Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-Oriented Architecture.

Pascal Zaragoza[1,2], Abdelhak-Djamel Seriai[1], Abderrahmane Seriai[2], Hinde-Lilia Bouziane[1], Anas Shatnawi[2], Mustapha Derras[2]

[1]*LIRMM, CNRS and University of Montpellier, Montpellier, France*

[2]*Berger-Levrault, France*

{*zaragoza, seriai, bouziane*}*@lirmm.fr,* {*abderrahmane.seriai, anas.shatnawi*}*@berger-levrault.com*

Keywords: Microservices, Monolith, Modernization, Reverse Engineering, Refactoring, Software Architecture

Abstract: The emergence of the microservice-oriented architecture (MSA) has led to increased maintainability, better readability, and better scalability. All these advantages make migrating a monolithic software towards an MSA an attractive prospect for organizations. The migration process is recognized to be complex and consequently risky and costly. This process is composed of two phases: (1) the microservice-based architecture recovery phase and (2) the transformation (i.e. materialization) phase. In this paper, we propose a systematic approach to transform an object-oriented monolithic application towards an MS-oriented one by applying a set of transformation pattern. To validate our approach we automated it with our tool MonoToMicro, and applied it on a set of monolithic Java applications to be migrated towards microservices-based ones.

1 INTRODUCTION

The microservice-oriented architecture (MSA), with its capabilities for quick deployment, better scalability, and maintainability, is a recent architectural style that has appeared to take advantage of the Cloud (Richardson, 2018). MSA allows the development of applications as a suite of small services, each running in its own process and communicating through lightweight interfaces (Lewis and Fowler, 2014), (Newman, 2019). Individually, each microservice can be technologically independent, functionally autonomous while guaranteeing its autonomy with regard to their manipulated data. As a consequence, this results in a more manageable codebase as each microservice can be managed by a smaller team (Baskarada et al., 2020).

In contrast, the monolithic architecture style involves building the server-side application as a single logical executable unit (i.e. monolith) (Lewis and Fowler, 2014). These monoliths can become quite large and complex and thus harder to maintain. Furthermore, a change to a small part of the application requires rebuilding and redeploying the entire monolith (Soldani et al., 2018). When deployed and running, increasing workloads requires duplicating the entire application. This is a very resource-intensive, as every part of the application must be replicated even though all of the application is not needed (Selmadji et al., 2020).

This paradigm shift from the monolithic style to the MSA-based one resulted in a demand for migrating these monolithic legacy systems towards an MSA. This migration process includes: (1) the recovery of a MSA-based architecture of the existing application, (2) the transformation of the monolithic source code to be conforms to MSA principles.

Many approaches have been proposed to address the first phase of the migration process by partitioning the OO implementation of a given monolithic application into clusters of classes that can be transformed and packaged as microservices in the second phase. Although the resulting clusters help understand the target MSA, the source code must be transformed to conform to the MSA style (service-based, message-oriented communication, etc.).

The goal of the second phase of the migration is to transform the monolithic source code to create runnable microservices that conform to its target MSA, while preserving the business logic of the application. In the case of OO applications, the primary difficulty is to transform the OO dependencies between the clusters of classes into MSA ones. These transformations must adhere to refactoring principles (i.e. preserve the business-logic) without degrading the performance. However, despite the importance

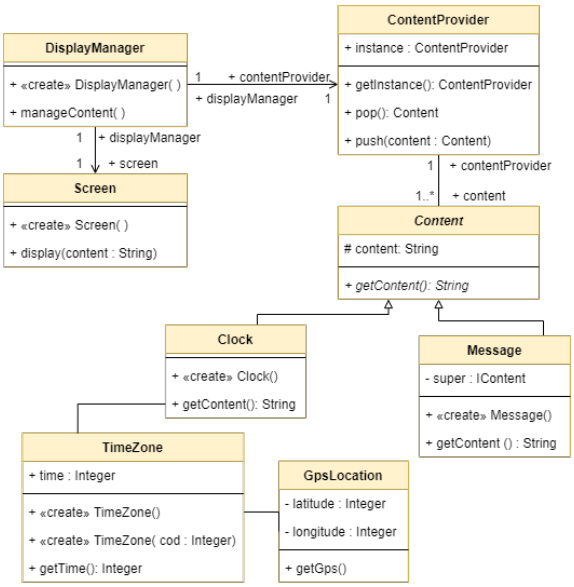

Figure 1: Information Screen class diagram inspired (Alshara et al., 2016)

of the second phase of the migration (i.e. costly and error-prone), and to the best of our knowledge, no approach to automate this phase has been proposed.

In this paper, we propose a systematic approach to transform an OO application from the monolithic style to an MSA one based on a set of transformation patterns. This set of transformation patterns create microservice-based communication mechanisms that preserve the semantic of the monolith while conforming to the principles of the MSA (e.g. message-based and data-oriented). Furthermore, we propose an automated process, and a tool, that applies our systematic transformation approach. Finally, we apply our approach on a set of monolithic applications to determine whether this approach is able to refactor the code while preserving the business logic and not negatively affect the performance of the application.

2 MATERIALIZING MICROSERVICES FROM AN OBJECT-ORIENTED APPLICATION

The overall objective of migrating a monolithic application is to produce structurally, behaviorally, and operationally valid microservices. Particularly, we define a microservice as one that follows the commonly accepted definitions which include structural and behavioral characteristics such as "*small and focused on one functionality*", "*structural & behavioral autonomy*", and "*data autonomy*" (Lewis and Fowler, 2014). Additionally, operational characteristics include "*running on its own process*", "*communicate with lightweight mechanisms*", and "*automatically deployed*" (Lewis and Fowler, 2014). The structural and behavioral characteristics are often used to guide the recovery of the architecture. While the operational characteristics cannot be used to guide the recovery of the microservice-oriented architecture, they can be used to guide the transformation phase.

The migration of a monolithic application towards a MSA is a process that can be divided into two phases: (1) microservice architecture recovery from an OO monolithic source code and (2) the transformation of the monolithic source code towards the MSA.

2.1 Motivating Example: Information Screen

To better illustrate the problems we face during the migration process, we use throughout this paper an illustrative example of a display screen management system (e.g. an information panel in an airport). It is composed of the *DisplayManager* class that is responsible for handling the information to be displayed on the *Screen* class (see Fig 1). It does so through a *ContentProvider* class that implements methods for stacking content such as incoming messages (i.e. *Message* instances) or the current time (i.e. *Clock* instances). Finally, The *Clock* class uses an instance of the *Timezone* class to get the time based on its GPS location.

2.2 Microservice-based Architecture Recovery

This migration phase aims to decompose the classes of a monolithic application into clusters which will form the basis of a structurally and behaviorally valid microservice. Existing approaches propose different clustering techniques to maximize the quality of the microservices by maximizing the cohesiveness of the classes within a microservice while minimizing the coupling between microservices.

Figure 2 shows the results of applying a microservice recovery approach on the "Information Screen" system presented as motivating example. The recovery identifies 5 clusters, where each cluster contains one or more classes. In our example, the microservice candidate **MS1** is responsible for managing the display of various content on a screen, through its classes *DisplayManager* and *Screen*. A cluster is composed of two types of classes: internal and edge classes. An **internal class** is a class which does not contain any

dependencies with classes belonging to another cluster (e.g. *Screen* and *GpsLocation*). On the other hand, an **edge class** is a class which contains at least one dependency with a class from another cluster. These dependencies can be of any type (e.g. method invocation, constructor calls, or inheritance).

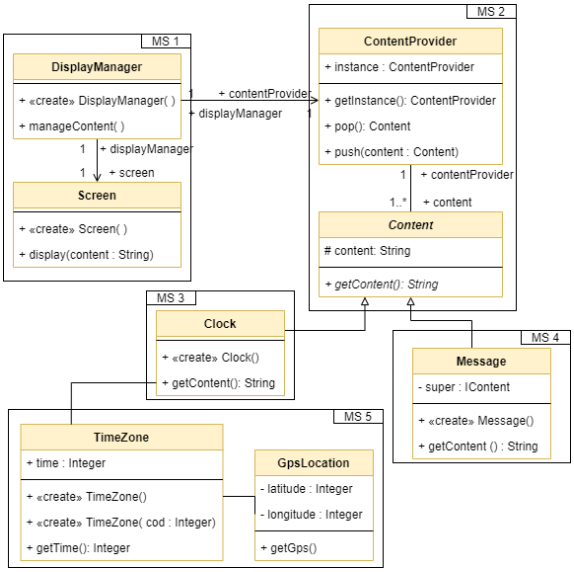

Figure 2: Recovered Microservice Architecture for the Information Screen application.

2.3 Transforming OO Source Code Towards an MSA One

The transformation phase involves materializing the identified MSA from the architecture recovery phase. It involves transforming object-oriented source code towards a MSA one. During this phase, each recovered cluster of classes is placed into its own microservice (i.e **microservice encapsulation**).

However, edge classes (e.g. *DisplayManager*, *Clock*), by definition, contain dependencies towards a class belonging to another cluster. These direct structural dependencies between the classes of different microservices are called **microservice encapsulation violations**. Before a microservice can be fully encapsulated, all violations must be resolved through refactoring methods which transform all OO-type dependencies into MS-type ones.

In this phase, we focus on the operational characteristic of an MSA to use message-oriented communication between different microservices. I.e., all procedural calls between classes belonging to different clusters (e.g. the method calls between *DisplayManager* and *ContentProvider*) must be restricted to a set of provided and required interfaces that define both the web services it provides and those it consumes. Furthermore, in an MSA, inter-process communications (IPC) calls are limited to value-based communication. I.e., only primitives and serialized data may be exchanged between microservice. However, a procedural call may pass object references between the invoking object and the invoked one. To fully encapsulate microservices, the mechanism of instance sharing between microservices must be resolved.

Beyond procedural calls there are other implicit dependencies between clusters that must be addressed to fully encapsulate the microservice candidates. The main OO mechanism that must be addressed is the inheritance mechanism. An inheritance violation appears when a class and its super-class are placed in different clusters (e.g, between *Message* and *Content*).

Lastly, operational characteristics of a microservice in which it must run on its own process and be deployable automatically, require an additional task of transformation. To conform to these two characteristics each microservice must be defined in their own independent project and they must be configured to generate their own deployable image. Both of these tasks must be addressed after each microservice have been properly encapsulated. In this paper, we focus on the code transformation phase that takes place after recovering the MSA.

3 MONOTOMICRO: A SEMI-AUTOMATED REFACTORING APPROACH

The goal of this approach is to transform an existing monolithic object-oriented source code to a MSA one by encapsulating the clusters identified in the first migration phase. To do so, we define a process composed mainly of four steps as presented in Figure 3. These steps include: (1) detecting encapsulation violations, (2) healing encapsulation violations, (3) packaging microservices and (4) deploying and containerizing microservices.

Step (1) involves detecting encapsulation violations that occur between clusters of classes which contain OO-type dependencies (e.g. instance creation, method invocation, and inheritance) to classes belonging to another cluster. These violations must be detected before they can be resolved. For each encapsulation violations, detection rules are defined and applied on the clusters.

In step (2), ”Healing encapsulation violations”, a set of transformation patterns is applied to the iden-

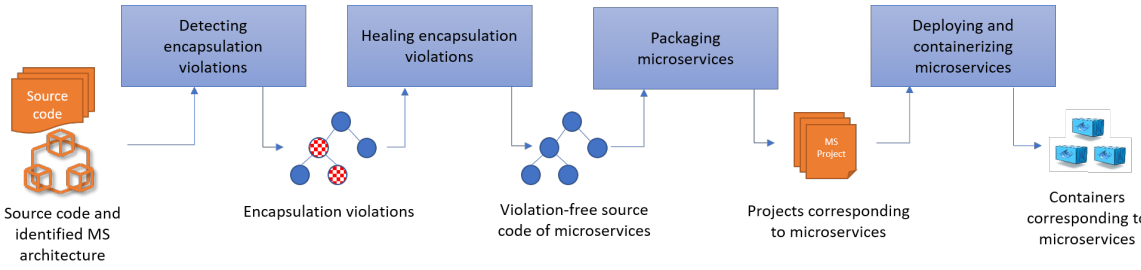

Figure 3: The Transformation Process using the MonoToMicro tool.

tified violations. As certain transformation patterns would create different additional violations, a transformation order is required to resolve all violations.

Next, in step (3), the violation-free microservices are packaged. To this end, a project is created for each microservice where the source code is generated. Then, the file structures and project dependencies are generated automatically.

Finally, in step (4), the microservice projects are containerized by generating instantiable images. For each microservice, an image description file is generated. In addition, a composition file is generated, which organizes and deploys all the microservices together.

In this paper, we focus on the first two steps of the transformation phase which contain the main scientific obstacles mentioned previously. More precisely, we focus on proposing a systematic approach to transform an OO application from the monolithic style to an MSA one, a set of transformation patterns, and an automated process that applies our systematic transformation approach. In Section 4, we present our first contribution, our approach to systematically detect microservice encapsulation violations. In Section 5, we present our second contribution for systematically healing microservice encapsulation violations through the use of transformation patterns. We leave the last two steps of the transformation phase for the implementation and we describe them in the section 6 along with the experimentation.

4 DETECTING MICROSERVICE ENCAPSULATION VIOLATION

To materialize the recovered microservice candidates from the source of object-oriented software, each recovered cluster of classes is encapsulated in its own microservice. However, upon encapsulation, OO dependencies between clusters are no longer permitted (i.e. encapsulation violations). To facilitate the detection of these encapsulation violations, a set of encapsulation violation rules are proposed to analyze the monolith:

(**Rule 1**): if a cluster's method invokes a method belonging to a class from another cluster then it is a method invocation violation.
(**Rule 2**): if a cluster's method accesses an attribute belonging to a class from another cluster then it is an access violation.
(**Rule 3**): if a cluster's class contains a reference targeting a class from another cluster then it is an instance violation.
(**Rule 4**): if a cluster's class inherits a class belonging to another cluster then it is an inheritance violation.
(**Rule 5**): if a cluster's method throws, catches or declares an exception defined in another cluster then it is a thrown exception violation.

Algorithm 1: Microservice Encapsulation Violation Detection Algorithm

```
   Input : clusters (sets of classes)
   Result: violations (a set of class couples)
1  violations ← { };
2  foreach ( cluster ∈ clusters ) {
3      foreach ( class ∈ cluster ) {
4          foreach ( dependency ∈ class ) {
5              if ((dependency → target) ∉ cluster)
                 then
6                violation ←
                   (class, dependency → target);
7              end
8          }
9      }
10 }
11 return violations;
```

These rules are applied on the AST representation of the OO source code. Using the target architecture description, the AST nodes that represent the classes in the OO source code are partitioned into clusters. Finally, the detection algorithm is applied on the clusters that are being encapsulated.

Algorithm 1 presents the procedure for detecting the encapsulation violations in each cluster. For each

cluster, it traverses each individual class and checks field accesses, method invocations, and type references. Each broken rule is represented as a typed violation based on the rule that was broken and is attributed to the cluster that is the source of the violation. Once each recovered cluster has been analyzed, the encapsulations violations can be healed.

5 HEALING ENCAPSULATION VIOLATIONS

To achieve the encapsulation of these microservices, the violations identified in the previous steps must be healed through the application of transformation rules. These transformation must either completely heal a violation or reduce it to another solvable type. We propose a set of transformation rules to heal the encapsulation violations identified in the first step. As mentioned previously, the encapsulation violations are as follows (1) attribute access, (2) method invocation, (3) instance creation, (4) and inheritance. We conclude by providing an order for resolving each type of violation to avoid creating more violations during the refactoring process.

5.1 Attribute Access

The direct attribute access violation involves direct access of attributes between classes of different microservices. It can be eliminated by applying the getter/setter pattern, which involves (1) limiting the access to the attributes to within its own class, (2) adding a method for returning the value of the attribute and a method for modifying the value of the attribute, and (3) refactoring the business logic to replace direct read/write of the attributes with method invocations to the get/set methods. This refactoring eliminates the direct access of attributes between classes of different microservices but creates a method invocation violation in its stead.

5.2 Method Invocation

The first encapsulation violation covered is the one caused by method invocation. To remove method calls between classes belonging to different microservices, they are refactored into interface-based calls. Consider our motivating example, where the *DisplayManager* invokes a method from *ContentProvider* depicted in Listing 1. For *DisplayManager* that uses the *ContentProvider*, both a required and a provided interface (*IContentProvider*) are created. The required interface is defined in the microservice of the class *DisplayManager* and the provided interface is defined in the microservice of the class *ContentProvider*. The references of the *ContentProvider* in *DisplayManager* are replaced with a reference towards the required interface *IContentProvider* (see Listing 2).

```
1  public class DisplayManager {
2      public ContentProvider cp;
3      ...
4      public void manageContent(){
5          String content = cp.pop().getContent(); ...
6      }
7  }
8  public class ContentProvider {
9      ...
10     public Content pop(){...}
11 }
```

Listing 1: Method Invocation Dependency in Java code.

```
12 public class DisplayManager {
13     public IContentProvider cp;
14     ...
15     public void manageContent(){
16         String content = cp.pop().getContent();
17     }
18 }
19 public class ContentProvider implements
       IContentProvider{...}
```

Listing 2: Decoupling method invocation by creating an interface.

Nevertheless, *ContentProvider* remains inaccessible to *DisplayManager* via the required interface. As microservice can only interact through inter-process communication protocols, an additional technological layer must be added. In the microservice with the provided interface, a *WebService* class is created to implement the interface and to expose the methods of *ContentProvider* to other microservices. For each publicly-available method of *ContentProvider*, a method is created in the *WebService* that acts as the intermediary to receive a request, forward it to the real method, and return the result of the operation. In the microservice with the required interface, a *WebConsumer* class is generated which implements the interface by preparing the network calls to its corresponding *WebService* class.

5.3 Instance Creation and Handling

After fragmenting the monolithic code into different microservices (i.e. clusters of classes), some instances can be created in one microservice and used (i.e. referenced) in others. To remove this type of violation, it is necessary to provide adequate answers to the following two questions: (1) How do we create an instance of a class belonging to another microservice? (2) When a given instance is referenced in several microservices, how do we ensure the sharing of

this instance while preserving the business logic of the application? These two questions are answered in two parts.

5.3.1 Instance Creation

To answer the first question, we propose a transformation combining two design patterns: the Factory Pattern and the Proxy Pattern. To address the instantiation of objects, we replace the instantiation of *ContentProvider* by the class *DisplayManager* with an interface acting as an object factory. This interface defines a method for each constructor of the *ContentProvider* class. For simplicity, the same provided/required interfaces defined in both microservices are used to define the object factory methods and the public methods of the *ContentProvider* class. The methods of this factory are implemented in the *WebService* class and call the corresponding constructor.

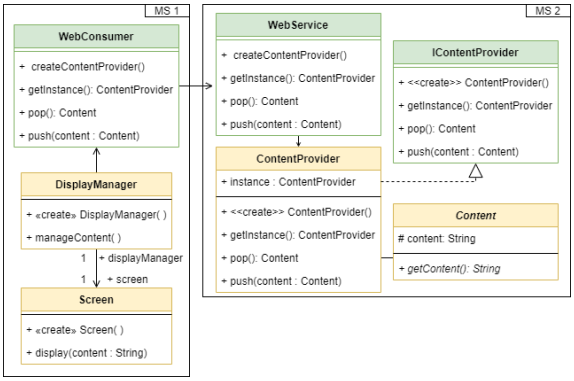

Figure 4: Replacing the class instantiation with the Factory Pattern (in yellow the monolith's class, and in green the classes created during the transformation).

The next step towards transforming the instance creation violation is based on the Proxy Pattern. The intent of the proxy pattern is to provide a surrogate or placeholder for another object to control access to it. In Figure 5, the proxy class (*ContentProviderProxy*) is a surrogate for the real class (*ContentProvider*) and acts as the surrogate for all method invocations. The role of the proxy class is to hide the absence of the class with which it is associated and which is referenced in one microservice and defined in another. Thus, it constitutes the place where calls to the class or its instances defined outside the microservice are transferred. Thus, for any class referenced in one microservice and defined in another, a proxy class is created in the microservice that references it. This class will have the same name as the real class, the same list of public methods and the same list of constructors. However, the implementations of its methods and constructors are different from the original class. Instead, the proxy class uses the *WebConsumer* class to interact with the real class definition.

5.3.2 Instance sharing

To preserve the business logic of the application, we propose a strategy to share an instance of a class defined in one microservice between several microservices. Furthermore, the proxy class handles the instances of this class. To differentiate, between the proxy class and the real class, the instances of the proxy class are called proxy instances, and instances of the real class are called concrete instances. A proxy instance has no state and they only reference their concrete instance. Each concrete instance is referenced by its proxy instances via the same unique reference, and any operation on a proxy instance is transferred to its concrete instance. Finally, all exchanges of the concrete instance (e.g. as a parameter, in or out) between the microservices are transformed into an exchange of the unique reference of the concrete instance.

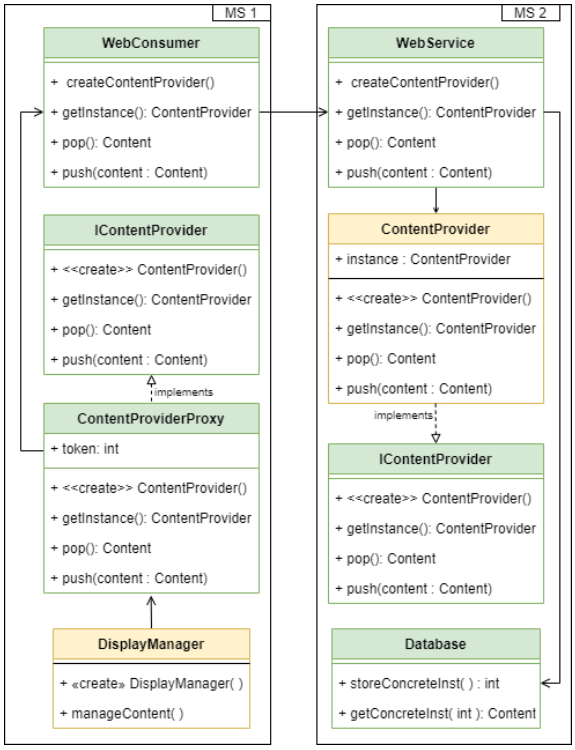

Figure 5: Replacing access to an object with the Proxy Pattern.

We implement the strategy as follows: A class is implemented to store all instances created in a microservice. When an concrete instance is created in a microservice through one of its factories, it sends the object to the corresponding storage class (in Fig

5, the *Database* class) to preserve it and this storage class returns a token for accessing the object. The microservice returns the token via its web service to the proxy instance, which stores it. Afterward, whenever the proxy instance receives a request, it transfers the request along with the token to the appropriate web service. The token provides the context whenever the web service is consumed, and the web service loads the concrete instance and invokes the corresponding method for that instance.

Furthermore, complex objects may be passed as a parameter between microservices. As microservice-type communication limited to passing simple data types such as strings and integers, we need to transform this type of exchange between microservices to be able to preserve the consistency of the application's business logic. A microservice may send or receive an object of class not belonging to it, and a microservice may send or receive an object of class that belongs to it.

For each situation, a token is passed between the microservices, but depending on the scenario the token is handled differently.

1. When a microservice receives an object of a class that does not belong to it, the microservice creates the relevant proxy and stores the token within it.
2. When a microservice receives an object of a class that belongs to it, the microservice uses the token to fetch it from its storage class.
3. When a microservice sends an object of a class that belongs to it, it must store the object and send the token.
4. When a microservice sends an object of a class that does not belong to it, the microservice must extract the token from its proxy and send it.

This approach enables the passing of complex objects between microservices, while the owner of the class handles the instances for other microservices. When a microservice receives a token, it is able to access the relevant object via the proxy class.

5.4 Inheritance Relationship

The inheritance violation is caused when a class that inherits from another class belonging to a different microservice are encapsulated. To heal this encapsulation violation, we propose a two-step transformation process inspired from (Alshara et al., 2015): (i) Uncoupling the child/parent inheritance with a double proxy pattern. (ii) Recreating subtyping via interface inheritance.

The inheritance link between two classes belonging to two different microservices contains several mechanisms that must be reproduced during the transformation process. The first mechanism is the extension of the definition of the parent by the child. A child class has access to the parent's attributes and methods. Furthermore, it may override the parent's methods. Finally, both child and parent method definitions may access each other's methods through the use of reference variables to the parent object or itself. The second mechanism that must be reproduced is the concept of polymorphic assignment. To preserve the first mechanism, we propose a double proxy pattern inspired by the work presented in (Alshara et al., 2015). In (Alshara et al., 2015), the authors propose a double delegate pattern to preserve the inheritance between the child and its parent class, residing each in a different components.

With this approach, when a child object is created, a parent object is also created as an attribute within the child object. The child class is refactored to forward invocations of any of its parent methods (e.g. using 'super' in JAVA) to its parent object attribute, which acts as a delegate. Furthermore, the child object is also stored in the parent object and acts as a delegate to preserve the dynamic calling of overridden methods (i.e. 'this' in JAVA). This effectively reduces the inheritance encapsulation violation into an instantiation violation and a method invocation violation which can be healed at a later point. When the parent class is abstract, the authors apply a proxy pattern in which the proxy class inherits from the parent class. The child class can then instantiate the proxy class as its delegate.

When this approach is applied in the context of microservices it necessitates two proxy classes as the delegate pattern creates an instance violation when the child accesses the parent object and vice versa. This is valid whether the parent is an abstract class or not. However, this approach requires refactoring the internal code of the child (and parent) class and requires informing the developer to use the delegate pattern instead of the native implementation. Instead, we propose a revised version that treats inheritance as a service more than an object, and reduces the refactoring of business classes.

5.4.1 Decoupling the Child/Parent inheritance

To transform the inheritance violation, the inheritance link between the child and parent classes must first be decoupled. First, a proxy class (e.g. *ContentConsumer*) is created and implements the methods defined by the common interface (*IContent*) extracted from the parent class. This class will act as the parent delegate for the the child class. The child class is refactored to extend it (e.g *ContentConsumer*) in-

stead of the parent class. Then, another proxy class (e.g. *MessageConsumer*) is defined to extend the parent class, and acts as the child delegate for the parent class. Figure 6 illustrates the severance of the inheritance link between the child (*Message*) and the parent classe (e.g. Content). However, the business logic between *Message* and *Content* is not preserved.

5.4.2 Recreating Subtyping through proxy inheritance

To preserve the internal logic between the child and the parent classes, the proxy classes are exposed as web services (as seen in section 5.3). This results in the creation of two web services (Figure 6). The parent proxy (e.g *ContentConsumer*) consumes the Parent web service upon creation. This has the effect of creating an *IContent* instance that acts as the superclass delegate. Whenever a method defined by the parent class is invoked by the child object, the parent object will be invoked via the parent web service. Furthermore, when the parent class references the instance, it will invoke the child object through the child proxy (e.g *MessageConsumer*) object.

5.4.3 Recreating Polymorphic Assignment through Interface Inheritance

Finally, the polymorphic mechanism is recreated by defining a child interface (e.g *IMessage*) which extends the parent interface (e.g. *IContent*), and which will be implemented by the child class (see Figure 6).

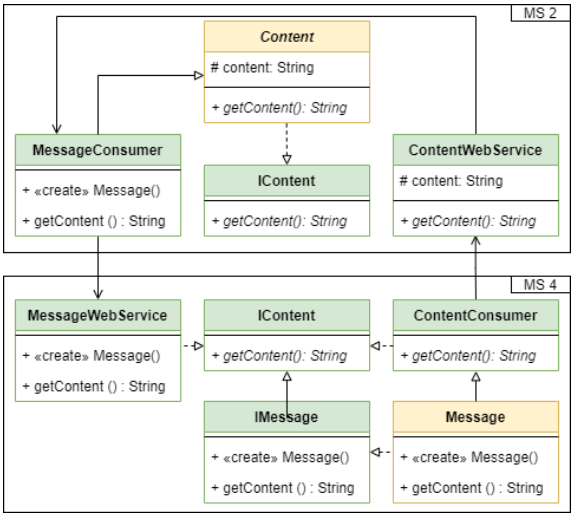

Figure 6: Polymorphic assignment can be recreated by applying an interface inheritance between the Parent Interface and the Child Interface.

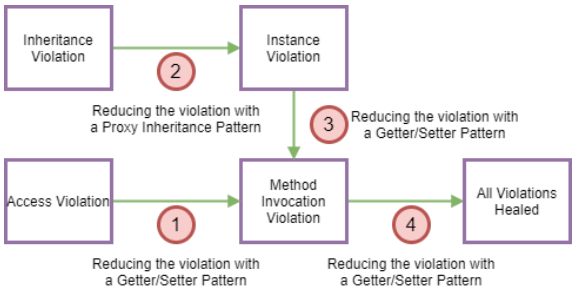

Figure 7: The transformation order of each microservice encapsulation violation.

5.5 Healing Violation Order

Once all the different microservice encapsulation violations have been identified, the transformation rules can be applied to completely encapsulate the clusters into their microservices. However, the application of these transformations in the wrong order may produce more encapsulation violations as a result (e.g. the application of the transformation for an attribute access violation produces a method invocation violation). The goal of this subsection is to present an order for applying these transformations to solve all encapsulation violation in one iteration.

Therefore, we propose an order of transformation to heal all these encapsulation violations which is presented in Figure 7. First, the attribute access violation is reduced as it adds public methods to its class that may be further refactored by inheritance violation. Then, the inheritance violations are reduced to an instance violation so all instance violations can be healed together. These instance violations are reduced to method invocation violations. Finally, the remaining method invocations violations are transformed into a set of web services. In the next section, we apply this transformation order when healing the encapsulation violations identified in the selected applications.

6 EVALUATION

6.1 Data Collection

To establish our experiment, we select a set of monolithic applications of various sizes (small, medium and large). We identified 6 applications whose source code is object-oriented and publicly available. The seventh application, *Omaje*, is a closed-source legacy application by Berger-Levrault, an international software editor. Table 1 provides some metrics on these applications.

Table 1: Applications on which the experiment was conducted.

Application name	No of classes	Lines of Code (LOC)
FindSportMates[1]	21	4.061
JPetStore[2]	24	4.319
PetClinic[3]	44	2.691
SpringBlog[4]	87	4.369
IMS[5]	94	13.423
JForum[6]	373	60.919
Omaje	1.821	137.420

6.2 Data Pre-processing: Microservice Identification

We used our approach, proposed in (Selmadji et al., 2020) to recovery an MSA represented as clusters of classes. This MS architecture recovery approach is semi-automatic as it combines the information extracted from the source code of the application (e.g. coupling and cohesion between classes, coupling compared to persistent data, etc.) as well as the architect's recommendations (e.g. granularity of microservices, their number, etc.). These clusters along with the source code of the applications are used as input for our approach. Table 2, refers to the extracted architectures for each application.

Table 2: Data on the applications being transformed.

Application	No. MSs	No. data classes	No. violations
Findsportmates	3	2	9
JPetStore	4	9	21
PetClinic	3	7	26
SpringBlog	4	8	104
IMS	5	18	113
JForum	8	37	1031

6.3 Research Questions and their Methodologies

To validate our approach, we conduct an experiment with the goal of answering the four following research questions.

[1]https://github.com/chihweil5/FindSportMates

[2]https://github.com/mybatis/jpetstore-6

[3]https://github.com/spring-petclinic/spring-framework-petclinic

[4]https://github.com/Raysmond/SpringBlog

[5]https://github.com/gtiwari333/java-inventory-management-system-swing-hibernate-nepal

[6]https://github.com/rafaelsteil/jforum2/

6.3.1 RQ1: What is the precision & recall of MonoToMicro approach when materializing a MSA?

Goal: The goal of this research question is to evaluate the correctness of the resulting microservice-based applications. By evaluating the syntactic and semantic correctness, we aim to demonstrate that our approach is able to transform the source code of a monolithic application towards an MSA one, while preserving the business logic.

Method: We measure the precision and the recall of our approach based on the syntactic and semantic correctness of the transformed microservices. It stands to reason that if the resulting MSA applications behaves in the same way as the monolithic applications then the business logic was preserved.

We consider that microservices are syntactically correct if there are no compilation errors. To measure the semantic correctness, we rely on whether the transformed microservices produce the same results compared to the functionalities of the monolithic applications at runtime. We identify a set of execution scenarios that are applied to both versions. We compare the outputs of the monolithic application with its MSA counterpart for each execution scenario. We consider that the transformation is semantically correct when the outputs generated by the monolith and the MSA are identical based on the same inputs.

When possible, the identification of execution scenarios is based on test cases defined by the developers of the monolithic applications (e.g. JPetStore). When test cases are not available, we identify a set of features and sub-features for each monolithic application (e.g. FindSportmates, IMS). From these features, we establish a set of user scenarios that cover all features of each application. These user scenarios are performed on the monolithic application and the results are saved. Then, these user scenarios are performed on the MSA, and the results are compared with those of the monolithic application. When they are identical we consider this as a passed test. Otherwise, they are marked as a failed test.

The precision is calculated by taking the number of tests passed by both architectures and dividing by the number of the tests passed by the MSA. While we calculate the recall by taking the number of tests passed by both architectures and dividing it by the number of tests passed by the monolith.

Due to time constraints related to the application packaging that is highly dependent on the technology of the monolith working with Spring, we study this research question with the *FindSportMates*, *JPetStore*,

and *InventoryManagementSystem* applications. For *JPetStore*, we ran the Selenium tests provided with the monolithic application. For *FindSportmates* and *Inventory Management System*, we manually ran these user scenarios.

Table 3: Type of violations caused by OO-type dependencies between microservices.

Application	No. Instances	No. Inheritances	No. Exceptions
Findsportmates	9	0	0
JPetStore	20	0	0
PetClinic	24	2	0
SpringBlog	95	7	2
IMS	110	3	0
JForum	1013	16	2

6.3.2 RQ2: What are the impacts of MonoToMicro on the performance?

Goal: The overall goal of our approach is to migrate while preserving the semantic behavior of an application. Moreover, an important aspect of the migration is that it must preserve the semantic without degrading drastically the runtime performance of the application. Therefore the primary goal of this RQ is to evaluate whether the performance impacts resulting from the migration of the monolithic application to microservices are negligible when compared to the original application.

Method: To answer RQ2, we rely on the execution time of user requests. The execution time measures the delay between the time when the request is sent and the time when the response is received by the user. We compare the execution time of both the monolith and the MSA. We establish a user scenario using *Omaje* to compare the performance of the monolithic application with its microservice counterpart. We chose *Omaje* for this evaluation because its business logic is the most complex of all 7 applications. To evaluate the performance, we simulate an increasing number of users connecting to both the MSA and the monolith, using JMeter[7] to simulate user load. As the number of user increases, we increase the number of instances of the microservice for both the monolith and the MSA. For the monolith, this involves duplicating the application. For the MSA, this involves duplicating the microservices involved in the current scenario. We consider that the refactoring results improve or maintain the quality and performance of the original code if the execution time difference between both architectures is negligible for the average user while the resource utilization is optimized. For our test we use a computer with an i7-6500U @ 2.5GHz and 16GB of ram.

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

6.4 Results

6.4.1 RQ1: What is the precision & recall of MonoToMicro approach when materializing a MSA?

Table 4 shows the results of **RQ1**. The results show that our approach has a 100% precision for *FindSportMates*, *JPetStore* and *InventoryManagementSystem* in terms of syntactic and semantic correctness. Therefore, our approach is able to preserve the business logic with a high precision. Furthermore, the results show that our approach has a 100% recall for *FindSportMates*, *JPetStore* and *InventoryManagementSystem*. The proposed transformation did not create a side-effect that was detected by failed functional tests that otherwise passed for the monolith. Therefore, our approach is able to preserve the business logic with a high recall. However, it should be noted that for *JPetStore* the Selenide test "testOrder" failed for both the monolithic version and the MSA version, as both checked the pricing notation using a period as a decimal separator while the testing was performed on a computer which defaults to using a comma instead.

Table 4: Number of tests performed based on the application and the resulting precision and recall based on those tests.

Application	No. Test	Precision	Recall
Findsportmates	7	100%	100%
JPetStore	34	100%	100%
IMS	36	100%	100%

6.4.2 RQ2: What are the impacts of MonoToMicro on the performance?

Figure 8 illustrates the number of users per scenario with the different architecture configurations. We can see, there is a small but negligible gain in performance upon the introduction of scaling for the MSA. The proposed transformations from MonoToMicro does not negatively affect the performance of the application. Our expectations were that by introducing additional network calls the performance of the migrated application would be affected negatively. However, in this scenario it was not the case. This was likely due to the parallelization aspect of scaling the requested service. By adapting the number of instances of microservices, the MSA was able to handle the increased

requests and compensate for the additional network layer. In fact, as the number of parallel requests increased, the MSA performed better (on average) compared to its monolith counterpart.

6.5 Threats to Validity

Internal Validity: First, we use static analysis techniques to analyze and detect microservice encapsulation violations in the source code of monolithic applications. This type of analysis may have two types of impact. The first one is that static analysis does not take into account polymorphism and dynamic binding when handling the instance violation. However, we deal this by preparing the worst case scenario where an instance dependency is created for every sub-type. The second impact is that the static analysis does not differentiate between the used and unused source code which may result in creating more services than required. The use of dynamic analysis could reduce both impacts. But it requires instrumenting and a considerable amount of tests. We consider our approach to be adequate for source code that is not based on frameworks (e.g. Spring for java). In this sense, we do not consider, for example, the injection of dependencies on-the-fly which is one of the properties of this type of framework. This type of dependency is not identified in our approach. Also, our approach does not consider dependencies and transformations linked to the reflexivity of programming languages. Thus, in our experiment, we identified and manually resolved this type of encapsulation violation.

External Validity: First, we consider that the use of a specific architecture recovery approach (Selmadji et al., 2020) to have an impact on the results of the transformation phase we highlight in the paper. We consider that the types of dependencies identified and their number strongly depend on the results of the architecture extraction phase used, and the use of another extraction method would impact the results of the proposed transformation process. However, the primary goal of this experiment is not to analyze the impact of the produced architecture, but the feasibility of migrating while preserving the intended behavior of the application. Second, the proposed approach is evaluated on monolithic applications that are implemented in Java. However, the obtained results can be generalized for any OO language. We argue for this generalization because all OO languages (e.g. C++, C#) are structured in terms of classes and their relationships are realized through the same mechanisms (e.g. method calls, field access, inheritance, etc.).

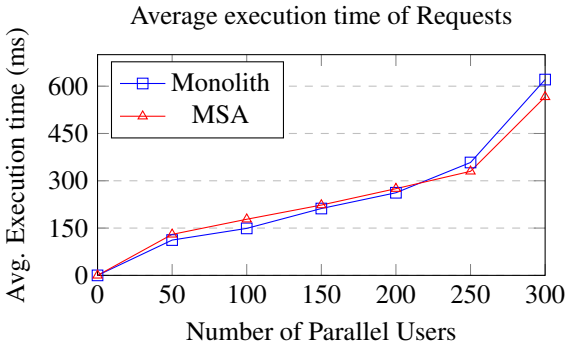

Figure 8: Average execution time of Omaje based on the number of users and the corresponding architecture.

7 RELATED WORK

Microservice-based Architecture Recovery: Recently, many works have been done on the process of extracting a MSA from an OO software, and several systematic reviews have been published on the subject (Abdellatif et al., 2021) (Fritzsch et al., 2019) (Francesco et al., 2017). These works can be categorized by the input and the process used to extract the architecture. Works such as (Baresi et al., 2017) use OpenAPI documentation of an application to extract its services. Similarly, an approach is proposed that leverages the documentation for component recovery (Chardigny and Seriai, 2010) (Chardigny et al., 2008). While (Selmadji et al., 2020) use both source code and expert recommendations as input for a clustering algorithm. (Jin et al., 2018) uses both source code analysis and dynamic behavior analysis to aid during clustering process. More broadly, component recovery proposes solutions to a similar problem. In (Kebir et al., 2012), genetic algorithms are used to recover components from an OO software. While in (El Hamdouni et al., 2010), relational concept analysis is used to extract a component-based architecture. Finally, (Shatnawi et al., 2018) uses dynamic analysis to identify components.

Transformation towards a MSA: The goal of refactoring is to extend the lifetime of an existing software product while preserving its functional behavior via code transformation to improve the structure of the source code. To the best of our knowledge, there does not exist any work in the transformation towards microservice that attempts to automate this process. Both an industrial survey (Francesco et al., 2017) and a systematic review (Fritzsch et al., 2019) on the subject of microservice migration, indicate a lack of tool to support the migration towards microservices. (Bigonha et al., 2012) proposes a technique for extracting modules from monolithic software architectures based on a series of refactoring

to modularize concerns through the isolation of code fragments. However, it focuses on the separation of concerns and does not address the deployment of independent modules as microservices. Besides, several works offer insights on the manual transformation such as (Fan and Ma, 2017), (Richardson, 2018), and (Amiri, 2018). (Fan and Ma, 2017) presents an experiment report where the authors share their migration process on an example. (Amiri, 2018) propose an extraction method accompanied by a manual transformation to validate their approach.

8 CONCLUSION

We have proposed an approach to automate the source code transformation phase of the process of migrating an application from the monolithic style to the MSA one. We focused on proposing a systematic approach to transform an OO application from the monolithic style to an MSA one, a set of transformation patterns, and an automated process that applies our systematic transformation approach. The evaluation of our approach revealed that the systematic transformation were possible and created syntactically and semantically correct microservices. Furthermore, an initial performance evaluation on an industrial case revealed that successfully-migrated microservices were able to scale with increased demands without degrading the overall service. In future works, we plan to take the lessons learned from the refactoring of these monolithic applications towards MSA ones and apply them to the identification of microservices to optimize the performance and the structure of the MSA.

REFERENCES

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., Boussaidi, G. E., Hecht, G., Privat, J., and Guéhéneuc, Y.-G. (2021). A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173:110868.

Alshara, Z., Seriai, A.-D., Tibermacine, C., Bouziane, H. L., Dony, C., and Shatnawi, A. (2015). Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. *GPCE*, page 55–64.

Alshara, Z., Seriai, A.-D., Tibermacine, C., Bouziane, H.-L., Dony, C., and Shatnawi, A. (2016). Materializing architecture recovered from object-oriented source code in component-based languages. *ECSA*, page 309–325.

Amiri, M. J. (2018). Object-Aware Identification of Microservices. In *2018 IEEE SCC*, pages 253–256.

Baresi, L., Garriga, M., and De Renzis, A. (2017). Microservices identification through interface analysis. In De Paoli, F., Schulte, S., and Broch Johnsen, E., editors, *Service-Oriented and Cloud Computing*, pages 19–33, Cham. Springer International Publishing.

Baskarada, S., Nguyen, V., and Koronios, A. (2020). Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems*, 60:428 – 436.

Bigonha, R. S., Terra, R., and Marco, T. (2012). An Approach for Extracting Modules from Monolithic Software Architectures. *WMSWM 2012*, (July 2016).

Chardigny, S., Seriai, A., Oussalah, M., and Tamzalit, D. (2008). Search-based extraction of component-based architecture from object-oriented systems. *ECSA*, pages 322–325.

Chardigny, S. and Seriai, A.-D. (2010). Software architecture recovery process based on object-oriented source code and documentation. *ECSA*, pages 409–416.

El Hamdouni, A.-E., Seriai, A., and Huchard, M. (2010). Component-based architecture recovery from object oriented systems via relational concept analysis. *CLA*, pages 259–270.

Fan, C. and Ma, S. (2017). Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In *2017 IEEE AIMS*, pages 109–112.

Francesco, P. D., Malavolta, I., and Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *ICSA*, pages 21–30.

Fritzsch, J., Bogner, J., Zimmermann, A., and Wagner, S. (2019). From monolith to microservices: A classification of refactoring approaches. *Lecture Notes in Computer Science*, page 128–141.

Jin, W., Liu, T., Zheng, Q., Cui, D., and Cai, Y. (2018). Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE ICWS*, pages 211–218.

Kebir, S., Seriai, A.-D., Chaoui, A., and Chardigny, S. (2012). Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. *C3S2E '12*, page 1–8.

Lewis, J. and Fowler, M. (2014). Microservices: a definition of this new architectural term.

Newman, S. (2019). *Building microservices: designing fine-grained systems*. O'Reilly Media.

Richardson, C. (2018). *Microservices Patterns*. O'Reilly Media.

Selmadji, A., Seriai, A.-D., Bouziane, H.-L., Mahamane, R., Zaragoza, P., and Dony, C. (2020). From monolithic architecture style to microservice one based on a semi-automatic approach. *ICSA*, pages 157–168.

Shatnawi, A., Shatnawi, H., Saied, M. A., Shara, Z. A., Sahraoui, H., and Seriai, A. (2018). Identifying software components from object-oriented apis based on dynamic analysis. In *ICPC*, page 189–199.

Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *JSS*, 146:215–232.