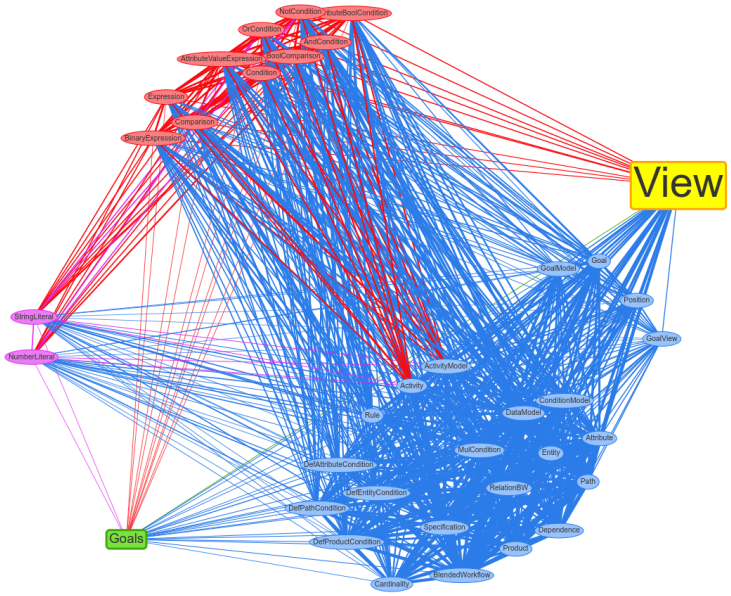
TÉCNICO LISBOA



Monolith Microservices Identification: An Extensible Multiple Strategy Tool

Telmo Domingues Lopes

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisor: Prof. António Rito Silva

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. António Rito Silva
Member of the Committee: Prof. Filipe Alexandre Pais de Figueiredo Correia

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to start by deeply thanking my parents for all their support and love during these years. Without them, my academic life would not be possible. Secondly, I would like to thank my family for being united in supporting me even when I was distant from them.

I would also like to thank my friends who have been with me through the good and troubling times, even during isolation, especially André Francisco. I also need to thank my colleagues that helped me during my years at university, namely José Miguel, my partner in war during the darkest hours.

Finally, I also need to acknowledge my dissertation supervisor Prof. António Rito Silva, for guiding me during the development of this tool since the beginning and for all his availability and commitment.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO).

Abstract

Several approaches have been proposed for the automatic identification of microservices in monolith systems. These approaches follow different strategies, from how they collect the monolith data, to the algorithms they apply for the identification, as well as the visualization of the candidate microservices. On the other hand, it seems that there is not a clear winning strategy. Therefore, more experimentation and comparison between approaches is required. However, there is no environment that facilitates it while providing assessment tools, since the experimentation and comparison is not the priority. This paper proposes an extensible multiple strategy tool, designed upon the specification of a microservices identification pipeline, in order to promote the comparison of decomposition approaches and help future investigators in the assessment of the qualities of new decomposition approaches. It supports several extension points that can easily be adapted according to a strategy. The tool is evaluated through the integration of two strategies, during which the focus will be on extensibility, pluggability and performance.

Keywords

Monolith decomposition strategies, microservices, experimentation environment, metrics.

Resumo

Ao longo dos os anos, múltiplas abordagens foram propostas para a automação da identificação de microserviços em monólitos. Cada uma destas abordagens seguem diferentes estratégias, desde o modo como os dados do monólito são coletados, ao algoritmo usado na identificação, tal como o método de visualização dos microserviços. Por outro lado, parece não haver uma estratégia claramente superior, levando à experimentação e comparação entre estratégias. No entanto, não existe nenhuma ferramenta que facilite tais testes e que contenha também mecanismos de avaliação, já que durante o desenvolvimento deste tipo de ferramentas, essas propriedades não são prioritárias. Este artigo propõe uma ferramenta extensível com capacidade para múltiplas estratégias de decomposição, desenhada a partir de uma sequência de etapas de identificação de microserviços de modo a promover a comparação de abordagens de decomposição e ajudar futuros investigadores na avaliação das qualidades de novas abordagens. Esta sequência de etapas suporta múltiplos pontos de extensão que conseguem ser facilmente adaptados de acordo com a estratégia escolhida. A ferramenta é avaliada com a integração de duas novas estratégias, onde é dado especial foco à extensibilidade, facilidade na adição de funcionalidade e desempenho da ferramenta.

Palavras Chave

Estratégias de decomposição de monólitos, microserviços, ambiente experimental, métricas.

Contents

1 Introduction **1**

1.1 Problem . . . 2

1.2 Research Question . . . 3

1.3 Approach . . . 3

1.4 Organization of the Document . . . 4

2 Related Work **5**

2.1 Stages . . . 6

2.1.1 Collection . . . 6

2.1.2 Decomposition Generation . . . 7

2.1.3 Quality Assessment and Comparison . . . 8

2.1.4 Visualization . . . 8

2.1.5 Editing and Modelling . . . 9

2.2 Existing Tools . . . 9

2.2.1 Summary . . . 9

2.2.2 Mono2Micro . . . 11

2.2.3 MonoBreaker . . . 11

2.2.4 IBM Mono2Micro . . . 12

2.2.5 Visualization Tool . . . 12

2.2.6 Process Mining Decomposition Framework . . . 13

2.2.7 Service Cutter . . . 13

2.2.8 Microservice Extraction . . . 14

2.3 Variety of Approaches . . . 14

3 Tool Design **16**

3.1 Pipeline . . . 17

3.1.1 Collection . . . 17

3.1.2 Decomposition Generation . . . 18

3.1.3 Visualization . . . 19

3.1.4 Quality Assessment and Comparison . . . 19
3.1.5 Editing and Modelling . . . 20
3.2 Extension Points . . . 21
3.3 Design . . . 23
3.4 Additional Features . . . 25
4 Usage **26**
4.1 Choose Strategy . . . 27
4.2 Decomposition Generation . . . 28
4.3 Visualization . . . 30
4.4 Comparison Tool . . . 34
4.5 Decomposition Recommendation . . . 36
5 Evaluation **37**
5.1 Previous Framework . . . 38
5.2 Current Framework . . . 38
5.3 Extensibility . . . 39
5.4 Pluggability . . . 42
5.5 Performance . . . 43
6 Conclusion **49**
6.1 Conclusion . . . 50
6.2 System Limitations . . . 50
6.3 Future Work . . . 51
Bibliography **51**
7 Appendix **56**
7.1 Strategy Selection . . . 57
7.2 Create Decompositions . . . 61
7.2.1 Similarity Generation . . . 62
7.2.2 Decomposition Generation . . . 68

List of Figures

3.1 Pipeline stages: processes and products . . . 17
3.2 Decomposition Generation Stage . . . 18
3.3 Visualization Stage . . . 19
3.4 Quality Assessment and Comparison Stage . . . 19
3.5 Editing and Modelling Stage . . . 20
3.6 Extension Points In the Pipeline . . . 21
3.7 Design Structure . . . 23
3.8 Domain Structure . . . 24
3.9 Additional Features . . . 25

4.1 Codebase Creation . . . 27
4.2 Add Representations . . . 27
4.3 Generate Strategy . . . 28
4.4 Generate Similarity . . . 29
4.5 Generate Decomposition . . . 29
4.6 Decomposition View with Metrics . . . 30
4.7 Only Show Neighbours Operation and Speed Dial . . . 31
4.8 Entity Menus . . . 32
4.9 Different Edge Menus . . . 33
4.10 Search Tool . . . 33
4.11 Additional Operations . . . 34
4.12 MoJoFM in Comparison Table . . . 35
4.13 Statistics in Comparison Table . . . 35
4.14 Recommendation . . . 36

5.1 Previous Tool Framework . . . 38
5.2 Tool Framework . . . 39

5.3 Decomposition Generation Extension 40
5.4 Decomposition Extension 41
5.5 Implementation of Additional Features 42

List of Tables

2.1 Decomposition Tools Comparison 10

5.1 Single Decomposition Generation 44
5.2 Multiple Decomposition Generations in Same Codebase 44
5.3 Clusters View Booting Time 45
5.4 Clusters View Booting Time With Saved Positions 45
5.5 Merge Operation and Clusters Redraw Time 46
5.6 Number of Clusters Comparison 47
5.7 Repository View Booting Time 48
5.8 Expand All in Accesses View 48

Listings

7.1 Codebase Class 57
7.2 Codebase Creation Service 58
7.3 Strategy Class 58
7.4 Strategy Creation Service 59
7.5 Abstract Representation 60
7.6 Representation Factory 61
7.7 Similarity Creation Service 62
7.8 Similarity Factory 62
7.9 Similarity Extension 63
7.10 Similarity Matrix Generation 64
7.11 Abstract Weights 65
7.12 Repository Weights 66
7.13 Extension of Similarity Forms 67
7.14 Create Decomposition Service 68
7.15 Decomposition Request 68
7.16 SciPy Clustering Service 69
7.17 Abstract Decomposition 70
7.18 Extension of Decomposition 70
7.19 Metric Extension Example 71

Acronyms

DTO	Data Transfer Object
UI	User Interface

1

Introduction

Contents

1.1 Problem 2
1.2 Research Question 3
1.3 Approach 3
1.4 Organization of the Document 4

Although monolithic applications simplify the implementation of the application business logic because of the existence of a shared domain model that is managed by a single transactional system [1], it also impairs the application's independent scalability of the business functionality and it does not promote development into the small agile teams. Therefore, development and deployment slows down, agile development approaches become impractical, and the transition to newer technologies becomes harder, making the monolith increasingly more outdated.

To address this problem, the microservice architecture is being adopted [2, 3]. As a result, there has been an increasing number of monolith applications that are being migrated to the microservices architecture [4]. However, one of the most difficult problems of these migrations is the identification of the microservices that should decompose the monolith [5].

1.1 Problem

Several approaches have been proposed to the identification of microservices in a monolith [5]. They vary in terms of the data they collect from the monolith, as well as how that data is processed. For instance, some microservice identification approaches do a static analysis of the monolith codebase, e.g. [6, 7]. This typically does not require any additional work other than the execution of a collection tool. The same cannot be said from a dynamic analysis, which involves a more elaborated setup, with the instrumentation and execution of code [8, 9]. Other approaches avoid this entirely and go for a high level model of the monolith, e.g. [10, 11].

On the other hand, when considering the produced decompositions of some approaches, they are composed of clusters that contain the monolith's classes or persistent domain entities for the candidate microservices, e.g. [6, 7, 11], while others consider clusters composed of the monolith's business functions, e.g. [12, 13]. The priority of grouping between elements also highly varies. While some approaches try to minimize the number of distributed transactions [7], others group according to the semantic similarity or the common authors in the development history [14, 15].

There is also the question of which clustering algorithms to use. Since there is a large variety to choose from and while some let the architect pick the number of microservices to decompose to (which is the case for hierarchical clustering algorithms), others do not (some of the community finding algorithms). This does not necessarily mean that one approach is better than the other, since in some cases, the architect might prefer a recommendation on the number of clusters.

So, the problem that now emerges with the constant increase of new microservices identification approaches is that there is no easy way of assessing and comparing each decompositions' qualities and approaches, since they are encapsulated into their specific implementation. In the development of new microservice identification approaches, the comparison is left as an afterthought (usually as

an external procedure), since it is in fact not the main focus in the development of these approaches, e.g. [16, 17], and thus not prioritizing flexibility and extensibility. For instance, they support a single type of data describing the monolith, a specific microservice identification algorithm, others contain specific metrics to evaluate the quality of the candidate decompositions while others do not focus on metrics usage, *etc.*. Even when using approaches that provide a visualization of the decomposition, some of them provide further tools to customize the decomposition while others do not.

1.2 Research Question

Considering the previous problem, we can now take a broader view of the available approaches and see that a common processing pipeline is present, according to the following stages:

1. collection of the monolith data;
2. generate candidate decompositions;
3. assess the candidate decompositions' qualities and compare between candidate decompositions;
4. visualize a candidate decomposition;
5. edit/model a candidate decomposition.

Having the pipeline in mind, the following research question is proposed:

- **RQ:** Is it possible to design and implement a tool that provides an experimentation environment for multiple strategies that identify microservices in monolith systems?

To answer this question, the pipeline needs to be integrated in a microservice identification tool and tested by introducing new microservice identification approaches.

1.3 Approach

In this thesis, we describe the work done on the extension of the Mono2Micro tool[1], in order to incorporate the different stages of the microservices identification pipeline, increasing its flexibility to support a wider range of microservices identification approaches.

The goal is to provide an experimental environment that developers and researchers can use and extend to analyse monolith's decompositions according to different approaches. Therefore, we describe the refactoring of Mono2Micro to support different approaches while achieving the following qualities:

[1] https://github.com/socialsoftware/mono2micro/tree/telmo-lopes-thesis

- **Pluggability**: Easy modification of the microservices identification pipeline to support new stages;
- **Extensibility**: The ability of easily adapting the existing stages of the microservices identification pipeline.

Additionally, the tool should provide an acceptable performance when an architect is interacting with it to experiment candidate decompositions of the monolith, even in the case of the analysis of large monolith systems.

The refactoring is driven by the design of each of the stages of the microservices' identification pipeline and by the definition of their extension points. A modular approach is used to decouple the stages and isolate the extension points.

The tool is evaluated to assess whether it achieves each one of the following qualities:

- **Pluggability**: Impact and effort of the addition of a stage;
- **Extensibility**: Impact and effort of adapting an existing stage;
- **Performance**: The operation latency while the architect is interacting with the tool.

The refactorization of Mono2Micro enabled an experimentation environment for researchers and architects, where different approaches to the identification of microservices can be easily integrated and the candidate decompositions compared, while also providing guided modelling capabilities.

1.4 Organization of the Document

Chapter 2 starts by describing the most common stages of a microservices identification pipeline. Then, it takes a closer look at how these stages are reflected in some existing tools. Chapter 3 further analyses the stages of this pipeline and goes onto discussing their extension points and design. After that, it discusses the introduction of some additional features, which serve as a test for the pluggability. With the tool's design defined, an example of the tool's usage is show in Chapter 4, where some of the extension points as well as additional features can be seen in practice. After considering the tool's usage, the implementation is evaluated in Chapter 5 according to the three qualities, pluggability, extensibility and performance, where pluggability and extensibility relate to the discussed information in the tool's design and the performance to the usage. Chapter 6 concludes this thesis and reasons about some of the improvement points as well as the future work in the extension of the tool. Finally in the Appendix, we take a closer look at the implementation of the tool. Its consultation is recommended after reading Chapter 4, to better understand some of the tools' behavior.

2

Related Work

Contents

2.1 Stages 6

2.2 Existing Tools 9

2.3 Variety of Approaches 14

Over the years, researchers have been proposing new approaches to identify microservices in a monolith. With the analysis of these approaches, it was concluded that amongst multiple tools, five stages were most frequently used, with some of them already investigated [5]. We will take a closer look to these five stages, which will then be mapped to the researched tools.

2.1 Stages

This section contains five subsections, which correspond to the most common five stages of the microservices identification pipeline. They will be described in the order they are usually met when generating, analysing and modifying decompositions.

2.1.1 Collection

The first stage is responsible for the collection of data to represent the monolith. The type of information can range from artifacts present in the source and runtime behaviour up to high-level abstractions such as business process models and use cases.

- **Source Code** [7, 18, 19]: Used to extract artifacts such as dependencies between elements, extract reusable services and map source code to other artifacts;
- **Database** [20]: Artifacts such as database contents, schemas and transactions provide relevant information about the services that manage the persistent data of the system;
- **Log Traces** [8, 21]: Depict the dynamic behaviour of systems. This is done by extracting runtime artifacts of the executions of the systems. This can either be done manually, or most commonly, by instrumenting the source code;
- **User Interactions** [19]: Capture the relation between users and system's functionalities through the user-interface inputs. Knowledge about the underlying business logic can also be extracted from its user interfaces;
- **Business Process Model** [22]: Describe the sets of activities and tasks that accomplish an organizational goal at a high-level of abstraction;
- **Use Case** [23]: They depict functional requirements and sequences of actions that can be used for service identification. This is done by helping the identification of the interactions between users and systems, at a high-level of abstraction, in order to achieve goals;
- **Activity Diagram** [18]: Describe steps involved in task execution. Frequently mapped to other approaches (such as **Use Case**) or used alongside other approaches;

- **Data Flow Diagram** [24]: Graphically represents functional dependencies based on the source code of software systems. Gather coarse-grained processes to form a service;
- **State Machine Diagram** [25]: Represents the dynamic view of a system by describing the different states that entities can have. Impractical for large systems;
- **Ontology** [26]: Structured set of terms representing semantics of a domain, whether through the metadata or elements of a knowledge domain. Ontologies are mapped to source code in order to identify candidate services;
- **Human Expertise** [18, 27, 28]: Manual parameter tuning for service identification algorithms, definition of data flow diagrams to then identify candidate services, manual analysis;
- **Documentation** [29]: Documentation for the system at different levels of abstraction, usually points to key functionalities, providing hints for service identification;
- **Process Mining** [30]: Based on an extended version of log traces, containing information from the user (such as a click in a user interface), or from an entry point (such as an API), up to the database accesses, until the results are returned back to the client.

2.1.2 Decomposition Generation

Once the collection of the monolith data is done, follows the generation of the candidate decompositions. This is done by applying identification techniques [5] to the monolith data:

- **Wrapping** [13]: Encapsulates the legacy system with a service layer without changing its implementation by providing access to the legacy system only through a service encapsulation layer that only exposes the functionalities desired by the software architect;
- **Genetic Algorithm** [18, 28]: Meta-heuristic for solving optimization problems based on ”natural selection”. Relies on the calculation of a fitness function to reach an optimal solution. This method can then be used to find a decomposition considered an optimal solution by the fitness function;
- **Formal concept analysis** [20]: Data analysis' method where implicit relationships are derived between objects in a formal way. A formal concept is defined as a grouping of all the elements that share a common set of properties. Rely on ontologies;
- **Clustering** [7, 31]: Partition data into clusters that share common properties. The clusters are built based on the internal homogeneity of their elements and the external separation between them.

2.1.3 Quality Assessment and Comparison

After identifying multiple decompositions, follows the assessment of said decompositions through the use of qualities, represented by metrics that can be divided into four major groups:

- Coupling [32]: Dependency level among services. The encapsulation of the services should be prioritized to reduce the impact of the changes to other services;
- Cohesion [33]: Measures the strength of the relationship between programming entities implementing the functionality provided by the service;
- Size [34]: Multiple metrics are based on the size of certain components such as, for example, services. A balance must be found since a larger size makes it harder to maintain e.g. [10];
- Complexity [34]: The amount and variety of internal work done by a service as well as the degree of interaction with other services. The higher the complexity is, the harder the maintainability.

If no metrics are provided, it usually requires the architect's careful analysis of the dependencies found in the gathered information. In works like IBM's Mono2Micro [35] and Visualization Tool [36], some metrics are referred but not integrated in the tool for instant analysis. They are usually used *a posteriori* to compare with other solutions from another tools.

2.1.4 Visualization

To better reason about the decompositions, different ways of visualizing the information about said decompositions have been proposed:

- Cluster graph [7, 10]: Usually represents a view of the microservices, each represented by a cluster node. The nodes connect through edges and represent the dependencies between microservices;
- Class/entity graph [16, 17, 36]: Each node represents a class and is connected to other classes through an edge, which represents the dependencies between them. Each node has some visual element related to him that indicates the microservice he belongs to;
- Sequence of accesses: Represents the sequence of accesses made through each microservice in order to accomplish a task. It is therefore more focused around the sequence of accesses and less on the microservices. Some representations are based on the static analysis of the source code e.g. [37, 38] by presenting the sequence of local transactions made to each microservice, while others are based on the dynamic analysis through log traces e.g. [30, 39], where the accesses to classes and methods are the main focus.

2.1.5 Editing and Modelling

Even though visualizations provide a closer look at the interaction between components, they leave room for improvement, since most often than not, decompositions can be improved or they do not meet the architects' expectations. With this, comes the editing and modelling of the decomposition through operations:

- Create a new microservice [36];
- Move selected class into another microservice [36];
- Clone the selected class into all the microservices that communicate with the class [36];
- Split a microservice by moving selected entities into a new microservice [7];
- Transfer the selected entities of a service into another microservice [7];
- Merge microservices into one single microservice [7];
- Duplicate classes or methods into another microservice [30];
- Filter from the visualization [39];
- Search elements in the visualization [39].

2.2 Existing Tools

Each tool presented next is divided into five points, those points corresponding to the most frequent five stages previously identified in the microservices decomposition pipeline. The tools analysed were Mono2Micro [7, 21, 37, 38, 40], MonoBreaker [16], IBM's Mono2Micro [17, 35, 41], Visualization Tool [36], Process Mining Decomposition Framework [30] Service Cutter [10] and Microservice Extraction [14, 15].

2.2.1 Summary

A recap of the following analysed tools and their key differences are presented in table Table 2.1.

Table 2.1: Decomposition Tools Comparison

Tools	Collection	Decomposition Generation	Quality Assessment and Comparison	Visualization	Editing and Modelling
Mono2Micro	Domain entities and Functionalities call graph (static) Trace logs (dynamic)	Hierarchical clustering algorithm (groups domain entities to minimize the number of distributed transactions per functionality)	Cohesion, complexity, size, coupling metrics. Decompositions detailed comparison Visual representation of cluster coupling	Graph with clusters as nodes and invocations between clusters as edges Functionality call graph	Merge, split clusters Transfer entities between clusters Rename cluster Redesign functionalities' transaction sequence
MonoBreaker	Database models, their relations and accesses points (view) (static) Trace logs (dynamic)	Community detection algorithm (groups views/endpoints and database models based on their coupling)	Visual representation of coupling between views and database models	Graph with views and database models as nodes and dependencies (coupling weight) as edge	Not discussed
IBM Mono2Micro	Metadata extractor collects classes and classes' dependencies in source code (static) Trace logs (dynamic)	Hierarchical clustering algorithm (groups by class relations from execution traces while accounting for meta-data dependencies)	Coupling, cohesion size, complexity metrics (not integrated) Visual representation of class coupling	Graph with classes as nodes and dependencies (call relations) as edges	Transfer class Split cluster
Visualization Tool	Compacted calling context tree, relates function calls and filters library calls (dynamic)	k-means++ clustering (groups classes by content similarity) Calling context tree clustering (groups classes by commu-nication reduction)	Decompositions Comparison (not integrated) Visual representation of class coupling	Graph with classes as nodes and function calls as edges	Split cluster Transfer class Clone class
Process Mining Decomposition Framework	Trace logs (dynamic)	Manual grouping (groups classes, methods and database tables by analysing processes' execution traces)	Coupling and size metrics	Process view of microservices coordination including accessed classes, methods and database accesses	Merge/split processes's sub-paths by leaving/cloning class or method into same/different microservice respectively
Service Cutter	Machine-readable representation artifacts describing intermediate stages of analysis and design	Community detection algorithm (groups by dependencies according to a set of criteria)	Visual representation of nanoentities exposed and shared between services	Nodes as nanoentities and services Edges as dependencies	Edit decomposition by changing grouping priorities i.e. clustering algorithm and coupling criteria
Microservice Extraction	Collects repository data, mainly class files, change history and developers responsible for modifications	Minimum spanning tree based graph clustering: Logical (groups by chronological similarity) Semantic (groups by content's similarity) Contributor (groups considering number of contributors in common)	Complexity and size metrics (not integrated) Visual representation of class and cluster coupling	Graph with clusters and classes as nodes (clusters can be expanded) and dependencies as edges	Not discussed

2.2.2 Mono2Micro

1. Collects the domain entities' call graph per functionality by doing a static analysis of the source code and generate a similarity matrix describing the domain entities' distance between each other [7]. On a related work [21], dynamic analysis is also used, to log the trace of a functionality;

2. Create a dendrogram describing the cophenetic distance between domain entities and generate candidate decompositions with a clustering algorithm by applying a cut to said dendrogram. The domain entities are grouped by minimizing the number of distributed transactions per functionality;

3. Compare decompositions based on cohesion, coupling, size and complexity [40] metrics. Other methods include the visual representation of cluster coupling and the detailed comparison of decompositions (with tools such as MoJo [42]);

4. Visualize the relations between the clusters as a graph, where the nodes are clusters and the edges represent the invocations between clusters. Also possible to visualize the functionality access sequence through the clusters [37, 38];

5. Apply operations to the decomposition's clusters and functionalities through the visualizations, triggering metrics' recalculation and the graphs' modification. The operations include merge clusters, split clusters, transfer entities between clusters, rename clusters and redesign the functionalities' transaction sequence.

2.2.3 MonoBreaker

1. Extracts static information regarding database models, their relations and how they are accessed, then use dynamic analysis to extract the class files, methods used as the "entrypoint" of the system, the models used and queries made, in order to update the static analysis' weight;

2. Creates an undirected weighted edge graph based on the static and dynamic information collected and finds a decomposition by applying a community detection algorithm. Repeatedly remove edges from the graph, defining multiple decompositions in the process. This results in grouping the database models and access points based on their coupling;

3. No metrics are used but conclusions can be obtained from the visual representation of coupling between views and database models. There is also information about the multiple steps of the community detection algorithm that can be analysed;

4. Visualize in a graph the access points and database models as nodes and their dependencies as edges (based on the coupling weight);

5. Not the focus of this project, although it is mentioned that small edits can be made to the graph through the visualization tool, although not intended as a decomposition modelling tool.

2.2.4 IBM Mono2Micro

1. A metadata extractor collects information about classes and classes' dependencies in a static analysis of the source code, then collects information from a dynamic analysis by running user scenarios. For the dynamic analysis to happen, the source code is instrumented to create trace logs about the user scenarios;

2. Finds a candidate decomposition using an AI-based partition recommender that applies temporo-spatial clustering on execution traces (hierarchical clustering algorithm alongside temporal relations between classes). It then uses the classes' dependencies obtained in a static analysis to ensure if certain partitions need to be merged. Classes are placed in the same partition according to a similarity score;

3. Comparison through the analysis of the coupling between classes in the view. Coupling, cohesion, complexity and size metrics are mentioned but not integrated in the tool;

4. Visualize the recommended partitions, where the classes identified during user scenarios represent the nodes and the runtime call relations between classes represents the edges;

5. Graph's customization is available alongside with the visualization. Classes not detected with the user scenarios can now be added to certain partitions. Operations such as transfer class and split cluster are present.

2.2.5 Visualization Tool

1. Uses function's relations to build a compacted calling context tree (filters out library calls). The dependencies are obtained from the instrumented source code. Multiple profilers are suggested to provide the instrumentation of the source code based on a dynamic analysis;

2. Suggests two approaches to find candidate decomposition, first by semantic-based clustering, which groups classes based on the similarity of their contents using a k-means++ algorithm, and the second approach, based on the calling context tree, by reducing the amount of communication between classes, using a calling context tree-based clustering;

3. Comparison through the analysis of the coupling between classes in the view. Decomposition comparison measure (used to compare with expert decomposition) mentioned but not integrated in the tool;

4. Graph visualization obtained from the calling context tree, represents the classes as nodes with their color corresponding to the cluster they belong to, and the function calls as the edges. Also allows the visualization of the source code when selecting the respective class;

5. Graph's customization is available alongside with the visualization. Operations change the decomposition's structure and dependencies between clusters. Operations such as split cluster, transfer class and clone class are present.

2.2.6 Process Mining Decomposition Framework

1. Collects log traces based on a dynamic analysis from the interaction with the user interface or from any entry point of the system. The information collected is related to each class and method traversed, with information about the entry and exit of functions as well as database accesses and timestamps. Achieved by instrumenting the source code;

2. With the log traces extracted, the business processes mined in the log traces are graphically represented by showing each class and database table used in the business processes connected by arrows. Circular dependencies between classes need to be removed and once that is achieved, the decomposition is found by visually inspecting said execution paths and manually grouping the classes in order to define the services. Both the removal of circular dependencies and the class grouping is based on human expertise, although it is mentioned that it can be automated;

3. The architect is helped in assessing the quality of the decomposition with the use of coupling and size metrics;

4. The only visualization available is by analysing the execution path accessing each class, method and database, represented by a node. The visualization is represented in a graph where the edges represent the frequency of the calls between the respective nodes;

5. During the usage of the view, the architect can merge the processes' subpaths by leaving the classes or methods into the same microservice, as well as split the processes' subpaths by cloning the classes or methods into a different microservice.

2.2.7 Service Cutter

1. Collects machine-readable representation artifacts describing intermediate stages of analysis and design, ranging from entity-relationship modules to domain-driven design entities and use cases;

2. With the collected system specification artifacts, a graph is produced, relating nanoentities. A community detection algorithm is then applied to group according to a set of criteria;

3. The architect assesses the qualities of the decomposition based on the analysis of the visual representation of nanoentities exposed and shared between services;

4. Graph representation of each service and its respective nanoentities connected by an edge. Dependencies between services also connected by an edge;

5. In the visualization, the grouping priorities can be changed in order to represent a new decomposition. Some of the operations can alter the community detection algorithm and the coupling criteria.

2.2.8 Microservice Extraction

1. Clones the projects' repository and then proceeds to collect data such as class files, the history of changes in a file and the developers responsible for said modifications;

2. Three decomposition methods are available, all based on a minimum spanning tree-based graph clustering. The first method groups classes by their chronological similarity, which means two classes are more likely to get grouped if modified at the same time. The second method groups by semantic similarity, which works by analysing its contents such as the names of each class. The third method groups based on the number of contributors in common;

3. The architect can assess the cluster and class coupling from its visual representation. Complexity and size metrics were also mentioned but not fully integrated in the tool;

4. Visualization of a graph with clusters and classes as nodes and dependencies as edges. The visualization is also able of expanding the clusters into its respective classes;

5. Not the focus of this project;

2.3 Variety of Approaches

Considering the previously identified stages and the corresponding tools, one can notice the divergence between methodologies and some of the most common strategies of each stage.

When looking at the collection stage, the most common approaches consist either in the analysis of the code structure (static analysis) or the analysis of the sequence of accesses during execution (dynamic analysis). Some less common cases also use information present in the project's repository, such as the change history.

As for the decomposition generation, even though multiple identification techniques were mentioned, clustering is the most predominant technique when automating the decomposition generation in a tool.

In clustering, the elements of the monolith are grouped based on an aggregation criteria. However, the application of each criteria can differ substantially with the use of different clustering algorithms.

With the generation of the candidate decompositions, the architect needs now to assess their qualities and compare them. To achieve this, metrics can be used. However, the majority of tools do not focus on their direct integration into the pipeline. They are usually kept as an afterthought, used as a comparison mechanism against other tools. In some less common cases, mechanisms are even provided to compare decompositions inside the same tool.

As a more detailed and fine-grained alternative, the architect can instead rely on the careful analysis provided by the visualizations. They contain a depiction of the dependencies between the elements of a monolith. The granularity of the visualization can also vary, since some tools use, for example, classes as their nodes, while others take a more coarse-grained approach, focusing on the clusters.

Both options provide their advantages, one being more detailed while the other being easier to understand, respectively. They also vary according to the elements in focus, which in turn reflects the collected information (clusters, entities, methods, *etc.*). Another approach, which is usually related with dynamic analysis, is by representing the sequence of accesses. Because of this, there is a trade-off between focusing on the elements/clusters or the functionalities.

Finally, the editing and modelling describes which operations can be done on a decomposition, for instance, splitting and merging clusters, or transfer a class/entity between clusters. These operations depend on the visualization and provide further control to the architect.

As it can be noted, there is a wide range of approaches to accomplish each stage, but there is a noticeable pipeline common to all tools. The hassle appears when trying to compare different approaches/tools and their respective decompositions. Each tool uses different metrics or none at all, some focus on the entities, classes or clusters while others focus on the execution traces or call graphs.

Therefore, the goal is to design a tool to support different approaches and that can be adapted or extended to incorporate new ones. In such a tool, the existing stages could be extended with new approaches and new stages could be added, creating an experimentation environment for the comparison of different approaches.

3

Tool Design

Contents

3.1 Pipeline 17
3.2 Extension Points 21
3.3 Design 23
3.4 Additional Features 25

In the following sections, we will firstly identify the main stages of the microservice decomposition pipeline. Once that is done, the identification of the extension points is made and the tool's design presented. With the conclusions of said sections serving as a foundation, some additional features will be presented.

3.1 Pipeline

To define an extensible microservice decomposition tool, first we need to understand its pipeline, each process, their requirements and products. This analysis follows the sequence presented in Figure 3.1.

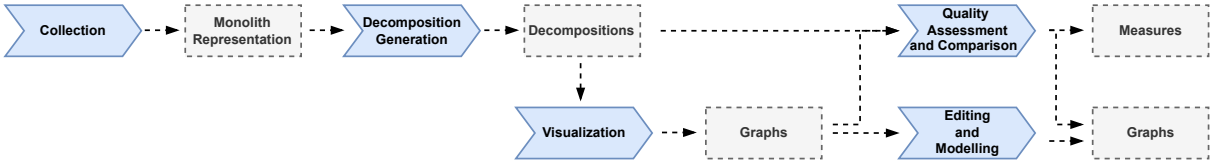


Figure 3.1: Pipeline stages: processes and products

3.1.1 Collection

The *Collection* stage is responsible for the generation of the *Monolith Representations*. Although its integration in the pipeline is possible and more practical to the architect, since a large variety of data collection tools as well as execution's tracing tools exist for different languages, it is more reasonable to not completely integrate this stage in the tool.

This comes with the advantage of the variety of information that can be used as input to the decomposition generator. Some of the examples are the monolith's persistent domain entities, the monolith functionalities' sequence of accesses and the relations between entities that were modified together.

One should also notice that even though some collectors do not require the execution of the monolith, which is the case when performing a static collection of the source code, other collectors, such as execution's tracing tools, that require the dynamic collection of information, usually need the previous instrumentation of the source code as well as its execution. Other methods such as the mining of the project's repository also have other prerequisites.

Because of this external requirements, the integration of the collector directly in the pipeline is further complicated and ends up limiting in the amount of versatility possible. By using a decoupled approach, the developer can freely use the collector of his choice without worrying with the possible structural impact to other available collectors or their mismatching requirements. With this in mind, the collectors' responsibility now solely becomes producing the Monolith Representation, which will be used as the input for the Decomposition Generation.

3.1.2 Decomposition Generation

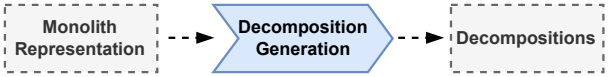


Figure 3.2: Decomposition Generation Stage

Following the collection of the Monolith Representation in the Collection stage, comes the *Decomposition Generation* stage. In this stage, the Decomposition will be generated based on the Monolith Representation, which contains the monolith elements as well as the relations between them. Amongst the most common monolith elements, we have classes, domain entities and methods. As for the relations between elements, they can be collected from different sources, such as sequence of accesses, source code, repository information, between others.

For the Decomposition to be generated, a decomposition algorithm is used, grouping the monolith's elements according to a certain criteria. A wide range of algorithms can be used, but hierarchical clustering and community detection algorithms are still amongst the most common. Although, as it was noticed in the Existing Tools section, other algorithms can be used, such as k-means++ and minimum spanning tree clustering.

As for the grouping criteria, it dictates how the clustering of the monolith elements should be prioritized. This is done by calculating the similarity level between monolith elements and heavily relies on the extracted Monolith Representation. Some of the criteria are recognized quite easily, such as when grouping by semantic similarity, which usually focuses on the class, attribute and method's names, or when analysing the similarities in code evolution, which usually involves finding similarities in the modification of the project along multiple commits, or similarities in the authors, grouping based on who modified said monolith elements.

Other criteria are harder to recognize, such as the criteria based on the monolith's static structure, which try to maximize modularity, and criteria based on the monolith's functionalities' sequence of accesses, reducing the amount of remote communication between microservices or the number of distributed transactions required to implement a functionality.

By the end of this stage, the generated Decomposition can proceed to two different stages. The first stage, the Quality Assessment and Comparison stage, intended to be evaluated by metrics or compared to another produced decompositions, or the second stage, the Visualization stage, where the architect can visualize the decomposition in order to understand the dependencies between monolith elements in finer detail. We will first take a closer look to the Visualization stage.

3.1.3 Visualization

Figure 3.3: Visualization Stage

With the Decomposition generated during the Decomposition Generation stage, follows its analysis. In the *Visualization* stage, this Decomposition is used to create a view. Since the goal of decomposing a monolith is to, in fact, group its elements, by providing a graph containing the dependencies between said elements, the architect can focus on the finer details of its decomposition while also having into account a large number of elements.

With that said, multiple views are possible, but among the most common are views that focus on the clusters, classes, domain entities and methods. These are represented as nodes while the edges represent their dependencies. The dependencies can usually be related to the criteria used during the Decomposition Generation stage.

Another type of view also common focuses on the monolith's functionalities and how they are mapped to the decomposition. In this case, nodes usually represent the clusters, classes, domain entities and methods accessed while the edges represent the sequence of accesses.

Because of the attention to finer details in this stage, it is not difficult to imagine seeing this tool being used as a comparison mechanism. This is indeed a viable option and used in the majority of tools reviewed in the Existing Tools section. This will be further discussed in the next section.

3.1.4 Quality Assessment and Comparison

Figure 3.4: Quality Assessment and Comparison Stage

In the *Quality Assessment and Comparison* stage, the Decomposition generated during the Decomposition Generation stage is used as input to this stage. Here, the Decomposition is analysed in order to assess its qualities as well as compared against other Decompositions.

Starting with the assessment of its qualities, the candidate Decomposition is assessed with the use of metrics. These metrics denote certain qualities of the Decomposition, although not all Decompositions

might be able to use the same metrics. The large majority of metrics belong to four groups, coupling, cohesion, size and complexity.

As for the comparison between Decompositions, there are techniques that can be used to observe how different two Decompositions are, such as MoJoFM [42], that calculates the minimum number of operations necessary to transform a Decomposition into another. Other techniques such as showing how different the same cluster is in both Decompositions, in terms of the monolith elements belonging to said cluster, can also be used. Also, in cases where Decompositions use the same metrics, they can be used to directly compare them, based on their qualities.

With that said, there is still the visual-oriented approach. With the use of graphical depictions of the Decomposition, which are usually found in the Visualization stage, the architect is provided with the dependencies between clusters or monolith elements. Some representations provide node highlighting [36], corresponding to the cluster they belong to, while others represent the amount of dependencies between two components by the thickness of the edge or the similarity between components by how distanced they are. So, by using different sizes, colors and distances, one can have an accurate visual representation of the Decomposition's quality.

With that said, when analysing two graphical representations of two different Decompositions, one can assess in which Decomposition the dependencies between clusters or monolith elements are handled better, or the trade-offs between both solutions. The downside to this method is the level of attention required to analyse both Decompositions in such a fine-detailed way.

3.1.5 Editing and Modelling

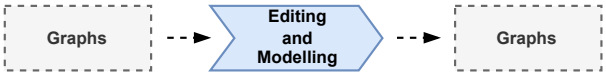


Figure 3.5: Editing and Modelling Stage

Once the architect carefully analyses the candidate Decomposition in the Visualization stage, it is to be expected that he will require further modifications, since, most often than not, the best grouping results, according to a criteria, do not correspond to the demands of the architect or the real-life limitations of the system. Because of this, the *Editing and Modelling* stage is introduced, which is aggregated to the graphs of the Visualization stage, since it eases the interaction between multiple elements and creates feedback cycles between the architect, his desired modification and the resulting graph.

So, by using operations, the architect changes the Decompositions' structure and the metrics are recalculated alongside this change, in order to update the architect about the current Decomposition's qualities. The available operations are deeply related to the component in focus on the view.

In views of clusters, the most common operations usually involve the splitting and merging of clusters

or the transfer of elements. In views focused on the functionalities' accesses, common operations involve the modification in the elements' sequence of accesses.

3.2 Extension Points

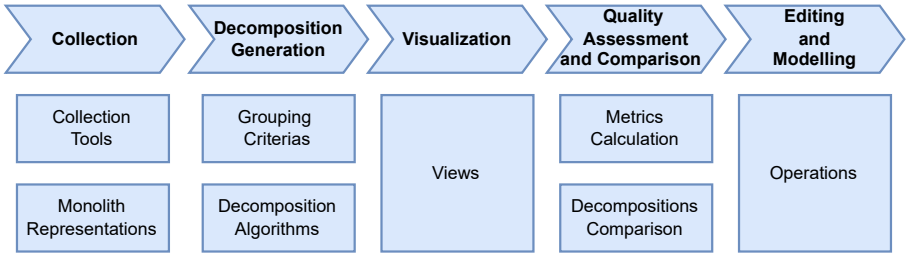


Figure 3.6: Extension Points In the Pipeline

With the microservice identification pipeline defined, follows a closer look at the variation points in the pipeline. In this section, the extension points of the pipeline will be identified and discussed, while aiming to improve the tool's extensibility and adaptability. The order in Figure 3.6 will be followed.

Starting by considering the *Collection* stage, there are two main implications during its extension. The first one being the extension of collectors and the second being the extension of the monolith's representation collected. As it was already identified, because of the external nature of collectors, a large variety of collectors can be used, but it isn't without implications. When choosing a collector, one must guarantee that the produced output is recognized as one of the supported monolith's representations.

A monoliths' representation is an abstraction needed when dealing with data collection, since microservices identification approaches can use a variety of sources. As an example, one can use two different collectors, each one extracting data from a different source but still produce the same representation. This is indeed the case when extracting the call graph's sequence, which is done with a static analysis, and when extracting the sequence of accesses, done in a dynamic collection [21].

Some examples of monolith representations are the sequence of accesses, the relations between files and the relations between developers and files in a repository. With this abstraction, the Collection stage can be decoupled from the Decomposition Generation stage, which will not need additional modifications as long as the format produced by the collector is accepted by the tool.

Now considering the *Decomposition Generation* stage, as it was already pointed out, different types of criteria exist, which makes them an extension point. Since the criteria uses the monolith's representation to dictate how the clustering should be prioritized, there is a dependency between these two extension points (this dependency being the agreement of the monolith's representation format). As for the decomposition algorithms, due to the variety of algorithms used to aggregate monolith's elements based on their relations, said algorithms also need to be considered an extension point.

Therefore, there is also dependency between the criteria and the decomposition algorithm in use, since an algorithm may require a particular codification of the criteria. For instance, an hierarchical clustering algorithm requires a similarity matrix as input, which implements a criteria.

As for the *Visualization* stage, the views available to a Decomposition depend on the type of information a Decomposition is composed of. For instance, one Decomposition can have a view composed of classes while other Decomposition a view composed of domain entities. For this reason, the Visualization stage can be extended with new views.

It should also be noted that one Decomposition type can have multiple views to its disposal. For example, the majority of Decomposition types should have at its disposal at least two views, one focusing on the clusters of the Decomposition while other focusing on the elements.

In the *Quality Assessment and Comparison* stage, there are two extension points, the metrics and the Decomposition comparisons. As previously pointed out, not all Decompositions might be able to use the same metrics. For this reason, metrics should be considered an extension point and should be shared whenever possible, for direct quality comparison between Decompositions. Because of this, there is also a dependency between each Decomposition type and metric.

As for the Decomposition comparison, the same logic is applied. Multiple approaches to Decomposition comparison are possible and because of it, should be considered an extension point. Some Decomposition comparison approaches also depend on the Decomposition types being compared.

Finally, in the *Editing and Modelling* stage, we have the *Operations* extension point. As previously mentioned, the available Operations depend on the type of Decomposition being analysed, which makes the Operation an extension point, required when adding new operations.

The extension of an operation can happen even when considering the same type of operation. If we take as an example the merge operation and two decompositions, where one Decomposition supports the duplication of elements while the other does not, the logic involved in the union of clusters will be different, since, as an example, the same element should not be placed twice in the same cluster.

3.3 Design

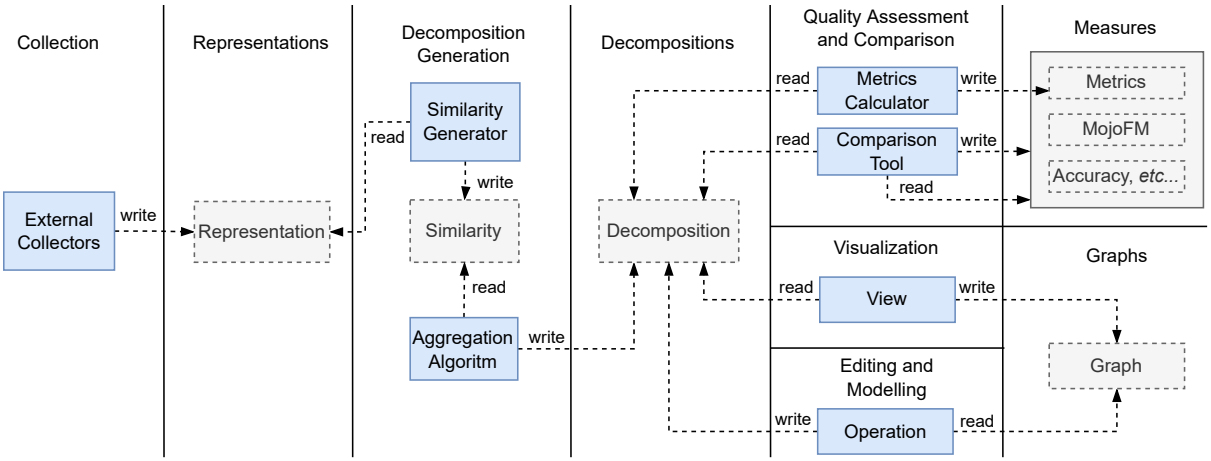


Figure 3.7: Design Structure

In this section we will present the design structure proposed for a microservice identification tool. Figure 3.7 represents this design structure. As previously mentioned, the collectors in the *Collection* stage are implemented as external modules (independent tools), since there is a large variety of data collection tools as well as execution tracing tools available. Because of this, collectors are decoupled from the tool and to integrate their output in the pipeline, the output needs to be produced according to a certain format, established by a *Representation*.

Therefore, there are two extension points, one associated with the collection, which is accomplished by external tools (represented by the *External Collectors* extension point) and the other being the file format (represented by the *Representation* extension point). Due to the JSON metaformat, new monolith representations can be easily added and new collectors integrated, but the constraint between these two extension points, the agreement in the file format, needs to be met.

The *Representation* files (consider now Figure 3.8) are stored inside the context of a *Codebase* entity, which denotes the monolith. Each Codebase can contain several representations.

The *Decomposition Generation* stage (Figure 3.7) is implemented by two modules and a file. The modules implement two extension points, those being the *Similarity Generator* for the criteria and the *Aggregation Algorithm* for the decomposition algorithm. They are decoupled through a *Similarity* file. Extending one of these two modules allows for the use of more criteria and algorithms.

The *Similarity Generator* module is responsible for creating the *Similarity* file, which requires that he reads and recognizes the monolith representations, given by Representation, to produce it. The Similarity file contains the similarity level (according to a criteria) between the monolith elements.

With the Similarity file generated, the Aggregation Algorithm can now use the similarity level between elements to produce the *Decomposition*. Much like the External Collectors and Similarity Generators, the

Similarity Generators need to agree on the format required by the Aggregation Algorithm, which means producing the correct Similarity. Not all Similarity Generators are intended to work with all Aggregation Algorithms, which further increases the necessity of using Similarities for decoupling the modules.

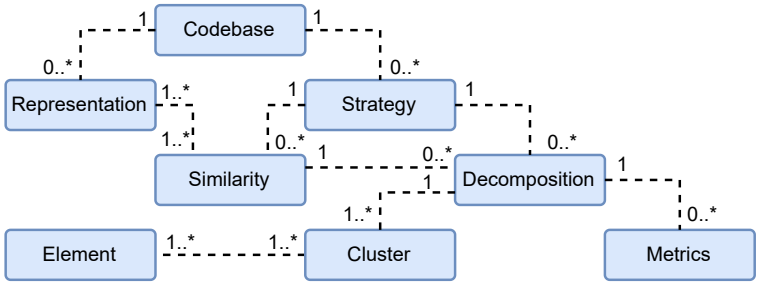


Figure 3.8: Domain Structure

The *Strategy*, represented in Figure 3.8, aggregates the implementation of several extension points by determining the type of decomposition to be generated and the monolith representation files required by the aggregation algorithm to generate the decomposition. Note that a strategy can have several *Similarity* files, since it is possible to combine different criteria to be fed into the algorithm. Additionally, it must also be highlighted that a Similarity file may require multiple Representations to be produced by the Similarity Generator.

Now taking a closer inspection to Figure 3.8, some details about the Codebase should be considered. As it was said, the Codebse denotes the monolith and when using multiple Strategies, it is sometimes possible that two different Strategies share some of the Representations. This avoids duplicating Representations, since they are shared inside the context of a Codebase. It should also be noted that once a Decomposition is produced, both the Strategy and the Similarity are related to said Decomposition.

Proceeding to the *Decompositions* stage, the *Decomposition* file decouples the *Decomposition Generation* stage from the other stages. Is is also an extension point, since several Decomposition implementations are allowed. The Aggregation Algorithm is responsible for filling the Decomposition with its *Clusters* and *Elements*, which are itself also extensible. For example, a type of Cluster might support duplication of elements while the other does not. As for the Elements, some might be classes and other domain entities. With this established, it is now only necessary that the consumers (the metrics, visualization and editing tools) comply with the generated Decomposition.

The *Decomposition* and its properties are presented by views, qualified by metrics, compared to other Decompositions and modified by operations. Since these functionalities are heavily dependent on the type of Decomposition, they are also extensible. They are represented in Figure 3.7 by the *View*, *Metrics Calculator*, *Comparison Tool* and *Operation* module, respectively.

3.4 Additional Features

With the design defined, it is now possible to consider its pluggability by integrating additional features in the tool. This features will be discussed according to Figure 3.9.

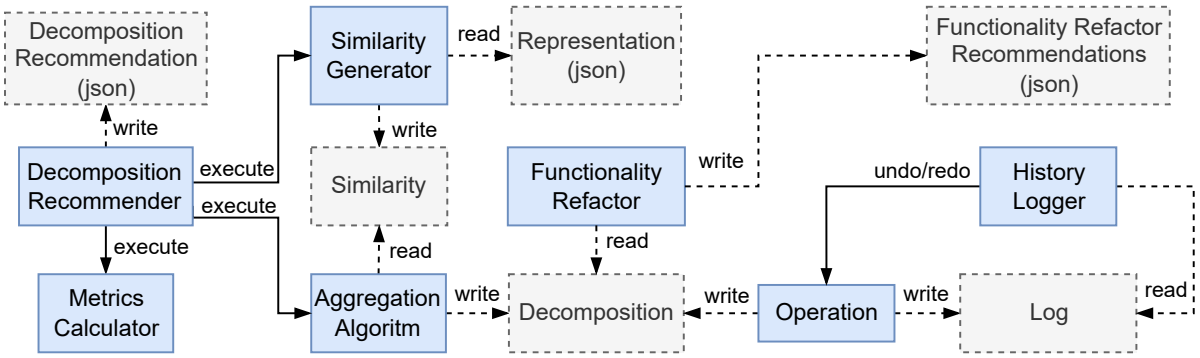


Figure 3.9: Additional Features

The *Decomposition Recommender* recommends decompositions based on their quality metrics. It invokes the *Similarity Generator*, *Aggregation Algorithm* and *Metrics Calculator* modules multiple times to generate the *Decomposition Recommendation* file, which aggregates the decompositions' metadata and their respective metrics. The architect can consult the results by ordering the decompositions according to their metrics. Not represented in Figure 3.9, a *View* module is also added to present the recommendations. The tool's design facilitates the addition of this feature.

The *Functionality Refactor* [37] uses as input a Decomposition. It implements an algorithm that generates refactorizations of the functionalities with low complexity values. This feature is also completely decoupled from the core design, depending only on the Decomposition's file format. This is because not all Decompositions contain information related to the monolith's functionalities.

During the *Editing and Modelling* stage refactorization, an *History Logger* was introduced, capable of undoing and redoing previous operations. Whenever an *Operation* is executed that modifies the *Decomposition*, the Operation's metadata is saved in a *Log*. When an undo is requested, the History Logger analyses the current history to find the operation in question to then generate its opposite operation and apply it to the decomposition. The redo is not as complex, since it only analyses the current history to find the operation in question and then apply it once again to the decomposition. Just like the previous additions, this new feature is easily integrated in the design core.

4

Usage

Contents

4.1 Choose Strategy 27
4.2 Decomposition Generation 28
4.3 Visualization 30
4.4 Comparison Tool 34
4.5 Decomposition Recommendation 36

In this chapter, we demonstrate the typical usage of this tool, according to the pipeline. First, we take a look at how to choose the strategy, then we produce a decomposition. Since there are two ways of producing a decomposition, either by choosing the weights or by recommendation, we take a look at both methods. Afterwards, the interaction with the view is shown. Through these views, the architect can inspect the decomposition and experiment with the impact of possible changes.

4.1 Choose Strategy

Before choosing the strategy, one must first create the context of the codebase being analysed. Because of this, the architect starts by creating a codebase. This can be seen in Figure 4.1.

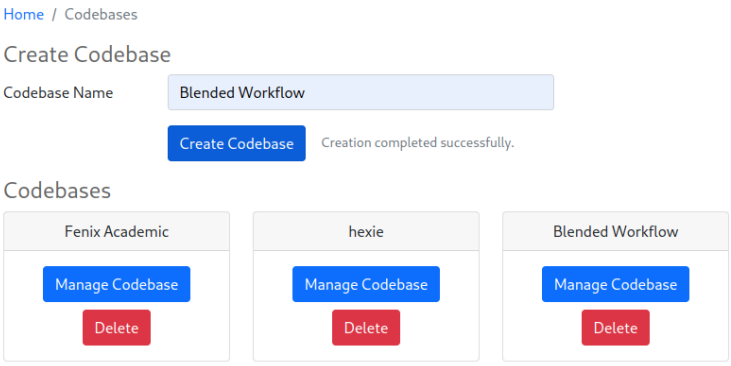
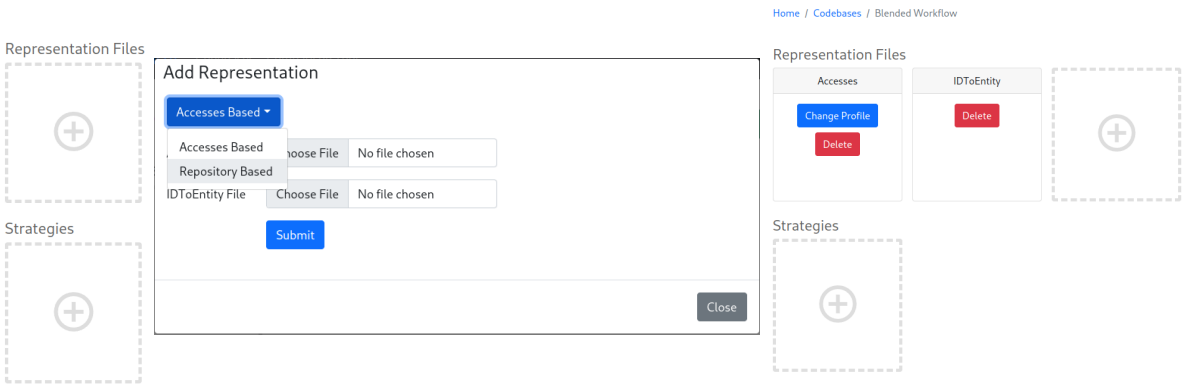


Figure 4.1: Codebase Creation

Once it is created, the architect proceeds to add the desired representations to include in the decomposition. To do this, the architect needs to click on *Manage Codebase*.



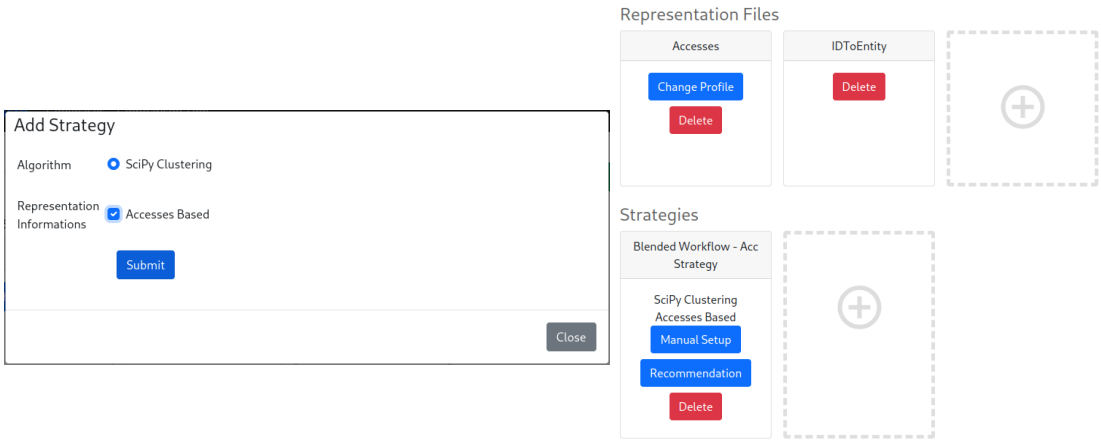
(a) Main Menu

(b) Adding Representations

(c) Representations Added

Figure 4.2: Add Representations

In this menu, corresponding to Figure 4.2(a), the architect will be presented with the Representations' Files added and Strategies generated. Since it is empty on creation, the architect starts by adding representations, which opens the window represented in Figure 4.2(b). In here, the architect selects the desired representation and adds its files, which once submitted, we can see its results on Figure 4.2(c).



(a) Strategy Options **(b)** Created Strategy

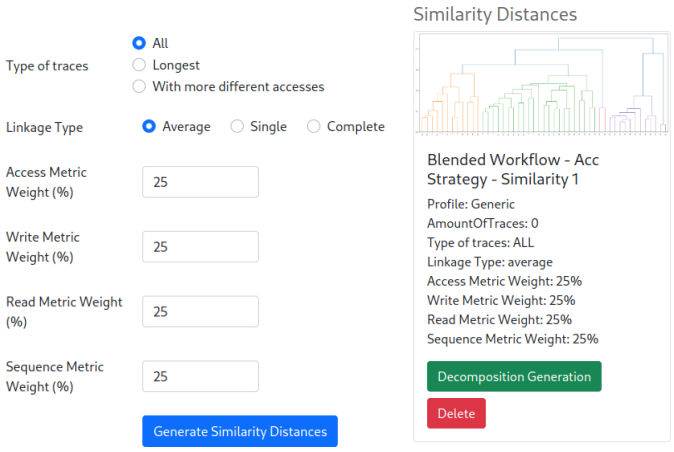
Figure 4.3: Generate Strategy

With the representations added, the architect can now generate its desired strategy. Once the "+" button is clicked, the strategy options appear, as presented in Figure 4.3(a). Three things should be kept in mind from this menu. First, since only one clustering algorithm exists, only one appears, and only one can be selected. It is possible to add multiple representations, but since only one was added (Accesses Based representation), only one appears. Third, each algorithm can only deal with certain representations, so, it might happen that the codebase has all available representations but only one appears, along with a specific clustering algorithm. With that said, the architect has now two options, *Manual Setup* (see Section 4.2) or *Recommendation* (see Section 4.5).

4.2 Decomposition Generation

When manually creating the similarity, the previous choice of strategy needs to be accounted for. Since the strategy requested the SciPy[1] clustering algorithm as well as an accesses-based representation, the information requested will be accordingly to said choices. In Figure 4.4(a) we can see part of the requested information, connected to the representations used, as well as the weights of the criteria. Once the architect fills the requirements, the similarity can be generated. The result of this generation is shown in Figure 4.4(b).

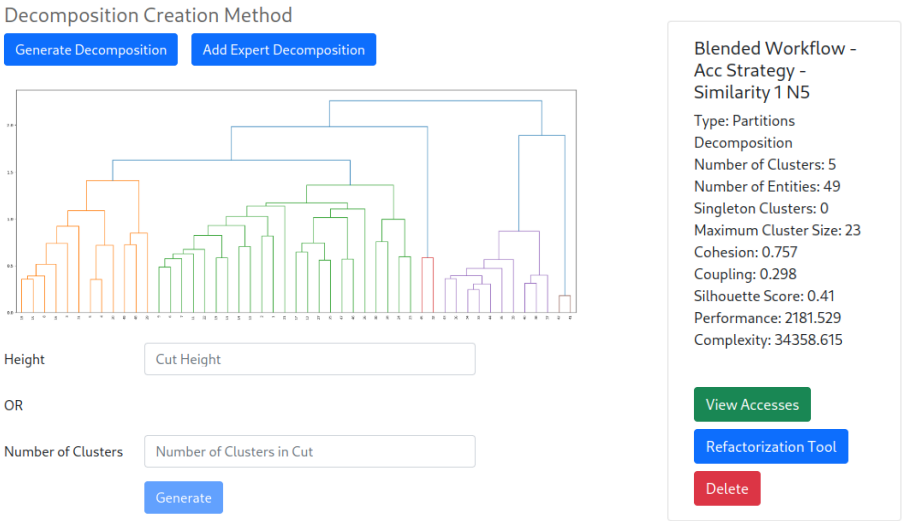
[1] https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html



(a) Similarity Options Example **(b)** New Similarity

Figure 4.4: Generate Similarity

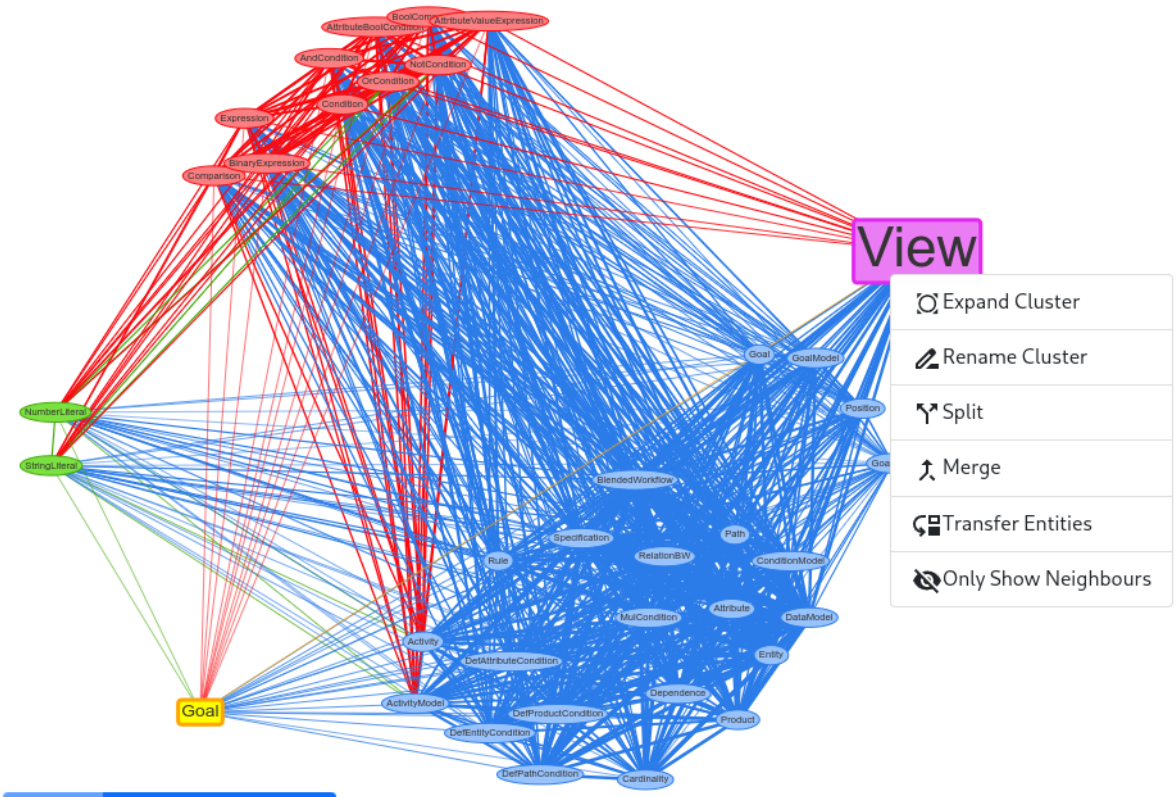
To conclude, we have the decomposition generation. It is accessed by clicking in *Decomposition Generation*, which will make the menu in Figure 4.5(a) appear. In this menu, the architect specifies information about the generated decomposition. In this case, the height of the cut or the number of clusters. It should be kept in mind that certain algorithms do not allow this precise behavior, for example, generating only with a certain amount of clusters. Because of this, both the similarity generation as well as the decomposition generation might require adding new information requests in the User Interface (UI). Form here, the architect can click in *View Accesses* to open the view in Section 4.3.



(a) Decomposition Options **(b)** New Decomposition

Figure 4.5: Generate Decomposition

4.3 Visualization



Cluster	Entities	Functionalities	Cohesion	Coupling	Complexity
BW	23	104	0.46	0.88	34358.62
Goal	2	4	0.88	0.05	29607.75
Conditions	10	22	0.92	0.24	159520.82
Literals	2	9	1	0.18	170655.67
View	12	33	0.52	0.14	43560.58

Figure 4.6: Decomposition View with Metrics

Figure 4.6 presents a decomposition graph based on accesses, composed of five clusters, where the rectangles represent the clusters and the ellipses represent the entities. Starting by the clusters, their vertical size is proportional to the number of entities it contains. Each cluster has its own color associated with it, which is picked randomly during the graph construction, although it is always different from the colors of the other clusters. Now focusing on the nodes, they inherit the color of its cluster and do not change the vertical size, since all entities have the same weight. As for the edges, their thickness

indicates the number of functionalities accessing both nodes, while its length represents the average cophenetic distance between the nodes. The cophenetic distance is obtained during the production of the dendrogram image. Their color is associated to one of the connected nodes.

Next, lets look at the menu above `View`. This menu contains some of the available operations for clusters. There are operations that change the decomposition, like `Merge` that merges two clusters, `Split` that lets the architect choose the entities to be extracted to another cluster, as well as `Transfer Entities` and `Rename Cluster`. All of this operations require the invocation of the backend and modify the persistent data. Other operations, such as `Expand Cluster` and `Only Show Neighbours`, only affect the view. `Expand Cluster`, when requested, updates the view to show each entity inside the selected cluster. This operation was applied to the three expanded clusters in the Figure. As for `Only Show Neighbours`, it isolates all the nodes and edges not directly related to the selected node. We can see this in Figure 4.7. To revert this, the `Show All` operation is used.

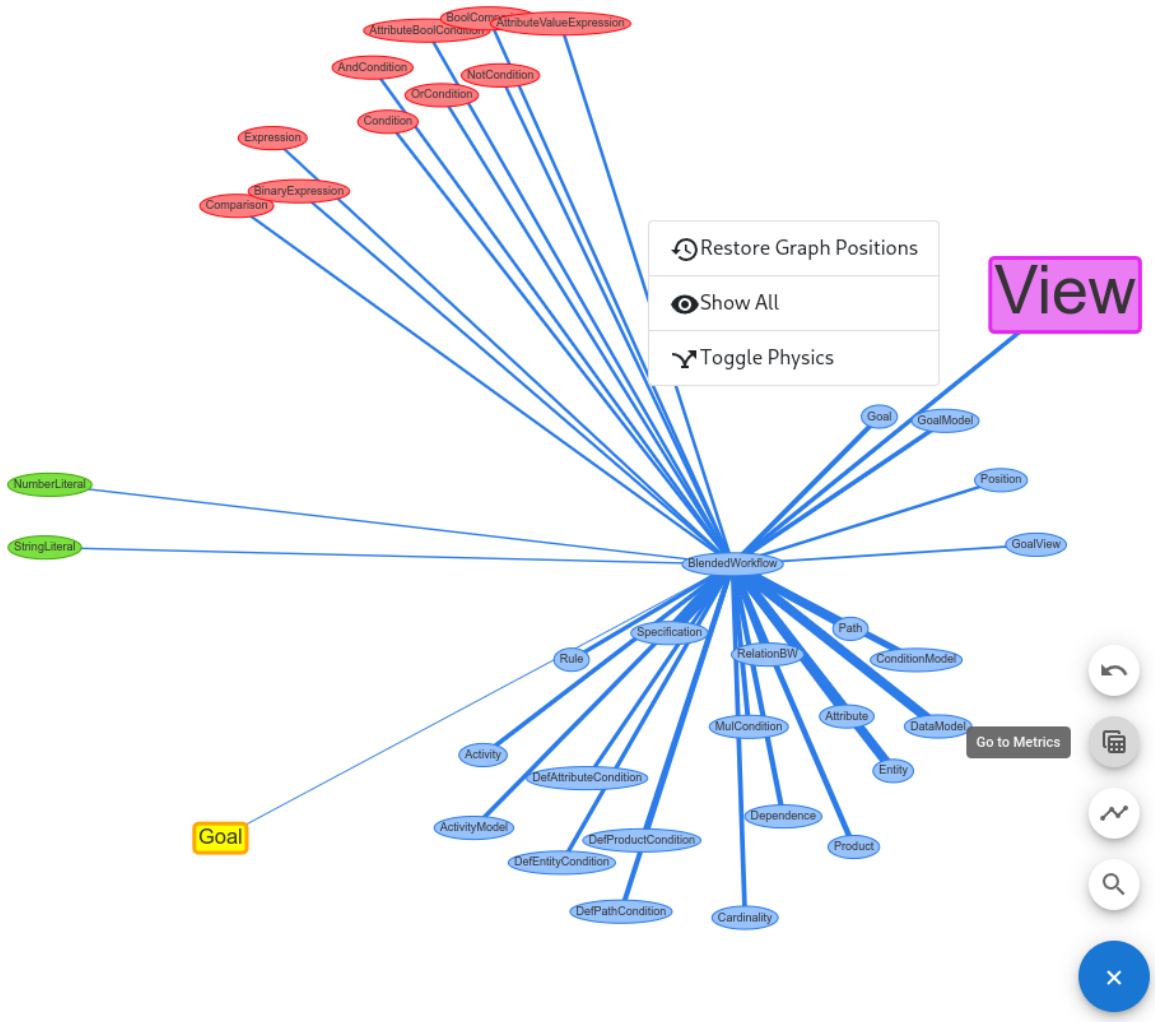
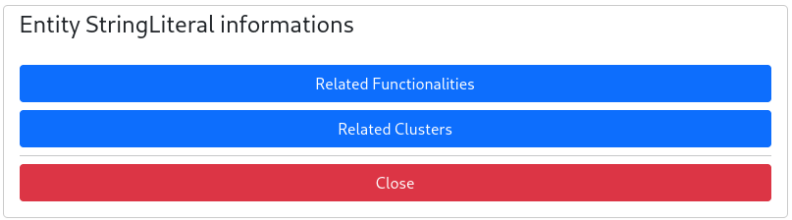


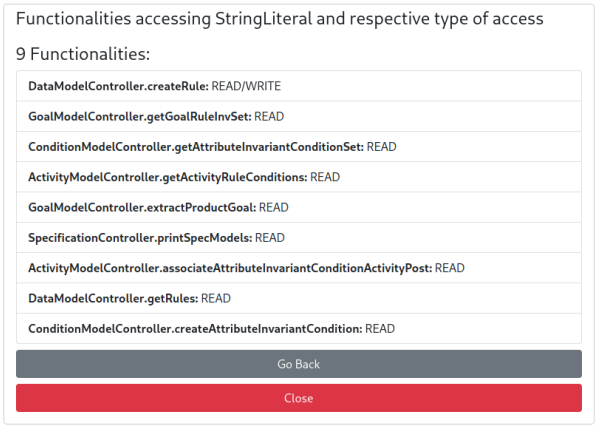
Figure 4.7: Only Show Neighbours Operation and Speed Dial

Another operation to have into account is `Toggle Physics`. Since the graph generation tool contains a physics model, it might be useful to deactivate it, so that nodes can be freely moved. Otherwise, the similarity distances are prioritized. The `Restore Graph Positions` operation will be discussed later.

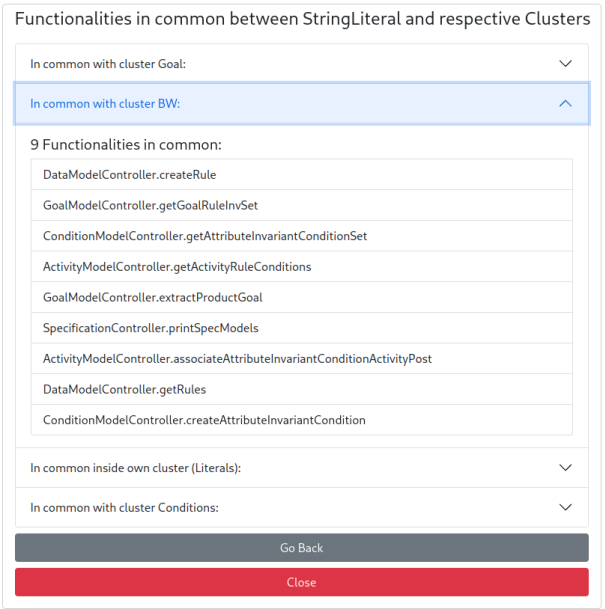
Going back to Figure 4.6, there is a table below the graph, which presents the clusters' metrics. When the decomposition is modified, the table is also updated accordingly. This table can be accessed through the circle with the "+" (speed dial), by clicking in `Go to Metrics`, as shown in Figure 4.7. This circle has other operations available, such as the undo/redo of operations, change to the functionality view and searching for a component. The undo/redo operation is part of the implementation of the aforementioned *History Logger* (see Section 3.4). Every time the decomposition is modified, the persistent data in the backend is updated, and with it, the operation's metadata is added into a log. When calling undo, based on said metadata, the decomposition is reverted to the state before the operation was applied. The inverse logic is applied with the redo. In both cases, undo and redo, the view is updated.



(a) Entity Informations



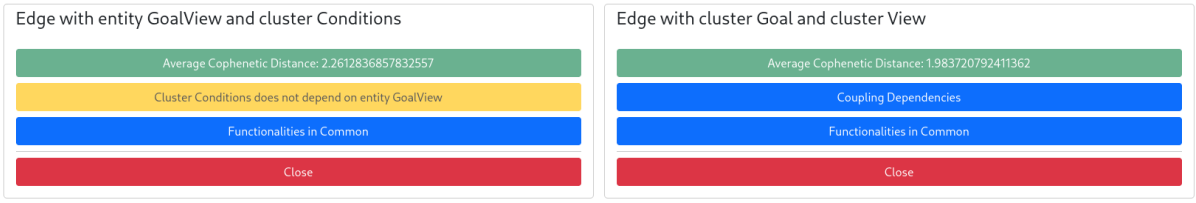
(b) Related Functionalities



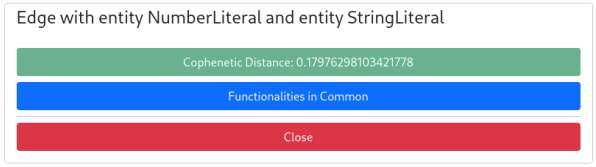
(c) Related Clusters

Figure 4.8: Entity Menus

Another thing to keep in mind is that entities, clusters and edges can be double clicked to obtain further informations. For example, when clicking in an entity, the menu in Figure 4.8(a) appears. Then the architect can select to go to the menu in Figures 4.8(b) and 4.8(c). Also, when clicking in an edge, the information varies according to the connected nodes. We can see this effect in Figures 4.9(a) to 4.9(c).



(a) Between Cluster and Entity **(b)** Between Clusters



(c) Between Entities

Figure 4.9: Different Edge Menus

As for the search tool, it is implemented with a fuzzy-search finder, and in the accesses view, it lets the architect search for entities, clusters and functionalities. Once the architect selects the desired item, the view focuses on the element like with the operation shown in Figure 4.7.

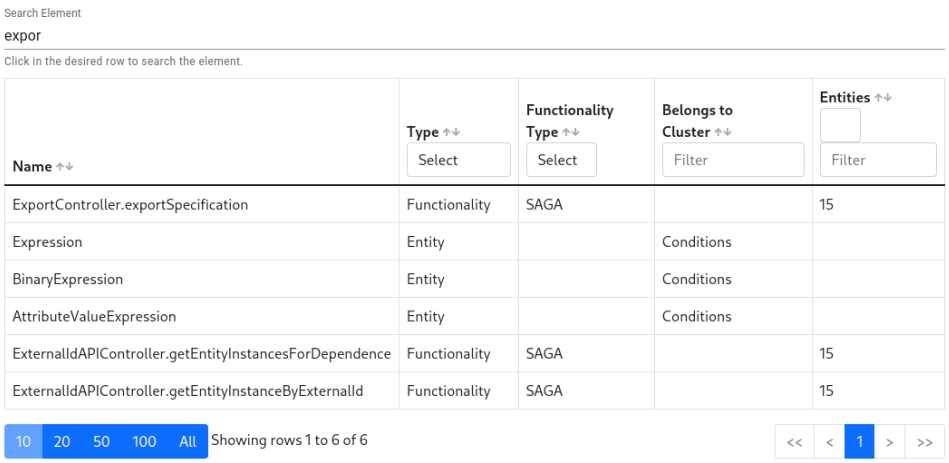


Figure 4.10: Search Tool

Before concluding this view, there are some additional features that should be considered. As it shown in Figure 4.11, there is a `Snapshot Decomposition` feature. Once requested, a copy of this

decomposition is done. This is useful when editing a decomposition without risking loosing it. It can also be combined with the undo/redo to save the progress, undo multiple steps and try other decomposition tactic. Another important feature is `Save Graph Positions`. With it, the architect can save the current positions of the clusters, entities and edges. The positions are associated with the operations' log, so if an undo is done, the previously saved positions will be restored. Also, if the architect does not like the current presentation of the components, the previous positions can always be restored with the `Restore Graph Positions` operation. The positions are also restored when reopening the view and can sometimes reopen faster when dealing with larger codebases (discussed in Section 5.5).

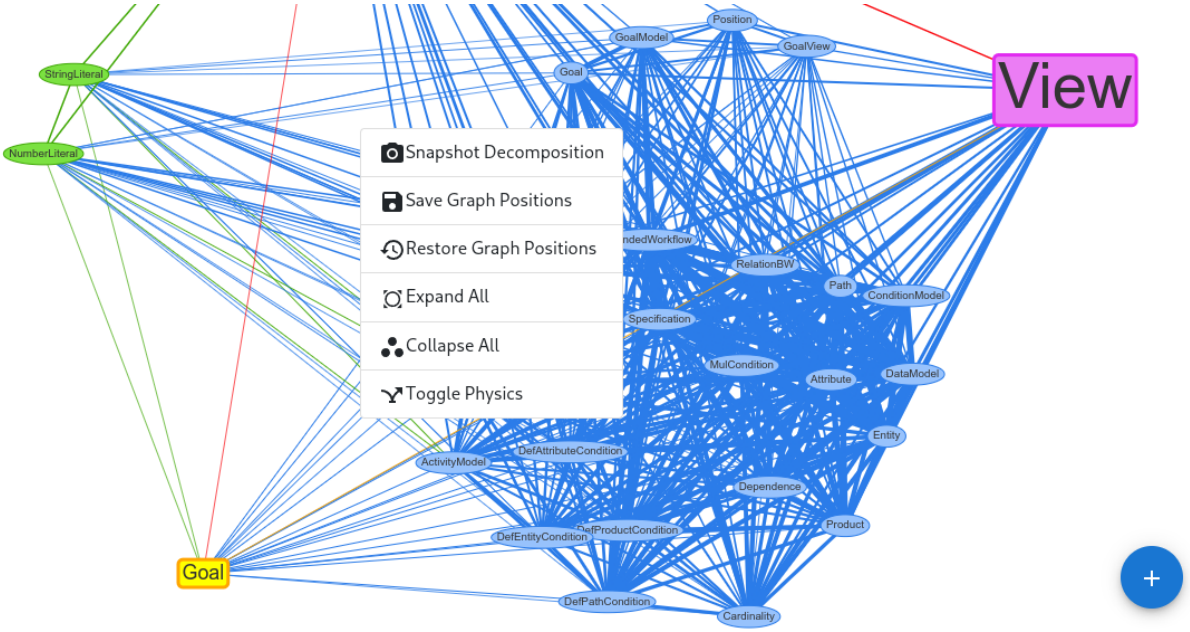


Figure 4.11: Additional Operations

4.4 Comparison Tool

According to what was discussed in *Quality Assessment and Comparison* 3.1.4, we provide some default comparison measures in the tool. In Figures 4.12 and 4.13 the comparisons are being made between two decompositions from different strategies, one with information about its accesses and another with information about its accesses as well as the authors and commits made in the codebase's repository. While the properties presented in Figure 4.12 are obtained from the execution of MoJoFM [42], which accesses each decomposition's clusters to compare them, in Figure 4.13 the properties are compared side to side, based on already obtained values, such as metrics and cluster sizes. If the compared decompositions don't contain the same metrics, which is the case for TSR, then *Not Present* replaces it.

Home / Comparison Tool

Comparison Tool

Codebase: bw

Source of Truth: bw - Accesses and Repository Decomposition Strategy - Similarity 2 N5

Compare to Cut: bw - Accesses Decomposition Strategy - Similarity 1 N5

Submit — Upload completed successfully.

Metrics

MojoFM

Accuracy	Precision	Recall	Specificity	F-score
0.98	0.99	0.94	1	0.96

Mojo Common Entities	Mojo Biggest Cluster	Mojo New Cluster	Mojo Singletons
97.73	97.73	97.73	97.73

Entity Pairs

True Positives	True Negatives	False Positives	False Negatives
344	808	2	22

False Pairs

Figure 4.12: MoJoFM in Comparison Table

False Pairs

Statistics

Property	bw - Accesses and Repository Decomposition Strategy - Similarity 2 N5	bw - Accesses Decomposition Strategy - Similarity 1 N5
Largest Cluster Size:	23	22
Smallest Cluster Size:	2	2
Number of Singleton Clusters:	0	0
Cohesion:	0.76	0.66
Coupling:	0.3	0.34
Silhouette Score:	0.41	0.41
Performance:	2181.53	2189.95
Team Size Reduction Ratio:	0.3666666666666667	*Not Present*
Complexity:	34358.62	36006.91

Decomposition Differences

Figure 4.13: Statistics in Comparison Table

4.5 Decomposition Recommendation

This functionality is part of one of the previously mentioned additional features (see Section 3.4). It helps the architect reason about multiple decompositions according to the chosen strategy. It works by attributing different weights to the available aggregation criteria (the number of criteria may vary) and then creates multiple decompositions based on the similarity level, given by said criteria. Once the decompositions are created, their metrics are calculated and saved in a list. The decompositions are not persistently saved, since the architect usually only wants a small selection of decompositions, according to his requirements. It would also take a bigger impact on performance, which needs to be minimized in this feature, since a large number of calculations is made and the architect is actively waiting for more results. In Figure 4.14 we can see the implementation of this feature.

Each line corresponds to a decomposition, while the columns with `Weight` correspond to the weights of the criteria (from `Access Weight` to `Sequence Weight`) and the columns from `Complexity` to `Performance` correspond to the metrics obtained from said decomposition. As we can see, there are multiple ways of filtering and ordering the columns. In this case, the decompositions with five clusters and the smallest possible complexity (ordered by complexity) were chosen. The first and third decompositions were selected by the architect to be created and persistently saved.

It should be noted that when the architect requests the recommendation of multiple decompositions, while they are being generated, the architect can already analyse some of the obtained results.

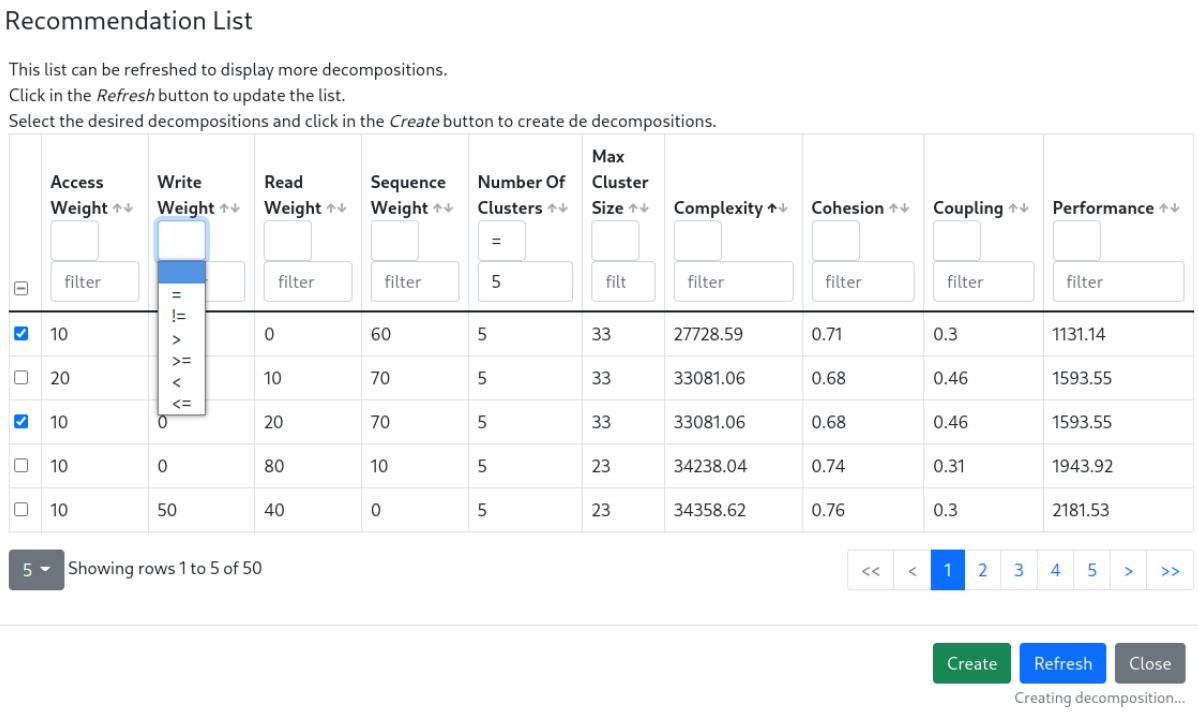


Figure 4.14: Recommendation

5

Evaluation

Contents

5.1 Previous Framework 38
5.2 Current Framework 38
5.3 Extensibility 39
5.4 Pluggability 42
5.5 Performance 43

The evaluation is focused on the three identified qualities: pluggability, extensibility, and performance. But before analysing each one of the qualities, we describe the previous object-oriented framework implementing the design, as well as the current framework.

5.1 Previous Framework

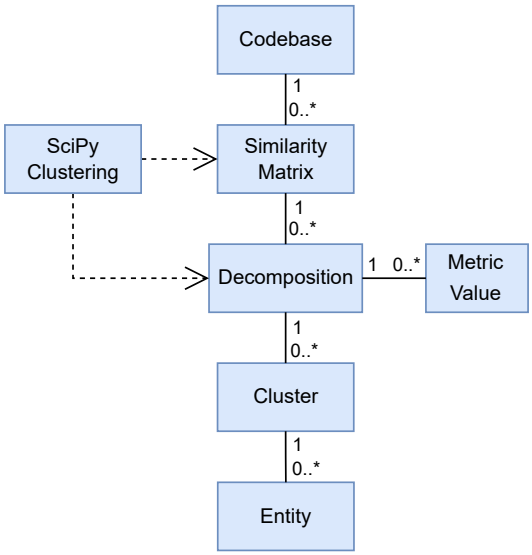


Figure 5.1: Previous Tool Framework

The previous framework (represented in Figure 5.1), presents the main classes that support the Mono2Micro's microservices identification approach. The *Codebase* class encapsulates the context of each monolith, while the *Similarity Matrix* class is responsible for creating the similarity between elements. From there, the SciPy Clustering uses the information obtained in the Similarity Matrix class to create a *Decomposition*, which in turn contains, *Metric Values*, *Clusters* and *Entities*. Since Mono2Micro was not designed to allow for extensibility and pluggability, no extension points were available and the addition of new features implied a strong coupling to the current structure.

5.2 Current Framework

Figure 5.2 presents the classes implementing the current design of the tool, emphasizing the main abstract classes that support the pipeline stages, their pluggability and extensibility. The *Representation* abstract class decouples the collection and decomposition generation stages, while allowing the extensibility of different monolith representations. In the previous framework, *Representations* was not yet present, as this information was attached to the Codebase class and did not allow for extension.

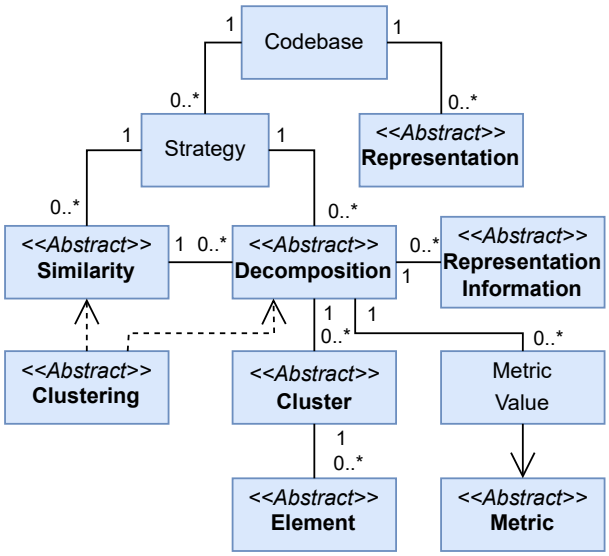


Figure 5.2: Tool Framework

Similarly, the *Decomposition* abstract class decouples the decomposition generation stage from other downstream stages, while allowing the extensibility of different types of decomposition through the abstract classes *Representation Information*, *Cluster* and *Element*. The abstract classes *Similarity* and *Clustering* support the decomposition generation stage, the abstract class *Representation Information* the visualization, editing and modelling stages, and the *Metric* abstract class the quality assessment stage. It is recommended that this figure is revisited when analysing Figures 5.3 to 5.5, which will focus on certain components of the tool's framework.

Comparing to the previous framework, *Strategy* was introduced to choose the microservice decomposition approach and *Similarity* as well as *Clustering* were refactored from *Similarity Matrix* and *SciPy Clustering*, respectively, to allow for their extension. As for *Decomposition*, it was also adapted to provide extensibility by introducing *Representation Information* and refactoring *Cluster* and *Entity* from their previous implementation, *Cluster* and *Element*, respectively. The contents of *Representation Information* were previously attached to Decomposition. Finally, the metric values were previously obtained from multiple metric's calculations that were highly coupled and with the refactorization, a modular approach was taken and each metric's calculations extracted into its own class.

5.3 Extensibility

To assess the tool's extensibility, we discuss how the tool's object-oriented framework is extended, through the extension points, to support the different approaches. As already stated in Section 3.1.1, collectors are not integrated in the tool, since they are specific for each type of collection and cannot

be generalized, but the *Representation* abstract class allows their integration into the pipeline. Overall, two concrete approaches have been implemented to answer **RQ** (see Section 1.2), resulting from the sequence of accesses [7] and the development history [43], but, whenever relevant, the support of other approaches is referred. While in the sequence of accesses, the codebase is collected to obtain the call graph's sequence of accesses, in the development history, the codebase is collected to obtain the relations between the entities and the authors that modified them, as well as the relations between the entities, based on the number of commits in common between each pair.

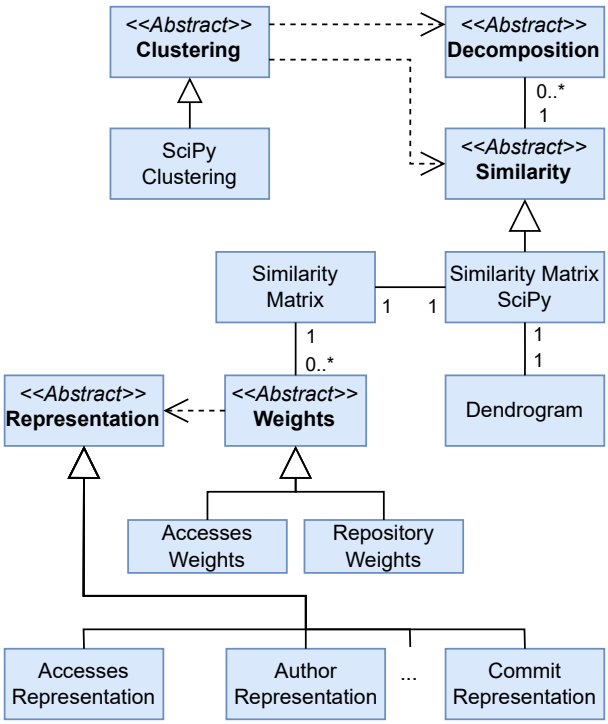


Figure 5.3: Decomposition Generation Extension

Figure 5.3 shows the extensions for the decomposition generation stage. The abstract classes *Representation* and *Decomposition* are, respectively, the input and output of the stage. *Representation* has several extensions, *Accesses Representation* [7], *Author Representation* and *Commit Representation* [43], which are supported by a JSON format, simplifying their integration with the decomposition generation stage. Inside this stage, two extension points, the criteria and algorithm, are supported by the abstract classes *Clustering*, *Similarity* and *Weights*.

The diagram illustrates that, actually, there is a single extension of *Clustering*, *SciPy Clustering*, which is responsible for the decomposition algorithm that generates the *Partition Decomposition*, an extension of *Decomposition* (which can be seen in Figure 5.4).

Since a single decomposition algorithm was implemented, there is a single subclass of *Similarity* (*Similarity Matrix SciPy*), which supports the algorithm's input format, a *Similarity Matrix*. The *Similarity*

Matrix decouples the criteria (which are represented by the *Weights*) from the algorithm. Therefore, if a new algorithm is added, it is only necessary to define a new *Similarity* subclass, keeping the *Weights* (criteria) subclasses. On the other hand, adding a new representation only requires defining a new *Weights* subclass, while keeping the algorithm and similarity. As for the *Dendrogram*, it is one of the specific properties provided by the SciPy clustering algorithm.

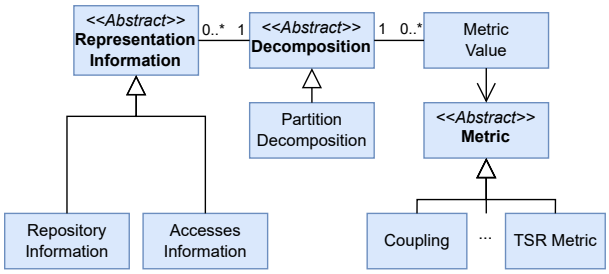


Figure 5.4: Decomposition Extension

Figure 5.4 shows part of the extensions for the decomposition's downstream stages. The *Decomposition* abstract class has a single subclass, which is related to the decomposition algorithm, in this case, it is a partition, where an element belongs to a single cluster. If other types of decompositions are to be considered, new subclasses should be defined.

The *Representation Information* subclass is extended with the set of representations used in the decomposition generation. This information allows the adaptation of the visualization, assessment and editing and modeling extension points. In what concerns visualization, the views depend on the type of information to present. Two extensions of views were done, one based on the sequence of accesses and the other based on the development history.

While in the view based on the sequence of accesses, the edges and displayed information depend on the functionalities that belong to the *Accesses Information*, in the view based on the development history, the edges and displayed information is related to the commits and authors in common, which belong to the *Repository Information*. The usage of the view based on the accesses can be seen in Section 4.3. Both views follow the *Observer* pattern.

Considering the assessment of the decomposition's qualities, new subclasses of *Metric* can be defined, depending on *Representation Information*. For instance, while *Coupling*, *Cohesion*, *Complexity* and *Performance* require the sequence of accesses (present in *Accesses Information*), *TSR Metric* (Team Size Reduction) [43] requires the development history (present in *Repository Information*). The support of the editing and modelling extension is shown in the next section.

5.4 Pluggability

The tool's pluggability focuses on how new stages can be added, which was exercised by the support of the aforementioned additional features (see Section 3.4) and shown in Figure 5.5.

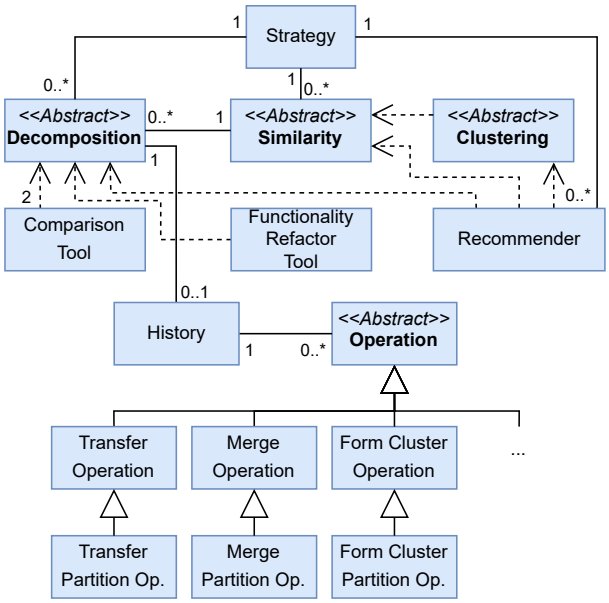


Figure 5.5: Implementation of Additional Features

The *Recommender*, *Comparison Tool* and *Functionality Refactorization Tool* features are implemented as external tools that use some of the tool's modules. Starting by the *Recommender*, it uses a brute force algorithm that generates a large number of decompositions to assess them. Its usage can be seen in Section 4.5. Only one extension of the *Recommender* was done (thus not mentioned in Figure 5.5), called *Recommend Matrix SciPy*, which uses the *Similarity Matrix* and *SciPy Clustering* classes to produce *Partition Decompositions*. Even with the addition of new *Representations* about the development history, no additional extensions of *Recommender* were needed, since *Recommend Matrix SciPy* is able of producing *Partition Decompositions*, independently of the representations available. As an additional note, although not evaluated, the *Recommender* is prepared for extension. Each extension should only depend on the *Similarity*, *Clustering* and *Decomposition* used.

As for the *Functionality Refactor Tool*, it uses a *Decomposition* and its *Accesses Information* to recommend functionality refactorizations [37] that reduce the decomposition's complexity. This feature is implemented in an external Docker container and only depends on the *Decomposition*'s structure.

Although not considered an additional feature, the *Comparison Tool* (see its usage in Section 4.4) was obtained by modifying an existing functionality that calculated the MoJoFM [42] values between two decompositions. It is prepared for extension, although not tested, but by default, it compares two decompositions even when they are not of the same type, since all decompositions contain *Clusters* with

Elements, which are the requirements for MoJoFM. Other comparisons varying on the decompositions used can be added, making this tool dependent on *Decomposition*.

Finally, the remaining additional feature, the *History Logger* feature (see Sections 3.4 and 4.3), represented by *History* in Figure 5.5. Its addition in the tool was fairly easy, only requiring the creation of an *History* instance in the *Decomposition* setup.

As for the *Operations*, they are responsible for the editing and modelling stage and even though operations were already available (*Transfer*, *Merge*, *Split* and *Rename*), they were not implemented in an object-oriented approach, only the operation's metadata was sent to the backend (the cluster's names, the affected entities, *etc.*) which did not allow for its extension and so the *Operation* class and respective subtypes were introduced, along with *History*, by implementing the *Command* pattern with its execute (also redo) and undo actions. An extension of *Operation* was also added, the *Form Cluster Operation*. Since the view now contained entities (see Chapter 4), it was sometimes useful to form a cluster by selecting the desired entities and clusters.

Also, *Operations* are dependent on *Decomposition*. This is because the behavior of the operation can vary according to the decomposition's components. For example, the merge operation behavior depends whether or not an element can be in several clusters. They also depend on the *Representation Informations* of the *Decomposition*, since each subclass of an operation might do some additional processing, depending on the present *Representation Informations*.

Therefore, when a new *Representation Information* is added, it is sometimes necessary to modify each of the operation's subclasses to handle it, since it might contain information that can be indirectly affected (such as the functionalities with *Accesses Information*), whereas, if a new extension of *Decomposition* is added, the definition of new subclasses is implied for each of the operations.

Since each *Operation* is responsible for its undo/redo, *History* is not affected. As a matter of fact, *History* does not have any kind of knowledge about the specific types of *Operation* stored in its log. Although not represented in Figure 5.5, *History* has one extension, *PositionHistory*, which also saves the positions of the components in the graph (related to *Save Graph Positions*, see Section 4.3). The positions are saved in a log that corresponds to the operations' log.

5.5 Performance

The development of this project implied the refactorization and expansion of the previous Mono2Micro's design. With the refactorization, there were also trade-offs between performance and better code structure. To mitigate some of the performance losses, optimizations were introduced.

Starting with the decomposition generation, the performance can be observed in Table 5.1. This test was made by comparing the creation of a decomposition with 5 clusters and using a strategy based

Table 5.1: Single Decomposition Generation

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic
Functionalities	133	104	225	278	863
Entities	29	49	92	123	487
Previous Version	46	1435	62	151	9111
Current Version	139	2033	232	362	12649

on the sequence of accesses, since the previous version of Mono2Micro only supported sequences of accesses. The measures were done by averaging the time (in milliseconds) of 5 executions once the time stabilized (avoiding initial longer executions), and with each codebase empty. As we can see, when comparing both versions, the previous version can be considerably faster. This is mainly due to the refactorization of the metrics. The previous metrics calculations were united, which allowed to share some of the performance-heavy calculations, but at the cost of sacrificing extensibility, as it did not allow for the extension of the metrics.

Table 5.2: Multiple Decomposition Generations in Same Codebase

Number of Decomposition Generations in Codebase (N)	1	5	10	15	20
Time Taken After N Generations	9111	9604	10992	12834	14973

However, the same cannot be said about the decomposition generation time when generating multiple decompositions in the same codebase, as it can be seen in Table 5.2. Since the current version uses a database instead of the file system previously used for storage, the performance is maintained. The database model is also much more partitioned, usually having a collection per each tool's component. This tests were done by decomposing the Fenix Academic codebase by the number of times indicated in the table and with the time taken in milliseconds. Although the performance hit is quite significant here, it is nowhere near as critical in the majority of codebases, since they present smaller codebase sizes, and thus requiring more decompositions to have a significant impact.

Now taking a closer look at the visualization stage, once refactored and extended, became concerning performance-wise. Because of it, optimizations were introduced to improve the architect's experience when interacting with the tool. The first example we have of this can be seen in Table 5.3. It shows the loading time of a candidate decomposition view in milliseconds (average time of 5 executions). As discussed, the refactorization introduced the *Element* class, for the cluster's elements, to improve the extensibility. Along with this, the elements also became nodes in the view, reducing the performance. When first introduced in the view, a built-in clustering feature of the graph tool was used to generate

Table 5.3: Clusters View Booting Time

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic
Functionalities	133	104	225	278	863
Entities	29	49	92	123	487
Previous Version	29	1920	42	67	1370
Refactored Version	329	2473	641	844	13621
Optimized Version	192	179	182	283	4503

the clusters, which easily grouped the elements of the graph into their respective cluster. However this feature was very slow during graph construction and updates. So to improve it, two actions were taken.

The first was to remove the built-in clustering feature and replace it with an implementation specific to these needs. Booting (and updating) times got significantly better, as shown in Table 5.3 when comparing the *Refactored Version* and the *Optimized Version*. The second action was the introduction of an optimization, which calculated the edges' properties in the backend, instead of in the frontend. In the majority of codebases, there were slight performance improvements, since most of the time is taken producing the graph. However, in *Blended Workflow* we can see that the time in the *Optimized Version* is significantly quicker than in the *Previous Version*. Even though it only has 104 functionalities, they are the longest (with more accesses) of all codebases, which significantly worsened the performance when processing them in the frontend. Nevertheless, *Fenix Academic*, which contains much more entities and functionalities, continues to perform worse than the *Previous Version*.

Table 5.4: Clusters View Booting Time With Saved Positions

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic
Functionalities	133	104	225	278	863
Entities	29	49	92	123	487
Optimized Version	192	179	182	283	4503
Saved Positions Version	195	185	175	233	1353

This was the best trade-off obtained, between the quantity of information provided and the performance hit. Although with the addition of the *History Log* feature, further improvements were made, since now the positions of the graph could be saved. In Table 5.4 we can see the performance when booting from a decomposition with saved graph positions. The average of 5 executions was again made, counting the time in milliseconds until the graph was fully booted. While in the first three codebases, the times were really close, sometimes better, sometimes worse, the difference was negligible or non existent.

With *Splunk AWS* the times were consistently better, but still not significant. Once in *Fenix Academic*, the difference was significant and improved to the point of equaling the *Previous version* from Table 5.3. Of course, for this to happen, it requires that the graph is fully booted once, to save said positions.

Now focusing on the operations, in Table 5.5, we can see the performance of the same merge operation across the three previously mentioned versions, done to the same decomposition and to the same clusters. This times were also an average of 5 executions.

Table 5.5: Merge Operation and Clusters Redraw Time

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic
Functionalities	133	104	225	278	863
Entities	29	49	92	123	487
Previous Version	55	3740	74	171	4264
Refactored Version	180	4805	308	900	14795
Optimized Version	100	88	148	168	1688

Multiple conclusions can be taken from this table. Starting by analysing between the *Previous Version* and the *Refactored Version*, one can notice the increase in processing time in *Blended Workflow* and *Fenix Academic*. This is caused by the recalculation of metrics, which mainly affects these two codebases. In decompositions with accesses, the functionalities change according to the clusters, since their sequence of local transactions is affected correspondingly. So once an operation is requested in these versions, all functionalities were invalidated and reconstructed to then recalculate the metrics.

Since these two codebases contain the longest (*Blended Workflow*) or many (*Fenix Academic*) functionalities, there is a quite noticeable spike in processing time. In *Refactored Version*, the performance is even further aggravated since, as it was previously noticed, the metrics calculation became slower with the refactorization and after each operation, the view had to be redrawn accordingly, which was also already concluded that it takes more time.

To solve this, several optimizations were introduced. In the *Current Version*, instead of eagerly recalculating the metrics, a lazy approach was taken. A new flag was introduced in the decompositions to invalidate them. So, once an operation is requested, instead of reconstructing the functionalities and recalculating the metrics, solely the affected functionalities (which correspond to the ones that interact with the affected nodes) are removed and the decomposition invalidated. Once the architect requests any information related to the functionalities or metrics, they are updated and the flag removed.

This makes it so that the response is quickly returned, but another optimization was made in the frontend. During the time waiting for the response, the frontend prepares the redraw of the view according to the requested operation, instead of waiting for the clusters and entities to then redraw the view. In this

case, three outcomes are possible:

1. The operation can fail in the backend and once the frontend finishes preparing the redraw, the frontend aborts the update of the view;
2. The backend operation can finish faster than the frontend preparing the redraw, in which case, once the frontend finishes preparing the redraw, the view is updated;
3. The frontend finishes first and waits for the response of the backend to then update the view.

The second outcome can be quite noticeable in the Fenix Academic codebase, since it contains the largest amount of entities, which difficults edge processing. The remaining operations follow the same logic as the merge operation since their procedure is the same, while the *Optimized Version* avoids eager calculations, the other versions prioritize them.

Table 5.6: Number of Clusters Comparison

	Number of Clusters	4	6	8	10	12
ACME Champions	Previous Version (ms)	31	54	54	59	65
	Current Version (ms)	124	142	142	149	158
Blended Workflow	Previous Version (ms)	1273	1411	1599	1750	1927
	Current Version (ms)	2049	2381	2207	2398	2403
hexie	Previous Version (ms)	56	62	64	69	73
	Current Version (ms)	207	227	232	231	225
Splunk AWS	Previous Version (ms)	110	133	174	237	199
	Current Version (ms)	357	386	429	402	421
Fenix Academic	Previous Version (ms)	8916	9593	12442	13941	13135
	Current Version (ms)	12736	13437	16647	16993	17694

Now let's take a look at the time taken when creating decompositions with different numbers of clusters, which can be analysed in Table 5.6. Once again, each time was taken from an average of 5 executions. As a general rule, with the increase of the number of clusters, there is an increase in processing time. These times heavily depend on the functionalities and how they process their local transactions graph. With the increase in the number of clusters, more local transactions will be made.

Now let's take a look at some of the performance values of the new view, focused on the development history, the repository view, represented in Table 5.7. The tests were also made with decompositions

Table 5.7: Repository View Booting Time

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic
Entities	29	49	92	123	487
Optimized Version	152	167	146	197	1904
Saved Positions Version	168	178	162	185	846

of 5 clusters and each time (milliseconds) is an average of 5 executions. In general, we can see that the performance values obtained are significantly better than when using accesses. This is due to the amount of information that needs to be processed in each of the views, as in the repository view, it is much inferior. It can also be concluded that the *Save Graph Positions* feature does not improve or impact the time for smaller codebases, while in *Fenix Academic*, it considerably reduces the processing time. Overall, during the usage of the repository view, the processing times are quite small, even with larger codebases, which can be considered irrelevant.

But even with all these optimizations, a troubling notes performance-wise needs to be addressed, related with the views. When visualizing graphs with the majority of nodes being elements, it is to be expected that the number of edges also increases compared to a view with a majority of nodes being clusters. With a view such as the accesses view, which usually has nearly an edge per each pair of entities, the performance significantly worsens. We can see this in Table 5.8. This table shows the average time to expand all clusters into its respective nodes, along with the number of edges, once all clusters are expanded. It is a very demanding operation since, once expanded, all nodes and edges need to be recalculated. This table presents the average of 5 executions of this operation, except for the *Fenix Academic Cold Start*, which is only executed once, with a newly created decomposition. While in the majority of codebases it is acceptable, once considering *Fenix Academic*, the time taken drastically changes. Unfortunately, no optimization cannot significantly improve the performance, other than removing edges. Also, when the architect is working with the tool, he usually does not want such a broader perspective with, 41725 edges and 487 nodes. Usually he will either only have part of the clusters expanded or use the *Only Show Neighbours* operation, which mitigates this problem.

Table 5.8: Expand All in Accesses View

Codebases	ACME Champions	Blended Workflow	hexie	Splunk AWS	Fenix Academic Repository	Fenix Academic Accesses	Fenix Academic Cold Start
Nodes	29	49	92	123	487	487	487
Edges	355	1143	915	2353	13948	41725	41725
Optimized Version	32	68	55	117	2310	6920	37563

6

Conclusion

Contents

6.1 Conclusion 50

6.2 System Limitations 50

6.3 Future Work 51

6.1 Conclusion

Although there is large number of strategies for microservices identification, there is also a lack of tools that facilitate and promote the experimentation and comparison. Because of this, such a tool would help software architects on the identification of what could be the best candidate decomposition or the trade-offs between different decompositions.

After analysing the existing approaches and their tools, we propose a microservices identification pipeline and identify the relevant variations that an extensible multiple strategy tool should have. Additionally, an object-oriented framework is proposed, which decouples the identification pipeline stages and supports several extension points, providing the qualities of pluggability and extensibility.

The solution allows for different types of monolith collectors, different criteria for the definition of similarity measures between the monolith elements, several algorithms for the decomposition of the monolith into microservices, different visualizations of the candidate decompositions, their assessment through metrics, and comparison.

The tool is evaluated through the integration of two identification strategies, one based on the monolith's sequences of accesses and another on the codebase repository history. These integrations led to the extension of the representations, criteria, representations' information, metrics, operations and views, as well as the introduction of new features to prove its pluggability.

Amongst the most important we have a *Recommendation* tool, helping the architect choosing the candidate decomposition, based on metrics, and a *History Log* tool, capable of undoing and redoing modifications, which when paired with the new snapshot operation, further incentivize the modification of the decomposition. For a more careful review of the implementation, consulting Chapter 7 is advised.

6.2 System Limitations

As mentioned at the end of Section 5.5, there are still some operations with a long waiting time, mainly due to the amount of information that needs to be processed to generate the graph. Although some workarounds, such as removing some of the edges or further code optimization could improve performance, the problem still remains when generating large dense graphs with the current graph generation tool. As it stands, Mono2Micro is also targeted to be used alongside JavaScript-based modules, which does not help when trying to use other external graph visualization tools, although workarounds, such as exporting files, would be possible.

As for the suggested pipeline, there is also the possibility that some of the approaches do not perfectly fit with the defined stages. This is the case when considering the Process Mining Decomposition Framework [30] tool. Although it is referred in this work that an algorithm can be used to provide the decomposition, the implementation described involves the manual building of the decomposition by the

grouping and splitting of elements, which would require a visualization stage as an alternative for the decomposition generation stage. The best solution would be to not implement the clustering algorithm and leave it to the editing and modelling stage, where the architect to manually make the decomposition. As it is expected, this is not the desired behaviour to have with the tool, but still possible. Just like in this example, there will always be some exceptions.

6.3 Future Work

Because of the nature of this project, which involves a lot of experimentation with the structure, as well as different features, some of the extension points were not tested to its full potential. This is the case with the similarities and the clustering algorithms. Considering that no additional clustering algorithm was added and it requires specific format of the similarity, neither were introduced. However, their addition should be straightforward, especially if considering that the outputted decomposition is of the same type as the *Partition Decomposition*. Nevertheless, it requires further experimentation.

The same goes for the extension of *Decomposition*. An interesting research would be the introduction of a *DuplicationDecomposition*, which allowed the duplication of elements across clusters, with newly introduced metrics. This also involves the implementation of new operation behaviors and possibly a new visualization.

Using other visualization tools would also be an interesting approach, having into account what was discussed in Section 6.2. Along with a new visualization tool, other views could be considered, for example, an improved version of the functionality view currently, present in Mono2Micro.

New comparison tools like MoJoFM would also be a relevant addition to the tool, especially those that do not take the decomposition type into account.

The introduction of an advanced *History* feature, which could be able of visualizing all the modifications done to a decomposition as a tree, branching when doing an operation after a certain amount of undo operations.

Bibliography

[1] D. Haywood, “In defense of the monolith,” *Microservices vs. Monoliths - The Reality Beyond the Hype*, 2017. [Online]. Available: https://www.infoq.com/minibooks/emag-microservices-monoliths/

[2] J. Thönes, “Microservices,” *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[4] ——, “Developing transactional microservices using aggregates, event sourcing and cqrs,” *Microservices vs. Monoliths - The Reality Beyond the Hype*, 2017. [Online]. Available: https://www.infoq.com/minibooks/emag-microservices-monoliths/

[5] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, “A taxonomy of service identification approaches for legacy software systems modernization,” *Journal of Systems and Software*, vol. 173, p. 110868, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302582

[6] A. Selmadji, A.-D. Seriai, H. L. Bouziane, C. Dony, and R. O. Mahamane, “Re-architecting oo software into microservices,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2018, pp. 65–73.

[7] L. Nunes, N. Santos, and A. Rito Silva, “From a monolith to a microservices architecture: An approach based on transactional contexts,” in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 37–52.

[8] A. Fuhr, T. Horn, and V. Riediger, “Using dynamic analysis and clustering for implementing services by reusing legacy code,” in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 275–279.

[9] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, “Service candidate identification from monolithic systems based on execution traces,” *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[10] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, “Service cutter: A systematic approach to service decomposition,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2016, pp. 185–200.

[11] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, “Identifying microservices using functional decomposition,” in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2018, pp. 50–65.

[12] Z. Zhang, R. Liu, and H. Yang, “Service identification and packaging in service oriented reengineering.” in *SEKE*, vol. 5. Citeseer, 2005, pp. 620–625.

[13] D. Saha, “Service mining from legacy database applications,” in *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 448–455.

[14] G. Mazlami, “Algorithmic extraction of microservices from monolithic code bases,” Master’s thesis, University of Zurich, February 2017.

[15] G. Mazlami, J. Cito, and P. Leitner, “Extraction of microservices from monolithic software architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.

[16] T. C. Matias, “Streamlined refactoring of modern web frameworks to microservices,” Master’s thesis, Faculdade de Engenharia Universidade do Porto, July 2019.

[17] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, “Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1214–1224. [Online]. Available: https://doi.org/10.1145/3468264.3473915

[18] H. Jain, H. Zhao, and N. R. Chinta, “A spanning tree based approach to identifying web services,” *International Journal of Web Services Research (IJWSR)*, vol. 1, no. 1, pp. 1–20, 2004.

[19] L. Baresi, M. Garriga, and A. D. Renzis, “Microservices identification through interface analysis,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 19–33.

[20] C. Del Grosso, M. Di Penta, and I. G.-R. de Guzman, “An approach for mining services in database oriented applications,” in *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. IEEE, 2007, pp. 287–296.

[21] B. Andrade, S. Santos, and A. R. Silva, “From monolith to microservices: Static and dynamic analysis comparison,” 2022. [Online]. Available: https://arxiv.org/abs/2204.11844

[22] M. Daoud, A. E. Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. E. Fazziki, "Automatic microservices identification from a set of business processes," in *Smart Applications and Data Analysis*, M. Hamlich, L. Bellatreche, A. Mondal, and C. Ordonez, Eds. Cham: Springer International Publishing, 2020, pp. 299–315.

[23] A. Vemulapalli and N. Subramanian, "Transforming functional requirements from uml into bpel to efficiently develop soa-based systems," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2009, pp. 337–349.

[24] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *Journal of Systems and Software*, vol. 157, p. 110380, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301475

[25] M. Aggarwal and S. Sabharwal, "Test case generation from uml state machine diagram: A survey," in *2012 Third International Conference on Computer and Communication Technology*. IEEE, 2012, pp. 133–140.

[26] S. Bechhofer, M. T. Özsu, and L. Liu, "Owl: Web ontology language," in {*Encyclopedia of Database Systems*}. Springer Nature, 2009.

[27] M. Nakamur, H. Igaki, T. Kimura, and K. Matsumoto, "Identifying services in procedural programs for migrating legacy system to service oriented architecture," in *Implementation and Integration of Information Systems in the Service Sector*. IGI Global, 2013, pp. 237–255.

[28] M. J. Amiri, S. Parsa, and A. M. Lajevardi, "Multifaceted service identification: process, requirement and data," *Computer Science and Information Systems*, vol. 13, no. 2, pp. 335–358, 2016.

[29] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[30] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," 05 2019.

[31] A. A. C. D. Alwis, A. Barros, C. Fidge, and A. Polyvyanyy, "Discovering microservices in enterprise systems using a business object containment heuristic," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2018, pp. 60–79.

[32] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *2007 Australian Software Engineering Conference (ASWEC'07)*. IEEE, 2007, pp. 329–340.

[33] M. Abdelkader, M. Malki, and S. M. Benslimane, “A heuristic approach to locate candidate web service in legacy software,” *International journal of computer applications in technology*, vol. 47, no. 2-3, pp. 152–161, 2013.

[34] J. Bogner, S. Wagner, and A. Zimmermann, “Automatically measuring the maintainability of service- and microservice-based systems: a literature review,” in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017, pp. 107–115.

[35] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, and D. Banerjee, “Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1606–1610.

[36] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii, “Visualization tool for designing microservices with the monolith-first approach,” in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 32–42.

[37] J. Correia and A. Rito Silva, “Identification of monolith functionality refactorings for microservices migration.” *Softw Pract Exper*, pp. 1–20, 2022.

[38] S. Santos and A. R. Silva, “Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures,” *Journal of Web Engineering*, Aug. 2022. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.2158

[39] J. Marshall and G. Kotonya, “A runtime visualizer for microservices,” in *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2021, pp. 72–80.

[40] N. Santos and A. Rito Silva, “A complexity metric for microservices architecture migration,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[41] L. Theivendra, “Transform monolithic java applications into microservices with the power of ai,” Dec 2020. [Online]. Available: https://developer.ibm.com/tutorials/transform-monolithic-java-applications-into-microservices-with-the-power-of-ai/

[42] Z. Wen and V. Tzerpos, “An effectiveness measure for software clustering algorithms,” in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.

[43] J. Lourenço, “Monolith development history for microservices identification: a comparative analysis,” Master’s thesis, Instituto Superior Técnico, University of Lisbon, 2022.

7

Appendix

In this appendix we focus with further detail on the implementation of some of the components in the domain structure and how they are managed across the Docker containers. Mono2Micro contains five Docker[1] containers. One of the containers is composed of MongoDB[2], a document-oriented NoSQL database, responsible for the storage and management of the monoliths' data. To analyse the databases' contents, a container with Mongo Express[3] is used, although other tools could be used for this purpose. The backend container is written in Java and it uses Spring-Boot[4], which also has integration with MongoDB. Since many clustering algorithms are implemented in Python, a container with FastAPI[5] is used to easily create services and execute said algorithms. The functionality refactorization tool [37] is also present in another container, written in Go and implemented to be used with decompositions containing information about the monolith's functionalities. Finally, the frontend is implemented with ReactJS[6] and Bootstrap[7]. To generate the graphs, vis.js[8] is used, but other tools are possible.

7.1 Strategy Selection

Before generating a decomposition, one first needs to import the codebase's representation, as in Figure 3.7, and choose what type of decomposition the architect wants. Because of this, the first step is to create a codebase. As mentioned, the codebase denotes the monolith and stores the representations.

```
@Document("codebase")
public class Codebase {
    @Id
    private String name;
    @DBRef(lazy = true)
    private List<Representation> representations;
    @DBRef(lazy = true)
    private List<Strategy> strategies;
    private List<String> representationInfoTypes;
    ...
}
```

Listing 7.1: Codebase Class

[1] https://www.docker.com/
[2] https://www.mongodb.com/
[3] http://mongodb-tools.com/tool/mongo-express/
[4] https://spring.io/projects/spring-boot/
[5] https://fastapi.tiangolo.com/
[6] https://reactjs.org/
[7] https://getbootstrap.com/
[8] https://visjs.org/

In Listing 7.1 we can observe the Codebase class. The `@Document("codebase")` annotation is used to define Codebase as a domain object persisted by MongoDB, while the `@Id` annotation defines `name` as its unique identifier. Finally, the `@DBRef(lazy = true)` annotation dictates that, when fetching a Codebase, the `representations` and `strategies` should not be immediately fetched. As we can see, Codebase is composed of Representations and Strategies and when creating a new codebase, only the `name` is initialized. This is done in Listing 7.2, where the service responsible for the codebase creation invokes `codebaseRepository` to persistently save the codebase in the database. The annotation `@Autowired` is used to inject a dependency, in this case, `CodebaseRepository`, while the annotation `@Service` is used to define a service, which is automatically instantiated when depending on it.

```
@Service
public class CodebaseService {
    @Autowired
    CodebaseRepository codebaseRepository;

    public void createCodebase(String codebaseName) {
        if (codebaseRepository.existsByName(codebaseName))
            throw new KeyAlreadyExistsException();
        codebaseRepository.save(new Codebase(codebaseName));
    }
    ...
}
```

Listing 7.2: Codebase Creation Service

Once the codebase is initialized, follows the creation of the strategy. The strategy is defined by the desired clustering algorithm (`algorithmType`), the representation's information to include in the decomposition (`representationInfoTypes`), its `similarities`, its `decompositions` and `recommendations`, as shown in Listing 7.3. As of now, two `representationInfoTypes` exist, one based on the accesses and another based on the change history. They can both be added to the same decomposition.

```
@Document("strategy")
public class Strategy {
    @Id
    private String name;
    private String algorithmType;
    @DBRef
```

```
    private Codebase codebase;
    @DBRef(lazy = true)
    private List<Decomposition> decompositions = new ArrayList<>();
    @DBRef(lazy = true)
    private List<Similarity> similarities = new ArrayList<>();
    @DBRef(lazy = true)
    private List<Recommendation> recommendations = new ArrayList<>();
    private List<String> representationInfoTypes;
    ...
}
```

Listing 7.3: Strategy Class

To create the strategy, the architect needs to load the desired representation information (one or more) and their respective files, then once loaded, the clustering algorithm needs to be chosen. This is done in several forms that appear in the UI (see Section 4.1). Now in the backend, Listing 7.4 shows the strategy service responsible for the creation of a strategy. As we can see, to create the strategy, its necessary to verify that no identical strategy exists, since if it does, then there is no need to generate it. If the condition is not met, then the strategy is created by saving the `algorithmType` and `representationTypes`.

Although not present in this code, it should be noted that this operation can only be executed if the required representation information has already been loaded to the codebase. Another implementation detail should be pointed out. If another strategy already requires part of the representation files, then they will not be requested to be added. In fact, a strategy can be created entirely without adding any representation information or files, if they already exist.

```
@Service
public class StrategyService {
 ... // Dependency injection
 public void createStrategy(
     String codebaseName,
     String algorithmType,
     List<String> representationTypes) {
  Codebase codebase = codebaseRepository.findByName(codebaseName);
  for (Strategy strategy : codebase.getStrategies())
  if (strategy.getAlgorithmType().equals(algorithmType) &&
    strategy.getRepresentationInfoTypes().size() == representationTypes.size() &&
    strategy.getRepresentationInfoTypes().containsAll(representationTypes))
```

```
            return;

        Strategy strategy = new Strategy(codebase, algorithmType, representationTypes);
        ... // Persistently saves strategy
    }
    ...
}
```

Listing 7.4: Strategy Creation Service

To support the representations' extension, only new subclasses of representations need to be introduced. The codebase and strategy are not affected with this extension. To introduce a new representation, the abstract class `Representation` (Listing 7.5) needs to be extended and an entry added to its respective factory (Listing 7.6). Some of the representations might need some additional processing, which is done in the `init` method, when created. The `getType` method is implemented in each extension of `Representation` and corresponds to the type requested in the factory (Listing 7.6). This procedure with the `getType` and factory is frequently used with classes intended to be extended.

```
public abstract class Representation {
    @Id
    protected String name;

    @DBRef(lazy = true)
    protected Codebase codebase;

    public abstract String init(
        Codebase codebase,
        byte[] representationFile) throws Exception;

    public abstract String getType();
    ...
}
```

Listing 7.5: Abstract Representation

As it stands, four representation types are possible, as shown in Listing 7.6. Corresponding to the presented order, the `AccessesRepresentation` contains the information about the functionality accesses (or call graph), the `IDToEntityRepresentation` maps the IDs of the monolith's domain entities to their name, the `AuthorRepresentation` maps the domain entities to the authors that modified them and the

`CommitRepresentation` maps the number of commits in common between each pair of domain entities. Both the `AuthorRepresentation` and `CommitRepresentation` were added with the extension. In the frontend, a class (in TypeScript[9]), respective to each specific representation, needs to be created and added to a factory, like in the backend. The render of each representation is implemented in this class.

```
public class RepresentationFactory {
  public static Representation getRepresentation(String representationType) {
    switch (representationType) {
      case ACCESSES:
        return new AccessesRepresentation();
      case ID_TO_ENTITY:
        return new IDToEntityRepresentation();
      case AUTHOR:
        return new AuthorRepresentation();
      case COMMIT:
        return new CommitRepresentation();
      default:
        throw new RuntimeException(...);
    }
  }
}
```

Listing 7.6: Representation Factory

After this, the architect generates the decompositions, based on the chosen strategy. The architect has two alternatives, either requesting recommendations of decompositions or generating them manually. We will take a closer look at the manual approach in the following section.

7.2 Create Decompositions

In Section 7.1 it was shown how the selection of the strategy is made. With that serving as a foundation, we will now go through the creation of a decomposition, but before doing this, the architect first needs to generate the similarity (see Figure 3.7). This section will be divided into these two stages.

[9] https://www.typescriptlang.org/

7.2.1 Similarity Generation

The information that will be requested in the UI about the similarity will vary according to the chosen strategy, but once submitted, the Data Transfer Object (DTO) of the similarity request will be sent to the service responsible for the creation of the similarities.

```
@Service
public class SimilarityService {
 ...
 public void createSimilarity(SimilarityDto similarityDto) throws Exception {
  Strategy strategy = strategyRepository
    .findByName(similarityDto.getStrategyName());

  if (strategy.getSimilarities().stream()
    .anyMatch(similarity -> similarity.equalsDto(similarityDto)))
      return;

  Similarity similarity = SimilarityFactory
    .getSimilarity(strategy, similarityDto);

  similarity.generate();

  similarityRepository.save(similarity);
  strategyRepository.save(strategy);
 }
 ...
}
```

Listing 7.7: Similarity Creation Service

Listing 7.7 shows how the procedure is done. First, it is verified if the similarity already exists in the context of the current strategy. This is done by comparing the received DTO with the information present in the existing similarities of this strategy. This avoids the duplication of similarities, which is unnecessary. If this condition is not met, the factory will then create the specific type of similarity, required by the clustering algorithm. This factory is represented in Listing 7.8.

```
public class SimilarityFactory {
```

```
    public static Similarity getSimilarity(SimilarityDto similarityDto) {
        if (similarityDto == null)
            return null;
        switch (similarityDto.getType()) {
            case SIMILARITY_MATRIX_SCIPY:
                return new SimilarityMatrixSciPy(
                  (SimilarityMatrixSciPyDto) similarityDto);
            default:
                throw new RuntimeException(...);
        }
    }

    public static Similarity getSimilarity(
      Strategy strategy,
      SimilarityDto similarityDto
    ) {
     Similarity similarity = getSimilarity(similarityDto);
     ... // Setup name and associate strategy to similarity
     return similarity;
    }
}
```

Listing 7.8: Similarity Factory

Three strategies are available and all use the same similarity, which contains a `Similarity Matrix` as the format of the criteria. This similarity is intended to be used by the SciPy hierarchical clustering algorithm, thus also containing an additional parameter `linkageType`, which can be seen in Listing 7.9, representing this extension of `Similarity`.

```
@Document("similarity")
public class SimilarityMatrixSciPy extends Similarity {
  // Used during Similarity Generation
  private String profile;
  private int tracesMaxLimit;
  private Constants.TraceType traceType;
  // Used in Clustering Algorithm
  private String linkageType;
  private SimilarityMatrix similarityMatrix;
```

```
  // Dendrogram created in the Python services
  private Dendrogram dendrogram;
  ...
  @Override
  public void generate() throws Exception {
    ...
    this.similarityMatrix.generate(...);
    this.dendrogram = new Dendrogram(
      getName(), similarityMatrix.getName(), getLinkageType());
  }
  public Clustering getClustering() { return new SciPyClustering(); }
}
```

Listing 7.9: Similarity Extension

With the similarity created, follows the generation of the similarities between elements, in this case, the `Similarity Matrix`. This is done in method `generate` (called in Listing 7.7 and corresponds to the method in Listing 7.9) and fills the `Similarity Matrix` with the similarity levels between domain entities by calling the `generate` method of `Similarity Matrix`. With the `Similarity Matrix` created, it can now produce the `Dendrogram`, which solely contains an image of said dendrogram and requires an invocation to SciPy to generate it. This dendrogram can be seen in Figures 4.4(b) and 4.5(a).

In Listing 7.10 we can see the `generate` method of the `SimilarityMatrix` and the introduction of the `Weights` class. In this method, an empty ”raw” matrix is created and for each `Weights`, the method `fillMatrix` is called, which fills the ”raw” matrix with the correct similarity values according to several criterias. With the ”raw” matrix filled, the method `getSimilarityMatrixAsJSON` is called, which produces the final matrix that will be saved.

The term ”raw” matrix is used because when creating a matrix, the architect might want to combine multiple criteria (see Section 3.3). What this in fact implies is that there is one matrix per criteria. The ”raw” matrix is a matrix with a size of $matrix_{size} = elements^2_{length} \cdot criteria_{length}$, which, for all intents and purposes, equals the creation of one matrix per criteria.

```
public class SimilarityMatrix {
  public String name;
  private List<Weights> weightsList;
  ...
  public void generate(...,
      Similarity similarity,
```

```
        Set<Short> elements
    ) throws Exception {
      float[][][] rawMatrix = getEmptyRawMatrix(
        elements.size(), getTotalNumberOfWeights());
      ...
      int fillFromIndex = 0;
      for (Weights weights : getWeightsList()) {
        weights.fillMatrix(..., similarity, rawMatrix, elements, fillFromIndex);
        fillFromIndex += weights.getNumberOfWeights();
      }
      JSONObject matrixJSON = getSimilarityMatrixAsJSON(
          elements, rawMatrix, getWeightsAsArray());
      ... // Save similarity matrix JSON
    }
    ...
}
```

Listing 7.10: Similarity Matrix Generation

So, when executing `fillMatrix`, what in fact being done, is filling the corresponding matrix of each criteria. Keep in mind that each `Weight` class contains the weight of at least one criteria and is responsible for filling the matrices of the criteria that belong to it.

Let's now look at the `Weights` class. When the architect requests the generation of the similarities in the UI, a form will be requested, asking for the weights of the required criteria (see Figure 4.4(a)). This is valid for the three strategies available, but each one asks for different criteria.

So, once the form is filled and sent, their corresponding `Weights` object in the backend needs to be instantiated. This is done in the lines with the `@Json` annotation, shown in Listing 7.11. They define which subclass of `Weights` should be instantiated, based on the information sent from the frontend in JSON. This requires that each JSON object (representing a `Weights` object) contains a parameter `type`, to make sure the correct instance of Weights is created.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
@JsonSubTypes({
  @JsonSubTypes.Type(value = AccessesWeights.class, name = ACCESSES_WEIGHTS),
  @JsonSubTypes.Type(value = RepositoryWeights.class, name = REPOSITORY_WEIGHTS),
})
public abstract class Weights {
    public abstract String getType();
```

```
    public abstract int getNumberOfWeights();
    public abstract float[] getWeights();
    public abstract List<String> getWeightsNames();
    ...
}
```

Listing 7.11: Abstract Weights

The `AccessesWeights` contains four criteria, those being the similarity according to the reads, the writes, by the accesses (considered similar when accessed together) and by the sequence (considered similar when accessed in the same sequence), while the `RepositoryWeights` contains two criteria, those being the similarity according to the authors and the similarity according to the commits.

```
public class RepositoryWeights extends Weights {
 private float authorMetricWeight;
 private float commitMetricWeight;
 ...
 @Override
 public float[] getWeights() {
    return new float[]{authorMetricWeight, commitMetricWeight};
 }
 @Override
 public void fillMatrix(..., float[][][] rawMatrix,
   Set<Short> elements, int fillFromIndex) {
     // Fills raw matrix from "fillFromIndex" to "fillFromIndex" + 2
   }
 ...
}
```

Listing 7.12: Repository Weights

Part of the `Repository Weights` class can be seen in Listing 7.12. It contains two attributes, the weights of the criteria, given by `authorMetricWeight` and `commitMetricWeight`. As mentioned, when calling `fillMatrix`, the `Weights` class will fill the spaces in the matrix corresponding to its criteria. In this case, since it contains two criteria, for each pair of entities, two spaces of the matrix will be filled by `RepositoryWeigths`. The same logic is applied to `AccessesWeights` but with the four weights previously mentioned. During the execution of `fillMatrix`, the necessary representations are consulted to calculate the similarity level. Once the ”raw” matrix is filled, the final matrix is created by summing the

multiplication of each matrix by its weight. So, the final similarity level between two domain entities e_1 and e_2 is given by the following formula:

$$matrix_{final}[e_1][e_2] = \sum_{i=0}^{weights.length-1} matrix_{raw}[e_1][e_2][i] \cdot weights[i]$$

The `getWeightsAsArray` method in Listing 7.10 represents the *weights* array mentioned in the formula above and it is constructed by calling the method `getWeights` from each `Weights` class. It should be noted that the sum of all weights needs to be equal to 100%.

So, to recap, two extensions are possible when considering what was discussed. The first extension is by introducing another `Similarity`, which once added, will need another entry in the factory represented in Listing 7.8, and the second extension, by introducing another `Weights` class. This avoids defining another type of `SimilarityMatrix`, while also allowing the combination of criterias, even though they are isolated in each `Weights` class.

```
return ( <div style={{ paddingLeft: "2rem" }}>
    ...
    {strategy.algorithmType === "SciPy Clustering" &&
        <SimilarityMatrixSciPyForm
            codebaseName={codebaseName}
            strategy={strategy}
            setUpdateStrategies={setUpdateStrategies}
        />
    }
    {similarities.length !== 0 && renderSimilarities()}
    ...
  </div>);
```

Listing 7.13: Extension of Similarity Forms

In the frontend, some additions are required when applying extensions. If another `Similarity` is to be added, then its corresponding form in the frontend also needs to be added. In Listing 7.13 we can see the current form, `SimilarityMatrixSciPyForm`, being associated to its respective clustering algorithm. Inside it, the necessary weights are requested according to the chosen strategy. Also, for each new type of `Similarity` and `Weights`, a TypeScript object needs to be created and its entry added to the factory. These objects are used when sending information between the backend and frontend in DTOs.

7.2.2 Decomposition Generation

Keeping in mind what was discussed in Section 7.2.1 and serving as a foundation to this subsection, now we go through the generation of a decomposition. At this moment in time, the architect has already generated its similarity and will now make a request to create a decomposition.

```
@Service
public class DecompositionService {
 ...
 public void createDecomposition(DecompositionRequest request) {
  Similarity similarity = similarityRepository
        .findByName(request.getSimilarityName());
  Clustering clustering = similarity.getClustering();
  Decomposition decomposition = clustering
        .generateDecomposition(similarity, request);
  setupDecomposition(decomposition);
 }
 public void setupDecomposition(Decomposition decomposition) throws Exception {
  decomposition.setup();
  decomposition.calculateMetrics();
  ... // Persistently save decomposition, similarity and strategy
 }
 ...
}
```

Listing 7.14: Create Decomposition Service

The architect might be asked, for example, for the number of clusters to be produced (see Figure 4.5(a)), and once the request is sent, it is received in the service shown in Listing 7.14 and converted to its specific implementation, much like the `Weights` in Listing 7.11. To differentiate the type of decomposition request, a parameter `type` (expected to be related to the clustering algorithm in use) is required. The request of extension should not need to contain more information other than the required by the clustering algorithm. The only extension of `DecompositionRequest` is given by `SciPyRequestDto` in Listing 7.15, that contains information about the cut to be done by the SciPy clustering algorithm.

```
public class SciPyRequestDto extends DecompositionRequest {
    private String cutType;
    private float cutValue;
```

```
    ...
}
```

Listing 7.15: Decomposition Request

Focusing again on Listing 7.14, since the similarity is generated to provide the input to a clustering algorithm, then each subclass of `Similarity` must also know the algorithm to use (in the case of analysis, `SimilarityMatrixSciPy` for the `SciPyClustering`, see Listing 7.9).

So, once the clustering algorithm is obtaied in Listing 7.14, the clustering algorithm uses the requirement in the `request` to create the decomposition. The `generateDecomposition` method needs to be implemented by all classes responsible for clustering (requirement from the extension point of clustering algorithm). As it stands, only one approach is possible, which is the SciPy clustering algorithm. Just like SciPy, many clustering algorithms are implemented in Python, which is the reason for creating a service to facilitate the addition of other clustering algorithms. The backend invokes the service and sends the similarity matrix file name, as well as the cut information.

```
@scipyRouter.get("/scipy/{similarityMatrixName}/{linkageType}" +
                 "/{cutType}/{cutValue}/createDecomposition")
async def createDecomposition(similarityMatrixName,
                              linkageType, cutType, cutValue):
    return createDecompositionScipy(similarityMatrixName,
                                    linkageType, cutType, float(cutValue))
```

Listing 7.16: SciPy Clustering Service

So, once the cut is made, the clustering algorithm creates the subclass of `Decomposition` he is prepared to produce and fills it with clusters and elements. The `Decomposition` class is represented in Listing 7.17 and by taking a closer look, we can see that there a map of `clusters`. We can also see that each decomposition contains a map of `metrics`. This map connects the metric type to its metric value. `Object` is used since a metric might assume different types, such as `Integer`, `Double` or `String` and since it is only consulted on the frontend, it is more practical using it as `Object`. There is also a `representationInfos` list. This list is associated to the strategy chosen. Instead of duplicating large quantities of code in each extension of a decomposition, the specific information and methods associated to the representations are stored inside a `RepresentationInfo` object.

Other important attributes are `outdated` and `history` attributes, which are part of optimizations discussed in Section 5.5, when referring the lazy approach to updating, and the *History Log* feature discussed in Section 5.4, respectively.

```
public abstract class Decomposition {
 @Id
 String name;
 String type;
 boolean expert;
 boolean outdated;
 Map<String, Object> metrics = new HashMap<>();
 Map<String, Cluster> clusters = new HashMap<>();
 @DBRef(lazy = true)
 Strategy strategy;
 @DBRef
 Similarity similarity;
 @DBRef
 History history;
 List<RepresentationInfo> representationInfos = new ArrayList<>();
 public abstract void setup() throws Exception;
 public abstract void update() throws Exception;
 public abstract void calculateMetrics();
 public abstract void renameCluster(RenameOperation operation);
 ... // also contains more operations, other than rename
}
```

Listing 7.17: Abstract Decomposition

Let us go back to Listing 7.14. Once `generateDecomposition` is completed, `setupDecomposition` remains, which will invoke the `setup` and `calculateMetrics` methods. Each extension of `Decomposition` needs to implement this two methods (see Listing 7.17). In Listing 7.18 we can see its implementation in `PartitionDecomposition`.

```
@Document("decomposition")
public class PartitionsDecomposition extends Decomposition {
 ...
 @Override
 public void calculateMetrics() {
  this.representationInfos.stream()
    .map(RepresentationInfo::getDecompositionMetrics)
    .flatMap(Collection::stream).forEach(metric ->
    this.metrics.put(metric.getType(), metric.calculateMetric(this)));
```

```
  }

  @Override
  public void setup() throws Exception {
   List<RepresentationInfo> representationInfos = RepresentationInfoFactory
     .getRepresentationInfosFromType(getStrategy().getRepresentationInfoTypes());
   for (RepresentationInfo representationInfo : representationInfos)
     representationInfo.setup(this);
   this.history = new PositionHistory(this);
  }

  @Override
  public void update() throws Exception {
   for (RepresentationInfo representationInfo : representationInfos)
     representationInfo.update(this);
  }
  ...
}
```

Listing 7.18: Extension of Decomposition

Looking at the `setup` method, we can see that the list of `RepresentationInfo` is created according to the list of `representationInfoTypes` of the strategy. Once it is created, each `RepresentationInfo` also invokes `setup`. Two extensions of `RepresentationInfo` are available. One related to the accesses while the other is related to the information extracted from the repository, each with the name `AccessesInfo` and `RepositoryInfo` respectively. In the case of `AccessesInfo`, in its setup, the functionalities need to be created, while on `RepositoryInfo`, the information about the authors and commits needs to be extracted from the representation files.

With this done, the metrics can finally be calculated. Each `RepresentationInfo` has some metrics associated to it, so once the method `getDecompositionMetrics` is called, a list of `DecompositionMetric` (which extends from `Metric`) is returned. `DecompositionMetric` is an intermediate extension of `Metric` since we have other components with their own metrics, for example, the functionalities' redesigns. So, extending `DecompositionMetric` we have, for example `CohesionMetric` and `CouplingMetric`.

```
public class TSRMetric extends DecompositionMetric {
  @Override
  public String getType() {
    return TSR;
```

```
    }
    @Override
    public Double calculateMetric(Decomposition decomposition) {
      ...
    }
  }
```

Listing 7.19: Metric Extension Example

In Listing 7.19 we can see the extension `TSRMetric`, a metric that uses `RepositoryInformation` to calculate the metric value. The calculation of the metric value is done in the method `calculateMetric`.