Service decoupler: Full dynamic decoupling in service invocation

Service Façade: Dynamic Decoupling in Service Invocation

DIONYSIS ATHANASOPOULOS, Victoria University of Wellington

Service-oriented software, built using concrete services, tends to change with difficulty, as analogously happens in the object-oriented software when it uses concretions. Towards achieving dynamic decoupling, adapter- and abstraction-based approaches have been proposed, which aim at developing a single service-client for alternative services, i.e. services that offer the same or similar functionality, usually through syntactically similar interfaces. However, the aforementioned approaches do not propose fully dynamic-decoupling solutions, i.e. service invocations are not adapted in a completely seamless and dynamic way. Technically, the existing approaches realize the invocation of alternative services by using delegation, which implements complex service-translation scripts, (re-)configured at compile time. To go beyond the state-of-the-art, we propose the *Service Façade* pattern that leverages the case of syntactically similar interfaces, offering a unified service interface, based on which service invocations are seamlessly bound and adapted at runtime (late binding and dynamic adaptation, respectively). Technically, our pattern replaces delegation with service polymorphism, without needing the development of service-translation scripts. Moreover, the pattern structure, along with its client, are closed to modifications when new alternative services are incorporated.

CCS Concepts: •**Software and its engineering** → **Software design engineering;**

Additional Key Words and Phrases: Service invocation, service interface, dynamic decoupling, façade, polymorphism.

ACM Reference Format:

Dionysis Athanasopoulos. 2017. Service Façade: Dynamic Decoupling in Service Invocation. *EuroPLoP* (July 2017), 10 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Famous vendors (e.g. `Google`, `Amazon`) provide access to their resources via adopting the Service-oriented Architecture (SoA) style [Erl 2005]. Service interface consists of a set of public operation signatures[1]. SoA software, built using concrete services, tends to change with difficulty (rigid software), as analogously happens in the object-oriented software when it uses concretions.

State-of-the-art. To decouple SoA software from concrete services (achieving the *service loose coupling* principle [Erl 2009]), various approaches have been proposed in the literature that range from *static*, realized at compile time (static binding), to *dynamic*, *where a single client is used at runtime*. In dynamic decoupling, client should be seamlessly (without code re-configuration and modifications) adapted at runtime to use alternative services. Alternative are the services that offer the same or similar functionality, usually through syntactically similar interfaces. Dynamic-decoupling approaches can be divided into two categories: *adapter*-based (e.g. [Ponnekanti and Fox 2004; M. Davydov 2005; Cavallaro and Nitto 2008; Athanasopoulos et al. 2009; Kongdenfha et al. 2014]) and *abstraction*-based (e.g.

[1]Without loss of generality, we assume that service interface and client are specified in the Java language.

978-1-4503-4848-5/2017/07...$15.00
DOI: 0000001.0000001

[Ruokolainen and Kutvonen 2006; Taher et al. 2006; Athanasopoulos et al. 2011; Liu and Liu 2012]) approaches. In the first category, client is developed with respect to a concrete service and maps it to alternative services, translating their invocations [Kongdenfha et al. 2014]. The task of developing service-translation scripts is very time-consuming and development-demanding, since service interfaces are usually very complex and large. For instance, Amazon Web service EC2[2] provides 78 operations, whose documentation is 812 pages. In the second category, client is developed with respect to an abstract service. Since abstract services are not available, existing approaches (manually or automatically) define abstract services that represent alternative services. In both categories, service mappings are identified, towards defining subtyping relations. Since alternative services are autonomously developed by different providers (without extending common or standardised interfaces), syntactically similar service interfaces do not necessarily have subtyping relations. To overcome this problem, a relaxed subtyping version is defined by existing adapter- (e.g. [Athanasopoulos et al. 2009]) and abstraction-based approaches (e.g. [Andrikopoulos and Plebani 2011; Athanasopoulos et al. 2011]). Technically, due to the relaxed subtyping, the existing approaches adopt delegation, instead of polymorphism, for forwarding service invocations, implementing complex translation scripts.

Limitations of the state-of-the-art. The approaches of both categories do not offer fully dynamic-decoupling solutions, i.e. service invocations are not adapted in a completely seamless and dynamic way. In particular, when service adapter (resp. service abstraction) is used, client is tightly coupled to a concrete service (resp. an abstract service), *without having access to the unmapped operations of alternative services*. Moreover, due to the relaxed subtyping, *unmapped fields possibly exist in the data-types of the alternative services*. Consequently, missing data-type values (i.e. data-types that cannot be instantiated by the client at compile-time) make possibly infeasible the runtime adaptation. The final limitation is that service adapter and abstraction are not closed to modifications in incorporating new alternative services, violating the Open-Closed Principle (OCP) [Martin 2002]. Specifically, *translation scripts are modified, incorporating delegation to the operations of a new service*.

Contribution. To overcome the limitations of the state-of-the-art, we are inspired by the classical façade pattern [Gamma et al. 1994] and its service version [Erl 2009; Demange et al. 2013]. The general idea of the classical façade is to offer a unified API for a set of software components. Its service version states that service façade should sit between a service and its client's contract, eliminating the tight coupling between them. However, the existing service façade provides only the general idea of the pattern, without specifying its structure and the way that a unified interface is built in the case of syntactically similar interfaces. To bridge this gap, *we specify the structure and the technical description of Service Façade that leverages the case of alternative services, offering a fully dynamic-decoupling solution and conforming to OCP. We also contribute on describing the structures and limitations of service-adapter and -abstraction patterns*, since their approaches have focused on describing their techniques for identifying service mappings, without modelling the underlying patterns.

Our contribution is summarized and structured as follows. Section 2 defines the related pattern structures and limitations. Section 3 defines the proposed pattern. Section 4 describes the core implementation details of our pattern. Section 5 presents an illustrative example for adopting our pattern. Finally, Section 6 summarizes our approach and discusses its future research directions.

2. RELATED PATTERNS

We describe the service adapter and abstraction patterns in Sections 2.1 and 2.2, respectively.

[2] aws.amazon.com/documentation/ec2

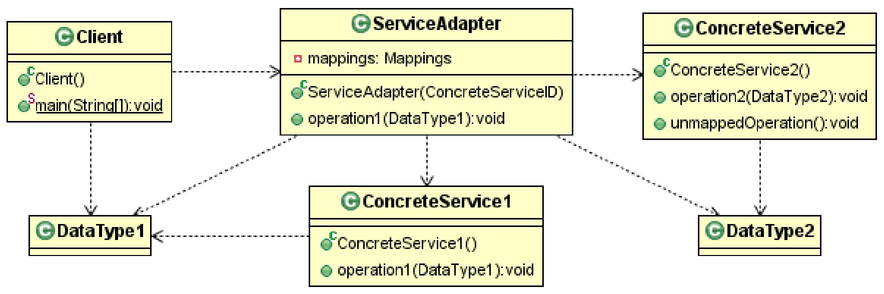

Fig. 1. The UML class diagram of the pattern structure of *Service Adapter*.

2.1 Service Adapter

We detail the structure and limitations of *Service Adapter* in Sections 2.1.1 and 2.1.2, respectively.

2.1.1 *Structure of the Service-Adapter pattern.* Based on its structure (Fig. 1), `client` is developed with respect to the `ServiceAdapter` class, whose operations are the same to one of the alternative services (e.g. `ConcreteService1`). Each operation of `ServiceAdapter` maps and translates operation invocations (e.g. the invocation of `operation1` of `ConcreteService1` to the invocation of `operation2` of `ConcreteService2`). The translation script is constructed based on the service `mappings` (`ServiceAdapter` field). Moreover, `client` selects at compile time the currently used service via providing value for the field `ConcreteServiceID`. If an alternative service is needed to substitute the previously selected service, `client` re-selects another service and is re-configured to use it.

2.1.2 *Limitations of the Service-Adapter pattern.* We divide the limitations into two groups, those related to the client-side and those related to the pattern-side.

– The limitations, related to the client-side, are the following:
  A. **Tight coupling**: client is essentially developed with respect to a concrete service;
     a. *limited access*: it can access only the operations of a concrete service (e.g. `unmappedOperation` of `ConcreteService2` is not accessible);
     b. *coarse access*: it is (re-)configured to use alternative services at compile time.
– The limitations, related to the pattern-side, are the following:
  B. **Delegation**: SoA developer writes complex service-translation scripts;
     – *problematic delegation*: since there are not necessarily mappings between all the data-type fields, translation script may miss values for unmapped fields (e.g. in `DataType2`), which are hidden from client.
  C. **Semi-dynamic decoupling**: to avoid translation failure, client possibly intervenes at runtime to provide values for unmapped data-type fields.
  D. **OCP violation**: SoA developer incorporates a new alternative service via modifying the code of each operation of `ServiceAdapter`, i.e. translation scripts from each operation of `ConcreteService1` to the mapped operations of the new alternative service.

2.2 Service Abstraction

The notion of abstract service is analogous, but not exactly the same, with that in the object-oriented domain. In particular, as proposed in [Athanasopoulos et al. 2011], an abstract service is defined based on the following relaxed version of the behavioural subtyping [Liskov and Wing 1994]:

– the operations of abstract service represent the common/similar operations of alternative services;

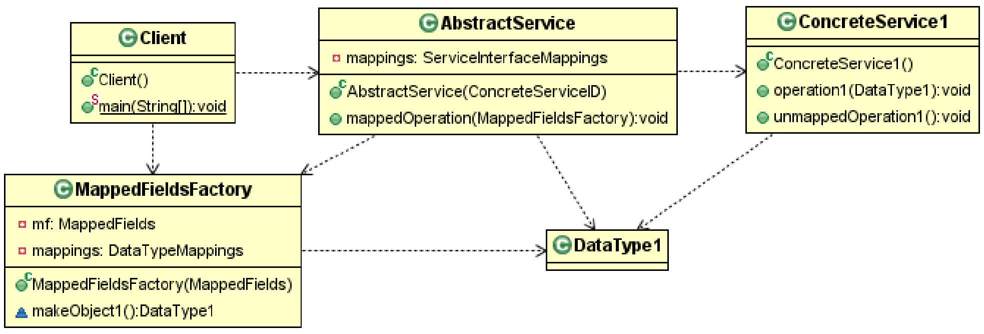
Client
Client()
main(String[]):void
AbstractService
mappings: ServiceInterfaceMappings
AbstractService(ConcreteServiceID)
mappedOperation(MappedFieldsFactory):void
ConcreteService1
ConcreteService1()
operation1(DataType1):void
unmappedOperation1():void
MappedFieldsFactory
mf: MappedFields
mappings: DataTypeMappings
MappedFieldsFactory(MappedFields)
makeObject1():DataType1
DataType1

Fig. 2. The UML class diagram of the pattern structure of *Service Abstraction*[3].

– the input (resp. output) data-types of abstract service represent the common/similar fields of input (resp. output) data-types of alternative services;
– the fields of an input (resp. output) data-type of abstract service should be more and more generic (resp. less and less specific) from that of alternative services.

This relaxed relation is captured by the mappings between the abstract and alternative services. Since the input/output data-types of service operations are usually complex tree-structured XML schemas[4], existing approaches (e.g. [Athanasopoulos et al. 2011]) consider only the leaf elements of the tree structures, since leaves are related to the data needed in the service-translation process (see the illustrative example in Section 5).

Sections 2.2.1 and 2.2.2 detail the structure and limitations of *Service Abstraction*, respectively.

2.2.1 *Structure of the Service-Abstraction pattern.* Based on its structure (Fig. 2), `client` is developed with respect to the `AbstractService` class, whose `mappedOperations` represent common/similar operations of alternative services. The implementation of `mappedOperation` delegates operation invocations, guided by the mappings (see the `ServiceInterfaceMappings` and `DataTypeMappings` fields of `AbstractService` and `MappedFieldsFactory`, respectively). Since `mappedOperation` uses delegation (instead of polymorphism), `AbstractService` is modelled as an abstract class. `MappedFieldsFactory` represents the common/similar fields of alternative data-types. It further provides factory methods (e.g. `makeObject1`) in order to instantiate a data-type (e.g. `DataType1`), which are used by `AbstractService`. Moreover, `client` selects at compile time the currently used service via providing value for the field `ConcreteServiceID`. If an alternative service is needed to substitute the previously selected service, `client` re-selects another service and is re-configured to use it.

2.2.2 *Limitations of the Service-Abstraction pattern.* We divide the limitations into two groups, those related to the client-side and those related to the pattern-side.

– The limitations, related to the client-side, are the following:
  – **Restricted loose coupling**: client is developed with respect to an abstract service;
    a. *limited access*: it can access only mapped operations (e.g. it cannot access `unmappedOperation1` of `ConcreteService1`);
    b. *coarse access*: it is (re-)configured to use alternative services at compile time.
– The limitations, related to the pattern-side, are the same with those of *Service Adapter*.

[3]We depict in this figure only one concrete service and one data-type of its operations in order to keep it simple.
[4]www.w3.org/TR/xmlschema-2

3. THE PROPOSED SERVICE FAÇADE PATTERN

We describe the synopsis, context, problem, forces, solution, and consequences of our pattern.

3.1 Synopsis

SoA software is developed with respect to a unified interface of alternative services (all of their operations are accessible), offered by our *Service Façade* pattern. It is defined in such a way that client is fully loose-coupled from the alternative services. SoA developer does not write complex service-translation scripts, since service polymorphism is used. Finally, SoA developer incorporates new alternative services, without modifications on both client and pattern sides.

3.2 Context

Services are autonomously developed by different providers, without extending common or standardised interfaces. Thus, SoA software, built by using concrete services, tends to change with difficulty, since alternative services do not necessarily have identical interfaces. To decouple SoA software from concrete services, it should be seamlessly and dynamically adapted to use alternative services, without any re-configuration at compile time.

3.3 Problem

Offering fully dynamic-decoupling solution is challenging, since the interfaces of alternative services are usually syntactically similar, but not identical. The related patterns of service adapter and abstraction do not offer fully dynamic-decoupling solutions.

3.4 Forces

We divide the forces in two groups, those related to the client-side and those related to the pattern-side.

– The forces, related to the client-side, are the following:
  1. client wants to access all of the operations of alternative services, without being tight coupled to concrete services.
– The forces, related to the pattern-side, are the following:
  2. SoA developer wants to avoid developing complex service-translation scripts;
     a. having at his disposal beforehand all of the data-type values, required for invoking services.
  3. he wants to be all service invocations bound and adapted at runtime.
  4. he wants to incorporate new alternative services, without client and pattern modifications.

3.5 Solution

The pattern structure of the proposed *Service Façade* is depicted in Fig. 3.

3.5.1 *Pattern structure.* To allow the achievement of the Forces, our pattern enhances the structure of façade, defining three abstraction layers: façade, abstract-service, and concrete-service layers (clear separation of concerns).

Façade layer. Given that the general purpose of façade is to unify software components, promoting the client decoupling, it is the proper pattern for achieving Force 1. However, façade does not define how the unified interface is built in the case of syntactically similar interfaces. To leverage this case, our pattern defines the following `ServiceInterfaceFacade` and `MappedFieldsFacade` classes, one for service operations and one for input/output data-types. `ServiceInterfaceFacade` provides two categories of operations: (i) those that represent the similar/common operations of alternative services (e.g. `mappedOperation`); (ii) those that correspond to the unmapped operations (e.g. `unmappedOperation1` of

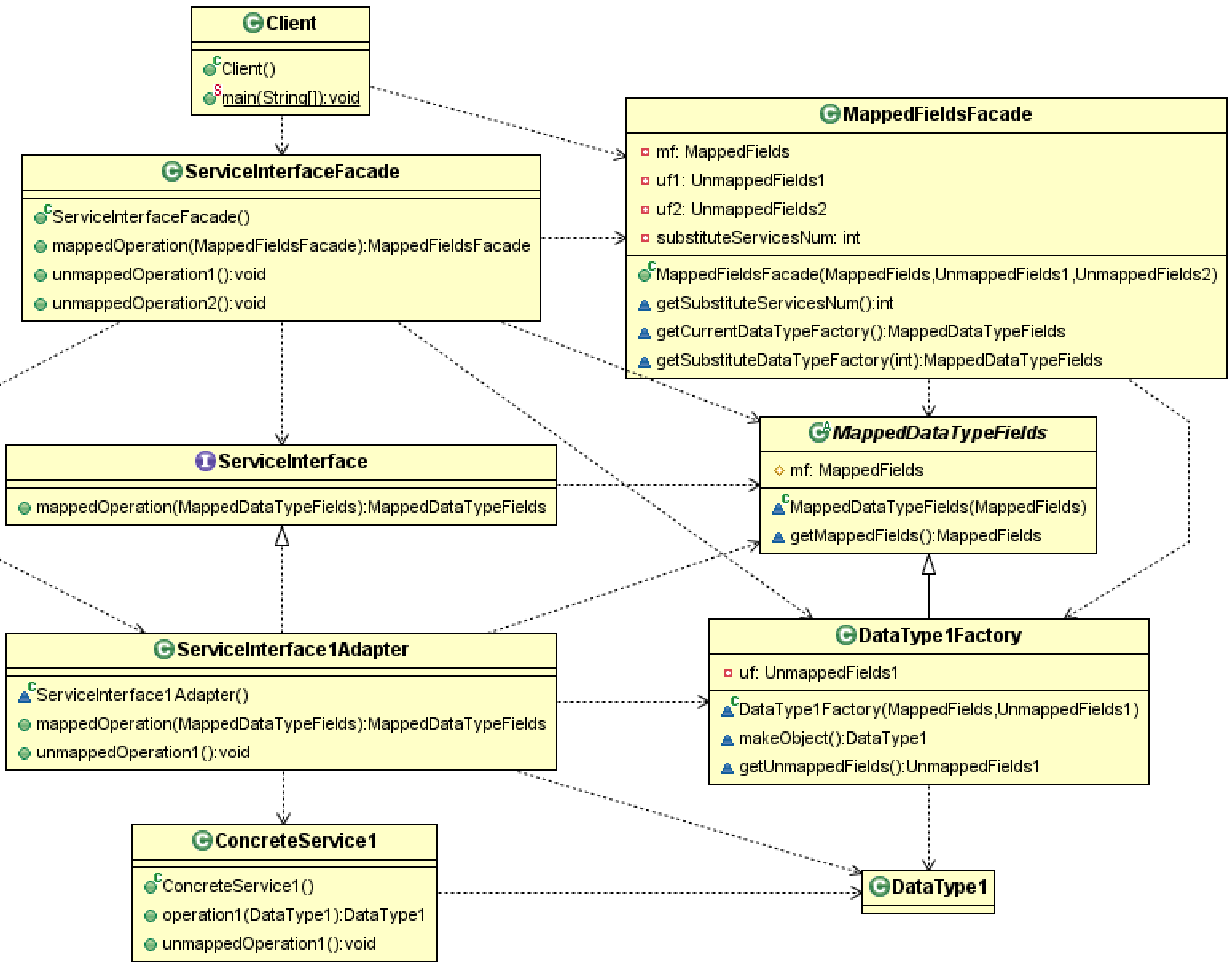
Client
Client()
main(String[]):void
MappedFieldsFacade
mf: MappedFields
uf1: UnmappedFields1
uf2: UnmappedFields2
substituteServicesNum: int
MappedFieldsFacade(MappedFields,UnmappedFields1,UnmappedFields2)
getSubstituteServicesNum():int
getCurrentDataTypeFactory():MappedDataTypeFields
getSubstituteDataTypeFactory(int):MappedDataTypeFields
ServiceInterfaceFacade
ServiceInterfaceFacade()
mappedOperation(MappedFieldsFacade):MappedFieldsFacade
unmappedOperation1():void
unmappedOperation2():void
ServiceInterface
mappedOperation(MappedDataTypeFields):MappedDataTypeFields
MappedDataTypeFields
mf: MappedFields
MappedDataTypeFields(MappedFields)
getMappedFields():MappedFields
ServiceInterface1Adapter
ServiceInterface1Adapter()
mappedOperation(MappedDataTypeFields):MappedDataTypeFields
unmappedOperation1():void
DataType1Factory
uf: UnmappedFields1
DataType1Factory(MappedFields,UnmappedFields1)
makeObject():DataType1
getUnmappedFields():UnmappedFields1
ConcreteService1
ConcreteService1()
operation1(DataType1):DataType1
unmappedOperation1():void
DataType1

Fig. 3. The UML class diagram of the pattern structure of *Service Façade*[3].

ConcreteService1). In a similar way, two categories of fields are provided in MappedFieldsFacade: (i) those that represent similar/common data-type fields (e.g. MappedFields); (ii) those that correspond to the unmapped fields (e.g. UnmappedFields1 of DataType1Factory).

MappedFieldsFacade provides one unified constructor, whose first argument corresponds to the similar/common data-type fields, and the remaining arguments correspond to the unmapped data-type fields offered from each alternative service. In this way, client configures the *Service Façade* to use either one service (providing values for the first field and values for one of the unmapped data-type fields) or a combination of alternative services (e.g. triple of arguments etc.). The other design decision of offering a different constructor for each combination of alternative services is not traceable (i.e. very high number of constructors). Since client provides beforehand all the data-type values, required for invoking alternative services, the unified constructor achieves Force 2.a. Notably, client provides data-type values, without selecting services, achieving the other part of Force 1. Finally, based on the provided (combination of) values, the binding to the proper service is performed at runtime (identifying dynamically to which services the provided data-type values belong), achieving Force 3.

Abstract-service layer. To avoid the development of complex service-translation scripts, delegation is replaced by *service polymorphism*, achieving Force 2. In particular, an abstract service is (manually or automatically) defined, whose interface (see `ServiceInterface` in Fig. 3) provides the common/similar operations of alternative services (e.g. `mappedOperation` of `ServiceInterface`). To realize service polymorphism, subclassing, between the interfaces of the abstract and the alternative services, is defined. In particular, for each alternative service (e.g. `ConcreteService1`), an adapter that extends `ServiceInterface` is defined, which further forwards the invocations to the alternative service (e.g. `ServiceInterface1Adapter`). An adapter uses the factory class of a data-type (e.g. `DataType1Factory`) in order to take an instance of the data-type (using its `makeObject` operation). `DataType1Factory` extends the abstract class, `MappedDataTypeFields`, which includes the common/similar data-type fields. Concluding, providing the necessary input data-type fields (e.g. `MappedFields`, `UnmappedFields1`, and `UnmappedFields2`), the late binding and dynamic adaptation are automatically performed, based on the aforementioned structure of the service polymorphism.

Concrete-service layer. In the third abstraction layer, our pattern defines wrappers for each concrete service and for each input/output data-type.

3.5.2 *Incorporating new alternative services.* To incorporate a new alternative service, only extensions and additions are required, achieving Force 4. In particular, the required steps are the following:

1. add in `MappedFieldsFacade` the `UnmappedFields` of the new service;
2. extend the constructor of `MappedFieldsFacade` with a new argument;
3. add so many `unmappedOperations` in `ServiceInterfaceFacade` as many the unmapped operations of the new service are;
4. add a new `DataTypeFactory` that extends `MappedDataTypeFields`;
5. add a new `ServiceInterfaceAdapter` that extends `ServiceInterface`;
6. add `ConcreteService` and `DataType` wrappers for the new service.

3.6 Consequences

3.6.1 *Benefits*

– The benefits, related to the client-side, are the following:
  A. **Fully loose coupling**: client is essentially developed with the interface of an abstract service;
    a. *unified interface*: client accesses all of the operations of alternative services;
    b. *seamless access*: client is bound to alternative services at runtime;
      – the layers of abstract and concrete services are hidden to client.
– The benefits, related to the pattern-side, are the following:
  B. **Service polymorphism**: SoA developer does not develop complex service-translation scripts;
    – *realization*: he defines subclassing between abstract and concrete services.
  C. **Dynamic decoupling**: all alternative services are encapsulated;
    a. *late binding*: service invocations are bound at runtime;
    b. *dynamic adaptation*: invocations are adapted at runtime, without code modifications.
  D. **OCP conformance**: client and pattern are closed to modifications when new alternative services are incorporated.

3.6.2 *Liabilities*

– The liabilities, related to the client-side, are the following:
  A. **Provision of extra values**: client provides values not only for the currently invoked service, but also for its substitute services that may not necessarily be needed.

– The liabilities, related to the pattern-side, are the following:
  B. **Service mappings**: the façade and abstract-service layers depend on the identified mappings;
    – a new alternative service should have similar operations with the existing one.
  C. **Unified constructor of the data-type façade**: if a high number of alternative services exists, then the arguments number of the unified constructor is also high;
    – it can be split into smaller constructors, based on the similarity of the alternative services.

```
1. public MappedFieldsFacade mappedOperation( MappedFieldsFacade input ){
2.     MappedFieldsFacade result = null;
3.     try{
4.         MappedDataTypeFields dt1 = input.getCurrentDataTypeFactory();
5.         ServiceInterface si1 = new ServiceInterface1Adapter();
6.         MappedDataTypeFields resultMappedDataTypeFields = si1.mappedOperation( dt1 );
7.         result = new MappedFieldsFacade( resultMappedDataTypeFields.getMappedFields(), ((DataType1Factory)
                                                   resultMappedDataTypeFields).getMappedFields(), null );
8.     }catch( Exception e ){
9.         System.err.println( "Exception caught, I will call operation2 of ConcreteService2." );
10.        for( int i = 0; i < input.getSubstituteServicesNum(); ++i ){
11.            MappedDataTypeFields dt2 = input.getSubstituteDataTypeFactory( i );
12.            ServiceInterface si2 = new ServiceInterface2Adapter();
13.            try {
14.                MappedDataTypeFields resultMappedDataTypeFields = si2.mappedOperation( dt2 );
15.                result = new MappedFieldsFacade( resultMappedDataTypeFields.getMappedFields(), null,
                                 ((DataType2Factory) resultMappedDataTypeFields).getMappedFields() );
16.                break;
17.            } catch( Exception e1 ){
18.                System.out.println( "Exception caught, I will call operation of the next substitute service." );
19.            }
20.        }
21.    }
22.    return result;
23.}
```

Fig. 4. The code snippet of an indicative Java implementation of a mapped operation between alternative services.

4. CORE IMPLEMENTATION DETAILS

The source code of the Java implementation of our pattern is online available at this location[5]. In this Section, we describe the core implementation details of the façade abstraction-layer of our pattern.

Concerning the implementation of `MappedFieldsFacade`, its unified constructor implements the following constraint checks[6]. Client should provide values for the `MappedFields` and for at least one of the `UnmappedFields` arguments. `UnmappedFields1` corresponds to the input data-type of the currently used service, `UnmappedFields2` to the first substitute service, and so on. The order of the unmapped fields are aligned based on their subtyping relations (e.g. `ConcreteService2` can substitute `ConcreteService1`, `ConcreteService3` can substitute `ConcreteService1` and `ConcreteService2`, and so on). Moreover, the constructor measures the number of the implicitly selected substitute services, based on the provided values for unmapped data-type fields.

Regarding the implementation of `ServiceInterfaceFacade`, `mappedOperation` uses the operation `getCurrentDataTypeFactory` of `MappedFieldsFacade` for retrieving an instance (based on the user-provided values) of the input data-type of the currently used service (Fig. 4 (line 4)). If there is a

[5]https://ecs.victoria.ac.nz/foswiki/pub/Main/DionysisAthanasopoulos/ServiceFacade.zip

[6]Due to the algorithmic simplicity of the constructor implementation, we do not provide its implementation code.

runtime need to substitute the currently used service (e.g. failure exception), `MappedOperation` uses `getSubstituteDataTypeFactory` of `MappedFieldsFacade` for retrieving an instance of the input data-type of a substitute service (Fig. 4 (lines 10-20)). `MappedOperation` catches the failure exception (Fig. 4 (line 8)) and dynamically invokes the substitute service (Fig. 4 (line 14)). Finally, `mappedOperation` creates an instance of `MappedFieldsFacade` for wrapping the output data-type of the invoked operation (Fig. 4 (line 7)), which will be returned to the client.

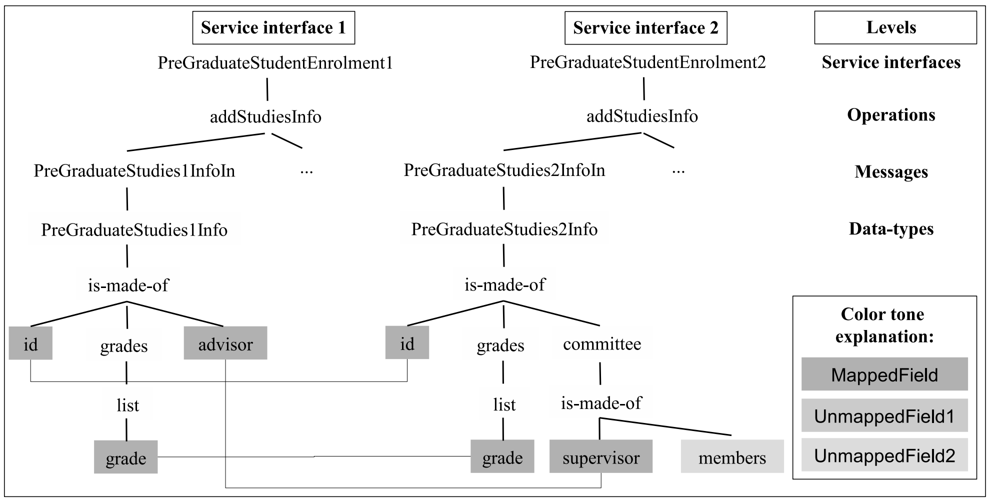

Fig. 5. The hierarchical representation of two example service interfaces, along with the mappings of their data-types.

5. EXAMPLE – INSTANTIATING THE PROPOSED SERVICE FAÇADE AND CONFIGURING ITS CLIENT

As an example, we take two services, whose overall functionality is to enrol pre-graduate students in a university department and to store information about their studies (e.g. course grades, etc.). We depict in Fig. 5, the hierarchical representation of their mapped `addStudiesInfo` operations[7]. The figure also depicts the mappings between the data-type fields of the operations, along with their decomposition in mapped and unmapped fields.

The code snippet of an indicative Java implementation of client for our example is provided in Fig. 6. Client configures the *Service Façade* to use `PreGraduateStudentEnrolment1` via providing values for the mapped fields, `id`, `grade`, and `advisor` (Fig. 6 (lines 3-6)). Since the input data-type of `PreGraduateStudentEnrolment1` does not include unmapped fields, client provides an empty instance of the corresponding data-type factory, instead of `null` value[8] (Fig. 6 (line 9)). Moreover, client provides values for the unmapped field `members` of `PreGraduateStudentEnrolment2`, configuring the *Service Façade* to use `PreGraduateStudentEnrolment2` for potential substitution (Fig. 6 (lines 7-8)). The source code of the whole example is also online available at this location[5].

6. CONCLUSIONS AND FUTURE WORK

We proposed the *Service Façade* pattern that leveraged the case of alternative services, offering a fully dynamic-decoupling solution and conforming to OCP. In this way, we overcome the limitations of the

[7]The interfaces are represented in a high-level way, i.e. without the technical details of the WSDL and XML languages.
[8]We assume that `null` argument value means that the corresponding service will not be used by the pattern.

```
public static void main(String[] args) throws Exception {
    StudentEnrolmentServiceInterfaceFacade f = new StudentEnrolmentServiceInterfaceFacade();
    MappedFields mf = new MappedFields();
    mf.id = 1;
    mf.grade = 90;
    mf.advisor = "Tom";
    PostGraduateStudiesInfoMappedFields uf2 = new PostGraduateStudiesInfoMappedFields();
    uf2.committeeMembers = new String[] { "Joe" };
    f.addStudiesInfo( new MappedStudiesInfoFieldsFacade( mf, new PreGraduateStudiesInfoUnmappedFields(), uf2 ) );
}
```

Fig. 6. The code snippet of an indicative Java implementation of client for the illustrative example.

related service-adapter and -abstraction patterns. We also provided the core implementation details of our pattern and an illustrative example of showing how our pattern is practically used.

Our future research direction is to extend our pattern in order to cover the decomposition of the whole tree structure (instead of its leaves) of the data-types of services. On top of this, we aim at developing a tool, incorporated in an Integrated Development Environment (e.g. Eclipse[9] plugin), supporting the automated generation of the pattern structure for a set of alternative services.

REFERENCES

V. Andrikopoulos and P. Plebani. 2011. Retrieving Compatible Web Services. In *International Conference on Web Services*. 179–186.

D. Athanasopoulos, A. Zarras, and V. Issarny. 2009. Service Substitution Revisited. In *Automated Software Engineering*. 555–559.

D. Athanasopoulos, A. Zarras, P. Vassiliadis, and V. Issarny. 2011. Mining Service Abstractions. In *International Conference on Software Engineering*. 944–947.

L. Cavallaro and E. Di Nitto. 2008. An Approach to Adapt Service Requests to Actual Service Interfaces. In *Software Engineering for Adaptive and Self-Managing Systems*. 129–136.

A. Demange, N. Moha, and G. Tremblay. 2013. Detection of SOA Patterns. In *International Conference on Service-Oriented Computing*. 114–130.

T. Erl. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR.

T. Erl. 2009. *SOA Design Patterns*. Prentice Hall PTR.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

W. Kongdenfha, H. R. M. Nezhad, B. Benatallah, and R. Saint-Paul. 2014. Web Service Adaptation: Mismatch Patterns and Semi-Automated Approach to Mismatch Identification and Adapter Development. In *Web Services Foundations*. 245–272.

B. Liskov and J. M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1811–1841.

X. Liu and H. Liu. 2012. Automatic Abstract Service Generation from Web Service Communities. In *International Conference on Web Services*. 154–161.

M. Davydov. 2005. *Ease Web Services Invocation with Dynamic Decoupling*. Technical Report. IBM.

R. C. Martin. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

S. Ponnekanti and A. Fox. 2004. Interoperability Among Independently Evolving Web Services. In *International Middleware Conference*.

Toni Ruokolainen and Lea Kutvonen. 2006. Service Typing in Collaborative Systems. In *International Conference on Interoperability for Enterprise Software and Applications*. 343–353.

Y. Taher, D. Benslimane, M-C. Fauvet, and Z. Maamar. 2006. Towards an Approach for Web Services Substitution. In *International Database Engineering and Applications Symposium*. 166–173.

[9] https://eclipse.org