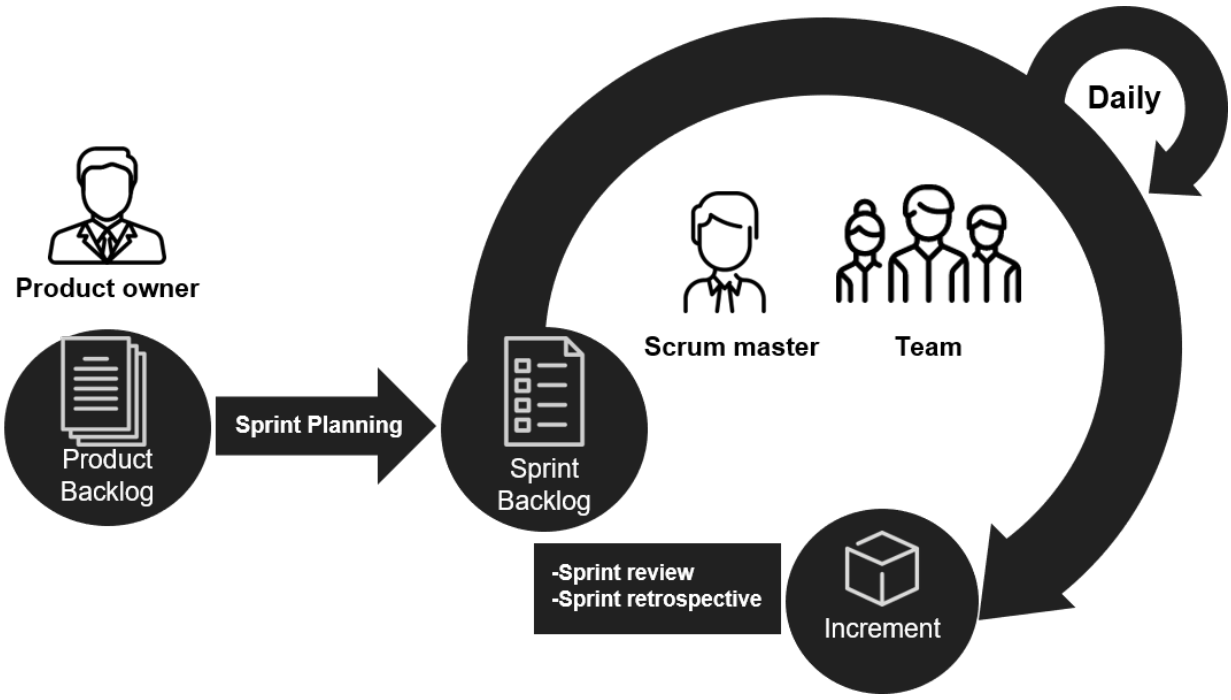


Fig. 2.14. Scrum: an agile framework for development

Scrum is characterized by the following artifacts:

- *Product Backlog:* It contains the exhaustive list of features, bug fixes, updates that a team is planning to deliver sooner or later.
- *Sprint backlog:* A set of tasks called user stories that should be done during the sprint.

Scrum is also characterized by these events:

- *Sprint Planning:* An event that occurs at the beginning of a sprint where the team agrees on the sprint backlog to be realized during that sprint. During this meeting, the user stories (tickets that contain title + description + story points) are affected by the different team members according to their capacity. Each member has some story points that he can accomplish during a sprint (e.g. $\frac{1}{2}$ day = 1 story point). Each user story has a score (X story points) that reflects its complexity (effort and time required to accomplish it).
- *Daily scrum meeting:* The development team meets for a short period (around 15 minutes) each sprint day to inspect the progress done the day before towards the sprint goal. Each team member describes his progress, what he's planning to do, and the problems he is facing, etc. This meeting is an opportunity to track the development team's progress daily and to have a global view of the progress of others.
- *Sprint review:* This meeting occurs at the end of the sprint and focuses on the product under development. The team evaluates and reviews the progress done during the last

sprint. During a sprint review, the scrum team presents and discusses the progress with the stakeholders. Therefore, the product backlog could be adapted based on feedback and discussion. The product owner has the opportunity to release one of the accomplished features.

- *Sprint retrospective:* During the sprint retrospective, the Scrum team gives feedback about the progress of the last sprint, highlights the issues, and suggests areas for improvement to promote the team's comfort, efficiency, and productivity.

Additionally, Scrum is characterized by the following roles:

- Product owner: He focuses on understanding the business, customer, and market requirements. Then he works on prioritizing the work to be done. Besides, the product owner ensures that the work is done in the right way and that it complies with the requirements and customer expectations.
- Scrum master: He leads and manages the scrum team. The scrum master helps the team to accomplish their tasks, collaborate, communicate, and to unblock them.
- Scrum development team: It's composed of members who are responsible for executing the work. The team could include scientists, developers, designers, etc. The scrum team members collaborate and contribute to the development of the product.

2.2. Related work

2.2.1. AI-based improvement for software-related tasks

2.2.1.1. Improvement opportunities.

Given the powerful capabilities and rise of AI, it has been used by almost every other field to leverage its potential to solve challenging and difficult problems. In the area of computer science, it is very important to have a clear picture of the opportunities for improvement in the AI-based software generation pipeline that could advance the field. Although this is a very important area of research, it has not been much addressed by the research community. Few works were either related or weakly linked to this area.

Breck et al. [**25**] provide a set of 28 specific tests and monitoring needs in Machine Learning systems based on their teams' experience. They conducted meetings with the different AI teams within Google and established a roadmap to improve the production readiness of Machine Learning systems, pay back the technical debt, and solve the quantified issues. Zinkevich et al. [**134**] presented some best practices to build solid and robust AI systems.

In [**7**], the authors conducted a study on Microsoft teams involved in the development of AI-based solutions. They have identified the challenges and issues that the engineers face

when building large-scale AI applications and suggested some guidance and best practices to tackle these challenges and build robust and high-quality large-scale AI solutions.

In [**133**], the authors discussed the evolution of machine learning applications in the field of software engineering by indicating the actual use of machine learning techniques to automate some software engineering tasks such as fault-proneness prediction, software quality classification, apply genetic algorithms in testing, automatic code generation, etc.

2.2.1.2. ML-based automation tools.

Menzies discussed in [**96**] some applications of machine learning in the software engineering pipeline such as predicting faults in software modules, automating the maintenance task, etc.

In [**19**], the authors present KNIME, a framework for information mining. It consists of a graphical workbench to create and execute data flows. The data pipeline is composed of a set of connected nodes where the first node is often responsible for reading data from sources such as files, databases, etc. Then, the other nodes could perform several actions such as modifications, transformations, visualization, building a ML model, etc. These actions could include the following operations: handle missing values, column or row filtering, sampling, normalization, partitioning into training and test data, etc. Then, some nodes could apply some ML algorithms on the input data such as Naive Bayes, Support Vector Machine(SVM), decision trees, etc. The final nodes are often responsible for inspecting the results by visualizing the output data or writing it to databases or files. The platform is extensible as it enables simple integration of new types of nodes such as new ML algorithms, data actions, visualization methods.

Figure 2.15 shows a small data pipeline created with KNIME workbench. The pipeline is composed of parallel data flows that are composed of preprocessing, visualization, and modeling nodes.

KNIME is a general-purpose tool that could be used for any domain. It was extended in [**72**] to be specifically applied for the DNA sequencing domain by integrating new functionality for next-generation sequencing analysis. Other types of nodes and reusable workflows were introduced that allow to easily perform next-generation sequencing analysis.

Similarly, Orange is a general-purpose machine learning and data mining tool [**38**]. The tool is suitable for all kinds of users: from novices to experts who prefer a scripting interface. Orange offers an interface for graphical programming to build ML pipelines that allow to preprocess data, apply ML models, visualize data, etc. The goal of Orange was to implement the most useful and commonly used techniques in a user-friendly, flexible, and extensible way.

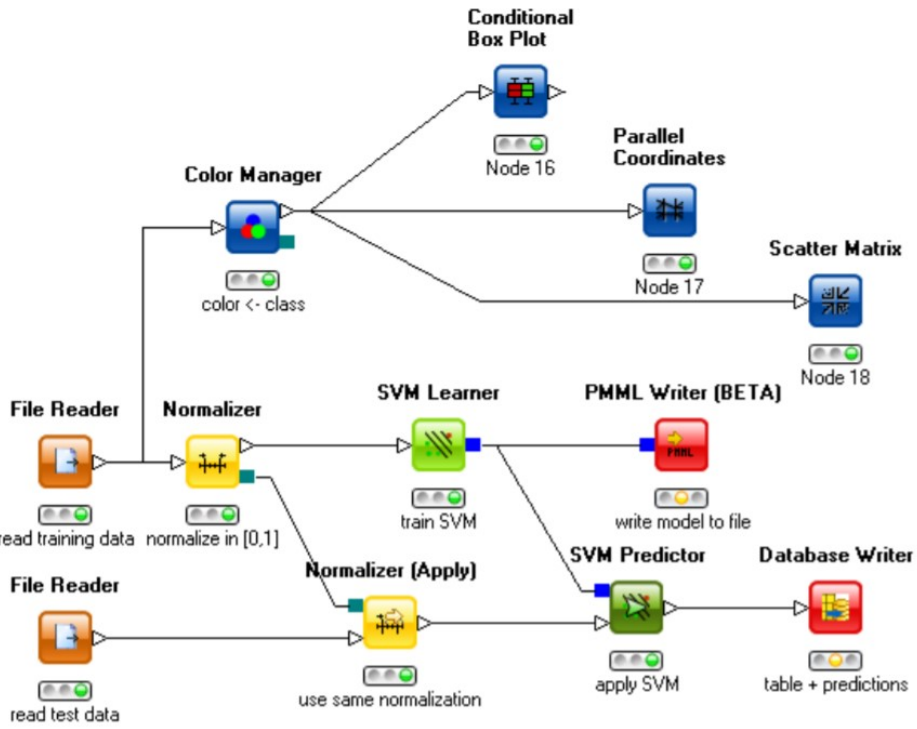


Fig. 2.15. Data flow example built with KNIME workbench
source: Berthold et al. [**19**]

In addition, RapidMiner [**65**] is an open-source tool that supports the machine learning process including data preparation, ML algorithms, visualization, etc. It provides a graphical user interface (GUI) called *RapidMiner Studio* to easily design and execute analytical workflows. RapidMiner provides a platform for developers to create and share algorithms with the community. This graphical workflow designer tool allows to increase the productivity of a data science team, speed up and automate the creation of ML models, ease the ML process, and provides a variety of built-in algorithms and methods for data science.

There are other domain-specific data analysis tools. Mobyle [**95**] is a framework that provides an efficient web-based solution to run and chain bioinformatics analyses. Galaxy [**52**] is an open web-based platform for genomic research. Taverna [**68**] is a biology-specific application that allows bioinformaticians to build data pipelines to perform different analyses such as sequence analysis and genome annotation on a wide integrated database. Kepler [**88**] is a free and open-source system for designing and executing scientific workflows. It provides support for Web service-based workflows and Grid extensions. Kepler supports hierarchy in workflows. It allows complex tasks to be decomposed into simpler components. Kepler differs from the other bioinformatics tools by its computational model, It separates the structure of the workflow from its model of computation. It enables also semantic annotation of workflow

components. Besides, Weka [**66**] is a popular machine learning workbench. It is a general-purpose and interactive tool for data preprocessing, visualization, applying ML models, validation, etc. Weka contains a collection of visualization methods and many algorithms for predictive modeling and data analysis. Weka supports several ML tasks such as data manipulation, feature selection, standardization, etc. Also, it provides a wide collection of built-in algorithms for classification, regression, and clustering.

2.2.2. Metamodel smells detection and refactoring

In this section, we investigate the related work done in metamodel smells detection and refactoring.

Many works have been done related to model refactoring. Bettini et al. [**21**] have summarized existing approaches for managing metamodel quality and refactorings and identified common parts which are: *quality attribute specification and evaluation*, *bad smell specification and detection* and *refactoring specification and application* to be able to identify and resolve bad smells.

Additionally, Bettini has proposed in [**21**] a quality-driven framework for the detection and resolution of metamodel smells where a set of bad smells are first identified. Then, based on the selection of what quality attributes we want to improve, all the bad smells affecting these quality attributes are eliminated by applying the correspondent refactoring. In this work, a domain-specific SDL language, called Edelta [**20**] is used to define bad smell finders and refactorings.

In [**11**] and [**10**], an EMF Refactor tool was presented, that provides model refactorings specification and application. Refactorings were defined using the model transformation language Henshin.

In [**114**], the authors present an extensible generic refactoring framework based on EMF for modeling refactorings for different modeling and meta-modeling languages. Generic refactorings can be reused for different languages only by providing a mapping. Furthermore, the same generic refactoring can be repeatedly applied to one language. Role models have been used to formalize the structural requirement for refactorings. Then, they used a mapping specification to bound role models to specific modeling languages. This mapping defines which elements of a language play which role in the context of refactoring. Then, generic transformation specifications are executed to restructure the models based on these defined mappings [**114**].

In addition, Gheyi used in [**51**] a constraint-based technique to refactor models. Indeed, the semantics of the Unified Modeling Language (UML) was expressed as a set of well-formedness rules. These rules control the refactoring operation of a model or a set of connected models. After performing a refactoring, the rules are checked if they are still satisfied,

otherwise, the refactoring tentative is not valid and should be rejected. Using constraint-based refactoring, the authors proposed to replace constraint checking with constraint solving by computing additional model changes required to have a valid and semantic-preserving refactoring transformation.

In [**123**], some guidelines were provided on how to prove the soundness of model refactorings with respect to the formal semantics of Alloy. First, the authors defined the grammar of alloy and its semantics as a set of logical constraints (e.g. well-formedness of models and signatures ... two signatures in the same module cannot have the same name, a signature cannot extend itself, etc.). Then, they defined some essential laws to create valid and sound Alloy transformations. Next, this approach was validated on two alloy refactoring operations. This certainly allows us to build more reliable model refactoring tools.

Conclusion

Throughout this chapter, we introduced and detailed the necessary background related to our thesis. Then, we presented a literature review on recent related works. In the next chapter, we present our first contribution which consists of cartography of ML-based opportunities to improve machine learning pipelines and help software practitioners in their tasks.

Chapter 3

Automation and improvement of the software development pipeline: A cartography of ML-based opportunities

Introduction

In this chapter, we present our first contribution in collaboration with an industrial organization. It consists of designing a cartography that portrays the potential AI-based improvement opportunities for software-related tasks. First, we outline the context of our research contribution. Second, we analyze the tasks and pipelines inside the AI team of our industrial partner. Third, we identify the improvement opportunities as well as the expertise required to accomplish each task. Finally, we depict the potential AI-based improvement opportunities that allow us to infer that knowledge from historical data.

3.1. Context & Motivations

3.1.1. Context

In light of the great research progress, the AI field is witnessing, the other fields are trying to adapt to this fact by taking advantage of the potential of AI. Indeed, the industry and the other research areas are trying to incorporate some kind of automation and intelligence through using AI. This definitely opens up prospects to promote innovation, enhance the productivity inside companies, assist teams and employees to be more efficient, and productive and allows undoubtedly to solve challenging problems and to build smart systems.

Nevertheless, AI should be used properly to be able to generate an added value. So, we should identify the right tracks where to employ AI.

In this project, we collaborated with an industrial partner that is considered among the most prominent IT companies in Canada. The project was mainly dedicated to their AI team and more generally to all the research community.

3.1.2. Motivations and objectives

AI is a rising and fast-growing field that has been used in multiple other areas to solve complex problems and to inject a kind of intelligence into systems. It's very important to take advantage of the strong capabilities of AI. The research is already progressing ahead in that direction. Nevertheless, we need sometimes to take a step back for a clearer view. Accordingly, having a global view about the flaws, problems, improvement, and automation opportunities that we can get by the mean of using AI is very important. This will help software practitioners in their common tasks and raise their productivity and performance towards delivering software with high quality.

The present project will help delineate some of the future research directions about using AI to improve the software-related tasks, especially, the activities related to producing machine learning models. These improvements involve automating and optimizing pipelines as well as assisting software practitioners in their common and complex tasks. Thus, software practitioners could focus on more valuable tasks that require more human creativity, intelligence, and intuition to solve complex problems. Furthermore, the present project will aid our industrial partner to polish its processes and to promote the effectiveness in delivering high-quality artifacts.

3.2. Adopted methodology

As previously mentioned, the goal of this project is to create a cartography that portrays the activities inside the AI team of our industrial partner, depicts the required expertise to accomplish them, and presents some AI-based improvement opportunities. These opportunities allow either to completely/partially automate these tasks or to assist software practitioners in their activities. To reach this target, we define a systematic approach as illustrated in figure 3.1.

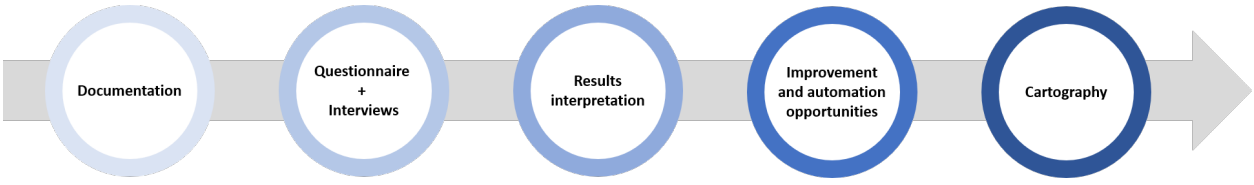


Fig. 3.1. Adopted methodology

3.2.1. Documentation

The first step of our adopted methodology consists of understanding the different pipelines and activities inside the AI team of our industrial partner. In fact, recommending improvements should start by building a strong understanding of the corresponding tasks. To this end, we looked through the documentation of the different solutions and pipelines.

3.2.2. Questionnaires & Interviews

Having understood the pipelines and activities, we interviewed different members of the AI team. The goal of this step is to grasp the different tasks realized by the different members, explain more in detail some processes, and to identify some challenges, issues, or existing limitations, etc.

We started by identifying different axes from the first step as shown in figure 3.2.

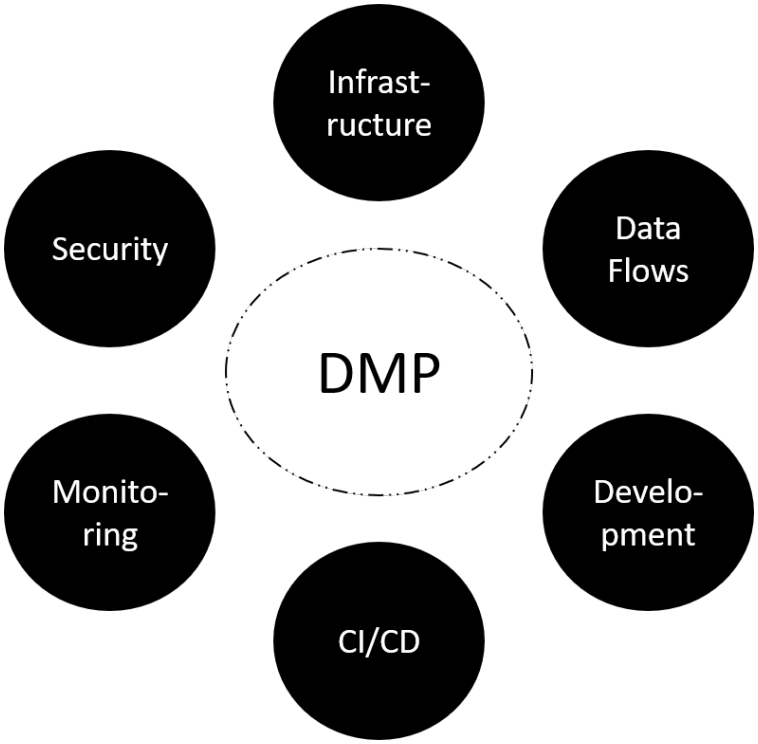


Fig. 3.2. Different axes related to the AI platform

The AI team has a main platform for the development of predictive artificial intelligence models, to accelerate the development, deployment, monitoring, and consumption of different use cases.

As shown in figure 3.2, the AI platform involves many axes: Security, Infrastructure, Data Flows, Development, CI/CD, Monitoring, etc. In Table 3.1, we detail these axes.

Table 3.1. Description of the different axes

Axis	Description
Security	It involves all the processes and activities related to the security and confidentiality of data , resources, network, applications, etc.
Infrastructure	It involves the management, monitoring and access to the different resources.
Data Flows	It includes the tasks related to data management, transfer, storage and transformation, etc.
Development	It includes the development, validation and evaluation of machine learning models.
CI/CD	It encloses everything related to the practices enforcing the automation in building, testing and deployment of applications (including machine learning models).
Monitoring	It refers to the monitoring of machine learning models. It consists of tracking and detecting the model performance deterioration. It involves as well the resolution of this issue.

Table 3.2. Information about the questionnaires and interviews

Number of questionnaires	6
Average number of questions per questionnaire	32
Number of interviews	14
Number of interviewees	11

Based on the documentation of these activities, we prepared a questionnaire for each axis. These questionnaires served as a support to conduct meetings with different people involved in each axis and who have sufficient knowledge about the different related activities. Table 3.2 shows some information about the interviews and questionnaires.

The questions were given in advance to the interviewees, so, they could get prepared and take enough time to think about some questions which are not simple but require both time and deep thinking to get relevant responses. For each meeting, we had different rounds for the sake of further details and clarifications. Table 3.3 shows an excerpt of questions from the Monitoring questionnaire with the monitoring scientist. This questionnaire was used as support to conduct the meeting by defining the main topics to be addressed.

3.2.3. Results analysis

After collecting information from the different meetings and having understood the different software-related tasks inside the AI team of our industrial partner, we documented the description of each task and its related activities. Then, we analyzed each task and identified

[3]TensorBoard: `https://www.tensorflow.org/tensorboard`
TensorBoardX: `https://github.com/lanpa/tensorboardX`
Visdom: `https://ai.facebook.com/tools/visdom/`

Table 3.3. An excerpt of the monitoring questionnaire

	Question
1	Could you tell me about your regular tasks and your role inside the team?
2	Could you give me some details about the monitoring platform features?
3	What are the technologies used in the implementation of this tool?
4	Why didn't you use existing solutions like TensorBoard[1], TensorBoardX[2]or Visdom[3]etc.
5	Is the monitoring dashboard highly integrated with the AI platform? If yes, could you give me some details?
6	What metrics are you using to monitor the performance of deployed machine learning models?
7	What metrics are you using to monitor data?
8	Are you using some real-time metrics?
9	Are you keeping track of the historical evolution of these metrics?
10	For the batch metrics that are computed periodically .. What's the frequency of computing these metrics? (daily, weekly or monthly, etc.)
11	How do you monitor these metrics? Are you using thresholds, statistical methods, or predictive techniques?
12	In case you are using thresholds to monitor models and data, How do you define those thresholds? Is it dependent on the type of problem and application? Are you considering adjusting these thresholds through time?
13	What actions are executed when an issue is detected by the monitoring tool?
14	Are those actions automated? Is there a set of automated actions that would be executed if an issue is detected?
15	In what cases do you consider re-training a deployed machine learning model?
16	Do you train the model on new data or re-train it from the beginning?
17	How do you select the re-training data?
18	Does the monitoring task involve any manual intervention? (Either to monitor models and data to detect potential issues or to resolve the identified problems)
19	Could you tell me about the automation and reusability level of the monitoring platform? Is it easy to monitor a new deployed model and import it to the platform? Do you use configuration files or a graphical console for this?
20	What are the most challenging tasks for a monitoring scientist?
21	What's the performance of the monitoring platform so far?
22	What are the limitations and problems of the monitoring platform?
23	To what extent this platform is re-usable for any kind of machine learning model and in other applications (e.g. natural language processing)?
24	To what extent the monitoring platform is scalable for a large amount of data and a big number of deployed models?
25	Could you show me a demo of the monitoring platform?
	⋮

the required expertise and the necessary knowledge to accomplish it. Furthermore, we have identified the problems and deficiencies related to each task. The identified issues could be related to the lack of optimization, complexity, potential bugs, waste of effort and time in some classical and typical activities, etc. Results are presented in section 3.2.4.

3.2.4. Improvement and automation opportunities

Once the results are analyzed by identifying the related problems and the expertise required for the accomplishment of each task; we locate different automation and assistance opportunities to improve these tasks. The suggested ideas could be categorized into 3 groups:

- Ideas that allow overcoming the identified problems.
- Ideas that allow to completely or partially automate some tasks by inferring the required knowledge using historical data. This will allow to alleviate the burden on software specialists regarding the typical and common tasks and let them focus on more valuable and difficult activities that require more human intuition, creativity, and intelligence.
- Ideas that allow improving some tasks by using more sophisticated and intelligent techniques and by assisting software specialists in their tasks. This could yield better performance, precision, and effectiveness while performing these tasks.

3.2.5. Graphical cartography

This step consists of delivering an artifact that encloses the results of the previous steps: tasks description, required knowledge to perform each task, and the improvement opportunities.

The deliverable of this step is a graphical cartography that encompasses all the aforementioned details which will serve as a road map for the AI team of our industrial partner to improve their internal processes in the short, medium, or long term. This cartography has a broader scope as it could serve as a guide for researchers who are interested in tackling real problems that software practitioners face in an industrial environment.

In the next section, we describe with details some software-related tasks based on the information collected from the AI team of our industrial partner.

3.3. Processes and tasks description

We had access to the documentation of the different pipelines and applications. This was consolidated with the conducted meetings which allowed us to build a strong understanding of the different pipelines and tasks performed inside the AI team of our industrial partner. This documentation step serves as a fundamental basis for the next steps.

3.3.1. General presentation of the AI team of our industrial partner

The AI team is responsible for the industrialization of AI models to support decision making in the business processes of the company.

The AI team has many roles and responsibilities including:

- Manage technical assets for the industrialization of AI.
- Guide and manage research projects.
- Develop knowledge, expertise, and maturity of teams on the subject of AI-related software development.
- Supervise development practices related to data engineering and software development as part of the industrialization of AI solutions inside the company.
- Spread the expertise of development, maintenance, evolution, and management of AI assets.
- Raise awareness among teams of the potential of using data to support innovation and decision-making in the information technology sector.
- Support the other IT teams in the development of innovative solutions exploiting data and AI techniques.

The AI team is managing the following assets:

- AI platform: it is a platform for deploying artificial intelligence solutions.
- Chatbot platform: it is a dialogue engine platform to deploy chatbots.
- Artificial intelligence operating environment (VM-AI): it's a cloud environment that includes all the tools and needs required to access different resources (network, databases, etc.) and develop AI solutions.
- Deployed ML and NLP models.

To sum up, the AI team is a platform for artificial intelligence software deployment to ease and automate decision-making from within the business processes of the company. The platform is transversal and available to various clients.

Figure 3.3 shows the process of generating AI software.

Many actors are involved in the process of generating AI software: data engineers, scientists, analysts, project managers, monitoring scientists, software engineers, line of business subject matter experts (LoB SME), etc.

Our industrial collaborator has multiple data sources exposed via APIs to be used by the IT teams after being tokenized [4].

Scientists could securely access data through the Data API from the AI development Virtual Machines. From the Extraction VM, scientists could access data sources that are not exposed yet via APIs. The extracted data is saved to Big Query (a data warehouse

[4]Tokenization is the process of transforming a sensitive piece of data into a random string called a token that has no meaning if breached.

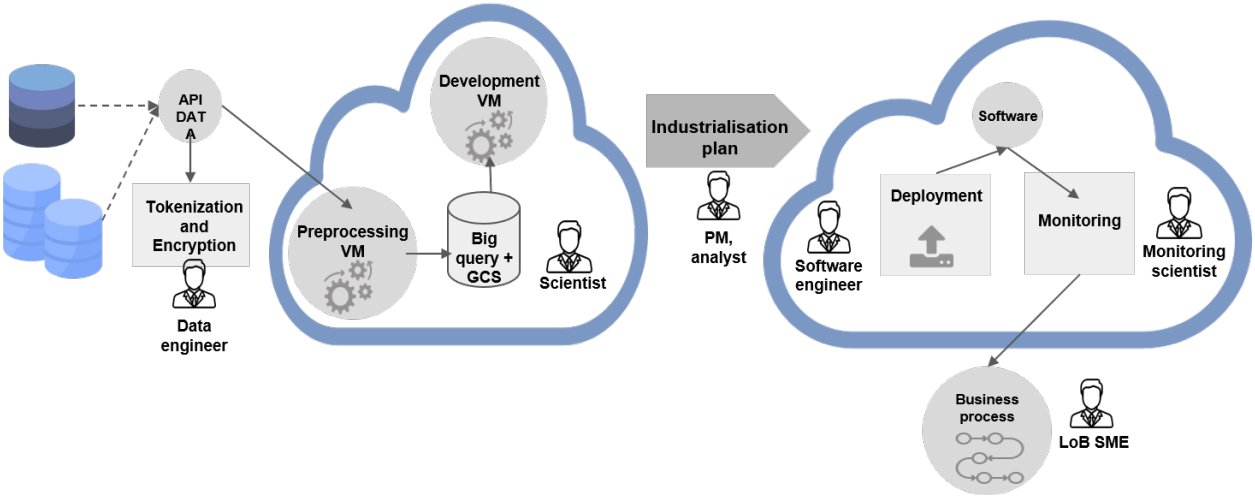


Fig. 3.3. How to generate an AI software?

in Google cloud) or to GCS (Google Cloud Storage is a file storage system in google cloud used to store and access data. It is that is accessible online via restful API). Direct access to data is controlled and supervised to ensure the confidentiality of sensitive pieces of data. The AI development VMs serves for developing machine learning models. Scientists are autonomous and free in setting up their work environment (IDE, programming language, library, etc.) and they have access to data sources. Afterward, the ML model goes through an industrialization plan to be deployed into production while being monitored to ensure its performance. Finally, the deployed ML model is integrated into the business processes to ease and automate decision making.

In the next section, we focus on the AI platform. We explore the different components and technical details of the platform as well as the tasks of the different actors.

3.3.2. AI platform

The AI platform aims to set up a transversal platform for the development of predictive artificial intelligence models, to ease and accelerate the development, deployment, monitoring, and exploration of ML models. The platform serves for data ingestion, prediction and results preparation for the production system.

Figure 3.4 highlights the different components of the AI platform.

The AI platform is mainly composed of two major parts:

(1) A machine learning workflow that contains steps representing compound tasks[5], connected by paths that define the flow to develop and deploy ML models. The ML workflow adopted in the AI platform is respectively composed of the following steps:

[5]Compound task: It is a complex task composed of a set of primitive and compound tasks.

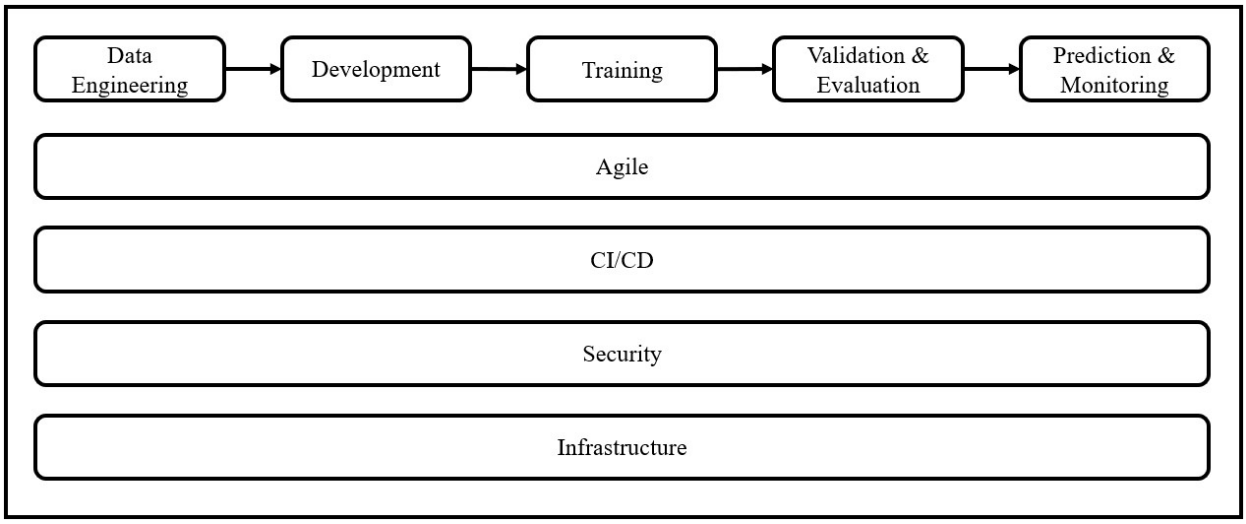


Fig. 3.4. AI platform: workflow and components

$DataEngineering \rightarrow Development \rightarrow Training \rightarrow Validation\&Evaluation \rightarrow Prediction\&Monitoring$

(2) A set of transversal activities that are the horizontal tasks involved in many steps of the ML workflow. The transversal activities are the following: *Agile, CI/CD, Security, Infrastructure.*

The AI platform is based on Google Cloud Platform and uses extensively the cloud services. In fact, using a cloud infrastructure has some key benefits:

- Efficiency: By using cloud infrastructure, a company could reduce the cost of purchasing and maintaining hardware and software equipment.
- Downtime[6] reduction: Using the cloud infrastructure allows to reduce the downtime of systems.
- Security: Cloud providers offer many security features. Some baseline protections are implemented in the cloud platforms and a bunch of configurable features (access control, authentication, encryption, network scans, and filtering, etc.) are offered.
- Scalability: Using a cloud infrastructure guarantee scalability by offering the option of handling a growing amount of work and dynamically add resources to the system. This level of flexibility and agility in infrastructure is very important for companies since they can adapt the infrastructure size and capabilities according to their needs.
- Performance and availability: The cloud providers guarantee a high-performance level for the resources and high availability of hardware, systems, and services.
- Mobility: The cloud infrastructure is always accessible from anywhere through connected devices.

[6]Downtime: time during which a system is down and unavailable for use.

Our industrial partner uses Google Cloud Platform (GCP) as an infrastructure. It is a suite of cloud computing services offered by Google including data storage, create and configure and manage computing resources, data analytics, machine learning, security, resources monitoring, etc. [**1**].

We present a subset of the services offered by GCP that are used in the AI platform [**2**]:

- Google Compute Engine (GCE): Google Compute Engine is an Infrastructure as a Service (IaaS) component offered by Google Cloud Platform. It is a core component built on top of the internal infrastructure that Google uses for its own products such as Google's search engine, Gmail, Youtube, etc. This service enables users to create Virtual Machines (VMs) on demand. Users can configure these VMs by setting up their computing power, memory size, disk size, etc. according to their needs. These steps could be controlled and managed by the organization. h
- Google Cloud Storage (GCS): GCS is a storage service offered by Google Cloud Platform. It is a web service that offers a restful file storage system to store and access data on the GCP infrastructure with other features such as sharing, security, backup, recovery, etc. GCS is an Infrastructure-as-a-Service (IaaS) component that's comparable to Amazon S3 online storage.
- Google Kubernetes Engine (GKE): It is a GCP service that enables cluster creation, management, and orchestration to run docker containers. It has several assets such as scalability and performance. GKE makes it easy to create a cluster that has a dynamic number of configured VMs instances based on containerization, then to easily deploy applications. It presents several features as logging, cluster monitoring, applications management, cluster configuration (to ensure scalability and performance in case of high traffic of requests). Besides, GKE offers several configuration options including cluster size, number of replicas, CPU capacity, memory size, etc.
- Google Container Registry (GCR): GCR is a GCP service that allows to save and manage Docker images. Docker images could be used to create instances of preconfigured VMs on demand. This GCR is securely accessible and enables users to pull, push, and manage images.
- BigQuery (BQ): BQ is a Platform-as-a-Service data warehouse offered by GCP that supports saving and querying data. BQ is a highly scalable and cost-effective data warehouse that stores heterogeneous and big amounts of data for further analysis. It has also built-in machine learning capabilities.
- Data Flow: It's a GCP service that offers features of batch and real-time data processing. It enables users to set up pipelines for data transformation, transfer, preparation, aggregation, and analysis, etc.

- PubSub: It's a messaging middleware that allows to send and receive messages among applications and services. This fully-managed real-time and asynchronous messaging service.

In the following sections, we detail the activities of the AI platform involved in building ML models.

3.3.2.1. Data engineering.

The data engineering step contains mainly two major categories of components: re-usable modules that are developed once and used in all the ML projects (data APIs, data encryption and tokenization modules, etc.) and tasks that are specific to each project (data cleaning, feature engineering, etc.).

As illustrated in figure 3.3, many data sources are securely exposed through API. Sensitive information (email, social security number, etc.) is tokenized and encrypted to ensure confidentiality.

Our industrial collaborator has multiple and various data sources in the production environment. It is not efficient to access the production data directly by the internal IT teams. This has many drawbacks: network congestion, error-proneness, performance degradation, etc. Therefore, the data is duplicated from these sources using secure APIs into the platform. This phase is called *Data Ingestion*. The cloud is used as a storage system, more specifically, BigQuery and GCS (Google Cloud Storage). During the ingestion phase, a process consumes data from source systems and brings them to the platform to facilitate and accelerate the subsequent phases. The duplicated raw data is mapped into domain data. In fact, using domains allows to organize data, extract knowledge, and can make it easier to understand the structure and content of a database.

On the other hand, the data engineering step includes tasks that are specific to each machine learning project:

- Collect the model dataset by pulling data from multiple domain data sources to create the ML model features.
- Pre-process the dataset which includes handing the missing features, pre-process textual fields (tokenization, stemming, lemmatization, stop words, etc.), remove duplicated values, etc.
- Data transformation by encoding textual and categorical data as well as images.
- Feature scaling includes standardization and normalization of numerical data. This allows to re-scale data to the same interval and eliminates the effect of having large numbers that dominate the small numbers. Another reason is that machine learning algorithms converge much faster with feature scaling and coefficients are penalized appropriately [**70**].

- Feature Engineering consists of analyzing the features and modifying them to ensure better performance and get better results. It includes features combination, reducing features using dimensionality reduction techniques, features selection, etc.
- Split the dataset into 3 subsets: train, validation, and test. This could include shuffling the dataset to ensure randomness and balance.

3.3.2.2. Development.

This part consists of designing and implementing machine learning models. The programming languages used by the scientists and developers are mainly Python and Java. Many frameworks could be used (e.g. TensorFlow, Pytorch, Keras, Scikit-learn, Theano, etc.) based on scientist preferences and qualifications. During this phase, the scientists choose and design appropriate models and this depends on many criteria such as the problem type (classification or clustering or regression), context (medicine, finance, automotive, etc.), expected output, data structure, etc. Then, scientists implement these models using the appropriate language and framework. The development phase takes place in some AI VMs in the cloud. These VMs are configured and created on-demand using the GCE service. Some docker images are already set up in GCR with the necessary dependencies, configurations, and security measures.

3.3.2.3. Training.

Training is an important step in the ML pipeline. It consists of training the developed model on the prepared data using the infrastructure of the AI team to fit the models on the data and learn the model parameters. The training process should be monitored to ensure that the model keeps improving along with iterations and to detect potential issues such as over-fitting. This step could be revisited after the model deployment in case of re-training the model on new data. This occurs if the model performance degrades and gets worse in the production environment.

3.3.2.4. Validation & Evaluation.

The validation step uses the validation dataset and aims to tune hyper-parameters and choose the optimal configuration leading to the best performance. Unlike model parameters, a hyper-parameter is an untrainable parameter that we cannot learn, however, we should set it manually to control the learning process. The performance of a ML model is dependant on the hyperparameters configuration. As examples of hyperparameters, we cite model type, learning rate, number of layers, number of nodes in each layer, number of clusters, etc. We cannot know in advance the best hyper-parameters' values for a given problem. The values of hyperparameters can be decided by setting different values, training different models, and choosing the values with the best performance. In the validation, we may also use a search strategy to test a bunch of values for each hyper-parameter and converge rapidly to the best hyper-parameters' configuration that leads to the best performance. This process

of finding the optimal hyper-parameters is called hyper-parameters optimization. The common hyper-parameters optimization techniques are Grid Search, Random Search, Bayesian Optimisation, Genetic Algorithms, etc. The validation process could also be done based on scientist expertise by selecting the best hyper-parameters according to past experiences with similar problems. The optimal hyper-parameters configuration that has the best accuracy in the validation phase is selected. Finally, the model is evaluated on the test set to estimate its final performance on a different dataset using a set of metrics such as precision, recall, F1-score, etc.

3.3.2.5. Prediction.

The evaluated model is packaged in a docker container as illustrated in figure 3.5.

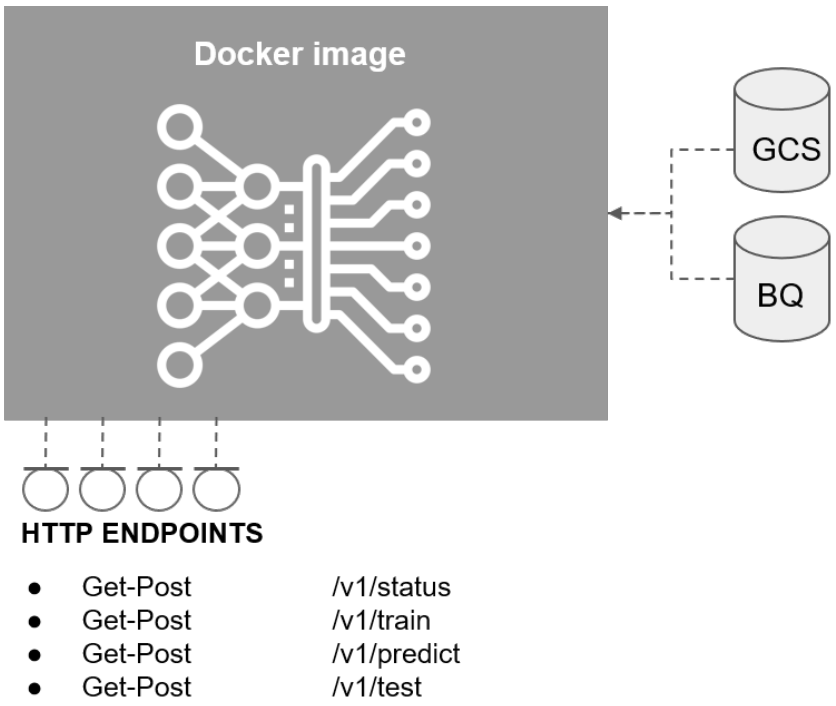


Fig. 3.5. The major components of an API template for a deployed ML model

The API template is composed of different components and is used to bring AI models into production to be used by operational systems. As shown in figure 3.5, the API template runs in a docker image which exposes several endpoints for model training, testing, individual or batch predictions, etc. The container is configured to be able to connect to both the BigQuery database to retrieve the raw data and GCS to save the results and the serialized files of the model.

There exist two kinds of predictions: batch and real-time. A real-time prediction is executed for an individual in quasi-real-time for online data. In contrast, batch prediction consists of applying the model periodically on an existing dataset. Finally, the predictions

are saved back to the operational databases, so that they could be used by operational systems or other employees to support decision making.

3.3.2.6. Monitoring.

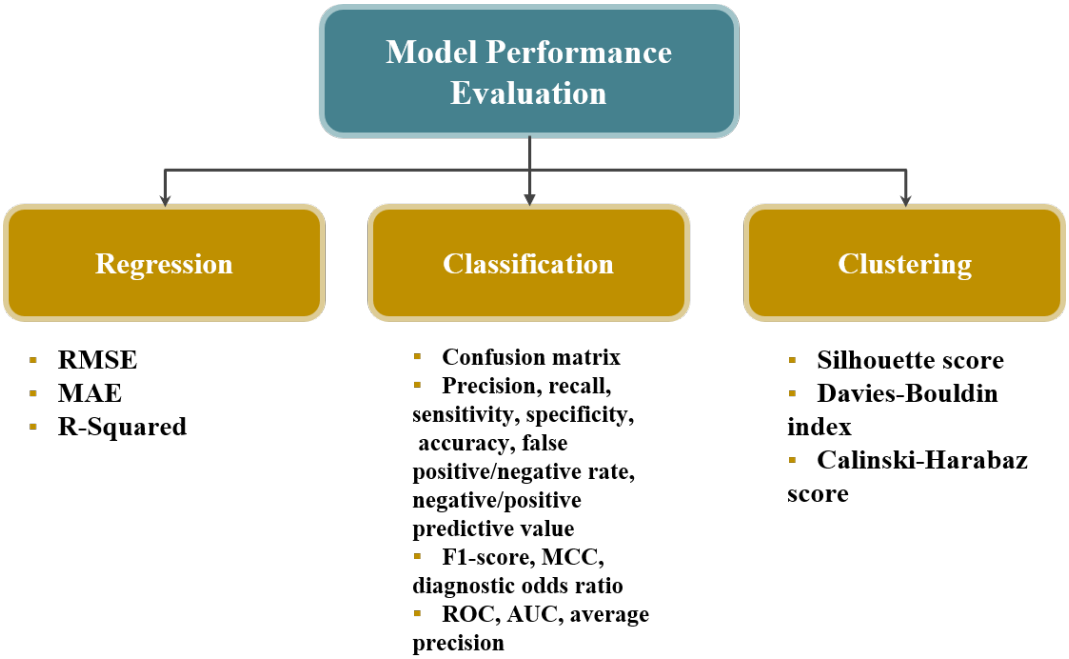
Monitoring refers to tracking and supervising model performance in production. The goal is to maintain a certain level of stability of accurate predictions. The detection of performance degradation in the monitoring phase should be followed by a re-training phase to adapt the ML model to the new data and to capture the new changes whether new concepts or new statistical relations between features, etc.

The monitoring of ML models has two main components:

- **Data health monitoring** consists of tracking the data to detect potential anomalies to reflect the quality of data. As illustrated in figure 3.6, we could decompose the data health evaluation into multiple components: *Summary for prediction* gives information about the predictions such as the proportion of each class, histogram of predictions, Population Stability Index (PSI) for prediction output that measures how much a population has shifted over time, etc. *Verification of data structure* gives details regarding the data structure such as the number of features and records. *Distribution test* consists of comparing the distribution of training data to the distribution of production data. It includes measuring PSI for all the features, Kolmogorov-Smirnov test (K-S test) to test if two data samples come from the same distribution [**94**]. The K-S test allows detecting if the production data is unrepresentative of the training data. In such cases, the model becomes invalid since it's trained on different data and its performance would deteriorate through time. *Summary of variables' statistics* comprises counting the number of records with missing data, computing some numerical metrics such as the number of records in each class, minimum, maximum, median, mean, standard deviation, quartiles [**6**], etc. Metrics and charts reflecting the *Variables' distribution* such as frequency charts, histograms with kernel density, etc. *Special data health check* contains other techniques of data health evaluation such as computing the pairwise correlation of features and with the output variable, etc.
- **Model performance monitoring** consists of monitoring the performance of the machine learning model in production. Tracking the model performance depends on the problem type (regression, classification, or clustering) and could be done using evaluation metrics as shown in figure 3.7. For instance, some metrics that could be used to measure a regression model's performance are: Root Mean Square Error (RMSE), Mean Absolute Error (MAE), R-Squared, Mean Absolute Percentage Error (MAPE), etc. [**23**] As shown in figure 3.7 a classification model's performance could be evaluated using precision, recall, sensitivity, accuracy, F1-score, etc. [**67**]

In regards to a clustering model, it's performance could be evaluated and tracked using Silhouette score, Davies-Bouldin index, Calinski-Harabaz score, etc. [**35, 115**] Further, more advanced ML techniques could be used to detect the performance degradation of a ML model that could be caused by the change of concepts, change of the statistical relation between the features, or the change of the user's behavior in the production environment.

Fig. 3.6. Sample of data health evaluation metrics



RMSE: *Root Mean Square Error* - **MAE:** *Mean Absolute Error* - **MCC:** *Matthews correlation coefficient* - **ROC:** *Receiver Operating Characteristic curve* - **AUC:** *Area Under the roc Curve*

Fig. 3.7. Sample of model performance evaluation metrics

3.3.2.7. Continuous Integration / Continuous Delivery (CI/CD).

The CI/CD pipeline is the backbone of the process of producing software artifacts. It is a set of practices used in software engineering that aims to automate the integration, building, testing, and deployment of source code. The main purpose of this paradigm is to

detect issues and conflicts in the early stages, minimize the cost and effort, and maximize the automation level in software engineering. CI/CD promotes frequent code changes as well as collaboration between the team members. Then, the source code of the different collaborators is integrated and merged after resolving the detected conflicts. The purpose is to reduce the probability of defects during the final phases of the project development where bug fixing would be more expensive. Figure 3.8 illustrates the paradigm of CI/CD and its impact on a software development pipeline.

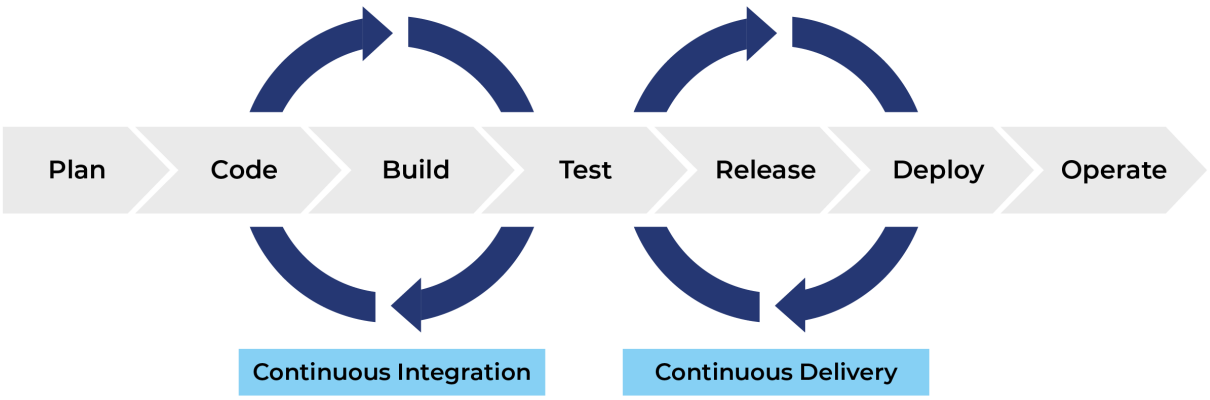


Fig. 3.8. Continuous Integration/Continuous Delivery (CI/CD) workflow

The CI/CD paradigm integrates the following components:

- *Source code versioning:* It consists of keeping track of the history of changes made on the source code. Some version control systems could be used in this context such as Git, SVN, Mercurial, etc. Many tools integrate these systems and provide powerful features for source code versioning, integration, and collaboration such as GitHub, Bitbucket, GitLab, etc. Our industrial partner uses Bitbucket as a version control repository for source code hosting and collaboration.
- *Source Code integration (Pull request & Conflicts resolution):* For each project, a shared repository is created to contain the source code of the software and to keep track of code changes history. The team members of our industrial partner work in parallel and collaborate on the development of the same software. They commit their changes frequently to their own branches. Then, the source code is merged into a master branch that contains the stable and final version of the software. The merging phase includes resolving the merging conflicts between collaborators. The version control tools provide powerful and advanced features to detect conflicting areas in the source code and gives assistance to resolve them.
- *Source Code Testing and analysis:* The CI/CD pipeline automates the testing phase. After the integration and merging of source code, the developed tests are automatically executed. If a test fails, the team is notified of the testing results to make the

appropriate corrections. The CI/CD pipeline involves an analysis phase to inspect the source code quality, security, and bugs. SonarQube is an open-source platform very commonly used for code review. Our industrial partner uses SonarQube to generate reports on source code bugs, vulnerabilities, code smells, coding standards, code complexity, code coverage, etc. [**3**] SonarQube could be easily integrated with many other tools (e.g. Eclipse and JetBrains IDEs, Jenkins, Maven, Ant, Gradle, MSBuild, etc.). It promotes continuous inspection of code quality to perform automatic reviews with static analysis on many programming languages [**29**]. Figure 3.9 shows an excerpt from a SonarQube report.

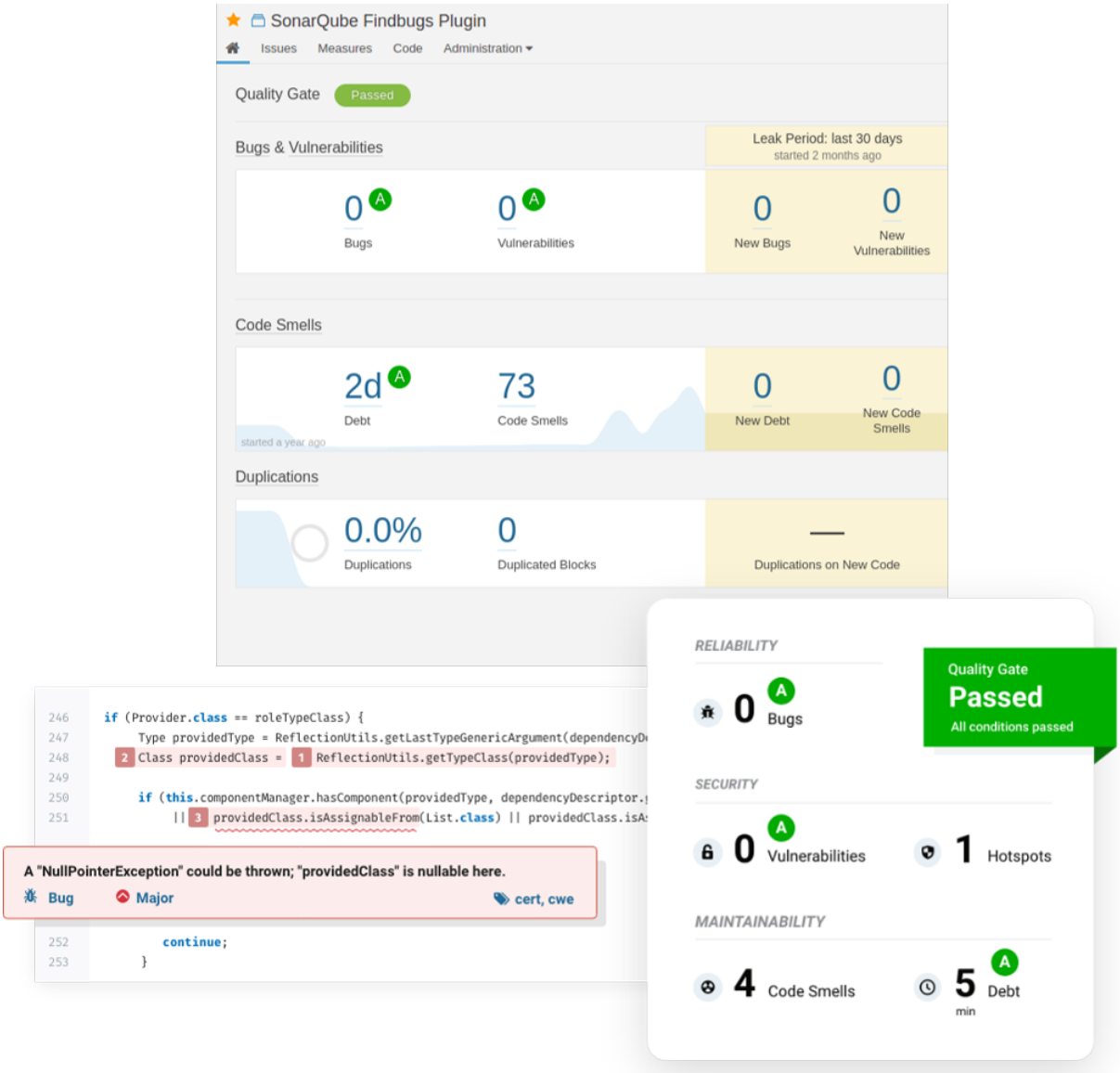


Fig. 3.9. SonarQube: an excerpt from the report generated on an example

SonarQube can record metrics history and their evolution through time. Furthermore, it provides suggestions to resolve the detected issues.

– *Source code Review (also called peer review):* Consists of assigning a reviewer to check the software quality and validate the new source code before its integration with the master branch.

Jenkins[7] is also used by our industrial partner to easily build CI/CD pipelines and that supports many tools (e.g. Github, SonarQube, etc.)

3.3.2.8. Security.

Security is an important and critical component for our industrial partner in particular and in software engineering in general. Indeed, developers and scientists do not have unlimited and unconditioned access to confidential data. Our industrial partner categorizes data into four levels according to their level of confidentiality:

- *Level 1:* Highly confidential data
- *Level 2:* Confidential data
- *Level 3:* Internal data
- *Level 4:* Public data

According to the confidentiality level, the data is encrypted and/or tokenized. Tokenization is the process of substituting a sensitive data element with a non-sensitive and randomly generated equivalent called a token. The token is a reference that maps back to the original data. This process is handled by a tokenization system. The advantage of tokenization is that it allows preserving the length and format of data. Encryption is the process of transforming input data into another form that only people having access to a certain key (decryption key) can read it. The encryption process relies on a mathematical relation between the original data, the secret key, and the ciphertext. It's very hard to decrypt the ciphertext and get the original data if we don't possess the secret key.

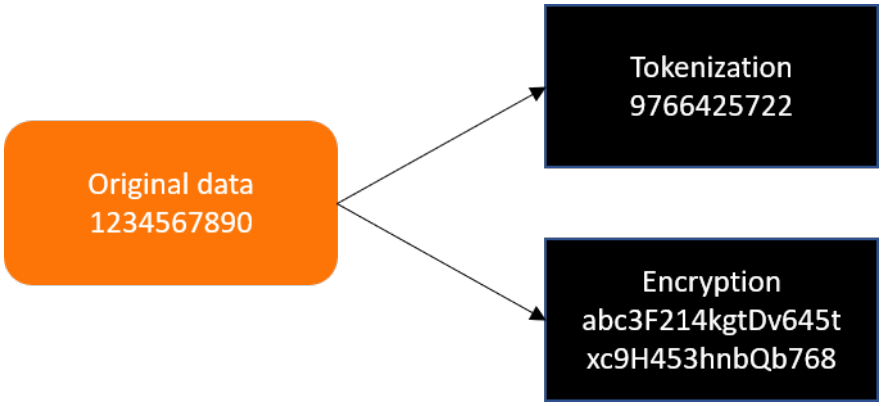


Fig. 3.10. Tokenization VS Encryption

[7]https://www.jenkins.io/

Figure 3.10 shows the difference between tokenization and encryption. The encryption is based on secret keys to encrypt and decrypt and the input data is related to the ciphertext with a mathematical function. It's almost impossible to reverse this process if we don't possess the keys due to its extremely high computational cost. On the other hand, the tokenization is based on a mapping function. There is no explicit relation between the original data and the token. In contrast to encryption, the tokenization is length and type preserving. Also, encryption is costly in terms of computation in contrast to tokenization which is a simple process that doesn't need computations or to manage the secret keys. However, encryption is more adapted for unstructured data such as text, files, etc. The encryption is ideal for data exchange since the receiver should only possess the decryption key to read the data. By contrast, it's difficult to exchange a tokenized data since it required direct access to the tokenization system.

Security covers also the source code. As stated before, our industrial partner uses SonarQube to detect vulnerabilities. In addition, it uses Nexus-IQ [8] tool to detect vulnerabilities in the code source code dependencies.

Figure 3.11 shows an example of a report generated by Nexus-IQ. The main use case for Nexus-IQ is to monitor the dependencies and track their vulnerabilities. The generated report by Nexus-IQ gives a description of the vulnerability and its evolution throughout the different releases. It recommends, as well, potential solutions or alternatives to solve the detected vulnerability.

Other security measures are also set up by our industrial partner to assess the VMs security: authentication, limiting admin permissions (restrict the actions that can be performed as root), network scan, logging, monitoring, access control, alerting, etc. These measures reduce the risk of data leakage and unauthorized access.

3.4. Identified problems and required expertise for software-related tasks

Having identified and described the tasks involved in the software generation pipeline, we analyzed these tasks to identify the problems as well as the expertise required in the accomplishment of each task. We do believe that identifying the problems of software-related tasks as well as the knowledge required to accomplish these tasks could lead to the identification of improvement opportunities of these tasks. Some software-related tasks are facing challenges and problems such as the lack of automation, difficulty, optimality, etc. These problems represent opportunities for improvement that we should work on. The improvement opportunities could be related to increasing the automation level, optimizing these tasks, assisting developers and scientists, etc. Furthermore, another way towards

[8] https://www.sonatype.com/nexus/iqserver

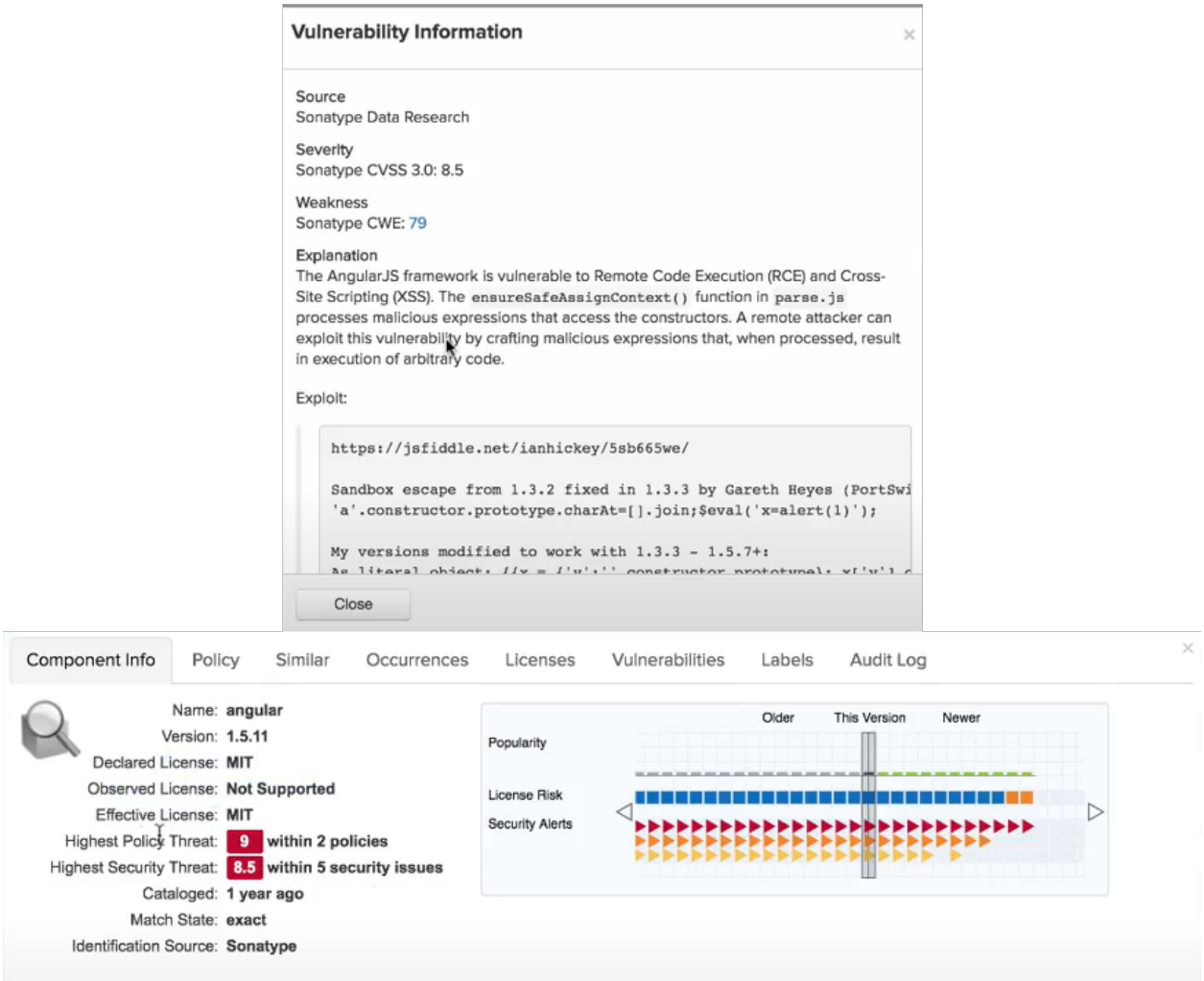


Fig. 3.11. Nexus-IQ: an excerpt from the report generated on AngularJS dependency

finding improvement opportunities consists of identifying the knowledge and skills required to accomplish each task. We formulate this knowledge and then suggest AI-based ideas that can infer this knowledge from historical experience and then assist software specialists in their tasks by the mean of automation (either full or partial automation), recommendation, or refinement. We identified this information (problems and required expertise) from the analysis of the different software-related tasks. Besides, in the meetings that we conducted with different software practitioners, we dedicated a part of our questionnaire for that purpose (e.g. Table 3.3).

We describe in Table 3.4 the problems and required knowledge towards the accomplishment of some software-related tasks.

	Task	Required Expertise
1	Data Engineering	- Data confidentiality is a preoccupation and needs to be constantly improved. - The access and usage of some confidential data is limited → waste of time and decreased team efficiency. - Setting up advanced security measures reduces the risk but doesn't eliminate it. There is always a risk even with a low probability. ⇒ A Naive solution to generate data is Random generator → Data could be inconsistent and unrealistic. - Software practitioners often need real data to simulate and test real-life cases. - Realistic data is useful for testing, visualization, development, data engineering... → Data is the fuel of the software development pipeline.
2	Development of ML models	- Implementing a Machine Learning algorithm is a recurrent task. - Depending on the problem type (regression, classification, clustering), many algorithms are often used thanks to their good performance. - The development task requires: • Language expertise: python, java, etc. • Library expertise: TensorFlow, PyTorch, etc. • Being able to re-use and adapt the old implementations to the new problem and input data and do not re-implement from scratch to gain time and effort. - The choice of the right ML model to use requires deep and diversified expertise in ML. - Each ML algorithm has its own advantages and drawbacks. The choice of the most appropriate ML algorithm to use depends on many criteria: • Problem description • Data description (type, distribution, size, etc.) • The desired output, etc.

3	The machine learning pipeline	- The machine learning pipeline is composed of the following steps: Data pre-processing → Development → Training → Validation → Evaluation. - These steps are recurrent and costly in terms of time and effort. - Scientists usually use the same algorithms depending on the problem type (classification: Logistic regression, SVM, Decision Tree, Random forest, etc.) - The most performant model is selected in the validation step and then evaluated. - According to '2018 Kaggle ML and Data Science Survey', *15–26% of the time of a typical data science project is devoted to model building or model selection.*
4	Evaluation of ML models	- Metrics are a static way to evaluate models and detect basic behaviors. - We need to understand better the strengths and weaknesses of the model. - We need to go deeper into the weaknesses of the model using other techniques and to identify the limitations of the model and correct them. - A machine learning model could be unstable in the prediction phase. Unstable model = similar input → considerably different output. ⇒ This input is called an Adversarial example. Adversarial examples could be critical in some sensitive cases (e.g. medicine, security, etc.) where we need a robust and performant model.

5	Monitoring	- Detect model performance degradation ¸and data drift is very important. - Thresholds are a static and classic way to detect model performance degradation. - Outliers are detected manually from the monitoring dashboard (graphs, distributions, tables, metrics, etc.) which is tedious and not accurate. - When we detect that the new production data distribution has changed (data drift) or the statistical relation between the features and the output variable is no more valid (concept drift); we should re-train and re-fit the ML model to the new data so it can learn the new statistical properties. - Data and concepts are changing, we should forecast the detection of these changes to act early and trigger the re-training process.
6	Scrum	- Sprint planning is an essential meeting in Scrum that consists of selecting a sub-set of stories from the product backlog and assigning stories/issues (tasks) to team members. - A user Story is characterized by: description – story points – priority – assignee. - Incorrect estimations of story points imply delays. - In the sprint planning, we should: • Prioritize the most important user stories. • Take into account the capacity and expertise of each of the team members. • Assign tasks according to the above criteria.

7	Security	- Security is a concern in IT companies and it includes the following aspects: • Ensure the security of the cloud environments. • Control access to different resources. • Assess the confidentiality of the data. - Security involves many challenges: • Receiving many security alerts. These alerts should be monitored, analyzed, and resolved by executing the appropriate actions. • We should be able to analyze many log files to localize security incidents (An intelligent log analyzer to analyze security threats is very important). • There could be new security risks that could not be detected using common tools. • After a security incident occurs, we should identify the causes of this incident and the related event from the different log files.

Table 3.4. Problems and required expertise to accomplish software-related tasks

3.5. AI-based improvement opportunities

In section 3.4, we identified the problems and required expertise for software-related tasks. Based on the collected information, we propose a set of improvement opportunities. These opportunities are mostly based on Artificial Intelligence techniques to either overcome the cited problems or to infer the required expertise to accomplish these tasks from past experiences and using intelligent techniques. These opportunities allow to raise the automation level, optimize the accomplishment of these tasks, and assist software specialists by the mean of recommendations or automation.

In the following sections, we detail some AI-based ideas that allow to automate, improve activities, or assist software practitioners in realizing their tasks. Besides, we depict these improvement opportunities in a graphical cartography (Appendix A) that could serve the industry and research community as a road map for the most challenging and important future research directions.

3.5.1. Data engineering

3.5.1.1. Intelligent data generator.

Generating fake data is very important and useful in software engineering. In fact, data is the fuel for most software-related tasks. For instance, testing newly developed functionalities, data pipelines, visualization modules, machine learning models, etc. require data. A naive approach that is commonly used consists of randomly generating data. However, this data could be unrealistic and inconsistent. Thus, it doesn't reflect the production environment and the real behavior of users. This might imply missing test cases that exist in the real world which might lead to errors. In some cases, we need real data to test on real test cases, to optimize the implementation and adjust it to the real world. On the other hand, there is always a tendency to restrict access to some real data due to its confidentiality.

To overcome these problems, we propose an idea that consists of generating fake data that's indistinguishable from real data. This approach will allow for overcoming data confidentiality restrictions. The idea allows generating coherent and real-like data that has the same characteristics and statistical properties of the real data. For this purpose, we suggest using Machine Learning techniques such as Auto-Encoders or Generative Adversarial Networks (GANs) to build unbiased generative models. These generative techniques are commonly used and have proved their high performance and the quality of their results.

- Generative Adversarial Networks (GANs): It's a deep neural network architecture proposed by Goodfellow et al. [**54**] in 2014. As shown in figure 3.12, this architecture is mainly composed of two components: A generator model that generates fake data from a random noise input. A discriminator model that differentiates between real and fake data.

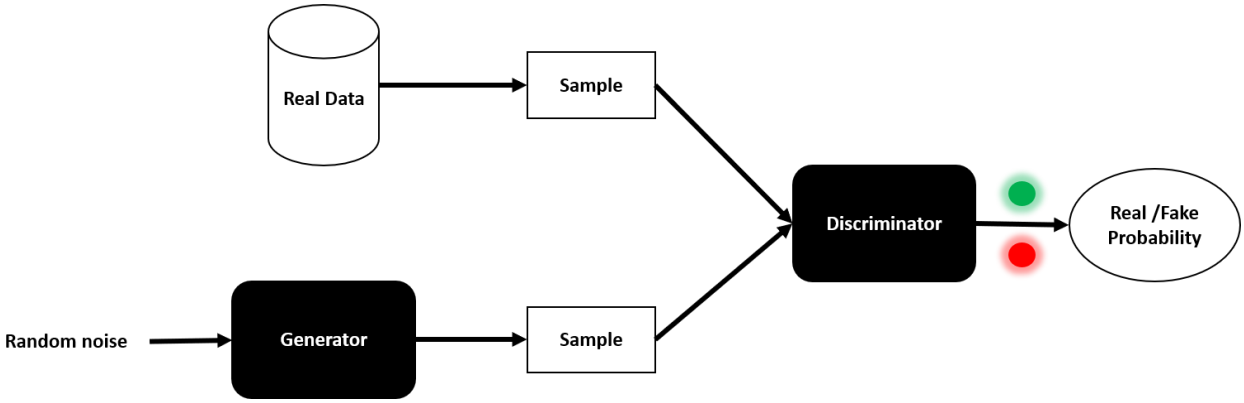


Fig. 3.12. Generative Adversarial Networks (GANs) architecture

The objective of the generator is to fool the discriminator by generating fake data that the discriminator can't distinguish from the real data. On the other side, the discriminator tries to get better at discriminating between fake and real data. The

idea is to have two models playing against each other. The two models compete and improve simultaneously over time. After some iterations, the generator is forced to create data as realistic as possible to reach its objective in generating real-like data and the discriminator improves his discrimination power.
After the training phase, we could use the generator model to produce high-quality fake data that we cannot distinguish from the real data. This data could be used for training, source code testing, testing data pipelines, visualizations, etc. Such a generator model enables simulating real test cases and allows to overcome the confidentiality restrictions.

- Autoencoders: It's an artificial neural network structure composed of two components as illustrated in figure 3.13.

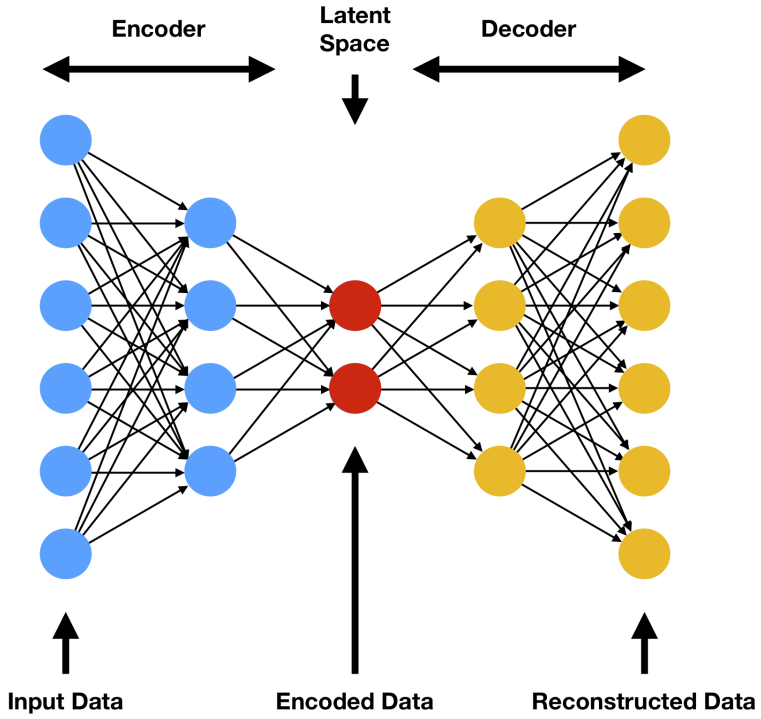


Fig. 3.13. Autoencoders architecture

The first component called an encoder, is an unsupervised model that tries to learn efficient data encoding [**84**]. The second component, called decoder, learns to reconstruct the original data from the reduced encoding representation while minimizing the reconstruction cost (the difference between original and reconstructed data).
After the training phase, we could use the decoder to generate real-like data. We feed this model with random noise and it generates data that is indistinguishable real data since the decoder has learned the statistical relationship between the latent space and the real data.

These two techniques could be used to generate fake data that is similar to the real data. This would promote the confidentiality of the real data by limiting access to real data since all tasks could use the generated data. Also, this would promote efficiency and productivity inside the team.

3.5.2. Implementation of ML models

In machine learning, some algorithms are often used (e.g. XGBoost and SVM are commonly used for regression problems) depending on some criteria: the problem type, the data structure, the desired output, etc. The Machine learning field is witnessing an increasing evolution. This great evolution gave access to a huge number of ML models. Each of which has its own strengths and limitations. It's quite difficult for a novice scientist to decide on the most suitable model to use to solve a specific problem.

To assist these junior scientists in their choice, we suggest building a Domain-Specific Language (DSL). We could use this DSL to describe the problem specifications and requirements (problem description, data structure, desired output, other constraints, etc.). Scientists would be able to use abstractions and notations from their own domains of expertise. Then, based on the input specification, this DSL suggests the most suitable ML model to be used in this specific use case.

A general and re-usable implementation of the most used algorithms in a company could also be useful. This would allow re-using the general implementation instead of building the model from scratch each time. The advantages of this are saving effort and time, ensuring high source code quality, minimize bugs proneness, accelerate the development process, etc.

3.5.3. Machine learning pipeline

The machine learning pipeline is composed of some recurrent steps: Data pre-processing $\rightarrow$ Development $\rightarrow$ Training $\rightarrow$ Validation $\rightarrow$ Evaluation.

To accelerate the generation of machine learning models and rise its automation level, we could use the Automated Machine Learning (Auto-ML) paradigm that consists of automating the whole machine learning pipeline. This approach promotes the quick generation of ML models which is very useful for generating prototypes in a short time. In fact, the standard ML pipeline is recurrent, time-consuming, resource-intensive, etc. It requires experts in several disciplines to generate high-quality deliverables. Automating the pipeline would ease the process of building machine learning models. It would incorporate machine learning best practices from experienced data scientists to make data science more accessible and simple. This allows to save both time and effort and to generate high-quality solutions without the need of having experienced and highly-skilled software practitioners.

Automated machine learning is a great change in the industry, the whole pipeline could be fully or semi-automated. To build a machine learning model that solves a certain problem, we should describe the problem specifications, then the pipeline automatically triggers the pre-processing of the data, some ML models are trained on these data, then the most performant model is selected, and finally evaluated. We don't need to go over all the ML steps, we just need to specify the problem specifications and then get the most performant ML model as output. Using automated ML pipelines enables scientists to work on more valuable and challenging tasks that require human intelligence, intuition, and creativity.

3.5.4. Evaluation of ML models

Metrics are a static way to evaluate models, however, they are commonly used to estimate the model's performance. Some dynamic behaviors might be detected using intelligent techniques. For instance, the stability of machine learning is very important. A stable machine learning model generates a similar output for similar input data.

We could evaluate the stability of ML models using adversarial learning. It consists of detecting individuals and a range of values to which the model is sensitive and unstable. These sensitive individuals are called *adversarial examples*. Adversarial training is a process that consists of injecting a small random noise to the input data that is imperceptible but the generated output is considerably different.

Therefore, there exist several adversarial defenses that allow making the ML more robust against the adversarial examples [**55, 108, 107, 12, 74, 110, 80**]. For instance, it's possible to re-train the model on the adversarial examples, increase the size of the dataset, modify the training features to eliminate the sensitive ones, use bagging by stacking several ML models to enhance the robustness and accuracy of predictions, using extra machine learning models to detect adversarial inputs, etc.

3.5.5. Monitoring

Monitoring ML models consists of detecting performance degradation and data drifts. Classic methods consist of putting threshold constraints on evaluation metrics. For instance, if the precision or recall metric falls below a certain value, an alert is raised, then scientists should re-train and adapt the ML model. However, the detection of model performance degradation could be anticipated. We could predict the model performance degradation earlier based on the evolution of the performance metrics. On the other hand, we could automate the detection of data drift rather than manually inspecting the data distribution graphs and data statistics (quartiles, mean, maximum, minimum, etc.). In fact, the detection of outliers could be automated by using some advanced ML techniques such as Isolation Trees, One-Class SVM, Logistic Regression, Autoencoders, etc. Furthermore, data drifts could be

detected by automatically comparing data distributions of the training and production data. If there is a considerable difference between the two distributions, then, we conclude that the data has changed and we should handle this issue (e.g. re-train the ML model on the new data). Another approach to avoid data and concept drifts is Continuous Learning: The ML model is trained and adapted continuously and in real-time to new data. This allows us to maintain a good performance of the ML model in production.

3.5.6. Scrum: sprint planning

Sprint planning is the main Scrum meeting in which the team decides on the tasks to be realized during the next sprint. In this meeting, tasks are assigned to different team members according to their availability, skills, task description, etc. We suggest to semi-automate the sprint planning since it could be considered as an optimization problem of resource allocation. We have a set of resources which are the team members. Each one has a certain availability, skills, etc. On the other side, we have resources which are the tasks that have some priority, complexity, and require certain qualifications.

We could use genetic algorithms, ant colony optimization algorithms [**40**], etc. to solve this problem and get some good and efficient configurations. This will allow minimizing delays in producing deliverables, non-completion of assigned tasks, affecting tasks to non-qualified persons, etc. This could contribute to raising the team's efficiency and productivity in delivering software artifacts.

3.5.7. Security

Detecting anomalies is a very challenging task for IT companies. To this end, several intelligent techniques could be used either supervised or unsupervised. For instance, we could detect new and abnormal behavior in the log files that raise suspiciousness. Besides, IT companies receive daily a high number of alerts where some of them could be false-positive alerts. For this purpose, it is interesting to use filtering or classification techniques to either eliminate the false positive and usual events or to classify these alerts according to their severity. Additionally, artificial ignorance [**26**] is useful in this context. It allows to ignore routine log messages such as regular system updates and for new or unusual messages to be detected and flagged for investigation. Furthermore, there are usually multiple records of related events in case of a security incident. It is challenging and difficult to locate this event in the logs and to identify related events. To this end, there exist some intelligent techniques called Correlation analysis techniques [**126**] that can discover connections between data in multiple log files.

Conclusion

In this chapter, we proposed our first contribution that was realized in collaboration with an industrial partner. We presented some software engineering tasks, in particular the machine learning pipeline. Then, we identified some problems related to these tasks as well as the expertise and knowledge required to accomplish these tasks. Afterward, we suggested some improvement ideas to overcome the highlighted problems, to fully or partially automate some of these tasks, and to assists software specialists in their tasks by inferring the required expertise from historical experiences. Finally, we depicted these ideas in graphical cartography (Appendix A) that could serve the industry and research community as a road map for the most challenging and important future research directions.

Chapter 4

Quality-driven multi-objective optimization approach for metamodel refactoring

Introduction

In this chapter, we present our second contribution on using AI for software development and maintenance. This contribution targets the refactoring of metamodels based on quality criteria using a multi-objective optimization approach. First of all, we state the problem. Next, we introduce our proposed approach to address this problem. Then, we present the implementation details. Finally, we present and discuss the validation results.

4.1. Problem statement & Motivations

In MDE, metamodels are considered as primary and fundamental artifacts. In fact, metamodels are used to represent and abstract domain concepts to build domain-specific languages. They also represent the backbone of transformations, models, domain-specific applications, etc.

Given this high importance of metamodels, they should be designed carefully taking into account many quality factors such as maintainability, reusability, extendibility, etc. However, considering the facts that the requirements continuously evolve, and that the domain environment is variable and continuously changing, continuous changes to the metamodels are necessary to maintain their consistency. These continuous changes may weaken the quality of the metamodel by making it unnecessarily less understandable and maintainable and more complex leading to significantly reduce productivity, increase fault-proneness and make the maintenance far more costly [**132**]. Thereby, metamodel quality should be maintained regularly as it constitutes a fundamental and strong building block in MDE.

On another note, maintenance has always been a challenging task as it is tedious, expensive and effort consuming [**57**]. Thus, its full or partial automation could alleviate the

burden on modelers. Maintaining a metamodel consists, among others, of detecting bad design choices (design smells) that are accumulated alongside the metamodel evolution and correcting them using refactoring operations. The correction of design smells (i.e. refactoring) shouldn't be applied haphazardly, though they should be performed with respect to some criteria or objectives. Indeed, the ultimate goal of applying refactoring operations is to improve the quality of the metamodel.

A smell-based refactoring approach is composed of two phases: an initial phase that detects design smells, and a correction phase that applies the corresponding refactorings to remove those smells [**21**]. Nevertheless, we do believe that removing all the design smells is not necessarily the best solution, it might not lead to the best values of quality attributes. As stated previously, the refactoring operation shouldn't be performed haphazardly but rather based on some well-defined objectives, such as improving some quality criteria. Quality attributes are potentially conflicting [**9, 59**]. Improving one quality criterion could lead to the degradation of another. Thus, a refactoring solution should find the best tradeoff with respect to quality attributes.

Besides, after the detection phase of design smells, a solution that applies all the corresponding refactorings to remove all the metamodel smells could be problematic. Indeed, refactoring a bad smell could remove another smell and thus invalid the application of its corresponding refactoring. Therefore, another objective, besides dealing with quality attributes, is to propose valid solutions.

For instance, figure 4.1 shows an examples that illustrates the case of having conflicting solution of refactorings: *MetaClassC* is a dead class and *entityName* is a duplicated feature. The order of applying refactorings matters; executing one refactoring operation could invalidate another one. To resolve the dead metaclass bad smell, we should remove that metaclass. To remove the duplicated features smell, we should apply the pull-up or create super metaclass operation. In figure 4.1, if we remove the dead metaclass *MetaclassC*, we will not be able to execute the pull up feature for *entityName*.

For instance, figure 4.1 shows an example that illustrates the case of having conflicting refactorings for two smells, dead metaclass (*MetaClassC*) and duplicated feature (*entityName* in *MetaClassB* and *MetaClassC*). If we resolve the dead metaclass smell by removing *MetaClassC*, then, we cannot apply the pull-up refactoring for *entityName* to resolve the duplicated feature smell.

In summary, an automated approach for smell-based metamodel refactoring should improve the conflicting quality attributes while generating a valid sequence of refactoring operations. In the next section, we propose an approach that allows satisfying both objectives.

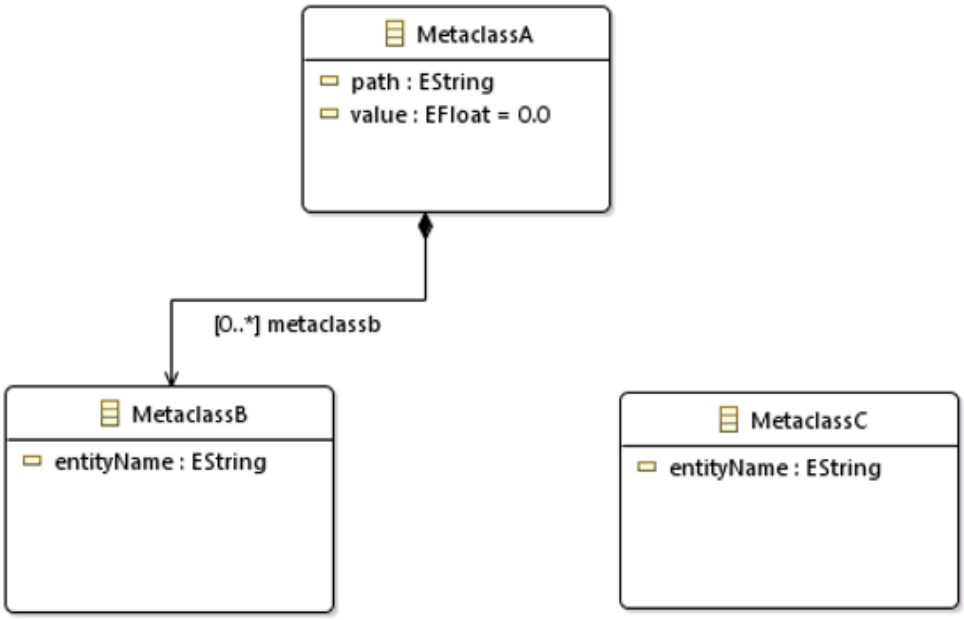


Fig. 4.1. A metamodel example that contains some design smells

4.2. Proposed approach

In this section, we first give an overview of the adopted approach. Then, we go through its different steps with more details.

our approach is based on the problem that in practice quality attributes are conflicting and the order of applying refactorings is important to generate valid recommendations. Figure 4.2 gives an overview of our automated approach to recommend metamodel refactorings. The process is based on bad smells detection and resolution. We first detect meta-model smells that exist in our input metamodel. Then, we try to resolve these smells with refactorings in a way that maximizes the metamodel quality using a multi-objective optimization approach. Afterward, we propose the Pareto-optimal solutions to the modeler. Once he chooses the solution that fits his preferences, we automatically apply the refactoring.

Figure 4.2 gives an overview of our automated approach to recommend metamodel refactorings. Our approach is based on bad smells detection and resolution. The refactoring process is driven by the optimization of many objectives: improve quality attributes and maximize the correction of smells. Our approach is composed of several steps: ❶ we define a quality metamodel that defines the relation between quality attributes and more low-level metrics. ❷ We create a model that conforms to the defined metamodel. We used the quality model presented in [**89**]. The model defines some quality attributes as a linear combination of lower-level design metrics. ❸ We developed a library that allows evaluating low-level design metrics on an input metamodel. The library was implemented using Edelta, a DSL for manipulating metamodels [**20**]. ❹ Next, we used the previous components to

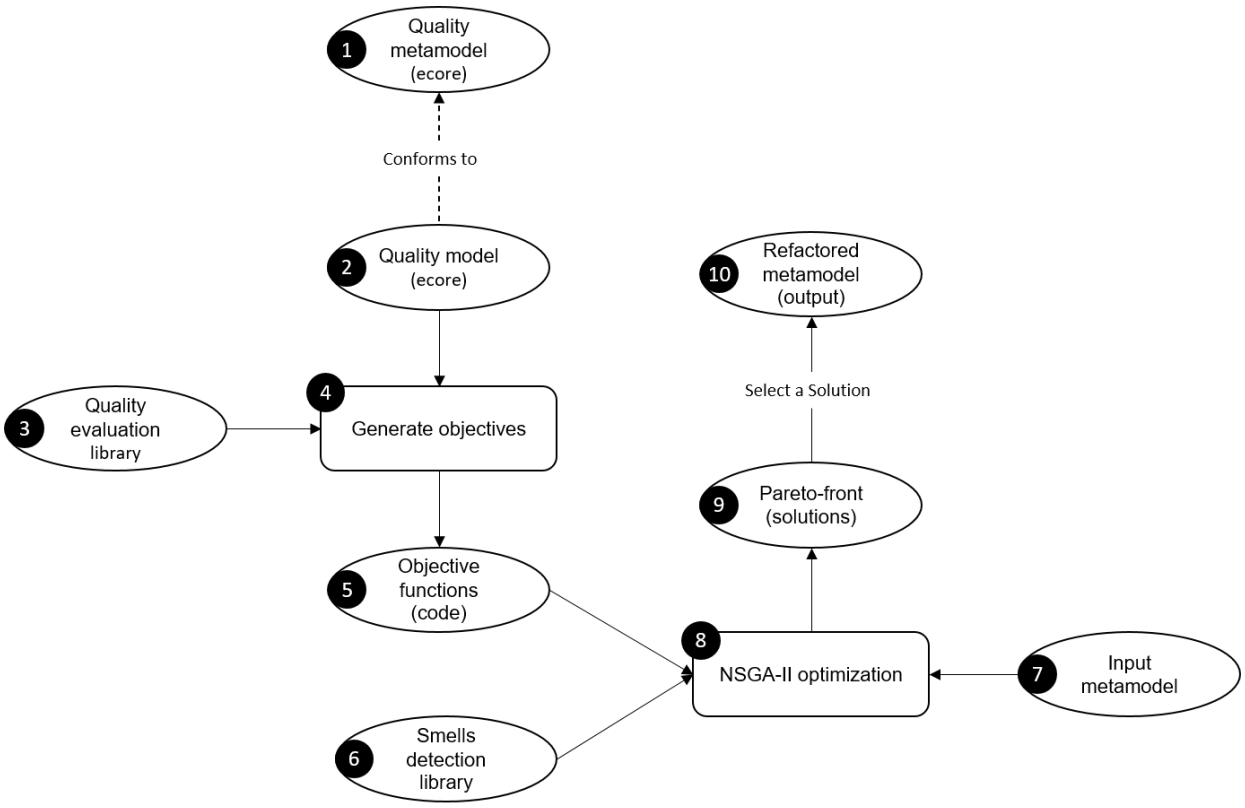


Fig. 4.2. Proposed approach

feed a source code generator implemented with Epsilon Generation Language (EGL). The generator transforms any input model, that conforms to the metamodel defined in (1), to Java source code that represents our 5 objective functions. The generated code includes the classes, attributes that represent the quality model structure as well as the methods responsible for automatically evaluating the quality attributes on an input metamodel. 6 We created a library for metamodel smells detection using Edelta. 8 We feed the previous components to NSGA-II, a search-based genetic algorithm, to optimize out objective functions (improve quality attributes an maximize the correction of smells) for 7 an input metamodel. We try to resolve the detected smells with refactorings in a way that maximizes the metamodel quality using a multi-objective optimization approach. 9 Afterward, we propose the Pareto-optimal solutions to the modeler. 10 Once he chooses the solution that fits his preferences, we automatically apply the refactoring.

4.2.1. Quality metamodel

We aim to generalize and automate our proposed framework to any quality model conforming to a certain quality metamodel. This allows us to easily make changes on the quality objectives (i.e. the way we evaluate the quality of an input metamodel). Therefore,

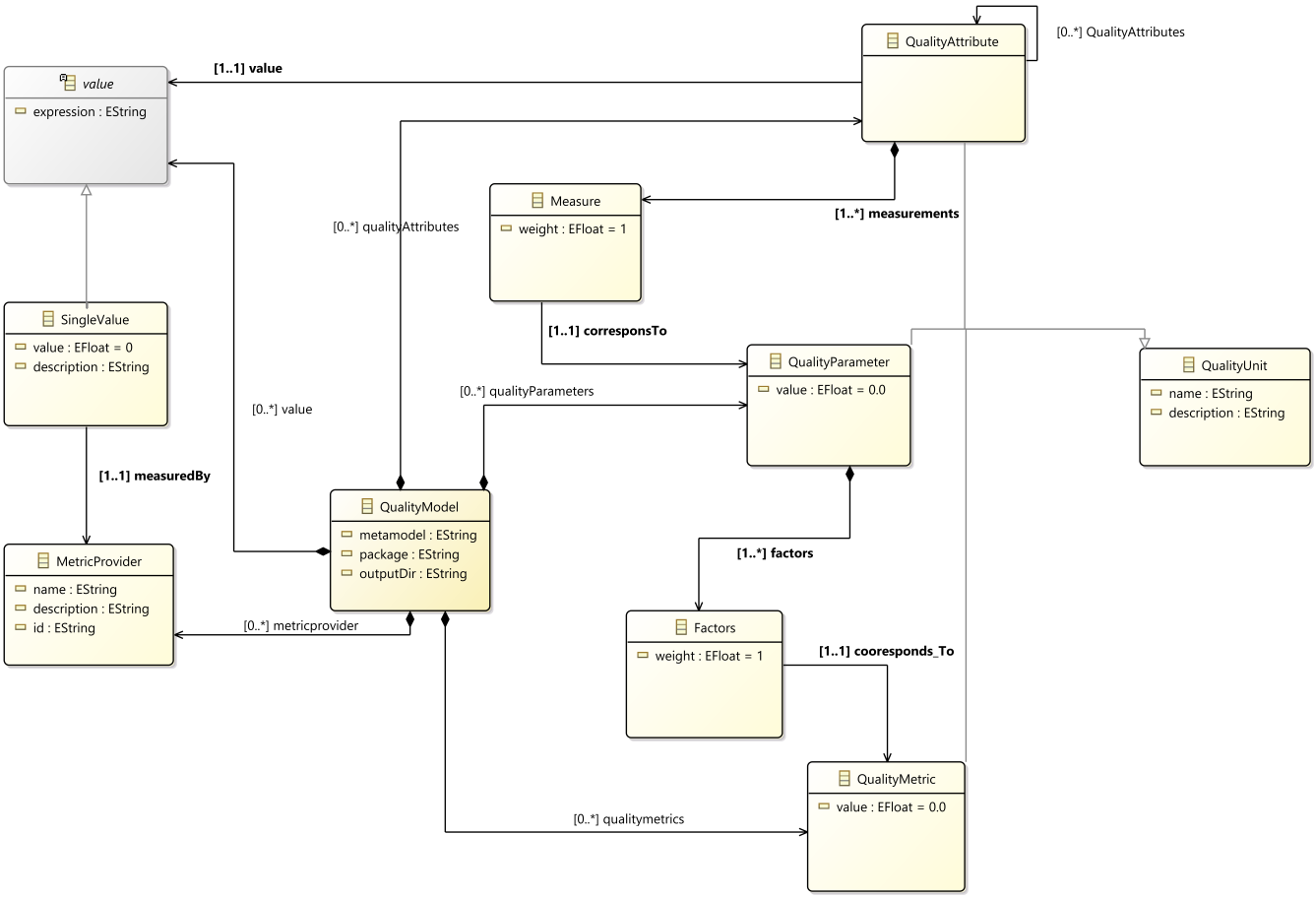


Fig. 4.3. Quality metamodel

we could modify the quality model, though, we don't need to make any changes for the other components of our approach.

To define a quality metamodel, we took inspiration from [**16**] and we augmented it based on the work done in [**89**]. The latter proposed a measuring mechanism for effectively assessing the quality of metamodels. This metamodel permits to define quality using three levels of abstraction with mappings from high-level to low-level.

A quality model, as shown in Figure 4.3 is composed of a set of quality attributes that have the highest level of abstraction (i.e understandability, maintainability, extendibility, etc.). These attributes are easy to interpret but they are not trivial to measure. Thus, they were considered by Ma et al. [**89**] as a combination of quality parameters (e.g., size, coupling, and inheritance) which are measured employing quality metrics (e.g., number of metaclasses, number of abstract metaclasses, and number of hierarchies) that have the lowest level of abstraction. The quality metrics are computed directly on the input metamodel, but their interpretation is far from trivial. Additionally, a quality attribute can be broken down into other attributes, e.g., maintainability can be defined in terms of changeability and

Table 4.1. Description of quality metrics [**89**]

Metric	Description
NOH	This metric value is the number of metaclass inheritance hierarchies in a metamodel
ADI	This metric value signifies the average depth number of metaclass inheritance hierarchies in a metamodel
ANA	This metric value signifies the average number of metaclass from which a metaclass directly inherits
ANDM	This metric value signifies the average number of metaclass with which a metaclass directly associates
ANM	This metric value signifies the average number of metaattributes of a metaclass
ANMC	This metric value signifies the average number of metacombinations of a metaclass
ANR	This metric value signifies the average number of well-formed rules of a metaclass
NAM	This metric value is the number of abstract metaclasses in a metamodel
NCM	This metric value is the number of concrete metaclasses in a metamodel

Table 4.2. Mapping from quality metrics to quality parameters [**89**]

Metric	Description
Modeling concept size (MCS)	NCM
Hierarchy	NOH
Coupling	ANDM+ANA
Intension	ANM+ANR+ANMC
Inheritance	ADI
Abstract metaclass size (AMS)	NAM

modularity. In Figure 4.3, MetricProvider is responsible for going through the structure of the metamodel to evaluate its quality attributes as a combination of lower metrics.

4.2.2. Quality model

Having defined our quality metamodel, we instantiated a quality model. Nevertheless, our approach is general and works for any model conforming to the metamodel defined in section 4.2.1. We have used the quality model proposed by Ma et al. [**89**] and expressed it using the defined quality metamodel. Ma et al. [**89**] defined a set of quality metrics as shown in Table 4.1. Next, they mapped them into quality parameters (Table 4.2) and, then, defined the measure of each quality attribute as a linear combination of some quality parameters (Table 4.3).

This linear combination was defined based on the negative or positive influence of each quality parameter on each quality attribute and the weight of this influence. Our metamodel was defined using ecore. We have created the quality model that conforms to this metamodel

Table 4.3. Weights: influence of quality parameters on quality attributes [**89**]

Quality attribute	MCS	Coupling	Intension	Inheritance	AMS	Hierarchy
Reusability	0.3	–0.3	0.8		0.3	
Understandability	–0.1	–0.2	0.7	–0.1	–0.1	–0.2
Functionality	0.2	0.4	0.4			
Extendibility	0.3	–0.2			0.3	
Well-structured		–0.2	0.8			–0.1

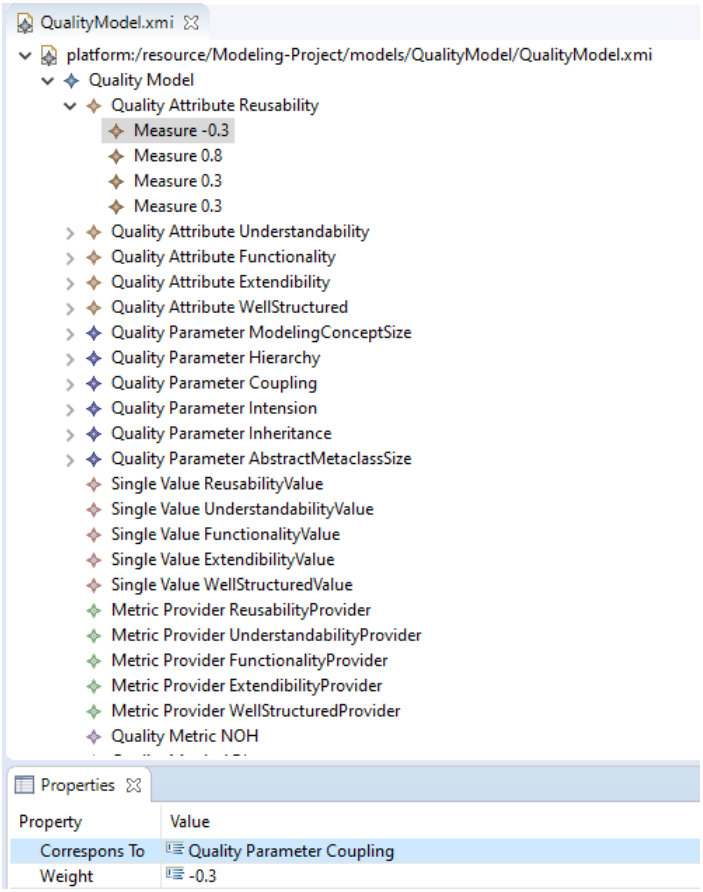


Fig. 4.4. Quality model

and that contains the quality attributes definition, as shown in figure 4.4. These quality attributes represent the objective functions that we tend to maximize when searching for a refactoring solution.

4.2.3. Objectives generation

As shown in figure 4.2, the quality model serves to automatically generate the code of the objective functions to evaluate the metamodel quality when searching for refactoring solutions. For the code generation, we have used Epsilon Generation Language (EGL) to implement the transformations that produce the Java code for our defined objectives.

To this end, we have defined four EGL transformations.

- With the first transformation, we generated a Java class for each quality metric.
- The second transformation was dedicated to generating a Java class for each quality parameter.
- We wrote the third transformation to generate Java code classes for quality attributes.
- The last transformation was dedicated to the generation of a class that contains the whole quality model and that defines the different linear combinations between the quality units.

```
rule DesignMetrics2JavaCode
  transform self : M!QualityModel {
  template : "QualityModel2JavaCode.egl"
  target : "../../src/QualityModel/QualityModel.java"
}

public class QualityModel
{
    //path to the input metamodel
    private String metamodel = "[%=self.metamodel%]";
    //package of the metamodel
    private String packageName = "[%=self.package%]";
    //directory to save output metamodels
    private String outputDir = "[%=self.outputDir%]";
    private Map<String, QualityAttribute> qualityAttributes =
  ↪ new HashMap<>();
    private Map<String, QualityParameter> qualityParameters=
  ↪ new HashMap<>();
    private Map<String, QualityMetric> qualityMetrics = new
  ↪ HashMap<>();

    public QualityModel()
    {
        //Instantiate Quality Metrics
        [%for(qm in self.qualitymetrics){%]
            qualityMetrics.put("[%=qm.name%]", new [%=qm.name
  ↪ %]());
        [%}%]
```

```
        //Instantiate Quality Parameters
        [%for(qp in self.qualityParameters){%]
            qualityParameters.put("[%=qp.name%]",new [%=qp.name
↪ %](
            new ArrayList<QualityMetric>() {
            {
                [%for(f in qp.factors){%]
                add(qualityMetrics.get("[%=f.cooresponds_To.
↪ name%]"));
                [%}%]
            }},
            new ArrayList<Double>() {
            {
                [%for(f in qp.factors){%]
                add([%=f.weight%]);
                [%}%]
            }}));
        [%}%]

        //Instantiate Quality Attributes
                                    .
                                    .
                                    .
```

Listing 4.1. Quality model to Java code: EGL transformation - code snippet

Listing 4.1 shows a code snippet of the fourth EGL transformation where an instance for each quality unit is created and the relations between them are defined. These transformations allow saving time and effort to write the java code representing the objectives. Whenever the model changes (weights and quality metrics are modified), we do not need to rewrite everything, we need to just adapt the model and automatically regenerate all the source code for objective functions.

4.2.4. Detection of metamodel smells

We implemented the detection of metamodel smells using a library that allows the search for five types of metamodel bad smells (Table 4.4). This library is described by Bettini et al. [**21**] and is implemented using Edelta, a domain-specific language [**20**].

Table 4.4. List of considered bad smells [**21**]

Bad smell	Description
Duplicated features in metaclasses	A feature (attribute or reference) that is present in different metaclasses (with the same type and properties) which may imply duplication of information might be induced.
Dead metaclass	A metaclass completely disconnected from the other elements of the metamodel (similar situations are referred to as dead code or oxbow code [**37**]). One of the possible refactoring operations that can be applied in these cases is the removal of the dead metaclass (even though such an action should be confirmed by the modeler)
Redundant container relation	Sometimes, it may be necessary to traverse containment relations from the elements to the containing ones. In EMF, the general and implicit eContainer reference is available. The presence of such a smell has several negative implications. In particular, it introduces redundancy, since the implicit eContainer reference is always present.
Classification by enumeration or by hierarchy	As discussed in [**124**] model elements can be classified using enumeration or by hierarchies. i.e. if a subclass doesn't add new features, it is useless to make the design bigger without any additional information. We can just an enumeration instead.
Concrete abstract metaclass	Depending on the particular situation being modeled, it can happen to have the superclass of a given class hierarchy being specified as concrete instead of abstract. If such a smell occurs, it can have negative impacts on the understandability of the considered metamodel and it may lead to erroneous situations.

4.2.5. Applying refactorings

Bettini et al. defined a weaving model to match every bad smell to the refactoring that allows to remove it [**21**]. Edelta was also used to define a library for refactorings to resolve each of the bad smells We have adapted this library to our particular context to apply a refactoring solution found and produce the refactored metamodel from the input metamodel. The refactoring application consists of resolving the selected bad smells in the solution.

4.2.6. Metamodel quality evaluation

To evaluate the quality of a metamodel, we adopted the quality model introduced in section 4.2.2 that has three levels of abstraction. First, we compute the quality metrics (Table 4.1) directly on the metamodel. Then, we use linear combinations to compute respectively the quality parameters (Table 4.2) and the quality attributes (Table 4.3). To do so, we have also used Edelta to build our own module to compute the quality metrics. A code snippet of the module that we have developed to compute the quality metrics is shown in figure 4.5.

For the first metric, Number of Abstract Metaclasses (NAM), we iterate over all the metaclasses and filter them to keep just the abstract metaclasses, then, we return the size

```
def computeNAM(EPackage epackage){

    return epackage.allEClasses.iterator.filter[c | c.abstract].size
}

def computeNCM(EPackage epackage){

    return epackage.allEClasses.iterator.filter[c | !c.abstract].size
}
```

Fig. 4.5. Code snippet of the Edelta library for computing quality metrics

of the filtered list. We do the same for the Number of Concrete Metaclasses (NCM) metric where we filter the metaclasses to just keep the concrete metaclasses.

4.2.7. Multi-objective optimization of quality attributes using NSGA-II

Having prepared all the different pieces of detecting smells, applying refactorings, computing quality, etc., we reach the core of our work which is the optimization part. Our ultimate goal is to decide what smells to remove in order to maximize the conflicting quality attributes. During this multi-objective optimization process, we make sure that the produced refactoring solutions are valid and applicable.

If the search space is small, i.e., small metamodel with a few smells, we can just use an exhaustive search that consists of exploring all the search space by testing all possible combinations and select the best ones. In this case, we are sure that the Pareto front we get is the globally optimal one. However, when it comes to large search spaces, it is difficult, if not impossible, to perform an exhaustive search considering the very high number of solutions to explore. In this case, a heuristic search allows reducing the search space by exploring fewer solutions and can find near-optimal solutions.

A popular way to perform a heuristic search is to use meta-heuristic algorithms like genetic algorithms that have proven to be efficient for many optimization problems. In our case, we select the appropriate search method depending on the search space size. We first evaluate the size of the search space depending on the problem parameters, i.e., the size of the metamodel and the number of detected smells. If an exhaustive search is doable in a reasonable timeframe, then, we use it. Otherwise, we use the multi-objective genetic algorithm NSGA-II to intelligently explore the space of potential solutions.

4.2.7.1. Problem formulation.

In this section, we formalize our problem by defining the objectives and the solution structure:

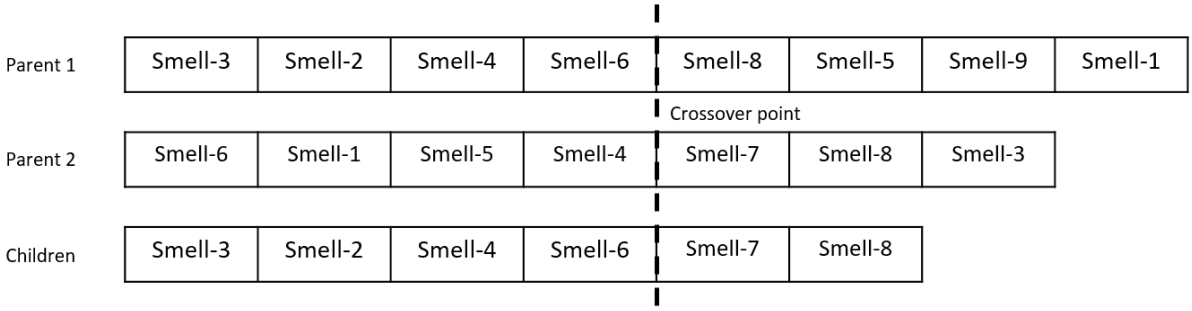


Fig. 4.6. Adapted single-point crossover operator

(a) *Objectives definition:* We have five objectives which are the quality attributes (Table 4.3) defined in our quality model as well as maximizing the number of corrected bad smells. Our goal is to optimize these conflicting objectives.

(b) *Solution representation:* Our solution is a set of bad smells to be removed. Our goal is to decide on the smells to be removed in order to optimize the objective functions on the input metamodel. For each bad smell, we associate the corresponding refactoring allowing us to remove it, therefore, we can also consider our solution as a set of refactorings. The solutions have different sizes, each solution is a subset of the complete list of smells initially detected.

4.2.7.2. Genetic operators.

(a) *Crossover* We have used single-point crossover and we adapted it to our context. A single point crossover consists of taking two parents from the existing population, choosing randomly a crossover point, and producing a new child that combines the left side of the first parent and the right side of the second parent relatively to this point. One problem that may occur is having a new child individual with duplicated bad smell instances. This case occurs if the left side of the first parent contains the same bad smell instance as the right side of the second parent. To overcome this problem, we adapted the crossover operator by removing duplications in the new individual. Figure 4.6 illustrates an example. The crossover operator is applied on *Parent1* and *Parent2*. The left side of the first parent and right side of the second one, relatively to the crossover point, contain the same bad smell instance which is *smell-3*. So, we apply the standard single-point crossover operator and we check for duplication. Then, we just keep one instance in the produced children.

(b) *Mutation* (a) We defined our crossover operator as following: we go through the parent individual elements, and for each element, we decide to keep it or not according to the mutation rate (chosen in the initialization phase of the genetic algorithm). This means, for every solution element (i.e. a bad smell instance) and according to mutation rate, we decide to keep that element or to remove it. Figure 4.7 shows an

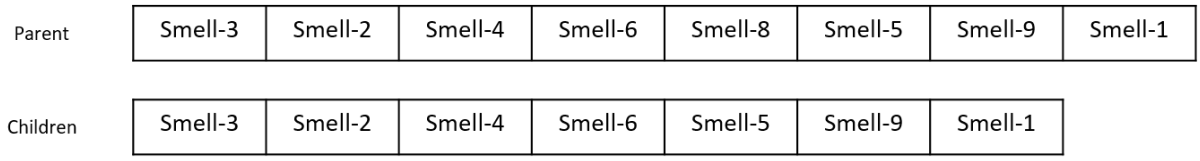


Fig. 4.7. Adapted mutation operator

example, where the application of the mutation operator on the parent individual led to removing the element *Smell-5* and keep others.

4.2.7.3. Solution evaluation.

A solution is a set of smells to be removed. To evaluate a solution, we translate it to a list of refactorings. This is done by replacing each smell with its corresponding refactoring operation that allows to remove it. Then, we apply these refactorings on the input metamodel, we produce the refactored metamodel the refactoring library (section 4.2.5). Afterward, we evaluate the quality of the output metamodel by computing its quality attributes using our evaluation module (section 4.2.6).

4.2.7.4. Solution validity assurance.

A solution is valid if their corresponding refactorings are not conflicting. This means the smells of this solution could be resolved without any problem. Sometimes, applying one refactoring by resolving one smell implies the vanishment of another smell. We try to resolve this by using a trial and error approach. We apply the solution, if a problem occurs, we just eliminate this invalid solution. We just keep the solutions that are valid to avoid misleading the modeler when recommending solutions.

4.2.7.5. Metamodel refactoring recommendation.

The execution of the optimization process produces a Pareto front of non-dominated solutions. These solutions are equally good, therefore, it is on the modeler to select the most appropriate solution that suits his preferences. We give further assistance to the modeler through some charts representing the solutions and their quality (Figure 4.9).

4.3. Illustrative case study

To validate our approach, we conducted a set of experiments using a dataset of 10 case studies selected from an online Ecore metamodel dataset [1]. Table 4.5 summarizes the evaluated metamodels.

Figure 4.8 shows a metamodel example for Customer Relationship Management (CRM), in which we have manually introduced some bad smells:

- Dead metaclass *LocatedElement*.

[1] `https://zenodo.org/record/2585456#.X4w0sNBKhnL`

Table 4.5. Description of the metamodels

	Name	NMC[i]	NMA[ii]	NA[iii]	NH[iv]	NDS[v]
1	CRM	11	6	8	5	6
2	DBLP	17	51	17	8	13
3	BibTex	29	55	4	48	17
4	Ant	51	95	28	40	25
5	Maven	8	28	6	2	3
6	jPQL	49	24	42	35	20
7	HTML	59	98	14	42	38
8	SQL	92	45	119	42	42
9	WikiML	29	10	22	24	12
10	MongoSQL	23	21	6	18	8

[i] Number of MetaClasses [ii] Number of MetaAttributes [iii] Number of Associations [iv] Number of Hiearchies [v] Number of detected smells

- Dead metaclass *Chart*.
- Duplicated feature *name* in many metaclasses.
- Redundant container relation between *CRM* and *Client*.
- Concrete abstract metaclass *Client*.
- Concrete abstract metaclass *Worker*.

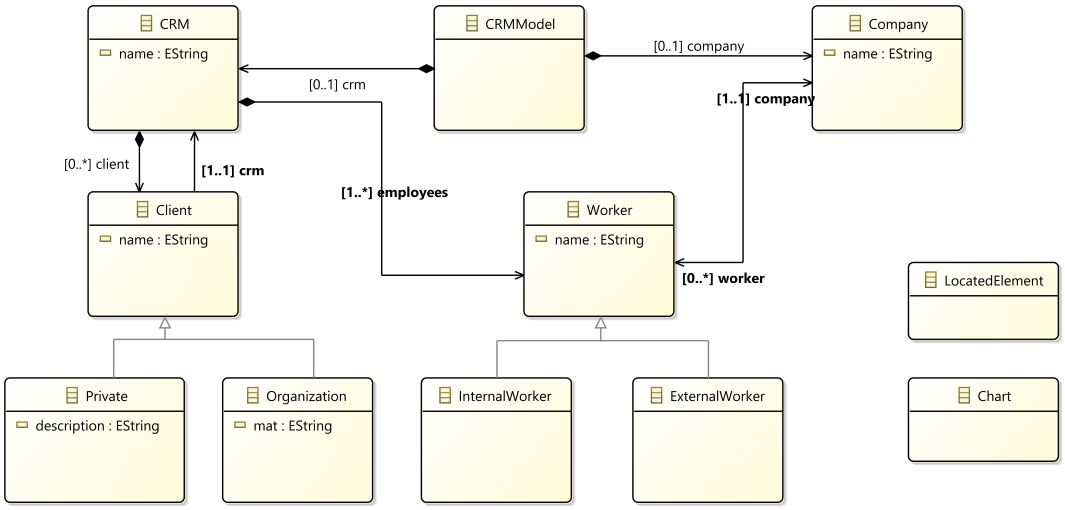


Fig. 4.8. Metamodel example for Customer Relationship Management (CRM)

In the next sections, we define research questions, describe the experimental setting and we present the results to answer our research questions.

4.3.1. Research questions

To evaluate our proposed approach, we define the following research questions:

- **RQ1:** *How does our approach perform compared to the classical approach?* To answer this research question, we compare the best solutions generated by our approach to those generated using the classical approach [**21**] that consists of removing all the design smells.
- **RQ2:** *How does our approach perform compared to the Random Search?* We answer this question by exploring the same number of solutions using random search and using our approach and we compare the best solutions from both strategies. This allows to know whether the results of our approach are attributable to the search strategy or to the number of explored solutions.

4.3.2. Experimental setup

To answer both questions, we use a dataset of metamodels (as described in table 4.5).

For the first research question RQ1, we compare our approach to the classical approach presented in [**21**] that consists of removing all the design smells.

In the second research question RQ2, we prove that the achieved results are attributable to the search strategy; and not haphazardly. We implemented the random search(RS) algorithm with a uniform distribution.

Random search is a direct optimization method that doesn't require a search strategy to generate solutions [**27**]. We use a uniform distribution to build our solution from the list of detected design smells (the probability of a smell to be removed from an input metamodel is 0.5). We run a number of fixed iteration. For each iteration, we build a new random solution and we only keep the top solution. An intelligent algorithm is usually compared to random search to check if it performs better or worse than chance.

4.3.3. Results

We executed our approach on the 10 metamodels and compared the results to RS and Classical approach outputs. We manually fixed the values of hyperparameters based on previous runs:

- Number of iterations: 20
- Population size: 5
- Mutation rate: 0.1

Results for RQ1: Comparison between our approach and the classical approach.

For RQ1, our approach outperformed the classic approach proposed by Bettini et al. [**21**] for all the metamodels dataset. The classic solution is generated with a two-step approach that consists of detecting the design smells of a given metamodel and then removing all the smells. On the contrary, our approach was based on the problem that in practice quality

attributes are conflicting and the order of applying refactorings is important to generate valid recommendations.

In figure 4.9, we compare the non-dominated solutions generated by our search-based approach with the best solution produced by a RS method for the CRM metamodel (Figure 4.8).

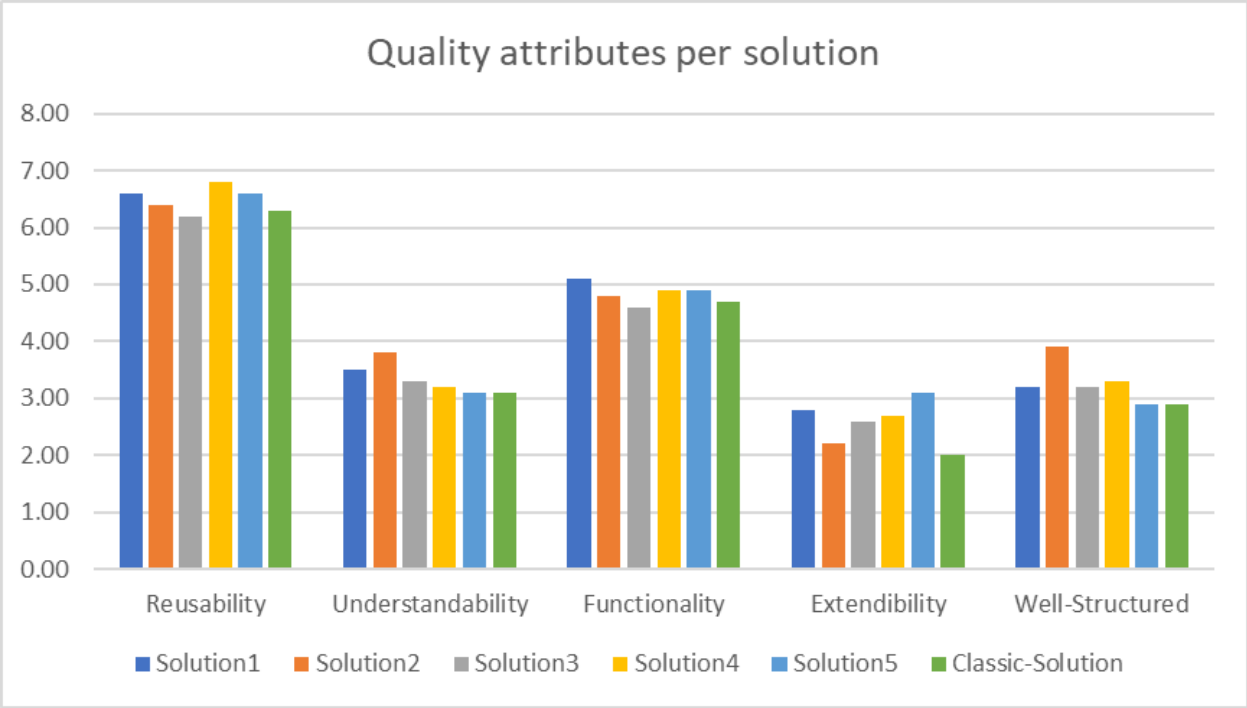


Fig. 4.9. CRM metamodel - Comparison between our approach and classical approach

As shown in figure 4.9, many non-dominated solutions outperform the classic solution. For instance, *Solution1* dominates the classic solution as it has better value for all the quality attributes. This justifies the usefulness of using a search-based algorithm to find near-optimal solutions. The results of RQ2 prove that our approach is better than the classical method by generating solutions with better quality.

Results for RQ2: Comparison between our approach and Random Search (RS).

For RQ2, our approach outperformed RS for all the input metamodels. In figure 4.10, we take the CRM metamodel as an example and compare the non-dominated solutions generated by our approach to the best solution produced by the RS algorithm.

For instance, *Solution1* dominates the random solution in all the quality attributes. The results of RQ2 confirm that our approach didn't reach the final solutions by chance but rather it is capable of generating good recommendations.

We have selected *Solution-2*, then, our framework generated the refactoring metamodel corresponding to this solution. Figure 4.11 shows the refactored metamodel corresponding to the second solution.

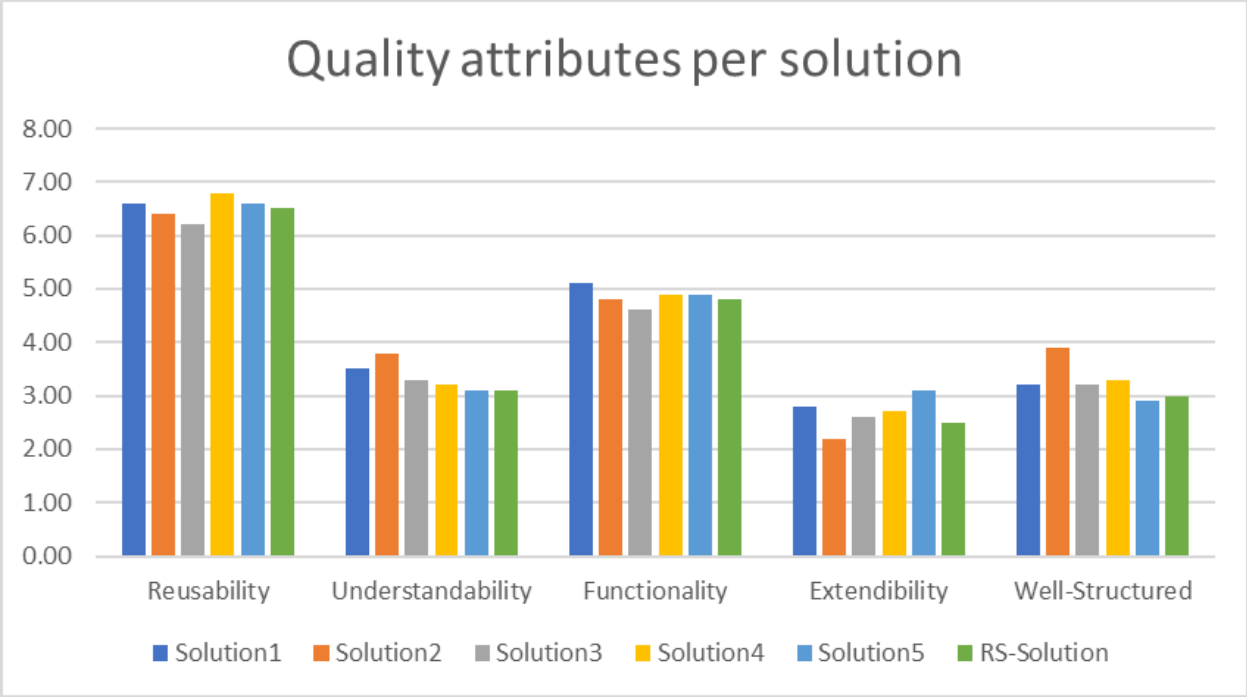


Fig. 4.10. CRM metamodel - Comparison between our approach and RS

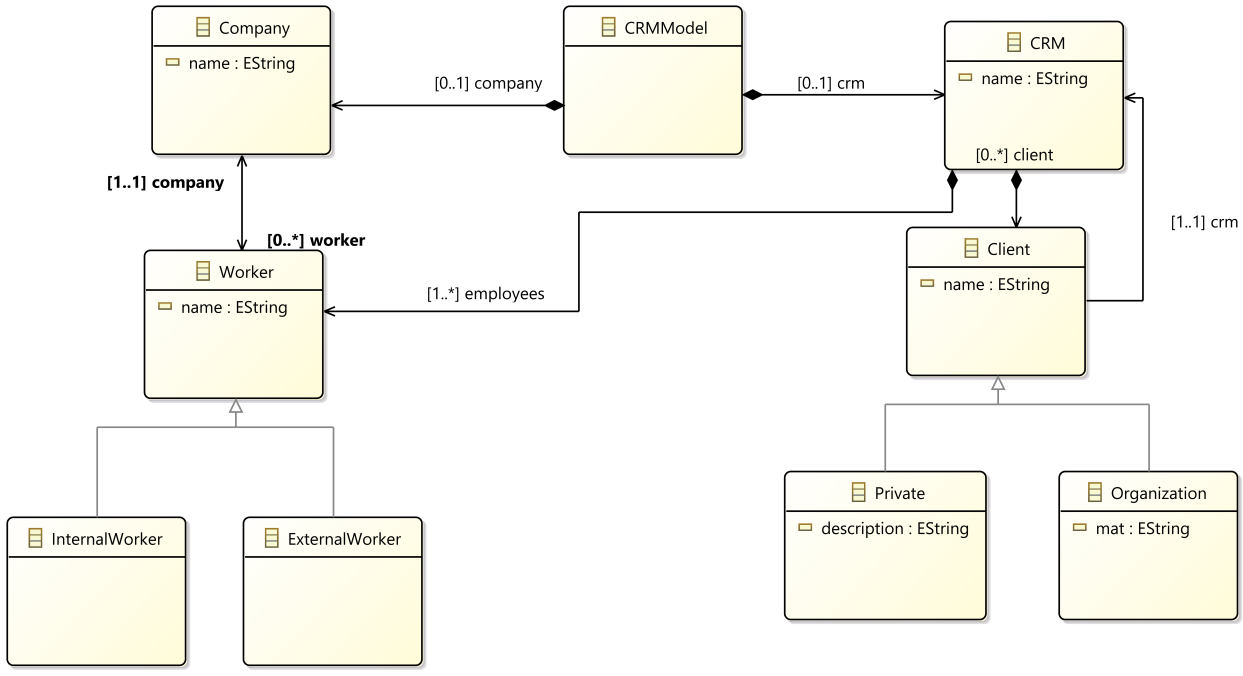


Fig. 4.11. Refactored metamodel corresponding to solution-2

Solution-2 has the best value in terms of understandability, and this can justify why the two dead metaclasses were removed. By examining the refactored metamodels corresponding to the other solutions, we could explain that *solution-2* has lower value Reusability and

Extendibility because unlike other solutions the bad smell duplicated feature (*name*) has not been removed. This might justify why this solution has worse values for reusability and extendibility.

4.3.4. Validity of the experiments

Our approach is general and could be applied to any input metamodel. We used metamodels with different properties, and we got good results using our approach. Therefore, our approach is probably able to generate good solutions for other DSLs.

We were able to provide relevant and diversified non-dominated refactoring solutions and leave the final choice to the decision-maker.

However, some threats to validity concern the used quality model which is arguable especially regarding the numeric weights. However, the used quality model was just a way to prove the validity of the problem as well the realization, feasibility, and utility of our approach. Our approach is generic and could use any kind of defined quality model.

Concerning the validation of our approach, we plan to expand it more by leading an empirical study to evaluate modelers' feedback (comprehension, ability to add new features, simplicity to work with, etc.) regarding the refactored metamodels to prove the relevance of the recommended solutions.

4.4. Conclusion & Future work

In this contribution, we have defined the improvement of metamodels' quality as an optimization problem. We proposed an automated approach to recommend metamodel refactorings using a search-based optimization process of many objectives. The goal is to produce a set of refactoring operations that are not conflicting, that improve a set of quality attributes, and maximize the number of bad smells to remove. Our approach provides a variety of good and valid refactoring solutions that let the modeler choose the most convenient solution based on his preferences. The provided framework allows the modeler to give his metamodel as input, then the optimization process is automatically executed to generate visualizations that highlight the characteristics of the different solutions. Once the modeler selects a solution, the refactored metamodel is automatically generated. We validated our approach on a set of 10 metamodels and we compared it to the classical approach and RS.

In future work, we plan to make the optimization process dynamic by taking into consideration the fact that applying refactorings may introduce new bad smells. Also, we aim to expand more our approach so it takes into consideration the predefined constraints (e.g. OCL constraints) of the metamodel to produce a valid refactored metamodel that satisfies these constraints. We plan to validate our approach with an empirical study to evaluate modelers'

feedback (comprehension, ability to add new features, simplicity to work with the initial and refactored metamodel, etc.) to prove the relevance of the recommended refactoring solutions.

In the next chapter, we propose a formal method based on Alloy, a bounded constraint solver, to refactor metamodels.

Chapter 5

Metamodel refactoring using constraint solving

Introduction

In this chapter, we introduce our third contribution on using AI techniques for software maintenance. We use constraint solving to recommend metamodel refactorings. First, we state the problem related to the maintenance of metamodels. Second, we propose an approach to solve this problem. Third, we present the implementation details. Finally, we conclude this chapter with an illustrative case study.

5.1. Problem statement

In MDE, metamodels represent the backbone of building domain-specific languages, transformations, etc. As highlighted in chapter 4, we should maintain a good quality of these metamodels.

In chapter 4, we used a multi-objective optimization approach to refactor metamodels using NSGA-II. However, Evolutionary Algorithms (EAs) have some drawbacks:

- EAs might converge to a local extremum and not lead to a point close to the global maximum.
- EAs are non-deterministic methods. Each execution on the same instance might lead to different results.
- The quality of results depends on values of parameters (e.g. population size, generation count, selection size, crossover and mutation rates, etc.).
- EAs are sensitive to the initial population.
- The genetic operators (crossover, selection, mutation) might have an impact on the final results.

- EAs involve a lot of computations, so, they might take a lot of time to converge to a good solution.
- EAs do not guarantee optimality. They often lead to good results (i.e non-dominated solutions).
- The solutions produced by EAs are not comprehensible. We cannot explain why EAs led to these results or know if the obtained solutions are good.

To overcome some of the aforementioned limitations, we tackle the problem of metamodel refactoring using another technique: Constraint Solving (CS). CS is an AI method that allows solving logical constraints and that is being widely used in many applications [**49**]. Unlike EAs, CS leads to an optimal solution that satisfies all the specified constraints. Furthermore, CS allows focusing on the high-level modeling of the problem rather than on specifying how to solve it. The results of CS are explainable, if a constraint solver fails to find a solution, we can know which constraint is violated, otherwise, all the constraints are satisfied. Furthermore, constraint solvers are efficient. They automatically make simplifications by pruning off all branches that violate constraints.

We use Alloy, a constraint solver, to encode the problem and solve constraints. We detail this approach in the next section (section 5.2).

5.2. Proposed approach

Alloy is a formal constraint specification language based on first-order logic. It allows to formally express the structural properties and behavior of a system. Alloy provides a lightweight modeling tool to express and check system properties [**71**]. It is designed to perform a bounded scope analysis by checking the encoded specification over a finite number of instances.

To tackle the problem presented in section 5.1, we propose a formal approach to refactor metamodels using Alloy. Rather than using a meta-heuristic with multi-objective optimization, we rely on a constraint solving method. We use Alloy as a specification language to express the quality criteria and the absence of metamodel smells as constraints. Then, we try to satisfy these constraints using Alloy as a constraint solver. The solutions that satisfy the specified constraints are the refactoring solutions that we recommend to improve the quality of the input metamodel.

Figure 5.1 depicts the proposed approach:

Our approach is composed of two parts: (1) a detection phase in which we check the non-existence of design smells (2) a refactoring phase where we remove the bad smells while satisfying some quality criteria.

In the first phase, we encode the non-existence of design smells as constraints. We run *CD2Alloy* transformation to translate the input metamodel from a class diagram to an alloy

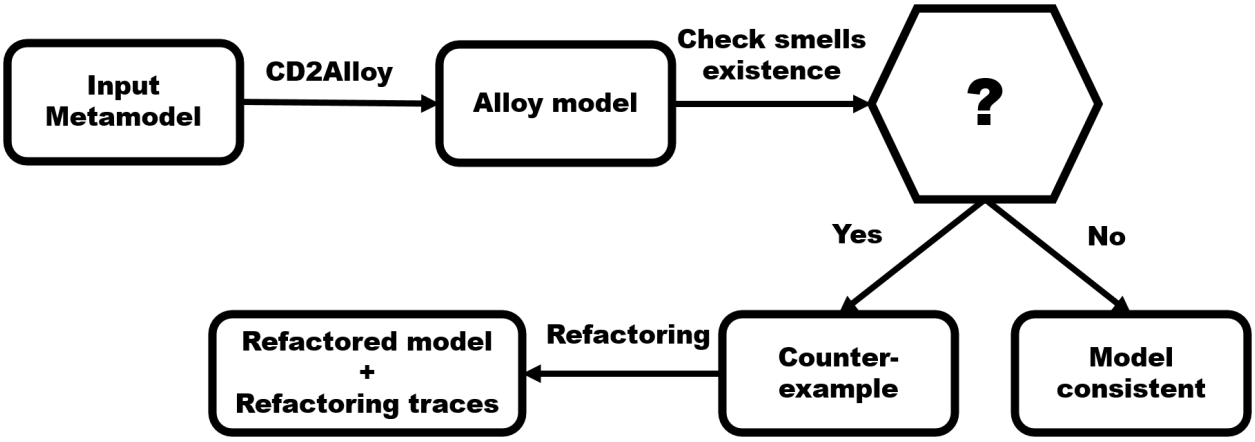


Fig. 5.1. Constraint solving approach to detect design smells and refactor metamodels

specification. Then, we use the Alloy analyzer to check the encoded constraints. If no design smell is detected, then, the metamodel is consistent. Otherwise, alloy generates a counter-example that consists of a model that contains some bad smells. After that, the refactoring phase starts.

In the second phase, we encode the non-existence of smells and some quality criteria as logical constraints. We execute the Alloy analyzer on the input metamodel to satisfy the encoded constraints and generate the refactored metamodel.

5.3. Implementation details

In this section, we present the implementation details of our approach.

5.3.1. Phase 1: Detection of design smells

In this phase, the goal is to check if an input metamodel is consistent and does not contain design smells. For this purpose, we use alloy as a modeling language to express the specifications and constraints of the input metamodel. This first phase is composed of three steps: (1) translate an input metamodel, represented as a class diagram, to alloy specification (2) encode the non-existence of design smells as constraints in Alloy (3) use alloy analyzer to check these constraints on the input metamodel.

5.3.1.1. Translate metamodel class diagram to alloy specification using CD2Alloy.

The input metamodel is a class diagram, so, the first step consists of translating it to alloy. For that, we used *CD2Alloy* that was presented by Maoz et al. in [**92**]. CD2Alloy is a translation from UML class diagrams to Alloy that is deeper than other suggested translations [**92**]. The authors provide an eclipse plugin that implements this transformation.

Figure 5.2 shows an example of a metamodel for Customer Relationship Management (CRM). This metamodel is a simplified version of the metamodel used in Chapter 4.

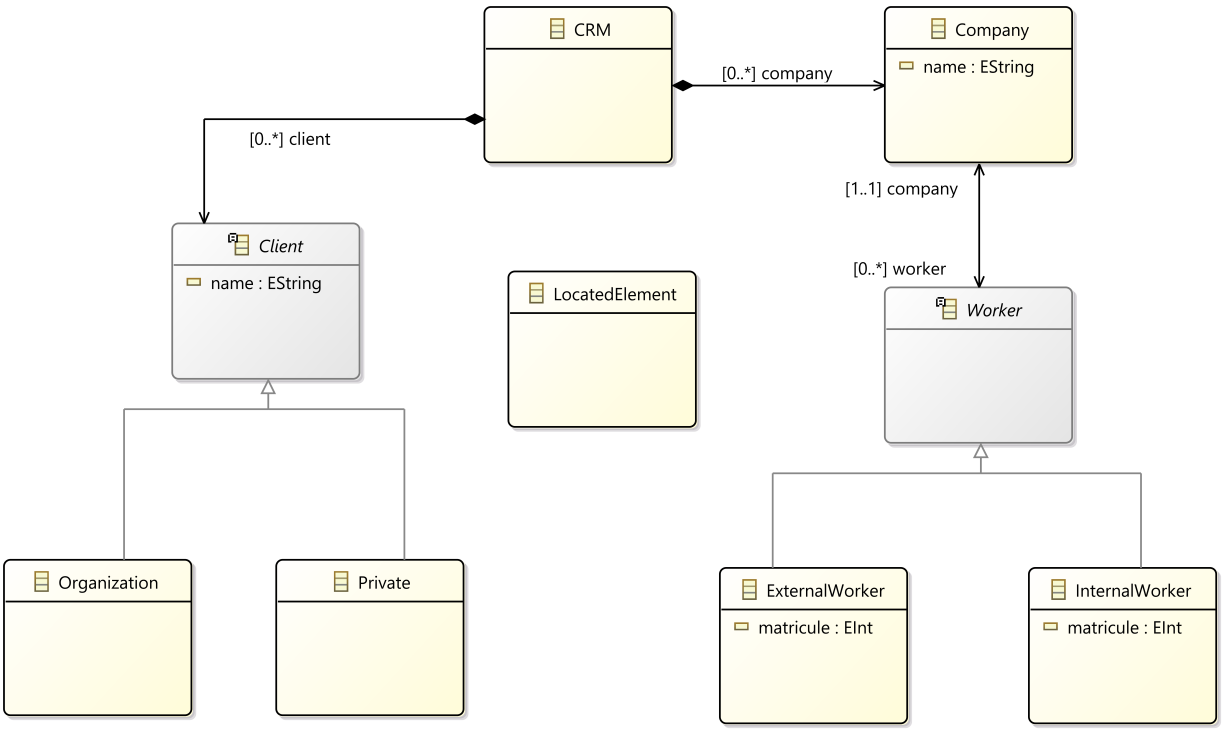


Fig. 5.2. Customer Relationship Management (CRM): class diagram

```
package CD2Alloy;

classdiagram CRM_CD {

    class CRM;
    class Company {string name;}
    abstract class Worker;
    class InternalWorker extends Worker{int matricule;}
    class ExternalWorker extends Worker{int matricule;}
    abstract class Client {string name;}
    class Organization extends Client;
    class Private extends Client;
    class LocatedElement;

    association composition1 [1] CRM (crm) -> (company) Company [0..1];
    association composition2 [1] CRM (crm) -> (client) Client [*];
    association staff [1] Company (company) -- (worker) Worker [*];
}
```

Listing 5.1. CRM class diagram representation in CD2Alloy plugin

Listing 5.1 shows a textual representation of this class diagram in the CD2Alloy framework.

A CRM represents the metamodel serves for managing the relationship between the company, its employees, and its customers. A CRM could manage many companies. A worker belongs to one and only one company while a company could have many workers

```
// Some markers not belonging to the model
one sig auxilary {}
//Names of fields/associations in classes of the model
abstract sig FName { is: one auxilary }
//Names of enum values in enums of the model
abstract sig EnumVal {}
//no enum values can exist on their own
fact enums {
    all v: EnumVal | some f: FName | v in Obj.get[f]
}
//Parent of all classes relating fields and values
abstract sig Obj {
    get : FName -> { Obj + Val + EnumVal }
}
//Values of fields
abstract sig Val {}
//no values can exist on their own
fact values {
    all v: Val | some f: FName | v in Obj.get[f]
}
pred ObjFNames[objs: set Obj, fNames:set  FName] {
    no objs.get[FName - fNames]
}
pred ObjAttrib[objs: set Obj, fName:one FName, fType: set {Obj + Val + EnumVal}] {
    objs.get[fName] in fType
    all o: objs | one o.get[fName]
}
pred ObjMeth[objs: set Obj, fName: one FName, fType: set {Obj + Val + EnumVal}] {
    objs.get[fName] in fType
    all o: objs | one o.get[fName]
}
pred ObjLUAttrib[objs:set Obj, fName: one FName, fType:set Obj, low: Int, up: Int]
    ↪  {
    ObjLAttrib[objs, fName, fType, low] and ObjUAttrib[objs, fName, fType, up]
}
pred ObjLAttrib[objs: set Obj, fName:one FName, fType: set Obj, low: Int] {
    objs.get[fName] in fType
    all o: objs | (#o.get[fName] ≥ low)
}
pred ObjUAttrib[objs: set Obj, fName:one FName, fType: set Obj, up: Int] {
    objs.get[fName] in fType
    all o: objs | (#o.get[fName] =< up)
}
pred ObjLU[objs: set Obj, fName:one FName, fType: set Obj, low: Int, up: Int] {
    ObjL[objs, fName, fType, low] and ObjU[objs, fName, fType, up]
}
pred ObjL[objs: set Obj, fName:one FName, fType: set Obj, low: Int] {
    all r: objs | # { l: fType | r in l.get[fName]} ≥ low
}
pred ObjU[objs: set Obj, fName:one FName, fType: set Obj, up: Int] {
    all r: objs | # { l: fType | r in l.get[fName]} =< up
}
pred BidiAssoc[left: set Obj, lFName:one FName, right: set Obj, rFName:one FName]
    ↪ {
    all l: left | all r: l.get[lFName] | l in r.get[rFName]
    all r: right | all l: r.get[rFName] | r in l.get[lFName]
}
fact NonEmptyInstancesOnly {
  some Obj
}
```

Listing 5.2. Common structure for class diagrams in Alloy

that could be either internal or external. A CRM manages many clients that could be either private clients or organizations.

The translation of this metamodel to the alloy domain using a modified version of the CD2Alloy framework. The CD2Alloy framework is designed to generate an alloy module from an input class diagram. The generated alloy module allows generating models that conform to that class diagram. Indeed, we are interested in manipulating the metamodel. We do believe that the problem of smells detection and design refactoring could be shifted to the metamodel level. This allows generalizing the problem by focusing on the metamodel instead of handling particular model instances. Since we are interested in metamodels rather than models generation, so, we modified the way associations and cardinalities are encoded in the CD2Alloy transformation. The modifications are automated in a way to be reused for any kind of application.

The generated alloy specification contains two parts: A general part that defines the common elements and structure in a class diagram (e.g. class, attribute, association, cardinality, etc.) as illustrated in 5.2 and a specific part that incorporates the constraints specific to the input class diagram as shown in listing 5.3.

```
// Concrete names of fields in cd
one sig matricule extends FName {}
one sig name extends FName {}
one sig client extends FName {}
one sig company extends FName {}
one sig worker extends FName {}
one sig crm extends FName {}
// Concrete value types in model cd
lone sig type_string extends Val {}
lone sig type_int extends Val {}
// Classes in model cd
one sig InternalWorker extends Obj {}
one sig LocatedElement extends Obj {}
//one sig Business extends Obj {}
one sig Company extends Obj {}
one sig Organization extends Obj {}
one sig Worker extends Obj {}
one sig Private extends Obj {}
one sig ExternalWorker extends Obj {}
one sig Client extends Obj {}
one sig CRM extends Obj {}
                                    .
                                    .
                                    .
// Values and relations in cd
pred generatedConstrtaints {
   // Definition of class InternalWorker
```

```
    ObjAttrib[InternalWorker, matricule, type_int]
    ObjFNames[InternalWorker, matricule + company + none]
    // Definition of class LocatedElement
    ObjFNames[LocatedElement, none]
    // Definition of class Company
    ObjAttrib[Company, name, type_string]
    ObjFNames[Company, name + worker + none]
    // Definition of class Organization
    ObjFNames[Organization, none]
    // Definition of class Worker
    ObjFNames[Worker, company + none]
    // Definition of class Private
    ObjFNames[Private, none]
    // Definition of class ExternalWorker
    ObjAttrib[ExternalWorker, matricule, type_int]
    ObjFNames[ExternalWorker, matricule + company + none]
    // Definition of class Client
    ObjAttrib[Client, name, type_string]
    ObjFNames[Client, name + none]
    // Definition of class CRM
    ObjFNames[CRM, client + company + none]

    // Associations
    ObjLAttrib[CRM, client, ClientSubsCD, 0]
    ObjLU[Client, client, CRMSubsCD, 1, 1]
    BidiAssoc[Company, worker, WorkerSubsCD, company]
    ObjLUAttrib[Worker, company, CompanySubsCD, 1, 1]
    ObjLAttrib[Company, worker, WorkerSubsCD, 0]
    ObjLUAttrib[CRM, company, CompanySubsCD, 0, 1]
    ObjLU[Company, company, CRMSubsCD, 1, 1]
}

// Run commands
run generatedConstrtaints for 5
```

Listing 5.3. Excerpt of the generated alloy specification for CRM metamodel

5.3.1.2. Encoding metamodel smells as constraints.

Alloy allows defining functions and predicates. Predicates are parametrized constraints. Alloy can check assertions in a bounded scope. If the assertion is true in the defined scope, then, the assertion might be valid. Otherwise, alloy gives a counterexample that contradicts the defined assertion. Alloy can execute a predicate by finding a model conforming to the defined specification and that satisfies the corresponding predicate (the predicate is true on the generated model).

We defined the non-existence of smells as predicates as shown in listing 5.4.

```
// [1] Duplicated feature in all sub-metaclasses of a parent metaclass
pred duplicatedFeature[cd: CD] {
    some fn: FName, v: Val | one op: cd.classes | isDuplicatedFeature[cd, fn, v,
    ↪ op]

}

pred isDuplicatedFeature[cd: CD, fn: FName, v: Val, parent: Obj] {
    #{child:cd.classes | child -> parent in cd.extensions}>1
    all child:cd.classes | child -> parent in cd.extensions implies child->fn->v
    ↪ in cd.features
}

// [2] Duplicated feature in some meta-classes
pred duplicatedFeature2[cd: CD] {
    some fn: FName, v: Val | isDuplicatedFeature2[cd, fn, v]
}

pred isDuplicatedFeature2[cd: CD, fn: FName, v: Val] {
    #{c:cd.classes | c->fn->v in cd.features}>1
    //=== OR ===
    //some c:cd.classes | c->fn->v in cd.features
}

// [3] Dead class
pred deadClass[cd: CD] {
    some c: cd.classes | isDeadClass[cd, c]
}

pred isDeadClass[cd: CD, o:Obj] {
    //No associations
    all x,y : associationEnd | (x->y in cd.associations) implies (not (o in x.
    ↪ class) and not (o in y.class))
    //No Extensions
    all p:cd.classes | not o->p in cd.extensions and not p->o in cd.extensions
}
```

Listing 5.4. Excerpt of the alloy module for design smells detection

As defined in chapter 4, a duplicated feature is defined as a field that is present in many metaclasses with the same name and type. Here, we differentiate two cases: (1) A feature that is present in all the metaclasses of a specific metaclass. In this case, the corresponding refactoring that allows to correct this metamodel smells is *Pull-up field*. It consists of pulling up the feature to the parent meta-class and removing it from all the sub-metaclasses. (2) A feature that is present in some meta-classes (not necessarily the sub-metaclasses of the same metaclass). To remove this metamodel smell, we should apply the *Extract Super Class* refactoring. It consists of creating a super-metaclass that incorporates this field. The metaclasses inherit this meta-attribute from this metaclass [**21**]. The first two segments in listing5.4 cover the cases of detecting duplicated features.

The third segment defines the predicates for ensuring the non-existence of dead meta-classes. A dead metaclass is defined in [**37**] as a completely disconnected metaclass from other

```
assert NoMetamodelSmells {
    all cd: CD | not (duplicatedFeature[cd] or duplicatedFeature[cd] or deadClass
    ↪ [cd])
} check NoMetamodelSmells for 9 but 1 CD
```

Listing 5.5. Assert the non-existence of metamodel smells

elements of the metamodel (similar situations are referred to as dead code or oxbow code in source code). In listing5.4, we detect dead metaclasses by searching for meta-classes that do not have associations and extension relations with the other elements of the metamodel.

5.3.1.3. Metamodel smells detection.

To ensure that our metamodel is consistent and does not contain design smells, we define an assertion to check the absence of metamodel smells.

Listing5.5 shows an example for checking the non-existence of metamodel smells on the CRM metamodel.

In the defined assertion, we check the non-existence of metamodel smells instances. In this example, we check the non-existence of two types of metamodel smells: duplicated features and dead classes.

We could check this assertion with alloy on the encoded specification. If the assertion is false, alloy gives a counterexample and thus the metamodel contains some metamodel smells that should be corrected (see Section 5.3.2). Otherwise, alloy validates the assertion in the defined scope and the metamodel might be consistent(i.e. probably, the metamodel does not contain metamodel smells).

5.3.2. Phase 2: Metamodel refactoring

If some metamodel smells are detected in the first phase. Then, the metamodel should be refactored to improve its quality. In our approach, we refactor metamodels with respect to some quality criteria.

As depicted in figure 5.3, we formulate the refactoring problem as an ordered finite state machine where:

- Each state represents a different version of the metamodel
- Each transition represents a single refactoring operation.

The figure 5.4 illustrates a flattened version of our approach.

- The initial state represents the input metamodel.
- The middle states represent the intermediate versions of the metamodel and that allows keeping track of the refactoring traces (i.e. the intermediate refactoring actions).
- The final state contains the refactored metamodel.

We use alloy as a constraint solver where we specify:

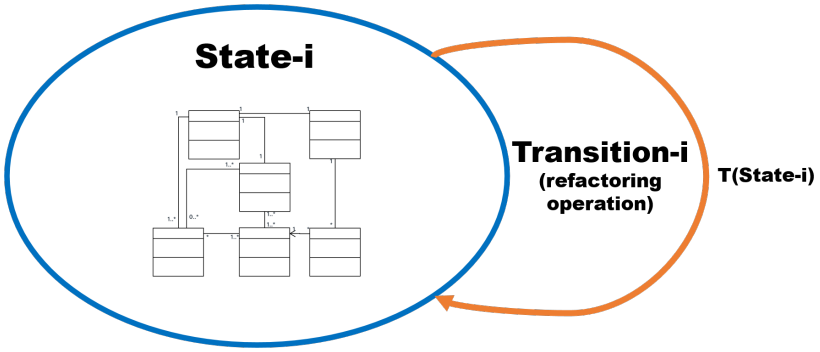


Fig. 5.3. Formulate the metamodel refactoring approach as an ordered Finite State Machine (FSM)

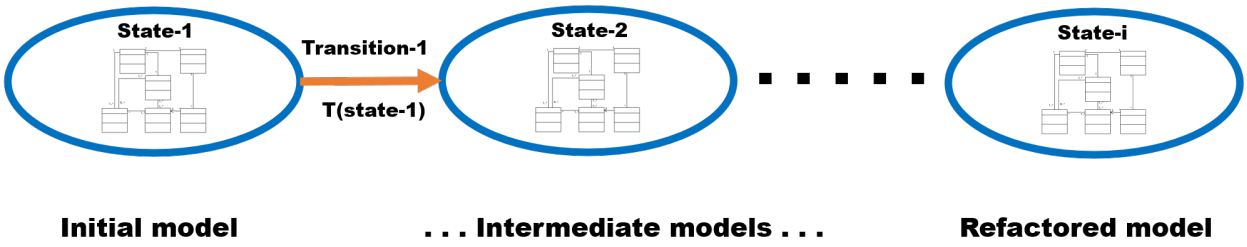


Fig. 5.4. Metamodel refactoring: flattened approach

- The initial version of the metamodel (i.e. the input metamodel).
- The transition types (i.e. the possible refactoring operations). For each refactoring, we define the conditions on the input state to be able to apply it. A transition is automatically determined based on the input state.
- The constraints on the final state: quality criteria and non-existence of smells.

The goal is to use a constraint solver to automatically search for a sequence of refactoring operations that allows refactoring some of the metamodel smells while satisfying some quality criteria.

Listing 5.6 shows an excerpt from the implementation of our approach in Alloy.

We used a built-in alloy module called *util/ordering* to force an ordering on the states. We defined a state as a signal element containing a class diagram that represents a metamodel. Every two states are connected by a transition which is the refactoring operation.

We define three refactoring operations (see listing 5.8):

- Remove dead metaclass: Removes the dead metaclass to resolve the corresponding smell.
- Pull-up field: If a feature is duplicated in all the sub-metaclasses of a parent metaclass, we apply the pull-up refactoring of this meta-attribute.

```
open util/ordering[State]

// state
sig State {
    cd: one CD
}

// types of transsitions OR possible refactoring operations
pred refactorOperation[cd, cd': CD] {
    refactor1[cd, cd']
    or
    refactor2[cd, cd']
    or
    refactor3[cd, cd']
}

//relation between states
fact {
  all s: State, s': s.next {
      refactorOperation [s.cd, s'.cd]
  }
    //Each class diagram should belong to one state
    all c: CD | one s: State | s.cd = c
}
```

Listing 5.6. Alloy code snippet: implementation of our refactoring approach

- Extract super metaclass: If a meta-attribute is duplicated in two or more metaclasses and does not satisfy the previous case, then, we create a super-metaclass that contains that meta-attribute.

Besides, we encoded with Alloy the quality model used in [**21**]. This quality model is composed of quality attributes: Maintainability, Complexity, Understandability, and Reusability. These quality attributes are defined in terms of some low-level design metrics (e.g. NC: Number of MetaClasses, NR: Number of References, NA: Number of metaAttributes, DITmax: max generalization hierarchical level, etc.).

We execute Alloy analyzer to solve the encoded constraints on the input specification. Alloy would find zero, one, or more configurations. For each configuration, the first state contains the input metamodel, the medium states contains the intermediate refactoring operations and allows to keep track of the execution traces and the final state contains the refactored metamodel. Then, it's on the modeler to choose the best configuration based on his own preferences and expertise.

5.4. Illustrative case study

We executed our approach on the CRM metamodel(figure 5.2). Table 5.1 shows some information about the problem size and the complexity of our approach.

This metamodel has three design smells:

- *LocatedElement* is *dead metaclass*.

```
//remove dead class
pred refactor1[cd, cd': CD]{
    one x: cd.classes | {
        isDeadClass[cd,x]
        cd'.classes = cd.classes - x
        cd'.associations = cd.associations
        cd'.extensions = cd.extensions
        all fn: FName, v: Val, c: Obj | c=x implies not c->fn->v in cd'.features
            else c->fn->v in cd.features implies c->fn->v in cd'.features
                else not c->fn->v in cd'.features
        }
}

//pull-up field
pred refactor2[cd, cd': CD]{
    one fn: FName, v: Val | one op: cd.classes | {
        isDuplicatedFeature[cd, fn, v, op]
        all f: FName, vv: Val, c: Obj | c not in cd.classes implies not c->f->vv
    ↪ in cd'.features
            else (fn=f and vv=v and c->op in cd.extensions) implies { not c->f->vv
    ↪  in cd'.features}
                else c->f->vv in cd.features implies c->f->vv in cd'.features
                    else (op=c and fn=f and vv=v) implies c->f->vv in cd'.features
                        else not c->f->vv in cd'.features

            cd'.classes = cd.classes
            cd'.    associations = cd.associations
            cd'.extensions = cd.extensions
        }
}
```

Listing 5.7. Alloy code snippet for refactoring operations

Table 5.1. Information about the execution of our approach on the CRM metamodel

Measure	Value
Scope	9
Number of variables	201429
Number of clauses	1331127
Execution time	9637 ms

- *matricule* and *name* are duplicated features.

Alloy detects metamodel smells and then generates a set of different refactoring solutions. The traces of the first solution are illustrated in figure 5.5. This solution is composed of three states.

The first state contains the initial metamodel (figure 5.6).

The second state contains an intermediate metamodel as shown in figure 5.7. The refactoring applied on the first state is *remove dead metaclass* (*LocatedElement*).

The resulting metamodel is shown in figure 5.7 that represents the second state. The dead metaclass *LocatedElement* is removed from the initial metamodel.

```
//Number of metaclasses
fun NC[cd: CD]: one Int {
    ans:Int | ans = #cd.classes
}
//Number of references
fun NR[cd: CD]: one Int {
    ans:Int | ans = #cd.associations
}
//Number of meta-attributes
fun NA[cd: CD]: one Int {
    ans:Int | ans = #cd.features
}
//Number of generalization hierarchies
fun NGenH[cd: CD]: one Int{
    ans:Int | ans = #cd.extensions
}
//Number of predecessors in hierarchy
fun PRED[cd: CD]: one Int{
    ans:Int | ans = (sum e:cd.classes  | DIT1[cd,e])
}
//Number of inherited features
fun Ai[cd: CD]: one Int {
    ans:Int | ans = (sum e:cd.classes | Ai[cd,e])
}
fun Ai[cd: CD, c: Obj]: one Int {
    ans:Int | ans = (mul[ minus[ (#c.*(~(cd.extensions))), 1] , #c.(cd.features)])
}

// ======= QA =======
fun Maintainability[cd: CD]: one Int{
    ans:Int | ans = negate[sum{NC[cd]+NA[cd]+NR[cd]+DITmax[cd]+NGenH[cd]}]
}
fun Understandability[cd: CD]: one Int{
    ans:Int | ans = div[PRED[cd]+1,NC[cd]]
}
fun Reusability[cd: CD]: one Int{
    ans:Int | ans = div [Ai[cd],add[Ai[cd],Ad[cd]]]
}
```

Listing 5.8. Alloy code snippet for the implementation of the quality attributes

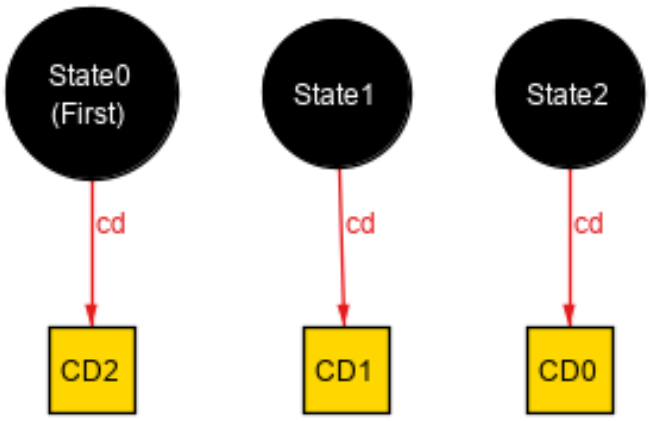


Fig. 5.5. Results of the execution of our approach on the CRM metamodel

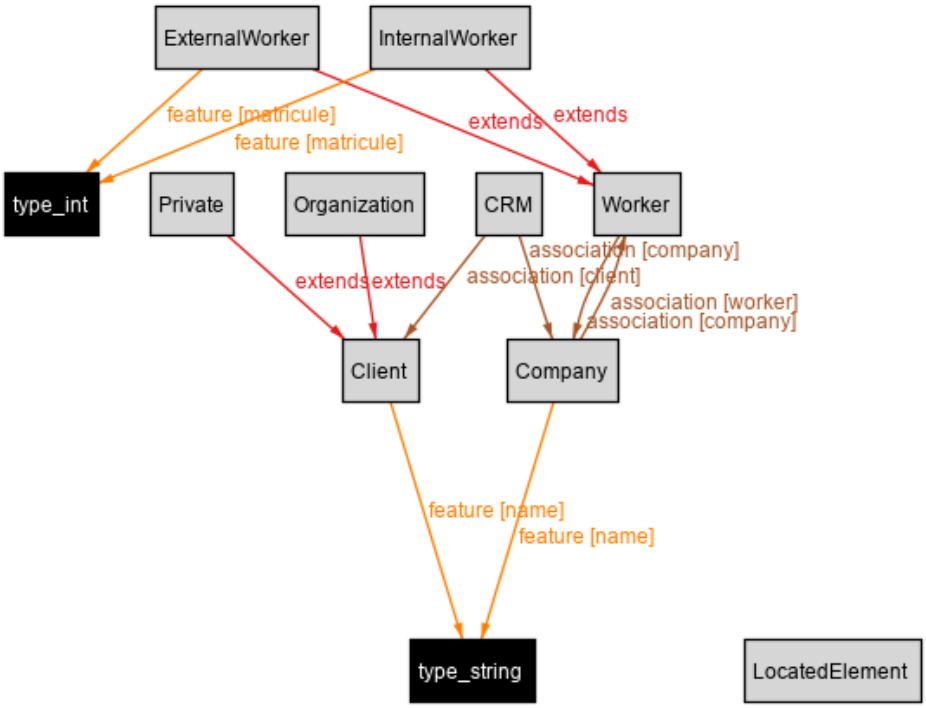


Fig. 5.6. Execution results: first state

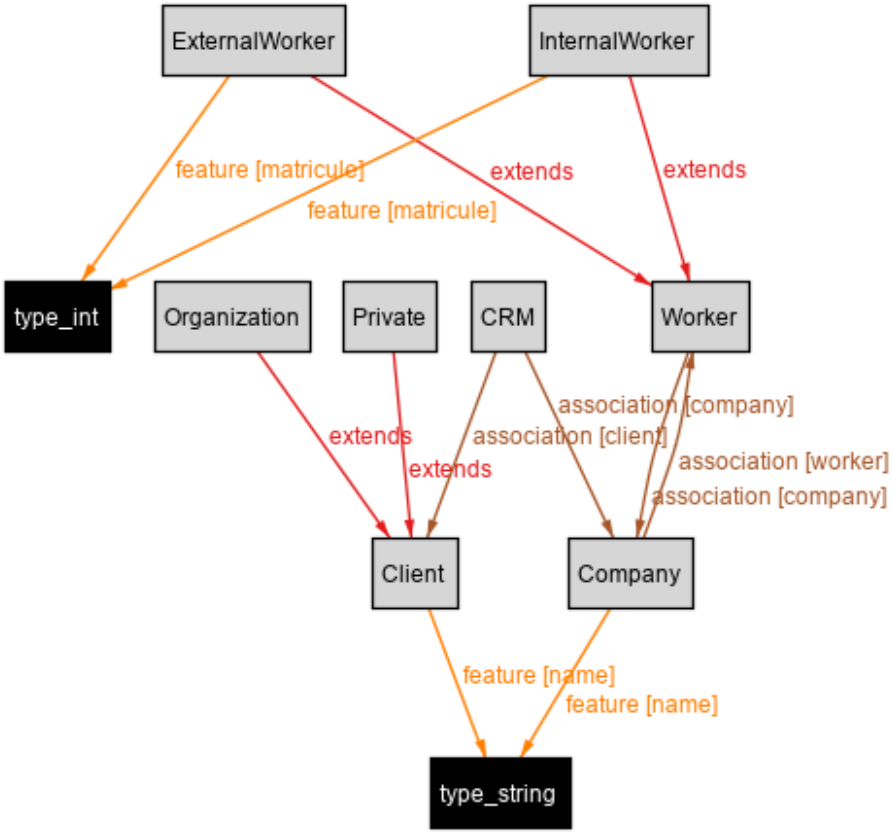


Fig. 5.7. Execution results: intermediate state

As shown in figure 5.8, in the third state the duplicated feature *matricule* was pulled up from the sub-metaclasses *InternalWorker* and *ExternalWorker* to the super-metaclass *Worker*.

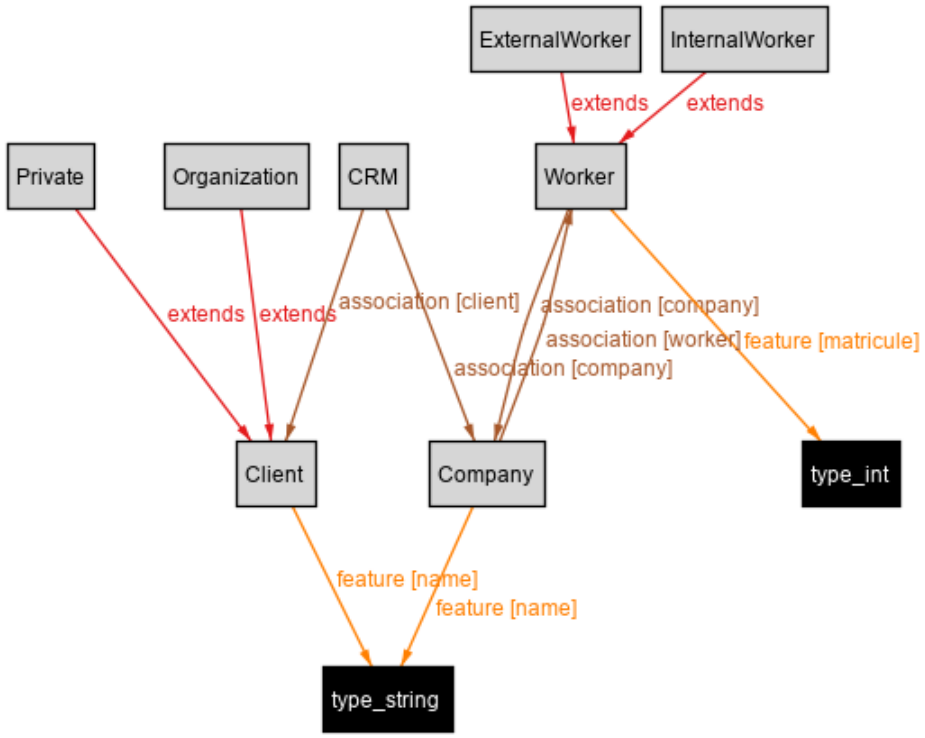


Fig. 5.8. Execution results: final state

The final state contains the refactored metamodel. For this example, we specified reusability, complexity, and maintainability as the three quality criteria that we want to improve for our input metamodel. The constraint solver suggests this configuration as one solution to refactor design smells while improving the quality attributes.

The final metamodel (figure 5.8) contains another bad smell which is duplicated feature (name). To refactor this bad smell, we should extract a new super-metaclass that contains that feature. This would increase the complexity of the metamodel which may explain why the constraint solver didn't apply that refactoring operation and has kept that smell to satisfy the quality criteria.

Our constraint solving-based approach is capable of generating a set of refactorings that satisfy certain quality criteria. This differs from the approach presented in chapter 4 in several points:

- Constraint solving is a more robust approach since it is based on mathematics.
- Search-based methods do not always find the best solution. In contrast, constraint solvers are capable of generating optimal solutions that satisfy the encoded constraints.

- The results generated by search-based methods depend on the initialization and on many hyper-parameters (number of iterations, crossover rate, mutation rate, population size, solution size) while our constraint solving approach depends only on the scope.
- Our search-based approach is more complex and time consuming.

We aim to explore more the performance of the two approaches and to compare them in terms of complexity, execution time and quality of generated solutions, etc.

Conclusion

In this chapter, we presented an approach for quality-driven metamodel refactoring. Our approach is based on constraint solving. We translate the input metamodel to an alloy specification. We encode the non-existence of smells and the quality criteria as constraints in Alloy. Then, we use Alloy analyzer to satisfy the requirements and obtain a refactored version of the metamodel that satisfies some quality criteria. The proposed approach allows us to track the execution traces (the intermediate steps of the refactoring process) and to translate back the refactored metamodel to a UML class diagram.

Chapter 6

Conclusion and future work

In this chapter, we summarize the contributions addressed in this thesis and discuss future research directions.

6.1. Summary

The main objective of this thesis is to assist software specialists in their tasks using AI techniques. We adopt a systematic approach to tackle the proposed problematics.

6.1.1. Automation and improvement of the software development pipeline: A cartography of ML-based opportunities

In our first contribution (chapter 3), we investigated the potential improvement opportunities for software-related tasks. First, we identified some of these tasks in an industrial setting. Second, we analyzed them to locate potential issues, improvement tracks, and the required knowledge to accomplish these tasks. Third, we recommend some AI-based solutions to overcome the identified problems, infer expertise from historical data, and increase the automation level of some tasks. Finally, we depict a graphical and abstract cartography that incorporates some of the suggested solutions.

6.1.2. Quality-driven multi-objective optimization approach for metamodel refactoring

In this contribution (chapter 4), we are interested in the metamodelling task. We propose an AI-based approach to refactor metamodels. This process is driven by the optimization of the quality attributes and the number of removed smells. We proposed a tool that implements our approach.

We defined a quality model that incorporates a set of quality attributes. We created a module that allows us to automatically evaluate a defined quality metamodel. Also, we

implemented a library for metamodel smells detection. Then, we employed all these components for our search-based approach to refactor metamodels. We used NSGA-II to improve quality attributes and maximize the number of smells to be removed.

Our framework recommends a set of non-dominated refactoring solutions. Then, it is on the modeler to select the most suitable solution based on his preferences. We validated our approach on a set of metamodels. Our framework was able to recommend relevant refactoring solutions that improve quality attributes.

6.1.3. Metamodel refactoring using constraint solving

In chapter 5, we propose a framework based on an AI-technique, constraint solving, to refactor metamodels while satisfying quality criteria.

For an input metamodel represented as a class diagram, we translate it to an Alloy specification. Then, we encoded the non-existence of smells as logical constraints. We used Alloy analyzer to check the constraints on the encoded specification.

If some metamodel smells are detected, a second phase is triggered to start the refactoring process. We encoded some quality criteria in Alloy. Furthermore, we formulated the refactoring approach as an ordered finite state machine where the initial state represents the input metamodel, the medium states represent the intermediate steps in the refactoring process allowing to keep track of the refactoring traces, and a final state that contains the refactored metamodel satisfying the specified constraints. A transition between two states represents a refactoring operation applied on the input state to refactor an existing bad smell and outputs a state containing the refactored metamodel. We used Alloy to satisfy the encoded constraints on the input specification.

We consolidated our framework with a visualization theme to keep track of all details of the refactoring process. Besides, we propose a reverse transformation Alloy2CD to translate back the refactored metamodel to the UML class diagram domain.

6.2. Future work

In this section, we discuss some future research directions.

First, we plan to enrich the cartography presented in chapter 3. We aim to investigate more the tasks related to software development as well as to extend the list of suggestions to improve these tasks.

Furthermore, we aim to tackle more software problematics from our proposed cartography. This would help to improve and automate software-related activities, enhance the quality of deliverables, and alleviate the burden on software specialists by assisting them in their tasks.

In addition, we plan to extend our search-based approach for metamodel refactoring. It might be challenging to select the most relevant solution among a large list of recommendations. Having a large design would make the task more complex. We intend to refine our approach by clustering the output solutions. This would assist modelers to choose the most suitable cluster that fits his refactoring preferences. A further improvement might be running a second optimization round to investigate more solutions in that cluster. Therefore, this allows us to recommend more specific solutions, and that fit the modeler's preferences.

We plan to compare the approaches presented in Chapters 4 and 5. We aim to conduct an empirical study to evaluate and compare the performance of the two approaches on the same use cases.

Besides, we aim to propose a tool that implements the Alloy2CD transformation.

Finally, we will address the problematics, tackled in this thesis, using different AI techniques, and conduct experimental comparisons of performance.

Bibliography

[1] Google cloud platform | documentation: https://cloud.google.com/docs.

[2] Google cloud platform | products and services: https://cloud.google.com/products.

[3] Sonarqube documentation: https://docs.sonarqube.org/.

[4] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão et António Ribeiro : Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015.

[5] Vahid Alizadeh, Marouane Kessentini, Wiem Mkaouer, Mel Ocinneide, Ali Ouni et Yuanfang Cai : An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 2018.

[6] Douglas G Altman et J Martin Bland : Statistics notes: quartiles, quintiles, centiles, and other quantiles. *Bmj*, 309(6960):996–996, 1994.

[7] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi et Thomas Zimmermann : Software engineering for machine learning: A case study. *In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[8] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino *et al.* : An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[9] Bill Andreopoulos : Satisficing the conflicting software qualities of maintainability and performance at the source code level. *In WER*, pages 176–188. Citeseer, 2004.

[10] Thorsten Arendt et Gabriele Taentzer : Integration of smells and refactorings within the eclipse modeling framework. *In Proceedings of the Fifth Workshop on Refactoring Tools*, pages 8–15. ACM, 2012.

[11] Thorsten Arendt et Gabriele Taentzer : A tool environment for quality assurance based on the eclipse modeling framework. *Automated Software Engineering*, 20(2):141–184, 2013.

[12] Anish Athalye, Nicholas Carlini et David Wagner : Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[13] Arash Bahrammirzaee : A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems. *Neural Computing and Applications*, 19(8):1165–1195, 2010.

[14] Ankica Barišic : Usability evaluation of domain-specific languages. 2017.

[15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz et Shin Yoo : The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[16] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino et Alfonso Pierantonio : A customizable approach for the automated quality assessment of modelling artifacts. *In 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 88–93. IEEE, 2016.

[17] Yoshua Bengio, Aaron Courville et Pascal Vincent : Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[18] Chris Benner : *Work in the new economy: Flexible labor markets in Silicon Valley*, volume 9. John Wiley & Sons, 2008.

[19] Michael R Berthold, Nicolas Cebron, Fabian Dill, Thomas R Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel et Bernd Wiswedel : Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.

[20] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino et Alfonso Pierantonio : Edelta: An approach for defining and applying reusable metamodel refactorings. *In MODELS (Satellite Events)*, pages 71–80, 2017.

[21] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino et Alfonso Pierantonio : Quality-driven detection and resolution of metamodel smells. *IEEE Access*, 7:16364–16376, 2019.

[22] Jean Bézivin : On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[23] Alexei Botchkarev : Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology. *arXiv preprint arXiv:1809.03006*, 2018.

[24] Marco Brambilla, Jordi Cabot et Manuel Wimmer : Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[25] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib et D Sculley : The ml test score: A rubric for ml production readiness and technical debt reduction. *In 2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE, 2017.

[26] Michael Brooks : Artificial ignorance. *New Scientist*, 236(3146):28–33, 2017.

[27] Jason Brownlee : *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.

[28] Massimo Buscema : Back propagation neural networks. *Substance use & misuse*, 33(2):233–270, 1998.

[29] G Campbell et Patroklos P Papapetrou : *SonarQube in action*. Manning Publications Co., 2013.

[30] Richard Chang, Sriram Sankaranarayanan, Guofei Jiang et Franjo Ivancic : Software testing using machine learning, décembre 30 2014. US Patent 8,924,938.

[31] Kasper Christensen, Sladjana Nørskov, Lars Frederiksen et Joachim Scholderer : In search of new product ideas: Identifying ideas in online communities by machine learning and text mining. *Creativity and Innovation Management*, 26(1):17–30, 2017.

[32] Skin Imaging Collaboration *et al.* : Machine learning and health care disparities in dermatology. 2018.

[33] Steve Cook, Gareth Jones, Stuart Kent et Alan Cameron Wills : *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[34] Hoa Khanh Dam : Artificial intelligence for software engineering. *XRDS: Crossroads, The ACM Magazine for Students*, 25(3):34–37, 2019.

[35] David L Davies et Donald W Bouldin : A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2):224–227, 1979.

[36] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal et TAMT Meyarivan : A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[37] Saumya K Debray, William Evans, Robert Muth et Bjorn De Sutter : Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):378–415, 2000.

[38] Janez Demšar et Blaž Zupan : Orange: Data mining fruitful and fun-a historical perspective. *Informatica*, 37(1), 2013.

[39] Arie Van Deursen et Paul Klint : Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[40] Marco Dorigo et Christian Blum : Ant colony optimization theory: A survey. *Theoretical computer science*, 344(2-3):243–278, 2005.

[41] Vinicius HS Durelli, Rafael S Durelli, Simone S Borges, Andre T Endo, Marcelo M Eler, Diego RC Dias et Marcelo P Guimaraes : Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212, 2019.

[42] Elfriede Dustin, Jeff Rashka et John Paul : *Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999.

[43] Agoston E Eiben et Marc Schoenauer : Evolutionary computing. *Information Processing Letters*, 82(1):1–6, 2002.

[44] Agoston E Eiben et James E Smith : *Introduction to evolutionary computing*. Springer, 2015.

[45] Marco Farina, Kalyanmoy Deb et Paolo Amato : Dynamic multiobjective optimization problems: test cases, approximations, and applications. *IEEE Transactions on evolutionary computation*, 8(5): 425–442, 2004.

[46] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva et Eduardo Figueiredo : A review-based comparative study of bad smell detection tools. *In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 18. ACM, 2016.

[47] M Folwer : Refactoring: Improving the design of existing progrmas, 1999.

[48] Robert France, S Chosh, Eunjee Song et Dae-Kyoo Kim : A metamodeling approach to pattern-based model refactoring. *IEEE software*, 20(5):52–58, 2003.

[49] Dov M Gabbay, Christopher John Hogger, CJ Hogger, JA Robinson et John Alan Robinson : *Handbook of Logic in Artificial Intelligence and Logic Programming: Volume 2: Deduction Methodologies*, volume 2. Oxford University Press, 1993.

[50] John S Gero : *Artificial Intelligence in design'92*. Springer Science & Business Media, 2012.

[51] Rohit Gheyi, Tiago Massoni et Paulo Borba : A rigorous approach for proving model refactorings. *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 372–375, 2005.

[52] Jeremy Goecks, Anton Nekrutenko, James Taylor, Galaxy Team *et al.* : Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[53] Ian Goodfellow, Yoshua Bengio et Aaron Courville : Deep learning http://www. deeplearningbook. org. *MIT Press, Cambridge, MA*, 2016.

[54] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville et Yoshua Bengio : Generative adversarial nets. *In Advances in neural information processing systems*, pages 2672–2680, 2014.

[55] Ian J Goodfellow, Jonathon Shlens et Christian Szegedy : Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[56] Katja Grace, John Salvatier, Allan Dafoe, Baobao Zhang et Owain Evans : When will ai exceed human performance? evidence from ai experts. *Journal of Artificial Intelligence Research*, 62:729–754, 2018.

[57] Aakriti Gupta et Shreta Sharma : Software maintenance: Challenges and issues. *Issues*, 1(1):23–25, 2015.

[58] Tracy Hall, Min Zhang, David Bowes et Yi Sun : Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[59] Neil B Harrison et Paris Avgeriou : Leveraging architecture patterns to satisfy quality attributes. *In European conference on software architecture*, pages 263–270. Springer, 2007.

[60] Regina Hebig, Djamel Eddine Khelladi et Reda Bendraou : Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, 2016.

[61] Adrián Hernández-López, Ricardo Colomo-Palacios et Ángel García-Crespo : Productivity in software engineering: A study of its meanings for practitioners: Understanding the concept under their standpoint. *In 7th Iberian Conference on Information Systems and Technologies (CISTI 2012)*, pages 1–6. IEEE, 2012.

[62] Adrián Hernández-López, Ricardo Colomo-Palacios, Ángel García-Crespo et Fernando Cabezas-Isla : Software engineering productivity: Concepts, issues and challenges. *International Journal of Information Technology Project Management (IJITPM)*, 2(1):37–47, 2011.

[63] Markus Herrmannsdoerfer : Cope–a workbench for the coupled evolution of metamodels and models. *In International Conference on Software Language Engineering*, pages 286–295. Springer, 2010.

[64] Bhadeshia HKDH : Neural networks in materials science. *ISIJ international*, 39(10):966–979, 1999.

[65] Markus Hofmann et Ralf Klinkenberg : *RapidMiner: Data mining use cases and business analytics applications*. CRC Press, 2016.

[66] Geoffrey Holmes, Andrew Donkin et Ian H Witten : Weka: A machine learning workbench. *In Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference*, pages 357–361. IEEE, 1994.

[67] Mohammad Hossin et MN Sulaiman : A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):1, 2015.

[68] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li et Tom Oinn : Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl_2):W729–W732, 2006.

[69] C-L Hwang et Abu Syed Md Masud : *Multiple objective decision making—methods and applications: a state-of-the-art survey*, volume 164. Springer Science & Business Media, 2012.

[70] Sergey Ioffe et Christian Szegedy : Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[71] Daniel Jackson : Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[72] Bernd Jagla, Bernd Wiswedel et Jean-Yves Coppée : Extending knime for next-generation sequencing data analysis. *Bioinformatics*, 27(20):2907–2909, 2011.

[73] Leslie Pack Kaelbling, Michael L Littman et Andrew W Moore : Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[74] Harini Kannan, Alexey Kurakin et Ian Goodfellow : Adversarial logit pairing. *arXiv preprint arXiv:1803.06373*, 2018.

[75] Steven Kelly et Juha-Pekka Tolvanen : *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[76] Wael Kessentini, Houari Sahraoui et Manuel Wimmer : Automated metamodel/model co-evolution using a multi-objective optimization approach. *In European Conference on Modelling Foundations and Applications*, pages 138–155. Springer, 2016.

[77] Ruth C King, Weidong Xia, James Campbell Quick et Vikram Sethi : Socialization and organizational outcomes of information technology professionals. *Career Development International*, 2005.

[78] Barbara Kitchenham et Emilia Mendes : Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering*, 30(12):1023–1035, 2004.

[79] Paul Klint, Tijs Van Der Storm et Jurgen Vinju : On the impact of dsl tools on the maintainability of language implementations. *In Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, pages 1–9, 2010.

[80] J Zico Kolter et Eric Wong : Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 1(2):3, 2017.

[81] Abdullah Konak, David W Coit et Alice E Smith : Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[82] Tomaž Kosar, Pablo E Marti, Pablo A Barrientos, Marjan Mernik *et al.* : A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, 50(5):390–405, 2008.

[83] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis et Dimitrios I Fotiadis : Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17, 2015.

[84] Mark A Kramer : Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

[85] Anders Krogh : What are artificial neural networks? *Nature biotechnology*, 26(2):195–197, 2008.

[86] Hanjun Lee, Keunho Choi, Donghee Yoo, Yongmoo Suh, Soowon Lee et Guijia He : Recommending valuable ideas in an open innovation community. *Industrial Management & Data Systems*, 2018.

[87] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan MK Selim, Eugene Syriani et Manuel Wimmer : Model transformation intents and their properties. *Software & systems modeling*, 15(3):647–684, 2016.

[88] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao et Yang Zhao : Scientific workflow management and the kepler system. *Concurrency and computation: Practice and experience*, 18(10):1039–1065, 2006.

[89] Zhiyi Ma, Xiao He et Chao Liu : Assessing the quality of metamodels. *Frontiers of Computer Science*, 7(4):558–570, 2013.

[90] Alan MacCormack, Chris F Kemerer, Michael Cusumano et Bill Crandall : Trade-offs between productivity and quality in selecting software development practices. *Ieee Software*, 20(5):78–85, 2003.

[91] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi et Amit P Sheth : Machine learning for internet of things data analysis: A survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

[92] Shahar Maoz, Jan Oliver Ringert et Bernhard Rumpe : Cd2alloy: Class diagrams analysis using alloy revisited. *In International Conference on Model Driven Engineering Languages and Systems*, pages 592–607. Springer, 2011.

[93] R Timothy Marler et Jasbir S Arora : Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[94] Frank J Massey Jr : The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[95] Hervé Ménager, Vivek Gopalan, Bertrand Néron, Sandrine Larroudé, Julien Maupetit, Adrien Saladin, Pierre Tufféry, Yentram Huyen et Bernard Caudron : Bioinformatics applications discovery and composition with the mobyle suite and mobylenet. *In International Workshop on Resource Discovery*, pages 11–22. Springer, 2010.

[96] Tim Menzies : Practical machine learning for software engineering and knowledge engineering. *In Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 837–862. World Scientific, 2001.

[97] Bart Meyers et Hans Vangheluwe : A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.

[98] Bart Meyers, Manuel Wimmer, Antonio Cicchetti et Jonathan Sprinkle : A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, 42, 2012.

[99] Tom M Mitchell *et al.* : Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

[100] Parastoo Mohagheghi, Vegard Dehlen et Tor Neple : Definitions and approaches to model quality in model-based software development–a review of literature. *Information and software technology*, 51(12):1646–1669, 2009.

[101] Emerson Murphy-Hill et Andrew P Black : Refactoring tools: Fitness for purpose. *IEEE software*, 25(5):38–44, 2008.

[102] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty HC Cheng, Philippe Collet, Benoit Combemale, Robert B France, Rogardt Heldal, James Hill *et al.* : The relevance of model-driven engineering thirty years from now. *In International Conference on Model Driven Engineering Languages and Systems*, pages 183–200. Springer, 2014.

[103] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson et Danny Dig : A comparative study of manual and automated refactorings. *In European Conference on Object-Oriented Programming*, pages 552–576. Springer, 2013.

[104] Bao N Nguyen, Bryan Robbins, Ishan Banerjee et Atif Memon : Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, 21(1):65–105, 2014.

[105] Vilém Novák, Irina Perfilieva et Jiri Mockor : *Mathematical principles of fuzzy logic*, volume 517. Springer Science & Business Media, 2012.

[106] William F Opdyke : Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *In Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[107] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik et Ananthram Swami : Practical black-box attacks against machine learning. *In Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[108] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha et Ananthram Swami : Distillation as a defense to adversarial perturbations against deep neural networks. *In 2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[109] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen et Mika V Mäntylä : Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. *In 2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE, 2012.

[110] Aditi Raghunathan, Jacob Steinhardt et Percy Liang : Certified defenses against adversarial examples. *arXiv preprint arXiv:1801.09344*, 2018.

[111] Soumaya Rebai, Marouane Kessentini, Vahid Alizadeh, Oussama Ben Sghaier et Rick Kazman : Recommending refactorings via commit message analysis. *Information and Software Technology*, page 106332, 2020.

[112] Soumaya Rebai, Oussama Ben Sghaier, Vahid Alizadeh, Marouane Kessentini et Meriem Chater : Interactive refactoring documentation bot. *In 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 152–162. IEEE, 2019.

[113] Ganesh B Regulwar et RM Tugnayat : Detection of bad smell code for software refactoring. *In Innovations in Computer Science and Engineering*, pages 143–152. Springer, 2019.

[114] Jan Reimann, Mirko Seifert et Uwe Assmann : Role-based generic model refactoring. *In International Conference on Model Driven Engineering Languages and Systems*, pages 78–92. Springer, 2010.

[115] Peter J Rousseeuw : Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[116] Jeffrey R Sampson : Adaptation in natural and artificial systems (john h. holland), 1976.

[117] Jürgen Schmidhuber : Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[118] Ken Schwaber et Mike Beedle : *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[119] Shane Sendall et Wojtek Kozaczynski : Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[120] Jahanzaib Shabbir et Tarique Anwer : Artificial intelligence and its role in near future. *arXiv preprint arXiv:1804.01396*, 2018.

[121] Mojtaba Shahin, Muhammad Ali Babar et Liming Zhu : Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[122] Jeffrey P Solloway, Jay W Yang et Ying He : Extensible automated testing software, mars 16 2004. US Patent 6,708,324.

[123] Friedrich Steimann : Constraint-based model refactoring. *In International Conference on Model Driven Engineering Languages and Systems*, pages 440–454. Springer, 2011.

[124] Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung et Robert Heinrich : Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. 2016.

[125] Eugene Syriani, Jeff Gray et Hans Vangheluwe : Modeling a model transformation language. *In Domain Engineering*, pages 211–237. Springer, 2013.

[126] Bruce Thompson : Canonical correlation analysis. *Encyclopedia of statistics in behavioral science*, 2005.

[127] Hans Vangheluwe et Juan De Lara : Meta-models are models too. *In Proceedings of the Winter Simulation Conference*, volume 1, pages 597–605. IEEE, 2002.

[128] Ulrich HH Wild et Mohamed Iyad Jabri : System and method for automated testing and monitoring of software applications, septembre 23 1997. US Patent 5,671,351.

[129] James R Williams : *A novel representation for search-based model-driven engineering*. Thèse de doctorat, University of York, 2013.

[130] James R Williams, Richard F Paige et Fiona AC Polack : Searching for model migration strategies. *In Proceedings of the 6th International Workshop on Models and Evolution*, pages 39–44. ACM, 2012.

[131] M-S Yang : A survey of fuzzy clustering. *Mathematical and Computer modelling*, 18(11):1–16, 1993.

[132] Ehsan Zabardast, Javier Gonzalez-Huerta et Darja Šmite : Refactoring, bug fixing, and new development effect on technical debt: An industrial case study. *In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 376–384. IEEE, 2020.

[133] Du Zhang : *Advances in machine learning applications in software engineering*. Igi Global, 2006.

[134] Martin Zinkevich : Rules of machine learning: Best practices for ml engineering. *URL: https://developers. google. com/machine-learning/guides/rules-of-ml*, 2017.

Appendix A

Cartography of AI-based improvement opportunities for software-related tasks

A.1. Graphical cartography: Overview

Here, we depict the ideas, presented in chapter 3 section 3.5, in a graphical format that is more usable and readable.

Figure A.1 illustrates a graphical cartography that depicts the general pipelines of machine learning. The cartography projects the proposed ML-based improvement ideas to that pipeline. It describes each step of the pipeline, highlights the problems and required expertise to accomplish each step and also suggests improvement ideas to overcome those problems, automate these tasks and infer the required expertise from historical experience to assist software specialists in the accomplishment of their tasks.

A.2. Projection over the cartography sub-parts

In this section, we detail the different sub-parts of the cartography (figure A.1).

Fig. A.1. Graphical cartography of AI-based improvement opportunities for software-related tasks

A.2.1. Data engineering

Table A.1. Data engineering: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Data confidentiality is a preoccupation and needs to be constantly improved. - The access and usage of some confidential data is limited → waste of time and decreased team efficiency. - Setting up advanced security measures reduces the risk but doesn't eliminate it. There is always a risk even with a low probability. ⇒ A Naive solution to generate data is Random generator → Data could be inconsistent and unrealistic. - Software practitioners often need real data to simulate and test real-life cases. - Realistic data is useful for testing, visualization, development, data engineering... → Data is a requirement for most of the software development tasks.	- Generate realistic data that is indistinguishable from the real data. This data is coherent and has the same characteristics as the real data. - Use Machine Learning (Auto-Encoders, GANs, etc.) to build unbiased models to generate fake data that is indistinguishable from real data (same statistical properties). - This allows to: • Promote the confidentiality and security of the real data. • Limit the access to real data to the final phase (all the tasks can be done using fake data. The access to the real data is required just in the final phase to train the final model). • Unlimited access to generated data → Better efficiency and productivity inside the team.

A.2.2. Development of ML models

Table A.2. Development of ML models: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Implementing a Machine Learning algorithm is a recurrent task. - Depending on the problem type (regression, classification, clustering), many algorithms are often used thanks to their good performance. - The development task requires: • Language expertise: python, java etc. • Library expertise: TensorFlow, PyTorch, etc. • Being able to re-use and adapt the old implementations to the new problem and input data and do not re-implement from scratch to gain time and effort. - The choice of the right ML model to use requires deep and diversified expertise in ML. - Each ML algorithm has its own advantages and drawbacks. The choice of the most appropriate ML algorithm to use depends on many criteria: • Problem description • Data description (type, distribution, size, etc.) • The desired output, etc.	- A General implementation of the most used models is useful and important. ⇒ This allows to re-use the general implementation and has the following advantages: + Save effort and time. + Ensure high source code quality. + Accelerate the development process. - Build a Domain Specific Language (DSL): a language that allows scientists to describe their specification needs (problem description, data, output etc.) using abstractions and notations from their own domains of expertise. - Being executed, the DSL recommends the most appropriate model to use based on the input specification.

A.2.3. Machine learning pipeline

Table A.3. ML pipeline: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- The machine learning pipeline is composed of the following steps: Data pre-processing → Development → Training → Validation → Evaluation. - These steps are recurrent and costly in terms of time and effort. - Scientists usually use the same algorithms depending on the problem type (classification: Logistic regression, SVM, Decision Tree, Random Forest, etc.) - The most performant model is selected in the validation step and then evaluated. - According to '2018 Kaggle ML and Data Science Survey', *15–26% of the time of a typical data science project is devoted to model building or model selection.*	- Automated-Machine Learning (Auto-ML): An approach that consists of automating the whole machine learning pipeline. Input = data → Output = evaluated model - The machine learning pipeline could be fully or partially automated. - This allows to save time and effort, get fast results (it is often used to generate fast prototypes, and then improve them). - Scientists could focus on more valuable/important tasks that require human intelligence and intuition.

A.2.4. Evaluation of ML models

Table A.4. Evaluation of ML models: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Metrics are a static way to evaluate models and detect basic behaviors. - We need to understand better the strengths and weaknesses of the model. - We need to go deeper into the weaknesses of the model using other techniques and to identify the limitations of the model and correct them. - A machine learning model could be unstable in the prediction phase. Unstable model = similar input → considerably different output. ⇒ This input is called an Adversarial example. Adversarial examples could be critical in some sensitive cases (e.g. medicine, security, etc.) where we need a robust and performant model.	- Detect individuals or range of values to which the model is sensitive and unstable using Adversarial Learning. - Adversarial learning consists of training a model that modifies the training input slightly with random noise and tries to maximize the difference of the output. - Generate and detect adversarial examples where the model performs badly by generating wrong results. - Make the machine learning model more robust by setting up adversarial defenses (i.e. Binary classifier, feature squeezing, adversarial re-training, etc.).

A.2.5. Monitoring of ML models

Table A.5. Monitoring of ML models: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Detect model performance degradation ¸and data drift is very important. - Thresholds are a static and classic way to detect model performance degradation. - Outliers are detected manually from the monitoring dashboard (graphs, distributions, tables, metrics, etc.) which is tedious and not accurate. - When we detect that the new production data distribution has changed (data drift) or the statistical relation between the features and the output variable is no more valid (concept drift); we should re-train and re-fit the ML model to the new data so it can learn the new statistical properties. - Data and concepts are changing, we should forecast the detection of these changes to act early and trigger the re-training process.	- The detection of model performance degradation could be anticipated → Predict model performance deterioration earlier (based on the evolution of the performance metrics). - Automatic detect of outliers (i.e. RCF, Isolation trees, One-class SVM, LR, AE, etc.). - Automatic detection of data drift (i.e. automatic comparison of distributions, Auto-Encoders, etc.). - Furthermore, We could avoid concept and data drifts and performance degradation by using a new emergent concept which is: Continuous Learning. ⇒ Continuous learning (CL): fine-tune and auto-adapt to new data continuously (real-time or batch) to maintain a good performance and to be always up-to-date.

A.2.6. Scrum: sprint planning

Table A.6. Scrum - sprint planning: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Sprint planning is an essential meeting in Scrum that consists of selecting a sub-set of stories from the product backlog and assigning stories/issues (tasks) to team members. - A user Story is characterized by: description – story points – priority – assignee. - Incorrect estimations of story points imply delays. - In the sprint planning, we should: • Prioritize the most important user stories. • Take into account the capacity and expertise of each of the team members. • Assign tasks according to the above criteria.	- We could estimate the story points from the previous tickets. - Assigning tickets to team members could be formalized as an optimization problem / Resources allocation problem where: • Tickets are characterized by: priority – complexity – skills • Team members are characterized by: capacity – expertise • Objectives: Higher priority tickets are assigned first – max served tickets – fill members' capacity – match skills-expertise. ⇒ we could have many sub-optimal configurations and choose/modify the most appropriate one. - This process could be semi-automated and take into account user preferences and interactions.

A.2.7. Security

Table A.7. Security: AI-based improvement opportunities

Required Expertise	Improvement opportunities
- Security is a concern in IT companies and it includes the following aspects: • Ensure the security of the cloud environments. • Control access to different bank resources. • Assess the confidentiality of the data. - Security involves many challenges: • Receiving many security alerts. These alerts should be monitored, analyzed, and resolved by executing the appropriate actions. • We should be able to analyze many log files to localize security incidents (An intelligent log analyzer to analyze security threats is very important). • There could be new security risks that could not be detected using the common tools. • After a security incident occurs, we should identify the causes of this incident and the related event from the different log files.	- Learn and detect anomalies patterns in logs. - Rank security alerts according to severity to alleviate the burden on the member responsible for this difficult task. - Discover connections between data in multiple log files (there are usually multiple records of a security incident): This is called Correlation analysis. - Artificial ignorance is very important as it allows to ignore routine log messages such as regular system updates but allow for new or unusual messages to be detected and flagged for investigation.