The Challenge of Transforming State in the Adaptation Objects

Dominic Seiffert

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

***Abstract*—When a provided interface and an expected interface need to be connected with each other, this connection is sometimes hindered by signature mismatches. In the world of object-oriented programming where objects play a key role, one important signature mismatch problem occurs when the expected interface expects an object data type that is per se incompatible, although semantically equal, to the object data type delivered by the provided interface. For example, suppose a birthday calendar is the parameter type expected by the expected interface, but another birthday calendar from another developer is the provided parameter type, then a mismatch on object data type occurs. To solve this problem, adaptation is one potential solution. However, because some programming language constructs are not amenable to adaptation, a mechanism based on transformation can be used instead to complement the adaptation process. The challenge is to retrieve the state of the object instance delivered by the provided interface, and to set it to an instance of the object type by the expected interface. In the literature, this problem, however, has been not tackled so far by the object-oriented community. This position paper aims to highlight this challenge and motivate the development of future adaptation tools to solve this problem fully automatically. The challenge is illustrated by typical transformation examples, ranging from more or less trivial to quite challenging tasks.**

Keywords–Signature Mismatches; Object Adaptation; State Transformation

I. INTRODUCTION

Software building blocks [1] can be connected by their provided and expected interfaces in order to create functionality. Unfortunately, direct connection is not always possible because of signature mismatches. A very simple example of a signature mismatch occurs when the provided interface and the interface to connect have different names although they actually provide the same functionality. This problem arises when the vocabulary in the problem domain is different from the vocabulary in the solution domain [2]. Another signature mismatch problem occurs in the object-oriented world where not only primitive data types can be used as parameters and return types but also object types. An important feature of an object is that it holds a state. When an object is defined as a parameter type or return type for a provided interface but the expected interface supports an object type which is per se incompatible, although semantically equal, a solution is required to solve this mismatch. Adaptation provides such a solution because it overcomes signature mismatches [3]. Several approaches have been proposed in the world of object-oriented programming to tackle this problem of signature mismatches such as [4] [5]. However, the problem of matching objects that hold state has not been considered. It is a crucial problem, however, because objects play an important role in object-oriented development. Therefore, it is important to highlight this challenge in order to improve adaptation in the object-oriented world.

The remainder of this paper is structured as follows: Section 2 provides background information on adaptation, especially in the object-oriented world. Section 3 delivers a motivational example to show that this problem is not just an academic one. Section 4 provides an overview of the different problems in more detail. Section 5 provides a conclusion and an outlook on future work.

II. BACKGROUND

In the early 90's Rittri [6], Runciman and Toyn [7] and Cosmo [8] proposed an approach for tackling the problem of retrieving functionality by matching types, laying the basis for the problem of type isomorphism. Two types A and B are definably isomorphic (A $\cong_d$ B) iff there exist functions (λ-terms) M:A $\rightarrow$ B and N:B $\rightarrow$ A such that M $\circ$ N = I_B and N $\circ$ M = I_A, where I_A and I_B are λx : A.x and λx : B.x, the identities of type A and B. The terms are called invertible.

A minor formal explanation is presented in [8] on page 38, which states that any two types A and B are equivalent if we can write two simple transformations h:A $\rightarrow$ B and $h^{-1} : B \rightarrow$ A such that

1) for any function f:A, $h(f) : B$ and $h^{-1}(h(f)) = f$
2) for any function g:B, $h^{-1}(g) : A$ and $h(h^{-1}(g)) = g$.

The main subject of interest are the transformation functions h and h^{-1}, which are written manually by the developers [8][page 38]. Existing approaches, however, have focused on the area of functional languages where objects do not play a key role. Object were first introduced by object-oriented programming [9]. In this context, Pierce defines in [10] on the pages 225 - 228, an object as a data structure that encapsulates some internal state and offers methods to access this state. The internal state is thereby represented by mutable fields that are shared among the methods and are inaccessible from the outside. The latter is a feature named *encapsulation*, i.e., the internal fields should only be accessed from the outside by methods, as first promoted by Parnas [11], by the principle of information hiding. As stated by Cosmo [8] on page 213, in the presence of state, former results on type isomorphism are no longer guaranteed to be complete. Therefore, it is important to tackle this problem in the object-oriented world by adapting superficially incompatible object data types that are equal from the semantic viewpoint to match one another.

 ISBN: 978-1-61208-463-3

The main challenge herein is to retrieve the state from an instance delivered by a provided interface and to set this state to an instance of an object type that can then be delivered to an expected interface. This mechanism will be named *transformation* in the following.

Definition 1: Let *newInstance* be an instance of type *New* that is delivered by a provided interface as a parameter type. Let *oldInstance* be an instance of a type *Old* that is the parameter type by the expected interface. Let *Old* and *New* be per se incompatible, i.e., no instance *newInstance* can be delivered when an instance of *Old* is expected. A transformation is then needed to get the state from the *newInstance*, create a new *oldInstance* and set the state of *oldInstance* appropriately. A transformation is valid when the *oldInstance* is actually what the expected interface expects.

It is important to raise awareness of this problem because automated adaptation in the object-oriented world requires this functionality. For example, one of the most interesting approaches for adaptation in the object-oriented world is based on the idea proposed by Hummel and Aktinson in 2004 [12] to use test cases to find candidates using a component search engine. In the test case, which is a simple unit test case, the client specifies the expected semantics, i.e., the expected interface with provided input and expected output parameters. The test case is used then during the search process by a component search engine in order to validate potential candidates against the semantics specified in the test case. This approach is known as test-driven reuse [4]. During the search, adaptation may become necessary, however, because candidates not necessarily fulfill the expected interface, i.e., signature mismatches may make it impossible to match the expected interface to a provided interface. The implemented adaptation engine tries to overcome signature mismatches using brute-force parameter permutation and so called relaxed signature matching on primitive data types (i.e., when an int is provided and a long is expected, the parameter types are regarded as a valid match [13]). Unfortunately, the approach does not consider the matching of object data types, which is a crucial requirement in object-oriented programming languages. The same applies for other approaches from the same field, proposed by Lemos et al. [14], named Sourcerer, and the search engine S6 proposed by Reiss [15]. The approach proposed by Kell [5] uses mapping rules in order to tackle the problem of object data type matching. This, however, is a cognitive overhead for a developer, especially when the details of the candidate to adapt are not really known. Seiffert and Hummel tackled the problem of matching well known data structures, such as linked lists, stacks and arrays on each other, by providing automated transformation mechanisms [16]. However, these are pre-defined mechanisms and not generally applicable on object types which are unknown beforehand.

As this brief overview reveals, there is currently no existing approach in the object-oriented world that tackles the problem of matching object-data types fully automatically. This will be further motivated in the next section.

III. MOTIVATION

Suppose a client wishes to use a BirthdayCalendar, where it is possible to set and retrieve birthdays for persons as shown by Figure 1. The classes *Person* and *Date* are assumed

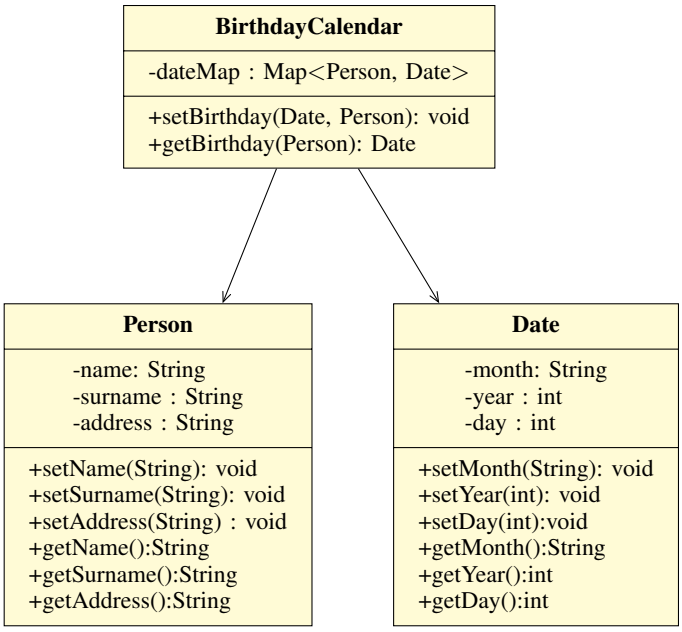

Figure 1. An expected BirthdayCalendar

be already fully implemented and available in the development project. The missing component is the *BirthdayCalendar* which supports the storing and retrieving of birthdays using the *setBirthday* and *getBirthday* methods. In other words, implementations of these methods are still missing. Although this is trivial it provides a simple example to explain the main problem.

Suppose *GeburtstagsKalender* is semantically equivalent to *BirthdayCalendar* (in fact, it is simply an implementation from another vendor for German speaking people). Figure 2 shows the corresponding class diagram. The type “Person” used by the *GeburtstagsKalender* has the same type name as the type Person expected by the *BirthdayCalendar*. The reason, in this case, is simply that both words have the same meaning in English and German. The types Person are semantically equivalent, i.e., a Person used by the *BirthdayCalendar* is described by its “name”, “surname” and its “address”. Correspondingly the type Person used by the *GeburtstagsKalender* is described by the German terms “name”, “nachname” and “adresse”. Neither of the types are connected by a type hierarchy. This is important, because according to the Liskov Substitution Prinicple [17] a subclass can be delivered in any situation where a parent class is expected. The construct *final* prohibits the creation of such a subclass relationship however. This also applies for Datum that is semantically equivalent to Date.

The *GeburtstagsKalender* would be useful because it provides the missing implementation, namely to store birthdays for persons and to retrieve them by the *setBirthday* and *getBirthday* methods. However, since Person and Datum do not match the expected types Person and Date this is not possible.

In order to let the client use *GeburtstagsKalender* to retrieve the expected functionality for storing and retrieving birthdates, the client relies on the well-known object adapter pattern proposed by the “Gang of Four” [18] as illustrated by Figure 3. The idea of this pattern is simple: an adapter implements the expected interface by the client and adapts

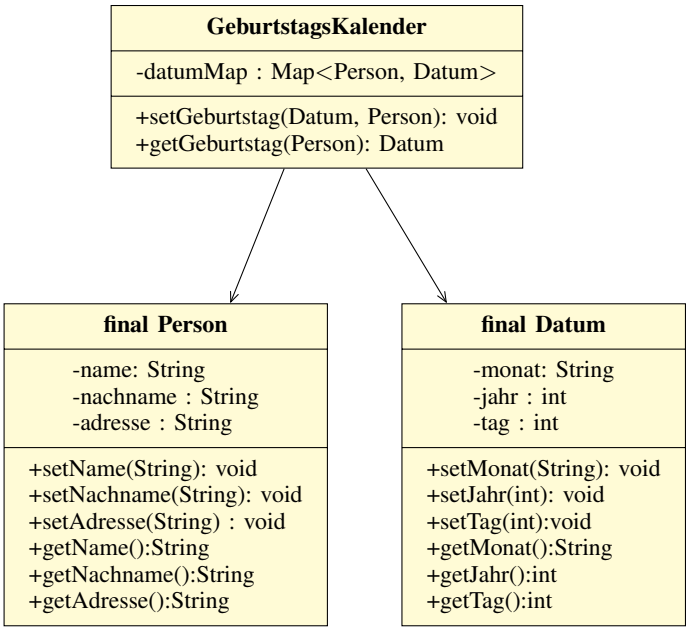

Figure 2. GeburtstagsKalender: a class semantically equivalent to BirthdayCalender

an object with the "wrong" interface. The client can use the adapter as if he would be working with the candidate to adapt directly. That is, all incoming messages are forwarded to the candidate and messages delivered by it are propagated back to the client. If the method names *setBirthday* and *setGeburtstag* would be the only mismatch, it would be sufficient to forward the parameters. However, in this case, mismatches on object data types exist. Therefore, more effort is required by the adapter to provide a suitable *transformation* mechanism.

In a former publication, we clarified the difference between adaptation and transformation [19]. Transformation complements adaptation and becomes relevant when adaptation cannot be applied. In the given example, adaptation would potentially solve the problem of the mismatching parameter types by creating adapters Datum2Date and Person2Person. These adapters are used by the adapter BirthdayCalendar2GeburtstagsKalendar within the *setBirthday* method. The arriving instance of Date would be set then to the Datum2Date adapter by a method *setAdaptee* as the candidate to adapt. This adapter can then be forwarded as a parameter to the *setGeburtstag* method. However, in order to create such an adapter, it must be able to subclass the parameter types Datum and Person of the *GeburtstagsKalender*. This is not possible in this case because of the *final* declaration in the given case for Person and Datum. In this former publication, we explained how transformation can be part of an adapter for building a facade, as a sophisticated example, however, we did not provide a more fine grained distinction for transformations and did not relate the problem on transforming state with type isomorphism. We also did not explain the main difference between objects from the object-oriented world and the abstract data types, used by the web service community in the web service description language, and did not mention the need for applying the transformation mechanism (and therefore also the adaptation mechanism) recursively, as well as the requirements for performing a transformation on an object type.

TABLE I. REQUIRED METHOD MATCHINGS FOR A PERSON TO SUPPORT OBJECT TRANSFORMATION

Output Method (Provided Instance)	Input Method (Expected Instance)
getName	setName
getSurname	setNachname
getAddress	setAddress

In the next section we provide an overview of the different challenges that need to be addressed in order to provide a transformation mechanism to complement adaptation.

IV. Providing a Transformation Mechanism

In order to provide a transformation mechanism, it is necessary to access the state provided by an object instance. Therefore, at least some of the following preconditions need to be fulfilled:

- the state can be accessed through the attributes provided by the object,
- the state can be accessed through the method(s) provided by the object.

According to the information hiding principle originally proposed by Parnas [11] objects should provide access to their internal attributes only through methods that are visible to the outside only. In order to investigate and access the methods on a given type in the Java language the Java Reflection API can be used. In the following, different case are considered, starting from low-level complexity to high-level complexity. The presented problems are enriched with intuitive solutions, but it is up to future research to investigate if these problems can be solved efficiently.

Case 1: In order to transform the state from the Person instance delivered by the *setBirthday* method to an instance of a Person expected by the *setGeburtstag* method, the method matchings shown in Table I have to be realized. The first column names the method of the provided instance whose output parameter needs to be set as an input parameter of the method of the expected instance in the second column.

To transform a Date instance to a Datum instance, the corresponding method matchings are shown in Table II. This challenge of matching output to input values has already been tackled by the web-service community [20], but has not been applied to objects in the context of object-oriented programming. There is a difference between an object and an abstract data type specified as a complex type in a web service description language (WSDL). A complex type does not provide any functionality through methods and WSDL prefers to use the xml schema definition, which provides a set of built-in data types, whereas an object from an object-oriented programming language can be of any type.

Case 2 The given example requires a flat transformation only, i.e., the parameter data types provided by the output methods are primitive data types only. The type of the *address* attribute of the type Person from the *BirthdayCalendar* can be changed to an an object type *Address* that holds data about the address, such as street and city, which can be set and retrieved by corresponding setter and getter methods again.

 ISBN: 978-1-61208-463-3

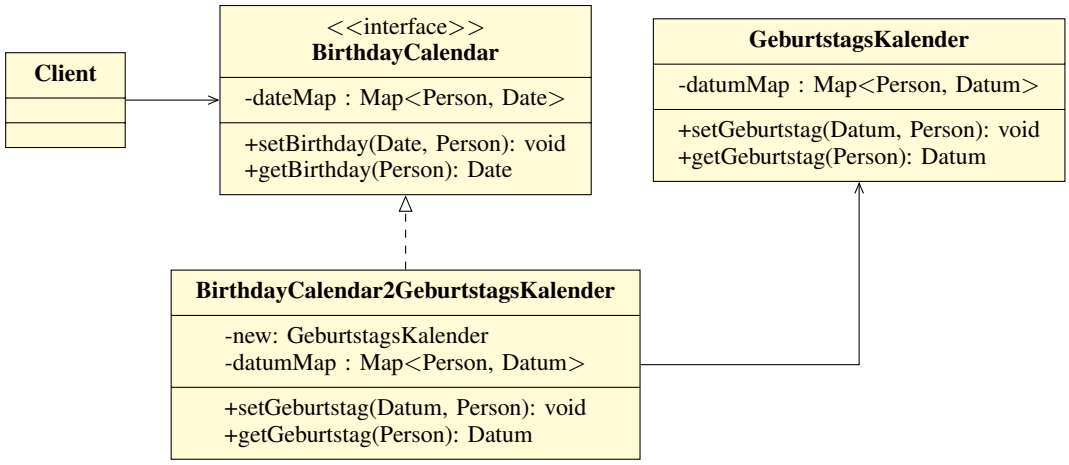

Figure 3. Adapter BirthdayCalendar2GeburtstagsKalender which adapts GeburtstagsKalendar

TABLE II. REQUIRED METHOD MATCHINGS FOR A DATE TO DATUM TO SUPPORT OBJECT TRANSFORMATION

Output Method (Provided Instance)	Input Method (Expected Instance)
getYear	setJahr
getMonth	setMonat
getDay	setTag

The type of *adresse* of the Person type expected by the *GeburtstagsKalender* can be changed to an object type *Adresse* accordingly, i.e., to an object type that is semantically equal but superficially incompatible. The problem of state transformation then needs to be applied recursively.

Case 3 Suppose the Person type of the *BirthdayCalendar* does not manage all its data by means of single fields, but by an internal array where each position specifies the content. For example, the position 0 might specify the name, position 1 the surname and position 2 the address, where for address the primitive data type, String, is assumed again. This array can be set by a *setInformation(String[] data)* method and retrieved by a *getInformation():String[]* method accordingly. In such a case, to provide a transformation mechanism, the array instance needs to be retrieved by the *getInformation* method and all possible permutations need to be applied as method invocation on the methods *setJahr*, *setMonat* and *setTag*, to set the content from the array to an instance of type Person appropriately as expected by the *GeburtstagsKalender*.

Case 4: Suppose the opposite situation to that stated in Case 3 occurs, i.e., suppose the Person type of *GeburtstagsKalender* expects an array of values describing the person, and suppose the Person type of the *BirthdayCalendar* uses single fields instead as in the initial example. Then, the adapter performing the transformation mechanism needs to create a new array instance which is then filled by invoking the getter methods on the delivered Person instance. This again requires that all possibilities are permutated in the worst case.

V. CONCLUSION

The aim of this position paper is to highlight the problem of transforming the state of object instances. This problem occurs when a provided interface delivers an object type, but the expected interface expects an object instance that is superficially incompatible with the type of the delivered instance and an adaptation mechanism itself can not be applied. In this paper we have used two implementations of birthday calendars from different developers that are are not connected by a type hierarchy, but provide the same semantics, to illustrate the problem. These types have the same functionality, i.e., the only differences are syntactic. Even semantic differences do not necessarily stop a transformation mechanism from being applied because transforming the state from a provided instance to an expected instance may be enough to let the expected interface use it. For example, a queue and a stack only share similar semantics. Equivalence is attained, however, when the queue instance created by transformation satisfies the expected interface. For this transformation, elements are retrieved from the stack and set in a queue instance which is forwarded. In the optimal case, this should be performed fully automatically by a transformation mechanism provided by the adapter. This problem of state transformation is, to the best of the author's knowledge, neglected in the literature so far. However, it needs to be tackled because object types provide an important signature mismatch problem to be solved. The availability of mechanisms to solve this problem fully automatically could, for example, significantly increase the recall of code search engines. Therefore, more tools and approaches need to be developed to automatically solve this problem.

REFERENCES

[1] M. Lenz, H. A. Schmid, and P. F. Wolf, "Software reuse through building blocks," *IEEE Software*, vol. 4, no. 4, pp. 34–42, July 1987.

[2] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, November 1987, DOI: 10.1145/32206.32212.

[3] 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*, ser. Lecture Notes in Computer Science, R. Reussner, J. A. Stafford, and C. Szyperski, Eds. Springer Berlin Heidelberg, 2006, vol. 3938, pp. 193–215, DOI: 10.1007/11786160_11.

[4] O. Hummel and W. Janjic, "Test-driven reuse: Key to improving precision of search engines for software reuse," in *Finding Source Code on the Web for Remix and Reuse*, S. Sim and R. Gallardo-Valencia, Eds.

 ISBN: 978-1-61208-463-3

Springer New York, 2013, pp. 227–250, DOI: 10.1007/978146146596-6_12.

[5] S. Kell, "Component adaptation and assembly using interface relations," in *Proceedings of the ACM international conference on Object oriented programming systems languages and application*, ser. OOPSLA'10, vol. 45, no. 10. New York, NY, USA: ACM, 2010, pp. 322–340, DOI: 10.1145/1869459.1869487.

[6] M. Rittri, "Using types as search keys in function libraries," *Journal of Functional Programming*, vol. 1, no. 1, pp. 71–89, 1991.

[7] C. Runciman and I. Toyn, "Retrieving re-usable software components by polymorphic type," *Journal of Functional Programming*, vol. 1, no. 2, pp. 191–211, 1991.

[8] R. D. Cosmo, *Isomorphisms of Types: from lambda calculus to information retrieval and language design*, R. V. Book, Ed. Birkhäuser, 1995.

[9] G. Booch, "Object-oriented development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 211–221, 1986.

[10] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002, ISBN: 978-0262162098.

[11] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," in *Communications of the ACM*, vol. 15, no. 12. ACM, 1972, pp. 1053–1058, DOI: 10.1145/361598.361623.

[12] O. Hummel and C. Atkinson, "Extreme harvesting: Test driven discovery and reuse of software components," in *Proceedings of the International Conference on Information Reuse and Integration*, ser. IEEE-IRI, 2004, pp. 66 – 72.

[13] ——, "Automated creation and assessment of component adapters with test cases," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds., vol. 6092. Springer Berlin Heidelberg, 2010, pp. 166–181, DOI: 10.1007/9783642132384_10.

[14] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, April 2011, DOI: 10.1016/j.infsof.2010.11.009.

[15] 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, DOI: 10.1109/ICSE.2009.5070525.

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

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

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

[19] D. Seiffert and O. Hummel, "How adaptation and transformation complement each other to potentially overcome signature mimismatch on object data types on the basis of test-cases," in *Adaptive 2015: The Seventh International Conference on Adaptive and Self-Adaptive Systems and Applications*. IARA, 2015, pp. 98–102, ISBN: 9781-612083919.

[20] 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.

 ISBN: 978-1-61208-463-3