# Deducing Classes

## Integrating the Domain Models of Object-Oriented Applications

Patrick Rein

Hasso Plattner Institute, University of Potsdam, Germany

patrick.rein@hpi.uni-potsdam.de

## Abstract

The interoperability of applications depends on a successful mapping between their domain models. Nowadays, common file formats serve as a mediator between the different domain models but cause friction losses during the conversion of data. These loses could be mitigated whenever the models are already working on the same concepts but are only using different representations for them. We propose the concept of deducing classes which interpret existing object structures and detect instances of themselves in this existing data. Further, we introduce a planning algorithm which combines deducing classes to allow unanticipated interactions between applications. We discuss some of the implications of this approach and illustrate upcoming research challenges.

***Categories and Subject Descriptors*** D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming

***Keywords*** object domain models, integration, abstractions

## 1. Introduction

When creating an application using object-oriented programming, developers define types of objects and their interactions which represent relevant concepts of the application domain. Thus, developers create an executable *domain model* [3]. When developing new applications, developers often start their domain model from scratch to fit it closely to their use case. Thereby, developers define new terms and object representations for domain concepts. For example, an address book application might define a class for contact details. This way of postulating an individual world model works well for developing single applications. If however, the application needs to integrate with other applications through sharing data, then this isolated definition of objects becomes an issue. One application might represent a concept very differently to the way another application represents it. As a consequence, if one application wants to enable another application to use its data, the objects have to be converted to fit the world model of the other application. As developers can not anticipate all the other world models, they convert the data to a common denominator often determined by common file formats such as the `vcard` format for contact detail information. Similarly, the receiving application requires a mapping from the common world model to its model.

These conversions introduce several issues. First of all, the conversion to a common world model can cause a loss in information such as relations which can not be expressed or the identity of single objects. Second, if an application does not support an import or export function for a format required by another application, users are stuck within the application and can not work with the data in other applications. For example, the address book application might export its data as a set of `vcard` files in one folder and a digital newsletter service might expect a comma-separated file including a name and an email address column. Finally, these issues hinder end-users to get the most out of their applications as they can not use them with all of their data even if it was possible on a semantic level. These "friction losses" due to the conversion are unnecessary, as the domain models of the two applications already describe concepts which are semantically at least similar, if not equal.

To improve the integration and interoperability of applications, we propose a different perspective on creating abstractions in object-oriented applications. Instead of creating abstractions by using new terms which define their own representations through objects, we propose *deducing classes* which instead interpret existing object structures and detect instances of that class. By automatically combining these abstractions the system might be able to mitigate unanticipated mismatches. Through this, we hope to be able to interchange semantically equivalent information almost seamlessly between applications regardless of their representation.

## 2. Integration through Deducing Classes

We propose the concept of *deducing classes* which are defined through a *query* and a set of *derived methods*. The query selects single or multiple objects which are then cap-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

*SPLASH Companion'16*, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998547

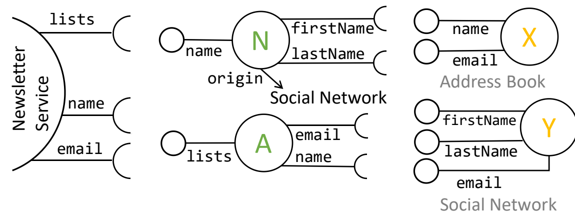


**Figure 1.** The deducing classes (N, A) bridge between the interface required by the newsletter service and the interface provided by the objects from different applications (X,Y).

tured in a new object. This new object has a set of derived methods which are defined in terms of the objects matched by the query. For example, the developers of the newsletter service could define an `AddressableEntity` which has a simple query which requires two fields storing a name and an email address (denoted A in Figure 1). In the context of the newsletter service it might provide a derived method to list all newsletters this address is registered to. This class would now match any contact objects from the address book application and at the same time would directly provide the methods which are useful in the context of the newsletter application. Using solely this approach would not solve the integration issue, as mappings would only be described in terms of one other kind of representation of the same information. If the contact information from a social network represents the "name" in two fields then the object can again not be used (see Y in Figure 1). Thus, we propose a mechanism based on planning algorithms which combines several of these deducing classes to find a way to make the objects compatible with the required interface [6]. This requires additional mappings to be available (e.g. mapping N in Figure 1). These mappings might be provided by developers in a way similar to the way plugins are provided today. More interesting, however, is that end-users might be able to provide these mappings. An assistant tool could ask how they would create the name of the social network contact information on the basis of the fields of the object being accessed.

We have implemented a first prototype based on graph matching and common classes within a Smalltalk execution environment. Thereby, we can experiment with design decisions without worrying about binary conversions between environments. At this stage developers can create a deduced class by sub-classing the `DeducedObject` class and specifying a graph query whose variables form the instance variables of the class. The class also includes method definitions based on these instance variables. The classes can be instantiated by asking for instances of the class in a given set of objects. These instances allow read-only access to the data of the original objects. Further, the execution environment was extended to use the deduced classes for mitigating message dispatch errors. Whenever an object does not understand a message, the system tries to combine the deduced classes in the system to create the required interface.

## 3. Discussion

Our work approaches issues which are also known from ontological merging in the field of data integration: *explication mismatch* (different object structures for the same concept) and *terminological mismatch* (different identifiers for the same concept) [2, 8]. Similar to the normalization assumption in data integration approaches, we currently assume that data is completely represented through object structures. This might be mitigated by future work on queries which do match more than graph structures (e.g. incorporating regular expressions). The explication mismatch is solved through the general approach of querying existing data, transforming it, and presenting it as new data, similar to the mechanism of database *views* [5]. However, the additional component of automatic combinations of several deductions makes the system more flexible with regard to its inputs. At the same time it introduces ambiguity which might lead to unintended behavior. To mitigate this, a mechanism would be required which allows developers to trade-off correctness and flexibility for specific sections of their code. With regard to the dynamic resolution of mismatches our work is related to the call-by-meaning work [7] which solves the mismatches by using constraints on the functions instead of names. Our approach still suffers from the issue of ambiguous identifiers. For example, "name" is a homonym for many different pieces of information. This could be resolved by extending our approach with the idea of using *URIs* as identifiers in code [1]. In general, as others have argued before, the definition of abstractions in terms of existing objects might be close to the way humans generally create abstractions [4].

## References


[1] J. M. Alcaraz Calero, J. B. Bernabé, J. M. Marin Perez, D. S. Ruíz, F. J. Garcia Clemente, G. M. Pérez, et al. Towards an Architecture to bind the Java and OWL languages. *JRPIT*, 44 (1):17, 2012.

[2] J. Euzenat, P. Shvaiko, et al. *Ontology Matching*, volume 2. Springer, 2007.

[3] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[4] T. Felgentreff, J. Lincke, R. Hirschfeld, and L. Thamsen. Lively Groups: Shared Behavior in a World of Objects Without Classes or Prototypes. In *Proceedings of the FPW 2015*, pages 15–22, October 2015.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[6] S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Pearson Education, Inc., 2003.

[7] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. D. Millstein. Call by Meaning. In *Proceedings of Onward! 2014*, pages 11–28, October 2014.

[8] P. R. Visser, D. M. Jones, T. J. Bench-Capon, and M. J. Shave. Assessing Heterogeneity by Classifying Ontology Mismatches. In *Proceedings of the FOIS*, volume 98, 1998.