Großer Beleg

Role-based adaptation

submitted by

Sebastian Götz

born 19.05.1984 in Dresden

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: MSc. Ilie Şavga
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted November 14, 2008

Contents

Chapter 1

Introduction

One of the basic principals in software engineering is *divide and conquer*. Software systems are developed to address problems, which shall be solved. Hierarchical decomposition is one way to derive subproblems, which are easier and can be solved separately. Mechanisms to merge subsolutions are required to obtain an overall solution for the original problem. This thesis examines a novel mechanism to merge subsolutions.

1.1 From Object-Orientation to Roles

40 years ago, in 1968, leading experts of software industry meet in Garmisch to discuss the so called "software crisis". The complexity of problems grew faster and faster and computer power increased rapidly. As consequence it became catchier to write correct, understandable and verifiable software. Object-orientation was one of the many solutions for this crisis. It combined data-centered approaches, which were beneficial to describe static properties of information, but weak in describing functionality, with behavior-centered approaches, which were weak in describing static properties of information, but excellent in describing functionality.

Object-oriented design is based on the concept of objects and classes as common abstractions on them. The Unified Modeling Language [8] (UML) provides a set of models to design the structure and the behavior of software systems. A model to describe the structure is the class diagram, whose main elements are classes, associations and inheritance relations. Furthermore operations and attributes are added to classes. Next the behavior is designed using sequence diagrams, in which the interaction of objects - instances of classes described in the class diagram - is described for *specific* scenarios. Collaboration diagrams may be used as an alternative to sequence diagrams. They describe *specific* scenarios of interacting objects, too. Models usable for *general*, i.e. non-exemplary, descriptions of collaborations are missing. Additionally state diagrams are used to describe states of objects, conditional events leading to a state change and actions triggered by state changes. But the essence of what is done today using the object-oriented paradigm is to describe *object interactions* - that is *collaborations* of objects.

The term *role* stems from theater. During a performance actors play different roles. E.g. in a theater play about "Dr. Jekyll and Mr. Hide" one actor plays two roles. In real life we play many roles at the same time. E.g. a person may play the role of a teacher, father and car-owner concurrently. If he has a car accident and totals his car he hopefully stops playing the role of a car-owner, but continues to play his other roles.

A more formal approach to describe roles is to identify **role types** as dynamic types, which are *non-rigid* and *founded* [11]. Non-rigid means that an actor may stop playing a role without losing its identity. A person playing the role student stays the same person if she stops being a student. On the contrary if a book is burned, it is not of type book anymore and loses its identity. Founded means, that a type is only defined in relation to other types. The type *driver* is founded, because the type driver relates to the type person or human being. A natural type is rigid and/or not founded. Because role types are founded, roles need to have a player. Roles further require a context, i.e. a delimited environment. For example role student belongs to the context "University".

Common problems and their solution have been (and still are) collected in catalogs as design patterns. The most famous catalog of design patterns is the one published by the gang of four (GOF) [9]. To describe a design pattern they first name and classify it. The intent of the pattern, a motivating scenario, the applicability and structure is part of each pattern description, too. Interestingly they identify participants - "classes and/or objects participating in the design pattern and their responsibilities" [9, p.8] - and describe collaborations of them. Indeed participants are roles, albeit they are not identified as such. The structure of the pattern, mainly described using UML class diagrams [8], is a prerequisite. The description of collaborating participants is the most important part of the pattern description. In the following description on implementation, all identified participants are mapped to classes. Thus much information about participants and how they collaborate is lost. Sample code, known uses and a set of related patterns finish the description of a pattern.

For example the adapter design pattern is described using 4 roles: client, target, adapter and adaptee. The client role is played by an object, which wants to consume functionality, which is provided by an object playing the adaptee role, whose interface does not fit to what the client expects. The interface the client expects is provided by an object playing the target role. To provide functionality from the adaptee object through the target object an object playing the adapter role is needed[1]. Depending on the scenario these roles are played by different objects. An object playing the adapter role may also play the target role, if it has the interface the client expects. In object-oriented design these roles are mapped to classes. This leads to a loss of information about the collaboration. This is because multiple roles may be merged into the same class and thus get intertwined.

Figure 1.1 depicts two kinds of object-oriented adapters. Figure 1.1(a) uses delegation between adapter and adaptee, figure 1.1(b) inheritance. Because delegation works on the level of objects and inheritance in this context on the level of classes the adapter types are called object and class versions of the adapter pattern. The choice of the kind of adapter depends on the language and is a design decision.

Both figures, 1.1(a) and 1.1(b), have 3 classes. One class for each role. Figure

[1]not necessarily a separate object

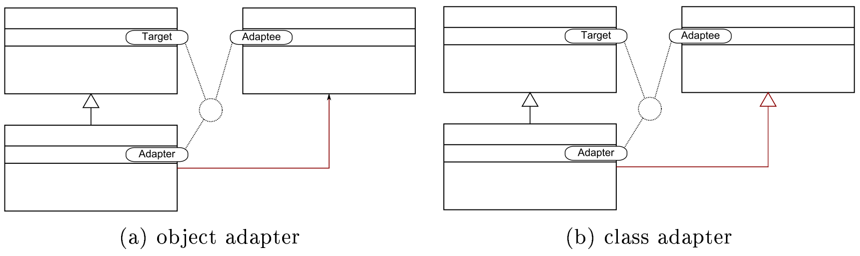


Figure 1.1: Structure of object-oriented adapters represented as class diagram. Roles are annotated as rectangles with rounded edges, based on a notation introduced by Riehle in [22]. To emphasize the togetherness of roles to a pattern a circle in the middle of the figures connects each role belonging to the same pattern using lines. The client role is omitted.

1.2 shows another possible class structure for the adapter design pattern. The difference is, that this time only 2 classes are used. The target and adapter role are both merged to the same class. Without deep knowledge of the patterns applied to these two classes it is impossible to identify which code stems from which role or pattern. This is because on the level of code there is no explicit hint, that the adapter design pattern has been applied. Hence the information, which patterns have been applied and which roles are mapped to which classes, is lost.

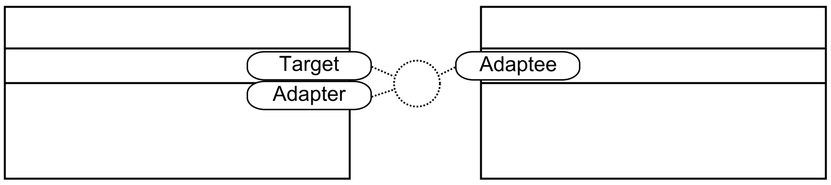


Figure 1.2: Another possible class structure of the adapter design pattern

The OOram Software Engineering Method [20] was introduced by Reenskaug. He argues the class/object duality, modeling on two abstraction levels, and investigates a unification of both concepts (classes and objects) - the OOram role model. This model abstracts from interacting objects to interacting roles. It abstracts from static types (classes) to dynamic types (role types). Roles are interconnected in role models. Because an object may play several roles this leads to separation of concerns, where each concern is encapsulated in a role model. The system is hence described using a set of role models, which finally have to be merged. The focus is on collaborations. not on structure, as it is in common object-oriented design.

As Riehle [21] showed, roles of a design pattern can be expressed in role models. Hence patterns used in software construction and evolution can be described separately and are not intertwined in the implementation. This heavily improves the maintainability of systems, because the original intend of engineers is no longer lost. Further improvement originates from programming languages supporting roles on the level of implementation, because even in case design documents get lost, the information about collaborations is still present in form

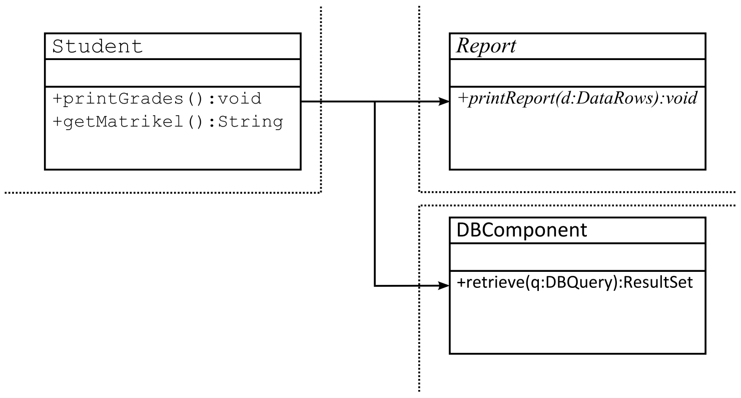


Figure 1.3: Class-based realization of the example scenario. Arrows denote UML associations. Dotted lines highlight boundaries of components. For clarity, only classes directly participating in the collaboration are shown.

of code. Thus roles greatly improve maintainability of software systems.

1.2 University Management System

The following example is based on [10]. Imagine a University Management System (UMS) having a class `Student`, which has a method `printGrades()`. This method prints out all grades a student got in each subject he took an exam in. The data needed is fetched from a comma separated file. The users of the UMS come up with new requirements. First they want to print not only to the console, but to a PDF, too. They furthermore do not want to use comma separated files for persistence anymore. Instead they prefer the usage of a database. To incorporate these changes the developers of the UMS decide on buying 2 commercial components. One for database persistence and another for reporting, which includes functionality for printing to PDF. They decided against implementing by themselves, because of 3 reasons. First they estimated that buying the commercial components is much cheaper. Second the commercial components are written by experts in the corresponding domain, hence those developers are much deeper in topic than the developers of the UMS. Finally the commercial components are maintained by a third party, hence the commercial components do not need to be maintained by the developers of the UMS.

A closer look at the commercial components shows, that each uses multiple classes. The reporting component provides class `Report`, which has a method `printReport`, which takes an instance of class `DataRows` as an argument. `DataRows` is a class specific to the reporting component and represents a single row in the (tabular) report. The database component provides a class `DBComponent`, which has a method `retrieve`, taking an instance of class `DBQuery` as an argument and returning an instance of class `ResultSet`. `DBQuery` and `ResultSet` are among `DBComponent` classes specific to the database component. A `DBQuery` encapsulates an SQL query, a `ResultSet` encapsulates a set of tuples returned by the database. The whole scenario is depicted in Figure 1.3,

Listing 1: intertwined code in `Student.printGrades`

```
DBComponent db = ...;
Report report = ...;

public void printGrades() {
  //Create Query from Student's Data
  String sql = 'SELECT * FROM StudentGrades WHERE matrikel = '+this.getMatrikel();
  DBQuery q = new DBQuery(sql)
  ResultSet rs = db.retrieve(q);
  //Transform ResultSet to DataRows
  DataRows data = new DataRows();
  while(rs.next()) {
  Subject sbj = new Subject(rs.getString(1), rs.getString(2));
  data.addString( sbj.toString() );
  //...
  report.printReport(data);
}
```

where the classes' names and methods are written in different fonts to demarcate, that they belong to different areas of concern. Everything, which belongs to the Report's concern, is written in *italic roman font*. The DB concern is demarcated by sans serif font. Finally the student's concern is written in `big typewriter font`.

A possible implementation of method `printGrades` in class `Student` is shown in Listing 1. The fonts used correspond to the fonts used in Figure 1.3.

As Listing 1 shows, the implementation of `printGrades` contains heavily intertwined code. This is because the task of printing a student's performance list requires instances of classes belonging to all three components to interact with each other in a very close manner. To integrate the functionality of a component one does not only need to call the appropriate methods of classes of the component, but transform the return values and passed parameters, too. In a class-based approach this type conversion is specific to each pair of components, which talk to each other. In the example three type mappings are defined and hence intertwined in method `printGrades`. These are Student — Report, Student — DBComponent and Report — DBComponent. Beside these type mappings the *integration logic*, that is how Report and DBComponent need to be used, is located in method `printGrades`. To make a long story short: four different concerns are tangled in method `printGrades`.

Adapter code needs maintenance, too. Imagine a new version of the reporting component released, which is not compatible with its former version that is the new version does not compile and/or link against the original adapter code. As a consequence the adapter code needs to be manually adjusted (*adapter upgrade*), whereby bugs may be introduced, which could affect the integration of other components. For example an adjustment for a new version of the database component may lead to bugs in the integration code of the reporting component. Imagine that method `retrieve` of class `DBComponent` is renamed to `retrieveData` and a new method `retrieve` is added to class `DBComponent`, which returns metadata about the database connection in form of a `ResultSet`. The original adapter code compiles and links against the new version of the database component, but misbehaves, because of the changed semantics of method `retrieve`. A maintainer who is not aware of the 2 changes applied to class `DBComponent` may try to change the code which makes instances of class `Subject`. This is because he might think that the content of the instance of `ResultSet` returned by method `retrieve` provides the data in some other format. This may lead to bugs in the integration code for class `Student`.

Maintainers of adapter code need to have deep knowledge about each domain their adapters address. This leads to high training costs. This problem aggravates when maintainers leave the company, whereby new maintainers need to be trained.

1.3 Separation of Adaptation Concerns

The problems identified in Section 1.2 all ascribe to a single problem: code belonging to different concerns is intertwined in the implementation if the adapter design pattern is realized in a class-based environment. Class-based adapters mix communication and computation. This thesis examines if the ability to describe patterns using roles combined with novel programming languages supporting roles on the level of implementation helps to overcome the aforementioned problems in the context of adaptation. Therefore the thesis focuses on the adapter design pattern as introduced in Section 1.1.

The application of programming languages supporting roles as first order programming constructs allows to separately realize the roles of the adapter design pattern. Code responsible for communication with classes providing functionality (providers), i.e. code belonging to the *adaptee role*, is separated from code belonging to the *adapter role*, which specifies providers to be called in which order and how to handle the results (computation). The *target role*, which is responsible for providing an interface expected by the client, is defined separately, too. The forth and last role of the adapter design pattern, the client role, does not need to be expressed separately, because the only responsibility of this role is to invoke methods of the interfaces provided by the target interface. Finally the type conversion concern is separately expressible.

Figure 1.4 depicts a *role-based adapter*. Its main parts are called mediator, In-Roles and Out-Roles. The part called mediator encapsulates the code of the adapter role. In-Roles encapsulate code of target roles, whereas Out-Roles represent adaptee roles. Clients call methods provided by the target role. That is they pass control to the target role, which is why I call such roles "In-Roles" - they take over control flow into the adapter and pass it to the mediator. The

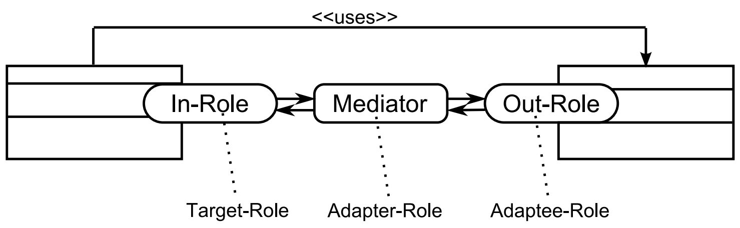


Figure 1.4: Overall structure of a role-based adapter mapped to a class-based implementation. The class on the left hand side uses functionality provided by the class on the right hand side, which however does not provide an interface the class on the left hand side expects. Adaptation is realized by the role-based adapter.

task of a mediator is to connect and intercede between different parties. The task of the mediator of a role-based adapter is to connect and intercede between In- and Out-Roles, i.e. specifying which Out-Roles have to be called in which order. Out-Roles are responsible for communicating with classes providing functionality, i.e. they represent adaptee roles. They are called Out-Roles, because they pass control flow out of the adapter.

Using this realization of the adapter design pattern, code belonging to different adaptation concerns is separated from each other. Thus the problem forming the origin of all problems identified in Section 1.2 can be solved using role-based adapters. The approach described in this thesis is partly published in [10].

Chapter 2

Role-Based Adaptation

Role-based adapters (RBAs) base on a set of concepts, which are described in this chapter. First the environment to which RBAs are applied is outlined in Section 2.1. The basic elements of RBAs are introduced in Section 2.2, followed by an advanced concept in Section 2.3. Finally achieved objectives are examined in Section 1.3.

2.1 Collaborating Components

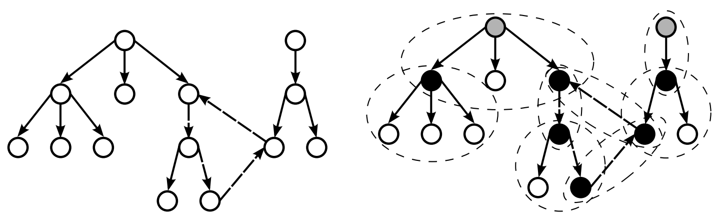
Components are units of composition with contractually specified interfaces and explicit context dependencies only [25]. Class-based components consist at least of a set of classes, a set of *required* and a set of *provided* interfaces. Required interfaces are used to explicitly model context dependencies. In order to specify a valid configuration of components each required interface needs to be bound to a provided interface of another component.[1] Provided interfaces are used as entry points to the components, that is methods of these interfaces are designed to take over control flow and direct it into the component to accomplish a specified task. Components may have multiple provided interfaces. Each is designed for a different use case or a different set of use cases.

In an integration scenario components play the role *service requester* and/or *service provider*. A component, which wants to use other components, is a service requester and the components used are service providers. This defines a *uses*-relation between components. Used components may use further components, too. A tree of requesters and providers is the result. A tree is a special graph. Graphs consist of nodes and edges, which connect nodes. Edges are either directed or bidirectional. If an edge is pointing to a node it is called an incoming edge for that node. If an edge points from a node it is called an outgoing edge. A tree has two special types of nodes: root and leaf. The root node does only have outgoing edges. Nodes, which are neither root nor leaf, have a single incoming edge and at least one outgoing edge. Leafs have no outgoing edges, but a single incoming edge, too. In the tree of requesters and providers leafs are service providers, nodes are both service providers and requesters, and the root is a service requester only. Components may be service providers for multiple service requesters, which leads to multiple request trees

[1]or to a provided interface of itself, which leads to recursion

which overlap each other and hence form a directed acyclic graph (DAG). DAGs differ from trees in that nodes are allowed to have multiple incoming edges, thus no distinction between root, node and leaf is needed. A further requirement of DAGs is that no cycles between connected nodes are allowed. Because service providers may use components which directly or indirectly[2] use themselves, the uses-configuration forms a directed *cyclic* graph.

Collaborations between service providers and requesters form contexts, i.e. delimited environments, for the participating roles. A context always delimits one service requester and all its used service providers. If multiple service requesters use a single service provider, the service provider is part of multiple contexts, because it participates in each context the service requesters form (one per requester). In case a service provider is using further components it is at least part of two contexts, namely the context where it is used by other components and the context where it is using other components. Figure 2.1(a) shows an example configuration of multiple components using each other and highlights all contexts, which exist. Figure 2.1(b) depicts three different types components may have regarding the uses-configuration.



(a) Uses-Configuration. Solid circles depict components. Arrows denote usage. Slashed arrows demarcate cyclic usage relations.

(b) Contexts and different types of components. Dashed circles demarcate contexts, grey circles denote service requesters, black circles service requester and providers and white circles services provides.

Figure 2.1: Uses-configuration of components.

It is important to note, that components being service provider **and** requester play only one of these roles per context, except when a component uses itself. Multiple contexts may coexist simultaneously.

2.2 Basic Elements of Role-Based Adapters

The ability to describe design patterns using roles combined with the usage of novel programming languages, which support roles on the level of implementation, enables the realization of the adapter design pattern in a way that avoids intermixture of adaptation concerns.

Figure 2.2 depicts the basic elements of a role-based adapter. The main parts are an In-Role, Out-Roles and a mediator. C1, C2 and C3 are components, where C1 is requesting functionality from C2 and C3.

The different parts of role-based adapters correspond to the different roles of the adapter design pattern. The target role is realized as In-Role, the adapter

[2]via other service requesters

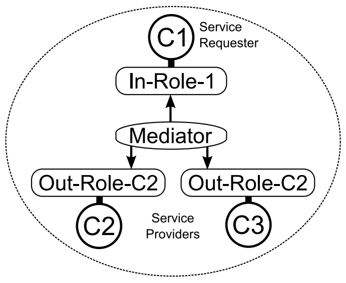


Figure 2.2: Basic elements of a role-based adapter. C1, C2 and C3 are components.

role as mediator and the adaptee role as Out-Role. The naming is based on the direction of control flow. In-Roles direct control flow *into* the adapter. Out-Roles direct control flow *out* of the adapter to service providers. The task of a mediator is to connect and intercede different parties. The mediator in role-based adapters connects and intercedes (or simply adapts) In- and Out-Roles. Thus the collaboration of In- and Out-Roles is described by the mediator[3]. In-Roles are connected to required interfaces of service requesting components, Out-Roles to provided interfaces of service providing components.

Whenever a client calls a service requesting component the control flow is intercepted by the corresponding In-Role, directed to the corresponding Out-Roles by the mediator, forwarded to the service providing components by these Out-Roles and finally returned through the Out-Roles, the mediator and the In-Roles to the service requesting component.

2.3 Composite Collaborations - The Necessity of Ambassadors

In case a service requester uses service providers, which themselves are service requesters, a composite collaboration results. Figure 2.3 depicts such scenario.

Now imagine that `C5` shall use `C0`, thus introducing a cyclic dependency, which indicates a potential deadlock. Additionally a further context is introduced, because each collaboration between a service requester and service providers is delimited by its own context. In such scenario four separate mediators exist, which only know about "their" collaboration. Hence deadlocks cannot be detected. To avoid this problem a further context — the supercontext, containing all contexts, needs to be introduced.

A further task, besides deadlock detection, of the supercontext is to contain intermediate types. Components may participate in a lot of contexts, thus playing multiple In- and/or Out-Roles. The task of type conversion should not be assigned to In- and Out-Roles, as this would lead to code replication. Imagine an extension to the running example. The reporting component, which consists of a class `Report` and a class `DataRows`, is using another component

[3]Another possible naming scheme as suggested by Stephan Herrmann is *Observer* instead of *In-Role* and *Actuator* instead of *Out-Role*, what leads to the abbreviate **OMA — Observer, Mediator, Actuator.**

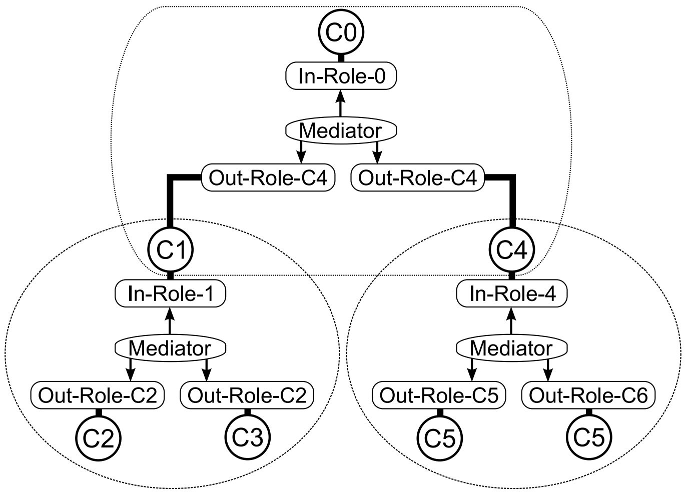


Figure 2.3: Composition of role-based adapters.

developed for generation and modification of PDF[4] files. Figure 2.4 depicts this scenario. The university management system (UMS) component is connected to the reporting component, which is connected to the pdf component, and to the database component.

Because the UMS component is connected to the reporting and database component, for a single use case only, one role based adapter is used to realize both connections. A further role based adapter is needed to connect the reporting component and the PDF component. Thus two roles based adapters (RBAs) are used. If In- and Out-Roles are responsible for type conversion, the conversion of class `DataRows` to an intermediate form is implemented twice. First for the Out-Role of the RBA between the UMS component and the reporting component. Second for the In-Role of the RBA between the reporting component and the pdf component. The task of the Out-Role is to convert data used inside the RBA to a type of the integrated component, for example a usual `Map`[5] to class `DataRows`. The task of the In-Role is to convert data of a type specific to the requesting component to a type specific for the adapter - an intermediate type. In the example, class `DataRows` is converted to a conventional `Map`. Using intermediate types in adapters decouples adapters from components they connect.

The extended running example does not suffer from code replication, yet. A further extension to the example reveals that In- and Out-Roles should not contain logic for type conversion. Imagine the reporting component is using a further component, enabling the generation and modification of ODT[6] files and another component for Microsoft Office Documents. The extended scenario is shown in Figure 2.5. The extension leads to further RBAs, connecting the reporting component to the ODT component and DOC component, respectively. The In-Roles of these newly introduced RBAs contain logic to convert class `DataRow` to a usual `Map`, too. If further formats (e.g. PostScript) shall be

[4]portable document format
[5]java.util.Map - part of the Java Runtime Library
[6]Open Document Text, part of the OASIS Open Document Format for Office Applications

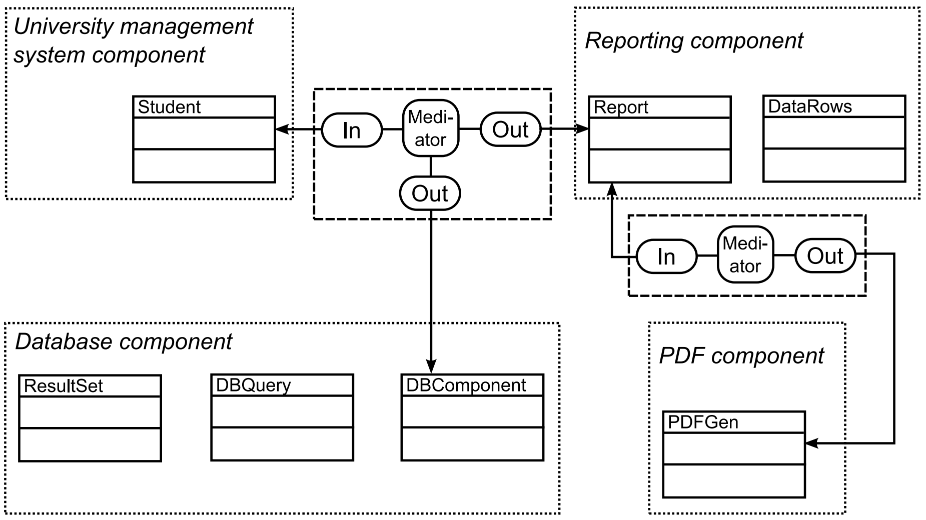


Figure 2.4: Extended running example. The reporting component uses a PDF component. Dotted rectangles denote component boundaries. Arrows represent the playedBy relation, i.e. some role is played by some class. Dashed rectangles denote the boundaries of a role based adapter. Solid lines emphasize how the parts of an RBA are interconnected.

supported by the reporting component each further integrated component aggravates the replication of type conversion code. Please note, that introducing further RBAs, instead of further Out-Roles to the RBA connecting reporting and PDF component, decreases coupling between the components. Improved reusability of RBAs is a direct consequence of constructing small RBAs, even though multiple small RBAs could be realized as a big one.

For each component a further role, responsible for type conversion and defined in the supercontext is introduced, to avoid code replication. Such roles are called *ambassadors*, because they act like representatives for "their" component and are able to explain classes of "their" component to foreigners. Figure 2.6 depicts the scenario shown in Figure 2.5 enriched by ambassador roles. If ambassadors are realized as roles, they need a player. In the example classes `Student`, `Report`, `DBComponent`, `PDFGen`, `ODTGen` and `DOCGen` are used as players. In general ambassadors do not need to be realized as roles. They could also be realized as classes, but then the component needs to assure that at any time a single instance of its ambassador is available. The additional class does not break the requirement that components shall be independently deployable, but the additional class does not contribute to the components functionality and thus should not be part of the component. Hence ambassadors need to be part of the supercontext specific for the system.

Ambassadors are developed for a set of role based adapters. Each RBA is developed for another use case. Different tasks are solved using different data structures, i.e. types. Therefore role based adapters may use different intermediate types for the same component specific type. Ambassadors need to know about all intermediate types, which belong to types specific to their component. The ambassador is reusable, if a component is reused in another system config-

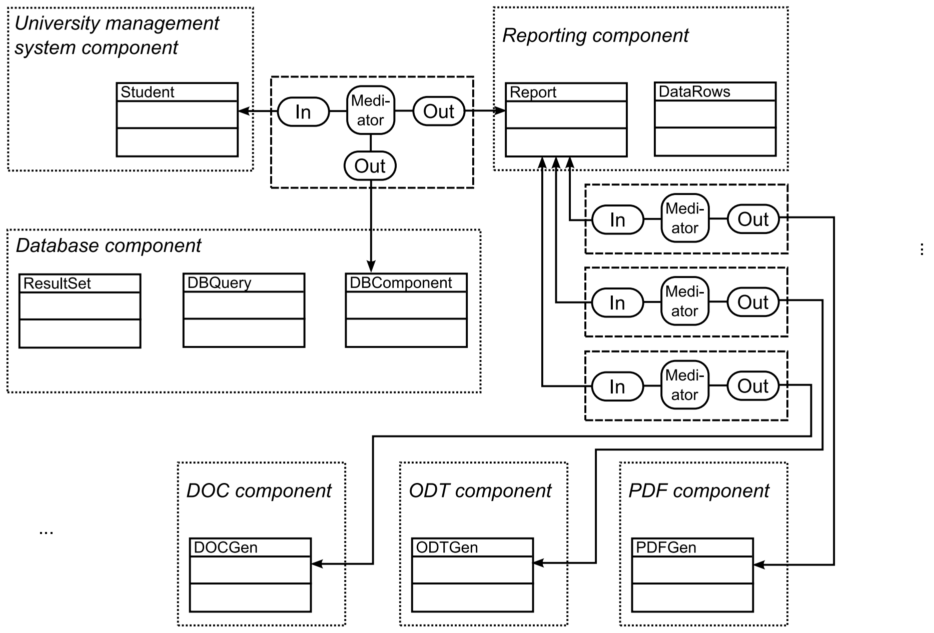


Figure 2.5: Further extended running example. The reporting component now uses multiple component for different output formats. For each component an additional Out-Role is introduced.

uration. In case a new intermediate type is needed, the ambassador, however, needs to be extended.

2.4 Achievement of Objectives

Comparing the class-based solution of the original example to the role-based solution reveals that the application of RBAs to simple systems leads to much more code that is to be written by developers. The ***advantage*** of this big initial effort is ***less effort regarding maintenance*** of the system. If, for example, method `printReport` of class `Report` of the reporting component is renamed, only the respective Out-Role needs to be changed. In case multiple components use the reporting component it is thus easy to localize the places in the code, which need to be changed - the Out-Roles connected to class `Report`. Because the only task of Out-Roles is to pass on control flow to their player, i.e. the class of the component the respective RBA integrates, Out-Roles are not complex and hence easy to maintain. Class-based adapters on the contrary contain all concerns of adaptation. The complexity of methods like `printGrades` shown in Listing 1 explodes when systems get bigger. The localization of places in the code, which need to be changed due to the aforementioned method rename in class `Report`, is thus much more complicated. Because of this the adjustment of such methods is demanding and highly error-prone.

Another ***advantage*** of role-based adapters in comparison to classical class-based adapters is ***improved flexibility*** in assigning tasks to developers. If a system is constructed using class-based adapters, developers who are responsible for maintenance of that system need to have deep knowledge about each component the system consists of. This is because class-based adapters use and transform types specific to the components they connect. Imagine a company, which developed the running example using class-based adapters and employed developers, who are specialized on reporting, other developers who are specialized on databases with only one of these developers knowing well both domains. Only the developer with knowledge about both domains is able to accomplish the task of maintenance in an efficient way, because all other developers first need to invest time to understand the types of the database component. If the systems were developed using RBAs each developer is able to accomplish the task of maintenance, because no special knowledge about the connected components is needed. Thus developers can be allocated in a much more flexible way.

In the original running example introduced in Section 1.2 the implementation of method `Student.printGrades` needs to be realized separately, to contain a mediator and to be split into three distinct roles defined in one context, which is depicted in Figure 2.7. One for class `Student`, one for class `Report` and one for class `DBComponent`. Instances of these classes are the players of the roles. Instances of class `Student` are played by an In-Role called `StudentIn`, instances of class `Report` and `DBComponent` are played by Out-Roles called `ReportOut` and

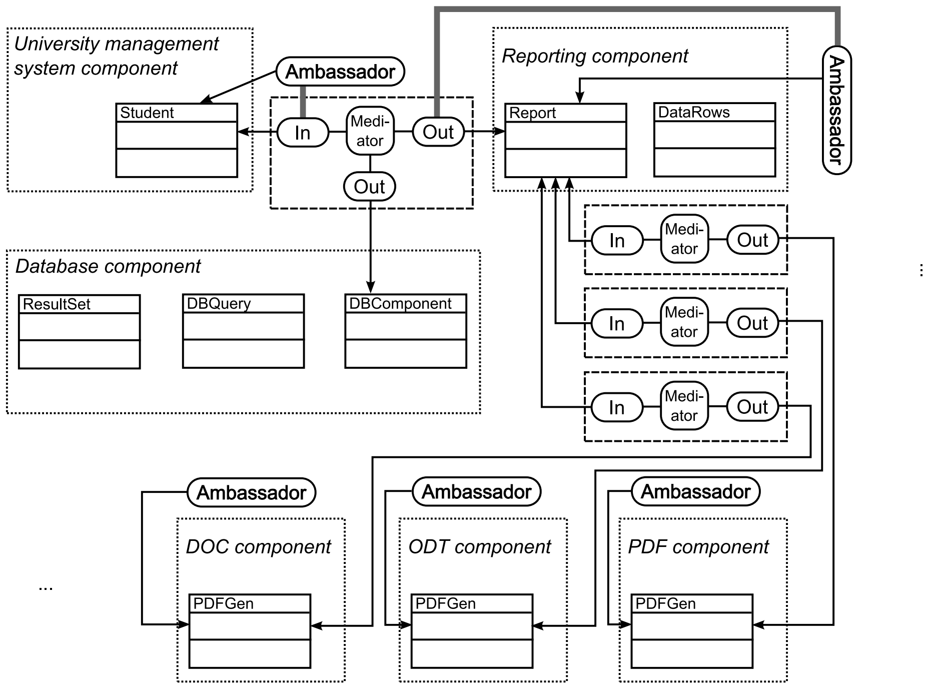


Figure 2.6: Figure 2.5 enriched with ambassadors. To ensure clarity the relation of In- and Out-Roles to ambassadors is only depicted for the RBA between the university management system component and the reporting component. Bold grey lines denote a usage relationship. In- and Out-Roles of an RBA use the ambassador of the component their player belongs to.

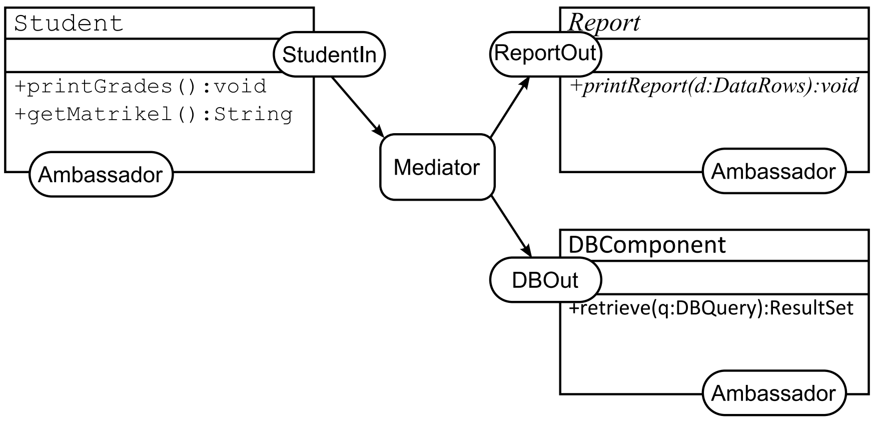


Figure 2.7: Role-based realization of example scenario. Roles are drawn as rectangles with rounded edges, based on Riehle's notation [22]. Arrows denote control flow.

`DBOut`. The mediator defines that first the database component needs to be used to fetch the data and then the reporting component is to be used to print the data. Furthermore a supercontext needs to be defined, which contains three ambassadors, one for each component, for type conversion. The data returned by the database component is transformed into an intermediate type by the role responsible for this component in the supercontext. Before this data is passed to the reporting component it is transformed from the intermediate type to the data type specific for the reporting component, i.e. `DataRows`, by the ambassador of the reporting component, defined in the supercontext.

The application of roles on the level of implementation to realize adapters leads to separation of adaptation concerns, which allows for separate construction and maintenance of different concerns. Multiple concerns in adaptation code have been examined:

1. code responsible to communicate with components, i.e. their interfaces
2. collaboration code to bind requesting and providing components
3. type conversion between components

Please remind that the **divide-and-conquer** principle says, that complex problems are easier to solve, if they are divided into subproblems. Each subproblem is solvable on its own. To derive an overall solution all subsolutions need to be merged. Separation of concerns is a divide-and-conquer strategy. Each concern is solvable on its own. Development of separate concerns is easier than developing all concerns at once, because the responsible developer needs to focus on a single concern at each step and does not need to care about other concerns at this point in time. Programming languages supporting roles on the level of implementation provide an infrastructure that accomplishes the merger of subsolutions, i.e. concerns. Because of this, developers using RBAs really only need to focus on a single concern at a time. The third ***advantage*** of RBAs is hence ***easier development*** of adapters, albeit it is most likely that experienced developers will not sense simplification. This is because experienced developers are used to construct complex adapters and thus do not sense them as complex.

In- and Out-Roles encapsulate and hence separate code responsible for communication with each component. Thus the responsibility to maintain the communication code for each component-adapter pair can be assigned to developers responsible for that component. The communication code differs between different pairs of components and adapters, because the communication is specific for each adapter. The benefit is, because the communication code is separate from each other and the code belonging to other concerns, the developer responsible for this code does not need to know about the other concerns. Maintenance of class-based adapters requires the maintainer to have deep knowledge in all concerns intertwined in the adapter. Mediators encapsulate collaboration code. Type conversion between components is separated by ambassadors defined in a supercontext, whereby code replication is overcome. Due to separate type conversion the mediators rely on intermediate types only. Hence maintainers responsible for this code only need to know about a single domain, too — the *intermediate domain*, which is based exclusively on intermediate types.

In summary the **advantages** of role-based adapters are:

1. less effort for maintenance,
2. more flexible allocation of developers and
3. easier development of adapters.

The **disadvantages** of role-based adapters are:

1. the need to write more code and
2. initial training costs, in order to teach the developers a new programming language, supporting roles on the level of implementation.

Role-based adapters *conceptually* overcome all problems mentioned in Section 1.2. Chapter 3 shows the realization of these concepts.

Chapter 3

Realization

In order to realize the concepts introduced in Chapter 2 an appropriate language has to be found. The approach was implemented for the example (and a couple of other examples) introduced in Section 1.2. This chapter first presents the process of language selection in Section 3.1, introduces the main concepts of the chosen language in Section 3.2 and finally presents the implementation of the approach in Section 3.3.

3.1 Language Selection

The selection of language is based on 5 requirements:
The language needs to

1. support roles on the level of implementation as first order constructs,
2. be mature[1],
3. support an explicit notion of contexts,
4. support nesting of contexts and
5. provide mechanisms for redirection of control flow.

A language can only be chosen, if it fulfills each of these requirements. Tool support was a further requirement, which is usually given implicitly by requirement (2).

The following 6 candidates have been examined for selection, focusing on the above mentioned requirements:

1. AspectJ
2. Scala
3. Lava
4. EpsilonJ
5. McJava

[1]more than 3 years of development, at least version 1.0

6. ObjectTeams/Java

AspectJ [1] only fulfills the requirements (2) and (5), that is of being mature and providing mechanisms for redirection of control flow. It does not support roles on the level of implementation, because aspects as defined in AspectJ are not roles. Apel et. al. [3] provides a distinction between aspects and roles[2]. Though Apel and colleagues focused on roles on the level of design, time variation (and more) distinguishes roles from aspects. Contexts and their nesting are not supported, too.

Scala [7] does not support roles in the level of implementation directly. An approach introducing the concept of roles for Scala was introduced in [19]. This approach is in an early phase of research and hence requirement (2) is not fulfilled. Traits are used to group roles in contexts, while roles themselves are described as inner traits. Hence requirement (3) is fulfilled, but as trait nesting is used to define roles (on nesting-level 2) nesting of contexts (requirement (4)) is not possible. Finally redirection of control flow is achieved via proxies, which intercept calls to objects potentially having some active roles. Thus requirement (5) is fulfilled, too.

Lava [15] is an extension for the Java language [4] which aims among other goals at providing explicit support for consultation and delegation fields, as well as anticipated parent types. Anticipated parent types can be used to realize roles. Possible roles of a player are modeled as potential superclasses of the player. Lava provides support to change these anticipated superclasses at runtime. Unfortunately Lava lacks support for multiple inheritance[3], although inheritance is modeled as specific form of (dynamic) delegation, due to 3 reasons. First Kniesel claims that using multiple delegation breaks binary compatibility. The second reason is the potential loss of simplicity. Finally the semantics of a class having multiple anticipated parent types active at the same time may lead to semantic errors. Thus in order to realize roles using anticipated parent types stacking of roles is needed. That is if multiple roles are to be active for the same player the player only points to one of these roles, where this role points to the next role and so on. In case these roles are defined using an own inheritance hierarchy no solution exists, yet. Hence requirement (1) is not fulfilled. An explicit notion of contexts is not supported, but can be derived by following the anticipated parent type relations. This does not lead to separate contexts, but to a boundary around all contexts the role is involved. Nesting of contexts is thus not supported, too. Redirection of control flow is one of the strengths of Lava, as multiple types of delegation exist (consultation, delegation, anticipated parent types). Lava is currently available in version 0.21, whereas many features, like e.g. the change of anticipated parent types at runtime, are not supported, yet. Hence Lava is not mature.

EpsilonJ [17] was designed to support roles on the level of implementation. A new keyword for roles (`role`) is used to explicitly define roles. Furthermore contexts are specified using a new keyword, too (`context`). EpsilonJ is a language extension to Java, where code written in the EpsilonJ language is to be transformed into usual Java code. Technically roles are translated to inner classes, while their outer class forms their context. Requirements (1) and (3)

[2]and features

[3]Lava conceptually supports multiple unanticipated parents, which is, however, not realized, yet.

are thus fulfilled. Nesting of contexts is not yet supported, though it's likely that in future versions this support will be integrated. Players are usual Java classes, which are marked with the keyword `player`. They thereby gain methods to bind roles to them. This binding leads to multiple objects representing the same conceptional object. Currently the language lacks a mechanism to ensure, that in case a role is active and needs to be called, clients cannot use the player directly without involving the active role. That is the problem of object schizophrenia [12] is not solved, yet. In order to redirect control flow to active roles the developer needs to explicitly cast the player to the role. Thus redirection of control flow is not supported as it needs to be manually specified. Requirement (5) is hence not fulfilled. Beside the problem of object schizophrenia a set of further problems are still to be solved in EpsilonJ. It is e.g. not possible to add multiple instances of the same role type to a single player. This is because the binding of a role to its player is saved as a hashmap in the player, whereas the name of the role type is used to address the role instances. Adding a further instance of the same role type simply overrides the pointer to the former role instance. Because there are still a set of problems to be solved requirement (2), maturity, is not fulfilled.

McJava [14] is a language extension for Java, developed by the same developers who developed EpsilonJ. McJava enriches Java with mixin types. Realizing roles as mixins instead of inner classes eliminates the problem of object schizophrenia. Redirection of control flow is supported by McJava. McJava does provide roles as first class citizens, namely as mixins, but no explicit notion of contexts. Nesting of contexts is thus not supported, too. A combination of EpsilonJ and McJava would improve EpsilonJ in that it additionally fulfills requirement (5). McJava is still in research (since 2004) and no compiler or translator is publicly available. Hence McJava has not reached maturity, yet. Only requirements (1) and (5) are fulfilled.

ObjectTeams/Java supports roles on the level of implementation as first order programming constructs. Furthermore contexts can be specified explicitly. Nesting of contexts is supported, too. Though ObjectTeams/Java does realize roles as inner classes, too, it has solved the problem of object schizophrenia by modifying the byte code of the players (by redirecting each call to a player to its active roles). Extensive tool support is provided by the ObjectTeams Development Tooling for Eclipse including debugger, syntax highlighting, code completion, quick fix hints and a lot more. ObjectTeams/Java is developed since seven years and its language definition is currently of version 1.2. Due to the fact, that ObjectTeams/Java fulfills all requirements, including extensive tool support, is under development since seven years and its currently available version is 1.2, it can be seen as mature.

Table 3.1 lists all languages examined and points out which languages fulfill which requirements. ObjectTeams[13] fulfills all 5 requirements. Because of this I choose ObjectTeams/Java for the realization of the concepts introduced in Chapter 2.

3.2 ObjectTeams/Java: A Short Introduction

ObjectTeams/Java[13] (OT/J) is a realization for Java of the language model called ObjectTeams. OT/J is realized as an extension to the Java language,

Language / Requirement	(1)	(2)	(3)	(4)	(5)
AspectJ	no	yes	no	no	yes
Scala	yes	no	yes	no	yes
Lava	no	no	no	no	yes
EpsilonJ	yes	no	yes	no	no
McJava	yes	no	no	no	yes
ObjectTeams/Java	yes	yes	yes	yes	yes

Table 3.1: Candidate languages for realization and which requirements they fulfill.

which does not need a modified Java Virtual Machine. Instead OT/J-code is compiled into JVM-conform bytecode.

The most important concepts of ObjectTeams with regard to this thesis are teams, roles, the played-by relation, callins, callouts and base classes.

Roles are defined in contexts. E.g. the role *Student* is defined for the context *University*. In other contexts this role would lack semantics. Contexts may have fields and methods. E.g. the context *University* could own a field for its name and methods to matriculate and exmatriculate students. In OT/J contexts are described using **teams**. Teams are modeled like classes and are denoted by the additional keyword *team*.

Inner classes of teams are **roles**, as long as they are not marked as teams. Hence nesting of teams is supported. As roles are always played by a player, they need to be bound to their player. This is done using the **playedBy** relation. Inner classes of teams, which are roles, define their player in a way similar to inheritance. Instead of *extends* the new keyword *playedBy* is used. The player is called **base** in ObjectTeams.

The semantics of roles is to change structure and behavior of their player as long as they are **active**. Roles can be activated in OT/J over their team. Either directly by invoking the method `activate()` or indirectly by specifying constraints under which the team shall be active. Roles may have fields and methods, too. E.g. role *Student* may have a field for its matriculation number and a method `learn()`. In case a role gets active these fields and methods are *"added"* to the player, thus changing the structure of the player. In order to change the behavior of the player 2 mechanisms are provided by OT/J: callins and callouts.

A **callin** is defined in a role and specifies which methods of the player are to be intercepted and redirected to methods of the role. Hence callins take over control flow from the player to the role. **Callouts** are defined exactly the other way around that is they specify which methods of the role shall forward the control flow to which method of the player. Both mechanisms strongly rely on method names. The adjustment of behavior currently only works for method executions. An improvement of the language, to react on thrown exceptions or on any event fired, is not planned, yet.

Problems like object schizophrenia are addressed using byte code manipulation on the player/base side. That is calls to a base, which has active roles are always redirected to the active role first. As ObjectTeams is under active development since more than 6 years many problems have already been addressed. Hence OT/J is the most mature language currently available for the purpose of using roles on the level of implementation.

3.3 Implementation Using ObjectTeams/Java

The implementation of the concepts introduced in Chapter 2 using ObjectTeams/Java is straightforward. In- and Out-Roles are implemented as roles. These are defined inside a team, which represents the Mediator. All roles are bound to their players using the `playedBy` relation. In-Roles use `callins` to intercept the control flow, whereas Out-Roles use `callouts` to forward the control flow. Whenever an In-Role intercepted a method call of its player it calls the mediator. The mediator further delegates control flow to the corresponding Out-Roles, which forward the control flow to their players.

The data flow depends on whether independent functionality is to be integrated or functionality, which delivers some result to the service requesting component. In case of independent functionality there are two data flows—one in the service requesting component and one between the service providing components. If the functionality is meant to deliver some result to the requester, there is only one data flow between all components. This data flow is bidirectional, that is, it starts from the requesting component, goes into the components and finally goes back to the requester.

3.3.1 Realization of the Motivating Example

Figure 3.1 shows a class diagram in ObjectTeams-notation of the running example introduced in Section 1.2.

Teams are represented as packages annotated with a yellow circle with a white T, whereas the package consists of 3 parts — the upper part lists fields, the middle part lists methods and the lower part lists roles. Roles are depicted as classes annotated with green circles and a white R. Like classes roles have fields and methods, but callin and callout definitions, too, which are listed in the lower part of the roles. Roles are bound to their base classes using an arrow annotated with "playedBy". Callins are expressed as yellow circles with "«", callouts with "»".

Callin expressions may take 2 forms. If the base method, which is to be intercepted, has the same signature as the role method the shorter form can be used, which consists of the role method name, followed by a "«", one of three possible modifiers (before, after, replace) and the base method name. Depending on the type of method, which is to be intercepted, a different modifier is to be used. If the base method is a hook method, that is it is meant to be filled by some third party, usually `replace` is used. If the base method is not a hook method, either `before` or `after` are used, though `replace` may be used, too. In Figure 3.1 the modifiers are omitted, for readability.

The second form of a callin expression is to be used, if the base methods signature differs from the role method. In this case the full signature of the role method, followed by "«", one of the three modifiers and the full signature of the base method need to be written. Furthermore the `with`-clause needs to be expressed, which specifies how the signature of the role method and the base method are to be transformed to each other. In Figure 3.1 this case is depicted like the short form, but followed by `with {...}`.

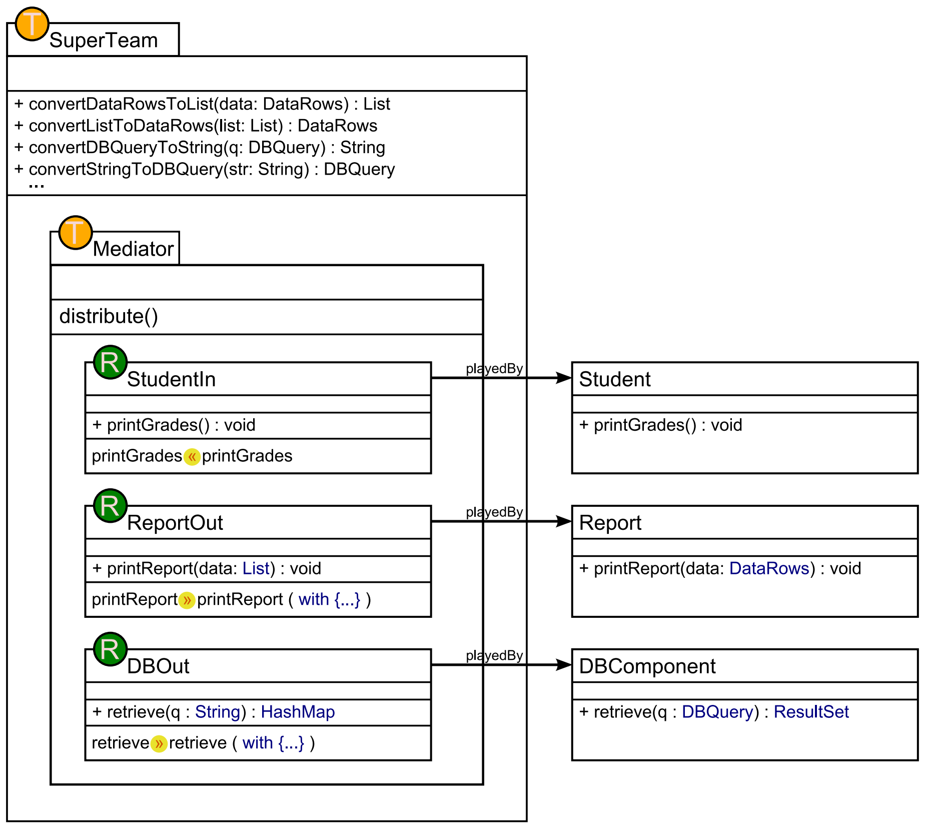

SuperTeam
+ convertDataRowsToList(data: DataRows) : List
+ convertListToDataRows(list: List) : DataRows
+ convertDBQueryToString(q: DBQuery) : String
+ convertStringToDBQuery(str: String) : DBQuery
...
Mediator
distribute()
StudentIn
+ printGrades() : void
printGrades « printGrades
ReportOut
+ printReport(data: List) : void
printReport » printReport (with {...})
DBOut
+ retrieve(q : String) : HashMap
retrieve » retrieve (with {...})
playedBy
playedBy
playedBy
Student
+ printGrades() : void
Report
+ printReport(data: DataRows) : void
DBComponent
+ retrieve(q : DBQuery) : ResultSet

Figure 3.1: Implementation of role-based adapter for the running example

It is important to note, that all types, which are specific to a component (i.e. `DBQuery`) always are transformed into an intermediate type, which is specific to the mediator/team. This way each role of the team only depends on the team itself and the component it is meant to integrate. For clarity ambassadors introduced in Section 2.3 are omitted. Instead all methods of them are realized as team methods.

Figure 3.2 depicts the control flow between the components. The control flow is demarcated as a sequence of red arrows, where each is annotated with a number in a red circle, used to order the arrows.

Steps 7 and 11 are depicted with a dotted arrow, because they are no explicit method invocations, but method call returns. Furthermore 4 steps have been omitted in Figure 3.2 for readability. Before steps 5, 6, 9 and 10 are executed additional calls to the super team's methods are necessary. E.g. before step 5, the invocation of method `retrieve` of class `DBComponent`, first the argument `q` needs to be translated. Thus first method `convertStringToDBQuery` of the super team is called. For step 6 the return value, which is an instance of class `ResultSet`, needs to be translated to a `HashMap`. Thus method `convertResultSetToHashMap` of the super team is called in advance to step 6. Similar type conversion calls are executed in advance to steps 9 and 10.

Listing 2 shows an extended version of the implementation of role `StudentIn`.

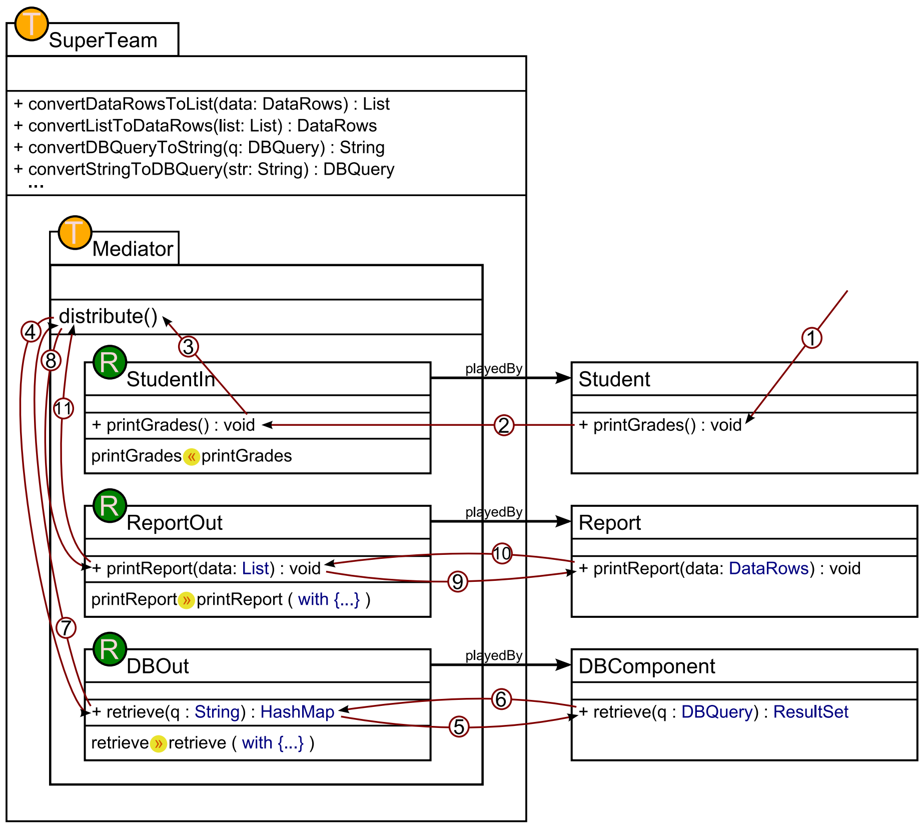

SuperTeam
+ convertDataRowsToList(data: DataRows) : List
+ convertListToDataRows(list: List) : DataRows
+ convertDBQueryToString(q: DBQuery) : String
+ convertStringToDBQuery(str: String) : DBQuery
...
Mediator
distribute()
StudentIn
+ printGrades() : void
printGrades « printGrades
ReportOut
+ printReport(data: List) : void
printReport » printReport (with {...})
DBOut
+ retrieve(q : String) : HashMap
retrieve » retrieve (with {...})
playedBy
Student
+ printGrades() : void
Report
+ printReport(data: DataRows) : void
DBComponent
+ retrieve(q : DBQuery) : ResultSet
1
2
3
4
5
6
7
8
9
10
11

Figure 3.2: Control flow (numbered arrows) among the components

This role additionally defines a callout to method `getMatrikel` of class `Student`, in order to fetch the matriculation id of the student. This id is e.g. used to construct the SQL query, send to the database component and hence is passed to method `distribute` as an argument.

Listing 3 shows method `distribute`, which encapsulates the whole communication and integration logic. It first passes control flow to the database component and uses the return value of this call to call the reporting component. It only communicates with roles.

Listing 4 denotes the implementation of the `DBOut` role along with method `transformResultSetToHashMap`, which is used by this role. The type conversion between String and `DBQuery` is not realized as a separate method call to the super team, but using the constructor of class `DBQuery`.

Finally Listing 5 shows the role `ReportOut` along with the method, which transforms the intermediate `HashMap` to the type `DataRows`, specific for the reporting component — `transformHashMapToDataRows`.

3.3.2 Challenges on the Level of Implementation

A set of problems on the level of implementation were identified. These differ from problems on the conceptual level, because they are based on the imple-

Listing 2: Implementation of role StudentIn

```
public class StudentIn playedBy Student {
  void printGrades() -> replace void printGrades();
 Integer getMatrikel() <- Integer getMatrikel();

  callin void printGrades() {
   distribute(getMatrikel()); //notify mediator
  }
}
```

Listing 3: Implementation of method distribute

```
public void distribute(Integer matnr) {
 Map<String, Double> grades =
     dbOut.retrieve("SELECT * FROM grades WHERE matrikel = "+matnr);
 reportOut.printReport(grades);
}
```

Listing 4: Implementation of DBOut role and method transformResultSetToHashMap

```
public class DBOut playedBy DBComponent {
  Map<String, Double> retrieve(String query) -> ResultSet retrieve(DBQuery query)
      with { new DBQuery(query) -> query, result <- transformResultSetTo-
HashMap(result) }
}

public Map<String, Double> transformResultSetToHashMap(ResultSet rs) {
  Map<String, Double> ret = new HashMap<String, Double>();
   try {
    while(rs.next()) {
      ret.put(rs.getString(2), rs.getDouble(3));
    }
    return ret;
  } catch(Exception e) {
    e.printStackTrace();
  }
   return null;
}
```

Listing 5: Implementation of ReportOut role and method transformHashMapToDataRows

```
public class ReportOut playedBy Report {
  void printReport(Map<String,Double> data) -> void printReport(DataRows data)
    with { transformHashMapToDataRows(data) -> data };
}

public DataRows transformHashMapToDataRows(Map<String,Double> data) {
  DataRows dr = new DataRows(2);
  for(String key : data.keySet()) {
    dr.addString(key+";"+data.get(key));
  }
  return dr;
}
```

mentation language used. The following problems hence are a consequence of choosing ObjectTeams/Java.

Team Activation

The first problem is **role or team activation**. In order for roles to get effective, they need to be activated. It somehow needs to be expressed, that an actor starts or stops playing a given role. ObjectTeams/Java calls this activation and provides different mechanisms to activate roles.

The most common mechanism is unconditional team activation. To activate a team the method `activate()` needs to be called on an instance of that team. All roles in a team are bound to a player using the keyword **playedBy**. The consequence of calling `activate()` is, that all instances of base-classes referenced by roles in that team now start playing the respective role. As counterpart to `activate()` the method `deactivate()` is provided. If this method is called, all instances of base-classes referenced by roles of that team stop playing the respective roles.

Additionally guard predicates can be used. Guard predicates can be used on four different levels. They can be applied to team classes, role classes, role methods or callin method bindings. The effect of these predicates is, that if they evaluate to true the players referenced start playing the respective roles until the predicates evaluate to false. If these predicates are used on the level of a team class this effects all roles encapsulated in that team. To specify the activation of roles in more detail the predicates may additionally be attached to role classes. As role classes consist of attributes, a set of constructors, a set of methods and a set of callin (as well as callout) bindings, an even more detailed specification is possible. Single methods and even single method bindings can be (de)activated using guard predicates on the appropriate level. The expressiveness of guard predicates forces up even more, because they can be combined on different levels at the same time. It is thus possible to provide a general condition for the whole team to be active and more special conditions for single methods to be active. If a role method binding is active or not depends on all guard predicates defined for this binding and all predicates defined in the hierarchy above this binding. Unfortunately the usage of guard predicates still needs the team to be explicitly activated using the method `activate()`.

The problem for role based adapters is, when and where to activate the team. If the role-based adapter is deployed as a separate component the activation can be done during component startup. E.g. in OSGi [2] the team can be activated in class `Activator` of the bundle containing the adapter. This means that code of the original application needs to be enhanced.

Fortunately in ObjectTeams/Java version 1.2 config files have been introduced with the possibility to specify which teams shall be activated at program startup time. These config files are simple text files containing the fully qualified names of the teams, which shall be activated. To start the application an additional argument needs to be passed to the VM: *'-Dot.teamconfig=<config-file-name>'*. This way the source code of the application does not need to be changed, which is an important property, because there are scenarios where developers do not have these sources. E.g. a team of developers, who want to write an extension for an application provided by a third party.

Team activation via config files poses another problem. If a team is activated this way, no reference to this team exists. Such teams hence cannot be deactivated. If guard predicates are used a complete team deactivation may be unnecessary. But if not, a so called *manager team* can be used. This manager team encapsulates the original team and needs to be activated via config file, too - instead of the original team. The manager team provides methods and callin bindings for activation and deactivation of the encapsulated team. Though this way the manager team can never be deactivated, the nested team, i.e. the team we want to automatically activate at startup time, can.

Method Calls from Team to Role and Between Roles

Teams encapsulate a set of roles. Teams and roles may have methods. Because each role is implicitly connected to its encapsulating team instance, a call from a role to a method of its team does not pose a problem. But how can a team call a method on one of its roles? And how may a role call a method of another role?

In order to call a method on a role a reference to this role is needed. Team methods usually instantiate roles and hence have a reference to the role. But sometimes the role shall not be instantiated, because it already exists. In this case the team method needs to get either directly the role as an argument or the respective base instance. To get an active role of a base instance ObjectTeams/Java provides a mechanism called **lifting**. Non-static team methods are allowed to use the keyword **as** to express a **declared lifting** of its parameters. The parameter is not expressed only by `Type name`, but by `BaseClass as RoleClass name`. Inside of the method the parameter *name* can now be used as it would be of type `RoleClass`.

Besides declared lifting ObjectTeams/Java provides **lifting constructors**. Lifting constructors are declared in roles and take an instance of the base class as argument. They are generated by the compiler, but it is possible to provide custom lifting constructors, too. The main task of these constructors is to check, whether the given base object, the given team object and the statically given role type are identical to an existing role object which then is returned. If no appropriate role object is found a new role object is implicitly created. Role based adapters make heavy usage of default lifting constructors.

In case roles and base-classes are part of an inheritance hierarchy a set of rules are evaluated to lift the given base object to the correct role object. These rules are necessary, because a developer may want to lift a base object to a specific role object, which is bound to a base class which is a superclass of the base objects class. The movement upwards the inheritance tree, to check whether there is an appropriate superclass, is called **smart lifting**.

Hence to call methods of a role from a team or from another role an instance of that role or its base class is needed. To get an active role of a base object lifting can be used.

As counterpart to lifting, ObjectTeams/Java provides **lowering**. This mechanism allows to derive the base object from a role object. This can even be done implicitly during an assignment. E.g. `BaseClass b = roleObject;`.

Whether a role and a base conform to each other and hence are substitutable is expressed by **translation polymorphism**. This special form of polymorphism is the result of merging inheritance-based polymorphism with the additional conformance between role and base objects.

Chapter 4

Related Work

Two approaches strongly relate to role-based adapters and are examined in the following. Programming languages supporting roles on the level of implementation have already been examined in Section 3.1.

4.1 Compositional Filters

In [6] Lodewijk Bergmans and Mehmet Aksit describe one of the first aspect-oriented approaches for composition of program elements. The approach is based on the object-oriented paradigm. Objects interact with each other by sending messages to one another. Such a message could be an event or a method call. To change the behavior of an object it hence suffices to change the incoming and outgoing messages of that object. This is in essence what compositional filters (CFs) do.

The main goals of the CF model are composability, evolvability, robustness, implementation-independence and dynamics. The composition capabilities shall be hierarchically, that is modules of behavior shall be composable into new modules. Existing systems shall be extensible in a modular way. The application of the CF model shall not impair the creation of correct systems. It shall be possible to provide multiple implementations with different characteristics. Finally dynamic adaptation of structure and behavior shall be supported.

The compositional filter model distinguishes two kinds of abstractions: *concerns* and *filters*. Concerns are primary program elements - classes as known from the object-oriented paradigm. The purpose of filters is to encapsulate extensions to concerns. An important difference to other aspect-oriented and role-based approaches is, that compositional filters focus only on behavior, not on structure.

Filters are grouped into *filter modules*, which provide an execution context for them - like roles, which need a context to exist, too. Filter modules are elements of reuse and are used to instantiate filter behavior. A *concern instance* as depicted in Figure 4.1 denotes an object, composed with a set of filter models. It is important to note, that concern instances are single entities with their own identity[1]. Because of this they do not suffer from object schizophrenia [12]. The different parts of concern instances can be identified on the conceptual level, but

[1]that is the object and its filters do not have separate identities, but a single one

not on the physical level. The object enclosed by a concern instance is called the *implementation object*.

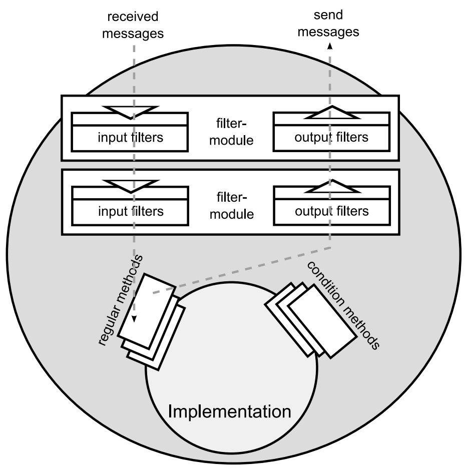


Figure 4.1: a concern instance and its parts, redrawn from [6]

Bergmans and Aksit distinct input and output filters. Messages sent to an object need to pass all input filters of that object. Messages sent from an object need to pass through all outgoing filters. The task of filters is to reject messages, change them or just pass them by. Thus each filter inspects and possibly manipulates the messages it receives.

An important requirement on implementation objects is, that they need to provide an interface, which consists of *regular methods* and *condition methods*. The functional behavior of the object needs to be accessible by regular methods. The current state of the object needs to be accessible by condition methods, which need to be side-effect free. The usage of condition methods to express the state of the object leads to implementation-independence and improve reuseability, because condition methods can be used by other filter modules, too.

Filters consist of two parts. A filter type and filter elements. Filter elements are composed to filter patterns. The filter pattern is used to declaratively express whether an inspected message *matches* or not. Depending on the composition operator used to construct a filter pattern the respective filter elements are evaluated. In general filter patterns and filter elements evaluate to a boolean value. If the pattern evaluates to false the message is *rejected* by this filter and will be passed to the next filter, unless the filter type does not impose an action to take place in this case. If there is no following filter an exception *'message not understood'* is raised. One of the predefined filter types is `Error`. Its semantics are, that if the message is rejected an exception is raised, else the message is passed to the next filter. If the pattern evaluates to true an action depending on the filter type is invoked and the message passes to the next filter or, if there is no following filter, to its target.

Concern instances encapsulate filter modules and the implementation object. This enables *intra*-object crosscuts, because the filters in the filter modules are only involved if a message is sent to this object. To enable *inter*-object crosscuts a mechanism called *superimposition* is provided by the CF model. Concerns are enriched with a superimposition specification, which describes which modules are to be superimposed where. To describe the positions where to superimpose, *selectors* are used. Their task is to select objects from the instance space, that is all objects of the current application. The Object Constraint Language [18] is used to express selectors. Figure 4.2 depicts 3 concerns whereof 2 are superimposed on the third.

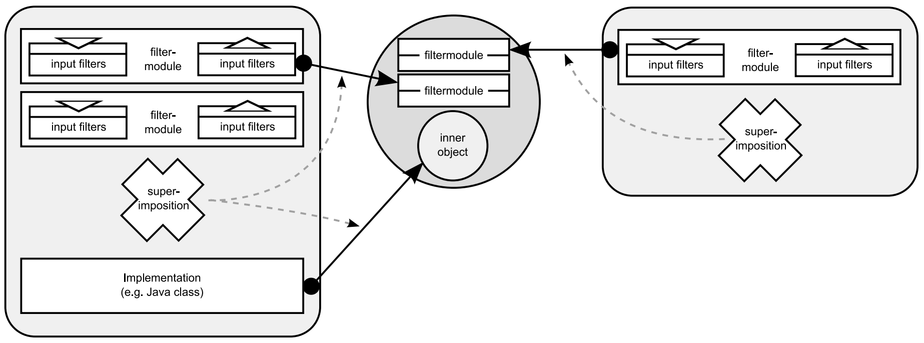


Figure 4.2: Superimposition using 3 concerns, redrawn from [6]

The closest relation between the CF model and RBAs is the adaptation of incoming and outgoing messages. RBAs use In- and Out-Roles to connect components. In-Roles intercept method calls and redirect them to the mediator. Thus conceptually In-Roles of RBAs can be seen as a special form of input filters. Out-Roles delegate method calls, whereby the arguments being send are potentially modified by the mediator. Thus conceptually Out-Roles are a special form of output filters.

The CF model is an extension to the object-oriented paradigm, having composability as focus. RBAs base on the role paradigm, but do not extend it. Compositional filters are more general than RBAs, because it is possible to express RBAs using the CF model. A role-based adapter is realized as a concern. In-Roles are realized as input filters of type `Meta` and Out-Roles as output filters of type `Detach`. The implementation object of the concern is the mediator, whose method `distribute` needs to be enhanced with an argument for the reified message, generated by the Meta-Filtertype. Every method call will thus be intercepted, reified and send to the mediator. Using the Detach-Filtertype outgoing method calls will be delegated to the correct target objects. Using superimposition nesting of these adapters is possible, too.

Thus compositional filters and RBAs have adaptation of incoming and outgoing messages in common, but compositional filters are more general.

4.2 Exogenous Connectors

Component based systems consists of a component model and a composition technique [5]. Components encapsulate computation. The composition of components is based on either direct or indirect message passing. Method calls are messages, specifying the method and the arguments, passed from a sender to a receiver *directly*. Control flow originates in components. Information about connections between components is thus hard-wired in components, which leads to bad flexibility.

An alternative is indirect message passing, which is realized using the mediator pattern. To compose two components A and B, A notifies the mediator, which in turn calls B. This introduces a level of indirection, but decreases coupling between components, because now A only depends on the mediator, but not on B. The mediator is called connector, because of his task. Besides the mediator pattern other patterns, like *decorator* or *observer* can be used to realize connectors.

Indirect message passing separates computation from communication, which is encapsulated in the connectors. But control flow is still mixed in components and connectors, as components need to notify the connectors, which call methods of other components in turn. Control flow thus originates in components and is passed to connectors. The aim of *exogenous connectors* is to minimize coupling between components. Moreover reasoning about systems build using exogenous connectors becomes more practicable.

The idea of exogenous connectors is to completely encapsulate control flow in connectors. Thus components do not call each other or notify some connector. Instead the connectors call components. Imagine component A having a method m_1, calling method m_2 of component B. Using exogenous connectors the method call from m_1 to m_2 is removed from the component. Instead the connector first calls method m_1 and than calls method m_2.

To compose systems consisting of more than 2 components a hierarchy of connectors is to be used. The simplest form of connector are L1 connectors. L1 connectors connect to a single method and hence are called *method invocation connectors*. Connectors on level 2 connect two L1 connectors. Connectors on higher levels connect n connectors of lower levels. Figure 4.3 depicts an example hierarchy. Interestingly connectors on levels higher than 2 are polymorphic[2], whereas connectors on level 1 and 2 are not. This is because level 2 connectors are tight to level 1 connectors, which are tight to components. One of the major drawbacks of exogenous connectors is, that big systems lead to very complex hierarchies of connectors, which hence are hard to handle.

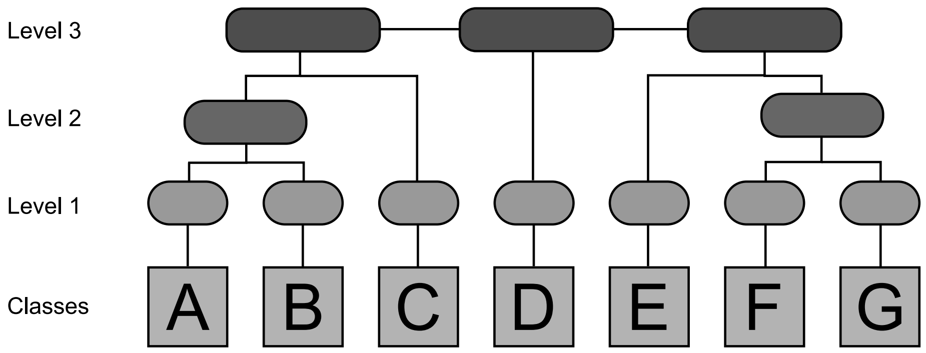


Figure 4.3: example of a connector hierarchy, redrawn from [16]

Role-based adapters can be seen as a special form of exogenous connectors if they are used exhaustively. The RBA approach does not require developers to remove "control flow related" code from components. But RBAs can be used

[2]L1 connectors connect to classes, L2 connectors connect L1 connectors, L_n connectors connect each type of exogenous connector

to realize exogenous connectors in an even better way, than described in [16], because of the ability to intercept method calls. The only violation is, that at one point the control flow needs to be initiated, which is usually not in a RBA. As soon as one method is called the In-Roles are able to intercept and pass the control flow to the mediator. Out-Roles are method invocation connectors (L1 connectors). The mediator contains the composition receipt for the L1 connectors, i.e. Out-Roles. As shown in 2.2 RBAs are composed hierarchically. Thus each RBA may connect multiple RBAs, whereas the mediator of this "super"-RBA contains the composition receipt. Thus RBAs are L2 and L_n connectors, too. RBAs are able to connect classes and themselves.

The usage of the mediator to describe compositional programs is powerful. It is up to the developer to decide how big such a single mediator shall be. It is on the one hand side possible to put a large amount of logic into the mediators and hence lower the amount of RBAs. On the other hand side it may be smarter to use more RBAs with smaller mediators. This can be seen as an advantage, due to the gained flexibility, or as disadvantage, due to the loss of developer-guidance.

Exhaustive usage of RBAs, i.e. all connections between components of a system are established using RBAs, leads to exogenous connectors and offers all benefits provided by them: minimized coupling between components (high flexibility), the ability to reason about composed systems and predictable assembly.

Chapter 5

Conclusion

This thesis introduces a novel mechanism to merge components using adapters. The idea of using roles on the level of implementation to realize adapters is motivated and conceptually described. A running example, highlighting the problems of class-based adapters (based on [10]) is introduced. For implementation the language ObjectTeams/Java (OT/J) has been chosen following a set of five criteria and the main concepts of OT/J are described. The realization of the running example using role based adapters, written in OT/J, is shown. In doing so problems identified for class-based adapters are shown to be solved. As any programming language, OT/J has its very own model. Section 3.1 shows, that all alternative languages are either not mature or do not support roles as first order programming constructs. Because of this, problems, specific to OT/J, examined during realization of the running example are described in more detail. Compositional filters and exogenous connectors are examined as related work. Finally the main contributions of the thesis at hand and future work are outlined.

5.1 Contributions

The first achievement is the identification of problems of class-based adapters in Section 1.2. If a set of components, where each has its own domain, are to be integrated, code belonging to these domains is heavily intertwined. Because of this a developer needs to have in-depth knowledge about all components. High training costs, which even get higher in case such trained employees leave the company, are a direct consequence. Tangling code of different concerns is identified as the root problem.

Using roles on the level of implementation solves the root problem. Code, which formerly was tangled, is now separated. Each participant in an integration scenario is integrated using a separately maintainable role. Chapter 2 describes in detail, how this separation of adaptation concerns is achieved. Adaptee code, i.e. code to communicate with components providing functionality to be used by others, is encapsulated in *Out-Roles*. Target code, i.e. code responsible for communication with components requesting functionality, is encapsulated in *In-Roles*. Adapter code consists of two parts. Which components are to be called in which order is the first part. How results and parameters are to be transformed is the second. The order of calling components is encapsulated in the *mediator*. Transformation of results and parameters is realized by reusable *ambassadors*, specific for each component.

A set of challenges specific to OT/J have been examined during the realization of the running example. The first problem - activation - is described in Subsection 3.3.2. Using OT/J it is necessary to explicitly activate roles. Role-based adapters shall be active all the time, but code responsible for activating them shall not be put into client code. Fortunately since version 1.2 of OT/J config files enable activation without touching client code. The second problem is due to the complex calling and substitution mechanisms of OT/J - lifting, lowering and translation polymorphism. Thus they are described in more detail in Subsection 3.3.2.

Finally role-based adapters are compared to compositional filters in Section 4.1 and exogenous connectors in Section 4.2.

5.2 Future Work

The recent appearance of mature programming languages, supporting roles on the level of implementation, opens a new field of research. The thesis at hand introduces role-based adapters, the realization of the adapter design pattern using roles on the level of implementation. In the remainder of this section three starting points for future work are explained.

5.2.1 Collection of Empirical Data Using an Experiment

The thesis at hand claims, that separation of adaptation concerns leads to lower maintenance costs. An **experiment** for verification should be done in the future, to show empirical data on the improvement due to using role-based adapters in spite of class-based adapters. Two teams of 5 students each shall develop an extended version of the university management system (UMS), as introduced in Section 1.2. Reporting shall support PDF, DOC and XLS as output formats. At least two different database vendors shall be supported. Furthermore students in the UMS attend courses, which take place at given points in time. A further component for scheduling is to be developed. The final application should allow the user to manage[1] students, their grades, courses and enrollments of students to courses. Besides printing support of a student's grades, the system shall be able to schedule courses of students. The first team shall use class-based adapters, the second team role-based adapters. The experiment consists of two phases. First both teams develop the base system, as described in Section 1.2. In the second phase both teams incorporate the changes listed in this paragraph. Besides time measurement and software metrics, a survey, which is to be answered by each student, shall be used to collect empirical data.

[1]support CRUD - create, read, update, delete

5.2.2 Using Role-Based Adapters for Change Encapsulation

Adaptation covers integration and evolution. Integration of new components into the overall system has been covered in the thesis at hand. Evolution has not been covered, though the adapter design pattern is usable for the evolutionary part of adaptation, too, as approaches like [23, 24] show.

Remind the example introduced in Section 1.2. As components are developed by different teams of developers, they may evolve independently from each other. Imagine a new version of the reporting component. Class `Report` is changed in that its method `printReport` is renamed to `generateReport` and class `DataRows` is refined to a class hierarchy having a class `Data` as root of the hierarchy and classes `RowSet` and `Table` as leafs. Class `RowSet` represents tuples of data, as class `DataRows` did in the first version of the reporting component. Class `Table` represents matrices of data.

To incorporate the new version of the reporting component each role-based adapter connected to class `Report` needs to be changed. The more adapters are connected to that class, the more effort is required to compensate the changes, even though using role-based adapters it is easy to find the appropriate locations in the code.

To reduce the effort for integrating a new version of a component, adapters can be used. In front of the new version of a component an adapter is to be put, which acts as the old version of the component. Adapters provide functionality to clients in a form clients expect. If a client expects the old version of a component an adapter is usable to provide the functionality of the new version in form of the old version. In [24] such adapters are realized using classes. The realization using role-based adapters should be examined in the future.

5.2.3 Parameterizable Role-Based Adapters

Role-based adapters connect components. The core of each RBA is the mediator, which specifies which functionality is to be called in which order and how to pass on the data. The thesis at hand does not pose any restrictions on mediators. However, special mediators and thus special kinds of RBAs can be identified.

For example a **sequence** RBA, which calls each component to be integrated and passes the result as argument to the next component. Such a role-based adapter is usable as a parameterizable template. The parameters are the components in the desired order.

Another example is a **pick** RBA, which is parameterized by multiple components[2], too. The difference to a sequence mediator is, that this kind of mediator asks the component specified by the first parameter, which of the other components, specified by the following parameters, to use.

Using such parameterizable RBAs compositions can be described like workflows. Thus parameterizable RBAs should be identified and examined in the future.

[2]i.e. methods of classes of the components

Bibliography

[1] AspectJ homepage. http://www.eclipse.org/aspectj/, 2008.

[2] The OSGi Alliance. OSGi Service Platform Specification. http://www.osgi.org, April 2007.

[3] Sven Apel, Don Batory, and Marko Rosenmüller. On the structure of crosscutting concerns: Using aspects or collaborations? In *Proceedings of 1st Workshop on Aspect-Oriented Product Line Engineering, AOPLE' 2006*, pages 20–24, 2006.

[4] Ken Arnold and James Gosling. *The Java programming language.* The Java Series, Reading, MA: Addison-Wesley, 1996.

[5] Uwe Assmann. *Invasive Software Composition.* Springer, Berlin, 2003.

[6] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development.* Addison-Wesley, 2004.

[7] Martin Odersky et. al. Scala language specification. http://scala.epfl.ch, 2004.

[8] Martin Fowler. *UML Distilled: Applying the Standard Object Modelling Language.* Addison-Wesley, 1997.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[10] Sebastian Götz and Ilie Şavga. Exploring role-based adaptation. In *Proceedings of the 5th ECOOP'2008 Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE '08*, 2008.

[11] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models.* PhD thesis, University of Twente, The Netherlands, 2005.

[12] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the 8th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '93.* ACM, 1993.

[13] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/-Java language definition - version 1.0. Technical report, TU Berlin, Fakultät IV - Elektrotechnik und Informatik, 2007.

[14] Tetsuo Kamina and Tetsuo Tamai. McJava - a design and implementation of java with mixin-types. In *Proceedings of Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, pages 398–414, 2004.

[15] Günther Kniesel. Type-safe delegation for run-time component adaptation. *Lecture Notes in Computer Science*, 1628:351–368, 1999.

[16] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proceedings of the 8th International SIGSOFT Symposium on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.

[17] Supasit Monpratarnchai and Tamai Tetsuo. The design and implementation of a role model based language, EpsilonJ. In *Proceedings of the Fifth International Conference in Electrical Engineering/Electronics, Computer, Telecommunications, and Information Technology, ECTICON' 2008*, 2008.

[18] OMG. Object Constraint Language, OMG Available Specification, Version 2.0. http://www.omg.org/docs/formal/06-05-01.pdf, 2006.

[19] Michael Pradel and Martin Odersky. Scala Roles - A lightweight approach towards reusable collaborations. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT '08)*, 2008.

[20] Trygve Reenskaug. *Working with objects - The OOram Software Engineering Method*. Taskon, Gaustadalléen 21, N-0371 Oslo 3 Norway, 1995.

[21] Dirk Riehle. A role-based design pattern catalog of atomic and composite patterns structured by pattern purpose. Technical report, Ubilab Technical Report 97-1-1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

[22] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '98*, pages 117–133. ACM Press, 1998.

[23] Ilie Savga, Michael Rudolf, and Sebastian Götz. Comeback!: a refactoring-based tool for binary-compatible framework upgrade. In *Companion of Proceedings of the 30th International Conference on Software Engineering, ICSE'08*, 2008.

[24] Ilie Savga, Michael Rudolf, Sebastian Götz, and Uwe Assmann. Practical refactoring-based framework upgrade. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE'08*, 2008.

[25] Clemens Szyperski and Cuno Pfister. Summary of the first workshop on component oriented programming. In *Proceedings of the 1st workshop on component oriented programming, WCOP'96*, 1996.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 14, 2008