

# **Domain-Driven Design Representation of Monolith Candidate Decompositions Based on Entity Accesses**

**Miguel Mota Fernandes Levezinho**

Thesis to obtain the Master of Science Degree in

**Computer Science and Engineering**

Supervisor: Prof. António Manuel Ferreira Rito da Silva

## **Examination Committee**

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Member of the Committee: Prof. André Ferreira Ferrão Couto e Vasconcelos

**May 2024**

This work was created using  $\text{\LaTeX}$  typesetting language  
in the Overleaf environment ([www.overleaf.com](http://www.overleaf.com)).

# **Declaration**

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.



# Acknowledgments

I would like to start by expressing my deepest thanks to my dissertation supervisor, Prof. António Rito Silva, who provided an astonishing amount of availability through out the development of this thesis, offering whenever necessary the knowledge and guidance to make this project possible. This gratitude also extends to Prof. Olaf Zimmermann and to Stefan Kapferer, for their time, feedback, and extensive knowledge on the area.

I would also like to thank my family and loved ones for their endless support and care in all parts of my life. Without their encouragement I would not be where I am today.

Last but not least, I would like to thank my friends, with whom I grew as a person over all these academic years.

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) [1].



# Abstract

Microservice architectures have gained popularity as one of the preferred architectural styles to develop large scale systems, replacing the monolith architectural approach. Similarly, strategic Domain-Driven Design (DDD) gained traction as the preferred architectural design approach for the development of microservices. However, DDD and its strategic patterns are open-ended by design, leading to a gap between the concepts of DDD and the design of microservices. This gap is especially evident in migration tools that identify microservices from monoliths, where candidate decompositions into microservices provide little in terms of DDD refactoring and visualization. This thesis proposes a solution to this problem by extending the operational pipeline of a multi-strategy microservices identification tool, called Mono2Micro, with a DDD modeling tool that provides a language, called Context Mapper DSL (CML), for formalizing the most relevant DDD concepts. The extension maps the content of the candidate decompositions, which include clusters, entities, and functionalities, to CML constructs that represent DDD concepts such as *Bounded Context*, *Aggregate*, *Entity*, and *Service*, among others. The results are validated with a case study by comparing the candidate decompositions resulting from a real-world monolith application with and without CML translation.

## Keywords

Monolith; Microservices; Architectural migration; Domain-Driven Design; Domain-Specific Language; Integration;





# Resumo

Arquiteturas de microserviços ganharam popularidade como uma das abordagens favoritas no desenvolvimento de sistemas de larga escala, substituindo o estilo monolítico. Da mesma forma, Domain-Driven Design (DDD) estratégico ganhou tração como a ferramenta de desenho arquitetural mais versátil para o desenvolvimento de microserviços. No entanto, DDD e os seus padrões estratégicos não estão completamente formalizados por design, o que leva a uma separação entre os conceitos de DDD, e o desenho de microserviços. Esta separação é especialmente evidente no contexto de ferramentas de migração que identificam microserviços em monólitos, onde decomposições propostas para microserviços incluem pouco em termos de refatorização e visualização do ponto de vista de DDD. Esta dissertação propõe uma solução para este problema, composta pela extensão da pipeline operacional de uma ferramenta que contém múltiplas estratégias para a identificação de microserviços, chamada Mono2Micro, com uma ferramenta de modelação DDD que fornece uma linguagem, chamada Context Mapper DSL (CML), que formaliza os padrões DDD mais relevantes. Esta extensão mapeia o conteúdo de decomposições propostas, que inclui grupos, entidades e funcionalidades, para estruturas CML que representam conceitos DDD, como *Bounded Context*, *Aggregate*, *Entity*, e *Service*, entre outras. Os resultados são validados com um caso de estudo, comparando decomposições propostas de uma aplicação monolítica real com e sem a tradução para CML.

## Palavras Chave

Monolito; Microserviços; Migração Arquitetural; Domain-Driven Design; Linguagem de Domínio Específico; Integração;



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	3
1.2	Problem . . . . .	4
1.3	Approach . . . . .	4
1.4	Contributions . . . . .	5
1.5	Document Structure . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Domain-Driven Design . . . . .	9
2.1.1	Domain and Model . . . . .	9
2.1.2	Bounded Context and Context Map . . . . .	9
2.1.3	Bounded Context Relationships . . . . .	10
2.1.4	Entity, Value Object and Service . . . . .	10
2.1.5	Aggregate . . . . .	10
2.1.6	Domain Event . . . . .	11
2.2	Mono2Micro . . . . .	11
2.2.1	Collection Stage . . . . .	12
2.2.2	Decomposition Stage . . . . .	13
2.2.3	Quality Assessment Stage . . . . .	13
2.2.4	Visualization Stage . . . . .	14
2.2.5	Editing and Modeling Stage . . . . .	14
2.3	Context Mapper . . . . .	15
2.3.1	Discovery Library . . . . .	17
2.3.2	Architectural Refactorings . . . . .	17
2.3.3	Generators . . . . .	19
2.4	Tool Analysis . . . . .	19
<b>3</b>	<b>Related Work</b>	<b>21</b>
3.1	DDD Modeling Tools . . . . .	23

3.2	Microservice Identification Tools . . . . .	24
<b>4</b>	<b>Solution Architecture</b>	<b>25</b>
4.1	Architectural Requirements . . . . .	27
4.2	Pipeline Extension . . . . .	29
4.2.1	Tool Integration . . . . .	29
4.2.2	DDD Mapping . . . . .	30
4.2.3	CML Representation and Interaction . . . . .	31
<b>5</b>	<b>Implementation</b>	<b>35</b>
5.1	Structure Collector . . . . .	37
5.1.1	Design . . . . .	37
5.1.2	Usage . . . . .	39
5.2	Mono2Micro Contract . . . . .	41
5.3	Translation to CML . . . . .	43
5.3.1	Entity Mapping . . . . .	43
5.3.2	Cluster Mapping . . . . .	43
5.3.3	Functionality Mapping . . . . .	45
5.4	Discovery Strategies . . . . .	47
<b>6</b>	<b>Evaluation</b>	<b>51</b>
6.1	Architectural Evaluation . . . . .	53
6.1.1	Modularity . . . . .	53
6.1.2	Extensibility . . . . .	54
6.1.3	Interoperability . . . . .	55
6.2	Case Study . . . . .	58
6.2.1	Decomposition Generation . . . . .	58
6.3	Discussion . . . . .	60
6.3.1	Results Validation . . . . .	61
<b>7</b>	<b>Conclusion</b>	<b>65</b>
7.1	Threats to Validity . . . . .	67
7.2	Future Work . . . . .	67
7.3	Conclusions . . . . .	69
	<b>Bibliography</b>	<b>71</b>

# List of Figures

2.1	The five stages of the Mono2Micro operational pipeline [2]. Each stage can use the output of former stages as input. . . . .	11
2.2	Example syntax of CML, containing the syntax for defining a <i>Context Map</i> (1-6); <i>Bounded Contexts</i> (8-29, 31-32); <i>Aggregates</i> (9-21); <i>Entities</i> (10-15,17-20); and <i>Services</i> (24-27). . . . .	16
4.1	Mono2Micro pipeline extension to support CML representation of candidate decompositions. In blue, the relevant pipeline steps (top) and modules. In orange, the extension to the pipeline, composed of the addition of new modules. . . . .	29
4.2	Generated Context Map diagram of the integration between Mono2Micro and Context Mapper. PL stands for <i>Published Language</i> , ACL stands for <i>Anti-Corruption Layer</i> , and CF stands for the <i>Conformist</i> pattern. U and D represent upstream and downstream modules respectively. . . . .	30
5.1	Steps of the collector when retrieving structural information from source code. . . . .	37
5.2	Class diagram of the structure collector. The Spoon API is represented in orange, and the implemented packages and classes are represented in yellow. . . . .	38
5.3	GUI for the structure collector. . . . .	39
5.4	Mono2Micro interface additions to support contract generation. . . . .	41
5.5	Mapping strategy of candidate decomposition concepts from Mono2Micro (M2M) to Domain-Driven Design (DDD) and Context Mapper DSL (CML). . . . .	43
5.6	Generated CML example, representing an <i>Aggregate</i> that contains 2 <i>Entities</i> . Since <i>Topic</i> referenced an <i>Entity</i> in its fields not present in the <i>Aggregate</i> , <i>Question_Reference</i> was generated locally to replace this reference. . . . .	45
5.7	<i>Coordination</i> construct in CML. The steps of the <i>Coordination</i> (4-6) represent ordered calls to <i>Service</i> operations (10,20,11). . . . .	46
5.8	Opening a <i>Coordination</i> in Sketch Miner. . . . .	48
5.9	Class diagrams of the Discovery Library extensions. . . . .	48

6.1	Mono2Micro decomposition visualization with fine-grained interaction between clusters. Edges represent functionalities shared between clusters. . . . .	59
6.2	Snippet of the generated CML related to the functionality <i>ConcludeQuiz</i> . Triple dots (...) represent omitted constructs for the purpose of the example. Service operation names were also truncated. . . . .	63
6.3	Graph representation of the <i>ConcludeQuiz</i> functionality in Mono2Micro, and the clusters that participate in it. Edge numbers represent the number of accessed entities in the clusters they point to. . . . .	64
6.4	BPMN representation of the <i>ConcludeQuiz</i> <i>Coordination</i> in Context Mapper, generated using BPMN Sketch Miner. The tasks in the diagram represent the steps of the <i>Coordination</i> , and each participant (lane) represents the <i>Bounded Context</i> where the step is defined. . . . .	64

# List of Tables

2.1	Available editing operations in Mono2Micro . . . . .	15
2.2	Available refactoring operations in Context Mapper, as listed in [3] . . . . .	18
4.1	Operations in Mono2Micro and Context Mapper. . . . .	33
6.1	Mono2MicroBoundedContextDiscoveryStrategy tests. . . . .	55
6.2	Mono2MicroelationshipDiscoveryStrategy tests. . . . .	56
6.3	Serialization tests of contract-based internal model. . . . .	56
6.4	SketchMinerCoordinationModelCreatorTest class tests. . . . .	56
6.5	SketchMinerGeneratorTest class tests. . . . .	56
6.6	SketchMinerLinkCreatorTest class tests. . . . .	57
6.7	ApplicationLayerValidationTest class tests. . . . .	57
6.8	Candidate decomposition measures for the QT case study. . . . .	58
6.9	Refactored functionalities for QT case study. CGI stands for Coarse-Grained Interaction, and FGI stands for Fine-Grained Interactions. . . . .	59
6.10	Generated CML constructs. The number of services is represented by four values: No heuristics used; <i>Full Access Trace</i> used; <i>Ignore Access Types</i> used; and <i>Ignore Access Order</i> used. The number of entities is represented by two values: original entities and reference entities. The most accessed entity is based on external accesses to the <i>Bounded Context</i> . . . . .	60









# Listings

5.1	Example output for the structure collector. . . . .	40
5.2	Segment of the Mono2Micro contract. . . . .	42
5.3	Client code to translate a Mono2Micro contract in the Discovery Library. . . . .	49



# Acronyms

<b>API</b>	Application Programming Interface
<b>AST</b>	Abstract Syntax Tree
<b>CML</b>	Context Mapper DSL
<b>DL</b>	Discovery Library
<b>DDD</b>	Domain-Driven Design
<b>ORM</b>	Object-Relational Mapping
<b>UML</b>	Unified Modeling Language



# Glossary

## **SAGA**

A design pattern for managing complex transactions between distributed services by breaking them into smaller coordinated steps. .... 5, 68





# 1

## Introduction

### Contents

1.1	Context . . . . .	3
1.2	Problem . . . . .	4
1.3	Approach . . . . .	4
1.4	Contributions . . . . .	5
1.5	Document Structure . . . . .	5



## 1.1 Context

Microservice architectures have become one of the architectures of choice for emerging large enterprise applications [4, 5]. This adoption results from the advantages of partitioning a large system into several independent services, which provide qualities such as:

- Strong boundaries between services, which minimize feature tangling and help maintain the system in a modular structure.
- Independent agile development, testing and deployment of each service, which speeds up production and contains failures inside the individual service where the failure occurred.
- Easier integration of new and diverse technologies, which encourages the refactoring of legacy code and promotes service-tailored infrastructures, including different scalability options for each service.

On the other hand, topics such as how to distribute the system and the consistency model might stagger the design early on. The use of a monolith architecture, where the business logic of the system is interconnected and focused in one place, has the advantage that it does not require early modularization. The neat identification of modules occurs through refactorings, after initial development, which allows one to explore the application domain first [6].

Therefore, it is common practice to start with a monolith and, as the system grows in size and complexity, migrate to a more modular architectural approach, such as a modular monolith [7] or a microservice architecture. Since this architectural migration is not trivial [8], recent research has proposed approaches and tools to help the migration process [9, 10].

This has led to the development of Mono2Micro, a modular and extensible tool for the identification of microservices in a monolith system [2]. Mono2Micro focuses on identifying transactional contexts to inform its generated candidate decompositions [11]. To this end, it integrates several approaches, such as static code analysis of monolith accesses to domain entities [12], dynamic analysis of monolith execution logs [13], lexical analysis of abstract syntactic trees of monolith methods [14], and analysis of the history of monolith development [15]. Furthermore, Mono2Micro supports a set of measures and graph views to evaluate the quality of the generated candidate decompositions [16].

## 1.2 Problem

Mono2Micro and most tools that identify microservices on monolith systems accomplish their goals by using a varied array of monolith data, coupling criteria, and clustering algorithms [9]. The result from this process is a decomposition artifact that models a microservice architecture.

However, as with most research on the identification of microservices in monolith systems, Mono2Micro does not allow software architects to further model generated candidate decompositions using Domain-Driven Design (DDD) [17], which has shown good results on microservice design [18] and growing interest in the industry [19]. Instead, Mono2Micro representations of candidate decompositions are based on sequences of read and write accesses to the monolith domain entities, which are difficult to work with when trying to redesign the original monolith system and its functionalities for a modular architecture.

If an architect wants to work on a generated decomposition from the perspective of DDD, they will find that the decomposition elements are either not always easily mapped, leading to extra work, or not complete enough to use the full set of tools DDD provides, specially its strategic patterns.

## 1.3 Approach

This thesis addresses this problem by providing a representation of the Mono2Micro candidate decompositions in terms of tactical and strategic DDD patterns, which is automatically generated. In this way, software architects can work on candidate decompositions using DDD.

This is achieved by extending the operational pipeline of Mono2Micro with a connection to Context Mapper, a DDD-focused modeling tool that provides a Domain-Specific Language, named Context Mapper DSL (CML). CML supports the declarative description of DDD domain models, using DDD concepts as building blocks of the language [20]. To integrate the tools, Context Mapper provides a peripheral module named Discovery Library, which is designed to be a CML mapping utility [21]. With this goal in mind, the following research questions are raised:

- **RQ1:** How can current approaches to the identification of microservices in monolith systems be extended to include DDD.
- **RQ2:** Can the results of a candidate decomposition based on entity accesses be represented in terms of DDD?
- **RQ3:** Can an architect benefit from the use of such an integrated tool when analyzing and working on a candidate decomposition?

To answer these research questions, a real monolith system is used as a case study. The resulting candidate decompositions of this system were generated with and without the new DDD modeling

capabilities and then compared.

## 1.4 Contributions

In practice, the integration of both tools resulted in the following contributions:

- A new data collector on the side of Mono2Micro, focused on collecting source code structural information. This information is lacking in the decomposition representations of the tool, but is necessary for its `DDD` view since `DDD` is structural in nature;
- A new contract between Mono2Micro and Context Mapper, which serves as the means of communication between tools by supplying information about Mono2Micro decompositions to Context Mapper. This also includes an export functionality on the side of Mono2Micro;
- New Mono2Micro decomposition discovery strategies using the Discovery Library module of Context Mapper, that convert decomposition data into valid `CML` code;
- New syntax rules in `CML` on the side of Context Mapper, labeled `Coordination` and `Coordination Steps`, that augment the language and facilitate the representation of not only Mono2Micro decompositions, but also generic inter-*Bounded Context* processes like `SAGAs`. This addition also includes diagram visualization of these rules in the BPMN format.

## 1.5 Document Structure

The remainder of this thesis is structured as follows. Chapter 2 gives some background on `DDD`, as well as the Mono2Micro and Context Mapper tools. Chapter 3 goes over the current literature on `DDD` application and microservice identification tools. Chapter 4 presents the solution to the aforementioned research questions, followed by how this solution was implemented in Chapter 5. Chapter 6 provides the validation of the solution with a case study application, and the results and answers to the research questions are discussed. Finally, Chapter 7 concludes the thesis work.



# 2

## Background

### Contents

---

2.1 Domain-Driven Design . . . . .	9
2.2 Mono2Micro . . . . .	11
2.3 Context Mapper . . . . .	15
2.4 Tool Analysis . . . . .	19

---





To better inform the integration of Context Mapper into the Mono2Micro pipeline, this section gives an overview of the architecture of both tools and analyses them. Before that, it also provides insight on the basics of Domain-Driven Design (DDD), including the main concepts and patterns, so that its value as a design philosophy is better understood.

## 2.1 Domain-Driven Design

Being introduced by Evans in his book [17] and expanded upon by Vernon in [22], Domain-Driven Design (DDD) provides a way to design software with a focus on the business for which the software is being developed. This top-down approach leverages the knowledge of domain experts, which know the inside-outs of the business, to help software developers envision domain models that represent the business domain.

The literature provides patterns that help organize and solve complex problems during the design and development of software. Tactical design patterns like *Entity*, *Value Object* or *Aggregate* are applied closer to implementation, and focus on defining meaningful objects that capture elements of the domain and clustering them appropriately. Strategic design patterns, like *Context Map* and *Bounded Context*, are applied at a higher level, and help partitioning the domain into more focused sub-domains, with different models and development teams, so that software becomes easier to maintain and scale.

### 2.1.1 Domain and Model

When talking about DDD, the word domain refers to a subject area that exists in reality. Stakeholders in a project are most of the time not interested in modeling the entirety of a business domain, only selected aspects of it that are crucial for the application at hand. A domain can thus be broken down into sub-domains, which are more easily tackled by modelers.

The process of modeling involves a continuous dialogue between business experts and developers, and produces what DDD calls domain models. These are images of reality that abstract the aspects relevant to what is being architected, and underlie not only design and implementation, but also team communication, since they are the source of common vocabulary to be used by all stakeholders in the context of the model. This last point is formally defined in DDD as *Ubiquitous Language*.

### 2.1.2 Bounded Context and Context Map

Given these definitions for domain and model, a *Bounded Context* functions as a border that contains a specific domain model, i.e. where the model is valid and applicable. As more *Bounded Contexts* are

identified, along with the relationships between them, a *Context Map* starts to be formed. A *Context Map* serves as an overview of the project structure.

### 2.1.3 Bounded Context Relationships

Although *Bounded Contexts* define barriers, some level of relationship must exist between different contexts in a *Context Map*. DDD defines several patterns to express how *Bounded Contexts*, and the teams that develop them, should interact with each other.

A *Shared Kernel* indicates that a subset of the domain model, and its design and integration, is shared between two *Bounded Contexts*, and their respective teams; A *Customer/Supplier* relationship between two *Bounded Contexts* ensures that the upstream context, i.e the supplier, which has influence over its peers, respects the needs of the downstream context, i.e. the customer; *Conformist* is used by the downstream context when there is a high level of dependency with the upstream, but the upstream does not intend to provide for the needs of the downstream. In this case, the downstream submissively conforms to the model and language of the upstream; *Anticorruption Layer* serves as a translation layer when the isolation and protection of the downstream model from the upstream model is paramount; *Open Host Service* specifies and exposes, through a *Published Language*, a set of services that provide access to a subsystem when the functionality of this subsystem is needed by many different downstream contexts.

### 2.1.4 Entity, Value Object and Service

Looking now at tactical DDD applied inside of *Bounded Contexts*, classes and their objects can be classified according to their internal behavior and intended purpose. Objects that have identity and a life cycle that spans different states are called *Entities*. On the other hand, objects that only serve to describe those states and are themselves stateless, transient and immutable are called *Value Objects*. When classes contain actions or operations that do not fit in as an *Entity* or *Value Object*, they are called *Services*, not to be confused with the concept of API service on the application layer of a system.

### 2.1.5 Aggregate

At the heart of tactical DDD is the concept of *Aggregate*. An *Aggregate* is a cluster of associated domain objects that can be seen as a unit for the purposes of data changes. Each *Aggregate* defines a root and a boundary: The boundary defines what is inside the *Aggregate*, which includes *Entities* and *Value Objects*, among others; and the root is a specific *Entity*, called the *Root Entity*, which serves not only as the *Aggregate* entry point, but also as the reference point outside the *Aggregate* boundaries. The

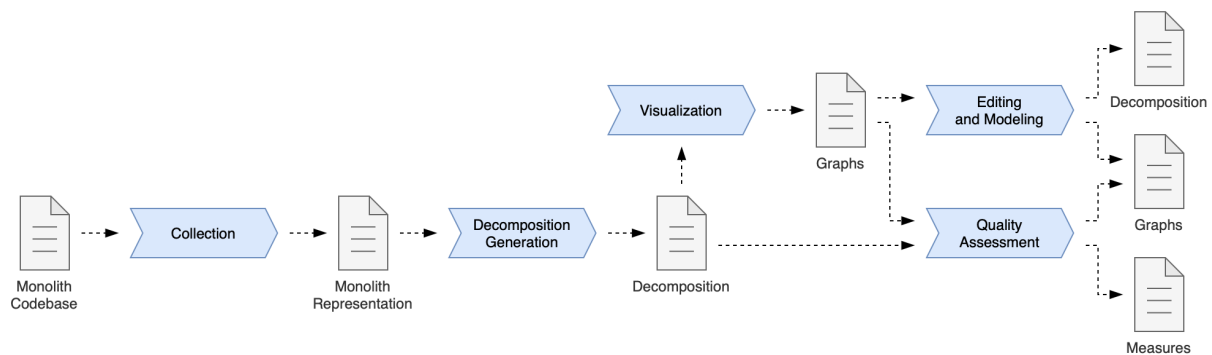
*Aggregate* must also remain consistent as a whole by following certain invariants, or consistency rules, contained in it.

### 2.1.6 Domain Event

*Domain Events* are objects that capture state changes inside *Aggregates*, and relegate this changes to the outside so that interested parts of the system, like other *Aggregates* or *Bounded Contexts*, can be notified of the change and act accordingly.

## 2.2 Mono2Micro

Mono2Micro is a migration tool that provides candidate monolith decompositions composed of clusters of domain classes. This work initially focused on the identification of microservices driven by the identification of transactional contexts [11], but other strategies have been added [13–15].



**Figure 2.1:** The five stages of the Mono2Micro operational pipeline [2]. Each stage can use the output of former stages as input.

Mono2Micro is designed as a pipeline [2], which is represented in Figure 2.1. The five stages of the pipeline are:

1. *Collection*: Implements several static and dynamic code collection strategies to represent monoliths, including representations based on accesses to source code domain entities, functionality logs, and commit history and authors.
2. *Decomposition Generation*: Partitions the monolith domain entities into clusters using a set of similarity criteria, with a focus on producing good quality decompositions.
3. *Quality Assessment*: Compares the decompositions and calculates the measures that are used to evaluate the generated decompositions. The measures include coupling, cohesion, size, and complexity.

4. *Visualization*: Depicts decompositions in the form of graphs with multiple levels of detail. Nodes and edges can represent different elements, depending on the chosen collection strategy.
5. *Editing and Modeling*: Provides an interface with operations to modify the automatically generated decompositions so that the architect can refine them. Quality measures are also automatically recalculated, if applicable.

The following sections describe the five stages of the tool in more detail.

### 2.2.1 Collection Stage

In Mono2Micro, the collection stage is represented by the External Collectors module. Unlike others, this module is not fully integrated in the architecture due to the variety of collection strategies that can be used when representing a monolith [2]. For example, while static analysis reads bare source code, dynamic approaches might need extra steps in this stage like source code instrumentation. Leaving the module decoupled adds flexibility for when new collection strategies need to be introduced with varying technology.

As it is, Mono2Micro implements several collection strategies, both static and dynamic, to represent monoliths: A call graph with the sequence of accesses to the source code domain entities (reads and writes) [11, 12]; a dynamic tracer that logs accesses per functionality [13]; and a development history collector based on the commit history and code authors [15].

One of the more prominent collectors Mono2Micro provides is the Spoon-Collector [12], which will be relevant later on when the solution pipeline is discussed. This collector uses a static analysis framework, called Spoon [23], to analyze source code and collect information on sequences of accesses to domain entities per functionality, supporting source code written using two distinct Object-Relational Mapping (ORM) frameworks: Fenix Framework and Spring Data JPA. The sequences of accesses to domain entities are collected by capturing the call graph associated with each controller. A controller represents a monolith functionality. From that point on, a depth first search is performed into each method call within the controller, and every single line of code is analyzed for a possible domain entity access. In broad terms, a read or write access is registered when either the fields of a domain entity are accessed, directly or indirectly, or a method from a repository responsible for persisting a certain domain entity is accessed.

The information collected is then compiled into a `Representation` artifact, to be used in the next stage. This `Representation` is what connects the External Collectors module output to the rest of the pipeline. It is also a point of extension in this stage for when new collectors are added.

### 2.2.2 Decomposition Stage

The decomposition stage is responsible for generating a decomposition of the monolith based on the information collected from its representation, and according to a set of similarity measures relating the different monolith elements within the representation. The process uses a hierarchical clustering algorithm, with a focus on maximizing internal cluster cohesion, while minimizing coupling between clusters and the complexity of distributed transactions in candidate decompositions [11].

This stage is broken down into two modules, Similarity Generator and Aggregation Algorithm. Like other modules, these exist as extension points, so that different similarity measures and different algorithms can be implemented in the tool, respectively. For example, to minimize distributed transactions, the tool uses four similarity measures: Access similarity, read similarity, write similarity and sequence similarity [12]. Other similarity measures are supported, such as commit similarity and authorship similarity formulae when the representation also includes the development history [15]. The first 4 similarity measures apply to pairs of domain entities, and are calculated based on the system functionalities, i.e. number of accesses, reads, writes or sequential accesses done by functionalities on the entities.

When reaching this stage, the architect can specify before the decomposition the weights of several metrics, described in the next paragraphs, and the number of resulting clusters.

### 2.2.3 Quality Assessment Stage

With the decomposition artifact, the Quality Assessment stage can compare the result with other decompositions and calculate the values of the metrics that are used to evaluate the generated decomposition. While the Comparison Tool module handles the comparisons, the module responsible for qualifying metrics is the Metrics Calculator. This module takes into account four metrics [11, 12, 15], whose values range from 0 to 1: Complexity (the cost of implementing the migration); cohesion (between the domain entities of a cluster); coupling (between clusters); and team size reduction (of authors of source code).

The complexity metric is defined, per functionality, as the cost of the functionality redesign when undergoing the migration. Here, functionality redesign is in regards to the transactional context, i.e. the change from an ACID context to a distributed one. The complexity depends on the number of intermediate states that other functionalities can introduce on the entities read by the functionality. The cohesion metric measures, per cluster, the percentage of entities that are accessed by a given functionality. The more entities that are accessed, the more cohesive the cluster is. The coupling metric measures, between two clusters, the percentage of entities that are exposed to one another. The team size reduction metric uses the number of authors of the original monolith system to measure the level of reduction in team sizes when clusters are formed, by giving an average of contributing authors per cluster.

### 2.2.4 Visualization Stage

After generating a decomposition, the next step for the system is to represent it in a way that enables the architect to visualize it, from different perspectives if possible [2]. The Visualization stage accomplishes this by depicting the decomposition in the form of graphs, whose nodes and edges can represent different elements depending on the view and level of detail the architect chooses.

Powered by the View module, the graphs rendered by the tool represent the different clusters from the decomposition with different colors. In the broadest of views, the graph nodes can map to clusters, and the edges between nodes can map to the dependencies between clusters. A more detailed view can instead represent the nodes as entities, and clusters become defined as the group of nodes that share the same color. In this case, edges play the crucial role of grouping together the same-colored nodes in the graph view so that clusters are easily spotted. This is done by having the length of edges between nodes represent their cophenetic distance, that is, how similar both nodes are to each other based on similarity measures from a previous stage [2]. Entities from different clusters are less similar to those in their cluster, so the length of the edges connecting two entities from different clusters becomes bigger compared to when connecting two entities from the same cluster. Additionally, the thickness of edges is also used to represent the number of functionalities between the two entities, or number of shared authors [15].

These different measurements that make up the graph representation can be accessed when selecting a cluster, entity, or edge in the view, which displays information regarding that element. Besides the metrics previously discussed and the values used for the graph representation, it is also here that one can see which functionalities access the selected element, and what type of access it is (read/write).

### 2.2.5 Editing and Modeling Stage

The last stage of the pipeline is responsible for providing an interface for operations to modify the resulting decomposition. The purpose of including this stage is so that the architect has the tools to refine the decomposition into something closer to what they have envisioned. As such, operations will trigger the recalculation of metrics and redraw of the view graphs each time they are applied [2]. The available operations are listed in Table 2.1.

Beside these four operations, there is one additional operation that can be applied on candidate decomposition: Redesign Functionalities Transaction Sequence.

This operation is implemented in the Refactorization Tool, which is an extension of Mono2Micro implemented in its own container, with its own model and language.

What is particular about this operation is that it refactors the functionalities of a decomposition in a way that minimizes their migration complexity. As addressed in [24, 25], this operation is the result of

**Table 2.1:** Available editing operations in Mono2Micro

Name	Subject	Description
OP-1: Merge Clusters	Cluster	Merges two clusters into a single cluster.
OP-2: Split Clusters	Cluster	Extract selected entities into a new cluster.
OP-3: Transfer Entities	Entity	Transfer selected entities from one cluster to another selected cluster.
OP-4: Rename Clusters	Cluster	Change the name of a selected cluster.

the realization that, due to the monolithic architectural style, the structure of functionalities in monoliths includes a lot of fine-grained interactions between objects, which can translate to an excess level of invocations between microservices if not considered and addressed during the migration. The consequence of not refactoring the functionalities structure when moving to microservices will be a higher complexity value presented by the metrics, leading to a non-optimal and skewed view of the migration complexity.

Besides the operations that modify the decomposition, there are also operations that only change the view of the graph, making it easier to evaluate the decomposition. These include: a way to show or hide the contents of a cluster; a way to show only nodes and edges which neighbor the selected node; and a way to unlock node positions so that the nodes can be moved and rearranged in the graph [2].

## 2.3 Context Mapper

Mono2Micro is a versatile microservice identification tool. However, its pipeline does not include any way for an architect to model candidate decompositions using DDD after the *Decomposition Generation* stage. More concretely, in the *Visualization* stage, graph representations of the decomposition include cluster-based views of the decomposition domain entities, and functionality-based views that represent its sequence of accesses to domain entities ("Graphs" in Figure 2.1). There are no DDD-based views that show the model of each candidate microservice. Likewise, the *Editing and Modeling* stage does not contain any operations related to the application of DDD. This is where Context Mapper comes in.

Context Mapper is a modeling framework that provides a DSL to design systems using DDD concepts. This DSL, henceforth called Context Mapper DSL (CML), was developed to unify the many patterns of DDD and their invariants in a concise language [20]. Figure 2.2 shows an example of the CML syntax, with the declaration of a *Context Map* containing two *Bounded Contexts*.

Within *Bounded Contexts*, one can define *Aggregates*, which consist of a group of closely related domain objects that form a unit for the purpose of data consistency. This consistency is enforced inside the *Aggregate* by its root *Entity*, which represents the only entry point. For example, in Figure 2.2 the

---

```

1 ContextMap InsuranceContextMap {
2     contains CustomerManagement
3     contains CustomerSelfService
4
5     CustomerManagement [U] -> [D] CustomerSelfService
6 }
7
8 BoundedContext CustomerManagement {
9     Aggregate Customers {
10         Entity Customer {
11             aggregateRoot
12
13             - List<Address> addresses
14             String name
15         }
16
17         Entity Address {
18             String city
19             int postalCode
20         }
21     }
22
23     Application {
24         Service CustomersService {
25             void createCustomer(String name)
26             @Customer getCustomer(String name)
27         }
28     }
29 }
30
31 BoundedContext CustomerSelfService {
32 }

```

---

**Figure 2.2:** Example syntax of CML, containing the syntax for defining a *Context Map* (1-6); *Bounded Contexts* (8-29, 31-32); *Aggregates* (9-21); *Entities* (10-15,17-20); and *Services* (24-27).

Customers aggregate has the Customer entity as its root.

Although DDD focuses on the *Domain Layer* of systems, where the business logic is residing, a CML Bounded Context can also represent the *Application Layer*, which manages services that call different parts of the system, including processes in other layers. Using the `Application` keyword, *Application Services* can be defined, among other constructs, and contain operations like `createCustomer` and `getCustomer` as represented in Figure 2.2.

In addition to CML, Context Mapper also contains other utilities to facilitate modeling activities. These include the following:

1. *Discovery Library*: Implements several strategies to reverse engineer source code artifacts and represent them in CML [26].
2. *Architectural Refactoring*: Includes operations to refactor and transform CML code for easier modeling.



3. *Diagram Generators*: Provide translators to visualize CML artifacts in diagram form, such as UML representations of *Bounded Contexts* and BPMN maps of *Aggregate* states.

### 2.3.1 Discovery Library

The Discovery Library [21] is implemented as a reverse engineering module that transcribes the source code to a CML representation that includes DDD constructs, like *Bounded Contexts*, *Aggregates* and *Entities* [26].

The module is implemented to work with Spring Boot<sup>1</sup> projects, using static analysis to find specific Spring Boot annotations in source code, and mapping them to the relevant DDD concepts used in CML as follows: Each `@SpringBootApplication` is mapped to a *Bounded Context*; Spring REST endpoints that use the `@RequestMapping` at the class level are mapped to *Aggregates* with one *Root Entity*; the methods in controllers that use `@PostMapping`, `@GetMapping`, etc are mapped to methods of the *Root Entity*; and parameters/return types from these controllers, which can include DTOs<sup>2</sup>, are mapped to *Value Objects*.

A *Context Map* is also generated through the analysis of the relationships of the several Spring Boot applications. This is implemented by interpreting the docker-compose file from Docker<sup>3</sup>, which lists the different containers in the system and their dependencies. The result from this process is an initial CML view of the source code, which can then be further refined either manually, or through the other available modules, like Service Cutter or Architectural Refactorings.

### 2.3.2 Architectural Refactorings

The Architectural Refactorings module of Context Mapper is composed of 11 code refactoring operations that can be applied to CML artifacts. These operations are split into two categories: structural refactorings and relationship refactorings. Both categories of operations apply modifications to CML code at the level of DDD: structural refactorings are able to re-structure the CML decomposition by modifying *Bounded Contexts* and *Aggregates*; and relationship refactorings are able to modify relationship types between *Bounded Contexts* in *Context Maps*. Table 2.2 lists the available refactoring operations in Context Mapper and the results off applying them.

---

<sup>1</sup><https://spring.io/projects/spring-boot>

<sup>2</sup>[https://en.wikipedia.org/wiki/Data\\_transfer\\_object](https://en.wikipedia.org/wiki/Data_transfer_object)

<sup>3</sup><https://www.docker.com>

**Table 2.2:** Available refactoring operations in Context Mapper, as listed in [3]

Name	Subject	Description
AR-1: Split Aggregate by Entities	Aggregate	Takes an aggregate which contains multiple entities and produces one aggregate per entity.
AR-2: Split Bounded Context by Features	Bounded Context	Takes a bounded context with several aggregates and groups them into different bounded contexts based on the aggregates use case(s) and/or user stories (features).
AR-3: Split Bounded Context by Owner	Bounded Context	Takes a bounded context with several aggregates and groups them into different bounded contexts based on the aggregates owner (team).
AR-4: Extract Aggregates by Volatility	Bounded Context	Takes a bounded context with several aggregates and extracts the ones with a given likelihood for change value (volatility) into a different bounded context.
AR-5: Extract Aggregates by Cohesion	Bounded Context	Takes a bounded context with several aggregates and extracts the ones chosen by the user into a different bounded context.
AR-6: Merge Aggregates	Aggregate	Merges two aggregates within a bounded context into one aggregate.
AR-7: Merge Bounded Contexts	Bounded Context	Merges two bounded contexts together. The result is one new bounded context containing all the aggregates of the two other bounded contexts.
AR-8: Extract Shared Kernel	Shared Kernel relationship	Extracts a new bounded context for the common model parts of the Shared Kernel and establishes two upstream-downstream relationship between the new and existing Bounded Contexts.
AR-9: Suspend Partnership	Partnership relationship	Suspends a Partnership relationship and replaces it with another structure how the two Bounded Context can depend on each other. The AR provides three strategies to suspend the partnership.
AR-10: Change Shared Kernel to Partnership	Shared Kernel relationship	Changes the type of a Shared Kernel relationship to a Partnership relationship.
AR-11: Change Partnership to Shared Kernel	Partnership relationship	Changes the type of a Partnership relationship to a Shared Kernel relationship.

### 2.3.3 Generators

Context Mapper provides a Generators module, capable of converting CML code into a varied set of diagrams. Context Maps can be converted into context map diagrams, as presented in the DDD book [17]. Bounded Contexts can be translated into Unified Modeling Language (UML) diagrams by using PlantUML<sup>4</sup> generators, including class diagrams that contain all the Entities information represented as classes, use case diagrams that convert Use Case and User Story definitions in CML, and state diagrams, from converting Aggregate state transitions if their are defined within the Aggregate. Processes that are modeled like Event Flows in the application layer of *Bounded Contexts* can also be converted and viewed with BPMN diagrams.

Another type of generator support are service contracts. This includes generating contracts in the Microservice Domain Specific Language (MDSL), which is another DSL for specifying microservices, and that can lead to direct code generation for Open API, gRPC, Jolie, GraphQL, and plain Java.

## 2.4 Tool Analysis

Each Mono2Micro stage is composed of one or more modules that output artifacts for the next stage in the pipeline. The underlying model of the tool that makes up these modules and artifacts is also built with several extension points, making it possible to support multiple decomposition strategies.

However, this pipeline does not include any way for an architect to model candidate decompositions using DDD after the *Decomposition Generation* stage. More concretely, in the *Visualization* stage, graph representations of the decomposition include cluster-based views of the decomposition domain entities, and functionality-based views that represent its sequence of accesses to domain entities ("Graphs" in Figure 2.1). There are no DDD-based views that show the model of each candidate microservice. Likewise, the *Editing and Modeling* stage does not contain any operations related to the application of DDD. This is where Context Mapper comes in.

The features of Context Mapper have many similarities with the features of Mono2Micro, which facilitate creating an integration strategy for both tools. First, the Discovery Library performs a similar job as the Collectors of Mono2Micro, but more importantly, it provides a way to generate CML from its input. Second, the Architectural Refactoring (AR) module supports the architect on the edition and modeling of CML models, as the Editing and Modeling stage of Mono2Micro. However, AR operations are built on DDD concepts. Finally, the Diagram Generators module can provide ways to view a candidate decomposition from the perspective of DDD, also something missing in Mono2Micro, which presents decompositions as a graph of clustered domain entities.

---

<sup>4</sup><https://plantuml.com>



# 3

## Related Work

### Contents

---

3.1 DDD Modeling Tools . . . . .	23
3.2 Microservice Identification Tools . . . . .	24

---



The previous chapter focused on analyzing both sides of the integration pipeline and the concepts in Domain-Driven Design (DDD). In this section, the focus is on how DDD is more generally interpreted and applied in other modeling activities from the industry, by analyzing the state of the art in: tools that bridge the gap between DDD and other models from different development stages by using concept translations; and microservice identification tools and their relation (or lack thereof) to the DDD approach to software architecture.

### 3.1 DDD Modeling Tools

Research on DDD modeling techniques is still sparse, especially in terms of modeling tools that leverage tactic and strategic DDD patterns [19]. A study on how practitioners implement DDD concepts when designing microservice APIs [27] shows that DDD, although practiced, is still poorly formulated and open to interpretation in its applicability.

Most research takes advantage of already developed models and diagram standards in the industry to convey DDD concepts. The use of annotated constructs is one of the most common approaches, such as in [28], where an annotation-based DSL was developed to scope objects and attributes within the concepts of DDD. Another implementation of this is [29], where a mapping from DDD to UML is presented with the use of annotations inside UML class constructs. The author highlights that DDD itself is not a formal modeling language and that its presentation in [17] already leverages UML constructs. He proposes in his paper an enhanced mapping of DDD elements to UML by providing a new UML profile. Aggregates, Entities and Value Objects are represented as annotated classes and Bounded Contexts as annotated packages. Integrating DDD in a widely used modeling framework like UML has the advantage of having a familiar diagram view of the systems modeled with DDD.

However, these approaches do not support all DDD patterns, especially strategic ones such as *Bounded Context* relationships, which are useful when modeling microservices from candidate decompositions. Nonetheless, this shows that there is common ground for the mapping between DDD concepts and other diagram based design tools like UML.

In [20] it is also noted how architecture description languages lack full support for all DDD patterns, and thus defines a meta-model of relevant DDD concepts, applying it through a DSL that emphasizes DDD strategic patterns, and leveraging the already existent Sculptor tool <sup>1</sup> for mapping tactical patterns.

Other research also explores the extensibility of DDD to better fit other stages of software development. In [30] they define *Domain Views*, which enable different stakeholders to perceive the domain model with their respective knowledge base. The Context Mapper tool also provides *Domain Views* through the definition of types of *Bounded Context* and *Context Maps* [20].

---

<sup>1</sup><http://sculptorgenerator.org>

## 3.2 Microservice Identification Tools

There has been extensive and recent research on the identification of microservices in monolith systems [9]. These approaches provide a rich set of decomposition criteria and metrics to assess the generated decompositions. In particular, Mono2Micro [2] is a modular and extensible tool for those criteria and methods. However, these tools do not provide output that enables DDD-based editing and modeling, they mostly provide decompositions that are service-oriented and not domain-oriented.

This does not mean DDD is not a good fit for microservice identification processes. In [31], the case is made for why DDD is part of the solution to identify microservices. DDD and its patterns advocate partitioning of the domain and parallelization of development. This design philosophy makes DDD an ideal tool to apply when working on a microservices architecture, since the distributed nature of microservices is closely related to the modularity of DDD, especially the concept of *Aggregate*. Aggregates can be used to cluster related domain objects, like entities and value objects, and are viewed as a whole when thinking of partitions or transactional contexts. The way in which aggregates reference each other also has similarities with the relationships between microservices. The fact that aggregates are referenced elsewhere by the ID of their root entity instead of a direct reference to the said entity gives them a loosely coupled property alike to microservices, since they are conceptual and physically separate.

The only tool that was found to support DDD-based editing and modeling is the Discovery Library tool [20], which provides a way to reverse engineer domain models from Spring Boot<sup>2</sup> service APIs using discovery strategies. From the analysis of the code, it finds specific Spring Boot annotations and maps them to the corresponding DDD concepts. It generates *Bounded Contexts* from `@SpringBootApplication` annotated classes and *Aggregates* from `@RequestMapping` annotated classes.

---

<sup>2</sup><https://spring.io/projects/spring-boot>



# 4

## Solution Architecture

### Contents

---

4.1 Architectural Requirements . . . . .	27
4.2 Pipeline Extension . . . . .	29

---



Before moving on to implementation details in the next chapter, it is important to first consider the architecture of the solution pipeline and what it aims to solve, especially when the scope of the problem is large enough to involve multiple tools and services developed by separate parties. With that in mind, the first section of this chapter lays out the practical requirements the solution should have, based on the overall objectives of this work and the analysis of both tools presented in Chapter 2. This is followed by a view of the solution architecture that passes through devising an integration strategy between Mono2Micro and Context Mapper, and includes a description of new modules, their reasoning, how they integrate within the existing architecture of both tools, and how they communicate with each other.

## 4.1 Architectural Requirements

As stated beforehand, the main problem this thesis is trying to address is the lack of Domain-Driven Design (DDD) representations in current microservices identification tools, which is relevant considering that DDD is extensively used to design microservices from scratch. To solve this problem, the pipeline of Mono2Micro was extended to incorporate a way to model with DDD using Context Mapper. This left a more concrete problem to solve: What is the optimal way to integrate Context Mapper in the Mono2Micro pipeline?

In Chapter 2 some points were already made about why these tools were chosen. From this analysis, it was concluded that the tools share some similarities not only in some of their modules and processes, but also in regards to their architectural properties.

At minimum, the solution for integrating both tools must share the same architectural properties without reducing their value in each tool, so it was decided that these properties should serve as the base requirements for every architectural decision made in the solution pipeline.

The following three architectural requirements were identified as priority requirements for the solution architecture:

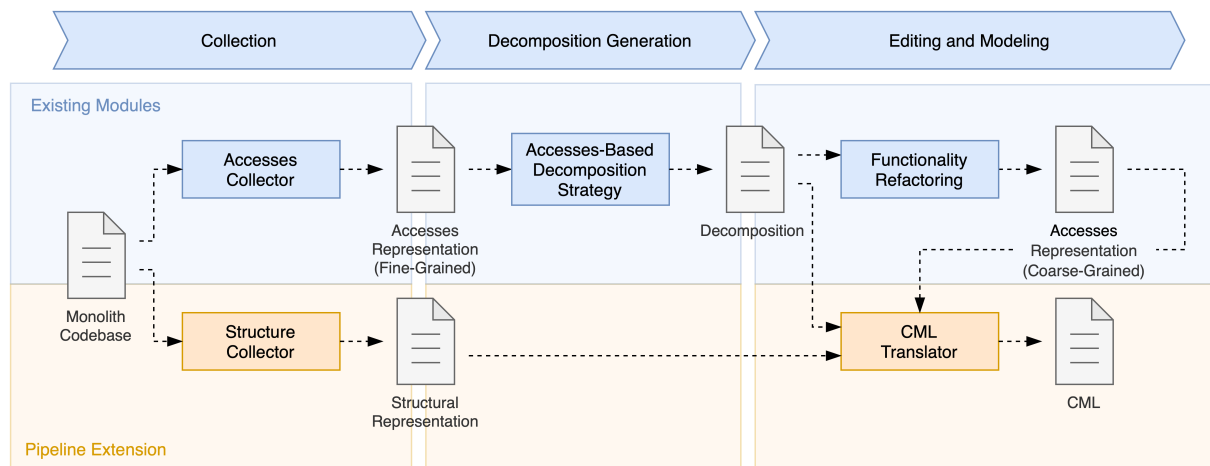
- **Modularity**, which deals with how divided a system is into logical modules that encapsulate specific self-contained functionality, improving separation of concerns and internal cohesion;
- **Extensibility**, which deals with how open for extension the features of a system are without putting at risk their core structure, improving the addition of new functionality;
- **Interoperability**, which deals with how separate systems effectively communicate to share data, and provide convenient data formats and interfaces to do so, improving system integration.

When looking at the practical context of integrating Mono2Micro and Context Mapper, these three architectural requirements can be used to derive the following:

- R1 The solution must not affect the internal cohesion of existing modules in both tools.** This means that modifications to existing modules should respect the module's internal model and objectives. An example of this requirement being broken would be to add new syntax to CML that did not conform with its declarative DDD nature.
- R2 The solution must keep modules of different tools loosely coupled.** This means that each tool should continue to work independently of the other, regardless if there exists a connection between them. A CML syntax change in Context Mapper should not make Mono2Micro stop working, and a model change in Mono2Micro should not make Context Mapper produce errors.
- R3 When possible, the solution should leverage the extensible nature of the tools.** This means that before creating new modules or functionalities from scratch, existing extension points should be considered if their purposes match the needs of the solution. For example, to add a new CML diagram representation in the UML format, the available PlantUML abstractions that exist in the Generators module of Context Mapper should be extended.
- R4 The solution should be open for extension.** Just like each tool, there should be room to extend the different parts of the implemented solution for future developments when applicable. For example, if a new source code collection strategy is added to Mono2Micro, this strategy should be implemented with abstractions for extension points.
- R5 The solution must define clear interfaces on both sides of the integration.** This means that there should be a concrete way to start data transferring on Mono2Micro, and start the translation of this data in Context Mapper. Together with R1, these interfaces should be correctly placed within each tool. If no interfaces are defined, there is no way for the tools to communicate effectively.
- R6 The solution must define a format for the data that is transferred between tools.** This means there must exist some type of contract between Mono2Micro and Context Mapper. Together with R2, this also means that this contract should be loose (simple key value pairs in a text document) rather than strict (using REST or gRPC).

## 4.2 Pipeline Extension

Given these architectural requirements for the solution, the pipeline extension can be defined. Figure 4.1 shows this extension in terms of modules and their input and output artifacts. The top process bar represents the relevant stages of the Mono2Micro pipeline, and the different colors separate existing modules from new ones. The following sections, each corresponding to one of the research questions, explain each module and artifact in more detail.



**Figure 4.1:** Mono2Micro pipeline extension to support CML representation of candidate decompositions. In blue, the relevant pipeline steps (top) and modules. In orange, the extension to the pipeline, composed of the addition of new modules.

### 4.2.1 Tool Integration

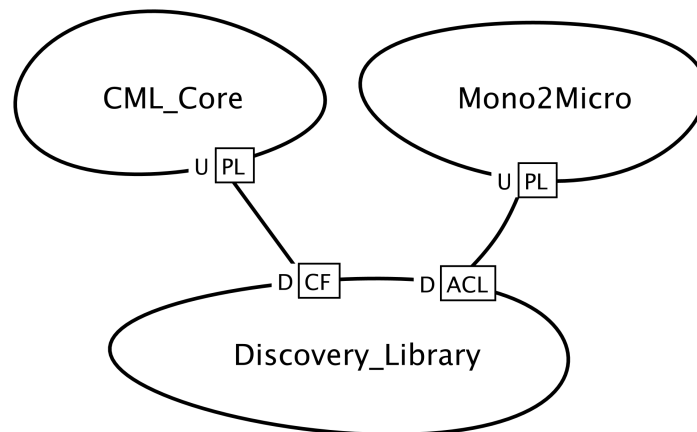
As stated before, Mono2Micro and Context Mapper share an emphasis on modularity and extensibility. This makes it viable for Context Mapper to integrate into the Mono2Micro pipeline. However, it is still important to respect the models of each tool to avoid compromising their internal cohesion. In practice, this meant pursuing a low-coupling solution when connecting the tools. This solution was achieved by leveraging on the Discovery Library (DL) module of Context Mapper.

As described in Chapter 2, the DL is a standalone tool capable of generating CML code. This is done using discovery strategies that translate input into CML. Since the DL was designed to be highly extensible, it also provides an API for the creation of these discovery strategies. Using this API, the Mono2Micro pipeline was extended with a module that defines new discovery strategies capable of translating candidate decompositions into CML. This module is represented by the *CML Translator* in Figure 4.1.

The *CML Translator* has two stages. In the first, the internal representations of a decomposition in the Mono2Micro model are used to create a JSON contract that contains all the information needed to map

a candidate decomposition in CML. This contract serves as input for the new discovery strategies and adds a layer of decoupling between the Mono2Micro model and the DL model, ensuring that changes made to the former do not inadvertently propagate to the latter. In the second stage, the new discovery strategies translate the contract to an internal representation of CML in the DL model. This model is, in turn, automatically converted to actual CML code.

Figure 4.2 shows how the DL is used in the integration process from the perspective of DDD. On the right, the DL acts as an *Anti-Corruption Layer* (ACL), protecting its internal model from the Mono2Micro model present in the inbound contract. On the left, the DL model aligns closely with the CML model to facilitate the generation of CML, represented by the use of the *Conformist* pattern (CF). In both cases, the DL is the downstream (D) context, meaning it is dependent on the upstream (U) contexts and their *Published Languages* (PL). CML is the *Published Language* of the CML Core context, and the JSON Contract is the *Published Language* of the Mono2Micro context.



**Figure 4.2:** Generated Context Map diagram of the integration between Mono2Micro and Context Mapper. PL stands for *Published Language*, ACL stands for *Anti-Corruption Layer*, and CF stands for the *Conformist* pattern. U and D represent upstream and downstream modules respectively.

### 4.2.2 DDD Mapping

For the new discovery strategies to perform the translation to CML, the concepts that form a candidate decomposition must be mapped to the DDD concepts first. Since DDD and its concepts are structural in nature [17], a candidate decomposition was also structurally defined, based on its internal representation in Mono2Micro.

A candidate decomposition is composed of three key concepts: **entities**, which represent domain classes in the monolith; **clusters**, which represent a set of entities grouped by similarity criteria through a clustering algorithm; and **functionalities**, which represent sequences of read/write accesses to entities in one or more clusters.

Mapping a candidate decomposition to DDD corresponds to mapping these three concepts and understanding what information is needed from Mono2Micro once a DDD concept is chosen. The particulars of this mapping is discussed in the next chapter. In this section, only the architectural consequences of implementing the mapping are explored.

When mapping a concept from Mono2Micro to a concept in CML, three possible scenarios can occur: There is a direct mapping between concepts; there is an indirect mapping between concepts; and there is no mapping between concepts.

The first case occurs when a concept in Mono2Micro is also considered and represented in CML. An example of this is the *Entity* concept, which is present on both tools. It is a direct mapping because both tools base their respective concept of *Entity* on the DDD definition of an *Entity*.

The second case occurs when a concept in Mono2Micro does not have a direct correspondent in CML, but can still be adapted to an existing construct without losing context to preserve **R1**. An example of this case is the cluster concept in Mono2Micro. CML does not define clusters, but the definition closely matches the definition of an *Aggregate*, or a *Bounded Context*, so an indirect mapping could be possible.

The third case occurs when a concept in Mono2Micro does not match any available construct in CML, or the match results in a noticeable distortion of the concept after translation. An example of this would be the concept of functionality in Mono2Micro. There is no CML syntax for this term, and the available candidates, such as *Use Cases* and *Flows*, both fundamentally change the concept of a Mono2Micro functionality, as described in the next chapter.

The consequences of this last case involve the extension of the syntax of CML with new constructs to map the incoming Mono2Micro concepts. Like other additions to the solution, this extension must take into account the requirements of the previous section. The most relevant one here is **R1**: Any additions made to the CML syntax must be in accordance with the internal model of the language. If this is the case, the additions are implemented in Context Mapper and architects can use the new concept along side the others with the same objectives. Otherwise, the additions must be implemented in a new separate module that still extends CML, but allows representation of concepts that are exclusively from the Mono2Micro model, using **R2** and **R3** as guidelines. In this last case, the architect is assumed to be working with migrations from the Mono2Micro tool.

### 4.2.3 CML Representation and Interaction

When using Mono2Micro, architects now have the option to convert candidate decompositions to CML using the translation strategy mentioned so far.

Like all CML discovery strategies, the initial representation of the candidate decomposition in CML is not final. Further refactoring is expected. However, an effort was made to automatically create a good starting point. The names of entities, clusters, and functionalities from the initial decomposition are

maintained and used for naming *Entities*, *Aggregates*, *Bounded Contexts*, and *Coordinations* in CML. The conversion of the format of functionalities to Sagas also creates additional constructs, in the form of *Service* operation calls, which correspond to *Coordination* steps in CML. These operations make up the interface of each *Bounded Context* in CML, but there is no straightforward name that can be used to name each operation. As such, several access-based naming heuristics were implemented in the translation strategy. Names are formatted with the names of entities, which are prefixed by the type of access and separated by a dash. For example, assuming that the sequence of accesses of the *Coordination* in Figure 5.7 is: *Tournament* read for step 0; *Quiz* read/write and *Question* read/write for step 1; and *Tournament* write for step 2. Each service call name would be generated as *rTournament*, *rwQuiz\_rwQuestion* and *wTournament*, respectively. The architect can further customize the level of detail they want the name to have regarding access information:

- Full Access Trace: Transcribe the entire ordered entity access sequence that happens within an operation into the name of that operation;
- Ignore Access Types: Omit the type of access to entities in operation names, i.e. read/write, replacing it with a "ac" prefix;
- Ignore Access Order: Omit the type and order of access to entities in the operation names.

Each heuristic used reduces the number of generated operations, at the cost of entity access details. In its most reduced form, each operation name shows which entities are accessed in that step. Access information is also added to each translated entity in the form of a comment, showing metrics related to the percentage of external and local accesses to the entity in comparison with the total external and local accesses to the *Bounded Context*. In contrast to these heuristics, there is also the option to generate generic names for operations that are not access-based. These names are composed of the name of the functionality and the step number in the corresponding *Coordination*, meaning each step of each *Coordination* will call a unique operation.

In terms of interaction, by opening this CML artifact in Context Mapper, architects have at their disposal a DDD-based view of the candidate decomposition, and several features to further refine, refactor, and visualize it. Table 4.1 shows some of these new features compared to what was previously available in Mono2Micro. Architects can refactor decompositions at a more structural level in CML. They can also use the additional DDD patterns present in CML to add to the initial translation, especially strategic ones, to define, for example, what type of relationships exist between the translated *Bounded Contexts*.



**Table 4.1:** Operations in Mono2Micro and Context Mapper.

<b>Mono2Micro</b>	<b>Context Mapper</b>
Expand / Collapse cluster views	Merge / Split Bounded Contexts
Merge / Split clusters	Merge / Split Aggregates
Transfer entities between clusters	Design Bounded Context Relationships
Rename clusters	Redesign Coordinations
Redesign functionalities	Redesign Entities
Analyse access-based measures	Generate UML / BPMN representations



# 5

## Implementation

### Contents

---

5.1	Structure Collector . . . . .	37
5.2	Mono2Micro Contract . . . . .	41
5.3	Translation to CML . . . . .	43
5.4	Discovery Strategies . . . . .	47

---



The previous chapter identified the modules and mappings required for the integration of Mono2Micro with Context Mapper. In this chapter, the implementation of these modules and mappings is described in detail.

## 5.1 Structure Collector

The objective of this new Mono2Micro collector is to recover static structural information on domain entities from the monolith source code, so that richer entity constructs can be written in CML. Figure 5.1 shows the steps taken by the collector to perform this task.



**Figure 5.1:** Steps of the collector when retrieving structural information from source code.

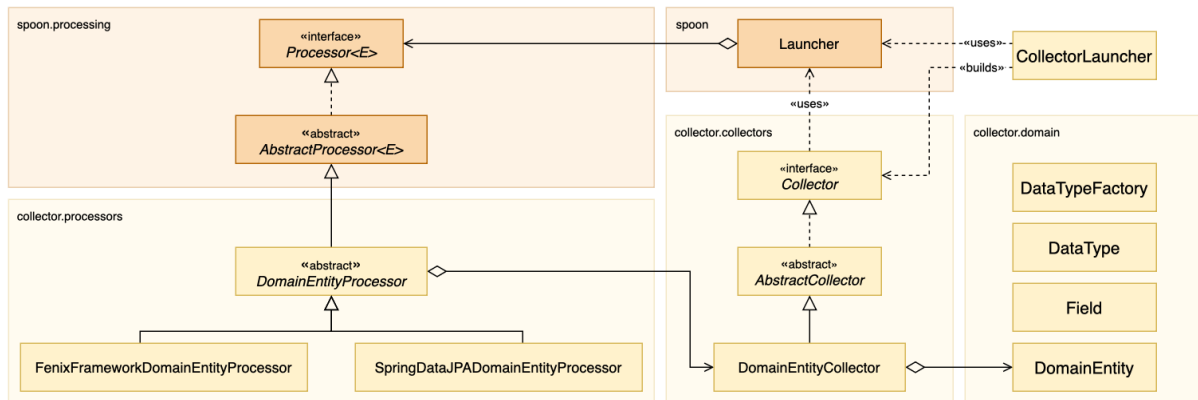
To perform a static analysis, the collector uses the Spoon Framework tool [23]. Spoon is capable of building an Abstract Syntax Tree (AST) of all source code elements, i.e types, methods, expressions, etc. It also provides an Application Programming Interface (API) to not only reference each element, but also perform analysis and transformations on them.

For the work at hand, the elements of interest are domain entity types in the source code. The definition of a domain entity follows the same definition of the *DDD Entity* pattern presented in Section 2.1.4: Objects that have identity and a life cycle that spans different states. Finding these statically will depend on the *ORM* the source code is using. In Spring Data JPA, a domain entity is a class annotated with the `@Entity` annotation, while in Fenix Framework a domain entity is a class whose superclass name ends in `_Base`.

When a domain entity is found while processing the *AST*, this element is translated into the collector internal model and then cached. After the *AST* has been completely traversed, the cached data is serialized and printed on the tool output.

### 5.1.1 Design

In Figure 5.2, the collector design is shown in a class diagram. Classes in yellow are part of the collector code, and classes in orange are part of the Spoon tool API. This image can also be broken down into three main sections: on the left, a processing section; in the middle, a collection section; and on the right, the domain model. Each section will be described in the following paragraphs, starting with the processing section.



**Figure 5.2:** Class diagram of the structure collector. The Spoon API is represented in orange, and the implemented packages and classes are represented in yellow.

To filter out domain entities in the AST, the Processor API of Spoon is used. This module provides several utility classes to traverse the AST and act on its elements. Figure 5.2 shows the implementation of this section in the packages `spoon.processing` and `collector.processors`.

The `DomainEntityProcessor` class extends the `AbstractProcessor<E>` class from the Spoon tool to process elements `E` of the type `CtClass`. Here, `CtClass` corresponds to the Spoon tool representation of class types in the source code. To check if a certain `CtClass` element is a domain entity, the method `isToBeProcessed` is overridden, but kept abstract in the `DomainEntityProcessor` class. This is because the way domain entities are represented in the source code depends on the source code ORM, so a subclass was created for each of the ORMs supported. Each of these subclasses knows how to identify domain entities in its respective ORM. In the case of this collector, the two implemented strategies are for the Fenix Framework and Spring Data JPA.

To build and process the AST in the first place, the `Launcher` class from the Spoon API is used. This `Launcher` receives the path to the source code and also instances of the `Processor` classes. When the `Launcher` runs, it will build the AST, and then process it with each of these provided processors, which are implementations of the Visitor design pattern. When a processor finds a domain entity, it performs two tasks: process the element by converting it into an object from the domain; and cache it using a `Collector`.

A `Collector` corresponds to a cache for the collected data. This data is represented with elements from the domain. In the case of the `DomainEntityCollector` class, `DomainEntity` instances will be stored. A `DomainEntity` is composed of a list of fields, represented by the `Field` class, and a superclass. The data types of these in the source code are represented with the `DataType` class.

When the `Launcher` finishes processing the AST, the data collected is serialized into JSON using the Jackson library. The objects are then printed into a JSON file by converting the objects themselves into JSON object tags. The JSON output has the same structure as the active cached domain of the

collector.

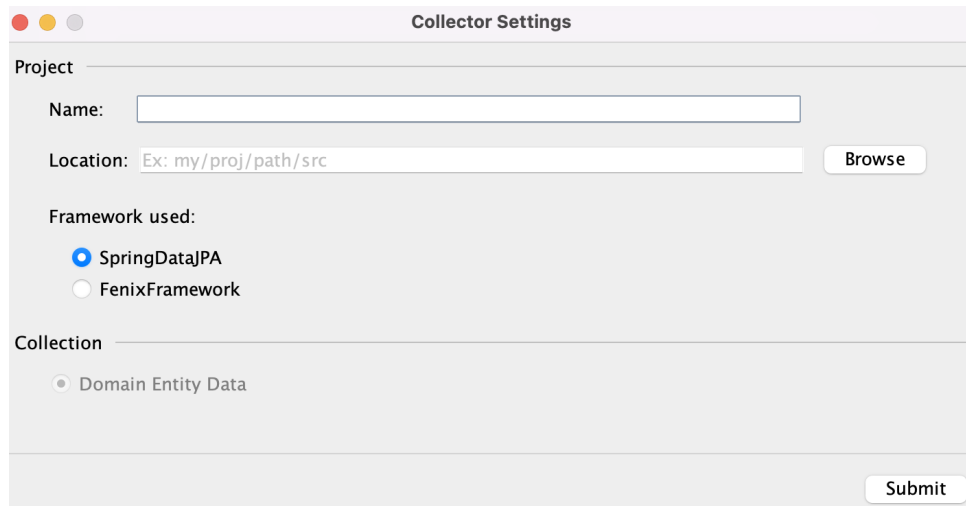
The class responsible for coordinating all these steps is the `CollectorLauncher` class. It creates the `Launcher` instance and builds the `Collector` and processors needed based on user input.

The decision on what type of processing to do depends on knowing two things: What is the `ORM` being used in the source code; and what data collection strategy to use on the source. Considering the **R4** requirement, these are also the two extension points that the collector provides. Figure 5.2 shows these extension points by means of abstractions present in the design of the collector. In the `collector.processors` package, new strategies can be implemented to identify domain entities in different `ORMs`. If structural data need to be collected outside the scope of domain entities, a new collector strategy can also be implemented in the `collector.collectors` package, accompanied by a new processor created for it. The `collector.domain` can also be expanded with other elements as needed, such as `Method` or `Controller`, for example.

As mentioned in the previous chapter, a way to check if a domain entity references others by using them in methods would be based on collecting information on domain entity methods, including analyzing the parameters, body, and return type of methods. With the current extension points, the implementation of this task in the collector is facilitated.

### 5.1.2 Usage

Figure 5.3 shows the GUI presented to the user when the collector is running.



**Figure 5.3:** GUI for the structure collector.

Four fields must be provided by the user. With respect to the source code being analyzed, the name of the project and the location of its sources are requested. Additionally, the `ORM` framework used in the project should also be chosen from the available options. As of the writing of this thesis, the only

strategies implemented are for the Fenix Framework and Spring Data JPA. In the side of what data to collect, the only strategy currently implemented is for domain entity data, which includes the names of domain entities, their field names and types, and the type of their super-classes when they exist.

When the tool finishes collecting and serializing the relevant data, it outputs into a local `data/collection` folder. An example of how this output looks can be seen in the code snippet represented in Listing 5.1. This output can then be used as input in Mono2Micro to represent a code base, as described in the next section.

**Listing 5.1:** Example output for the structure collector.

---

```
1 {
2   "entities" : [
3     {
4       "name" : "Question",
5       "fields" : [{
6         "name" : "id",
7         "type" : {
8           "name" : "Integer"
9         }
10      }],
11     "name" : "title",
12     "type" : {
13       "name" : "String"
14     }
15   }, {
16     "name" : "creationDate",
17     "type" : {
18       "name" : "LocalDateTime"
19     }
20   }, {
21     "name" : "options",
22     "type" : {
23       "name" : "List",
24       "parameters" : [{
25         "name" : "Option"
26       }]
27     }
28   }
29 ],
30 "superclass" : null
31 }
32 }
```

---

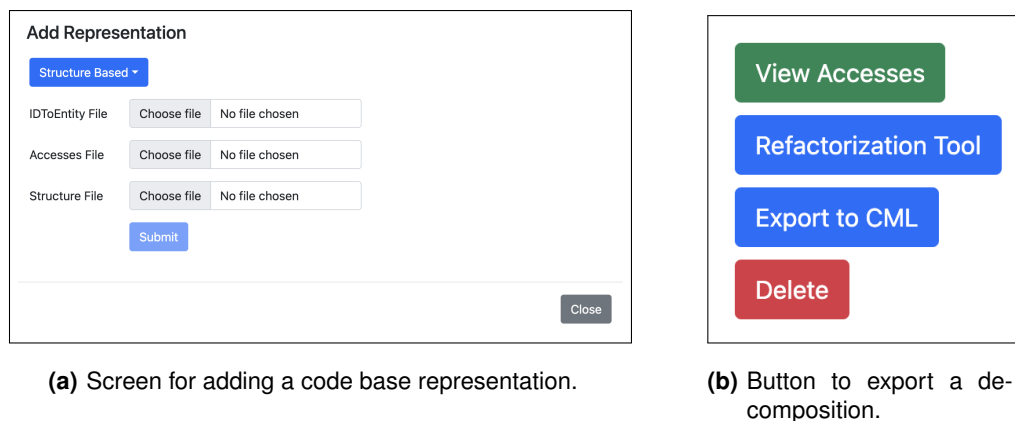


## 5.2 Mono2Micro Contract

In Mono2Micro, decompositions are represented and stored in the form of MongoDB<sup>1</sup> documents. As they are, these documents are not suitable for export directly into Context Mapper as they are prone to change and thus could break the translation process in Context Mapper. They also contain unnecessary data that is not used for mapping. To avoid these problems and further decouple the tools, the previous chapter defined the characteristics of a contract between tools, independent of the structure of the database document inside Mono2Micro, and containing only the necessary information to send.

Three sources of information were identified to generate the Mono2Micro contract. The first one is the structural data, which includes information on entity field names, types and superclass if it exists. The second one is the decomposition, which includes information on clusters and their entities. The third is the refactored functionalities, which include information on the sequences of accesses to entities per functionality. The first step to writing the contract is to retrieve these sources of information from within Mono2Micro.

In the case of structural data, since it constitutes as output from an outer collector, it can be persisted in Mono2Micro by using `representations`. A `Representation` is the result of a collector output and it is a form of monolith representation valuable for the downstream pipeline stages. Extending this abstraction with a `StructureRepresentation` is enough to ensure that the structural data is persisted when a user inputs it into the tool. Figure 5.4(a) shows the screen to add this new `StructureRepresentation`, along with information on accesses also needed for decomposition.



**Figure 5.4:** Mono2Micro interface additions to support contract generation.

The other two sources of information become available once a decomposition is generated, and functionality refactoring is applied to it.

The next step is to compose and export this data in the form of a JSON contract. This step is initiated

---

<sup>1</sup><https://www.mongodb.com/docs/manual/>

by the user when they decide to export a decomposition as seen in Figure 5.4(b).

The contract will be composed of three root elements, one for each of the information sources respectively, entities, clusters and functionalities. The internal format for entities was already shown in Listing 5.1. For clusters and functionalities, the format can be seen in Listing 5.2.

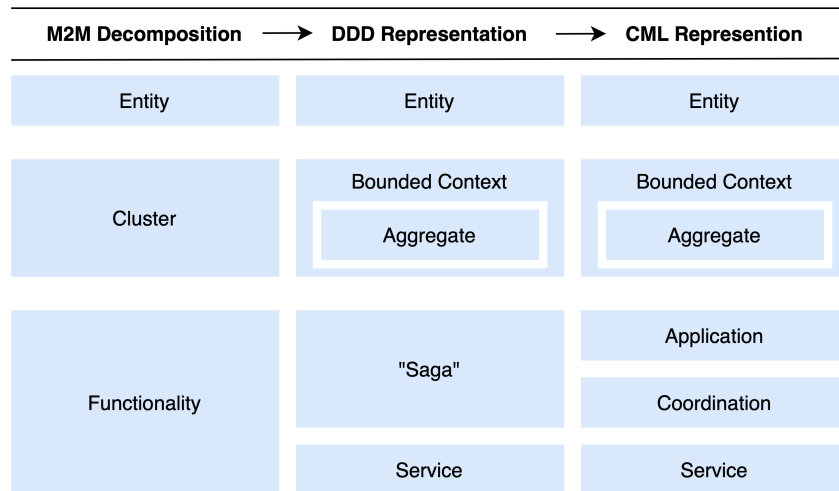
**Listing 5.2:** Segment of the Mono2Micro contract.

```
1 {
2   "name": "Quizzes Tutor",
3   "entities": [],
4   "clusters": [
5     {
6       "name": "Cluster1",
7       "elements": [
8         {"name": "Quiz"},
9         {"name": "QuizQuestion"},
10        {"name": "QuizAnswer"},
11        {"name": "Question"},
12        {"name": "QuestionOption"},
13      ]
14    }
15  ],
16  "functionalities": [
17    {
18      "name": "AnswerController.concludeQuiz",
19      "orchestrator": "Cluster1",
20      "steps": [
21        {
22          "cluster": "Cluster1"
23          "accesses": [
24            {
25              "entity": "QuizQuestion",
26              "type": "R"
27            },
28            {
29              "entity": "Quiz",
30              "type": "R"
31            },
32            {
33              "entity": "QuizAnswer",
34              "type": "RW"
35            }
36          ]
37        }
38      ]
39    }
40  ]
41 }
```

Taking into account the architectural requirements presented in the previous chapter, the functionality for the generation of the contract was designed and implemented in an extensible way, by providing abstractions for the contract definitions, ensuring that the current fields of the contract can be updated and new fields can be created, if needed.

## 5.3 Translation to CML

This section presents the concept mappings between Mono2Micro and Context Mapper based on the three mapping scenarios described in Section 4.2.2. Figure 5.5 shows a summary of the achieved mappings, which are discussed in the next paragraphs.



**Figure 5.5:** Mapping strategy of candidate decomposition concepts from Mono2Micro (M2M) to DDD and CML.

### 5.3.1 Entity Mapping

The entities of a candidate decomposition, by definition, are already based on the concept of *Entity* from DDD, which facilitates this mapping. The main difference is that Mono2Micro does not require the internal structure of entities to generate candidate decompositions, while in DDD and CML the internal state and relationships with other entities are relevant information to model an *Entity*.

To guarantee a more complete translation of candidate decomposition entities into CML, a new source code collector module was added to the Mono2Micro Collection stage, aptly named *Structure Collector* as shown in Figure 4.1. This module uses the Spoon Framework library [23] to analyze and collect structural information from entities in the monolith domain, including entity names, entity attributes, and relationships between entities, i.e. composition or inheritance.

### 5.3.2 Cluster Mapping

The main criteria that dictate how entities are clustered in the Mono2Micro Decomposition stage are based on transactional similarity. This means entities commonly accessed together (i.e. read/write) during the same transactions are more likely to belong in the same cluster. Similarly, a DDD *Aggregate*

is defined as a group of tightly coupled domain objects that can be seen as a unit for the purpose of data changes during transactions, which makes it a good fit to represent a cluster.

However, the concept of cluster also fits the concept of a DDD *Bounded Context*. This is because clusters define physical boundaries between microservices in a candidate decomposition and can be evaluated based on coupling with other clusters in the same decomposition. This dual mapping of the cluster concept could be achieved with different variations in the number of generated *Bounded Contexts* and *Aggregates*, but in the end the chosen mapping was to take each cluster and generate a corresponding *Bounded Context* and single *Aggregate* inside it, which in turn contains all the entities in the cluster.

Compared to other mapping combinations, this solution has several advantages. To start with, it satisfies all the given definitions of a cluster in Mono2Micro. In contrast, solutions such as mapping each cluster to an *Aggregate* and adding all *Aggregates* to a single *Bounded Context* (which would represent the whole decomposition) would remove the distributed nature on which the decomposition is based on; and mapping each cluster to a single *Bounded Context* while defining one *Aggregate* per *Entity* would completely ignore the transactional similarity that exists between entities on a cluster. This does not mean that the end product is to have one *Aggregate* per *Bounded Context*. It is important to remember that the generated CML code is by no means final and that further refactoring is expected by the architect doing the modeling. Starting from this initial mapping that satisfies the definition of a cluster, architects have the ability to further refine the model by partitioning the generated *Aggregate* of each *Bounded Context* using not only entity access information, but also the new structural context of entities that is not available in Mono2Micro.

On the subject of this structural context, there is also a caveat to take into account in this mapping. Entities that share explicit structural relationships in the monolith code are likely to end up in different clusters after decomposition. In the Mono2Micro graph representations, this is not a problem, since structural information is neither collected nor used. However, in the structure-driven environment of Context Mapper, it is necessary to ensure that the generated CML code does not contain *Entities* directly referencing other *Entities* in outer *Bounded Contexts*.

This problem is solved in two ways. First, when an *Entity* references an outer *Entity*, i.e. from an outer *Bounded Context*, the *Context Map* is updated with a relationship between *Bounded Contexts* in the direction of the referenced *Entity*. Second, this reference is replaced with a reference to a newly created local *Entity*, which represents the outer *Entity* locally. This is done so that the architect can better visualize which references in *Entities* need to be refactored, facilitating the modeling work. Figure 5.6 shows an example of an outer reference being resolved. A comment is also automatically generated to make these references stand out.

---

```

1 Aggregate Tournaments {
2
3   /* This entity was created to reference the 'Question' entity of the
4    * 'Questions' aggregate. */
5   Entity Question_Reference
6
7   Entity Topic {
8     String name
9     - Set<Question_Reference> questions
10  }
11 }

```

---

**Figure 5.6:** Generated CML example, representing an *Aggregate* that contains 2 *Entities*. Since *Topic* referenced an *Entity* in its fields not present in the *Aggregate*, *Question\_Reference* was generated locally to replace this reference.

### 5.3.3 Functionality Mapping

Functionalities are more challenging to represent in DDD since each functionality is composed of a sequence of read and write accesses to entities, which is a concept very particular to Mono2Micro and without apparent DDD equivalent concept. Additionally, the sequence of accesses that represents a functionality can be quite extensive. The reason for this is the fine-grained nature of the accesses collected from monolith code, due to their object-oriented design. This contrasts with the coarse-grained communication that is expected between microservices to avoid distribution communication costs. Without resolving this granularity issue, it becomes very impractical to represent functionalities compactly.

Fortunately, Mono2Micro provides a Functionality Refactoring tool that rewrites the functionalities of a candidate decomposition as Sagas [24, 25]. The tool converts several fine-grained microservice invocations into some coarse-grained ones, incorporating the Remote Façade pattern in the decomposition. Refactoring the functionalities using this tool has an extreme positive effect on the granularity of the access sequences, so the Functionality Refactoring module is represented in Figure 4.1 as a crucial step in the extended pipeline.

Refactoring functionalities as Sagas also makes a possible map to DDD more adequate. Although the Saga pattern is not a DDD pattern, in practice it can be used in conjunction with DDD to model distributed transactions [32]. Similarly, Context Mapper also supports constructs in addition to the original DDD patterns that enrich the modeling capabilities of the tool. For the case of functionality representation, the definitions of *Use Case* and *Event Flows* are the two most obvious constructs to use. However, they have some drawbacks. *Use Case* definitions would imply reverse engineering of the functionalities to a previous stage of the development cycle, losing in the process the information of the entity accesses and their Saga format. *Event Flows* seem more plausible to model Sagas, but they would lock the architect into using Event-Driven Design for all functionalities, which should be left for the architect to decide later case by case instead of enforcing it through the automatic translation to CML.

To supply a construct for the representation of Sagas meeting the current requirements, an expansion to the CML syntax was proposed and implemented in Context Mapper, which allows for the definition of distributed workflows without specifying the communication model of the process. For the current functionality mapping, this new concept can be used to simply state the steps of the Saga, without any implementing technology commitments.

This construct is called *Coordination*, and is based on the coordination property of Sagas that specifies whether the steps of a Saga are orchestrated or choreographed [32], and nothing more.

In CML, *Coordinations* can be used to coordinate defined *Service* operations, the same way a Saga coordinates steps. Figure 5.7 shows an example of the syntax in CML. Functionalities that do not access other *Bounded Contexts* are simply mapped to a *Service*, also defined in the *Application* layer of the *Bounded Context* where they are defined.

---

```

1 BoundedContext Tournament {
2   Application {
3     Coordination UpdateTournament_Coordination {
4       Tournament :: Tournament_Service :: updateTournament_step0;
5       Quiz :: Quiz_Service :: updateTournament_step1;
6       Tournament :: Tournament_Service :: updateTournament_step2;
7     }
8
9     Service Tournament_Service {
10      void updateTournament_step0
11      void updateTournament_step2
12      ...
13    }
14  }
15 }
16
17 BoundedContext Quiz {
18   Application {
19     Service Quiz_Service {
20      void updateTournament_step1
21      ...
22    }
23  }
24 }

```

---

**Figure 5.7:** *Coordination* construct in CML. The steps of the *Coordination* (4-6) represent ordered calls to *Service* operations (10,20,11).

To adhere to **R1**, it is important to place this construct in the correct place inside the CML syntax. Since *Coordinations* define a process, and assume communication between bounded contexts, placing this construct inside the domain goes against **DDD**, and thus breaks the cohesion of the language. Instead, *Coordinations* are defined within the *Application* layer of a *Bounded Context*. Other process-defining constructs such as *Flows* and *Events* are also defined within this construct, so it adheres to **R1**. An additional implication of this placement is that the operations mentioned within *Coordinations* must also be defined inside *Applications*, by using the available *Service* constructs.

Alongside the keyword, a `Coordination` must also have a name that refers to the process or functionality it represents. Regarding the body, a `Coordination` is composed of a set of ordered steps, where each step represents an invocation to a service operation in a bounded context. These steps are referred to in the syntax as `CoordinationSteps`, and are composed of references to operations defined within `Services`. These `Services` can belong to the current bounded context, or other ones. To reference a `Service` operation, a coordination step is divided into three segments, separated by the `::` symbol: The name of the Bounded Context where the operation is defined; the name of the application `Service` where the operation is defined; and the name of the operation.

Since a `CoordinationStep` is composed of references to other constructs, the following verification is done to avoid ambiguity when pinpointing the operation referenced in the step:

- All references within a step must refer to existing rules in the written CML code;
- A reference to an outer bounded context must be reachable by Context Map relationships;
- A reference to a service must originate from an Application Layer Service;
- A reference to an operation should be unique within the service.

These validation rules were enforced by extending the available validation package present in Context Mapper with new syntax checks for `Coordinations`.

In addition to this, the Generators module of Context Mapper was also extended to provide the new `Coordination` rule with a mapping to BPMN, that can be opened and edited using Sketch Miner<sup>2</sup> in a browser if the architect deems it necessary. This is achieved by leveraging the abstractions of the Sketch Miner tool that are present in Context Mapper.

By implementing this functionality in Context Mapper, every written `Coordination` can now be open in Sketch Miner. Figure 5.8 shows the steps required to do this.

## 5.4 Discovery Strategies

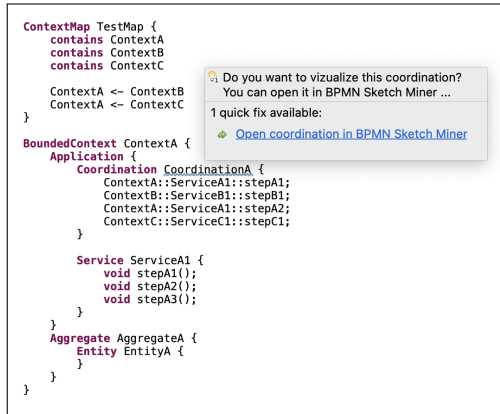
Some points were already made in Section 4.2.1 in regards to how the Discovery Library (DL) uses an intermediate model to facilitate translations to CML. This means that the implementation of new discovery strategies should focus on translating the Mono2Micro contract to the internal DL model.

To do so, the module provides a set of extension points for translating to this internal model. These are implemented as class abstractions and can be used to create new discovery strategies.

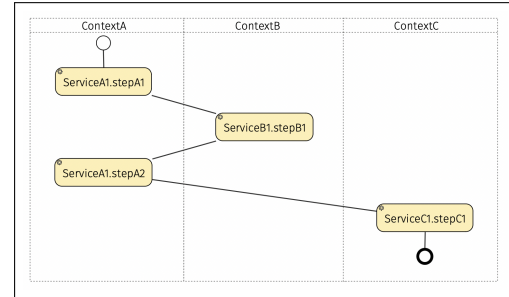
Two new Mono2Micro discovery strategies were implemented to realize the mappings defined in the previous section: One to discover `Bounded Contexts` and everything contained in them; and one to

---

<sup>2</sup><https://www.bpmn-sketch-miner.ai>



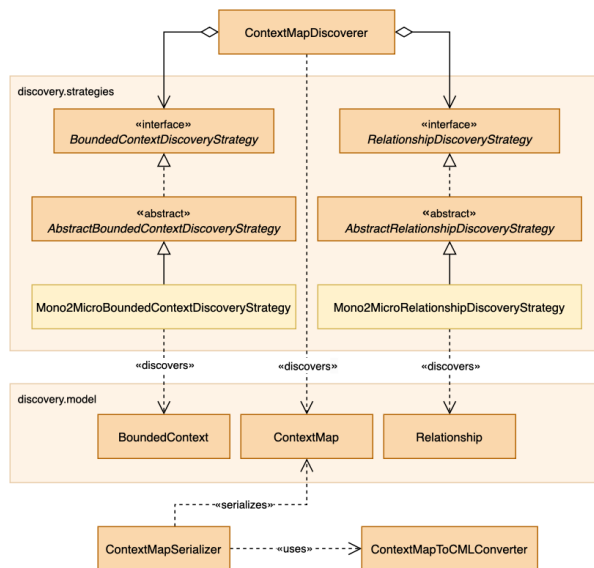
(a) Tool-tip in the Eclipse IDE.



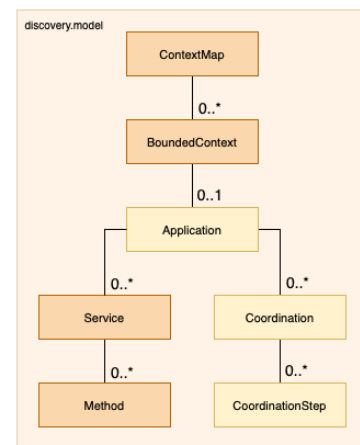
(b) BPMN representation of a Coordination.

Figure 5.8: Opening a Coordination in Sketch Miner.

discover relationships between Bounded Contexts. Figure 5.9(a) shows these two strategies in yellow, while the rest of the class diagram in orange represents the available API of the DL.



(a) Discovery Library strategy extension.



(b) Discovery Library model extension.

Figure 5.9: Class diagrams of the Discovery Library extensions.

The new strategies receive the contract as input. The elements on the contract are then converted into Java objects so that they can be more easily mapped to the internal DL model.

Figure 5.9(b) shows the relevant domain classes that make up this model. Like before, yellow classes represent extensions added to represent all the new mappings, and orange classes represent the already existing model. Since this model conforms to CML, all additions also directly represent CML



constructs, respecting **R1**.

It is also worth noting that, before the extension, the CML constructs represented in the internal model were the minimum necessary to support the already existing discovery strategies. For example, the Application construct had no prior representation in the internal model because up to now no discovery strategy provided a mapping of it.

The `Mono2MicroBoundedContextDiscoveryStrategy` works in three stages. It starts by discovering `BoundedContexts` from clusters, also adding to each `BoundedContext` an `Aggregate` and an `Application`. When creating aggregates, it also discovers and creates `DomainObjects` from the entities of each cluster.

Once all `BoundedContexts` and `DomainObjects` have been created, the strategy enters stage two and starts to discover the `Coordinations` of each `BoundedContext` based on the contract functionalities. Each functionality maps to a `Coordination`, and each of the steps of a functionality maps to a `CoordinationStep`. Since a `CoordinationStep` references a `BoundedContext`, `Service` and `Method`, all these need to be previously created.

The final step is to update the created `DomainObjects` with the structural information of the contract entities tag. If during this process a type reference to a `DomainObject` that is in another `BoundedContext` is found, the strategy creates a new `DomainObject` in the current `BoundedContext`, and replaces the outer type reference with a reference to this new `DomainObject`. At this stage, a comment is also generated informing the architect of this translation decision.

The `Mono2MicroRelationshipDiscoveryStrategy` is much simpler in that it only serves the purpose of populating the relationships between the found `BoundedContext` constructs. This is done by looking at the functionalities tag in the contract, and finding references to other `BoundedContexts` in the steps of each functionality. These references are then translated to a *Relationship* construct.

After the mappings to the internal model are complete, the `ContextMapperSerializer` will call the `ContextMapToCMLConverter` class, where all the functionality for translating the DL model to CML model is placed. This will result in the output of valid CML code, that can then make use of the rest of the features available in Context Mapper. Listing 5.3 shows the client code that needs to be run to translate the contract to CML.

**Listing 5.3:** Client code to translate a Mono2Micro contract in the Discovery Library.

---

```
1 ContextMapDiscoverer discoverer = new ContextMapDiscoverer()
2   .usingBoundedContextDiscoveryStrategies(
3     new Mono2MicroBoundedContextDiscoveryStrategy(new File(sourcesPath), namingMode)
4   ) .usingRelationshipDiscoveryStrategies(
5     new Mono2MicroRelationshipDiscoveryStrategy(new File(sourcesPath))
6   );
7
8 ContextMap contextmap = discoverer.discoverContextMap();
```

---

It is also in these strategies that the naming heuristics are implemented. Each heuristic is implemented as a strategy class. The Strategy pattern was used to provide abstraction and extensibility to

the code for the development of new naming heuristics.

# 6

## Evaluation

### Contents

---

6.1 Architectural Evaluation . . . . .	53
6.2 Case Study . . . . .	58
6.3 Discussion . . . . .	60

---



The evaluation of the work is divided into two sections. In the first section, an evaluation based on the defined architectural requirements is performed, which includes measuring the modularity, extensibility, and interoperability of the implemented solution pipeline. In the second section, the solution is applied to a case study, and the results are discussed.

## 6.1 Architectural Evaluation

### 6.1.1 Modularity

In the context of the solution pipeline, modularity will be measured by looking at the level of cohesion inside each module and the level of coupling between modules. This corresponds to evaluating the presence of **R1** and **R2** in the solution, respectively.

Starting with module cohesion, this measure dictates whether a module is compromised or not by additions that do not conform to its responsibility.

Unlike other modules, the structure collector is built from scratch and has the sole purpose of collecting structural information on source code domain entities. This means that the internal model was also defined from the ground up, so the only clash the model can have is with its dependent tool, the Spoon Framework. This is also not the case, since the internal model of the structure collector was designed taking into account that it would extend the functionalities of the Spoon Framework.

The contract generation module in Mono2Micro is made up of the new `Representation` for structural information, and a new feature in the decomposition API to export a decomposition. The new `Representation` is implemented with the same rules and conventions as the other existing `Representations`, so it keeps its own responsibility while confirming with the more generic responsibility associated with the `Representation` concept. As for the export decomposition feature, it was placed in the context of a decomposition, according to the pipeline separation of responsibilities. Only parsing utilities were placed in their own utility class as to not muddy the internal cohesion of the decomposition.

Next, in the Discovery Library, two new discovery strategies were implemented. Both these strategies also follow the module responsibilities by translating the Mono2Micro contract to the internal model, and then calling the provided API to serialize the model into `CML`.

Finally, in regards to the `CML` extension for the `Coordination` construct, cohesion was also preserved. In addition to following implementation conventions, the concept itself also fits the internal model of `CML`, since it can represent generic processes in the application layer.

Moving on to coupling, the evaluation here will be based on understanding the level of coupling each module has to outer modules, be it modules from the pipeline or other dependencies.

The structure collector, like the other collectors in the Collection Stage of Mono2Micro, is decoupled from the main processes of the tool due to the variety of collection strategies that exist. To join the

collectors with Mono2Micro, each collector has a contract in the form of a `Representation` class in the main tool. This is a coupling point, but one that is very loose and also necessary. More particularly to the structure collector, this module has a very high coupling with the Spoon framework it uses, but since the Spoon model is part of the collectors model, this coupling is healthy and shows that internally the module has cohesion.

Since `Representations` were used to implement the contractor generator, extra coupling was introduced in this section of the pipeline. However, the export functionality needs to access three separate data sources to build the contract, with the possibility of adding more in the future. This makes the functionality more volatile, since a change in the data model has a higher likelihood of breaking the data source retrieval and translation to the contract. Fortunately, the contract ensures that this volatility, which had already been mentioned as a characteristic of Mono2Micro, stays in Mono2Micro and does not propagate to Context Mapper.

The strategies implemented in the Discovery Library are only coupled to the contract format that they are responsible for translating. This is an example of the Strategy Design Pattern being used to decouple the code. Each strategy has its own translation code.

### 6.1.2 Extensibility

Extensibility will be measured on the basis of the ability to extend the implemented modules with new functionality and how well they adhere to the Open-Closed Principle. **R3** and **R4** are within the scope of this evaluation.

Starting with the Structure Collector, this module design takes into account future plans to support the collection of new types of structural information from the monolith. With that in mind, and looking back at Figure 5.2, two main extension points were provided in the form of abstractions.

The first is related to the `Processor` strategies. If a new type of `ORM` needs to be supported in the future, one can simply subclass the existing `DomainEntityProcessor` abstract class and define the `ORM` specific concept mapping.

The second extension point refers to the collection strategy. As it stands, the only implemented collection strategy uses only domain entity classes to inform its collection. However, if the information is not strictly in domain entity classes, the tool design supports the addition of new collector strategies by sub-classing the `AbstractCollector` and the `AbstractProcessor` classes.

These extensions also adhere with the Open-Closed Principle, since the explained strategies are open for extension, but closed for modification.

Moving on to contract generation, this module used the `Representation` abstraction to extend the existing tool, maintaining the extensibility of this portion of the code. In terms of contract generation, the fact that the contract is loose makes it easier for extensions to be added if needed, as long as they do

**Table 6.1:** Mono2MicroBoundedContextDiscoveryStrategy tests.

Test Name	Outcome	Coverage
canDiscoverBoundedContextsFromClusters	Success	100%
canDiscoverAggregatesFromClusters	Success	100%
canDiscoverDomainObjectsFromClusterElements	Success	100%
canDiscoverCoordinations	Success	100%
emptyResultIfM2MContractHasInvalidFormat	Success	100%

not break interoperability between tools.

### 6.1.3 Interoperability

To measure interoperability, several integration tests were written for the various sections of the pipeline that require inter-module communication. This includes the translation of the Mono2Micro contract into CML and the translation of CML into other diagram formats. The results of this evaluation will show how the solution pipeline followed **R5** and **R6**.

Starting with the communication between Mono2Micro and Context Mapper, this communication is defined by the JSON contract that exists between tools. This contract is interpreted in the DL of Context Mapper, so several integration tests were written in this module to evaluate how well both tools inter-operate, i.e. the capacity of the discovery strategies to understand the incoming information from Mono2Micro.

Based on the internal test structure of the DL, the tests were divided into the following groups: discovery tests and serialization tests. Discovery tests measure the capacity of the strategies implemented to translate the incoming information into the internal model of the DL. Serialization tests measure the ability of this internal model to be translated into valid CML. This separation of tests also adds a level of maintainability to them.

For both groups, the Junit<sup>1</sup> Java library was used to write the tests, and a contract was used with the results of a decomposition of the Quizzes-Tutor<sup>2</sup> application. This contract has more than 5000 lines of data, including 7 clusters, 42 entities, 22 functionalities, and 198 sequences of accesses.

The discovery tests for the two strategies were implemented in respective test classes, one for each strategy. Table 6.1 and Table 6.2 show these tests by name, outcome, and the overall strategy code coverage. In terms of serialization tests, Table 6.3 shows the implemented test.

<sup>1</sup><https://junit.org/junit5/>

<sup>2</sup><https://github.com/socialsoftware/quizzes-tutor>

**Table 6.2:** Mono2MicroelationshipDiscoveryStrategy tests.

Test Name	Outcome	Coverage
canDiscoverRelationship	Success	100%
throwExceptionIfFileDoesNotExist	Success	100%

**Table 6.3:** Serialization tests of contract-based internal model.

Test Name	Outcome	Coverage
canSaveDiscoveredModelAsCMLFileM2M	Success	100%

**Table 6.4:** SketchMinerCoordinationModelCreatorTest class tests.

Test Name	Outcome	Coverage
canGenerateSimpleSequenceWithTwoActors	Success	100%
canGenerateSimpleSequenceWithThreeActors	Success	100%

**Table 6.5:** SketchMinerGeneratorTest class tests.

Test Name	Outcome	Coverage
canGenerateFilesForBoundedContextCoordinations	Success	100%
cannotGenerateFileIfNoCoordinationExists	Success	100%



**Table 6.6:** SketchMinerLinkCreatorTest class tests.

Test Name	Outcome	Coverage
canCreateSketchMinerLinkForCoordinationModelUni	Success	100%
canCreateSketchMinerLinkWithCoordinationEObjectI	Success	100%

**Table 6.7:** ApplicationLayerValidationTest class tests.

Test Name	Outcome	Coverage
canOfferSketchMinerLinkInCoordination	Success	100%

Moving on to the interoperability between the CML Core module and the Generators module, several tests were written for the Sketch Miner<sup>3</sup> generator. This generator is used to create BPMN diagrams of CML constructs that are structurally similar to processes, as is the case of the new `Coordination` rule. Contrary to other generators, the implementation of Sketch Miner also allows an architect to open process-like CML constructs in a web browser, meaning that interoperability of this process also needs to be tested.

Several tests were written for the different stages of generating BPMN diagrams to ensure the translation process is stable. These are shown in Table 6.4, Table 6.5, Table 6.6 and Table 6.7, and are divided as follows respectively: Tests for the translation of CML to the Sketch Miner model; tests for the generation of local BPMN files; tests for generating the URL to open Sketch Miner in a browser; and tests for confirming that a URL is available for the architect to see.

---

<sup>3</sup><https://www.bpmn-sketch-miner.ai>

**Table 6.8:** Candidate decomposition measures for the QT case study.

Cluster	Entities	Functionalities	Cohesion	Coupling	Complexity
Cluster0	6	7	0.81	0.185	787.571
Cluster1	27	107	0.212	0.657	106.832
Cluster2	4	11	0.727	0.179	431.091
Cluster3	9	35	0.654	0.753	322.486

## 6.2 Case Study

Quizzes-Tutor (QT)<sup>4</sup> is an online quizzes management application developed for educational institutions. It can be used to create, manage, and evaluate quizzes composed of varying types of question formats. Teachers can add questions related to topics of the courses they preside over, while students can answer these questions within quizzes. Other functionalities include the creation of quiz tournaments between students, question proposals from students, and ways to discuss question answers. This real-world monolith, composed of 46 domain entities and 107 functionalities, was used as a case study to validate the Mono2Micro pipeline extension, which provides DDD modeling capabilities.

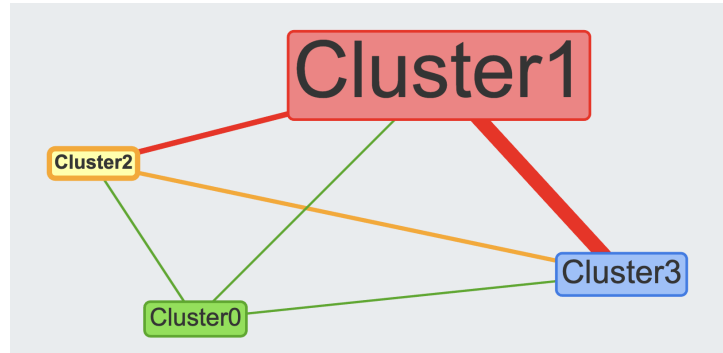
Since QT is actively extended with new modules, it is easy to introduce ad hoc dependencies, and changes to some functionalities start to more easily break other unrelated parts of the application. This modular incremental development is the ideal context for microservices because it will enforce a border between the modules. It is more difficult to break modularity due to the distributed context of the module interfaces. Besides introducing modularity, performance is also an aspect to consider. Parts of the system would benefit from independent scaling, like answer submission during live quiz sessions. Microservice architecture would help in this regard.

### 6.2.1 Decomposition Generation

To start the validation, a candidate decomposition for the QT application must be generated and chosen. To this end, around 2000 candidate decompositions were generated with different values of similarity criteria and the number of clusters. Candidate decompositions were then filtered on the basis of the values of their measures. The heuristic used was to order decompositions based on their coupling value in ascending order, and then based on their cohesion value in descending order, to prioritize decompositions with low coupling and high cohesion. Of the top 100 results, the candidate decomposition with the lowest complexity value was chosen. Data for this candidate decomposition can be seen in Table 6.8. Figure 6.1 also shows the clusters view of the decomposition shown in Mono2Micro.

Table 6.8 shows two noteworthy pieces of information. To start with, the complexity of each cluster is very high. The complexity measure represents the migration cost of the functionalities in a cluster. This migration cost is measured as the cost of re-designing from an ACID context to a distributed one.

<sup>4</sup><https://quizzes-tutor.tecnico.ulisboa.pt/>



**Figure 6.1:** Mono2Micro decomposition visualization with fine-grained interaction between clusters. Edges represent functionalities shared between clusters.

**Table 6.9:** Refactored functionalities for QT case study. CGI stands for Coarse-Grained Interaction, and FGI stands for Fine-Grained Interactions.

Name	#Clusters	CGI	FGI	Reduction%
concludeQuiz	3	4	73	94.52
getQuizByCode	3	4	33	87.88
getQuizAnswers	4	8	84	90.48
exportCourseExecutionInfo	4	9	110	91.82
importAll	3	5	119	95.8
createQuestion	2	3	24	87.5
getQuizAnswers	4	8	92	91.30

Of the initial 107 functionalities, 31 involve distributed calls composed of several hops between clusters that drive the complexity high. This is because functionalities are still represented by the fine-grained monolith interactions between entities that can now be in different clusters. To reduce this complexity, the *Functionality Refactoring* tool represented in Figure 4.1 is used to create coarse-grained interactions between clusters. Table 6.9 shows the reduction of invocations for some of the QT functionalities. Applying this complexity reduction also makes it viable for functionalities to be represented in a structural language such as CML. Otherwise, any translation strategy would culminate in thousands of operation definitions for just a subset of the functionalities, as the FGI values show in Table 6.9.

The other noteworthy piece of information is the high number of entities inside Cluster1 compared to the other clusters, which means that the entities inside this cluster are more entangled when it comes to the functionalities that use them, and are more difficult to separate without creating an overly complex decomposition. It is also the reason for the non-optimal levels of cohesion and coupling in this cluster. At this stage, when some manual refactoring of functionalities is needed, modeling using DDD can help.

The candidate decomposition is translated into CML by the discovery strategies, which outputs a .cml file with a representation of the candidate decomposition. Figure 6.2 shows a modified snippet of the generated CML, related to the ConcludeQuiz functionality of a decomposition. Without any opti-

**Table 6.10:** Generated CML constructs. The number of services is represented by four values: No heuristics used; *Full Access Trace* used; *Ignore Access Types* used; and *Ignore Access Order* used. The number of entities is represented by two values: original entities and reference entities. The most accessed entity is based on external accesses to the *Bounded Context*.

Cluster	#Services	#Entities	Most Accessed Entity
Cluster0	15/7/6/4	6/4	QuizAnswerItem (35.14%)
Cluster1	59/53/52/34	27/3	Quiz (12.2%)
Cluster2	11/5/5/2	4/0	QuestionAnswerItem (30.0%)
Cluster3	36/22/21/8	9/4	QuestionDetails/Image (16.46%)

mization, the translation generates a total of 121 operations, used by 31 Coordinations that represent the distributed functionalities. With naming heuristics, the number of service calls can be reduced to 87 using *Full Access Trace*, to 84 using the *Ignore Access Types*, and to 48 using the *Ignore Access Order*. Table 6.10 shows the reduction of generated services according to which heuristics are used per cluster.

Regarding entity generation, a total of 11 reference entities were generated to signal structural dependencies between entities, also shown in Table 6.10, and every entity is generated with information on the number of accesses to it, from the total *Bounded Context* accesses (external and local), as shown in Figure 6.2. This information can help to refactor the decomposition further. For example, in Cluster3, there are 2 dominant entities out of 9 in terms of external accesses: *QuestionDetails* and *Image*. These make them good possible candidates to serve as *Aggregate Roots* to the *Bounded Context*. A look at their structural relationships reveals that *QuestionDetails* also contains 3 subclass entities in the same cluster: *MultipleChoiceQuestion*, *CodeOrderQuestion* and *CodeFillInQuestion*; which in turn are structurally related to: *Option*, *CodeOrderSlot*, *CodeFillInSlot* and *CodeFillInOption*. *Image* is not structurally related to these, but its presence in this *Bounded Context* means its commonly accessed together with the other entities. A possible first refactoring of this cluster could then be the separation of the Cluster3 *Aggregate* into 2 *Aggregates*, which is an automatic architectural refactor supported by CML. The first *Aggregate* would control question types, formats, and related invariants with *QuestionDetails* as its *Aggregate Root*. The second *Aggregate* would represent a repository of images and be controlled through its *Image Aggregate Root*.

## 6.3 Discussion

This section discusses the findings of applying the DDD-based extension to the operational pipeline of Mono2Micro by analyzing how the implemented solution and the results of its application in the case study answer the research questions raised in the introduction of this paper.

### 6.3.1 Results Validation

Starting with the first research question (RQ1), to understand whether the Mono2Micro operational pipeline can be extended to integrate DDD, through the use of CML, the first step taken was to compare the architectural characteristics of each tool.

Mono2Micro follows a pipe-and-filter architecture with its stages, and provides extension points in the form of abstractions for each stage; while Context Mapper follows a hub-and-spoke architecture, and also provides extension points in its core, the CML language, and each of its spoke modules, including the Discovery Library (DL). As described in the solution, these characteristics show that the tools emphasize modularity and extensibility, which made the development of an integration strategy that follows the same requirements possible. By measuring the level of modularity and extensibility of the solution, RQ1 can be evaluated.

First, modularity deals with how divided a system is into logical modules that encapsulate specific self-contained functionality, improving separation of concerns and internal cohesion. The solution is composed of two new modules in Mono2Micro, the *Structure Collector* and *CML Translator*. In terms of cohesion, both modules respect the pipeline architecture of Mono2Micro, and are placed accordingly inside it based on their responsibilities. Regarding coupling, Figure 4.1 also shows the dependencies of each model. The *Structure Collector* is highly decoupled, depending only on the monolith source code. It does not know about the subsequent stages of the pipeline. The *CML Translator* has a contract between it and the Mono2Micro model, ensuring that it is only coupled to one source.

Second, extensibility deals with how open for extension the features of a system are without putting at risk their core structure, improving the addition of new functionality. The *Structure Collector* was designed from scratch. It provides abstractions for the collection of data from new frameworks and the collection of other types of structural data. The CML Translator is an extension of the DL API, so it follows that the discovery strategies implemented follow the same design and are also open to extension by providing abstractions. Additionally, the contract generation is also built with abstractions if new information is needed for future discovery strategies.

Moving on to the second research question (RQ2), the mappings of the cluster, entity, and functionality concepts to DDD demonstrate how a candidate decomposition based on entity accesses can be represented with DDD concepts. Mono2Micro entities are already based on the concept of DDD *Entities*, so the mapping is consistent in this regard. Further more, to improve usability, structural information about entities is collected for a more complete representation in CML. In the case of clusters, consistency was maintained by mapping each of them to a *Bounded Context* and an *Aggregate*. The structural context of the CML was also considered. To avoid having the cluster entities break the bounds of outer *Bounded Contexts* with direct references, reference entities were added. This improves usability by highlighting where refactoring needs to be applied to entity relationships. For the mapping of function-

alities, the sequence of accesses to entities that composed them was first converted into a structured Saga. This significantly reduced the complexity of the sequence in terms of size and hops between clusters, and made it simpler to represent with DDD, since Saga is a well-known pattern in distributed systems. Sagas were mapped to *Coordinations* in CML, which encode an ordered sequence of service calls, just as Sagas encode a sequence of steps. To improve the usability of this mapping, hints to which entities are accessed in each service call were generated in the names of each service call.

Finally, in regard to the third research question (RQ3), analyzing and comparing the decomposition artifacts represented in each tool, as demonstrated in the case study, can show how an architect can benefit from the use of this extension.

In the case of entity representation, structural information is now available in the CML representation. This information is novel in regard to the old pipeline, as Mono2Micro representations can at most show the names of entities within each cluster, not its internal state or structural connections, as illustrated in the QT decomposition in Figure 6.1. With the CML representation of entities, it is possible to observe the attributes of each entity and also the structural refactorings that must be made in existing entities based on reference entities to other Bounded Contexts, demonstrated in Figure 6.2.

In the case of clusters, *Aggregates* can now be defined and used to further partition a cluster and its entities based on access patterns, access percentages, and the structural information provided at generation time in the form of commented constructs.

In the case of functionalities, the architect now has the option of editing their Saga representation in CML, by editing the generated *Coordinations*. Mono2Micro only allowed for the visualization of fine-grained functionality traces in a graph representation, and the Functionality Refactoring module of the tool only produced the data for the Sagas, without any way to edit or visualize them in a graphical representation. Using CML, these functionalities can be modeled as *Coordinations* and edited in the language with the added context of the internal structure of the *Bounded Contexts* they interact with. Furthermore, *Coordinations* can be visualized in BPMN using a prompt to generate the diagram in BPMN Sketch Miner<sup>5</sup>. Figures 6.3 and 6.4 show the comparison of the views of a functionality that interacts with three of the four clusters before and after integration with CML.

Additionally, the service naming heuristics allow the architect to reduce the number of generated service calls that exist in each cluster. This does not reduce the number of functionality steps but increases the level of reuse of services.

---

<sup>5</sup><https://www.bpmn-sketch-miner.ai>

---

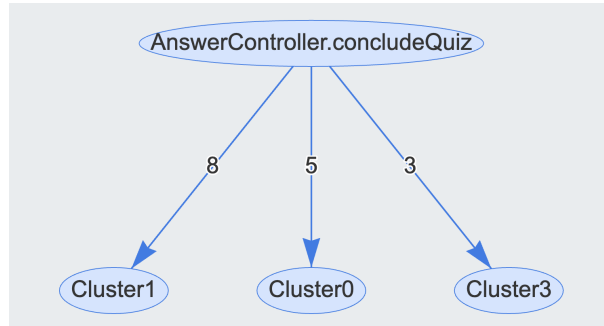
```

1 BoundedContext Cluster3 {
2   Application {
3     Coordination ConcludeQuiz_Coordination {
4       Cluster3 :: Cluster3_Service :: acQuestionDetails_acOption;
5       Cluster1 :: Cluster1_Service :: acQuiz_acQuizAnswer_acQuestion;
6       Cluster0 :: Cluster0_Service :: acAnswerDetails;
7       Cluster1 :: Cluster1_Service :: acStudent_acDashboard;
8     }
9   }
10  Service Cluster3_Service {
11    void acQuestionDetails_acOption;
12    ...
13  }
14 }
15 Aggregate Cluster3 {
16   /*
17    * Metrics:
18    * - Percentage of external accesses: 16.46% (13/79)
19    * - Percentage of local accesses: 16.41% (21/128) */
20   Entity QuestionDetails {
21     Integer id
22     - Question_Reference question
23   }
24
25   /* This entity was created to reference the 'Question' entity of the
26    * 'Cluster1' aggregate. */
27   Entity Question_Reference
28   ...
29 }
30 }
31
32 BoundedContext Cluster1 {
33   Application {
34     Service Cluster1_Service {
35       void acQuiz_acQuizAnswer_acQuestion;
36       void acStudent_acDashboard;
37       ...
38     }
39   }
40   Aggregate Cluster1 {
41     /*
42     * Metrics:
43     * - Percentage of external accesses: 10.06% (33/328)
44     * - Percentage of local accesses: 8.37% (41/490) */
45     Entity Question {
46       Integer id
47       LocalDateTime creationDate
48       Integer numberOfAnswers
49       Status status
50       String title
51       String content
52       - Course course
53       - Set<Topic> topics
54       - Set<Discussion> discussions
55       ...
56     }
57     ...
58   }
59 }
60
61 BoundedContext Cluster0 {
62   Application {
63     Service Cluster0_Service {
64       void acAnswerDetails;
65       ...
66     }
67   }
68 }

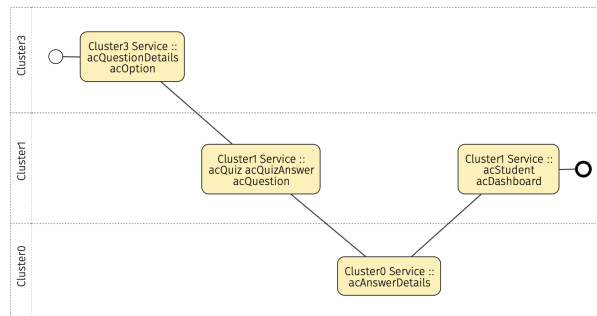
```

---

**Figure 6.2:** Snippet of the generated CML related to the functionality `ConcludeQuiz`. Triple dots (...) represent omitted constructs for the purpose of the example. Service operation names were also truncated.



**Figure 6.3:** Graph representation of the `ConcludeQuiz` functionality in Mono2Micro, and the clusters that participate in it. Edge numbers represent the number of accessed entities in the clusters they point to.



**Figure 6.4:** BPMN representation of the `ConcludeQuiz` *Coordination* in Context Mapper, generated using BPMN Sketch Miner. The tasks in the diagram represent the steps of the *Coordination*, and each participant (lane) represents the *Bounded Context* where the step is defined.



# 7

## Conclusion

### Contents

---

7.1 Threats to Validity . . . . .	67
7.2 Future Work . . . . .	67
7.3 Conclusions . . . . .	69

---



## 7.1 Threats to Validity

With respect to internal validity, there are two points worth considering. First, the functionalities used in the decomposition and CML mapping process are all linear in nature. This is due to the existent static entity access collection tool in Mono2Micro, which flattens code branches into a single access sequence in a depth-first fashion. However, previous research that used the same sequences to develop the Saga representation of functionalities as showed this has little impact on the final results [25], and support for multiple traces per functionality is being developed. By extension, the current implementation of *Coordinations* in CML is also based on linear Sagas. However, *Coordinations* can be opened in BPMN Sketch Miner, where they can be further edited with more complex workflow logic. Context Mapper also supports branching processes in its *Event Flows*, so the concept could be expanded to *Coordinations* as well in the future.

Second, the provided solution for resolving service operation names with access traces can sometimes generate verbose names, but these are meant to be temporary and only inform on what a certain step is doing, which is more appropriate and informative than the alternative of generating incremental "step" names. Future work could expand on the level of collected information to create heuristics based on method names to reduced verbosity.

In terms of external validity, the current implementation assumes the use of Java and the Spring Boot JPA Framework to collect entity access and structure information, but the process is general enough to be applicable to other programming languages and frameworks. The modules that assume these limitations are also built with abstractions for the implementation of other technologies.

The implementation of the solution is also realized by using Mono2Micro and Context Mapper, but this does not mean that a more general solution cannot be derived for other tools. Mono2Micro itself is designed as a generalization of the state of the art in microservice identification [2]. As for Context Mapper, the tool is built on top of the concepts and patterns of tactical and strategic DDD. If other tools follow representations of the same patterns, they can be used by the same mapping strategies applied to Context Mapper.

## 7.2 Future Work

The process of developing this solution pipeline and viewing the final results of it left many possible improvements in the air that could be tackled in future work. These can be broken down into improvements to the CML representations and improvements to the Mono2Micro tool itself. Before explaining each one, a short enumeration of these identified points of interest for future work follows. Regarding CML improvements:

- The addition of other decomposition representations in the mapping to CML;
- The refactoring of Mono2Micro to represent entity accesses using create/read/update/delete types instead of the current read/write representation;
- The expansion of the concept of `Coordination` in CML to include additional information on the nature of its steps, such as their synchronicity or consistency;

Regarding Mono2Micro improvements:

- The addition of a structure-based decomposition strategy to Mono2Micro based on the new structural representation of the monolith;
- Investigating whether duplicating certain entities, such that they are present in more than one cluster, can reduce the overall complexity of the migration and its distributed transactions.

Starting with the CML improvements, and as mentioned in the previous sections, the addition of more Mono2Micro decomposition representations could lead to the identification of new ways to improve both tools, as was the case in the work done here. Due to the process of trying to map functionalities in CML, a lack of a construct for representing simple inter-context processes was identified since the available mappings were either `Use Case` constructs, which were not fit to represent `SAGAs` without sacrificing information, or `Flows`, which would enforce the definition of an `Event` and `Command` pair for each functionality step, making the mapping much more verbose and coupling it with event-driven design.

An example of the next representation to support is the code authorship and commit history representation. Context Mapper provides different ways to interpret its `Bounded Contexts`, and one of them is team-based, with the addition of a `type = TEAM` field in the `Bounded Context` definition. This means that decompositions whose clusters are based on team size reduction could be mapped to team-based `Bounded Context`.

Using CRUD accesses in another way to improve the value of the information provided to CML without adding size complexity to the contract. Together with the naming heuristic, it would give the architect even more information on what certain steps of functionalities are doing.

Next, this suggestion for future work is the continuous development of the new `Coordination` rule of in CML. This may include an addition to it to specify in more detail properties of each of the steps. As it is now, it constitutes a minimum viable product, so there is potential here to transform the rule further. An example of this would be to provide three fields inside `Coordinations` that specify if the construct is: Orchestrated or choreographed; atomic or eventually consistent; and synchronous or asynchronous.

Moving on to improvements specific to Mono2Micro, since a consequence of implementing this pipeline is the addition of structural representations of a code base in Mono2Micro, there is future work here to expand decomposition strategies to take into account structural data. Just like other strategies,

the new structural-based representation could be mixed with other representation, and new weights such as afferent and efferent coupling between entities could be defined in Mono2Micro.

Finally, as discussed in the previous section, multiple references to the same entity in different clusters give the impression that the entity in question could be a central part of all the processes of the system. Since the context is distributed, there is a tendency for duplication to be preferable to code re-usability to avoid congesting the network unnecessarily [32]. One conclusion taken from the evaluation of this occurrence was that the more fine-grained the clusters, the less likely any entity would be referenced in multiple clusters. Therefore, whether Mono2Micro can gain anything from considering duplication is left to future work.

## 7.3 Conclusions

A significant amount of research has been done on the migration of monoliths into a microservice architecture, but almost no tools incorporate mappings to DDD concepts in their migration processes. Research showed a trend for developing DSLs to represent DDD concepts and adapting the concepts to work in other diagramming tools to develop microservices, but to our knowledge never directly in a migration tool.

Practitioners and software architects who need to migrate their current monolith architectures due to economic reasons, scalability, or more autonomous development teams would benefit from a solution that not only proposes candidate decompositions, but also automatically generated design-level DDD artifacts from that decomposition, as a starting point to facilitate the refactoring process towards a microservice architecture.

This paper proposes a solution pipeline for this problem composed of the integration of Context Mapper, a modeling framework that provides a DSL to represent DDD patterns, into the Mono2Micro decomposition pipeline, a robust microservice identification tool.

The proposed solution achieves the integration of both tools by defining a mapping of concepts between tools, whilst respecting each of the tool models. To support this mapping, the solution includes several new modules and modifications to the tools, including a new static collector of entity structural information, a contract for effective communication between the tools, a translation strategy to generate CML from Mono2Micro decompositions, i.e. entities, clusters, and functionalities, an extension to the CML syntax to support concepts from decomposition in the form of *Coordinations*, and new diagram generators from CML based on translated decompositions.

The artifacts developed in the project are publicly<sup>1</sup> available together with the description of the procedures necessary to use them.

---

<sup>1</sup><https://github.com/socialsoftware/mono2micro/tree/master/tools/cml-converter>



# Bibliography

- [1] “Research project PTDC/CCI-COM/2156/2021: Data consistency in microservices compositions.” [Online]. Available: <https://doi.org/10.54499/PTDC/CCI-COM/2156/2021>
- [2] T. Lopes and A. R. Silva, “Monolith microservices identification: Towards an extensible multiple strategy tool,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, 2023, pp. 111–115.
- [3] S. Kapferer, “Architectural refactorings,” <https://contextmapper.org/docs/architectural-refactorings/>, 2020, accessed: 2023-01-04.
- [4] C. O’Hanlon, “A conversation with werner vogels,” *Queue*, vol. 4, no. 4, p. 14–22, May 2006.
- [5] P. Di Francesco, P. Lago, and I. Malavolta, “Migrating towards microservice architectures: An industrial survey,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909.
- [6] M. Fowler, “Monolith first,” 2015. [Online]. Available: <https://martinfowler.com/bliki/MonolithFirst.html>
- [7] D. Haywood, “In defence of the monolith,” 2017. [Online]. Available: <https://www.infoq.com/minibooks/emag-microservices-monoliths/>
- [8] F. Ponce, G. Márquez, and H. Astudillo, “Migrating from monolithic architecture to microservices: A rapid review,” in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, pp. 1–7.
- [9] 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>
- [10] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, “Decomposition of monolith applications into microservices architectures: A systematic review,” *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023.

- [11] L. Nunes, N. Santos, and A. Rito Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings*, 2019, pp. 37–52.
- [12] S. Santos and A. R. Silva, "Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures," *Journal of Web Engineering*, 2022.
- [13] B. Andrade, S. Santos, and A. R. Silva, "A comparison of static and dynamic analysis to identify microservices in monolith systems," in *Software Architecture*, B. Tekinerdogan, C. Trubiani, C. Tibermacine, P. Scandurra, and C. E. Cuesta, Eds. Cham: Springer Nature Switzerland, 2023, pp. 354–361.
- [14] V. Faria and A. R. Silva, "Code vectorization and sequence of accesses strategies for monolith microservices identification," in *Web Engineering*, I. Garrigós, J. M. Murillo Rodríguez, and M. Wimmer, Eds. Cham: Springer Nature Switzerland, 2023, pp. 19–33.
- [15] J. Lourenço and A. R. Silva, "Monolith development history for microservices identification: a comparative analysis," in *2023 IEEE International Conference on Web Services (ICWS)*, 2023, pp. 50–56.
- [16] 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.
- [17] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.
- [18] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?" *IEEE Access*, vol. 9, pp. 32 721–32 733, 2021.
- [19] O. Özkan, Önder Babur, and M. van den Brand, "Domain-driven design in software development: A systematic literature review on implementation, challenges, and effectiveness," 2023.
- [20] S. Kapferer. and O. Zimmermann., "Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, INSTICC. SciTePress, 2020, pp. 299–306.
- [21] S. Kapferer, "Context map discovery," <https://contextmapper.org/docs/reverse-engineering/>, 2021, accessed: 2023-01-04.
- [22] V. Vernon, *Implementing Domain-Driven Design*. Addison Wesley, 2013.



- [23] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01078532/document>
- [24] J. F. Almeida and A. R. Silva, "Monolith migration complexity tuning through the application of microservices patterns," in *Software Architecture*, 2020, pp. 39–54.
- [25] J. Correia and A. Rito Silva, "Identification of monolith functionality refactorings for microservices migration," *Software: Practice and Experience*, vol. 52, no. 12, pp. 2664–2683, 2022.
- [26] S. Kapferer, "A modeling framework for strategic domain-driven design and service decomposition," Master's thesis, University of Applied Sciences of Eastern Switzerland, 2020.
- [27] A. Singjai, U. Zdun, and O. Zimmermann, "Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 25–35.
- [28] D. M. Le, D.-H. Dang, and V.-H. Nguyen, "On domain driven design using annotation-based domain specific language," *Computer Languages, Systems and Structures*, vol. 54, pp. 199–235, 2018.
- [29] F. Rademacher, S. Sachweh, and A. Zündorf, "Towards a uml profile for domain-driven design of microservice architectures," in *Software Engineering and Formal Methods*, 2018, pp. 230–245.
- [30] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing microservice-based applications by using a domain-driven design approach," *International Journal on Advances in Software (1942-2628)*, vol. 10, pp. 432 – 445, 12 2017.
- [31] C. Richardson, "Developing transactional microservices using aggregates, event sourcing and cqrs," 2017. [Online]. Available: <https://www.infoq.com/minibooks/emag-microservices-monoliths/>
- [32] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts*. O'Reilly Media, Inc., 2021.



