How Adaptation and Transformation Complement Each Other to Potentially Overcome Signature Mismatches on Object Data Types on the Basis of Test-Cases

Dominic Seiffert

Software Engineering Group
University of Mannheim
Mannheim, Germany
Email: seiffert@informatik.uni-mannheim.de

Oliver Hummel

Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: hummel@kit.edu

***Abstract*—The challenge of providing fully automated adaptation is tackled by many approaches in literature. Thereby, the class of signature mismatches presents the challenge of matching object data types that provide the same semantics but are syntactically incompatible. We explain in this paper how adaptation, complemented by transformation, can potentially solve the problem on the basis of test cases in the object-oriented world.**

Keywords*–*automated adaptation; signature mismatches; object data types; test cases.

I. Introduction

Software building blocks such as objects need to connect their interfaces in order to create new functionality. Unfortunately, this connection is not always possible because of signature mismatches. A simple example for a signature mismatch occurs, when the interfaces to connect have different names. A more challenging task is a signature mismatch for deviating parameter or return object data types. For object data types a subclass instance can be delivered when a super class instance is expected, according to Liskov Substitution Principle [2]. This is not possible, however, when the subclass relationship does not exist, even for a semantically equal instance of a different type.

Signature mismatches are tackled by many approaches in literature, towards the goal of providing fully automated adaptation. Thereby, an adapter gets interposed which handles the message wiring between the involved software building blocks. However, current approaches from the object-oriented communities lack the ability to overcome signature mismatches on object data types that are syntactically incompatible but provide the same semantics. We believe that it is a need to challenge this task in order to further improve fully automated adaptation.

In the remainder of this paper, we propose adaptation complemented by transformation, on the basis of test cases, as the possible solution on overcoming signature mismatches for object data types in object-oriented programming. The necessary background information on adaptation and a more detailed description of the problem is illustrated in Section 2. In Section 3 the solution is proposed. Section 4 refers to related work. Section 5 states the conclusion.

II. Background

Signature mismatches can potentially be solved by an adapter that gets interposed between the software building blocks [3]. The Gang of Four Object Adapter pattern [4] provides a well-known solution in the world of object-oriented programming. The following adaptation example in Figure 1 explains the pattern in more detail: Let A and B be two simple object data types that are not connected by hierarchy, and simply provide each the methods set(int) and get():int, to set and retrieve an integer value. Let further be StackA and StackB two simple stack implementations, which are also not connected by type hierarchy. An instance of StackA allows an instance of A to be pushed on and popped off, whereas StackB allows the same for an instance of B. The Client depends on the StackA interface but wishes to use the methods push(B) and pop():B from the adaptee StackB. The StackA interface provides the methods push(A) and pop():A that use the parameter and return type A. Therefore, a signature mismatch on object data types occurs for A and B.

In an idealized scenario, the Adapter implements the StackA interface and wraps the adaptee. The adaptee receives forwarded messages from the Client via the Adapter. More exactly, the Client invokes the set(A) method and the Adapter forwards the messages to the set(B) method of the adaptee StackB. The same happens vice versa for the pop methods and their return values.

In order to support a fully automated process, Hummel and Atkinson [7] proposed the idea of specifying the expected adapter's functionality in a test case that is used then during the adaptation process for checking the semantics of the adaptee. Figure 2 provides a sample test case for the previous example, where the client expects an adapter StackAB to be created that adapts StackA on StackB. This is specified in line no. 1 by the import statement. The expected functionality is simple: The object data type A gets instantiated and the value 5 is set to it in line no. 8. This instance is pushed on the adapter StackAB in line no. 9. In line no. 10 the pop() method gets invoked, which is expected to deliver an instance of A again. An invocation of get() on this instance is expected to deliver the value 5. This is required in order to pass the test as specified by the assertEquals statement in line no. 11.

Such a unit test case and a candidate to adapt serves as an input for the adapter generation tool [8] developed by Seiffert and Hummel [9]. The tool generates adapters that are based on the Managed Adapter Pattern [5], which is based on Fowler's identity pattern [6, p. 195]. The Managed Adapter Pattern solves the problem of the Gang of Four Object

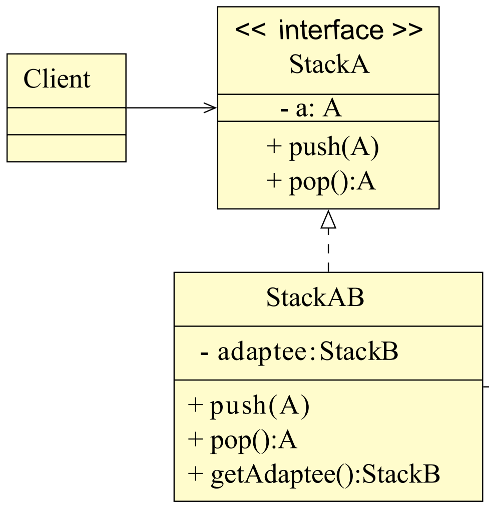

Figure 1. Adapter StackAB where StackB is the adaptee.

```
import adapter.StackAB;

public class TestCase extends junit.framework.TestCase {

  public void test {
   StackAB adapter = new StackAB();
   A a = new A();
   a.set(5);
   adapter.push(a);
   A a = adapter.pop();
  assertEquals(5, adapter.get());
  }
}
```

Figure 2. Test case where StackA adapts StackB.

Adapter Pattern when the adaptee expects its own type as a parameter argument or delivers it as a return type. The tool further overcomes signature mismatches on primitive data types, parameter permutations and a subset of object data types, namely arrays and collections [10] that share common semantics.

In the following example the situation has changed, as illustrated by UML diagram in Figure 3: Let the Client depend on StackOwnerA and let StackOwnerB play the role of the adaptee. The Client wants to adapt StackOwnerA to StackOwnerB, which requires that the set(StackA) method must be matched on the set(StackB) method, and the get():StackA method must be matched on the get():StackB method. Therefore, a signature mismatch on object data types occurs for StackA and StackB. A potential solution on this problem is proposed in the following section.

III. Adaptation and Transformation on the Basis of Test Cases

When the client delivers the StackA instance to the adapter StackOwnerAB, by invoking the set(StackA) method, the first idea is to simply reuse the adapter AdapterBA from the previous example, as illustrated by Figure 4. Reusing an adapter requires the existence of an adapter repository, which has already been proposed by Gschwind [11].

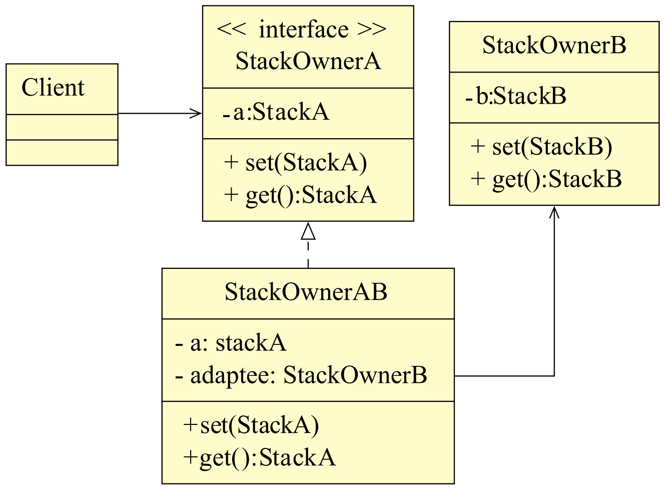

Figure 3. Adapter StackOwnerAB where StackOwnerB is the Adaptee.

```
 1 StackOwnerAB {
 2 set(StackA stackA) {
 3  StackOwnerB adaptee = new StackOwnerB();
 4  StackBA reusedAdapter = null;
 5  If(repository.entryExists(adaptee, stackA.getClass())){
 6   reusedAdapter = repository.getEntry(adaptee, stackA.
         getClass()));
 7  }else{
 8    reusedAdapter = generateAdapter(adaptee, stackA.
          getClass());
 9    repository.setMapping(stackA, reusedAdapter);
10  }
11  reusedAdapter.setAdaptee(stackA);
12  adaptee.set(reusedAdapter); //forwarded.
13 }
14 }
15 Repository {
16  ...
17  public Object generateAdapter(Class from, Class to){
18   //here the adaptation happens
19  }
20 }
```

Figure 4. Reusing an existing adapter.

More precisely, in line no. 5 the repository is checked whether an adapter StackBA already exists for the arriving instance stackA. If the repository cannot find an adapter StackBA, a new adapter StackBA must get generated, as indicated by line no. 17, and set to the repository. Nasehi and Maurer [12] propose that test cases should be equipped with a standard API, which we believe would support our adapter generation tool, because it takes takes test cases as an input. However, if test cases are not provided, the idea is to generate them automatically, which is not a futuristic scenario. For example, Galler and Aichernig [13] provide an overview on current test case generation tools, where some tools seem to be practical.

In line no. 11, the arriving StackA instance needs to be set as the adaptee for the adapter StackOwnerAB. This is different to a ”regular” adapter that usually creates its adaptee instance itself.

The adapter StackBA is assumingly not a subclass of StackB. Therefore, StackBA and StackB are mismatching object data types, and no instance of StackBA can be delivered

```
1 import adapter.StackOwnerAB;
2
3 public class TestCase extends junit.framework.TestCase {
4
5   public void test {
6    StackOwnerAB adapter = new StackOwnerAB();
7    A a = new A();
8    a.set(5);
9    StackA stackA = new StackA();
10   stackA.push(a);
11   adapter.set(stackA); //forward.
12   ...
13  }
14 }
```

Figure 5. Test case snippet where StackOwnerA adapts StackOwnerB.

when an instance of StackB is expected, as intended by line no. 12. To realize this, some cases must be considered, before the following background:

- A class declared as final can not be sub-classed.
- When the expected type is an interface type any instance of a class, which implements this interface, may be provided.

Therefore, the following cases are possible:

1) If StackB is a class type declared as final, then the adapter StackBA is not able to subclass it. Therefore, it cannot be forwarded, which is a severe limitation of this approach and calls for a more generally usable solution.
2) If StackB is a class type not declared as final, the adapter can subclass StackB. Therefore, it can be forwarded.

For the first case that StackBA cannot subclass StackB or StackB is not declared as an interface type, the following two potential solutions exist: First, if we would know the methods and their parameter values which were invoked on the arriving instance of StackA, i.e. the protocol, we could reuse the adapter StackAB. This is realized by reusing the protocol of instance stackA, in order to invoke the same methods with the same values on the adapter StackAB. The adapter StackAB provides the getAdaptee():StackB method as indicated by Figure 1. Therefore, by calling stackAB.getAdaptee(), the adapter StackAB returns an instance of StackB, which can be forwarded to the set(StackB) method of StackOwnerB. This idea requires that instances are monitored and that this information could be retrieved accordingly by a protocol. The idea of encapsulating such protocol information is proposed by Pintando [14] by using ”gluons”. These are special objects embedding interaction protocols between components. We believe that this protocol information could also be provided by a test case. For example, the test case snippet in figure 5, where StackOwnerA wants to adapt StackOwnerB, provides from line no. 7 to line no. 10 the protocol information that an instance of A is set a value of 5 and that this information is pushed on a StackA instance. Therefore, the idea is, to extract the protocol information from the test case.

The second solution to this problem is, in our believe, to provide a transformation mechanism. Transformation transforms ”state”. This means transformation extracts state from one instance and sets it to another instance. Thereby, the

```
StackOwnerAB {
set(StackA stackA){
  StackB stack = transform(stackA);
  StackOwnerB adaptee = new StackOwnerB();
  adaptee.set(stack);
 }
}
```

Figure 6. Adapter StackOwnerAB uses transformation.

```
Transformer.transform(StackA stackA):StackB {
  StackB stackB = new StackB();
  stackB.push(stackA.pop());
  return stackB;
}
```

Figure 7. A simple transformation example.

instances are not necessarily of a different type. The instances can mismatch by their state only. For example, an instance of StackA with the values 1,2,3 pushed on it, is different to another instance of StackA keeping the values 3,2,1. Compared to adaptation transformation requires different matchings for the methods, parameters and return types. Figure 6 and Figure 7 show how the adapter StackOwnerAB would implement a transformation mechanism, to transform the state of an instance of StackA to an instance of StackB. The transform method is assumingly provided by a class Transformer.

The challenge for this transformation is that the output parameter of the pop method, from StackA, is matched on the input parameter of the push method, from StackB, which is already challenged by the web service community ([15] [16]). But transformation, in the context of adaptation, does not mean that output parameters of methods are matched on input parameters of other methods only. Examples can be more challenging, as an adapter potentially needs to involve functionality from other classes, which are not necessarily provided as potential adaptees, in order to provide the expected functionality. For illustration purpose, let the Client expect the standard deviation to be calculated for the values he puts on the stack. This functionality needs to be assembled by the adapter. The adapter thereby works more like a facade [4]. Again, the client specifies the expected functionality by a test case in the first step. Thereby, he adds a specific comment to it, as shown by Figure 8 in line no. 10. This comment states that he expects the standard deviation to be calculated. The idea is, that this comment gets extracted out of the test case. Data mining tools could be used then, to determine that the standard deviation should be calculated. The standard deviation requires the calculation of the mean and variance. This knowledge could be retrieved from an ontology for instance. The information gets provided to the adapter. Figure 9 shows the potential transformation function that takes an array specified as *functionality* as an input. This array is filled with the content *mean*, *variance* and *deviation*, representing the information retrieved from the comment and ontology. Therefore, the adapter has to involve other ”helper”-classes, such as Mean, Variance and StandardDeviation, which it retrieves from the repository. Instances of these classes can either be already existing adapters or generated adapters. The latter are generated on the basis of equipped test cases or

```
import adapter.StackAB;

public class TestCase extends junit.framework.TestCase {

  public void test {
   Adapter adapter = new Adapter();
  adapter.push(10);
  adapter.push(5);
  adapter.push(9);
  //I want the standard deviation to be calculated.
  assertEquals(2.16f, adapter.pop());
  }
}
```

Figure 8. Test case enriched by comment.

```
 Transformer.transform(StackA stackA, String[]
      functionality) {
   Adapter mean = repository.getClass(functionality[0]);
   Adapter variance = repository.getClass(functionality
        [1]);
   Adapter standardDev = repository.getClass(
        functionality[2]);
  StackB StackB = new StackB();
       //calculate Mean
  int mean = mean.calculate(values);
     //calculate Variance
   int variance = variance.calculate(mean);
      //calculate StandardDeviation
  int result = standardDev.calculate(variance);
   StackB.push(result);
   return StackB;
  }
```

Figure 9. Transformation from StackA to StackB.

generated test cases.

IV. Related Work

The Morabit approach presented by Brenner et al. [17] is based on a prototype component framework that uses built-in tests. However, it does not consider the possibility of automated test case generation. The Java Object Instrumentation Environment, named JOIE, is one of the early adaptation approaches proposed by Cohen et al. [18]. JOIE allows the transformation of Java classes through byte code modification. For example, it allows the insertion of new code into the class to be modified and the change of data types or method names. Another approach proposed by Reiss [19] modifies the adaptees on the source code level to meet the expected requirements made by the Client. For such an invasive modification license problems can be problematic [20]. But we believe that it becomes necessary for adaptation on a deeper nested level. For example, let us assume that the content of the stackB instance, arriving at the set(StackB) method of StackOwnerB, should be displayed by a class Display which offers the methods show(OtherStack). The invocation of these methods should be inserted into the set(StackB) method provided by StackOwnerB either by source-code or byte-code modification. This requires that also the adapter generation of the adapter that adapts StackB to OtherStack gets inserted invasively. Kell [21] provides a rule-based approach named Cake which allows the transformation of object data types, but it requires the user writing mapping rules. We believe that the possible mappings should be detected and verified automatically during the adaptation process. The problem of service adaptation is already challenged by the web-service community by orchestration. For example, Eslmamichalandar et al. [22] provide an overview on web-service adaptation approaches. But approaches in the web-service community rely on xml-schema matching or ontology based reasoning. This is appropriate for syntactic matching, but it does not solve the problem in the object-oriented world when two syntactically incompatible but semantically equal type instances need to be adapted and potentially transformed. Gschwind [11] proposes the idea of an adapter repository where adapters can be retrieved by some meta information. Non existing adapters need to be provided by the client. The idea of creating them automatically is not considered.

V. Conclusion and Future Work

In this paper, we have provided potential solutions on overcoming signature mismatches on object data types on the basis of test cases. The potential solutions are adaptation and transformation. Transformation complements adaptation but works differently, as it extracts the state of an instance and sets it to another, and thereby uses different method matchings than adaptation. Future work should implement the proposed ideas in this paper around our adapter generation tool to provide a working application. The overcoming of signature mismatches is a big challenge, therefore, more research is needed in this area to further push fully automated adaptation.

References

[1] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, “Towards an engineering approach to component adaptation,” in Architecting Systems with Trustworthy Components, vol. 3938, 2009, pp. 193–215, ISSN: 0302-9743.

[2] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” in ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, no. 6, 1994, pp. 1811–1841.

[3] D. Yellin and R. Strom, “Protocol specifications and component adaptors,” in ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 19, no. 2, 1997, pp. 292–333.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 1994, ISBN: 9780321700698.

[5] O. Hummel and C. Atkinson, “The managed adapter pattern: Facilitating glue code generation for component reuse,” in Formal Foundations of Reuse and Domain Engineering, S. Edwards and G. Kulczycki, Eds. Springer Berlin Heidelberg, 2009, pp. 211–224.

[6] M. Fowler, Patterns of Enterprise Application Architecture. Addison-Wesley, 2003, ISBN: 978-0321127426.

[7] O. Hummel and C. Atkinson, “Automated creation and assessment of component adapters with test cases,” in Component-Based Software Engineering, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer Berlin Heidelberg, 2010, pp. 166–181.

[8] “The Adapter Generation Tool,” 2015, URL: http://oliverhummel.com/adaptation/tool.zip [accessed: 2015-01-02].

[9] D. Seiffert and O. Hummel, “Improving the runtime-processing for component adaptation,” in Lecture Notes in Computer Science, Springer, J. Favaro and M. Morisio, Eds. Springer Berlin Heidelberg, 2013, pp. 81–96.

[10] D. Seiffert and O. Hummel, “Adapting arrays and collections: Another step towards the automated adaptation of object ensembles,” in Lecture Notes in Computer Science, Springer, I. Schaefer and I. Stamelos, Eds., vol. 8919. Springer International Publishing Switzerland, 2015, pp. 348 – 363.

[11] T. Gschwind, Type Based Adaptation: An Adaptation Approach for Dynamic Distributed Systems, A. Coen-Porisini and A. van der Hoek, Eds. Springer Berlin Heidelberg, 2003.

[12] S. Nasehi and F. Maurer, “Unit tests as api usage examples,” in International Conference on Software Maintenance (ICSM). IEEE, 2010, pp. 1–10, ISSN: 1063-6773.

[13] S. Galler and B. Aichernig, “Survey on test data generation tools,” in International Journal on Software Tools for Technology Transfer, vol. 16, no. 6, pp. 727–751, 2013, ISSN: 1433-2787.

[14] X. Pintando and B. Junod, “Gluons: Support for software component cooperation,” in Object Frameworks, D. Tsichritzis, Ed. Universite De Geneve, Switzerland, 1992.

[15] L. Cavallaro, E. Di Nitto, P. Pelliccione, M. Pradella, and M. Tivoli, “Synthesizing adapters for conversational web-services from their wsdl interface,” in Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2010, pp. 104–113.

[16] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, “Semi-automated adaptation of service interactions,” in Proceedings of the 16th international conference on World Wide Web, 2007, pp. 993–1002.

[17] D. Brenner, C. Atkinson, B. Paech, R. Malaka, M. Merdes, and D. Suliman, “Redicing verification effort in component-based software engineering through built-in testing,” in Information Systems Frontiers, vol. 9, no. 2-3, 2007, pp. 151–162.

[18] G. A. Cohen, J. S. Chase, and D. L. Kaminsky, “Automatic program transformation with joie,” in Proceedings of the USINEX 1998 Annual Technical Conference, 1998, pp. 167 – 178.

[19] S. P. Reiss, “Semantics-based code search,” in Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). IEEE Computer Society, 2009, pp. 243 – 253, ISBN: 978-1-4244-3453-4.

[20] U. Hoelzle, “Integrating independently-developed components in object-oriented languages,” in ECOOP 93Object-Oriented Programming, 1993, pp. 36–56.

[21] S. Kell, “Component adaptation and assembly using interface relations,” in OOPSLA ’10 Proceedings of the ACM international conference on Object oriented programming systems languages and application, vol. 45, no. 10, 2010, pp. 322–340.

[22] M. Eslamichalandar, K. Barkaoui, H. Reza, and H. Motahari-Nezhad, “Service composition adaptation: an overview,” in Second International Workshop on Advanced Information Systems for Enterprises (IWAISE), 2012, pp. 20–27.