# Exploring Role-Based Adaptation

Sebastian Götz and Ilie Şavga

Department of Computer Science, Dresden University of Technology, Germany,
{sebastian.goetz|is13}@mail.inf.tu-dresden.de

**Abstract.** The adapter design pattern [1], commonly used for integration and evolution in component-based systems, is originally described by *roles*. In class-based systems, the conventional realization of the pattern maps these roles to classes. The recent appearance of mature languages supporting roles as first order programming constructs poses the question whether realizing this pattern directly in roles offers benefits comparing to class-based realization. This paper explores the feasibility of role-based adaptation and discusses its benefits and challenges.

## 1 Introduction

When assembling independently developed components, it is often the case that their public interfaces do not fit to each other. If components cannot be adjusted directly (e.g., when assembling third-party components), an adapter needs to be placed between them to bridge interface incompatibilities. Gamma et al. [1, p. 139] describes the adapter design pattern by 4 collaborating roles (*Client*, *Target*, *Adapter* and *Adaptee*) and shows a possible pattern implementation as a mapping of these roles to classes.

For our running example, assume a university management system (UMS), in which the concept of student is modeled by interface `Student` and implemented by class `StudentImpl`. Among other interface methods, the class implements the `getGrades` method that retrieves subjects and grades of the student from a file used for serialization. This method is used also in the implementation of `printGrades` that prints out subjects and grades of a student.

Later, due to new system requirements, it is decided to buy a sophisticated reporting component that replaces the simple functionality previously realized directly by `StudentImpl`. Moreover, UMS is integrated with a persistence component that is now responsible for saving and retrieving data. To retrieve student grades, `StudentImpl` must now call the persistence component to get data. To print this information, `StudentImpl` must wrap it before sending to the reporting component, because the signature of printing method in `Student` (expected by existing clients) differs from the one of the reporting component (`Report.printReport` expecting report component's specific `DataRow` as its parameter). So, `StudentImpl` must translate between the two interfaces and becomes effectively a class-based adapter (Figure 1).

Figure 2 shows internals of the `printGrade` method of `StudentImpl` that performs the actual translation. Using student identity (for simplicity, "this"), the method constructs the corresponding SQL query, retrieves data from the persistence component using the `getGrades` method and fills them into the type required by the reporting component. In addition, now the `getGrades` method (code not shown) itself is an

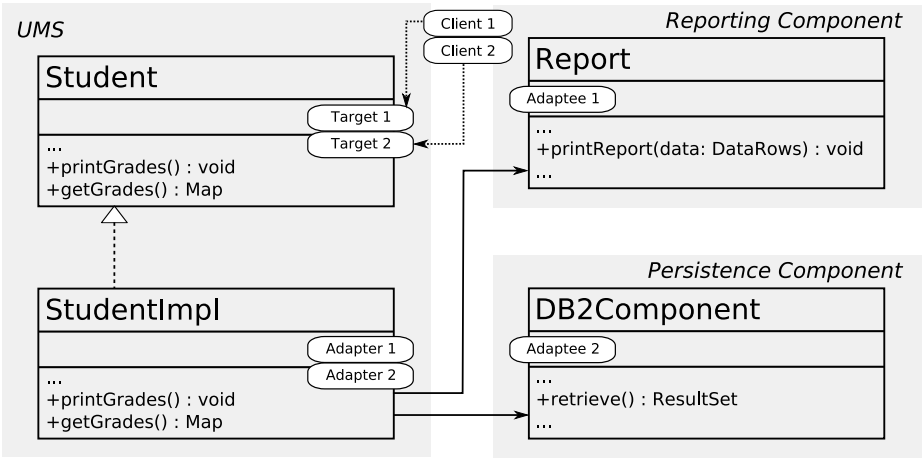


**Fig. 1.** Class-based adapter. Classes are annotated with roles these classes implement.

adaptation method calling `retrieve` of the newly introduced persistence component and converting its `ResultSet` to the `Map` of the `Student` interface being adapted.

The main drawback of this class-based adapter realization is that the code responsible for different tasks is highly intertwined. For instance, in lines 11 and 12 the code realizing logic for the data retrieval and for reporting concerns is joined. When realizing this adapter, developers need to consider in fact the static types and semantics of all three domains involved (i.e., of the report and persistence components and of the UMS itself). In real life scenarios with possibly many interrelated components being integrated, such inability to separately realize each concern increases the time and

```
1  Report report;
2  DBComponent db;
3  public void printGrades() {
4    //construct an SQL query for this student
5     String query = createSQLQueryByTime(this);
6     //retreive student subject-mark pairs
7     ResultSet srs = this.getGrades(query);
8     //fill in and send the report data
9     DataRow reportData = new DataRow();
10    while (srs.next()) {
11     reportData.add(srs.getString(''subject''));
12     reportData.add(srs.getString(''mark''));
13   }
14   report.printReport(reportData);
15 }
```

**Fig. 2.** Implementation of `StudentImpl.printGrades`

error-proness of adaptation. Even more important, an adapter is itself a software artifact inevitably requiring maintenance. In case the adapter needs to be modified (for example, to improve its performance), developers need to understand its often extremely complex implementation.

The situation aggravates furthermore when the public interfaces of components, on which the adapter depends, evolve as well. In our running example, an upgraded version of the report component may change the signature of `Report.printReport`. For example, in an older component version, its void method was throwing an exception in case of a printing failure and in the new version the method returns a new type `DocumentPrinting` containing details of method's execution. To accommodate the adapter to these changes, its whole code needs to be thoroughly investigated and understood. Often this needs to be done by developers others then the adapter's initial developers. Because the adaptation decisions are made dependent on each other in the code, a bug made when adjusting one component may propage to other adapter's parts. For instance, if `getGrades` of the persistence component evolves and a bug is made when adjusting to its changes, this bug will also be reflected in the behavior of the adapter's `printGrades`.

All in all, these maintenance problems stem from the fact that the adaptation concern mentally modeled by four roles of [1] is lost in transition to the class-based adapter implementation. Presumably, preserving these roles explicit in the implementation brings benefits comparing to class-based adaptation. Using a language supporting roles as first-class citizens, we investigate the feasibility, benefits and drawbacks of role-based adaptation.

## 2 Role-based Adaptation

To implement the role-based adapter of our running example, we use a relatively new yet rich language ObjectTeams/Java—a stable well-tested Java extension supporting roles and collaborations [2]. However, since the language consists of several specific terms that need lengthy explanation, in this paper we refrain from its specific terminology. Instead, taking into consideration the run-time responsibilities, we dissect the concept of the Adapter role into *In*- and *Out*-(sub)roles. Similarly to conventional aspects, an In-role is responsible for handling the incoming data into the adapter and an Out-role is in charge of passing data flow further to the adaptee.

Figure 3 depicts how role-based adaptation can be realized for our running example. Each role is played by (instances of) and mapped to a single class. The key difference to the class-based adapter is that Adapter is realized directly as a role. As a consequence, it is now possible to define a separate adapter role for each commercial component to be integrated. Additionally there is another separate role for each target class. Note, that In- and Out-roles for each adaptation role in concern are realized separately and are encapsulated in the corresponding role realizations, preserving thus adaptation decisions. Regardless of how they are actually mapped to classes and boil down to the execution code, these explicit roles can be maintained separately and do not intertwine. Moreover, adding new adapter's responsibilities (e.g., to adapt yet another commercial component) becomes easier due to the separation of adaptation concerns. In a class-

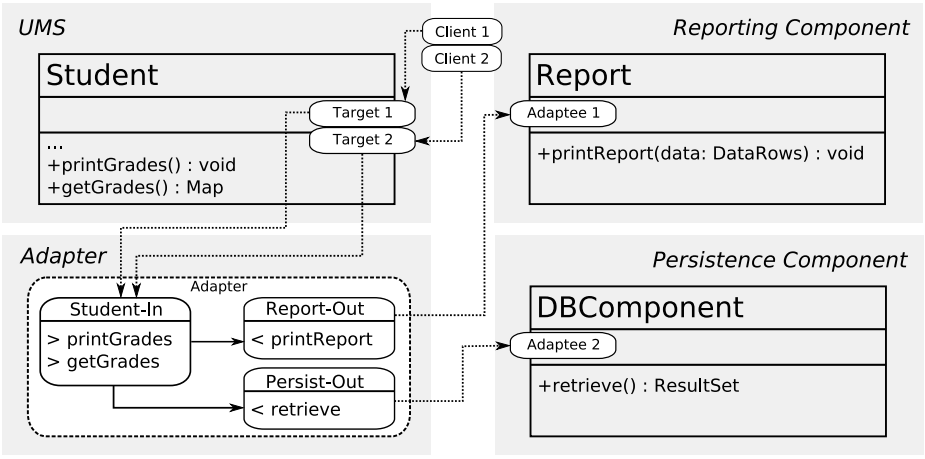


**Fig. 3.** Role-based adapter. Roles annotate classes playing those roles.

based realization, such separation is only possible using the complex role object pattern [3], which is in fact a workaround of language limitations to realize roles directly.

## 3 Challenges and Limitations

An important conceptual issue to be mentioned is that applying role-based adaptation to adapt class-based components reduces potential power of a pure object-based design (as envisioned by Reenskaug [4]). In our case it is not possible to realize the pattern only in roles, because at least some of them need to be bound to actual components' classes. In particular, in a strongly-typed class-based system, at least the target class needs to be specified statically.

A limitation inherent to ObjectTeams/Java is that a role can only have a single base class. As a consequence, the Adapter role cannot be realized by instances of different classes at run-time. This decreases reuse, because adapter roles, once defined, can only be used for a single class. If another class needs the same functionality, another role needs to be defined again, possibly duplicating the same implementation.

The major practical challenge we stipulate regarding role-based adaptation is that the learning curve implied the application of a new technique may not be accepted by developers. Since developers are in general reluctant to learn new programming languages and, even more important, have to admit a certain degree of obsoleteness of their conventional class-based adapter realization, it is not clear, whether such technique can be easily accepted by them.

## 4 Related Work

Technically most closely related work to our approach is of Bergmans and Aksit [5] on composition filters—a technique enhancing ordinary objects with input and output

filters for incoming and outgoing messages correspondingly. Each filter may reject or accept a message using certain acceptance conditions. If a message is accepted, it can optionally be altered and forwarded to a target. The target itself can be chosen using certain selection logic. However, there is no discussion whether their approach is applicable for adaptation in case of component integration.

## 5 Conclusion

Using conventional class-based realization of adapters leads often to highly complex adaptation code that is hard to understand, maintain and evolve. A role-based realization of adapters in a language supporting roles explicitly may considerably reduce code complexity due to the separation of adaptation concerns in the resulting implementation. Even more important, such realization preserves initial adaptation decisions made and contribute furthermore to the maintainability of adapters.

We will further investigate the frontiers of role-base adaptation, its practical realization, advantages and limitations in one of the authors' Bachelor thesis (currently in progress) [6].

## References


1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
2. Herrmann, S., Hundt, C., Mosconi, M.: ObjectTeams/Java Language Definition - version 1.0. Technical Report 2007/03, Technical University Berlin (2007)
3. Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The role object pattern. In: PLoP'97: Proceedings of the 4th Pattern Language of Programming Conference. (1997)
4. Reenskaug, T.: Working with Objects: The OOram Software Engineering Method. Manning Publications (1996)
5. Bergmans, L., Aksit, M.: Principles and design rationale of composition filters. In R. Filman, T. Elrad, S.C.M.A., ed.: Aspect-Oriented Software Development. Addison-Wesley (2004) ISBN 0-32-121976-.
6. Götz, S.: Role-based adaptation (2008) http://www1.inf.tu-dresden.de/~s9288421/papers/goetz-gb-thesis.pdf.