TÉCNICO LISBOA

From a Monolithic to a Microservices architecture

Luís Fernando Ambrósio Nunes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

September 2018

Acknowledgments

I would like to thank my parents and my girlfriend for all their support, caring and encouragement through all my academic path. Without this interest and effort to pursue my dreams and interests, this dissertation would not have been possible.

I would also like to acknowledge my dissertation supervisor Prof. António Rito Silva for his tremendous availability, accompaniment, insight, support and sharing of knowledge that made this thesis achievable.

Lastly, to all my friends and colleagues that helped me grow as a person and future professional, sharing the same experiences and being there for me when needed, thank you!

To each and every one involved, thank you for everything.

Abstract

Microservices are considered a hot trend, with every company that implements it having an exclusive blog detailing every change in its codebase. Its functional decomposition and scaling capabilities coupled with autonomous development makes it highly valuable in business comparing to a traditional monolithic approach. This thesis researches the scope regarding the incremental migration from a monolithic to a microservices architecture, merging all background and principles necessary for the development of tools that assist developers on this process while addressing software re-engineering concepts, such as domain-driven design, code analysis and graph clustering.

The proposed method starts with the retrieval of information regarding the relationship between the classes and controllers of an application, subsequently applying hierarchical clustering with a similarity measure based on controller access. This allows us to explore transactional contexts while addressing the migration issue of context separation. The results of this approach, paired with an user interface with visualization capabilities, allows the filling of a gap in terms of support tools that exist for software architecture migrations.

The comparison of our approach with software architecture formal tools and expert decompositions resulted in high precision values, that reflect that accurate service candidates are produced. The approach proposed in this thesis asserts that from the transactional contexts of an application, modular conclusions can be drawn, while also showing that it can better inform developers in the process of an architecture migration to microservices.

Keywords

Monolithic, Microservices, Domain-Driven Design, Static Analysis, Dynamic Analysis, Hierarchical Clustering, Modularity Re-engineering.

Resumo

As arquiteturas de microserviços estão a ser cada vez mais utilizadas por todo o tipo de empresas de software, sendo que a sua adoção proporciona uma decomposição funcional e capacidades de escalabilidade particulares aos seus diferentes serviços. Esta dissertação vai investigar o âmbito do processo de migração incremental de uma arquitetura monolítica para uma de microserviços, apresentando e juntando todos os princípios necessários, tal como design orientado a domínio, análise estática de código e clustering de grafos.

O método por nós proposto requer a extração de informação das relações entre controladores e classes de uma aplicação de software, gerando uma representação em grafos destas relações. De seguida aplicamos clustering hierárquico com uma medida de similaridade personalizada baseada nos acessos dos controladores. Isto permite-nos lançar uma hipótese em relação à separação de contextos do processo de migração entre arquiteturas. Os resultados do nosso método são de seguida integrados com uma interface gráfica com capacidades visualização dos grupos de contextos gerados. A sua visualização e manipulação expõem uma ferramenta que permite suportar este processo de migração arquitetural.

Comparando com ferramentas já existentes, a nossa abordagem obteve valores de precisão mais elevados, refletindo-se nos serviços originados. A abordagem proposta assume que os contextos transacionais de uma aplicação de software podem dar origem a informação sobre a sua modularização, mostrando a sua utilidade em informar os programadores num processo de migração arquitetural.

Palavras Chave

Monolítico, Microserviços, Design orientado a domínio, Análise Estática, Clustering Hierárquico, Reengenharia de modularidade.

Contents

List of Figures

List of Tables

Listings

Acronyms

BCEL Byte Code Engineering Library

CQRS Command Query Responsibility Segregation

FN False Negatives

FP False Positives

GUI Graphic User Interface

LSM Levelized Structure Map

MVC Model-View-Controller

ORM Object-relational mapper

WALA Watson Libraries for Analysis

TF-IDF Term Frequency–Inverse Document Frequency

TP True Positives

1

Introduction

Contents

Companies all over the world are more frequently adopting Microservices[1] as their software architecture, a modular approach that has the main characteristic of decomposing an application into a set of services, as illustrated by **Fig. 1.1**.

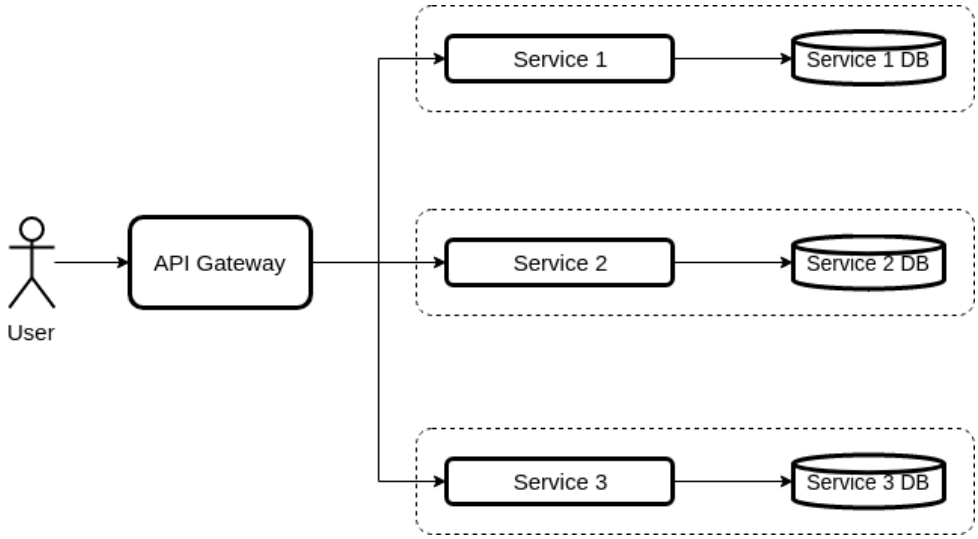

Figure 1.1: Illustration of the components of an usual Microservices Architecture

This type of architecture results in numerous advantages, such as:

- Increase in *quality and speed* on the development of applications with a large and complex domain due to the existence of autonomous teams developing different services that integrate into the same system. The presence of small teams allows an agile software development approach and requires only point-to-point collaboration, driven by the interaction between microservices.

- *Heterogeneity in implementation* of services when using new technologies, since every service has its own solid boundary, different services can be written in different programming languages and organized around its business capabilities. There also exists the option of re-implementing a service using a different language and discarding the previous implementation.

- Validation of the deployment of each service on optimal hardware, resulting on independent *scalability*. Its plausible that certain services will have more affluence than others and as each service runs separately, an independent scaling is possible.

The Microservices architecture is an alternative to a Monolithic architecture. This type of architecture is mostly present on *Enterprise Applications* [5], handling a lot of complex and persistent data while implementing the required business logic. The common architectural style followed in the development of a Monolithic architecture is a 3-tier approach **(Fig. 1.2)** that consists in:

[1] http://microservices.io/patterns/microservices.html

- **Client-side UI:** Display of information/services to be accessed by the user of the application (*Example:* Front-end HTML and JavaScript on the user's browser).
- **Server-side Application:** The modules that execute all domain logic, operate the database, handle HTTP requests and select the HTML views. Represented in the image by the replication of the agglomerate of services.
- **Database:** Relevant tables to the system that are accessed by all instances of the web application.

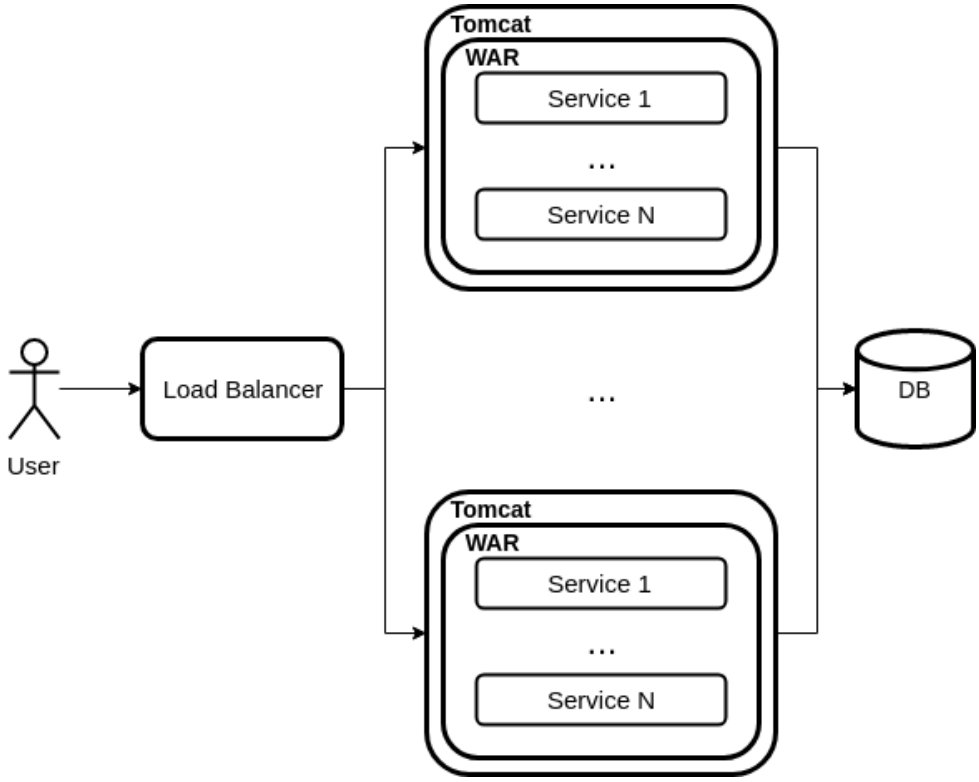

Figure 1.2: Illustration of the components of an usual Monolithic Architecture

In this tiered application there is a clear Monolith, the server-side application, being that every change that we make in the system will involve the rebuild and deployment of the entire server. Since we are before a Monolithic architecture, there is a clear vision on why it cannot have the advantages of Microservices. As *Chris Richardson* explains[2] the whole context behind this type of architecture, its clear to say that the business logic is all gathered on the same place and the increasing complexity of the system, as it grows, will create situations where *no developer can understand object intents or related models*, changing them and leading to a *loss in coherency and integration*. This also causes a *bottleneck regarding scalability*, since we can only scale the entire functionality of the system horizontally with a limit and not just allocate the desired resources to the different services, scaling according to its

[2]http://microservices.io/patterns/monolithic.html

independent needs. These applications are also implemented using a single development stack, *limiting tools and new technologies*, as in the creation of new components the same technology has to be used, thus limiting the capability of choosing the right tool for the functionality to be developed. Furthermore, the possibility of developing on top of a framework that stops being updated also exists, creating a probability that to migrate to an updated framework can lead to rewriting the whole application.

Although monoliths have such disadvantages, experts such as Dan Haywood defend them [6] noting that they are *change tolerant* regarding module responsibility, since that with a complex domain, early defined boundaries on Microservices can hinder the development of a robust domain model, which will be difficult to refactor because of the distributed boundaries between its elements. In a monolithic context, *Cyclic dependencies between modules* are solved with tools such as Maven, while in Microservices there are difficulties finding and solving this type of problem. Modules having its own persistent data and co-locating it enables retrieving *data from multiple models with a simple SQL query*. The effort put into solving this problem in a microservices context is higher, as different tables are owned by different services, existing the need to keep materialized views using CQRS *(Command Query Responsibility Segregation)*. This aspect also makes the *Transactional atomic behavior simple and user friendly* in a Monolithic architecture. Following a business operation we can expect state changes on many modules, these modules being co-located in the same RDBMS can make us rely on its transactional behavior to assure atomicity. When using services, every service will have its own data store, making the previous state changes independent and relying on asynchronous messaging between the several domains to make sure the transactions are done accordingly for the end user, which requires relaxing the transactional model associated with each end user request. Finally, the interaction between modules is just a method call and services require an interaction protocol, data format and other formalities.

One more expert in favor of the Monolithic approach is Martin Fowler[3], stating that we should start with a Monolith and explore the complexity of the system before diving in inconsistent and undefined boundaries. When the complexity rises, then the separation of the system into services will be valuable.

Companies that made such changes include *Ebay* [4], *Uber* [5], *Amazon* [6] and *Netflix* [7]. All of these companies are considered hyper-growth companies, meaning that the primary factor in its core is its potential for fast growth in the market they are engaged in. This complexity uprising is a perfect match for a Microservices architecture.

When a Monolithic system starts to require the qualities provided by a Microservices architecture, it is necessary to migrate to this type of architecture. The problem here is exactly this migration and how we can do it incrementally and in a controlled way, managing the underlying modularity and transactionality

[3] https://martinfowler.com/bliki/MonolithFirst.html
[4] http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf
[5] https://eng.uber.com/soa/
[6] http://highscalability.com/amazon-architecture
[7] https://medium.com/netflix-techblog

due to the architectural changes. This problem of not only identifying the correct boundaries that define the different services that compose a microservices architecture, but creating an ideal migration step-by-step workflow is the main topic addressed in this thesis.

1.1 Thesis Objectives

The goals of this thesis are the following:

- To present the theoretical methodologies and concepts behind the creation of microservices.
- To research the state of the art code analysis and clustering methods.
- To present already existing architecture migration approaches.
- To develop a tool that merges the previously studied methodologies and is capable of supplying information to a developer of a software application that is useful for its migration to microservices.
- To study the impact and veracity of the information given in the scope of a microservices architecture.

All these together corroborate into a tool that, given a Java application, presents a dendrogram that ultimately shows domain clusters that mimic services in a microservices architecture. These are presented in an interactive web view, with knowledge of the classes of the clusters and the controllers that access them.

1.2 Thesis hypothesis

The stepwise migration from a Monolithic architecture to a Microservices architecture should be supported by tools that help the developers in this task. in our case, we developed a tool that tackles the problem of context separation by **validating whether the transactional contexts of an application can translate into its modularization**. This hypothesis tackles the current state of the art software architecture migration techniques, as information about the modularity of software applications is usually retrieved from its structural relationships and not its transactional context.

1.3 Document organization

This thesis is organized as follows: Chapter 1 contextualizes the types of architectures that exist in software development, their advantages/disadvantages, the presentation of the thesis hypothesis and

the thesis objectives. Chapter 2 presents the state of the art by exposing ideas that are directly related to the problems and the desired solution, including the methods used for designing modular software and the existent types of source code analysis. In Chapter 3 there is a description of the thesis problem and the approach used to solve it, explaining all the approach steps. Chapter 4 presents the results obtained from applying the approach to two monolithic applications, posteriorly evaluating these results with the help of a business expert and a set of metrics. Finally, Chapter 5 shows the main consequences and results obtained from the development of this thesis, summarizing all contributions and proposing future work.

2

Techniques in architecture migration

Contents

When we are talking about developing an application using a Microservices architecture, there is a clear path that we should follow and it all begins by having the fundamental concepts of *Domain-Driven Design* (as presented by Eric Evans and Vaughn Vernon on their books [7] [8]), because a well-structured and modular monolith is the best way to start a service decomposition.

2.1 Domain-Driven Design

The objective of following a strategic design is knowing how to design, comprehend and manipulate a large and complex system, which plays a huge role on the development of a sizable project, where we usually have a very large model. Therefore, due to the large number of concepts in the model, there is a tendency for bugs, unreliability and lack of understanding, resulting in a confusing communication between team members. To address this problem, several context models should be defined, coupled with a set of conditions that associate them with meaning, making explicit not only their boundaries but also their code base and data schema.

2.1.1 Bounded Contexts

A crucial term related with this scope is a *Bounded Context*, delimiting the context to which a model can be applied while helping team members understand its consistency and how it relates to other contexts, but only worrying about the applicability inside its bounds. The importance of this term is supported by many experts in the area, including Mike Gehard, who on one of his several presentations, describes the journey from Monolith to Microservices [6].

A Microservices architecture is more successfully achieved when starting with a modular monolith following DDD principles, being able to experiment freely and making mistakes that are reversible, while in a Microservices architecture, the constant change in boundaries without system knowledge will lead to an unstable code base and separation difficulties[1]. These boundaries are exactly the ones created by bounded contexts, requiring a great deal of experimentation to get them right. With correct and clear boundaries, we can just break apart a monolith without worrying about code tangling and create a messaging interface for it, rapidly building a microservice. Due to the hardship behind defining bounded contexts, several precautions should be kept in mind:

- *Coded interfaces* between models need to match in order to get the expected behavior.
- Avoid *duplicate concepts* that lead to having the same element on two models, updating both when there is a change. Two versions of the same concept will always lead to incoherent data and bugs.

[1] https://martinfowler.com/bliki/MonolithFirst.html

- Redefine processes till they are explicit and there are no *False Cognates*, terms that represent several concepts at the same time, leading to difficult communication amongst team members.
- Keep a policy of *Continuous Integration*, merging all work contexts and correcting them, avoiding fragmentation on the system and the associated teams.
- Define a *Context Map* to help team members understand the global view and how context models interact, making the names of the bounded contexts part of the *ubiquitous language* and emphasizing communication between them.

Bounded contexts have distinct patterns associated with its interactions:

- **Shared Kernel:** Share a part of the domain model between two teams, including the code base and database associated with that part, reducing the cost of synchronization between them.
- **Customer/Supplier:** Upstream and Downstream modules compose two bounded contexts and two different teams. Upstream modules will be the ones associated with the source of the software while downstream modules will be related to its use cases, meaning that having a module, its upstream modules are what it depends on, being affected by its bugs or design, and its downstream modules the other parts of the system that our changes in our module affects. In order to remove the power to change from the downstream team a relationship should be built, having the downstream team play customer to the upstream team, negotiating tasks and developing joint acceptance tests on interfaces.
- **Conformist:** Establishing a customer/supplier relationship without having the teams under the same management can create several issues. In the possibility of lack of motivation from the upstream team, the downstream will have no options left regarding their necessities. Instead of abandoning the upstream, there is a conformity to the path that the upstream model takes, deleting every translation layer between bounded contexts and simply accepting the upstream model. This makes integration simple and creates a *ubiquitous language* in the customer/supplier relationship between two teams that had difficulties collaborating.
- **Separate Ways:** Having functionalities that are seen as distinct with no type of relation between them, usually means that there is no need for integration. Therefore, having a bounded context with no functionality, object or data sharing with other contexts, means we need to worry about the scope of the context and its solutions.
- **Open Host Services:** Designing a subsystem that will be highly stressed (performance and availability) also means designing translation layers for the systems outside its context that need integration with it. This pattern calls for a definition of a protocol that makes everyone that needs integration with it, use it as a set of services.

- **Anticorruption Layers:** Legacy models integrations with newer ones will lead to modifications of the new model in order to match interfaces/semantics, even when it does not fit the project requirements, thus "corrupting" this model. Reuse of legacy models can have value, but layers with no possibility to communicate using different protocols or platforms will most probably create errors and can even corrupt data layers. In order to avoid corruption of any kind, this patterns proposes the creation of a layer that isolates clients, doing bidirectional translations and communicating with the other system through its interface.

Another important approach to bounded contexts, as presented by S.Newman [1], is that every existing domain is composed by several bounded contexts that have explicit interfaces, where their inner components can share or not relevant information. This structure paired up with private persistent data to each relevant context can be considered a Microservices architecture, as these well-defined contexts constitute appropriate business boundaries and compose small and independent services that communicate through its APIs. Considering a warehouse business with a real domain that includes the manipulation of all orders and underlying finances, analyzing the mapping between this domain and the modular division by bounded contexts will confirm these aspects: The warehouse department (that manages all orders) and the finance department (that takes care of all economic affairs) represent important roles on the system, having particular information that is exclusive to that department and other that they need to share. Therefore we can separate this business into two bounded contexts, the Warehouse and Finance context, as seen on **Fig.2.1**, where the *Stock item* composes the shared model of both bounded contexts, creating a **Shared Kernel**. Besides providing high cohesion to our models, these contexts are also perfect contenders for microservices as they represent the same boundaries as services.

This identification can lead to the recognition of nested contexts, creating the possibility to have a boundary on the top hierarchy, grouping all of these contexts or several small boundaries on the smaller contexts, meaning that in a division for a Microservices architecture, this would be the separation of one service into even smaller services. Outside this context, this choice would depend on team division for each service or testing capabilities. The challenge is now to incrementally establish these boundaries on the code and corresponding databases, following the following steps:

1. Understand the system capabilities and analogous code, identifying the different contexts and encapsulating them in packages, subsequently doing the same process inside these packages.

2. Infer which components are reading from and/or writing to the database, creating a repository for each context that only accesses a part of the database.

3. Split the repository layer as a mapping between contexts and its particular schemas.

In this final step of splitting the Monolithic schema, we need to take in account three problems:

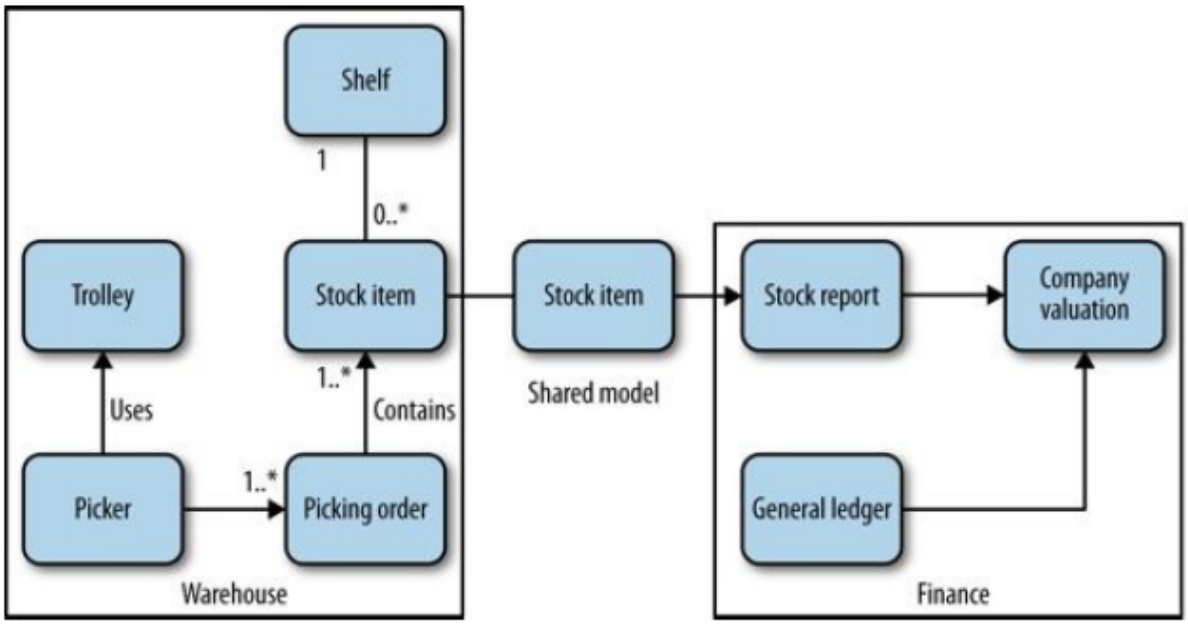

Figure 2.1: Division in bounded contexts for the Warehouse business domain, from [1]

1. **Shared Static Data:** Can be solved by duplicating the table in question, treating the static data as code (or a property file) or pushing the static data into its own service.

2. **Shared Mutable Data:** When mutable data is shared and we verify that there is an average amount of reads and an occasional number of writes, there will be a need to create a concrete context for the concept of that table.

3. **Shared Tables:** Divide the table in two, creating the corresponding and needed tables for both services.

2.1.1.A Transactions and Bounded Contexts

Having approached bounded contexts using a modular perspective, there is a need to understand its transactionality. Transactions are very important on software systems as they guarantee consistency on the states of our data. On a Monolithic schema, an individual transactional boundary is involved in its CRUD operations, but the division of the database for its several contexts will lose such property, having the need for several transactional boundaries. An update for two tables, that used to occur in the context of a single transaction boundary, is now split into two different boundaries, raising inconsistency issues when one table is successfully operated and the other is not, existing several approaches:

1. **Retry:** Implement a queue that captures the operation to be realized on the table and do it posteriorly. This is labeled *eventual consistency*, as the consistency of our system is not immediate, but

promised in the future.

2. **Abort:** Nullify all operations, initiating a particular transaction that will undo all the changes so far until a consistent state is reached.

3. **Distributed Transactions:** The extension of several transactions on different systems that are handled by a transaction manager.

2.1.2 Aggregates

In a domain model, aggregates are objects that are associated as a cluster with the purpose of being treated as one, in order to guarantee the consistency of changes to objects in that model. An aggregate will have a root (reference for external objects) and a boundary (defines the content of the aggregate), representing a clear relationship between objects and presenting a clear understanding of its context in the domain model. Chris Richardson considers them the **"key to developing microservices"** [6] as they follow important rules:

- They do not reference each other through object references but through **primary key**, making the aggregates loosely coupled and allowing them to be referred from different services.

- **Transactions will only affect an individual aggregate**, assuring that the transaction is limited to the scope of the service in question, which is a crucial property we want in a Microservices architecture.

- Aggregates will update sequentially and using fine-grained components will enable the capacity of the application to process concurrent requests and have room for scalability upgrades.

Due to the fact that we can only create or update one aggregate transactionally, the necessity to provide *data consistency* between aggregates will be at stake. We can update multiple aggregates if they are present in the same service (and in the same RDBMS) but the correct way is using events, releasing an event when an important change happens and other aggregates subscribed to it will update accordingly.

This way there is a clear view of how a Microservices architecture can be built from this point of view, where each service is composed by one or more aggregates following a DDD, transactions are made by services on an individual aggregate and consistency between aggregates is conditioned by an asynchronous event processing mechanism.

2.2 Static Analysis

To decompose a system into services, there is a need to capture the relationships that exist between the already existing modules of a monolithic architecture. As the objective of a migration into microservices starts with the corresponding monolithic source code, one method to retrieve such information is static analysis, allowing feature extraction without executing any of the code. Statically extracting the method calls between classes/modules or its dependencies allows us to visualize the data flow of the application and establish their boundaries, creating groups of classes that can be classified as services.

Practically, this type of analysis consists on the execution of tools that inspect the source code of an application through the use of several techniques, including: *Data flow*, *Control flow*, *Taint* and *Lexical analysis*.

Deducing modules and its relationships through the analysis of the source code of an application can be used for several purposes, such as code revision or just software comprehension. Its applicability can also include the scope of software architecture, such as discovering *Design Patterns* in the code, as presented by G.Antoniol et al. [9]. Identifying a Design Pattern is architecturally relevant because it helps, in a complex system, to understand underlying abstractions and used design choices. This approach will result in a graph where classes are represented by nodes and its relationships by edges, being that the pattern will be identified by requiring a number of relationships between nodes, enforcing the conditions and discarding the ones that do not match their required number. In order to guarantee that the intermediate information is not dependent on the programming language, an abstraction is used called AOL(*Abstract Object Language*). The relevant information for identifying every pattern is not the same, meaning that for some patterns, its structural properties are not enough, existing the need to also capture runtime behavior. This is why this and many other approaches do not try to identify patterns that have poor structural information, making the problem impractical.

N.Tsantalis et al. [2] also approaches Design Pattern detection with a graph algorithm, identifying patterns based on a method that computes the vertex similarity between the graph generated by the system under detection and the corresponding pattern we want to identify. To represent the static information of the code and build a comparable unit between the system and the desired pattern, a Java program was built that outputs the relationship between classes as matrices, being that a specification of a class is mapped to a directed graph and that graph is mapped to a matrix. First, features (attributes or relationships between classes) need to be chosen that characterize the pattern we want to identify, as for example, the creation of objects, inheritance, etc. That information has then a graph and matrix representation (**Fig.** 2.2) and is compared to the representation extracted from the system we intend to capture patterns from, using a similarity algorithm.

This approach was evaluated by analyzing the results (given as True Positives (TP), False Positives (FP) and False Negatives (FN)) by using relevance metrics, such as Precision (TP/(TP+FP)) and Recall

(TP/(TP+FN)), being that precision reported 100% on all the tested patterns, and Recall reported 100% on 83% of the patterns. This result of the precision metric is only possible due to the nonexistence of False Positives of the example, that has a direct correlation with using only core information/features of the patterns and excluding characteristics that are not unique to the system in cause. If the common features that exist between patterns would be taken into account, there would be no absolute certainty on the pattern classification, resulting in the presence of False Positives. The recall metric did not return a 100% value on the Factory and State/Strategy patterns, justified by small changes in the original pattern that do not exactly match the literature in which the this approach was based.

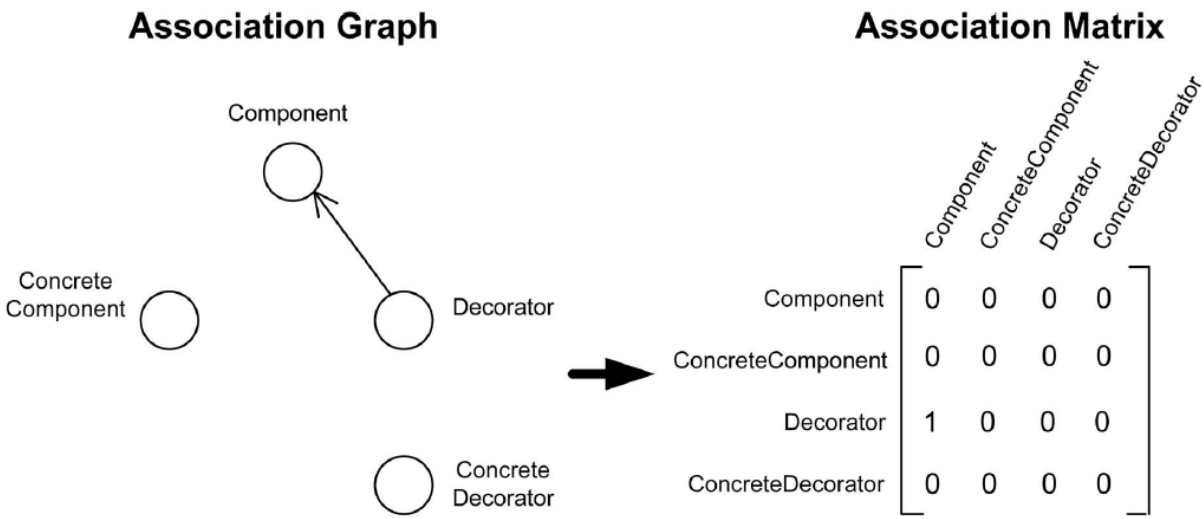

Figure 2.2: Graph and Matrix representation of the *Associations* that exist between the classes of a *Decorator Design Pattern*, from [2]

One clear and common obstacle to every static analysis approach is the state space explosion associated with big systems. Design patterns have a big number of class relationships that describe them and those combinations need to be tested in the entirety of its domain to classify the patterns that exist on it. This problem results in lack of efficiency, being that this approach solves it by relying on inheritance, since it is a common relationship in the corresponding domain, building clusters and running the algorithm on these subsystems instead of its whole. This means that if while running static analysis on a code, the information retrieval step is too slow, a subsystem-type approach should be taken.

2.2.1 Tools for Static Analysis

Associated with every source code analysis, several tools with different functionalities exist:

1. Watson Libraries for Analysis (WALA)[2], developed as the DOMO project at IBM[3], is an example

[2] http://wala.sourcefourge.net
[3] http://www.research.ibm.com/

of a set of libraries that have the capabilities to statically and dynamically analyze *Java* bytecode and other relevant languages. By providing a *.jar* file as an argument to the several configurations, distinct types of interprocedural analysis can be done, including:

(a) **Class Hierarchy**, retrieving information about the types used by building a tree from *ClassLoaders*. This usually represents the first step in using this library, as bytecode is read into memory and parsed according to its type.

(b) **Call Graph**, having the possibility to be sensitive or not to context, represents a graph with methods on its nodes and call targets on its edges. These call graphs can also be manipulated, having the possibility to prune them (providing the nodes to remove) or just access a partial graph (providing the roots and nodes for the new graph). The construction of a call graph is done through the execution of *pointer analysis* (associating heap references to a variable) with simultaneous graph building.

(c) **Intermediate Representation (IR)**, defining a control-flow graph that represents the instructions of a method.

(d) **System Dependence Graph**, representing the dependencies that exist between the objects of the system.

This tool has the advantage of being designed for performance and customization while also having visualization possibilities in trees (SWT[4]) and graphs (Dot tool from Graphviz[5]).

2. **SOOT** [10] is a Java framework that analyzes bytecode and source code, being capable of generating call graphs and control flow graphs. The workflow that represents this framework will be different depending on whether the tool is dealing with *.class* files (bytecode) or *.java* files (source code). Analyzing bytecode will be directly mapped onto the intermediate representation, while examining source code will have the need to be parsed first. Posteriorly, the workflow will be the same for both, generating three kinds of intermediate representations:

   (a) **Baf:** Stack-based approach, representing a collection of instructions.

   (b) **Jimple:** Existence of local variables in order to eliminate the need for a stack.

   (c) **Grimp:** Similar to a Java code that has been decompiled, being appropriate for reading purposes.

   Besides having a lot of flexibility regarding intermediate representations, this framework has downsides concerning the non-existing implementations of system dependence graphs and poor customization in comparison to **WALA**.

[4] https://www.eclipse.org/swt/
[5] http://www.graphviz.org

3. **Java-Callgraph**[6] is composed by a suite of programs that enables the user to execute static analysis over a given jar file. It uses Apache Byte Code Engineering Library (BCEL) in order to manipulate the .class files, returning a text representation inside a .txt file with the following syntax:

   - **M**:class1:<*method1*>(arg_types) (typeofcall) class2:<*method2*>(arg_types)

   This syntax means that the *method1* of *class1* called the *method2* of *class2*, including the types of the arguments given to the call. This tool gives us the output that is easiest to read and manipulate, as it generates the call graph for the entire application, existing only the need to parse of the information that is wanted.

In the scope of our thesis, all of these tools could be used for information retrieval, generating call graphs for the applications and enabling that data to be further manipulated in order to create appropriate clusters.

2.3 Dynamic Analysis

As we are looking to identify every kind of feature that can be helpful on decomposing the system into services, the method calls between classes during the execution of the system can also be taken into account, this is, the dynamic information of the application. Dynamic Analysis is the monitoring of the execution of a program, capturing characteristics such as memory consumption, time of execution and value computation. Examples of this type of analysis are the execution of software tests and profiling.

As a complement to statically analyzing source code, Dynamic Analysis is introduced by M. Ernst [11] as an approach that has the capability to improve the previously obtained static results by providing data that would not be available otherwise. Static Analysis constructs an easy to manipulate model that has the main characteristic of being abstract, while Dynamic Analysis is not, as it can capture the real behavior of the system under test. The outcome of both approaches are affected in the following way:

- A static approach translates into weak results as a consequence of its conceptualizations, but has the capacity of abstracting a model to be used in the future.
- A dynamic approach has an increase in precision of the results due to certainty of the captured information, but requires the generation and selection of test suites.

Therefore, researchers have been using both analysis together instead of separately.

D.Heuzeroth et al. [3] approaches the Design Pattern problem using not only static information but also the dynamic aspects present in design patterns, more specifically the communication sequences that exist between objects of certain classes. As a design pattern defines certain rules, an instance of

[6] https://github.com/gousiosg/java-callgraph

such will be characterized through a tuple that contains attributes, methods, or classes. The approach taken is represented in **Fig.** 2.3.

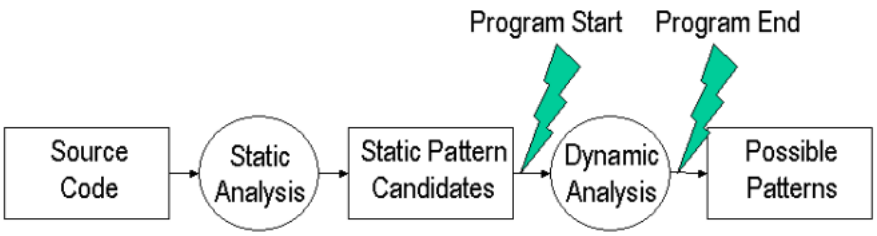

Figure 2.3: Overview of the Design pattern detection approach, from [3]

The rules previously defined that characterize design patterns are separated into a static and dynamic context, being that each context analysis rejects the patterns that do not comply to its restrictions, acting like a two-step filtering process. Examples of these restrictions for a static context is the existence of a class X with method declarations with the name *addListener()* and *removeListener()* and for a dynamic context, the presence of a 1:N relationship between the elements of certain classes, obtained from the monitoring of the objects of such classes. The application of Static Analysis on the source code of the system creates a considerable number of candidates, being that these proposed instances are evaluated by executing the program and monitoring the runtime rules defined for the considered patterns, returning the instances that satisfy such rules. The technique applied for Static analysis is the construction of abstract syntax trees (AST) from the code, where the static context rules for the detection of a pattern are defined by the existing relationships between nodes of the AST. For Dynamic Analysis, the execution of set of nodes of the AST is monitored, being that the dynamic context rules of a pattern are actions and relations of particular nodes. This approach was evaluated using a real system that had the *Observer Design Pattern* with an explicit documentation. After the filtering regarding both contexts, the number of False Positives was small (no precision or recall metrics were presented).

2.3.1 Tools for Dynamic Analysis

Dynamic Analysis can be done through the use of automatic case test generation or profiling, using tools such as:

1. **Evosuite**, as presented by G.Fraser et al. [12] [13], consists of a tool that has the objective to automatize test case generation. It automatically generates JUnit test cases for the Java classes given, using code coverage criteria as:

    - Line Coverage
    - Branch Coverage

- Exception Coverage
- Weak Mutation Coverage
- Output Coverage
- Top-Level Method Coverage

These criteria are used with a Genetic Algorithm to produce maximum coverage test suites, evolving the population of randomly chosen test suites until this coverage is obtained, using techniques as selection, crossover and mutation. In each evolution iteration, the best individuals of the previous population are used. Two kinds of output are originated from this tool:

(a) **Scaffolding Java file** containing JUnit annotations that ensures proper method execution in relation to the tests and provides a way to execute all tests in an equal and consistent state.

(b) **Test Java file** including the JUnit test suite for the desired class.

This tool uses bytecode instrumentation (static analysis) for the retrieval of class information and the tracing of the coverage fitness values and dynamic symbolic execution (dynamic analysis) for testing new inputs on the execution of alternative program paths. The resulting test suites can also be used for a further dynamic analysis.

2.3.2 Modularity patterns in code

Working with existing code can lead to the identification of patterns of modularity, whether in the definition of the code itself or its intermediate representation after running static or dynamic nalysis approaches.

Clustering techniques are used in the context of data analysis, more specifically in the grouping of data that is similar according to a defined measure or distance. This type of analysis can also be applied to software, as described in the following related work.

R.Schwanke [14] presents a tool for software systems that addresses this kind of modularization through the use of heuristics. These heuristics will serve as advice for the architects in the context of improving the modularity of the system, providing them with a set of recommendations that they can approve or disapprove. The two core services that contribute to the heuristic advice regarding modularity are:

- **Clustering:** The sharing of information related to design by a group of procedures, enabling its organization into a module. This type of information is based on an heuristic, presenting that if two procedures have unit-names in common, then they are more likely to share design information and could be joined in the same module.

- **Maverick Analysis:** The discretization of singular procedures that do not make sense in a module, sharing more information with others than the module that it is currently in.

Modularization is treated as an heuristic process and its core principle is **information sharing**. In this context, a **feature** is a non-local name with a scope that contains two or more procedure bodies, being that if a procedure A calls B, then A acquires the feature ”B” and B the feature ”Called-by-A”. The notation for representing common and different features between procedures is presented as:

$$a \cap b \qquad \text{Common features between procedure A and B} \tag{2.1}$$

$$a - b \qquad \text{Difference of features between procedure A and B} \tag{2.2}$$

The similarity function used to compare procedures uses the previous notations and is presented as follows:

$$Sim(A, B) = \frac{W(a \cap b) + k * Linked(A, B)}{n + W(a \cap b) + d * (W(a - b) + W(b - a))} \tag{2.3}$$

Being that:

$$W(PROC) = \sum_{x \epsilon PROC} w_x \qquad \text{Weight of PROC (procedure) is the weight of all its features x} \tag{2.4}$$

$$Linked(A, B) = \begin{cases} 1 & \text{if A calls B or B calls A} \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

In formula (2.3), *k*, *n* and *d* are selected by trial and error, and the weight of a feature x is estimated through its *Shannon information*:

$$Weight(x) = -log(Probability(x)) \tag{2.6}$$

where the probability is represented by the fraction of procedures that have the feature x.

The **Clustering** service will use *hierarchical clustering*, creating a group for each procedure and joining them by the highest similarity. The similarity measure used in this calculation is given by the equation (2.3). The **Mavericks Analysis** service will consider a procedure as being in the wrong module when it has a poor similarity measure with its neighbors.

This approach was applied to real software systems, being that the application of the clustering service was limited to one system that had 7 modules and 64 procedures. The algorithm made 56 choices and reduced the groups of 64 to 7. Out of those 56 choices made, architects concluded that 40 of them were already in the same module in the past and 10 out of the remaining 16 choices were correct modular choices, proving that the approach provides good advice regarding modularity. The mavericks service was evaluated on 4 systems and flagged 10-30% of the procedures as mavericks,

being that only 20-50% of these derived from modularity errors.

Applied to graphs, modularity represents a quality measure of the connections between nodes, respecting the following equation:

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij}) \delta(C_i, C_j) \tag{2.7}$$

Summing all pairs of vertices, being $A(C_i, C_j)$ the adjacency matrix, P_{ij} the number of edges between vertices i and j, m the total number edges and δ a function that returns 1 if the vertices are in the same cluster. Its maximization constitutes approaches to graph clustering methods. As presented by S.Fortunato [15], a high value of modularity is a sign of a good partition and the maximization of such quality constitutes an appropriate approach for cluster detection. Several algorithms exist, including:

- **Greedy method of Newman:** Method based in *agglomerative hierarchical clustering*, grouping nodes into clusters with the condition that modularity is increasing. This method is based on the same technique as the **Clustering** service above.

- **Extremal optimization:** Method based in local variable optimization that explicits the contribution that each node has to the system. A random partition of the graph in two (with the same number of vertices) is done and on each iteration the node with the the lowest value of fitness(local modularity divided by its degree) is added to the other cluster. The algorithm stops when global modularity cannot be increased further.

2.3.2.A Hierarchical Clustering

Reviews of data clustering methods [16] show that algorithms responsible for natural group discovery fall into two categories:

1. **Partitional:** Simultaneous separation of the data into clusters without the knowledge of a hierarchical structure. K-means [17] is an example of a partitional algorithm.

2. **Hierarchical:** Search for nested clusters, establishing parallelism with a hierarchical structure. This exploration can be agglomerative, separating each object into its individual cluster and joining those who have a higher similarity, or divisive, grouping all points on a single cluster and iteratively partitioning it. Merging two clusters together can use three types of linkage:

   (a) **Single Linkage:** The similarity between two clusters is the similarity between the two most similar points in that cluster (lower distance). Has the property of being local, being that it ignores outliers in the clusters and its overall structure.

(b) **Complete Linkage:** The similarity between two clusters is the similarity between the two less similar points in that cluster (most distance). Has the property of not being local since the decisions that affect the merging can depend on the overall structure of the cluster.

(c) **Average Linkage:** The similarity between two clusters corresponds to the average similarity between each member of one cluster and each member of the other.

From the previous categories, information retrieval using clustering approaches for a better understanding of software architecture or modules is usually done through the use of Hierarchical Clustering [18] [19] [20]. The referred authors use this type of clustering when there is no capability to know the number of clusters that the software system will be divided into. Moreover, these types of systems have a natural hierarchical structure, being that the existence of modules and submodules is a common software practice. Another challenge, that depends on the complexity of the problem to be solved, is defining the similarity measure by which the software components are coupled with and that will be used in the clustering algorithm.

In our approach, the class information retrieved from the analysis of the monolithic code needs to be grouped and the number of resulting clusters will be unknown. This study shows that hierarchical clustering is the optimal method to structure the resulting data.

2.4 Domain-specific languages

As written by Arie Van Deursen et al. [21] a *Domain-specific Language* (DSL) is a short language that is applied to a particular domain, offering a high amount of expression by the use of abstractions. This type of programming languages have the characteristic of being *declarative*, being associated with specification languages and end-user programming, having the possibility to create applications or generate libraries. Examples of DSL's advantages are:

- Ability to recognize the expression of a solution as a language with an abstraction related to the domain.
- Support of domain level optimizations, including reuse of core domain knowledge.
- Enables testing with high coverage.

Its disadvantages include:

- Cost associated with the development and management of a DSL, including its design and scope analysis.
- Lack of efficiency when compared to regular programming languages.

A DSL is normally developed following three steps:

1. Clearly describe the domain and its information, gathering all of it by building semantic concepts and properties.

2. Implement the semantic approach through a library and build a compiler to translate the desired DSL programs into library invocations.

3. Write and compile the DSL programs for the intended applications.

2.4.1 DML

An example of a DSL that was built on the context of developing web application architectures is DML, as presented by J. Cachopo et al. [4]. To solve issues that are present in web applications, such as the persistence of states, the concurrent execution of processes and core domain verifications, a domain modeling language was built. This language has the property of being perfectly integrable with an usual Java web application (representing classes and its relationships with a similar syntax) and allowing developers to focus their attention on designing the system in an object-oriented fashion. The relationships between the domain entities of the system are bidirectional and its operations accomplished with transaction semantics by relying on a software transactional memory implementation to ensure atomicity and isolation.

The domain entities described by DML have the key property of representing the structural aspect of the domain, being that its behavior is not in its context. As seen on **Fig. 2.4**, the syntax is straightforward and Java-like. The **relation** keyword defines a relationship, its existing roles and subsequent multiplicity. This code is then compiled, generating two class types:

1. **Base Class:** Represents the structural information of the class, having the correspondent getters, setters and relationship management methods, responsible for keeping these relationships up to date.

2. **Controllable base class extension:** Extends the base class with no predefined implementation, allowing complete manipulation by the developer to add any functionality/methods desired.

2.5 Existing Microservice Migration Architectures

Although recent studies show that there is a lack of support tools [22] for the automation of microservice migration, in the current state of the art, there are some approaches that propose some tools.

```
class PetStore;

class NamedEntity {
  String name;
  String description;
}
class Category extends NamedEntity {
  String categoryId;
}
relation PetStoreHasCategories {
    PetStore playsRole petStore;
    Category playsRole category {
        multiplicity *;
        indexed by categoryId;
    }
}
```

```
abstract class PetStore_Base {
  int getCategoryCount() {...}
  boolean hasAnyCategory() {...}
  boolean hasCategory(Category category) {...}
  Category getCategory(String categoryId) {...}
  Set<Category> getCategorySet() {...}
  void addCategory(Category category) {...}
  void removeCategory(Category category) {...}
}
abstract class NamedEntity_Base {
  String getName() {...}
  void setName(String name) {...}
  String getDescription() {...}
  void setDescription(String desc) {...}
}
abstract class Category_Base extends NamedEntity {
  String getCategoryId() {...}
  void setCategoryId(String id) {...}
  PetStore getPetStore() {...}
  void setPetStore(PetStore petStore) {...}
  boolean hasPetStore() {...}
  void removePetStore() {...}
}
class PetStore extends PetStore_Base {}
class NamedEntity extends NamedEntity_Base {}
class Category extends Category_Base {}
```

Figure 2.4: DML's domain expression example and its equivalent Java code generated by the compiler that implements the domain model, from [4]

G. Mazlami et al. [23] present a suggestion of ideal microservice candidates following an algorithmic approach subsequent to its extraction from a version control system (e.g., GitHub[7]) of a monolithic application. It starts with the definition of a monolith *M* as a tuple (*Cm*, *Hm*, *Dm*), being that *Cm* represents the class files associated with the monolith, *Hm* the change history that contains the sequence of change events that happened in the code base and *Dm* the developers that participated in the development of the application. The history *Hm* allows the authors to have access to the changes in class files and its respective timestamp and developer ownership. With such notation, the solution has the following workflow:

- **Construction**: The monolith in question is represented as a graph G = (E,V), being that every vertice v ϵ V is a class and every edge e ϵ E is a weight associated to the coupling level between classes. This weight is calculated through 3 individual strategies:
  - **Logical Coupling**: Based on the principle that software is designed with the intention of grouping its members that have the same reason to be changed (*Single Responsibility Principle [24]*), the classes that are changed as a group will therefore reside in the same microservice. The weight is then calculated between the pairs of classes, resulting in the value 1 if both classes were committed together or 0 otherwise.

[7] https://github.com/

- **Semantic Coupling**: Based on the domain-driven design concept of bounded contexts that separate the domain into several contexts, establishing ideal microservice boundaries, the source code of the class files are analyzed in order to group those that contain information about the same domain entities. This information retrieval process from the classes uses the caught expressions (as method or variable names) as input to an inverse document frequency method [25], computing two vectors. The weight is computed by applying statistical measures, being that its value between two classes is the cosine between the two vectors.
- **Contributor Coupling**: Starting with the team organization of the development, ownership of class files can also lead to a coupling measure, calculating such measure through the intersection of developers that contributed to both classes in question. This information is accessible by having a change history containing a set of changed files associated to the developers that changed them.

- **Clustering**: The graph G is partitioned into microservice candidates using graph clustering. The approach chosen benefits weights with a high value to group the classes associated into the same service. The algorithm chosen is the MST (Minimum Spanning Tree) Clustering algorithm [26], inverting all weights on the edges of graph G. Then the edges of the MST are calculated by Kruskal [27], sorted and reversed, resulting on the lowest coupling to be first on the list. Subsequently these edges are deleted iteratively to a maximum number given as input. Finally, clusters of classes that are not small enough are broken even further.

This formal extraction model was evaluated with the open-source code of 21 projects and in terms of two key factors:

1. **Performance:** Assessing the execution times for the coupling strategies, commit count, contributor count and size of the code, the returned values show that for the average project, performance is satisfactory, but the semantic coupling strategy is directly affected by the size of the code repository, as the Term Frequency–Inverse Document Frequency (TF-IDF) similarity is computed by scanning each line of the class files. The performance values in seconds for semantic coupling range from 545 seconds to 643179 seconds, while contributor coupling has its values between 1 and 1136 seconds and logical coupling from 6 to 540 seconds.

2. **Quality:** The metrics used for quality assessment are entirely related to object-oriented design and microservices, being such:

   (a) **Team size reduction ratio metric:** Improvement in terms of team size and complexity, being that with a scale from 0 to 1, 1 represents that no reduction was made. With median values below 0.5, this approach shows that for most of its applications, the size of the teams was

reduced in half. As one of the advantages of a microservices architecture is a team alignment to the services, reducing the previous established teams in two is clearly satisfactory, considering it ideal if the number of team members are no bigger than 10 and averagely, from 6 to 8, as written by Neil Gehani[8].

(b) **Average domain redundancy metric:** This metric pretends to establish a parallelism with bounded contexts and responsibility encapsulation, showing in a scale from 0 to 1 the extent of redundancy between the created services, being that 1 characterizes the service as a duplicate. In 4 of the 6 experiments the median was lower than 0.25, showing that less than a quarter of the domain is duplicated amongst all services. For the rest of the experiments the median was below 0.5 but not as good as the other 4. This happens due to front-end developers contributing to the same files with no regard for its domain.

As a microservices architecture has the intention of setting hard boundaries for domain logic, the use of the contributor coupling alone gives a value of redundancy that is too high (around 0.4), but the authors classify this problem as being dependent on the developers of open-source projects and their ability to be domain-oriented.

Not all existing architecture migration models are automatic as the one previously presented and usually, the process of moving from a monolithic to a microservices architecture is an incremental and manual process of decomposition with the use of several techniques, as written by Zhamak Dehghani[9] and Christian Posta[10]. This process, with the final objective of creating the ideal microservices architecture that encapsulate business capabilities and have an independent life-cycle (build, test and release), goes through the following steps:

1. Begin the separation of the system with the capabilities that are significantly decoupled from the monolith and with no need for a data repository. The intent is to introduce simple change with a high rate of success.

2. The capabilities of the system that have no dependencies towards the monolith should be decoupled first (dependencies in the inverse direction, from the monolith to the services, do not interfere). In case the creation of the services forcefully implies an access to the monolith, an API should be created and accessed through an anti-corruption layer.

3. Having separated the most decoupled and with least dependencies, the harder concepts/capabilities should be reconstructed, identifying all the domain ideas behind them and joining them accordingly into services. Now, our objective is to reduce dependencies, specially data.

[8] https://techbeacon.com/want-develop-great-microservices-reorganize-your-team
[9] https://martinfowler.com/articles/break-monolith-into-microservices.html
[10] http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/

4. Decouple the data, creating a data repository for each service that is driven by its domain. This is, analyzing the schema and creating the persistent data tables needed for only that service/-functionality, iteratively. In some cases, business capabilities can require queries that span across multiple services, this can be solved by using API composition, making the application do the join and not the database, or Command Query Responsibility Segregation (CQRS), by maintaining materialized views[11].

5. Develop API's that are easily used by developers and redirect all the front-end applications to them.

Additional to this workflow, tools to identify the dependencies in the monolith and the overall code structure should be used, in order to get the ideal services with the highest level of coupling and maximum capability encapsulation. An example of a tool with these functionalities is Structure101[12].

[11] http://microservices.io/patterns/data/database-per-service.html
[12] https://structure101.com/

3

Solution Architecture

Contents

The core objective of this dissertation is to build a set of tools that support developers on the process of migrating from a monolithic to a microservices architecture. The solution relies on the identification of contexts from the priorly presented domain-driven design, applying clustering techniques in retrieved static analysis data from applications developed using the domain modeling language (DML). The information captured from the system will not only help identify the different services that compose the monolith, but also to improve the quality of the migration itself.

3.1 Modular Context

As presented before, *Domain-Driven Design* explores concepts that are relevant when trying to develop a modular approach to a system. One way to modularize our code or to define modular boundaries is by separating our system into bounded contexts. This concept is usually presented with team-based views, being that it is used with a huge focus on teamwork, management and work division, making sure software development in complex domains is made easier. The objective here is to identify code boundaries that have the properties of high cohesion and low coupling, appealing to the modular side of bounded contexts. The encapsulation of these contexts can be done manually through experts or by applying automatic techniques (as presented in Section 2.5). In a typical decomposition, the experts have a shared understanding of the system and how the modules work together, identifying the right contexts, but in a manual fashion. The objective of this solution is to assist and improve the process underlying this identification of contexts by providing more information to the experts that is relevant to its context decomposition, supporting an **informed decision-making** process.

One challenge in following this approach is the non-existing link between modularity and transactionality. Being that the modules of a system do not have a fixed transactional pattern relationship, and so, the separation of the system into contexts also has no direct translation in terms of transactions, which may result on the generation of transactional contexts that do not fit into the set microservices.

3.2 Transactional Context

To achieve consistency on data that belonged to modules that are now partitioned, the resulting bounded contexts will need to be related in terms of the transactional dependencies between them. As a monolithic schema establishes a single transactional boundary, its subsequent separation will also lead to fragmenting this boundary. We explore this fragmentation by analyzing the controllers of a software application and the relationship that it has with the domain classes, and as controllers usually encapsulate functionality, the classes accessed by them can also mimic this encapsulation. For a broader and future consistency analysis, we may infer what type of transactions are associated with these new

bounded contexts using the following methods:

- Bounded contexts have the ability to interact with each other, meaning we can classify the interaction strategies between them and **see how they can be interpreted as inter-transactional behavior between microservices** while managing the overall consistency of the system. These contexts have the crucial property of withholding particular business values in its boundaries, implying that **inside a context**, the operations on data in its independent databases is **transactional** with an **ACID consistency**. Moreover, the existing interactions between the contexts will represent the communication that exists between the business capabilities of the system, making operations on data **among contexts** be **non-transactional**, meaning its consistency will need to be relaxed.

- Using a more general approach, conclusions can be taken by extracting information regarding the read and write operations performed on the system, but as this information is not going to be available for this iteration of the solution, future work will be presented at this scope.

This transactional behavior between bounded contexts is important on the characterization of the relationships that exist between them, as we extract information regarding the classes generated from **DML** and its methods that manipulate the database.

3.3 Context information retrieval process

Our approach begins by analyzing monolithic systems that model their domains using DML. The monoliths used as the core of this study are Java-based applications that use the *Fénix Framework*, implying that the structure of the domain will be purely represented by DML and that the mapping of objects to the database will not be done directly but through an Object-relational mapper (ORM), implemented by the *Fénix Framework*. This means that the database will be hidden, leading to the programmer not being able to use aggregation or other database related functions and having to write and read objects through the use of *Set* and *Get* methods, which are generated into the *Base classes*. To retrieve information regarding new modules and transaction boundaries that constitute a decomposition into services, static analysis to the code regarding the methods responsible for writing and reading objects is the initial step. Therefore, there is a need to visualize and understand the typical architecture of a web-application with these characteristics, as presented in **Fig.3.1**:

The *Domain* layer will be defined by DML and the *Data Access* layer will be the *Base Classes* generated, the ORM ordinary data flux will start in the *Controllers*, passing through the *Interface* to reach the *Domain* objects. A pattern on the flow of data can be identified as seen on **Fig. 3.2**. As a controller reaches the different domain objects through the invocation of several methods, this flow corresponds to the flow found in a service belonging to a microservices architecture. Therefore the accesses from

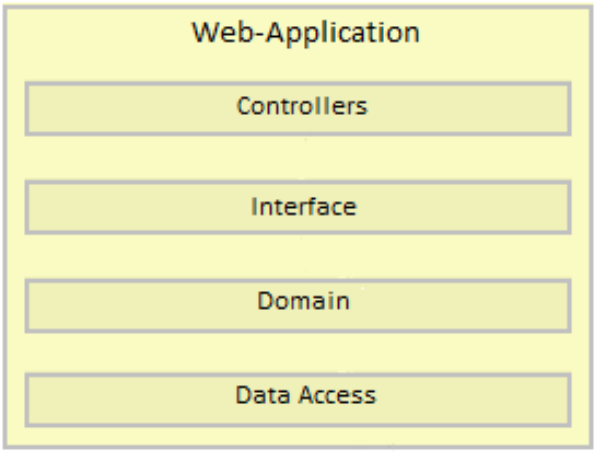

Figure 3.1: Typical architecture of a web-application that uses DML to model its domain.

the controller to the objects correspond to a **transactional context (1)** that is similar to a transactional boundary defined by a service, creating a boundary on the classes that the controllers accesses. On the other hand, we can also identify a **modular context (2)**, which represents the way those objects are associated with each other and their respective controllers, being that there is an intention to see them as a context/aggregate. As we are identifying modular contexts from the control-flow of the controllers, we are defining new modules inside the applications transactional contexts, retrieving modularity from transactionality. To identify a cluster of operations that defines a module accessed by different controllers, graph-based clustering algorithms are used. The intention behind defining a modular context is the grouping of similar classes into modules that form the boundaries of services in a microservices architecture. The information retrieved from these algorithms represent clusters of classes that are similar in terms of a certain measure. To group these classes into the desired modules through a clustering approach, the following is provided:

1. **Similarity Measure:** To achieve the property of high cohesion and low coupling between classes of the same cluster, the measure used for similarity between classes must reflect this aspect. Hence the similarity being calculated through an attribution of a weight to the relationship between two classes, defined by the subsequent formula:

$$W_{C1C2} = \frac{N_{Ctrl}(C1C2)}{N_{Ctrl}(C1)} \tag{3.1}$$

   Given two domain classes, $C1$ and $C2$, the weight from class $C1$ to class $C2$ is the quotient between the number of controllers in which the invocation tree has both $C1$ and $C2$ as nodes ($N_{Ctrl}(C1C2)$) and the total number of controllers in which the invocation tree has $C1$ as a node ($N_{Ctrl}(C1)$). When applying this measure to a clustering approach, in an ideal decomposition, the classes in the same cluster are accessed by the same controllers. One characteristic of the

similarity measure we are defining is that it is naturally asymmetric. This means that the value of this measure from class $C1$ to $C2$ is not the same as from $C2$ to $C1$, as we are computing the ratio according to the total number of controllers of only one of the classes.

2. **Clustering Method:** To cluster the classes, *Hierarchical Clustering* will be used, computing the results for its methods of *Single Linkage*, *Complete Linkage* and *Average Linkage*.

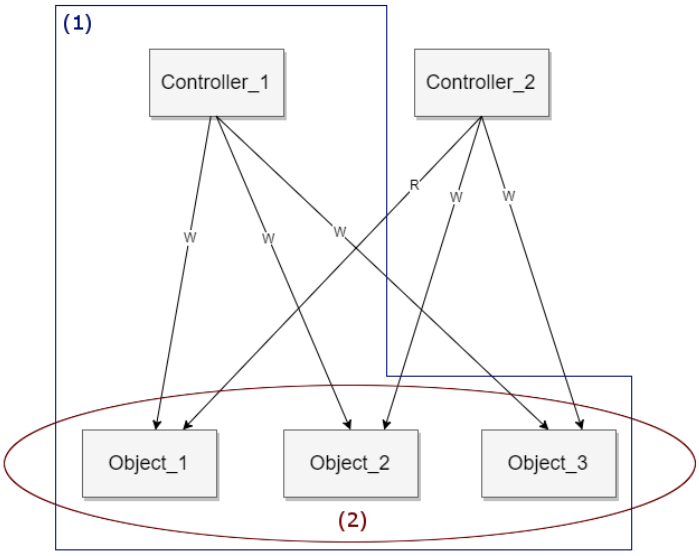

Figure 3.2: Data operations (reads and writes) from the controllers to the objects, identifying a **(1) transactional context** and a **(2) modular context**.

The overview of the process to examine the monolithic application can be seen as the architecture presented in **Fig.3.3**, being executed in the following manner:

1. Running static analysis on the code regarding the controllers of the web application, using the tool **Java-Callgraph** previously described. As the control flow patterns identified are part of the *Fénix Framework*, the generation of a call graph associated with the controllers and all its called methods will represent the invocation of the domain objects generated by the DML at the end the method call stack (through get and set methods), being able to not only capture the domain classes accessed by each controller but also to divide and manipulate the output of this tool into two different contexts:

   (a) **Transactional Contexts Partial Graphs:** Partial graphs that represent the method invocations from the controller to the domain objects. Each graph will represent the transactional boundary associated to the respective controller analyzed. This graph has as a main node the controller under analysis, being that the nodes linked to them are the classes that it accesses. Its edges represent an invocation of such class, being able to portrait that there is

an access to the object of such class, but not discriminating if it is a read or a write. This information is retrieved using a regular expression implementation on the text representation given by the java-callgraph tool, outputting a serialized python dictionary that contains all the classes that the controller under analysis accesses.

(b) **Modular Context Dendrograms:** The modular context of the application is derived from its transactional context. This is done by executing hierarchical clustering on the classes obtained from the concatenation of all the partial graphs into a system-level transactional graph. The result of the clustering will be presented as a Dendrogram, as the several possible cuts can define alternative modular contexts depending on the desired cohesion between clusters, also allowing an exploratory approach.

2. Through the creation of a Graphic User Interface (GUI), the user is prompted to select a cut on the dendrogram, enabling the creation of several combinations of cluster formations depending on the desired cohesion between classes. Moreover, the interface should offer the possibility of adding already existing clusters together, providing an extra level of customization if the user wants to assess the results with certain tweaks, for example, with two particular clusters joined together into one.

3. The result of a particular cut of the dendrogram can be accessed and analyzed through a browser.

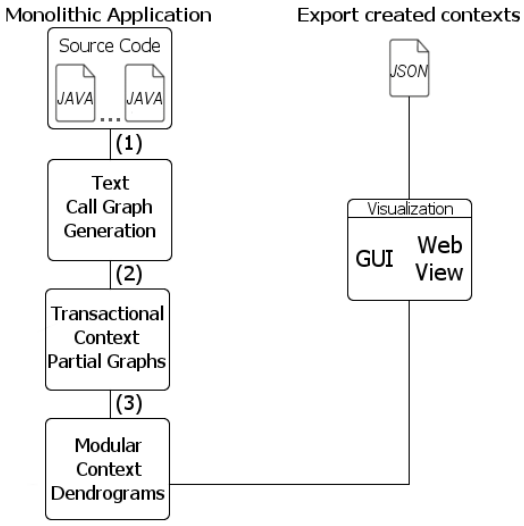

Figure 3.3: Data flow schema of the tools to be developed.
(1): Static analysis using Java-Callgraph.
(2): Regex implementation on the text-based call graph, returning controller-classes relationships.
(3): Hierarchical clustering application.

The end result of applying these techniques originates a .json file that contains the final cluster formation

defined by the user.

3.4 Particular examples and challenges

In a complex application, the approaches used to identify modular and transactional contexts are not trivial, meaning that they require the solving of additional problems and/or addition of restrictions, for example:

- **Bounded Context Sharing (Fig. 3.4):** In the current figure, we face the issue of creating bounded contexts that are accessed by different controllers. This problem is solved by choosing the highest values between classes calculated by the similarity measure of the clustering method, as the bigger the value, bigger the probability of the classes involved being modularly related. The similarity measure calculations among classes (**Table 3.1.**) are bidirectional and give us that the highest values of similarity between two distinct classes are 1, corresponding to the weight from A,B and C,D (note that the values of B,A and D,C have different values). This suggests the creation of two bounded contexts, one that includes classes A,B and other the classes C,D, showing the application of the basic similarity measure calculation between classes that is used in the clustering method.

- As a software application is naturally hierarchical and its data asymmetrical, we cluster it using hierarchical clustering and an asymmetric measure. We did further experimentations on the dendrograms generated by our applications using both asymmetric and symmetric measures and the difference between them were negligible. Therefore, we are opting for the integration of the asymmetrical measure on our clustering, as we consider that it better represents the true similarity between the classes.

Table 3.1: Weights between classes of **Fig. 3.4** given by the equation (8)

	A	**B**	**C**	**D**
A	1	1	0	0
B	0.3(3)	1	0	0.3(3)
C	0	0	1	1
D	0	0.50	0.50	1

3.5 Method Implementation

This section contains the code and details regarding the method proposed and represented by the data flow presented in Figure. 3.3.

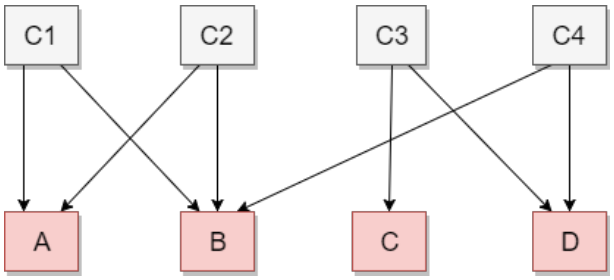

Figure 3.4: Bounded context sharing problem represented with controllers (C1...C4) and classes (A...D).

With the execution of the bytecode libraries (Java-Callgraph) on the classes of the web applications LdoD[1] and Blended Workflow[2], our intention is to use the generated text call graph representation to extract meaningful information about the services that the monolith can be divided into and its relevant boundaries. The objective of this approach is establishing a parallelism with a Model-View-Controller (MVC) architectural pattern. In a monolithic architecture, a layered approach has as a component a presentation layer, responsible for the user logic and interaction. The user makes a request to a controller, in the presentation layer, which manipulates the model and updates the view returned to the user with useful information. This means that the requests the user makes to the controllers are actually calls to independent functionalities encapsulated in those same controllers. A request from the user creates a series of method calls through domain classes that manipulate the model to achieve the desired view. These domain classes are the target for clustering into services and discovering new boundaries, being that the analysis of the control flow of the controllers will be our key focus.

The behavior analysis of the controllers was made using Python, mainly the libraries NetworkX[3] and Scipy[4], having the following workflow:

1. **Regex Parsing:** The text file containing the call graph of the application and obtained from Java-Callgraph is parsed by iteratively checking each controller and the methods called by it. On the next step it will search the methods inside the previously caught methods until the method stack ends. When a domain class is accessed for its get or set method, this class is added to the list of domain classes of the controller under analysis. To this implementation we also added the possibility of knowing the abstract classes of the classes that are being analyzed, as in some cases, the methods corresponding to inherited classes of this scope were not caught correctly (represented with **S:** *class1 class2*, being that *class1* extends *class2*). These controller-class relationships are then serialized into a file in form of a python dictionary so there is no performance impact of rerunning the static analysis procedure every time we need to analyze the system. Note that one of the limitations of our solution comes from this statically retrieved information, as Java 8 streams

[1] https://github.com/socialsoftware/edition
[2] https://github.com/socialsoftware/blended-workflow
[3] https://networkx.github.io/
[4] https://www.scipy.org/

are not caught by our tool and certain parts of the code manipulate domain classes with this type of manipulation.

2. **Graph Construction:** With the set of all the unique classes used by the controllers, a graph for each controller will be created, establishing edges with the domain classes that it interacts with (A.2).

3. **Graph Joining:** All graphs are composed into a single general graph (A.3), representing the system's controllers behavior related to its domain.

4. **Similarity Matrix Creation:** A square matrix with the size of the number of classes is created to represent the similarity between all domain classes to be used in the clustering method (A.4). In the similarity matrix *Sij*, each *i,j* position represents the computation of the similarity measure defined in Equation 3.1, in which *i* represents $C1$ and *j* represents $C2$.

5. **Hierarchical Clustering:** Application of a hierarchical clustering method with all linkage types (average, single and complete) (A.5). The similarity measure is obtained from the similarity matrix previously calculated.

6. **Dendrogram Cut:** The clustering results are shown with the use of a dendrogram. A certain cut that maximizes the cluster cohesion of the classes is made (A.6), but can also be freely chosen by the user using an interface.

7. **Statistic Generation:** After the cluster generation, a .txt file is generated for all linkage types containing details about the clusters formed (A.7), the controllers that access each cluster and the percentage of classes inside each cluster that each controller accesses.

8. **GUI:** Parallel to this implementation, an user interface was developed that allows the user to cut the dendrogram at its desired height and assessing the resulting clusters on a browser. This interface is explained in greater detail later in the document.

In order to keep refining the results and to assess different types of data, several iterations of this workflow were made:

3.5.1 Controller granularity

The simpler way to analyze the behavior of the controllers is running Java-callgraph on them as-is, this is, on controller classes that have several controller methods. This approach will not make any alterations to the already existing code, as seen on **Fig. 3.5**.

The execution of our workflow on controllers with several controller methods has one main disadvantage. If each controller class does not represent each functionality of the system then the domain classes

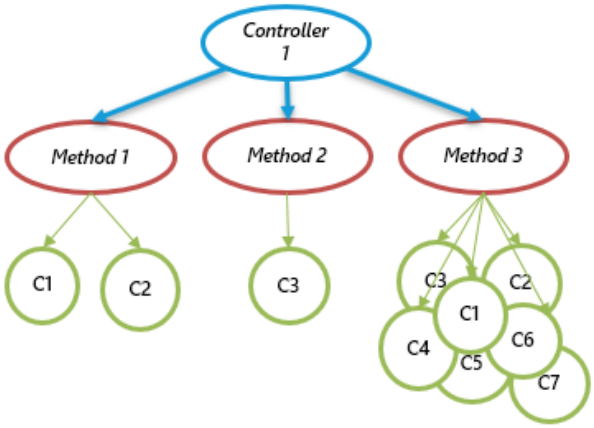

Figure 3.5: Method distribution inside the controllers of the applications on the first iteration.

access information can be inaccurate. An example is if controller classes contain a controller method that accesses all the domain entities, for instance, a class that implements some type of housekeeping functionality, resulting in an inaccurate association of the domain classes within the same functionality. Our objective is to separate functionality and therefore separate the domain classes accurately. An approach to avoid this problem is separating each controller class so that each copy of it has only one method, as seen on **Fig.** 3.6.

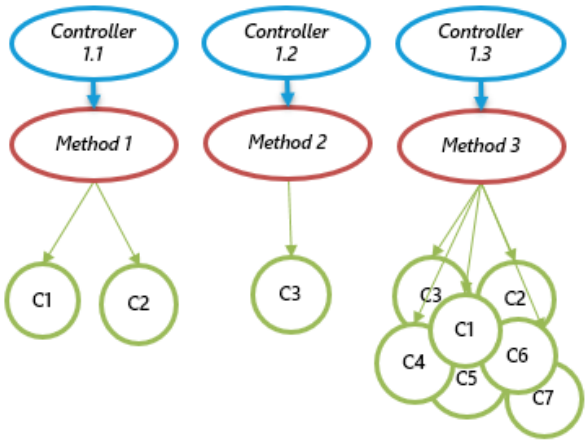

Figure 3.6: Method distribution inside the controllers of the applications on the second iteration.

4

Results

Contents

The results presented in this section are the consequence of the application of the workflow described in section 3.5 to the web applications LdoD and Blended Workflow. The parsing of the text-representation of the call graph generates a system-level graph representation that can be visually represented, illustrated by its relationships between controllers and classes. Using this information the similarity measure is calculated (given by equation 3.1) and given as input to the hierarchical clustering algorithm. This produces a dendrogram with several possible cuts, depending on the desired number of clusters and similarity thresholds.

The analysis of cluster formation on dendrograms is a method based on exploration that is also context-dependent. Our approach is based on the behavior of the controllers domain access with the intention of creating tight service boundaries, establishing trade-offs between cuts of low dissimilarity, this is, cutting where the intra-cluster distance is minimal and cuts where the inter-cluster distance is bigger, which we are going to verify that they represent the system's functionality.

Additionally, note the different dendrograms resulting from the application of different linkage types, are going to provide the same conclusions, as the minimum cut that maximizes intra-cluster similarity are always going to produce the same number of clusters and in general, its differences are not significant. Therefore, **in the dendrogram linkage assessment, we are interpreting the average linkage type**, as the inter-cluster similarity of the clusters have the most consistent value, as it is calculated with averages and not maximum or minimum values (as single or complete).

Furthermore, **the results for the 1st iteration with the unchanged controllers are also not going to be assessed**, as its conclusions are not interesting, actually, the decomposition of the controller classes into their set of methods led us to results from which stronger conclusions are drawn while confirming the inaccuracy of the previous iteration.

4.1 2nd Iteration - Split Controllers

The separation of each controller class into copies that only have one method allows the visualization of the class behavior of each method, establishing a parallelism with functionality encapsulation. There was a benefit of using this approach on both applications, as both similarity matrices had values that **better represented the domain access of the system by the controllers**, not allowing that certain methods that can access all classes of a system generalize the behavior of the entire controller. A proper representation of the system allows that the functionality encapsulation of the clusters, that result from a dendrogram cut, is also well represented. This is supported by the results of the application of the method-splitting technique to the controllers, that resulted in more homogeneous clusters.

4.1.1 LdoD

The LdoD archive[1] is a collaborative digital archive that contains the Book of Disquiet, originally written by Fernando Pessoa. As this book is fragmented, several interpretations of it can be done by reading it in different sequences. This application, in addition to allowing the reading and comparison of four editions of the book, it also enables users to create their own virtual editions based on the existing fragments.

LdoD originally contains 152 controllers and 56 domain classes, being that 37 of the controllers do not make contact with the domain (24% of the systems controllers).

Taking into account the behavior of the controllers, our intention was to capture every method call inside them using a call graph, as certain calls represent the writing or reading of domain objects. This was done using Java-Callgraph paired up with a Regex implementation. The used static analysis techniques can lead to certain inaccuracy on class detection, mainly due to the fact that collection types are only known on runtime and the tool incompatibility with Java 8 streams. The writing and reading of domain objects can be done using collections with no instance iteration, being that the compiler does not statically know the type that is inside it. On the latter case, the calls inside a stream are not caught by the tool. Having this kind of issue, an intermediate representation in form of a Python dictionary was created that can be manipulated by the user, allowing the addition or removal of classes to a controller. On LdoD, its manual inspection revealed that 7/115 (6%) controllers had missing classes.

The controller-class relationship graph is represented in **Fig .4.1**, as the controllers are the circles in blue and the classes in red.

The radial composition of the graph allows us to interpret some outliers exist, seeing that they are only accessed by a few controllers and also that there is also an agglomerate of classes that are being accessed by the same controllers (bottom of the graph), which represents the text elements that have low dissimilarity between them.

The application of hierarchical clustering using the previously defined controller access similarity measure, calculated from the graph representation, originates a dendrogram:

- **Average Linkage:** As represented in **Fig. 4.2**, the dendrogram mimics the cluster effect of the graph representation in **Fig. 4.1**, as there is no dissimilarity between some of the Text class elements (e.g. Rend, PbText, LbText, NoteText) and also some outliers with big dissimilarities (e.g. NullEdition or PhysNote). To interpret these results, several levels of granularity can be chosen, as different height cuts can be explored. Opting for a medium dissimilarity cut of the dendrogram at 1.00, there is the creation of the following 19 clusters:

  Cluster 0: ['AddText', 'AltText', 'AltTextWeight', 'AnnexNote', 'DelText', 'GapText', 'LbText', 'Note-

[1] https://ldod.uc.pt

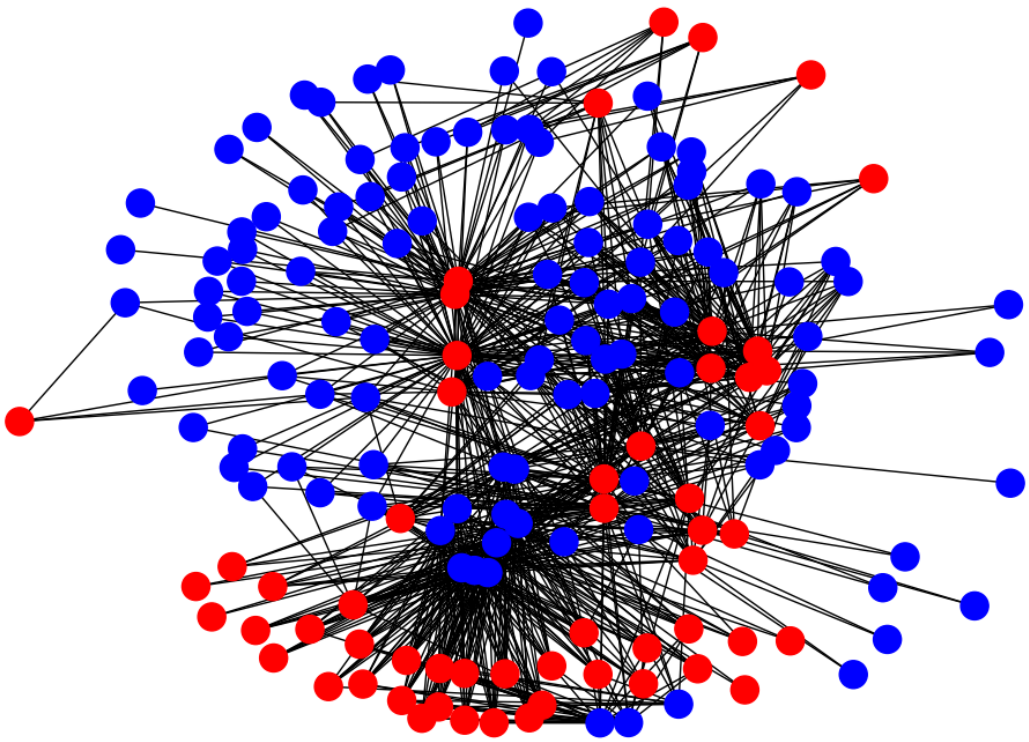

Figure 4.1: Domain class access by controllers in a graph representation for LdoD

Text', 'PbText', 'RdgGrpText', 'RdgText', 'Rend', 'SegText', 'SpaceText', 'SubstText', 'TextPortion', 'UnclearText']

Cluster 1: ['Annotation', 'Category', 'Range', 'Section', 'Tag', 'Taxonomy', 'VirtualEditionInter']

Cluster 2: ['AppText']

Cluster 3: ['Dimensions', 'HandNote', 'ParagraphText', 'PrintedSource']

Cluster 4: ['Edition', 'FragInter', 'Fragment']

Cluster 5: ['ExpertEdition']

Cluster 6: ['ExpertEditionInter', 'Heteronym', 'LdoDDate']

Cluster 7: ['Facsimile', 'RefText', 'Surface']

Cluster 8: ['LdoD', 'LdoDUser', 'VirtualEdition']

Cluster 9: ['ManuscriptSource']

Cluster 10: ['Member']

Cluster 11: ['NullEdition']

Cluster 12: ['NullHeteronym']

Cluster 13: ['PhysNote']

Cluster 14: ['RecommendationWeights']

Cluster 15: ['RegistrationToken', 'Role', 'UserConnection']

Cluster 16: ['SimpleText']

Cluster 17: ['Source', 'SourceInter']

Cluster 18: ['TypeNote']

The resulting clusters are supposed to be service candidates for a microservices architecture and to do so, these must encapsulate business capabilities and also have size restrictions, being that there should not be clusters composed by only one class or as big as the whole system, as it does not portray a good service division. Knowing the functionality of the application, the clusters that do not make sense on the topic of context separation are the singleton clusters (composed by only one class), as the others do make plausible service candidates. The **Clusters 2, 5, 9, 10, 11, 12, 13, 14, 16 and 18**, as singletons, are no plausible candidates, as with the cut given, they are too dissimilar from the bigger clusters. To solve this issue, the clusters can later be joined in the GUI or by cutting at a higher cut-off height.

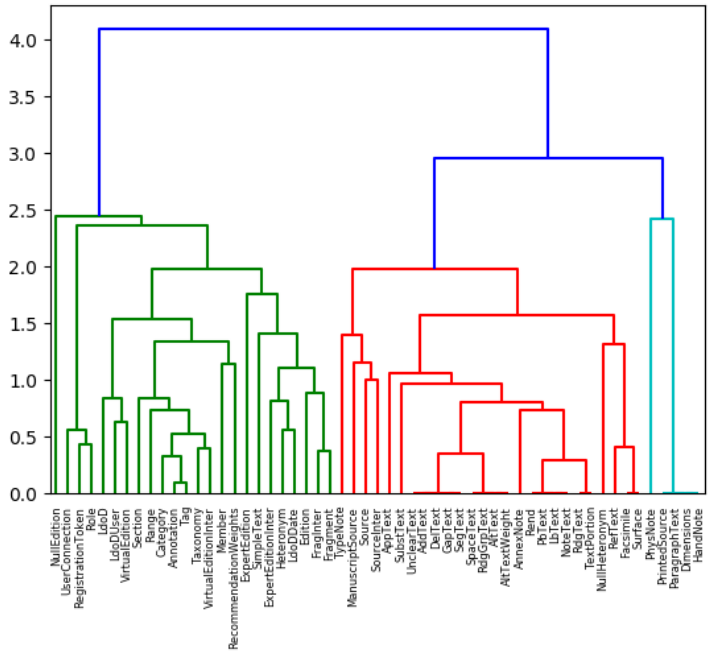

Figure 4.2: Average linkage dendrogram of applying hierarchical clustering to the LdoD application with split controllers

The assessment of the dendrogram shows that the information retrieved from the controllers regarding its domain class reachability is enough to make a division of the system into contexts, as the ideal microservice context separation is found with the trade-offs from different cuts. Let us draw some further conclusions from the retrieved clusters:

1. LdoD is an application responsible for storing a digital archive of a book, having several interpretations of it according to several editors. The classes representing the text, shared by all editors, results in an aggregation that contains 17 classes (**Cluster 0**) at this cut-off height. If it correctly encapsulates a business capability of the system, the number of classes in the clusters is irrelevant.

2. The content of the remaining clusters will further be analyzed in comparison to the expert decomposition.

4.1.2 Blended Workflow

The Blended Workflow application has the intention of fusing an activity-based approach with a goal-based approach by allowing its end users to follow a predefined behavior on the work they are developing.

Blended Workflow originally has 94 controllers and 46 domain classes. 1 of the 94 controllers has no domain access as it handles the exceptions of the system.

The static analysis results of applying Java-Callgraph/Regex to the Blended Workflow source code led to a bigger inaccuracy than LdoD, having the need to add domain classes to 24 out of 93 (26%) controllers. The manual code inspection for the assessment of the static analysis results revealed that Java 8 streams were being used abundantly in the code, specially for methods responsible for cleaning/deleting objects. The methods that contained such streams were missing classes, while the remaining produced accurate results.

The graph of the controller-class relationship is illustrated by **Fig. 4.3**, with the controllers in blue and the domain classes in red. The graph disposition shows that all the domain classes are reached by the controllers.

The existing connections between controllers and classes allows us to create a similarity matrix and execute an hierarchical clustering algorithm, generating the following linkage type dendrogram:

- **Average Linkage:** Defined by the dendrogram in **Fig. 4.4**, we can see some groups of classes that have little to no dissimilarity between them. To assess the cluster results we perform a cut with a height of 0.9, creating 15 clusters:

  Cluster 0: ['Activity', 'BlendedWorkflow', 'Entity', 'Goal', 'Product', 'Specification']

  Cluster 0: ['Activity', 'ActivityModel']

  Cluster 1: ['ActivityWorkItem', 'PostWorkItemArgument', 'PreWorkItemArgument', 'ProductInstance', 'WorkItem', 'WorkItemArgument']

  Cluster 2: ['AndCondition', 'BinaryExpression', 'BoolComparison', 'Comparison', 'Expression', 'NotCondition', 'OrCondition']

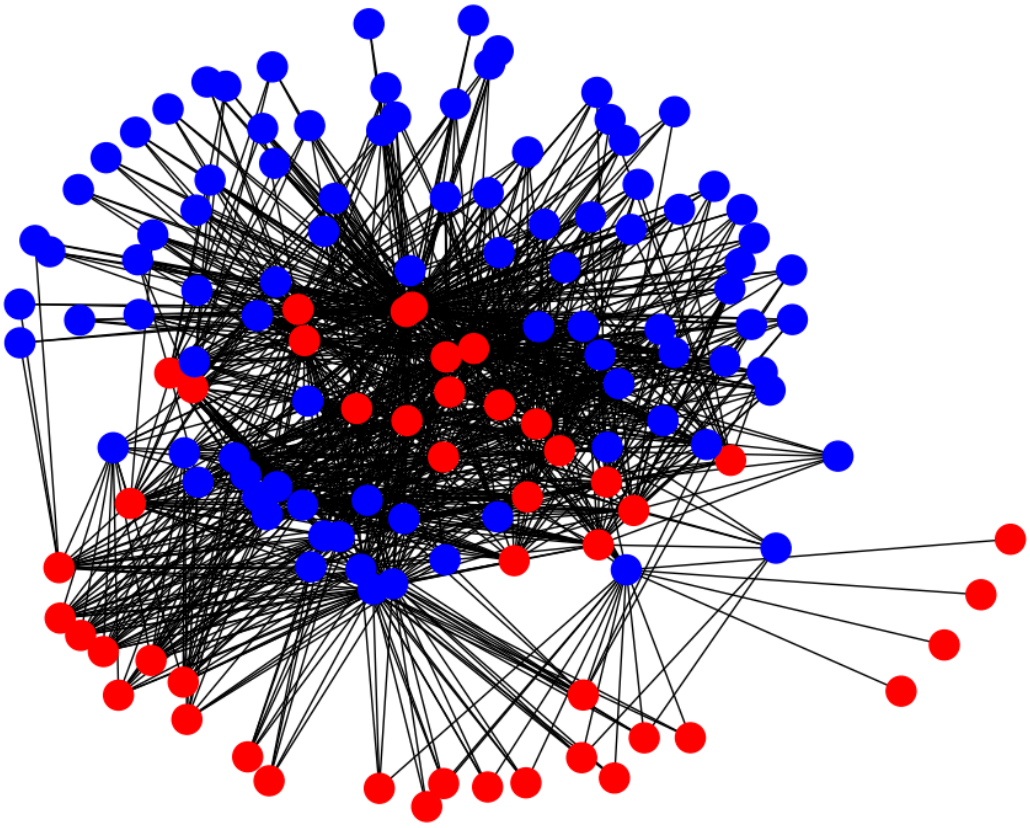

Figure 4.3: Domain class access by controllers in a graph representation for Blended Workflow

Cluster 3: ['Attribute', 'BlendedWorkflow', 'Condition', 'ConditionModel', 'DataModel', 'Entity', 'Path', 'RelationBW', 'Specification']

Cluster 4: ['AttributeBoolCondition', 'AttributeValueExpression']

Cluster 5: ['AttributeInstance']

Cluster 6: ['Cardinality']

Cluster 7: ['DefAttributeCondition', 'DefEntityCondition', 'DefPathCondition', 'DefProductCondition', 'MulCondition', 'Rule']

Cluster 8: ['Dependence', 'Product']

Cluster 9: ['EntityInstance']

Cluster 10: ['FalseCondition', 'NumberLiteral', 'StringLiteral', 'TrueCondition']

Cluster 11: ['Goal', 'GoalModel']

Cluster 12: ['GoalWorkItem']

Cluster 13: ['RelationInstance']

Cluster 14: ['WorkflowInstance']

The challenges that arise from this cut are similar to the ones exposed in LdoD, with the appearance of 6 singleton clusters (**Cluster 5, 6, 9, 12, 13 and 14**). The other clusters make plausible service candidates.

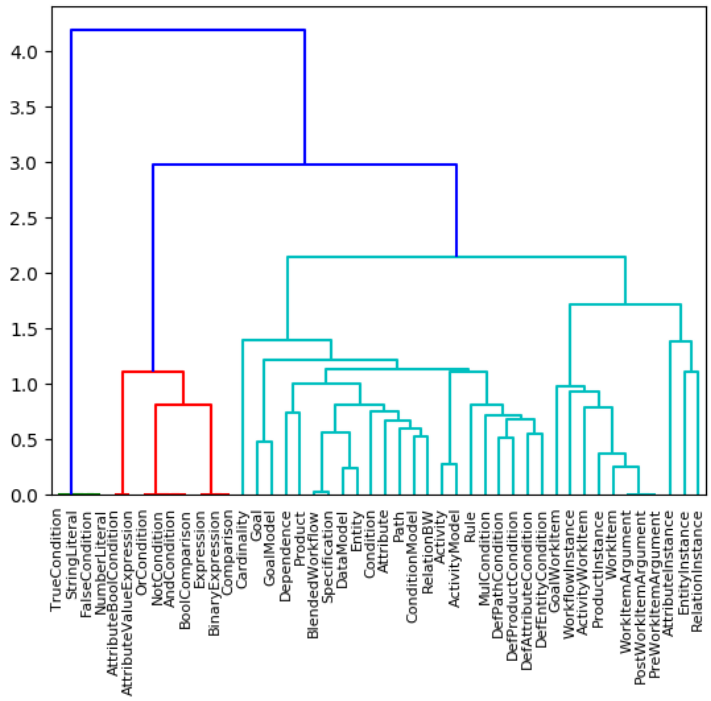

Figure 4.4: Average linkage dendrogram of applying hierarchical clustering to the Blended Workflow application with split controllers

In comparison to the same iteration for LdoD, this approach has the same core problem, which is the presence of singleton clusters. However, this is the opposite problem of the first iteration (not presented), in which most of the classes had little-to-no dissimilarity between them, meaning that any cut would have clusters with a huge size. Between those two cases, we prefer the finer granularity, as we can easily fix it with a higher cut on the dendrogram or the composition between several cuts.

To formally analyze the cluster results, several metrics are used, as presented in the subsequent sections.

4.2 GUI

In this section the resulting interface is presented along with all its capabilities, including the results obtained and how to manipulate them properly for a successful identification of microservices.

4.2.1 GUI visualization

The cutting of the dendrograms and posterior cluster formations must be manipulated by the user, as proposed in section **3.6**. The interface was developed using TkInter[2] in Python and the cluster visualization using D3.js[3], being that the first is illustrated by **Fig. 4.5**.

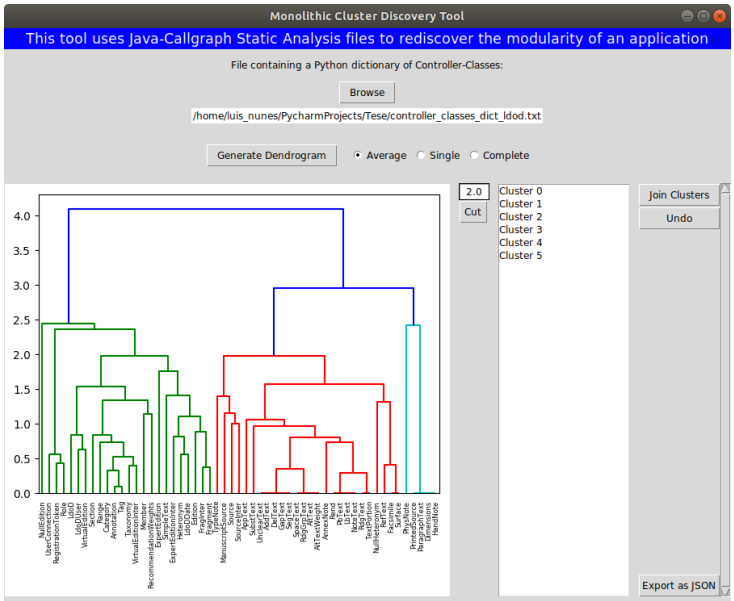

Figure 4.5: View of the graphic interface developed for the user to manually cut the dendrogram and manipulate clusters

The different existing buttons provide the following functionalities:

- **Browse:** Allows the user to select the serialized dictionary with the information of the controllers and its accessed domain classes.
- **Generate Dendrogram:** The selection of one mutually exclusive option for the type of linkage (Average, Single or Complete) will generate the corresponding dendrogram image on the left-hand side of the screen.
- **Cut:** Input value for the height at which the dendrogram is cut. This cut will result in several cluster formations that are listed in the list view on the right-hand side. This button will also open a web-browser view for customization that will be fully described later.
- **Join Clusters:** After analyzing the clusters on the web page, there is the option to join one or more clusters. This can be done through the selection of the clusters (Pressing CTRL + left-clicking the clusters) and by subsequently pressing the button. This generates a new web-browser visualization with the updated clusters.

[2]https://wiki.python.org/moin/TkInter
[3]https://d3js.org/

- **Undo:** Grants the ability to undo the last cluster join, returning to the previous state of clusters.
- **Export as JSON:** Exports the current formation of clusters as a .json file.

As described above, cutting the dendrogram or joining its clusters results in a dynamic visualization in a web page, as illustrated by **Fig.** 4.6. This generates a cluster HTML view in which all clusters are mapped as different nodes and their size depends on the domain classes that it clusters. The notation of the cluster information is given by ClusterN(D, C), being that N is a number attributed to the cluster, D the number of domain classes and C the number of controllers. This means that Cluster0(26, 20) is cluster number 0, composed by 26 domain classes and 20 controllers. This visualization provides the following functionalities:

1. Dragging of the clusters to the desired location.
2. Hovering of the cluster nodes shows the domain classes that compose the corresponding cluster.
3. Clicking a node creates a focus-view on that node and its links, allowing a better visualization by changing the opacity of the rest of the system links to 0. Moreover it also presents a table with the controllers that access the selected cluster.
4. Clicking a link shows the controllers commonly accessed by both clusters and on the end of that list, on a red background, the controllers that represent the difference between them (in a way to assess the functionality differences). This can be seen in **Fig.** 4.7, where we can assess the controllers shared (3) and its relevant difference (8) with a red background.

4.2.2 GUI workflow

Now that the functionalities given by the interface are clear, we present how the user can optimally use it to retrieve the ideal results for a microservices architecture. After cutting the dendrogram and obtaining the cluster formations presented above, the results can be further manipulated by using the functionality of joining clusters while being able to see the domain classes of each clusters and the controllers shared between them. To do so, let us explain the exploratory process behind the interpretation of a software application when migrating from a monolithic architecture, using our transactional context based tool (these approaches are also reinforced by M. Kalske [28] and Z.Dehghani[4]):

1. Start with a maximum similarity cut at 0.01. The clusters that are not singletons are composed by classes that are all accessed by exactly the same controllers. This says that in the entirety of the application, these classes are always used together, indicating a very large probability of being in the same service. Moreover, if this cut has classes that are accessed by more than 90% of

[4]https://martinfowler.com/articles/break-monolith-into-microservices.html

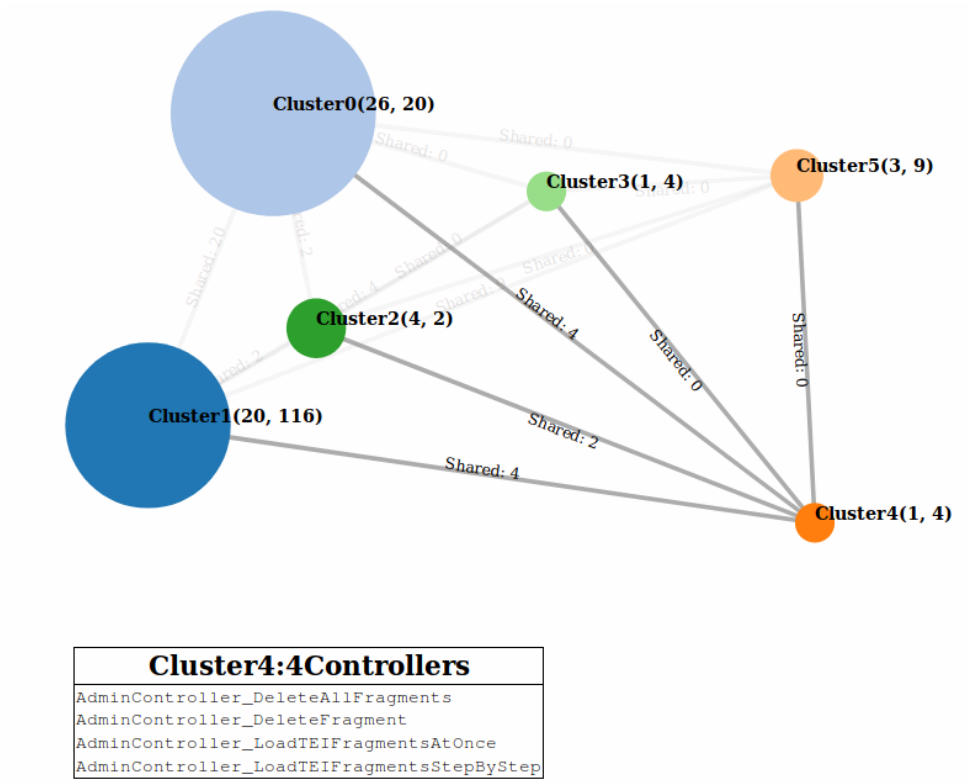

Figure 4.6: D3.js view of the clusters formed by a dendrogram cut presented in a HTML page of the web-browser.

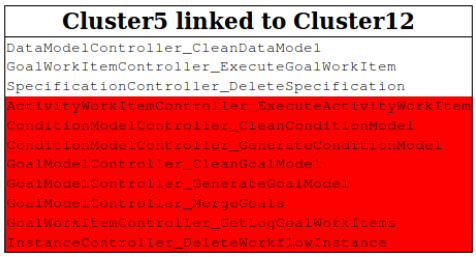

Figure 4.7: Graphic difference between shared and difference in controllers between two clusters.

the controllers, it is also likely that this is part of a **critical microservice**, this is, a group of core classes that can be susceptible to overloading, as they are accessed by the whole system.

2. Continue the process by cutting the dendrogram at high cut-off height as to divide the system into 2-4 clusters. This allows the visualization of a bigger picture of the application and the classes that compose them.

3. Finally, cut at an intermediate level (this part is exploratory, existing the need to test several levels of granularity/cuts) and assess the smaller clusters that are now sub-clusters of ones originated from the previous cut, analyzing if they match smaller functionalities, and if so, services.

4.3 Metric evaluation

The clusters retrieved from the hierarchical clustering algorithm that uses as a similarity measure the ratio of controllers that access each class is now going to be evaluated in terms of the quality of the decompositions formed. As supported by the evaluation of other approaches for software architecture recovery [29] [30], an internal and external assessment of the clusters will be made.

Internal evaluation is defined by a group of intrinsic cluster measures that will assess the direct results from the dendrogram while the **external evaluation** is composed by the assessment of the results by an expert, this is, the domain-knowing professional that will use the GUI and the system as a whole in order to produce and explore the application under test, comparing the results with a previously produced collection.

As the second iteration of the solution (where controllers have a single method) gives the results which better mimic the behavior of the system, only the possible cluster formations of this iteration for both systems will be assessed.

4.3.1 Internal evaluation

Several common methods for automatically cutting a dendrogram exist, but there is no golden rule for it, as they depend entirely on the data of our system. Our approach is based on the **manual cutting of the dendrogram** and the posterior **exploration of the cluster decompositions** according to its key factors related to a microservices architecture. This way, we pretend to evaluate the quality of what we find to be the ideal cut that explores the cluster formations. We want to avoid formations composed by either **many singleton clusters** or **one big cluster**.

The current state of the art microservice migration and analysis [31] reinforces that the microservices should be small in size, but this size does not directly reflect into a number, as the knowledge that one service can be split into several others is only acquired when we investigate the business capabilities of the service and its concrete profile of operation. Therefore, there is no single solution for the decomposition, as to observe in the subsequent internal evaluation, where we analyze the clusters according to several cuts. In this analysis we do not consider some of the previous issues, but in the final conclusions, we draw the limitations of this type of analysis.

To perform an internal evaluation of the clustering results for our applications, we perform an ad hoc analysis with metrics proposed by us (except for the silhouette score), being those:

1. **Number of Singleton clusters**, being that having more than 2 singleton clusters is considered negative. Considering a final microservice architecture with clear functional boundaries established, it is likely that there are not two services in which its content is only one class.

2. **Average cluster size**, should be between 2 and half of the size of the system. Even with a cluster size inside this range, there is also a dependency regarding the number of instances inside them, due to data consistency and concurrency. Establishing transactional locks on the operations that will occur in these clusters can make us rethink the size of the clusters, because if they have too many instances, it negatively impacts the system's performance.

3. **Maximum cluster size**, should not be bigger than half of the size of the system. As an application does not usually have just one functionality, there should be at least two clusters, representing this maximum size boundary.

4. **Silhouette score**, given by Equation 4.3, where a represents the mean intra-cluster distance (Equation 4.1: distance between object o_i and the remaining objects in the cluster) and b the mean nearest-cluster distance (Equation 4.2: distance between object o_i and the objects of the neighbor cluster). This score ranges its values from -1 to 1, representing incorrect clustering (samples on wrong clusters) and highly dense clustering respectively. For every object in a cluster, when this score is high (closer to 1) the mean intra-cluster distance is going to be smaller than the mean nearest-cluster distance, implying that the object is well classified. This metric creates a parallelism with the overall coupling of the clusters of the system, as our objective was to obtain a high intra-cluster similarity and a low inter-cluster similarity, so the partition between clusters is well defined. The silhouette value evaluates exactly this information. In the scope of our problem we present the silhouette score for the entire cluster data of the presented cut, meaning that we have to calculate the silhouette of each cluster by averaging the score of all the objects inside them and then average the score of all the clusters, reaching a value for the entire dataset.

$$a(o_i) = \frac{1}{|C_A| - 1} \sum_{o_j \epsilon C_A, o_j \neq o_i} d(o_i, o_j) \tag{4.1}$$

$$b(o_i) = min_{C_b \neq C_A} \frac{1}{|C_B|} \sum_{o_j \epsilon C_B} d(o_i, o_j) \tag{4.2}$$

$$Silhouette(o_i) = \frac{(b(o_i) - a(o_i))}{max(a(o_i), b(o_i))} \tag{4.3}$$

There are several evaluation techniques [32] (as MoJoFM or c2c) appropriate for cluster assessment, but most require a ground-truth level decomposition. As we pretend to help developers in separating a system that has not yet been separated, a prior ground-truth decomposition is not available for assessment.

In the following evaluations, we consider several cuts:

In both tables, four cuts are presented for the scope of both applications:

Table 4.1: Internal evaluation results for the second iteration of the solution on LdoD.

	Cut(0.01)	Cut(1.5)	Cut(2.5)	Cut(3.5)
Number of singleton clusters	34	3	0	0
Maximum size of clusters	5	18	26	31
Average size of clusters	1.38	5.0	18.33	27.5
Number of retrieved clusters	40	11	3	2
Silhouette coefficient	0.38	0.48	0.55	0.56

Table 4.2: Internal evaluation results for the second iteration of the solution on Blended Workflow.

	Cut(0.01)	Cut(1.1)	Cut(2.0)	Cut(3.0)
Number of singleton clusters	30	4	0	0
Maximum size of clusters	4	11	22	42
Average size of clusters	1.31	4.18	11.5	23
Number of retrieved clusters	35	11	4	2
Silhouette coefficient	0.35	0.47	0.57	0.55

1. The maximization of intra-cluster similarity, given by a cut with the lowest value possible. Note that we do not want to cut at a 0.0 cut-off height, as this will not group the classes into clusters that have maximum similarity but create one cluster for each domain class.

2. A cut at an intermediate height, establishing an attempt to make a trade-off between granularity cluster similarity.

3. Two high valued cuts that try to split the system into its main components, usually with a size of 2-4 clusters.

From the results of table 4.1 we can see that LdoD, in its maximum intra-cluster similarity (0.01 cut), has a number of singleton clusters that is almost the number of retrieved clusters, resulting in a low average cluster size and the worst silhouette coefficient of all cuts. This happens due to the fact that in the application, most classes are accessed by different controllers, being that clustering will produce these clusters as different. When there is such problem, the cut should be higher, as we assess in the next cut. With a cut at the height of 1.5 we can see that the number of singleton clusters drastically decreases to 3 and there is also the presence of a cluster with 18 classes. Moreover, the silhouette coefficient increases, showing that we are moving towards a more appropriate clustering. Finally, the high-level views of the system represented by the last cuts present 0 singleton clusters, defining 3 and 2 clusters respectively. The 2.5 cut gives us maximum size of 26 and average of 18.33 while the 3.5 cut returns 31 and 27.5. Note also that, the higher the cut, the better the silhouette coefficient, indicating a more explicit boundary on the clustering data, even that between the two last cuts this difference is minimal. To know if these big-sized clusters actually represent functionalities of the system, these cuts should be made in the GUI and manually assessed by the workflow presented in Section **4.3.2**.

Table 4.2 presents the internal metrics for Blended Workflow. We can see that the intra-cluster

maximum similarity cut presents results similar to ones obtained from LdoD (table 4.1), with a number of singleton clusters of 30 our of 35 retrieved clusters, an average size of 1.31 and a maximum size of 4. The intermediate cut in 1.0 shows significant progress regarding the first one, reducing the number of singleton clusters to 4 and increasing the silhouette coefficient. Lastly, the high cuts for a 4-tier and 2-tier division is presented, resulting in no singleton clusters. For the 2.0 cut we have a maximum size of 22 and an average of 11.5 while for the 3.0 cut there is a maximum size of 42 and an average of 42. Being that the Blended Workflow application is composed by 46 domain classes, the ranges by us defined on the ad hoc metrics fit the 2.0 cut but not the 3.0. Note also that, the silhouette coefficient increases with the height of the cut until 2.0, but after that, it starts to decrease. This shows us that the optimal clustering of this application is around the dendrogram cut-off height of 2.0.

4.3.2 External evaluation

This type of evaluation consists on the creation of a golden solution of ideal clusters by an expert that is compared to the results obtained. In order to analyze our methods thoroughly we decided to also analyze the results of decomposing the system through another software architecture analysis tool, Structure101[5], which is frequently suggested in the approaches on how to migrate a monolith to a microservices architecture. This allows us to do a better analysis of our tool because we can compare its results with the ones provided by a tool that is actually used in the industry. Additionally, since Structure101 follows a decomposition based on the structure of the modules, its results provide a good reference model to compare with our approach that is based on transactional contexts, allowing us to validate whether approaches based on transactional contexts provide significantly different results and if they constitute a better base to help on the division of a monolith into a microservices system. Overall, we have done the decomposition of the system according to three different approaches:

- **Hierarchical Clustering of Transactional Contexts**: Representing the method we proposed, this approach captures the relationships that exist between classes through the transactional context of the controllers. These relationships are then clustered through hierarchical clustering to form modules, which are classified as services. The applied cuts were presented in the prior internal evaluation, corresponding to the height 1.5 and 2.5 for LdoD, 1.1 and 2.0 for Blended Workflow.

- **Structure101**: As a tool for software architecture discovery and extensive analysis, it has a feature for the creation of cohesive clusters. The clusters are formed by the number of common dependencies between its members and its level on the Levelized Structure Map (LSM), which aims for a package-level analysis. In this tool, dependencies are classified as a direct relationship between two items (e.g. the extension of a class by another) or code relationships that exist between them

[5]https://structure101.com/

(e.g. a method of one class calls a method of another class). We classify a forward dependency as a usage dependency between a class C1 and another class C2 (e.g. method call), being that C1 knows implementation details about C2. A backward dependency is when a class C2 is being used by C1, not being aware of its implementation. The clusters formed are never tangled, meaning that they do not have any cyclic dependencies between them. This kind of relationship is also suitable for our tool's scope, being discussed later in the topic of future work. The decompositions resulting from applying this tool in the two case studies can be seen in the **Annex B**, **Fig.** B.3 and **Fig.** B.4.

- **Expert Decomposition**: A decomposition made by the expert that developed the applications. As the applications have not been migrated into a microservices architecture, this division represents the expert analysis on the structural/modular behavior of the system in terms of the domain and its relationships. An important note is that the expert does not claim that these divisions are necessarily correct, therefore they do not classify as a ground truth service-wise. However, it is important in the assessment of the tools to compare its results with the expert insights about the system decomposition, as it contains his domain knowledge of the applications. The decomposition for both applications can be seen in the **Annex B**, **Fig.**B.1 and **Fig.**B.2

Usually, in the computation of evaluation metrics following the use of clustering, these are done in a pairwise fashion, as presented by S.Walde [33] in her PhD thesis. The most appropriate metrics for our approach are the subsequently calculated pairwise precision, recall and f-score, given by Equations 4.4, 4.5 and 4.6 respectively:

$$Pairwise\,Precision = \frac{\text{Number of common pairs}}{\text{Number of pairs in the clustering}} \tag{4.4}$$

$$Pairwise\,Recall = \frac{\text{Number of common pairs}}{\text{Number of pairs in the expert classification}} \tag{4.5}$$

$$F\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4.6}$$

First, we assess the reasoning behind using these metrics:

Precision: The computation involved in this evaluation metric means that we have to assess all possible pairs of classes and consider it a True Positive (common pair) when the pair is present together in our clustering approach and also in the expert decomposition. The number of pairs in the decomposition is the combination of pairs available in the given cluster that is being evaluated. Note that the combination of pairs in the clustering approach depends on the particular clustering and not on the total number of entities being clustered, as it is going to be the sum of the number of combinations of pairs of the several given clusters, not the pair combination of the entire system

or expert decomposition.

Recall: This metric also includes the computation all possible pairs of classes and common pairs. But, contrary to precision, we want to divide by the total number of pairs that exist in the decompositions created by the expert instead of the decomposition from our clustering approach. In the internal evaluation phase we established that the existence of singleton clusters in our approach was to be avoidable, as single classes rarely depict system functionalities. With the computation of a recall metric, a decomposition with many singleton clusters will have a high precision but a low recall if the golden solution does not have singletons. By computing such metric we are establishing a technique to measure such characteristic.

F-Score: The harmonic mean between precision and recall.

The pairwise assessment of these metrics for system-level cuts is made on the Tables 4.3, 4.4 and 4.5:

Table 4.3: External evaluation result of the pairwise precision metric on both applications using both tools (Transactional Contexts and Structure101).

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 2.5 Cut Blended - 2.0 Cut	73% (445/611)	67% (220/328)
Structure101	58% (166/285)	56% (145/261)

Table 4.4: External evaluation result of the pairwise recall metric on both applications using both tools (Transactional Contexts and Structure101).

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 2.5 Cut Blended - 2.0 Cut	48% (445/926)	48% (220/462)
Structure101	18% (166/926)	31% (144/462)

Table 4.5: External evaluation result of the F-Score metric on both applications using both tools (Transactional Contexts and Structure101).

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 2.5 Cut Blended - 2.0 Cut	0.58	0.56
Structure101	0.27	0.40

Table 4.3: We can see that for both applications, our pairwise precision value is higher, in fact, 15% for LdoD and 11% for Blended Workflow. This shows that, at a system-level (low number of clusters/high cut), our tool presents clusters that have a greater similarity to the expert decomposition

than a proposed architectural tool for analyzing the structure of the code. As we are assessing precision, this metric takes into account our relevant pairs according to our clustering approach, validating that our approach is indeed returning better results than Structure101.

Table 4.4: This table shows that, similarly to the precision metric, our values are also higher, 30% for LdoD and 16% for Blended Workflow. As the recall metric assesses the relevant pairs according to all pairs from the expert decomposition, if the number of clusters of the expert decomposition is lower, the recall values will also be lower, as the combination of pairs among the clusters are higher, resulting in a bigger denominator. This is exactly what happens here, as our approach and Structure101 return a bigger number of clusters, its combinations will be smaller, resulting in a higher precision value than recall. In these divisions, Structure101 returns 6 singleton clusters for LdoD and as the expert decomposition has no singleton clusters, we can see that the recall value is 40% lower than the precision value, showing a singleton cluster existence penalization.

Table 4.5: The metric presented in this table joins the conclusions of both previous tables, as F-Score defines the combination of precision and recall into a single measure. Since both previous calculated metrics had already showed superior results, this one will also reflect such aspect, with a difference of 31% for LdoD and 16% for Blended Workflow.

Let us now analyze such cluster formations and the corresponding expert decompositions, starting with LdoD(2.5 cut):

Cluster 0: ['AddText', 'AltText', 'AltTextWeight', 'AnnexNote', 'AppText', 'DelText', 'Facsimile', 'GapText', 'LbText', 'ManuscriptSource', 'NoteText', 'NullHeteronym', 'PbText', 'RdgGrpText', 'RdgText', 'RefText', 'Rend', 'SegText', 'Source', 'SourceInter', 'SpaceText', 'SubstText', 'Surface', 'TextPortion', 'TypeNote', 'UnclearText']

Cluster 1: ['Annotation', 'Category', 'Edition', 'ExpertEdition', 'ExpertEditionInter', 'FragInter', 'Fragment', 'Heteronym', 'LdoD', 'LdoDDate', 'LdoDUser', 'Member', 'NullEdition', 'Range', 'RecommendationWeights', 'RegistrationToken', 'Role', 'Section', 'SimpleText', 'Tag', 'Taxonomy', 'UserConnection', 'VirtualEdition', 'VirtualEditionInter']

Cluster 2: ['Dimensions', 'HandNote', 'ParagraphText', 'PhysNote', 'PrintedSource']

Analyzing the following clusters in comparison to the expert decomposition (Fig. B.1), we can see that firstly, **Cluster 0** is a sub-cluster of the Document cluster defined by the expert, capturing all of the text related classes (that contain "Text" in the class name) and other classes that directly manage these class elements. Assessing the controllers of this cluster, we can see that out of the existing 115 controllers, this cluster is accessed by 20 of those, being that most of them are reads. 18/20 out of the 20 controllers are used in the functionalities of exporting, loading or reading, reinforcing that this cluster

is used mostly in a read-based access to the text elements. Furthermore, **Cluster 1**, represents a join of the Virtual Edition, Authentication and a small subset of the Document cluster of the expert. This represents the main functionality of the system, being used by 100% of the controllers and existing a need to split this cluster into smaller parts, so better conclusions can be drawn. So many functionalities are gathered in this cluster due to the high value of the cut and the difference of cluster access to the clusters. In terms of classes being used together by controllers and grouped in clusters, these types of high cuts will capture the major groups of classes that have the most difference in controller access between them. These groups can form functionalities if the controller access separation is ideal, but in the case of this separation not happening, there is a need for a more thorough analysis and this is where our tool can help and what is discussed in the section 4.3.2.A. Finally, **Cluster 2** represents a small sub-cluster of the Document expert decomposition that is used by the controllers in a different way than the rest of the system, given that it is separated from the rest of the clusters. These 5 classes are only used by 4 controllers, being those responsible for the deletion and loading of fragments. These exclusive classes are only used in this context as deleting deletes every object of every class in the system and the loading of fragments implies the use of these classes.

The following clusters are the ones returned by the Structure101 approach for LdoD:

Cluster 0: ['AppText', 'AnnexNote', 'Dimensions', 'ExpertEdition', 'ExpertEditionInter', 'HandNote', 'LbText', 'RdgText', 'RefText', 'NoteText', 'PrintedSource', 'RecommendationWeights', 'RdgGrpText', 'Section', 'SourceInter']

Cluster 1: ['DelText']

Cluster 2: ['GapText']

Cluster 3: ['UnclearText']

Cluster 4: ['SpaceText']

Cluster 5: ['AddText']

Cluster 6: ['AdHocCategory', 'Annotation', 'LdoD', 'Taxonomy', 'VirtualEdition', 'VirtualEditionInter', 'NullHeteronym']

Cluster 7: ['SegText', 'AltText', 'AltTextWeight'],

Cluster 8: ['ParagraphText', 'SubstText', 'PbText', 'PhysNote', 'Rend', 'TextPortion', 'TypeNote', 'ManuScriptSource', 'SimpleText']

Cluster 9: ['Range']

Cluster 10: ['ExpertEdition', 'Fragment', 'Heteronym', 'FragInter', 'Category', 'LdoDDate', 'Tag', 'LdoDUser', 'Source', 'Surface', 'Edition', 'Facsimile', 'Member', 'RegistrationToken', 'Role', 'UserConnection']

From the given clusters, 6 out of 10 are singletons, which have no value to the decomposition and are the cause of its low recall value. **Cluster 0** presents a sub-cluster of the document cluster but with a mispositioned class, RecommendationWeights, which is positioned in this cluster as it has the same forward dependencies as the other classes. **Cluster 6** is a sub-cluster of the Virtual Edition decomposition but with another mispositioned class, the NullHeteronym. **Cluster 8** is a sub-cluster of the Document expert decomposition and **Cluster 10** is a join of all the expert decomposition, containing classes from every defined boundary.

Switching to the Blended Workflow application, our system-level cut at 2.0 height retrieves the following clusters:

Cluster 0: ['Activity', 'ActivityModel', 'Attribute', 'BlendedWorkflow', 'Cardinality', 'Condition', 'ConditionModel', 'DataModel', 'DefAttributeCondition', 'DefEntityCondition', 'DefPathCondition', 'DefProductCondition', 'Dependence', 'Entity', 'Goal', 'GoalModel', 'MulCondition', 'Path', 'Product', 'RelationBW', 'Rule', 'Specification']

Cluster 1: ['ActivityWorkItem', 'AttributeInstance', 'EntityInstance', 'GoalWorkItem', 'PostWorkItemArgument', 'PreWorkItemArgument', 'ProductInstance', 'RelationInstance', 'WorkItem', 'WorkItemArgument', 'WorkflowInstance']

Cluster 2: ['AndCondition', 'AttributeBoolCondition', 'AttributeValueExpression', 'BinaryExpression', 'BoolComparison', 'Comparison', 'Expression', 'NotCondition', 'OrCondition']

Cluster 3: ['FalseCondition', 'NumberLiteral', 'StringLiteral', 'TrueCondition']

Examining the previous cluster formations, we can firstly see that **Cluster 0** contains the whole Models cluster of the expert with a sub-cluster of the Design cluster. Being that this cluster is accessed by 92/93 controllers of the system, we can classify it as being the one that deals with the whole aspect of the system and its main functionalities. Similarly to the issue of many functionalities in LdoD, we can see through the relevant dendrogram (**Fig. 4.4**) that reducing the height of the cut would start separating this cluster into several groups, which can make us reason about the clustering and high controller accesses. This assessment will be made in section 4.3.2.A. **Cluster 1** is a perfect match to the Execution cluster of the expert, where we can add that the controllers that access this cluster will also be execution-related and responsible for generating, executing and cleaning the existent models. **Cluster 2** represents a subset of the Design expert cluster, being that it captures subclasses of Condition, representing particular conditions applied to the entities of the system. Analyzing the controllers associated to these classes

we can see that they are used in the creation of rules, as they are enforcing a certain condition, and the cleaning and deletion of the models. Finally **Cluster 3** represents another subset of the Design cluster with other particular conditions. These are separated from **Cluster 2** as they are used exclusively on the creation of rules.

The Structure101 decomposition for this application was the following:

Cluster 0: ['BlendedWorkflow']

Cluster 1: ['AttributeBoolCondition', 'AttributeValueExpression']

Cluster 2: ['BinaryExpression', 'AndCondition', 'AttributeInstance', 'BoolComparison', 'FalseCondition', 'NumberLiteral', 'EntityInstance', 'Comparison', 'NotCondition', 'OrCondition', 'Expression', 'ProductInstance', 'StringLiteral', 'RelationInstance', 'WorkflowInstance', 'TrueCondition', 'PostWorkItemArgument', 'WorkItem']

Cluster 3: ['ActivityWorkItem', 'GoalWorkItem', 'PreWorkItemArgument', 'WorkItemArgument']

Cluster 4: ['Activity', 'ActivityModel', 'GoalModel', 'Goal', 'Rule', 'ConditionModel', 'Condition', 'DefAttributeCondition', 'DefEntityCondition', 'MulCondition']

Cluster 5: ['Cardinality', 'DefPathCondition', 'Dependence', 'RelationBW', 'Path', 'Specification', 'DataModel', 'Entity', 'Attribute', 'Product', 'DefProductCondition']

Similarly to the Structure101 division of LdoD, this decomposition also contains singleton clusters, in this case, one, which is denoted by its low recall value. Starting by **Cluster 0**, which is the singleton cluster, we can see that analyzing the relevant expert decomposition, the class BlendedWorkflow a direct connection to specification which is not shown here, but in fact, analyzing the code of the application, BlendedWorkflow is part of a Singleton design pattern, which can be a reason for this separation. **Cluster 1** expresses a subset of the conditions that deal with attributes, being part of a small cluster inside the Design cluster of the expert. **Cluster 2** shows a mix between the Execution expert decomposition and Design, mixing conditions and classes related to the WorkItem's. **Cluster 3** forms a sub-cluster of the Execution cluster, as their dependencies are similar. **Cluster 4** represents a sub-cluster of the Models expert decomposition while having Rule as a mispositioned class. Analyzing the structural dependencies of Rule, we can see that it does have dependencies towards Goal and Activity and that is why it is positioned in this cluster. Finally, **Cluster 5** represents the joining of Design and Models classes.

After assessing the system-level cuts on the dendrograms of the applications, we want to start an exploration phase, where we cut at intermediate levels and then compare resulting clusters to the expert decomposition. The comparison between several cluster decompositions, derived from different cuts, is one technique that allows us to reason about which clusters represent functionalities and which do

not, by using our tool and assessing the relevant controllers. This exploratory process is done in the subsequent section.

4.3.2.A Exploratory Analysis

The second step of the external evaluation is the assessment of the tool's capability of exploring sub-clusters and its potential to group classes into functionalities. For this type of analysis we are using the previous pairwise metrics defined in Equations 4.4, 4.5 and 4.6, as they calculate the relevant pairs according to both our approach and the expert, while penalizing singleton clusters in the recall metric. In this analysis we will not assess Structure101, as its cluster formations have already been shown and there are no additional decompositions available to draw conclusions. The Tables 4.6, 4.7 and 4.8 show the computation of these values for chosen intermediate cuts of the system:

Table 4.6: External evaluation result of the pairwise precision metric on both applications using our tool based on Transactional Contexts.

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 1.5 Cut Blended - 1.1 Cut	99% (233/234)	88% (125/142)

Table 4.7: External evaluation result of the pairwise recall metric on both applications using our tool based on Transactional Contexts.

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 1.5 Cut Blended - 1.1 Cut	25% (233/926)	27% (125/462)

Table 4.8: External evaluation result of the F-Score metric on both applications using our tool based on Transactional Contexts.

	LdoD (Expert)	**Blended Workflow (Expert)**
Transactional Context Ldod - 1.5 Cut Blended - 1.1 Cut	0.40	0.41

Table 4.6: The precision of the dendrogram cuts at an intermediate level are much higher than the ones at a system-level, even showing close to 100% for LdoD. This happens due to the fact that the smaller sub-clusters formed by the a smaller cut are most of them, subsets of the big clusters of the expert decomposition.

Table 4.7: Due to the appearance of singletons and the high number pair combination of the expert decomposition, the recall values are going to be lower than the system-level.

Table 4.8: The F-score measure is lower than the system-level cuts, showing that even with very high precision values, the presence of singletons and smaller clusters drastically harms the quality of the clusters when comparing them to a gold collection.

Starting by the LdoD application and assessing, for the transactional contexts, our finer approach (represented by the cut at 1.5 cut-off height), which has the following clusters:

Cluster 0: ['AddText', 'AltText', 'AltTextWeight', 'AnnexNote', 'AppText', 'DelText', 'GapText', 'LbText', 'NoteText', 'PbText', 'RdgGrpText', 'RdgText', 'Rend', 'SegText', 'SpaceText', 'SubstText', 'TextPortion', 'UnclearText']

Cluster 1: ['Annotation', 'Category', 'Member', 'Range', 'RecommendationWeights', 'Section', 'Tag', 'Taxonomy', 'VirtualEditionInter']

Cluster 2: ['Dimensions', 'HandNote', 'ParagraphText', 'PrintedSource']

Cluster 3: ['Edition', 'ExpertEditionInter', 'FragInter', 'Fragment', 'Heteronym', 'LdoDDate', 'SimpleText']

Cluster 4: ['ExpertEdition']

Cluster 5: ['Facsimile', 'NullHeteronym', 'RefText', 'Surface']

Cluster 6: ['LdoD', 'LdoDUser', 'VirtualEdition']

Cluster 7: ['ManuscriptSource', 'Source', 'SourceInter', 'TypeNote']

Cluster 8: ['NullEdition']

Cluster 9: ['PhysNote']

Cluster 10: ['RegistrationToken', 'Role', 'UserConnection']

As the smaller clusters are mostly going to be part of subsets of the expert decomposition, we will evaluate their content following a subset analysis, starting by the assessment of the clusters who are not a subset of the gold collection, which corresponds to **Cluster 6**, containing the only pair that is not in the expert decomposition. From the classes seen in this cluster (['LdoD', 'LdoDUser', 'VirtualEdition']), LdoDUser and VirtualEdition are classified as being used in two different scopes, Authentication and Virtual Edition management, respectively. LdoD is irrelevant as it is an immutable singleton, being able to be part of any cluster. Our tool classified these classes as being part of the same cluster as they appear together transactionally so, we are going to analyze these cases by assessing the D3.JS visualization. First, counting that the VirtualEdition class is out of the cluster of Virtual Edition management and LdoDUser is out of the Authentication cluster, the remaining elements of those clusters for the expert decomposition are perfectly represented in our division by **Cluster 1** and **Cluster 10**. Moreover, **Cluster**

6 is accessed by 105/115 (91%) controllers of the system, indicating that it could be a candidate for a critical microservice. On a microservices architecture, its components are built around the business capabilities of the system, and as this cluster captures the main classes of two functionalities, it is useful for understanding system functionality and the classes that are susceptible to being overloaded (critical microservice). If each of these classes represents the core class of a functionality, this means that they can be scaled in a different way than the rest of the functionality if we introduce them as an entry point service. As the controller access is 91%, every execution of the service can overload the service access-wise, existing the need to apply replication techniques. The expert added to this matter that the entities involved in these operations are created once and rarely changed. So, we can establish a parallelism between this cluster and an architectural pattern of a shared kernel, as every use of the system has to go through these classes, but not as a domain-sharing pattern, but as api-sharing, as the core classes we are analyzing constitute the first entry-points to the different functionalities that exist in this application. Analyzing the shared controllers between **Cluster 6** and the clusters of its supposed functionalities by the expert (**Cluster 1 and 10**), we can see that all the controllers of the Authentication are shared (9/9) and from the Virtual Edition management, most of them are also shared (47/49), concluding that LdoDUser and VirtualEdition are present in the context of these two functionalities but, extensively used as entry points for all functionalities. Comparing the clusters originated from this cut with the system-level cut at 2.5, we can see that the cluster that had mixed functionalities and that was hard to reason is now split into 6 clusters (**Cluster 1, 3, 4, 6, 8 and 10**), which allows us to explore the controllers that access each of these individual clusters instead of one big cluster. Assessing the granularity of the cluster divisions of the correct subsets, we use our tool to make the 1.5 height cut and assess the controllers that access the different clusters, the controllers that access each unique class and its relevant code, allowing us to draw the following conclusions:

Cluster 0: Describes a big-sized cluster of text elements that are used to write the content of fragments. This establishes a sub-cluster inside the expert's Document cluster that deals with text population of text portions.

Cluster 1: Describes the Virtual Edition management functionality.

Cluster 2: Sub-cluster inside the Document cluster, responsible for the loading of fragments.

Cluster 3: Sub-cluster of the Document cluster, having the functionality of manipulating fragments and its corresponding elements.

Cluster 5: Sub-cluster of the Document cluster responsible for exporting information and loading fragments.

Cluster 7: Sub-cluster of the Document cluster responsible for loading/deleting fragments and application searches.

Cluster 10: Describes the authentication functionality.

Finally, for the Blended Workflow application, the cluster decomposition of our chosen cut (1.1 cut-off height) is as follows:

Cluster 0: ['Activity', 'ActivityModel']

Cluster 1: ['ActivityWorkItem', 'GoalWorkItem', 'PostWorkItemArgument', 'PreWorkItemArgument', 'ProductInstance', 'WorkItem', 'WorkItemArgument', 'WorkflowInstance']

Cluster 2: ['AndCondition', 'AttributeBoolCondition', 'AttributeValueExpression', 'BinaryExpression', 'BoolComparison', 'Comparison', 'Expression', 'NotCondition', 'OrCondition']

Cluster 3: ['Attribute', 'BlendedWorkflow', 'Condition', 'ConditionModel', 'DataModel', 'Dependence', 'Entity', 'Path', 'Product', 'RelationBW', 'Specification']

Cluster 4: ['AttributeInstance']

Cluster 5: ['Cardinality']

Cluster 6: ['DefAttributeCondition', 'DefEntityCondition', 'DefPathCondition', 'DefProductCondition', 'MulCondition', 'Rule']

Cluster 7: ['EntityInstance']

Cluster 8: ['FalseCondition', 'NumberLiteral', 'StringLiteral', 'TrueCondition']

Cluster 9: ['Goal', 'GoalModel']

Cluster 10: ['RelationInstance']

The sub-clusters which are not exactly the same are **Clusters 3 and 6**. Starting by **Cluster 3**, the only reason this cluster is not considered correct (as a sub-cluster of the Design cluster), is because of the class ConditionModel, which the expert classifies as being part of the Models cluster, and not the Design. This class is introduced in this cluster as it is accessed by almost the same controllers as the remaining classes of the cluster. By analyzing the controllers that exclusively access ConditionModel we can see that this class is being used for the writing and reading of conditions related to the entities of the system. This means that the condition model is used in the data model, which is responsible for the management of all of the entities of the system, creating such entities and subsequently generating its conditions while also reading them when needed. As this is what a model should do, ConditionModel should be in the Models cluster and not in the Design cluster. For **Cluster 6**, the decomposition is also

not considered correct only due to the class Rule. All the remaining classes are part of the Models cluster and contribute to the definition of conditions to certain elements of application. Why is our tool including Rule in this cluster? Assessing not available dynamic information of the system through the expert (whether the access is a read or a write), the rules are created once (write) and then, every time a condition is enforced over a certain data entity of the application, those rules need to be read (through the method applyRules()). This way, Rule should be positioned inside the Models cluster. Note also that the cluster that had mixed functionalities for Blended Workflow in its 2.0 cut is now split into 5 clusters (**Cluster 0, 3, 5, 6, and 9**), showing that our tool is able to determine and examine sub-clusters of a previously big cluster that was hard to reason. Assessing now the true positives of the cluster decomposition (excluding singletons):

Cluster 0: Sub-cluster of the Models cluster, supporting every action related to activities, as it is accessed by every ActivityModelController and ActivityWorkItemController.

Cluster 1: This cluster corresponds to the runtime functionality of the application. Two classes that are classified as singletons should be joined here, from **Clusters 10 and 4**. With a higher cut on the dendrogram, these classes would form the Execution cluster of the expert, but as we opted for a finer granularity, some of those were classified as singletons.

Cluster 2: Sub-cluster of the Design cluster, representing the group of specific conditions to apply to the data structures of the application

Cluster 8: Sub-cluster of the Design cluster, representing another group of conditions, but that are used in a different scope of the ones in **Cluster 2**

Cluster 9: Sub-cluster of the Models cluster, supporting functionality related to goals, as it is accessed by every GoalModelController and GoalWorkItemController.

Finally, following our GUI workflow, the maximum similarity cut on the system also led to the discovery of a candidate for a critical microservice. From the retrieved clusters of that particular workflow, the classes BlendedWorkflow and Specification are part of a separate cluster and seen as being accessed by 90/93 (97%) of the application's controllers. This is confirmed by analyzing the related code, as to execute any functionality in this application, the specification that we are working on must be retrieved. And to retrieve such specification, the specification identifier is fetched from the corresponding blended workflow instance. Therefore, these two classes can also be part of an entry-point service, as we know that most of the accesses to the system will have to go through these classes. From a software architecture point of view, there is no unique design, but it is relevant to analyze the differences and reason about their impact on the systemic properties of the system, as in our case, the relaxing of transactional behavior by comparing unique cluster decompositions originated from different dendrogram cuts. Therefore, our

intention with this metric is to verify that our tool (or Structure101) can identify clusters, big or small in size depending on the analysis, and translate this identification into groups of functionality that can be classified as services. With the use of Structure101 there is no ability to control the number of clusters given as output, but with our approach we can control such divisions through the cut-off height of the dendrogram given. Our approach shows higher precision, recall and f-score values in comparison with Structure101 on its assessment to the expert decomposition. Even though we do not have a 100% precision on the clusters found, we conclude that an hierarchical clustering of the transactional context brings benefits to the developer in terms of functionality identification and their dependencies, which are characteristics that are useful in the migration from a monolithic to a microservices architecture.

4.3.3 Solution limitations

With our presented approach we pretend to tackle the problem of microservice migration in terms of context separation, however, some limitations are present:

1. The manual inspection of the static analysis approach creates a system which is hardly replicated, due to the difficulties associated with manual code analysis. This issue is discussed later in the Future Work section.

2. The applications we are examining are part of a codebase that has not yet been migrated to microservices, being that the expert's decomposition does not bring a total ground-truth.

3. We are only dealing with the problem of context separation, this is, a subset of a migration to a microservices architecture. However, as this work is planned to be continued, the objectives of the first step of the migration and this thesis, were met.

5

Conclusion

Contents

In summary, this document introduces an approach that uses static analysis of the source code of a monolithic business system to discover significant information that is crucial in the division of the monolithic code into contexts. The main objective of this approach is constructing an **informed decision-making** process for the migration from a monolithic architecture to a microservices architecture by presenting the cluster formations that mimic service behaviors and enabling the user to customize them.

Microservices adoption as a core software architecture is starting to become more common due to the advantages it has, not only related to its capability of generating cross-functional teams that work on particular business capabilities, but also for its evolutionary design in terms of scalability. A considerable number of authors have wrote about this particular design and the knowledge necessary to make the transformation between architectures, but no support tools exist that help the developers in such transformation. This thesis addresses this problem by presenting a tool that, with the use of Java-Callgraph in source code analysis on Java-based applications that implement the Fénix Framework, generates call graphs that are transformed into controller-class relationships, manipulating and pruning its information to the contexts that are useful on an architecture migration, modularity and transactionality. Lastly, the modular context is used to generate dendrograms through the use of hierarchical clustering and a GUI and web-browser visualization is built on top of the communities detected by the cut on the previous dendrograms, allowing the user to further manipulate the results into its ideal form.

The several comparisons made between an approach of hierarchical clustering of transactional contexts with the formal tool Structure101 and the expert decomposition culminated in high precision values, reflecting that most of the clusters returned are plausible service candidates. Furthermore, when they do not return the ideal cluster, the results are still useful for the understanding of the functionality and information flow of the system as a whole.

Due to the values presented in our evaluation being higher than an already existing formal tool, we consider that our hypothesis is validated and that there is the capability of retrieving modularity from the transactional contexts of an application, as the relationships between the controllers and domain classes can be used to group domain classes together in contexts.

The outcomes presented will also be useful as a benchmark to be used in further examination of these systems or other iterations of our approach.

5.1 Future Work

- Separating the information retrieved from the controller into reads and writes, enabling specific system information regarding the type of interaction, direction and frequency. This addition can create the opportunity for the increased discovery of architectural software patterns and the assessment of cyclic dependencies. The direction of these interactions will also allow us to see the

entry and exit points of each functionality, providing a clear view of the usage of the domain classes in certain workflow. On the tool Structure101, cyclic dependencies constitute an important feature on the creation of its clusters, this is the reason that this kind of dependency relationship between code elements is proposed to be implemented in the future work of our approach when we are able to capture the direction of such relationships, as in its current state, we capture the accesses but not its direction.

- Using, or implementing, another tool to capture the static/dynamic information of the system, such that a subsequent manual inspection is not needed.
- Test the existing approach for applications that have an existing code divided into two architectural versions: Monolith and Microservices. This way the clustering results can be directly assessed in comparison with a ground-truth.
- Integrate the functionalities with an IDE (as Eclipse), so it provides division capabilities into workable projects.

Bibliography

[1] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media, Inc., 2015.

[2] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.112

[3] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *11th IEEE International Workshop on Program Comprehension, 2003.*, May 2003, pp. 94–103.

[4] J. a. Cachopo and A. Rito-Silva, "Combining software transactional memory with a domain modeling language to simplify web application development," in *Proceedings of the 6th International Conference on Web Engineering*, ser. ICWE '06. New York, NY, USA: ACM, 2006, pp. 297–304. [Online]. Available: http://doi.acm.org/10.1145/1145581.1145640

[5] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] T. Betts, C. Richardson, D. Haywood, M. Gehard, and A. Cockcroft, "Microservices vs. monoliths, the reality beyond the hype," *InfoQ*, vol. eMag Issue 51, 2017. [Online]. Available: https://www.infoq.com/minibooks/emag-microservices-monoliths

[7] Evans, *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[8] V. Vernon, *Implementing Domain-Driven Design*, 1st ed. Addison-Wesley Professional, 2013.

[9] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, pp. 181–196, 2001. [Online]. Available: http://www.rcost.unisannio.it/mdipenta/papers/jss01.pdf

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced*

Studies on Collaborative Research, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[11] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[12] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 362–369.

[13] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb 2013.

[14] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *[1991 Proceedings] 13th International Conference on Software Engineering*, May 1991, pp. 83–92.

[15] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75 – 174, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0370157309002841

[16] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651 – 666, 2010, award winning papers from the 19th International Conference on Pattern Recognition (ICPR). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865509002323

[17] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979. [Online]. Available: http://www.jstor.org/stable/2346830

[18] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov 2007.

[19] Z. Zhang and H. Yang, "Incubating services in legacy systems for architectural migration," in *11th Asia-Pacific Software Engineering Conference*, Nov 2004, pp. 196–203.

[20] F. B. e Abreu, G. Pereira, and P. Sousa, "A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems," in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, Feb 2000, pp. 13–22.

[21] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[22] C. Pahl and P. Jamshidi, “Microservices: A systematic mapping study.” in *CLOSER (1)*, 2016, pp. 137–146.

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

[24] R. C. Martin, “The single responsibility principle,” *The principles, patterns, and practices of Agile Software Development*, vol. 149, p. 154, 2002.

[25] J. Ramos *et al.*, “Using tf-idf to determine word relevance in document queries,” in *Proceedings of the first instructional conference on machine learning*, vol. 242, 2003, pp. 133–142.

[26] J. C. Gower and G. J. Ross, “Minimum spanning trees and single linkage cluster analysis,” *Applied statistics*, pp. 54–64, 1969.

[27] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[28] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges when moving from monolith to microservice architecture,” in *Current Trends in Web Engineering*, I. Garrigós and M. Wimmer, Eds. Cham: Springer International Publishing, 2018, pp. 32–47.

[29] O. Maqbool and H. Babri, “Hierarchical clustering for software architecture recovery,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov 2007.

[30] N. Anquetil and T. Lethbridge, “Experiments with clustering as a software remodularization method,” pp. 235–255, 11 1999.

[31] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[32] J. Garcia, I. Ivkovic, and N. Medvidovic, “A comparative analysis of software architecture recovery techniques,” in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 486–496. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693106

[33] S. Schulte im Walde, “Experiments on the Automatic Induction of German Semantic Verb Classes,” Ph.D. dissertation, Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 2003, published as AIMS Report 9(2).

A

Code of Project

This appendix provides relevant code excerpts of the method implementation.

Listing A.1: Regular expression parsing of the text callgraph file

```
def retrieve_domain_classes(controller, file):
    '''Receives the path to the .txt that has the callgraph representation
        and parses for the domain classes of the controller'''
    with open(file, 'r+') as file_content:
        domain_classes = []
        file_string = file_content.read();
        controller = controller.split('/')[-1][0:-5]

        #Find all the methods called first by the controller
        first_order_method_calls = re.findall(controller + ':' + '(.*)' + '()
            ', file_string)
        method_calls = []
```

```
11
12          for method in first_order_method_calls:
13              method_calls.append(method[0])
14
15          for method in method_calls:
16              function_arguments = method.split('(')[-1][0:-1]
17
18              # Retrieve domain from arguments passed on function
19              if 'domain' in function_arguments:
20                  if 'Base' in method:
21                      pass
22                  elif '$' in method:
23                      pass
24                  elif 'java' in method:
25                      pass
26                  else:
27                      domain_class = function_arguments.split('.')[-1]
28                      domain_classes.append(domain_class)
29                      if controller not in dict_controller_classes.keys():
30                          dict_controller_classes.setdefault(controller, []).
                                append(domain_class)
31                      elif domain_class not in dict_controller_classes[
                            controller]:
32                          dict_controller_classes.setdefault(controller, []).
                                append(domain_class)
33              notsplitmethod = method
34              method = method.split(' ')[-1][3:].split('(')[0]
35              # Also search in its abstract classes
36              classname = method.split('.')[-1].split(':')[0]
37              if classname in abstract_classes.keys():
38                  absclass = abstract_classes[classname]
39                  absmethodcall = notsplitmethod.replace(classname, absclass[0
                        ], 1)
40                  if absmethodcall in method_calls:
41                      pass
42                  else:
43                      method_calls.append(absmethodcall)
44
```

```
45            # If has get on method we need to search the _base class
46            if 'get' in method:
47                if 'Base' in method:
48                    # replace $ with \$ so it can be caught in the regex
49                    search_calls = re.findall("M:" + method.replace('$', '\$'
                          ) + "\((.*)" + "()", file_string)
50
51                    for call in search_calls:
52                        # Remove cyclic dependencies
53                        if call[0] in method_calls:
54                            pass
55                        else:
56                            method_calls.append(call[0])
57                else:
58                    split_method = method.split(':')
59                    method_with_base = split_method[0] + '_Base:' +
                          split_method[1]
60                    search_calls = re.findall('M:' + method_with_base + '
                          \((.*)' + '()', file_string)
61                    for call in search_calls:
62                        # Remove cyclic dependencies
63                        if call[0] in method_calls:
64                            pass
65                        else:
66                            method_calls.append(call[0])
67            # See if the name of the package contains domain
68            if 'domain' in method:
69                if 'Base' in method:
70                    pass
71                elif '$' in method:
72                    pass
73                else:
74                    clazz = method.split(':')[0].split('.')[-1]
75                    domain_classes.append(clazz)
76                    if controller not in dict_controller_classes.keys():
77                        dict_controller_classes.setdefault(controller, []).
                              append(clazz)
78                    elif clazz not in dict_controller_classes[controller]:
```

```
                        dict_controller_classes.setdefault(controller, []).
                            append(clazz)

            search_calls = re.findall('M:' + method + '\((.*)' + '()',
                file_string)
            for call in search_calls:
                # Remove cyclic dependencies
                if call[0] in method_calls:
                    pass
                else:
                    method_calls.append(call[0])

        return remove_duplicates_list(domain_classes)
```

Listing A.2: Construction of the NetworkX Graph

```
def graph_construction(controller, classes):
    # Adds controller node and its edges to nodes of corresponding classes
    G = nx.Graph()
    G.add_node(controller, is_ctrl='yes')
    for domain_class in classes:
        G.add_node(domain_class, is_ctrl='no')
        G.add_edge(controller, domain_class)
    list_of_graphs.append(G)
```

Listing A.3: Joining of the single controller graphs into a general one

```
def join_graphs():
    G = nx.compose_all(list_of_graphs)
    final_graph.append(G)
```

Listing A.4: Creation of the similarity matrix used in clustering

```
def create_similarity_matrix():
    list_unique_classes = sorted(list(get_all_controller_classes()))
    similarity_matrix = np.zeros((len(list_unique_classes), len(
```

```
        list_unique_classes)))
    for index, value in np.ndenumerate(similarity_matrix):
        if len(set(index)) == 1:
            similarity_matrix[index] = 1
        else:
            number_controllers_c1c2 = get_number_controllers_two_classes(
                list_unique_classes[index[0]], list_unique_classes[index[1]])
            number_controllers_c1 = get_number_controllers_one_class(
                list_unique_classes[index[0]])

            similarity_measure = number_controllers_c1c2/number_controllers_c
                1
            similarity_matrix[index] = similarity_measure
    return similarity_matrix
```

Listing A.5: Hierarchical clustering

```
def hierarchical_clustering(similarity_matrix, linkage_type):
    if linkage_type == 'average':
        hierarc = hierarchy.average(similarity_matrix)
    elif linkage_type == 'single':
        hierarc = hierarchy.single(similarity_matrix)
    elif linkage_type == 'complete':
        hierarc = hierarchy.complete(similarity_matrix)

    hierarchy.dendrogram(hierarc, labels=sorted(list(
        get_all_controller_classes())))

    plab.savefig("dendrogram_" + linkage_type + "_2.png", format="png",
        bbox_inches='tight')
    plt.close('all')
    return hierarc, linkage_type
```

Listing A.6: Dendrogram cutting

```
def cut_dendrogram(hierarc, linkage_type, height):
    # Empty dictionaries for repeated cuts
```

```
    dict_class_cluster.clear()
    dict_cluster_controller_access.clear()
    dict_ctrl_class_percentage.clear()

    cut = hierarchy.cut_tree(hierarc, height=height)

    list_unique_classes = sorted(list(get_all_controller_classes()))

    for i in range(0, len(list_unique_classes)):
        dict_class_cluster.setdefault(cut[i][0], []).append(
            list_unique_classes[i])

    return dict_cluster_controller_access
```

Listing A.7: Statistic creation

```
def calculate_cluster_controller_access(linkage_type):
    f = open('statistics_2_' + linkage_type + '.txt', 'a')
    print("---Cluster Percentage-------------", file=f)
    dict_ctrl_class = get_dict_controller_classes()

    for ctrl, _classes in dict_ctrl_class.items():
        set_classes = list(set(_classes))
        for cluster, _classes2 in dict_class_cluster.items():
            for _class in set_classes:
                if _class in _classes2:
                    dict_cluster_controller_access.setdefault(cluster, []).
                        append(ctrl)
                    break

    ordered_dict_cluster_controller_access = collections.OrderedDict(sorted(
        dict_cluster_controller_access.items()))

    for cluster, controllers in ordered_dict_cluster_controller_access.items
        ():
        print("Cluster ", end=" ", file=f)
        print(cluster, end=" ", file=f)
        print("-- Controllers accessed: " + sorted(controllers).__str__(),
```

```
            end=" ", file=f)
        print(file=f)
        print("--------------------------------", file=f)
    f.close()

def calculate_controller_percentage_classes(linkage_type):
    f = open('statistics_2_' + linkage_type + '.txt', 'a')
    print("-----Classes Percentage-----", file=f)
    dict_ctrl_class = get_dict_controller_classes()
    ordered_dict_ctrl_class = collections.OrderedDict(sorted(dict_ctrl_class.
        items()))
    class_counter = 0

    for ctrl, _classes in ordered_dict_ctrl_class.items():
        set_classes = list(set(_classes))
        print(ctrl + ":", file=f)
        for cluster, _classes2 in dict_class_cluster.items():
            for _class in set_classes:
                if _class in _classes2:
                    class_counter += 1
            print("Cluster: " + cluster.item(0).__str__() + " --", end=" ",
                file=f)
            print(float(class_counter).__str__() + " of " + float(len(
                _classes2)).__str__() + " -- " +
                  ((float(class_counter)/float(len(_classes2)))*100).__str__
                      () + "%", file=f)
            class_counter = 0
    f.close()
```

B

Expert and Structure101 application decompositions

Two types of data are presented in this appendix:

1. The expert decomposition of both web applications.
2. The Structure101 clustering of the classes of the web applications.

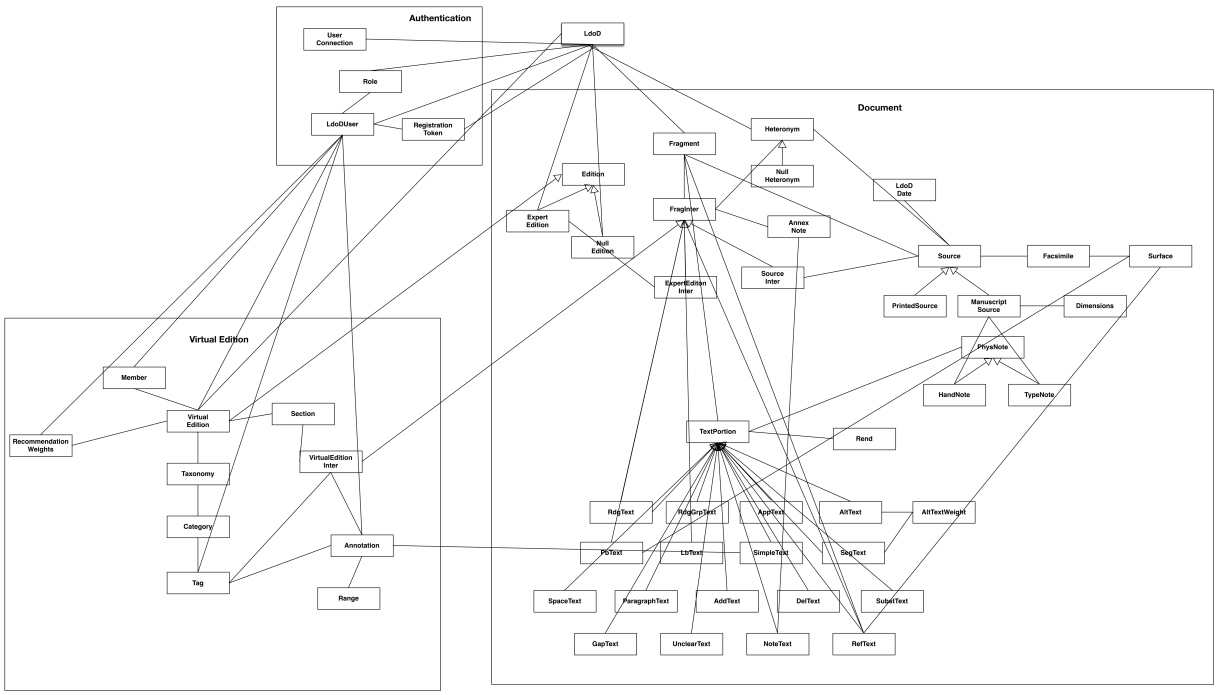

Figure B.1: Expert decomposition of the LdoD application

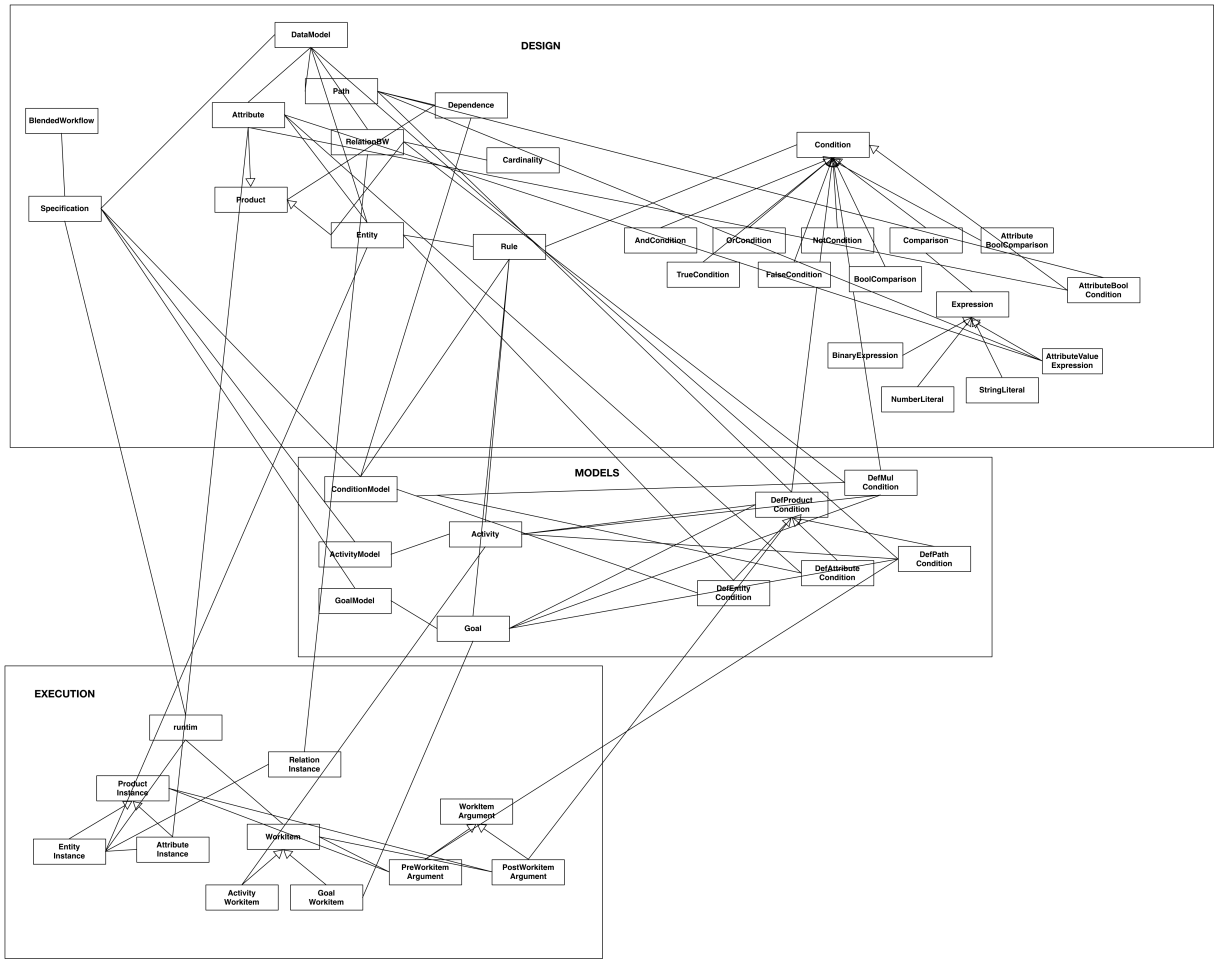

Figure B.2: Expert decomposition of the Blended Workflow application

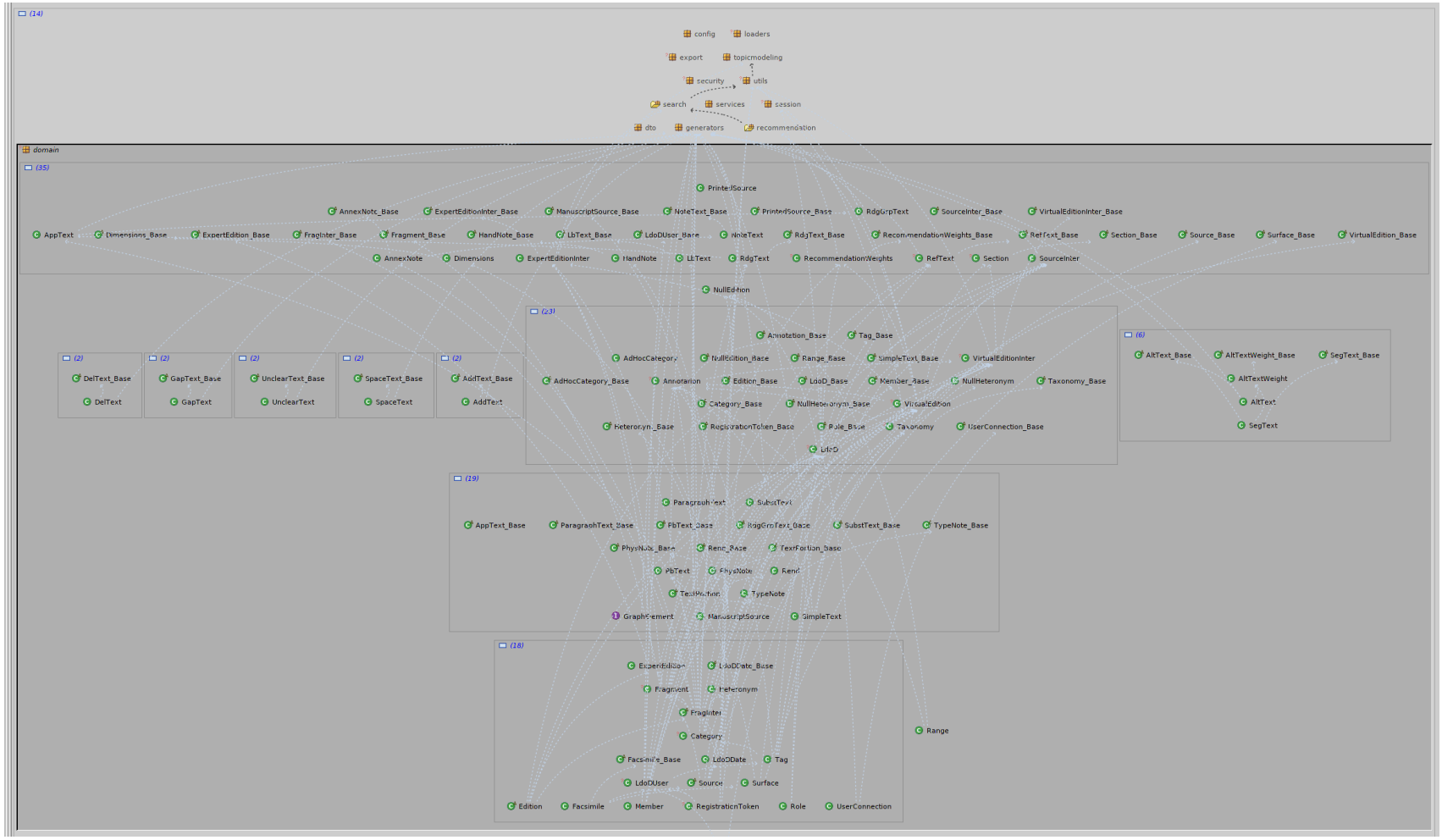

Figure B.3: Structure101 decomposition of the LdoD application

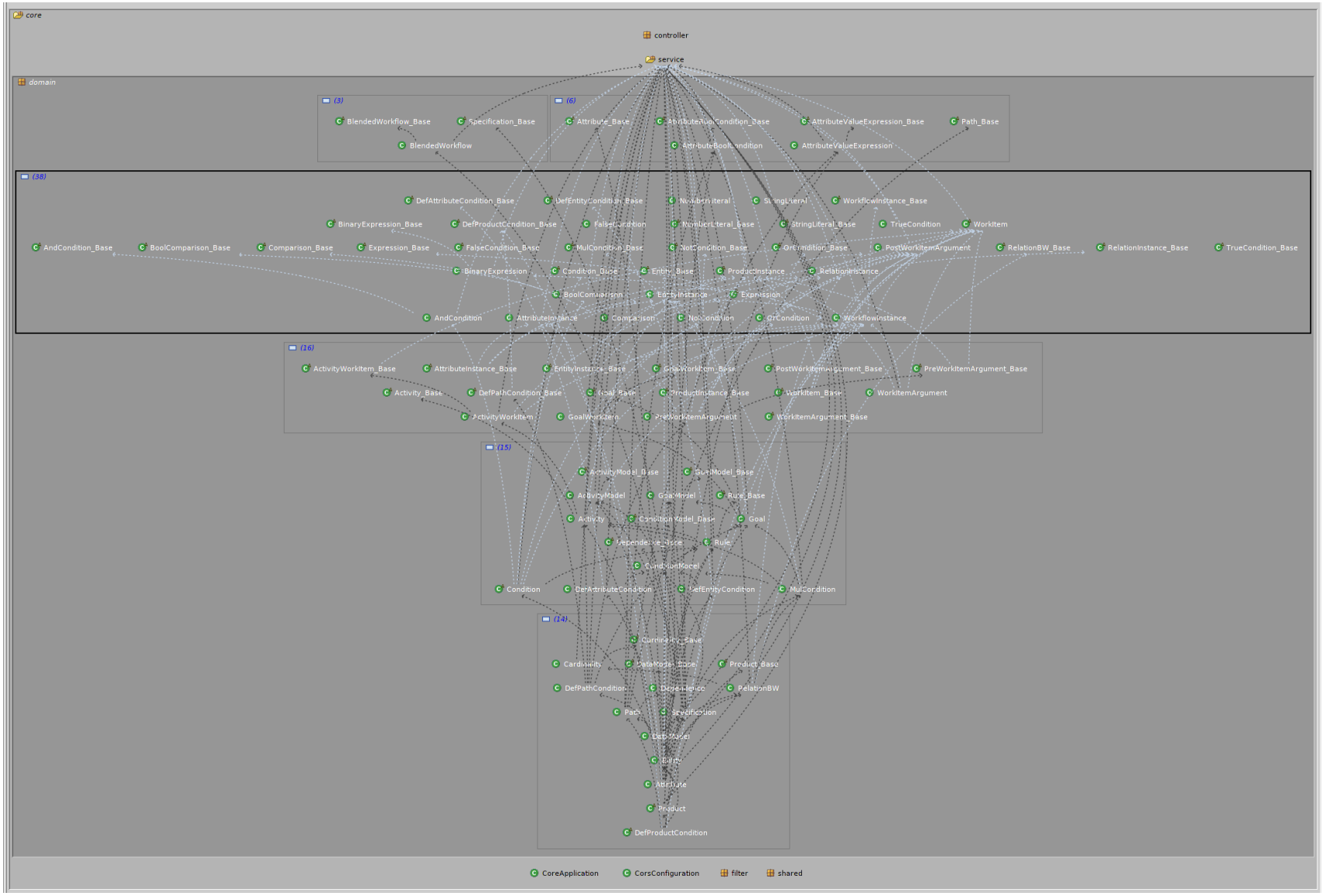

Figure B.4: Structure101 decomposition of the Blended Workflow application