Monolith Modularization Towards Microservices: Refactoring and Performance Trade-offs

Nuno Gonçalves*, Diogo Faustino*, António Rito Silva*, Manuel Portela†
*DPSS - INESC-ID, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
{nuno.m.bagulho.goncalves, diogofaustino, rito.silva}@tecnico.ulisboa.pt
†CLP - Centre for Portuguese Literature, University of Coimbra, Coimbra, Portugal
mportela@fl.uc.pt

***Abstract*—The agility inherent to today's business promotes the definition of software architectures where the business entities are decoupled into modules and/or services. However, there are advantages in having a rich domain model, where domain entities are tightly connected, because it fosters reuse. On the other hand, the split of the business logic into modules and its encapsulation through well-defined interfaces introduces a cost in terms of performance. In this paper we analyze the impact of migrating a rich domain object into a modular architecture, both in terms of the development cost associated with the refactoring, and the performance cost associated with the execution. Current state of the art analyses the migration of monolith systems to a microservices architecture, but we observed that migration effort and performance issues are already relevant in the migration to a modular monolith.**

***Index Terms*—Domain-driven design, Modular architecture, Refactoring effort, Performance evaluation, Microservices.**

I. Introduction

Domain-driven design [1] advocates the division of a large domain model into several independent bounded contexts to split a large development team into several small, and business focused, teams to foster an agile software development. Additionally, these bounded contexts can be implemented as modules, each one of them with a well-defined interface, to further decouple the teams by reducing the number of required interactions between them. This modularization is also suggested as an intermediate step of the migration of a monolith to a microservices architecture [2].

The correct identification of what should be the responsibilities associated with each module is not trivial, and has to be done through several refactoring steps [1], [3], [4]. These refactoring operations can be more easily performed in a strongly connected domain model where the business logic is scattered among the domain entities, a rich domain model, and in the absence of interfaces between the domain entities [5]. So, it is common that projects start with a single domain model, because in the first development phases the domain model is not completely understood. Premature modularization adds a cost to development, because of the need to refactor modules' interfaces, since it initial design does not capture the correct abstractions.

The use of interfaces between modules requires the transformation of data between them to encapsulate their domain models, anticorruption layers [1], which implies significant changes do the domain model, and have impact on the performance of the system. Therefore, the process of modularizing a monolith system, or further migrating it to a microservices architecture [6], has to address these problems.

Research has been done on the comparison of the performance quality between a monolith system and its correspondent implementation using a microservices architecture, but these results are sometimes contradictory, e.g. [7], [8], addresses different characteristics of a microservices system, e.g. [9], [10], or are evaluated using simple systems, e.g. [11], where the monolith functionalities do not need to be redesigned in order to be migrated to the microservices architecture.

In this paper, we describe the refactoring process of a large monolith system into a modular architecture, removing circular dependencies between modules, focusing on the refactoring effort and the performance overheads. Concerning the refactorings, we describe the types of refactoring that were applied and measure their impact on the migration effort. Regarding performance, we describe the architectural tactics that were applied to improve the performance of the modular monolith, and compare it with the monolith performance.

Therefore, besides the identification of refactorings to modularize a monolith, we get insights on what is the impact on performance. The results of this paper also provide a contribution to the methods and techniques for migrating a monolith to a microservices architecture, since the modular monolith can be used has an intermediate step in the process of migration to a microservices architecture. For instance, our results shows a significant impact on latency, even in the absence of the remote invocations between modules. Additionally, it has the advantage that, in our analysis, we separate the aspects of a monolith architectural redesign to a microservices architecture from the technological aspects associated with the implementation of a microservices architecture.

In the next section we describe the architectural elements of a modular architecture for a monolith application. Then, in section III, we present the refactoring activities that split a monolith in a set of modules, and discuss some performance tactics that should be considered during the process. A large monolith system is introduced in section IV, in which we describe how it was modularized by applying the migration refactorings. In section V the migration process is evaluated in terms of refactoring cost and performance, and the results are discussed in section VI, in which they are generalized to applications that have a similar architecture and how these results can apply to the migration to a microservices architecture. Section VII presents related work and section VIII the conclusions.

II. Modular Monolith

To decompose the monolith into a set of modules that do not have circular uses dependencies we followed [5], where the domain model is split into several non-interrelated domain models, and the interactions between these smaller domain models is done through the module interfaces that contain them. Therefore, the modular monolith defines two types of interactions between modules, a uses interaction, where a module uses another by invoking its interface, and a notification interaction, where a module notifies of its changes the modules that subscribe to the notifications. The uses interaction defines a dependence in which the module that uses requires the used module to be present. In this type of interaction, the module that is used has a *provides interface* that match a *requires interface* defined in the client module. The notification interaction allows a module to inform other modules about the occurrence of a change without creating a dependence between them. This is achieved by sending events that contain the information. The module that notifies has a *publish interface* while the module that is notified has a *subscribe interface*, and they only need to agree on the event structure. In the uses interaction, between a requires and a provides interface, data is transferred in the form of Data Transfer Objects (*dto*) [12], which contain information on the module domain model in an inter-module shared format that preserves the encapsulation of the used module domain entities, which contributes to modules' loose-coupling. The notification interactions defines the events a module publishes and that are subscribed by other modules. These events contain information about what occurred in the module and which may be relevant to subscribing modules. Usually, the events contain minimal information and, if necessary, the subscribing modules may afterwards start a uses interaction to obtain more information about the state of module that sent the notification. Note that the uses interaction establishes a flow of control from the use modules to the used modules, and the notification interface defines the inverse flow of control. However, the dependencies only occur in the uses flow of control direction, because the module that notifies does not require the correct implementation of the module that is notified.

By applying the proposed decomposition strategy we obtain two types of modules: front-end and back-end modules. Front-end modules contain the end user interface and only have *requires* and *subscribe* interfaces to interact with the back-end modules. Each back-end module contains part of the domain model, which is persistently store in the database, and have *provides* and *subscribe* interfaces to be used by both, front-end and back-end, modules.

An advantage of the modular monolith is that it conciliates small teams, one for each part of the domain model, with transactional execution of the monolith functionalities, because all parts of the domain model are stored in the same database [5].

III. Refactoring

Most of the effort in the modularization of a monolith occurs on the decompositon of its domain models into the different modules, such that each one of the modules does not interdepend on a shared data repository [5].

To partition the monolith domain model, it is necessary to consider the relationships between the domain entities that will belong to different modules. These relationships need to be replaced by invocations to the modules interface. Since circular dependencies are not allowed in a modular monolith, it is necessary to decide what will be the dependency between the modules, and implement their interactions using uses and notification interfaces. The implementation of these interactions requires the definition of *dtos* and events. Therefore, the cost associated with the migration effort depends on the number of relationships between the new modules' domain entities. Note that, although a domain entity may not have a direct relationship with a domain entity of another module, it may receive it as a parameter of an invocation. This is the dependence problem addressed by the Demeter Law [13]. So, this indirect relationships also need to be dealt with during the refactoring process.

Another aspect that needs to be addressed, when modularizing a monolith, is the presence of god classes [14], classes that tend to centralize the intelligence of the system, in the monolith. Typically, god classes are singletons that are the entry point for all the domain entities. Therefore, it is necessary to split these classes into several classes, one for each partition of the domain model. This refactoring requires the identification of which methods of the god class access which part of the domain model, and move them to the corresponding new singleton. This task is not complex, except when the methods access more than a single module, but this case is like the above problem of refactoring inter-module relationships. Therefore, the complexity of the god classes refactoring reduces to the complexity of removing inter-module relationships.

In our domain model two types of relations between entities are possible: associations, which allow for the specification of various types of multiplicities, such as one-to-one and one-to-many, and inheritance. When a superclass has subclasses that will belong to different modules, it is necessary to add all methods that are implemented in the parent class to the child classes. Therefore, similarly to the refactoring of god classes, the complexity of refactoring these methods reduces to the refactoring of the inter-modules associations they use.

We conclude all the identified types of refactoring reduce to the case of removing an association between modules. To do this refactoring, it is necessary to keep a unique identifier of the domain entity of the used module in the module that use it. This unique identifier is used by the use module to obtain a *dto* with information of the entity, and for the used module to notify a change of its state by publishing an event. Note that, the returned *dto* may contain unique identifiers of other domain entities, due to the indirect relationships between domain entities.

Figure 1 describes the uses interaction where two domain entities, `E1` and `E2`, that belong to two different modules, respectively, `A` and `B`. On the left side is represented an association between these domain entities that will be removed in the refactoring. The arrow denotes that, as result of the refactoring, module `A` will depend on module `B`, the former is aware of entities of the latter, but not the opposite. So, the previously bidirectional interaction is transformed into a unidirectional one, where module `A` can use module `B`, and module `B` can notify module `A`. To support these interactions, a `E2Dto` data transfer object needs to be defined.

On Figure 1 right-hand side is a used interaction between the modules, where module *A* obtains an instance of `E2Dto`. Therefore, `e1:E1` has to hold a unique identifier of entity `e2:E2` and when executing method `m` it interacts with module `A` requires interface (`MARI`) to obtain `e2Dto:E2Dto`. In this interaction, module `B` provides interface (`MBPI`) queries module `B` singleton god class (`MBGC`), to obtain the `E2` instance, given the unique identification sent by module `A`, and creates a `e2Dto:E2Dto` that is returned to `MARI`. The returned data transfer object, `e2Dto:E2Dto`, may contain unique identifiers to other domain entities of the used module (the indirect inter-module relationships).

The definition of several modules in the modular monolith raises performance problems associated with the inter-module communication and the need of data transformations to isolate the domain model of each module. When, during refactorization, performance evaluation of the migrated functionalities show poor performance, the number of invocations between modules had to the reduced. This is achieved by redesigning the inter-module interactions into coarse-grained invocations. Monoliths are often implemented using fine-grained object-oriented interactions.

The following optimizations were systematically applied:

- Associate a database access index to the unique identifiers of domain entities exported to other modules, due to the frequent invocations to get a *dto* object of a domain entity, given its unique identifier.
- Whenever a *dto* is generated, by default, it is loaded with the values for all its attributes, to reduce further inter-module invocations to obtain each one of its values. This introduces a memory overhead, because some of the attribute values may not be necessary in the context of that particular interaction. However, overall, after systematically applying this tactic, the performance of the system improved.
- In a few cases, after performance evaluation, it was necessary to have larger *dto* objects, which, additionally to the attribute values, also contain the information associated with several interrelated domain entities. For instance, a *dto* of a list of *dto*s may reduce the number of inter-module invocations when a module is interacting with a collection of domain entities in another module.

IV. Case Study

The LdoD monolith[1] is a digital archive for digital humanities that offers features like searching, browsing and viewing the original text fragments, different variations of the text fragments, as well as different editions of a book. It also provides a way for users to create their own (virtual) editions of the book, in which they can add, remove and order the fragments as they wish. Other features include a recommendation feature, that defines a proximity measure between different fragments according to a set of criteria, a game feature, which implements a serious game where users tag text fragments, a reading feature, which suggests reading sequences of the text fragments, and a visual feature that provides graphical visualizations of the relations between text fragments.

The monolithic application is implemented in Java using the Spring-Boot framework[2] and an Object-Relational Mapper (ORM) to manage the domain model's persistence. The domain model has 71 domain entities and 81 bidirectional relations between domain entities, which resulted in a strongly coupled rich domain model. This solution provides high reusability, because the business logic is split among the domain entities and can easily be reused, and also due to the bidirectional relations that facilitate the navigation in the domain model. In total, the monolith has 25.862 lines of Java code and 20.039 lines of JSP code.

The modular monolith implements the same features as the monolith. These features are implemented as a set of back-end modules that apply the uses and notification interactions to preserve the dependencies between features: `Text`, which represents the fragments and their expert editions; `User`, which provides users registration, authentication, and access authorization; `Virtual`, which allows the definitions of virtual editions by end users and requires the `Text` and `User` modules; `Recommendation`, which calculates similarity distances between the fragments, given a set of criteria, like date and tf-idf (Term Frequency Inverse Document Frequency) [15], between text fragments and uses the `Virtual` and `Text` modules; `Reading`, which uses the `Text` and `Recommendation` modules to provide reading recommendations; `Search`, which implements the search feature and uses `Virtual` and `Text` modules; `Visual`, which uses the `Recommendation`, `Virtual` and `Text` modules to offer an graphical interaction with the archive; and `Game`, which implements a tagging serious game to classify the fragments in a virtual edition, and use the `Virtual` module. The modular monolith has a single front-end module that provides the user interface and is implemented using a server-side technology, Java-Server Pages (JSP). Besides, the described uses relations, the modules also interact through notifications to guarantee

[1]https://ldod.uc.pt/
[2]https://spring.io/

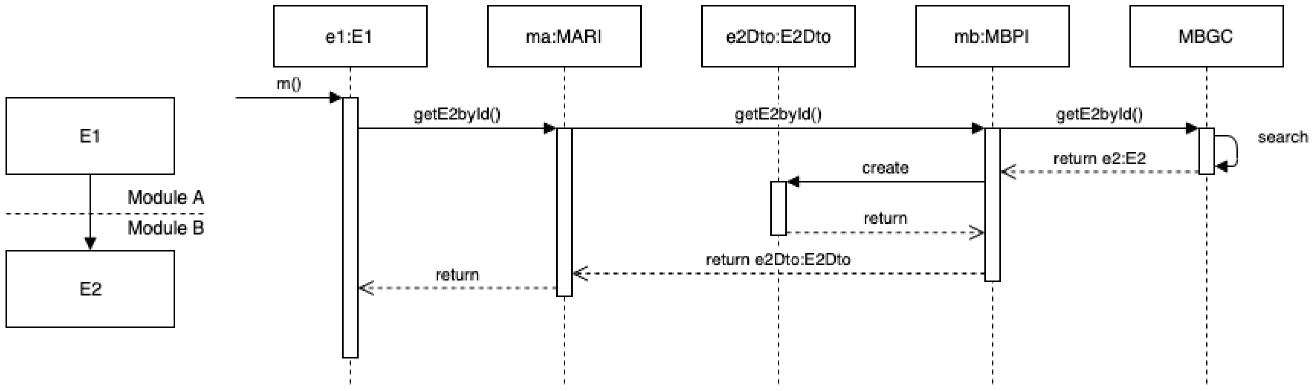

Fig. 1: Example of an entity obtaining a *dto* of a domain entity in another module

that the state is kept in a consistent state. For instance, when a text fragment is deleted, which occurs in the `Text` module, an event is published, which is subscribed by the `Virtual` module, to remove the fragment from all virtual editions that refer to it.

Figure 2 shows the refactoring that was applied to part of the monolith domain model, where abstract superclasses `Edition` and `FragInter` are removed and their subclasses split between `Text` and `Virtual` modules. The association between `FragInter` and `Fragment` is preserved in the `Text` module in entity `ScholarInter`, but in the `Virtual` module the fragment will be represented by its unique identification. When necessary, the `Virtual` module can request the information of a fragment, given its unique identifier, and receive the corresponding `FragmentDto` object. Additionally, when a `Fragment` is deleted, the `Text` module emits an event that is subscribed by the `Virtual` module in order to delete the corresponding `VirtualEditionInter` entities. A similar refactoring is applied to the association between `FragInter` and `VirtualEditionInter` which results in a reflexive association in entity `VirtualEditionInter` together of a unique identifier of a `ScholarInter` entity.

V. Evaluation

The evaluation presents the impact of the migration of the monolith to the modular monolith from the perspectives of refactoring and performance.

A. Refactoring

Table I presents the impact of these factoring in terms of domain entities, and their relationships, of the different modules. It can be observed that there is a big impact on the domain models, where more than 50% of the domain entities had to be changed, and in the case of the `Virtual` module this value reaches 82%. This corresponds to a considerable refactoring effort associated with the migration from the monolith to the modular monolith. A similar situation can be observed in the percentage of *dtos* that had to be created, given the total number of domain entities, which has impact on the changes to the modules that use these *dtos*, and in particular in the domain entities that interacted, before the refactorization, with the original domain entities.

Note that while the number of domain entities modified in the `Text` module might appear higher than it should be, since it does not depend on any other module, the majority of the changes where relatively small and focused mainly on the module's publish interface. It is necessary to remove code that interacts with entities that are located in other modules, and emit events instead. For instance, the `remove` method of the `FragInter` entity, that is moved to the `ScholarInter` subclass. Originally, this entity had a direct relationship to a set of `VirtualEditionInter` instances, allowing it to directly trigger the removal of those instances when it is removed. Since this relation was removed due to connecting two separate modules, this code was replaced with a call to the event interface to notify all modules of the removal of an instance of `ScholarInter` (`FragInter`).

In what concerns the impact on associations, two situations can occur: (1) modified relations, which correspond to associations where one of the involved domain entities changes, but not the overall meaning and purpose of the association; (2) removed associations, which correspond to associations between domain entities of different modules that have become represented by the domain entities unique identifiers. Considering the refactorings in Figure 2, where the association between `FragInter` and `Fragment` is initially refactored into an association between `ScholarInter` and `Fragment` and another between `VirtualEditionInter` and `Fragment`, the former new association is modified, and the latter is removed.

It can be observed in Table I that there is a smaller impact of the refactoring on the associations, because only the associations between domain entities belonging to different modules are impacted, while more domain entities can be impacted due to the indirect relationships. However, the percentage of changed associations is higher if a module has less domain entities, because most of them will be in the interface with the other modules. Therefore, in the `Text` module a high number of relations are fully contained within the module,

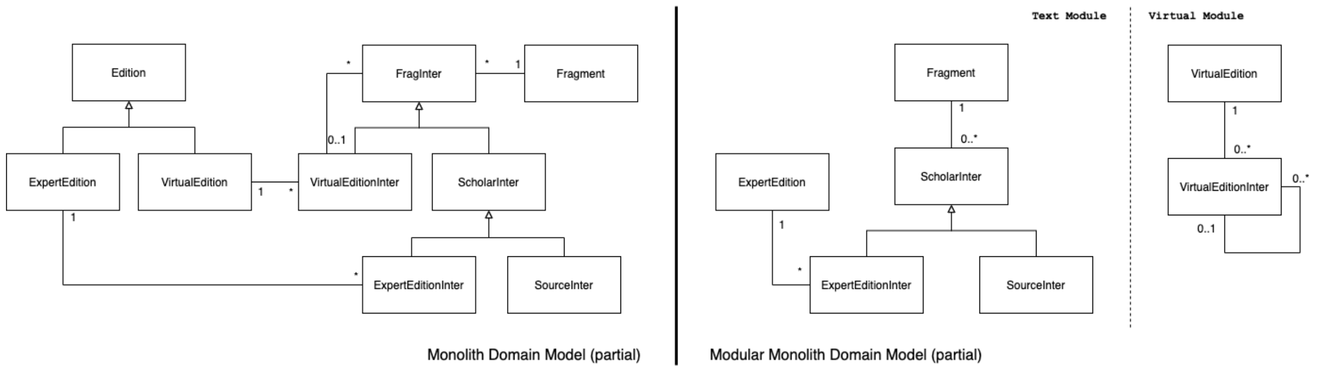

Fig. 2: Domain model refactoring (partial)

	Text	User	Virtual	Recommendation	Reading	Game	Visual
Modified Entities/Total Entities	23/42 (55%)	2/5 (40%)	14/17 (82%)	1/2 (50%)	0	3/5 (60%)	0
Defined Dtos/Total Entities	10/42 (24%)	1/5 (20%)	7/17 (41%)	0/2 (0%)	0	0/5 (0%)	0
Modified Relations/Total Relations	6/38 (15%)	4/7 (57%)	7/26 (27%)	0/2 (0%)	0	0/6 (0%)	0
Removed Relations/Total Relations	2/38 (5%)	0/7 (0%)	3/26 (12%)	2/2 (100%)	0	5/6 (83%)	0

TABLE I: Impact of the refactoring in the domain model

and it leads to a low percentage of modified relations when compared to other modules that required modifications. On the other hand, the `Game` module is the outlier among the modules. The fact that all its relations are removed is the result of having only two domain entities, which had associations with domain entities belonging to other modules. Finally, the `Reading` and `Visual` modules that do not have domain entities, and so no values are presented.

B. Performance

To evaluate the impact that the new architecture has on the performance of the system, we performed performance tests on the previously described features of the application, using the JMeter[3]. The testing was done on a dedicated machine with an Intel I7 6 cores, 16GB of RAM and a 1TB of SSD.

Four functionalities were selected for the analysis. They were chosen due to their impact in terms of: the number of the domain entities they interact with, the number of modules accessed by the functionality, and the amount of processing require by the functionality.

The *Source Listing* functionality presents the listing of the archive sources, where a source corresponds to a physical source document. When using this functionality, the end user obtains all the information about each one of the 754 sources, such as date, dimensions, and type of ink used in the document. The implementation of this functionality is done through an interaction between the front-end and `Text` modules. The modular monolith implementation was optimized to have a single invocation between the modules and a composed *dto* object is returned, which contains the list of all `FragmentDto`s where each `FragmentDto` embeds its list of `SourceDto`s.

[3]https://jmeter.apache.org/

The *Fragment Listing* functionality is implemented in the modular monolith as an interaction between the front-end and `Text` modules to present the list of all text fragments. There are 720 text fragments in the archive. For each fragment it is presented the information of its several interpretations, where each fragment can have 2 to 7 interpretations. The front-end does a total of 5 invocations to the `Text` module. Only one of the invocations required an additional optimization, in which the font-end obtains the list of `FragmentDto`s, and each fragment contains its `ScholarInterDto`s, both `ExpertEditionDto`s and `SourceInterDto`s.

The *Interpretation View* functionality presents an interpretation of a text fragment. Its implementation in the modular monolith requires several interactions between the front-end module and the `Text`, `User` and `Virtual` modules. To do the performance test it was chosen an interpretation of a virtual edition, to have a test that includes more different domain entity types. It was not possible to optimize the implementation of the functionality using a coarse-grained invocation. On the other hand, the amount of information retrieve in each inter-module interaction is small.

The *Assisted Ordering* functionality orders the fragments according to a set criteria, such as date and text similarity (tf-idf). This ordering requires more than 250 000 fragment comparisons (considering for instance a virtual edition with 720 fragments, we get $720 * 719/2 = 258\ 840$ comparisons between fragments), and each comparison requires information about the fragment for up to 4 criteria. For its implementation the front-end accesses 4 back-end modules, `Text`, `User`, `Virtual` and `Recommendation`. Because this functionality repeatedly interacts through the same set of data, the information about the fragments is cached in the monolith implementation, to improve performance. The modular monolith implementation also uses this cache.

	Text	User	Virtual	Recommendation	Reading	Game	Visual
Source Listing	8/42 (19%)	0/5 (0%)	0/17 (0%)	0/2 (0%)	0	0/5 (0%)	0
Fragment Listing	9/42 (21%)	0/5 (0%)	0/17 (0%)	0/2 (0%)	0	0/5 (0%)	0
Interpretation View	22/42 (52%)	1/5 (20%)	6/17 (35%)	0/2 (0%)	0	0/5 (0%)	0
Assisted Ordering	21/42 (50%)	1/5 (20%)	3/17 (18%)	1/2 (50%)	0	0/5 (0%)	0

TABLE II: Coverage of the domain entities by each of the functionalities

Table II presents the number of domain entity types of each one of the modules that a functionality accesses.

For each functionality, a test case was designed to compare the performance in the modular architecture to the performance in the monolith. Each test case run simulates a user that sequentially submits 50 requests, after a first request to warm the caches. The test cases are run for two different loads of the database, for 100 and 720 fragments, respectively.

Table III presents the results. It can be observed that the impact on performance of the migration to the modular monolith increases with the number of fragments. In 3 of the 4 functionalities the difference on the performance, between the monolith and the modular monolith, is much higher for 720 fragments than for 100 fragments. This is due to the number of *dtos* that are generated. Note that for the *Interpretation View* functionality, there is no difference, because it is a functionality that does not depend on the number of fragments in the database. In the *Assisted Ordering* functionality the difference is not so significant as it would be expected, considering the number of operations. This is due to a cache that is being used in both implementations, which reduces the need to transfer *dtos* between modules, although this is the most computationally demanding functionality.

A little bit surprising, for 100 fragments some functionalities have slightly better performance in the modular monolith. This is due to the fact that one of the modular monolith optimizations is the association of an index to the unique identifiers that are sent to the other modules. The monolith is implemented following an object-oriented approach, where objects are retrieved by searching in an object graph. The optimization in the modular monolith allows a faster retrieve of the domain entities, given their unique identifier, because there are less round trips to the database. This was verified through some test cases in which this optimization is removed, and the results shown a huge decrease of the performance of the modular monolith. Note that, since the *Interpretation View* functionality does not depend on the number of fragments, it performed better than the monolith even for the tests with 720 fragments.

VI. DISCUSSION

The process of modularizing a monolith requires an extensive refactoring and has a significant impact on the performance.

In what regards the refactoring effort, the changes to the monolith code have to address the domain entities encapsulation in modules, as well as changes on the granularity of the interactions between domain entities, due to the impact on performance. Although refactorings start by addressing the relationships between domain entities that belong to different modules, the changes propagate to domain entities that do not have a direct relationship, because references to domain entities are sent through methods invocations. Additionally, the removal of circular dependencies between modules also requires changes on the design of the functionalities.

On the other hand, in what concerns performance, different types of impact occurred. The amount of information sent between modules, *dtos*, has a direct impact on performance, but not necessarily because of the number of invocations. The number of invocations are only relevant due to the amount of information they transfer. For instance, emitting events do not have impact on performance. The functionalities that do not depend on large transfers of information can even get a better performance because the modules optimize the access to domain entities by their unique identifiers.

In the context of a stepwise migration of a monolith to a microservices architecture, the results show the complexity that has to be addressed, even before starting to implement microservices. Note that in the context of the microservices architecture extra complexity is added due to the latency associated with remote invocations, the decision whether to use synchronous or asynchronous communication, and the lack of ACID transactional behavior in the execution of functionalities. Anyway, these results inform about some concerns that have to be addressed when migrating to microservices, and that are often neglected in the literature that focuses more on technical aspects. Besides, the migration to modular monolith already prepares for the final migration because the modules have well-defined interfaces, and the uses and notification interactions can be implemented into, respectively, synchronous and asynchronous remote communication.

The following threats to the validity of this study were identified: (1) it is a single example of a migration; (2) it depends on the technology and programming techniques used in the monolith.

Despite being a single case study, it has some level of complexity and the literature lacks descriptions of the problems and solutions associated with the migration from monolith to microservices architecture. Even though it only describes the migration to a modular monolith, the problems it addresses can provide feedback to the overall process.

The technology and programming techniques used in the implementation of the monolith follow an object-oriented approach, where the behavior is implemented through fine-grained interactions between objects. A more transaction script based architecture [12] may result in different types of prob-

Functionality	**Source Listing**			**Fragment Listing**		
Samples	1x50			1x50		
	Monolith	**Modular**	**Variation**	**Monolith**	**Modular**	**Variation**
Avg Time (ms)	25/151	37/974	48%/545%	133/839	119/1412	-11%/68%
Min Time (ms)	24/149	35/936	48%/528%	131/832	116/1372	-11%/65%
Max Time (ms)	41/163	42/1181	2%/635%	138/880	146/1593	6%/81%
Std. Dev.	2.35/2.45	1.44/58	-	1/11	5/55	-
Throughput (/sec)	39.1/6.6	26.6/1	-32%/-85%	7.5/1.2	8.3/0.70	11%/-42%
Functionality	**Interpretation View**			**Assisted Ordering**		
Samples	1x50			1x50		
	Monolith	**Modular**	**Variation**	**Monolith**	**Modular**	**Variation**
Avg Time (ms)	42/43	31/41	-26%/-5%	180/8957	360/17386	100%/94%
Min Time (ms)	41/42	30/39	-27%/-7%	169/8793	348/17150	106%/95%
Max Time (ms)	45/46	35/56	-22%/22%	213/9491	384/17738	80%/87%
Std. Dev.	0.79/1	0.98/2	-	8/169	9/170	-
Throughput (/sec)	23.5/22.9	32/24.2	36%/6%	2.1/0.101	1.5/0.057	-29%/-44%

TABLE III: Performance results for sequentially executing 50 times each functionality for 100 and 720 fragments in the database. Results are separated by / in each cell, for instance, by sequentially executing 50 times the Source Listing functionality in the monolith we observed an average latency of respectively 25, and 151, milliseconds, where there are respectively 100, and 720, fragments in the database (25/151).

lems. The conclusions of this study apply when the monolith is developed using a rich object-oriented domain model. On the other hand, the monolith is implemented using Spring-Boot technology which follows the standards of web application design.

VII. Related Work

There are several challenges when migrating from monolith systems to a microservices architecture [16], [17], such as the effort to redesign the monolith and the performance impacts, and we can find in the literature the description of the migration of some large monoliths [18]–[20].

There are several reports on the migration of large monoliths to a microservices architecture. The migration described by Gouigoux and Tamzalit [18] discusses the aspects of service migration, deployability and orchestration. It reports an improvement in the performance, but does not describe details of the refactorings and optimizations done. Bucchiarone et al [19] describe the lessons learned from their migration of a monolith in the banking domain, focusing on the benefits and challenges of the new system, but doe not discuss the migration effort and the impact on performance. Barbosa and Maia [20] discuss the migration of a large monolith to a microservices architecture, where the monolith business logic is implemented using stored procedures and they focus on the process of identifying microservices.

Therefore, it is necessary to have more case studies that describe the migration efforts and the architectural trade-offs, besides the advantages, and drawbacks, of the final product. In this paper we contribute by describing the refactoring, and related effort, associated with the migration of a large monolith to a modular monolith, which can be used as an intermediate step of the complete migration.

On the other hand, there is work on impact that the migration of a monolith to a microservices architecture has on performance, although it follows different perspectives.

Ueda et al [8] compare the performance of microservices with monolith architecture to conclude that the performance gap increases with the granularity of the microservices, where the monolith performs better. Villamizar et all [7] show different results, concluding that in some situations the performance is better in the microservices context and that it reduces the infrastructure costs, but request time increases in microservices due to the gateway overhead. Al-Debagy and Martinek [10] conclude that they have similar performance values for average load, and the monoliths performs better for a small load. In a second scenario the monolith has better throughput, but similar latency, when the system was stressed in terms of simultaneous requests. Bjørndal et al [21] benchmark a library system, that has 4 use cases and considers synchronous and asynchronous relations between microservices. They observe that monolith performs better except for scalability. Therefore, they identify the need to carefully design the microservices, in order to reduce the communication between them to a minimum, and conclude that it would be interesting to apply these measures in systems that are closer to the kind of systems used by companies.

Some other perspectives compare the performance of monolith and microservices systems in terms of the distributed architecture of the solution, such as master-slave [22], the characteristics of the running environment, whether it uses containers or virtual machines [9], the particular technology used, such as different service discovery technologies [10], or other microservices deployment aspects [11].

Our approach allows to separate concerns when measuring the performance impact of the migration, and led us to conclude that it is already visible when migrating to a modular monolith, in the absence of distributed communication and microservices implementation technologies. And contrarily to some of the related work, in which there is no redesign of the functionalities of the monolith, we highlight that such redesign is required and has impact on the performance.

VIII. Conclusion

In this work, we describe the migration of a large object-oriented monolith to a modular monolith. The migration effort is analysed as well as the impact on performance. A set of refactorings for the partitioning of a domain model into different modules were described and the impact on the overall performance of the system measured through four distinct functionalities.

The results show that the refactorings have a large impact on the system, implying a noteworthy impact on the migration effort. Additionally, part of the functionalities have to be re-designed in order to have coarse-grained interactions between modules. This is also due to the impact the migration has on performance, given that the data transformations between the domain models of different modules are one of the main contributors for performance deterioration.

The results also inform about the migration process of a monolith to a microservices architecture. The modular monolith can be seen as an intermediate artifact, and it can even facilitate the process by separating concerns. Interestingly, the need to redesign the functionalities is already evident in this first step, even before their ACID transactional behavior is replaced by eventual consistency. Additionally, the two types of interactions, uses and notification, applied in the modular monolith, are an initial fit to the introduction of distribution, and remote invocations between microservices, synchronous and asynchronous. Overall, the aspects addressed in the first step are required in a complete migration process and are consistent with what is necessary to do in the next steps.

Although we describe a single case study of migration, it is of a system that is in production, and implemented using the best practices for the development of web applications.

The code repository is publicly available in a GitHub repository[4], in which the branch *master* contains the monolith and branch *icsa2021* the modular monolith.

Acknowledgement

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.

[4]https://github.com/socialsoftware/edition

References

[1] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] C. Richardson, *Microservices Patterns*. Manning, 2019.

[3] W. F. Opdyke and R. E. Johnson, "Creating abstract superclasses by refactoring," in *Proceedings of the 1993 ACM Conference on Computer Science*, ser. CSC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 66–73. [Online]. Available: https://doi.org/10.1145/170791.170804

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2018.

[5] D. Haywood, "In defense of the monolith," in *Microservices vs. Monoliths - The Reality Beyond the Hype*. InfoQ, 2017, vol. 52, pp. 18–37. [Online]. Available: https://www.infoQ.com/minibooks/emag-microservices-monoliths

[6] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[7] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[8] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[9] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 342–346.

[10] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 149–154.

[11] F. Tapia, M. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied Sciences*, vol. 10, no. 17, 2020.

[12] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[13] K. J. Lienberherr, "Formulations and benefits of the law of demeter," *SIGPLAN Not.*, vol. 24, no. 3, pp. 67–78, Mar. 1989. [Online]. Available: http://doi.acm.org/10.1145/66083.66089

[14] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[15] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing Management*, vol. 24, no. 5, pp. 513 – 523, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0306457388900210

[16] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909.

[17] 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.

[18] J. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 62–65.

[19] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018.

[20] M. H. Gomes Barbosa and P. H. M. Maia, "Towards identifying microservice candidates from business rules implemented in stored procedures," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 41–48.

[21] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien, "Migration from monolith to microservices: Benchmarking a case study," 2020, unpublished. [Online]. Available: http://10.13140/RG.2.2.27715.14883

[22] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.