Compatibility Layers for Interface Mediation at Run-Time

Patrick Rein Robert Hirschfeld Stefan Lehmann Jens Lincke

Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
{firstname}.{lastname}@hpi.uni-potsdam.de

Abstract

In adaptable systems, one module might require an interface from another module which the second module does not provide. For some cases, the particular provider module and its interface which will be available at run-time can not be anticipated during development time. In such situations with various provider interfaces, current mitigation strategies for interface mismatches struggle as they often rely on advanced knowledge about one particular providing module. Therefore, we propose the concept of compatibility layers which is based on modular interface mappings. These mappings are applied to adapt the provided interface at run-time. Each mapping contains a set of general requirements for the provided interface and a set of derived functions based on the required features. We have implemented the concept of compatibility layers in a Squeak/Smalltalk prototype based on context-oriented programming. Based on this prototype, we discuss the resulting trade-offs and illustrate exemplary interface mismatches in Squeak/Smalltalk which could be mediated by the prototype.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Modules and interfaces

Keywords interfaces, modules, interface compatibility, dynamic adaptation, context-oriented programming

1. Introduction

In open systems, situations might arise in which a client module requires an interface from a provider module, which the provider module does not support. If the provider module does not represent the semantically correct resource, then there is an error in the system design as the client has got a provider which it can not use. If however, the provider matches semantically and only the interface does not match the expected ones, a system failure would be unnecessary. This could for example occur when an object does represent the required resource but does not exhibit the expected set of methods.

This situation can arise if there are numerous different execution environments for the module under development. For example, developers of JavaScript libraries regularly face this issue. Their library might be used in various browsers and different browser versions which provide different JavaScript standard library interfaces.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892683

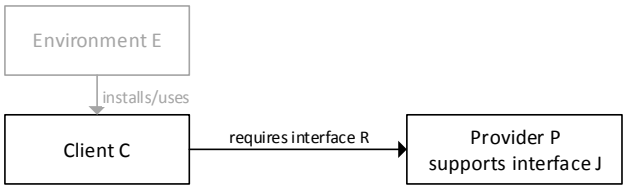

Figure 1. An illustration of the issue compatibility layers address. A client module C requires an interface R from a provider module P which in turn support interface J which is not compatible with R. Additionally, the module P and the interface J it provides is only known at run-time.

In such cases, where the number of potential provider modules is large and might change often, it becomes difficult for the developer of a client module to anticipate the available interfaces of provider modules at development time.

To improve the stability of a system in such situations, several mechanisms have been proposed which temporarily adapt the provided interface to match the required interface. Among these concepts are aspect-oriented programming (AOP) [13], context-oriented programming (COP) [9], or design patterns of object-oriented programming [7] like the adapter pattern. When using some implementations of these approaches, the developer uses knowledge about the provided interface to design the mapping. For example, the adapter pattern is designed to adapt one particular interface and is explicitly inserted between the client and the provider. In ordinary use-cases of context-oriented programming in object-oriented environments, the developer specifies the adapted class in advance.

This knowledge about the interfaces might not be available in settings in which the set of modules and their combinations are only known at run-time. In an object-oriented class-based system, this would be a situation in which we want to send a message to an object whose class or super-classes we do not know at development time. Thus, we propose compatibility layers, an approach to enable the run-time adaptation of provided interfaces which are unknown during development. It is based on the idea of interface mappings, which include a set of requirements on the provided interface and a set of derived functions, defined in terms of the required interface. These mappings are used in a context where compatibility is required to dynamically map the interface of incoming objects to the locally required interface. This approach is independent of any modularity concepts used to implement the interface extension.

1.1 Contributions

- The description of the compatibility layers concept to establish compatibility between interfaces of objects for scenarios in

algorithm can not deduct combinations of mappings to construct a specific interface.

Additionally, whenever a mapping is applied, all derived methods are installed. This might be desirable as this creates coherent interfaces in cases where the derived methods should be used in combination. At the same time, this might override existing implementations of methods which already had the correct behavior.

5.2 Applicability to Squeak/Smalltalk

After implementing compatibility layers, we used the resulting mappings to determine potential exemplary interface mismatches in the Squeak standard library of Squeak version 5.0. Then, we have implemented five exemplary mappings and evaluated which classes could be extended by them. The number of mappings is not sufficient for a quantitative analysis. However, we have found some incomplete interfaces which are worth noting:

- The classes String and Point implement the comparison method <= but do not provide the derived method between:and:.
- The class Path and its subclasses, which represent geometrical figures expressed through points, implement at:, size, and even select: but do not implement collect or reject:.
- The PipeJunction from the CommandShell package and the ImageReadWriter classes both implement the basic streaming protocol next and next: but do not provide streaming into a buffer with next:into:

Although they illustrate the nature of such interface incompatibilities, these examples could be solved at development time in Squeak/Smalltalk. As the complete Squeak standard library is available to developers, these mismatches can be mitigated by adding the methods to the base system. This might not be possible in other environments and standard libraries.

Additionally, the missing methods are a result of the reuse mechanism of Squeak which is single-inheritance. By using sideways composition mechanisms, for example MixIns [3], these issues might be mitigated.

5.3 Correctness of Method Calls

The interface mappings can cause issues when a provider does semantically not represent the required resource but does still fulfill the requirements to apply an interface mapping. In our prototypical implementation this might happen for example, when a Dictionary is used in place of a Collection. Smalltalk Dictionaries do understand at: and size which might be the requirements to implement the basic collection protocol methods like mapping or filtering. However, the interface only matches on the syntactical level as the Dictionary>>#at: method does actually accept keys of arbitrary type. This issue occurred in our prototype as the requirements are on the syntactical level and thus not strict enough.

Another issue which arose in the prototype are the return values of methods. Although, the derived functionality might transform the original return value before returning it, our mappings do not encode any requirements on the original return values. Stricter interface requirements might achieve this.

There might be a trade-off at this point between the strictness of the requirements to guarantee correctness and the flexibility of a mapping regarding the modules it can be applied on.

5.4 Impact on Adaptability

Generally, the concept of compatibility layers seems to allow more flexible combinations of modules. As the compatibility layer can also be activated in the surrounding environment E, it does not require any compatibility logic in the client C.

While the mappings introduce the risk of incorrect interface semantics, they also allow modules to be used in environments with previously unknown provided or required interfaces. This makes them suitable for scenarios in which the developer can not anticipate how and in which environments a client or provider module will be used.

6. Related Work

6.1 Call-By-Meaning

The call-by-meaning approach solves the interface mismatch issue by coupling client and provider only through a query for a suitable function [18]. The client code includes a query for a function which describes the requirements in natural language. The provider modules on the other hand describe their functions with extensive documentation, also written in natural language. To find a suitable match for the query in the client the approach uses a constraint solver. The facts the constraint solver works on have previously been extracted from the documentation strings using natural language processing techniques. Thus, the mapping between client and provider modules is solely implemented on the basis of general interface requirements. While solving syntactic interface mismatches, this also solves semantic mismatches. Although it is a very general solution, its implementation also requires a fundamentally different way of designing systems around facts instead of components.

6.2 Shims

Shims are a concept which is used to achieve compatibility between different versions of execution environments. Two noteable examples are the ECMAScript shims for the compatibility between JavaScript versions and the Microsoft Windows shims for compatibility between operating system APIs.

Both approaches do introduce a mapping from one interface to another. However, both implementations contain very static mappings. They replace or extend one particular function of a previously known class or object in the system. These shims can not react on an object which provides the required functionality but has a previously unknown type.

6.3 Data, Context and Interaction (DCI)

The data, context and interaction (DCI) paradigm uses a mechanism similar to mappings, but differs in its goals [17]. In the DCI paradigm there is data which represents the existing domain objects. These are selected with object queries and dynamically augmented with specific behavior (interactions) in a specific context. This means that an object gets to play a certain role in a context. For example, a crossing might play the role of a vertex and a street the role of an edge in a graph algorithm.

The matching of data entities and the augmentation of the entity does match the concept of a mapping. Further, the idea of a context matches the scoping mechanism in our approach.

However, both approaches differ in their targeted challenge. Our approach focuses on the practical issue of interface mismatches in object-oriented systems. In contrast, the DCI paradigm aims to change the way we perceive object-oriented systems in general, which might, as a side-effect, also render the interface mismatch issue obsolete.

Role-based Programming Role-based programming [8, 15] is similar to the DCI paradigm and our approach. It also includes the concepts of an extension of the behavior of an object in a particular context. Role-based programming can be used as one implementation strategy for the concept of compatibility layers. Whether a particular role-based programming implementation is suitable, depends on the flexibility of the requirements one can define for the selection of objects to augment with a role.

algorithm can not deduct combinations of mappings to construct a specific interface.

Additionally, whenever a mapping is applied, all derived methods are installed. This might be desirable as this creates coherent interfaces in cases where the derived methods should be used in combination. At the same time, this might override existing implementations of methods which already had the correct behavior.

5.2 Applicability to Squeak/Smalltalk

After implementing compatibility layers, we used the resulting mappings to determine potential exemplary interface mismatches in the Squeak standard library of Squeak version 5.0. Then, we have implemented five exemplary mappings and evaluated which classes could be extended by them. The number of mappings is not sufficient for a quantitative analysis. However, we have found some incomplete interfaces which are worth noting:

The classes String and Point implement the comparison method `<=` but do not provide the derived method `between:and:`.

The class Path and its subclasses, which represent geometrical figures expressed through points, implement `at:`, `size`, and even `select:` but do not implement `collect` or `reject:`.

The PipeJunction from the CommandShell package and the ImageReadWriter classes both implement the basic streaming protocol `next` and `next:` but do not provide streaming into a buffer with `next:into:`

Although they illustrate the nature of such interface incompatibilities, these examples could be solved at development time in Squeak/Smalltalk. As the complete Squeak standard library is available to developers, these mismatches can be mitigated by adding the methods to the base system. This might not be possible in other environments and standard libraries.

Additionally, the missing methods are a result of the reuse mechanism of Squeak which is single-inheritance. By using sideways composition mechanisms, for example MixIns [3], these issues might be mitigated.

5.3 Correctness of Method Calls

The interface mappings can cause issues when a provider does semantically not represent the required resource but does still fulfill the requirements to apply an interface mapping. In our prototypical implementation this might happen for example, when a Dictionary is used in place of a Collection. Smalltalk Dictionaries do understand `at:` and `size` which might be the requirements to implement the basic collection protocol methods like mapping or filtering. However, the interface only matches on the syntactical level as the `Dictionary>>#at:` method does actually accept keys of arbitrary type. This issue occurred in our prototype as the requirements are on the syntactical level and thus not strict enough.

Another issue which arose in the prototype are the return values of methods. Although, the derived functionality might transform the original return value before returning it, our mappings do not encode any requirements on the original return values. Stricter interface requirements might achieve this.

There might be a trade-off at this point between the strictness of the requirements to guarantee correctness and the flexibility of a mapping regarding the modules it can be applied on.

5.4 Impact on Adaptability

Generally, the concept of compatibility layers seems to allow more flexible combinations of modules. As the compatibility layer can also be activated in the surrounding environment E, it does not require any compatibility logic in the client C.

While the mappings introduce the risk of incorrect interface semantics, they also allow modules to be used in environments with previously unknown provided or required interfaces. This makes them suitable for scenarios in which the developer can not anticipate how and in which environments a client or provider module will be used.

6. Related Work

6.1 Call-By-Meaning

The call-by-meaning approach solves the interface mismatch issue by coupling client and provider only through a query for a suitable function [18]. The client code includes a query for a function which describes the requirements in natural language. The provider modules on the other hand describe their functions with extensive documentation, also written in natural language. To find a suitable match for the query in the client the approach uses a constraint solver. The facts the constraint solver works on have previously been extracted from the documentation strings using natural language processing techniques. Thus, the mapping between client and provider modules is solely implemented on the basis of general interface requirements. While solving syntactic interface mismatches, this also solves semantic mismatches. Although it is a very general solution, its implementation also requires a fundamentally different way of designing systems around facts instead of components.

6.2 Shims

Shims are a concept which is used to achieve compatibility between different versions of execution environments. Two noteable examples are the ECMAScript shims for the compatibility between JavaScript versions and the Microsoft Windows shims for compatibility between operating system APIs.

Both approaches do introduce a mapping from one interface to another. However, both implementations contain very static mappings. They replace or extend one particular function of a previously known class or object in the system. These shims can not react on an object which provides the required functionality but has a previously unknown type.

6.3 Data, Context and Interaction (DCI)

The data, context and interaction (DCI) paradigm uses a mechanism similar to mappings, but differs in its goals [17]. In the DCI paradigm there is data which represents the existing domain objects. These are selected with object queries and dynamically augmented with specific behavior (interactions) in a specific context. This means that an object gets to play a certain role in a context. For example, a crossing might play the role of a vertex and a street the role of an edge in a graph algorithm.

The matching of data entities and the augmentation of the entity does match the concept of a mapping. Further, the idea of a context matches the scoping mechanism in our approach.

However, both approaches differ in their targeted challenge. Our approach focuses on the practical issue of interface mismatches in object-oriented systems. In contrast, the DCI paradigm aims to change the way we perceive object-oriented systems in general, which might, as a side-effect, also render the interface mismatch issue obsolete.

Role-based Programming Role-based programming [8, 15] is similar to the DCI paradigm and our approach. It also includes the concepts of an extension of the behavior of an object in a particular context. Role-based programming can be used as one implementation strategy for the concept of compatibility layers. Whether a particular role-based programming implementation is suitable, depends on the flexibility of the requirements one can define for the selection of objects to augment with a role.

algorithm can not deduct combinations of mappings to construct a specific interface.

Additionally, whenever a mapping is applied, all derived methods are installed. This might be desirable as this creates coherent interfaces in cases where the derived methods should be used in combination. At the same time, this might override existing implementations of methods which already had the correct behavior.

5.2 Applicability to Squeak/Smalltalk

After implementing compatibility layers, we used the resulting mappings to determine potential exemplary interface mismatches in the Squeak standard library of Squeak version 5.0. Then, we have implemented five exemplary mappings and evaluated which classes could be extended by them. The number of mappings is not sufficient for a quantitative analysis. However, we have found some incomplete interfaces which are worth noting:

The classes String and Point implement the comparison method <= but do not provide the derived method between:and:.

The class Path and its subclasses, which represent geometrical figures expressed through points, implement at:, size, and even select: but do not implement collect or reject:.

The PipeJunction from the CommandShell package and the ImageReadWriter classes both implement the basic streaming protocol next and next: but do not provide streaming into a buffer with next:into:

Although they illustrate the nature of such interface incompatibilities, these examples could be solved at development time in Squeak/Smalltalk. As the complete Squeak standard library is available to developers, these mismatches can be mitigated by adding the methods to the base system. This might not be possible in other environments and standard libraries.

Additionally, the missing methods are a result of the reuse mechanism of Squeak which is single-inheritance. By using sideways composition mechanisms, for example MixIns [3], these issues might be mitigated.

5.3 Correctness of Method Calls

The interface mappings can cause issues when a provider does semantically not represent the required resource but does still fulfill the requirements to apply an interface mapping. In our prototypical implementation this might happen for example, when a Dictionary is used in place of a Collection. Smalltalk Dictionaries do understand at: and size which might be the requirements to implement the basic collection protocol methods like mapping or filtering. However, the interface only matches on the syntactical level as the Dictionary>>#at: method does actually accept keys of arbitrary type. This issue occurred in our prototype as the requirements are on the syntactical level and thus not strict enough.

Another issue which arose in the prototype are the return values of methods. Although, the derived functionality might transform the original return value before returning it, our mappings do not encode any requirements on the original return values. Stricter interface requirements might achieve this.

There might be a trade-off at this point between the strictness of the requirements to guarantee correctness and the flexibility of a mapping regarding the modules it can be applied on.

5.4 Impact on Adaptability

Generally, the concept of compatibility layers seems to allow more flexible combinations of modules. As the compatibility layer can also be activated in the surrounding environment E, it does not require any compatibility logic in the client C.

While the mappings introduce the risk of incorrect interface semantics, they also allow modules to be used in environments with previously unknown provided or required interfaces. This makes them suitable for scenarios in which the developer can not anticipate how and in which environments a client or provider module will be used.

6. Related Work

6.1 Call-By-Meaning

The call-by-meaning approach solves the interface mismatch issue by coupling client and provider only through a query for a suitable function [18]. The client code includes a query for a function which describes the requirements in natural language. The provider modules on the other hand describe their functions with extensive documentation, also written in natural language. To find a suitable match for the query in the client the approach uses a constraint solver. The facts the constraint solver works on have previously been extracted from the documentation strings using natural language processing techniques. Thus, the mapping between client and provider modules is solely implemented on the basis of general interface requirements. While solving syntactic interface mismatches, this also solves semantic mismatches. Although it is a very general solution, its implementation also requires a fundamentally different way of designing systems around facts instead of components.

6.2 Shims

Shims are a concept which is used to achieve compatibility between different versions of execution environments. Two noteable examples are the ECMAScript shims for the compatibility between JavaScript versions and the Microsoft Windows shims for compatibility between operating system APIs.

Both approaches do introduce a mapping from one interface to another. However, both implementations contain very static mappings. They replace or extend one particular function of a previously known class or object in the system. These shims can not react on an object which provides the required functionality but has a previously unknown type.

6.3 Data, Context and Interaction (DCI)

The data, context and interaction (DCI) paradigm uses a mechanism similar to mappings, but differs in its goals [17]. In the DCI paradigm there is data which represents the existing domain objects. These are selected with object queries and dynamically augmented with specific behavior (interactions) in a specific context. This means that an object gets to play a certain role in a context. For example, a crossing might play the role of a vertex and a street the role of an edge in a graph algorithm.

The matching of data entities and the augmentation of the entity does match the concept of a mapping. Further, the idea of a context matches the scoping mechanism in our approach.

However, both approaches differ in their targeted challenge. Our approach focuses on the practical issue of interface mismatches in object-oriented systems. In contrast, the DCI paradigm aims to change the way we perceive object-oriented systems in general, which might, as a side-effect, also render the interface mismatch issue obsolete.

Role-based Programming Role-based programming [8, 15] is similar to the DCI paradigm and our approach. It also includes the concepts of an extension of the behavior of an object in a particular context. Role-based programming can be used as one implementation strategy for the concept of compatibility layers. Whether a particular role-based programming implementation is suitable, depends on the flexibility of the requirements one can define for the selection of objects to augment with a role.

algorithm can not deduct combinations of mappings to construct a specific interface.

Additionally, whenever a mapping is applied, all derived methods are installed. This might be desirable as this creates coherent interfaces in cases where the derived methods should be used in combination. At the same time, this might override existing implementations of methods which already had the correct behavior.

5.2 Applicability to Squeak/Smalltalk

After implementing compatibility layers, we used the resulting mappings to determine potential exemplary interface mismatches in the Squeak standard library of Squeak version 5.0. Then, we have implemented five exemplary mappings and evaluated which classes could be extended by them. The number of mappings is not sufficient for a quantitative analysis. However, we have found some incomplete interfaces which are worth noting:

- The classes String and Point implement the comparison method <= but do not provide the derived method between:and:.
- The class Path and its subclasses, which represent geometrical figures expressed through points, implement at:, size, and even select: but do not implement collect or reject:.
- The PipeJunction from the CommandShell package and the ImageReadWriter classes both implement the basic streaming protocol next and next: but do not provide streaming into a buffer with next:into:

Although they illustrate the nature of such interface incompatibilities, these examples could be solved at development time in Squeak/Smalltalk. As the complete Squeak standard library is available to developers, these mismatches can be mitigated by adding the methods to the base system. This might not be possible in other environments and standard libraries.

Additionally, the missing methods are a result of the reuse mechanism of Squeak which is single-inheritance. By using sideways composition mechanisms, for example MixIns [3], these issues might be mitigated.

5.3 Correctness of Method Calls

The interface mappings can cause issues when a provider does semantically not represent the required resource but does still fulfill the requirements to apply an interface mapping. In our prototypical implementation this might happen for example, when a Dictionary is used in place of a Collection. Smalltalk Dictionaries do understand at: and size which might be the requirements to implement the basic collection protocol methods like mapping or filtering. However, the interface only matches on the syntactical level as the Dictionary>>#at: method does actually accept keys of arbitrary type. This issue occurred in our prototype as the requirements are on the syntactical level and thus not strict enough.

Another issue which arose in the prototype are the return values of methods. Although, the derived functionality might transform the original return value before returning it, our mappings do not encode any requirements on the original return values. Stricter interface requirements might achieve this.

There might be a trade-off at this point between the strictness of the requirements to guarantee correctness and the flexibility of a mapping regarding the modules it can be applied on.

5.4 Impact on Adaptability

Generally, the concept of compatibility layers seems to allow more flexible combinations of modules. As the compatibility layer can also be activated in the surrounding environment E, it does not require any compatibility logic in the client C.

While the mappings introduce the risk of incorrect interface semantics, they also allow modules to be used in environments with previously unknown provided or required interfaces. This makes them suitable for scenarios in which the developer can not anticipate how and in which environments a client or provider module will be used.

6. Related Work

6.1 Call-By-Meaning

The call-by-meaning approach solves the interface mismatch issue by coupling client and provider only through a query for a suitable function [18]. The client code includes a query for a function which describes the requirements in natural language. The provider modules on the other hand describe their functions with extensive documentation, also written in natural language. To find a suitable match for the query in the client the approach uses a constraint solver. The facts the constraint solver works on have previously been extracted from the documentation strings using natural language processing techniques. Thus, the mapping between client and provider modules is solely implemented on the basis of general interface requirements. While solving syntactic interface mismatches, this also solves semantic mismatches. Although it is a very general solution, its implementation also requires a fundamentally different way of designing systems around facts instead of components.

6.2 Shims

Shims are a concept which is used to achieve compatibility between different versions of execution environments. Two noteable examples are the ECMAScript shims for the compatibility between JavaScript versions and the Microsoft Windows shims for compatibility between operating system APIs.

Both approaches do introduce a mapping from one interface to another. However, both implementations contain very static mappings. They replace or extend one particular function of a previously known class or object in the system. These shims can not react on an object which provides the required functionality but has a previously unknown type.

6.3 Data, Context and Interaction (DCI)

The data, context and interaction (DCI) paradigm uses a mechanism similar to mappings, but differs in its goals [17]. In the DCI paradigm there is data which represents the existing domain objects. These are selected with object queries and dynamically augmented with specific behavior (interactions) in a specific context. This means that an object gets to play a certain role in a context. For example, a crossing might play the role of a vertex and a street the role of an edge in a graph algorithm.

The matching of data entities and the augmentation of the entity does match the concept of a mapping. Further, the idea of a context matches the scoping mechanism in our approach.

However, both approaches differ in their targeted challenge. Our approach focuses on the practical issue of interface mismatches in object-oriented systems. In contrast, the DCI paradigm aims to change the way we perceive object-oriented systems in general, which might, as a side-effect, also render the interface mismatch issue obsolete.

Role-based Programming Role-based programming [8, 15] is similar to the DCI paradigm and our approach. It also includes the concepts of an extension of the behavior of an object in a particular context. Role-based programming can be used as one implementation strategy for the concept of compatibility layers. Whether a particular role-based programming implementation is suitable, depends on the flexibility of the requirements one can define for the selection of objects to augment with a role.

7. Future Work

7.1 Evaluation on Productive Systems

To finally determine whether this concept can solve interface mismatches appearing in practice, a more extensive evaluation is required. One study might be to test to which extent compatibility layers allow Smalltalk developers to use Smalltalk libraries originating from another Smalltalk environment. To determine how well unanticipated interfaces can be adapted, we would first implement a set of mappings based on the differences between the environment A providing the libraries and the environment B in which they will be used. In this scenarios the libraries play the role of the client C which requires interfaces from the standard library which is the provider P. Only after implementing the mappings, we would select a number of libraries which were implemented in the environment A. We would then try to execute the corresponding test suites with and without compatibility layers and we could measure the improvement in succeeded test cases.

7.2 Mapping Varieties

From the perspective of an implementation, it might be interesting to explore further possibilities for expressing mappings.

First of all, it might be beneficial to investigate how a more powerful requirements definition mechanism, for example a pointcut language, influences the capabilities of the mappings. Further, the algorithm for determining suitable combinations of mappings to mediate to interfaces could be extended. Especially, an algorithm to construct combinations of mappings to get the right interface would be interesting.

Additionally, there is the open challenge of a consistent interface mediation during an interaction between two objects. If the client starts an interaction with a provider through a mediated interface, subsequent calls to the provider might need to be mediated through the same interfaces. For example, if one derived method from a mapping allocates a resource the corresponding derived method for releasing the resource needs to be applied too and stay active for the rest of the interaction.

Applicability to Structural Incompatibilities Another open challenge is the issue of structural interface mismatches. The interface might not only consist of methods or functions but also of specific types or groups of objects. Conceptually, the model of compatibility layers allows for such situations. However, it is yet unclear how to describe a mapping for such incompatibilities.

7.3 Interface Mappings as Modules

Interesting effects might arise from a system which uses the concept of interface mappings as its primary module system. The trade-off between correctness and adaptability discussed in section 5 would become central in such a system. Other interesting aspect would be the impact of a scoping mechanism and differences to role-based programming systems.

7.4 Tool Support

Finally, the mappings might also create new opportunities for tool support. The mappings could be used to suggest new interfaces to a developer. For example, a tool might constantly check whether any mapping can already be applied to a class under development. If that is the case, the tool can suggest the mapping to the developer, who might choose to integrate the derived methods permanently into the class.

8. Conclusion

We have illustrated the issues involved in mitigating interface mismatches in situations in which the actual provider available at runtime can not be anticipated. These issues included the entanglement of client and compatibility logic and the scoping of the adaptation of provider interfaces. As a consequence, we proposed the concept of compatibility layers, which dynamically apply mappings from one interface to another in a limited scope. We illustrated one way to implement this concept with an implementation in Squeak/Smalltalk. Based on the prototype and the conceptual description, we discussed the resulting trade-off between correctness and adaptability. However, it remains to be shown, whether this concept can solve interface mismatches on a larger scale. In total, compatibility layers might be a concept to improve the modularization of interactions of components in systems in which components are exchanged dynamically.

References

[1] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In *Proceedings of Joint Modular Languages Conference JMLC03*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin Heidelberg, 2003.

[2] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by example*. Square Bracket Associates, 2010.

[3] G. Bracha and W. Cook. Mixin-based inheritance. *ACM Sigplan Notices*, 25(10):303–311, 1990.

[4] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, 1991.

[5] S. Conder and L. Darcey. *Android Wireless Application Development*. Addison Wesley, Upper Saddle River, NJ, 2nd revised edition. edition edition, Dec. 2010. ISBN 978-0-321-74301-5.

[6] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *Proceedings of ECOOP 2006, Nantes, France, July 3-7, 2006, Proceedings*, pages 328–352, 2006.

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

[8] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264. Springer, 2003.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[10] K. Honda. Types for dyadic interaction. In *CONCUR 1993*, pages 509–523. Springer, 1993.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM SIGPLAN Notices*, volume 32, pages 318–326. ACM, 1997.

[12] T. Kaehler. Squeak wiki: Method finder. `http://wiki.squeak.org/squeak/1916` (accessed 12 January 2016).

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP 1997*, pages 220–242, 1997.

[14] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76(12):1194 – 1209, 2011.

[15] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM, 2003.

[16] S. Powers. *JavaScript Cookbook*. O'Reilly Media, Inc., 2015.

[17] T. Reenskaug. The common sense of object oriented programming. *Department of Informatics, University of Oslo, Oslo, Norway*, 2009.

[18] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. D. Millstein. Call by meaning. In *Proceedings of Onward! 2014, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 11–28, 2014.