Diploma Thesis

# DAMPF - Dresden Auto-Managed Persistence Framework

submitted by

Sebastian Götz

born 19.05.1984 in Dresden

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dipl.-Inf. Sebastian Richly
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted March 6, 2010

## Confirmation

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

Dresden, March 6, 2010

# Contents

# Chapter 1

# Introduction

The transformation of domain objects between object-oriented systems and relational databases has been investigated for a long time, resulting in industry-wide accepted standards for object-relational mappers, like the Java Persistence API[30]. In consequence software engineers got a sophisticated abstraction layer for persistency. Common object-relational mappers disburden software engineers from manually creating database schemata, but combine the schema definitions of the object-oriented and the relational world. Software engineers can even apply optimizations to the databases, like specifying indexes, without manually changing the database schema. Central to all systems is the data they process. Notably, the bulk of data storages used in combination with systems follows the relational model, whereas systems using the storage are object-oriented. Therefore, this thesis addresses problems in the field of object-*relational* mapping and does not utilize object-oriented database management systems (OODBMS). Additionally OODBMS lack generally accepted standards and still do not perform well, in contrast to relational database management systems.

This thesis presents a novel approach to persistency for object-oriented systems and systems utilizing roles as first-order constructs: the *Dresden Auto-Managed Persistency Framework*. Its acronym, DAMPF, is the German word for steam.

A lot of persistency mechanisms exist, but none of them is really transparent. If a developer decides to use a JPA implementation, he needs to provide vast amounts of information on how and what he wants to persist. Usually whole books need to be read on order to take full advantage of common object-relational mappers. This thesis aims on creating an object-relational mapper, which is transparent at a degree, so developers need to read only a single page to take full advantage.

Furthermore roles as an extension to the object-oriented paradigm will be supported. Notably, this thesis does not focus on roles known from role based access control! In short, roles describe dynamic behavior, are played by objects in a context and thereby allow expressing dynamic collaborations in a clean and compact way. Olaf Otto's extension[54] to the JPA provides persistency for systems written in a language supporting such roles, but does not fully utilize the power of roles themselves. Though the extension itself is written in that language, the real strength of roles, that is their dynamic properties, is not used. This is because only the chosen language has been investigated and that language is based on one of many understandings about the concept of roles, which focuses less on the dynamic properties, but more on the detachment of behavioral concerns.

Separate approaches for schema evolution, distribution in heterogeneous environments and context-based security in general exist, but no solution for object-role-relational mapping supports them. Special about all these three features is that they cannot be realized in such mappers with feasible effort, except the power of roles is utilized. Hence, the features of DAMPF are:

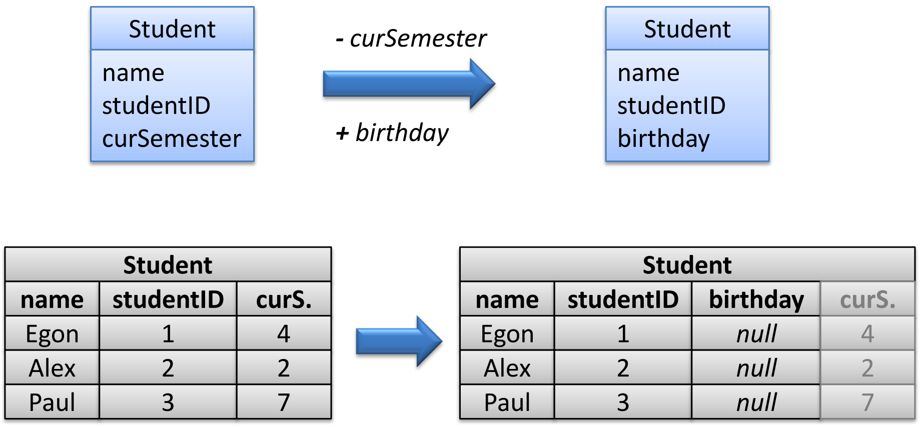
1. Transparency, by convention over configuration.
2. Support and utilization of roles.
3. Support for schema evolution.
4. Support for distribution in heterogeneous environments.
5. Support for context-based security.

These features make DAMPF unique, because no other approach supports all of them. Another contribution of this thesis is a Prolog program, realizing fully automatic database normalization.

The next paragraphs describe the three main features of the approach presented in this thesis in more detail.

**Schema Evolution.** The implementation of object-oriented and role-oriented software is an iterative, incremental process. Each iteration ends with an executable application, which is subject to tests. The next iteration modifies, that is mainly extends, the existing application. If the application utilizes a role relational mapper for persistency, the data of the application is likely to be lost after each iteration. Changes to the applications schema boil down to changes in the database. But object relational mappers do not support structural changes to the application without losing the data, which has been persisted in the database. DAMPF is the first tool, that offers this feature. In order to realize this feature, DAMPF provides facilities to change the schemata in the database and to migrate the data from old to new schemata. Notably, this feature allows for the evolution of the applications schema, which is bound to the database schema. The approach does not focus on database schema evolution, but on the integration of changes of the applications schema into the corresponding database schema.

**Example 1** *One of the most important domain concepts of a university management system is `Student`. When such a system is under development, such a concept is likely to change often. Figure 1.1 depicts an example scenario. In a first version, class `Student` comprises the attributes `name`, `studentID` and `curSemester`, which denotes the current semester of the corresponding student. The next development iteration leads to the removal of `curSemester` and the addition of the attribute `birthday`. The data, which already existed for the first iteration, needs to be migrated to the new schema.*



| Student | | |
|---|---|---|
| name | studentID | curS. |
| Egon | 1 | 4 |
| Alex | 2 | 2 |
| Paul | 3 | 7 |

| Student | | | |
|---|---|---|---|
| name | studentID | birthday | curS. |
| Egon | 1 | *null* | 4 |
| Alex | 2 | *null* | 2 |
| Paul | 3 | *null* | 7 |

Figure 1.1: Example for Schema Evolution Support in DAMPF.

As Example 1 shows, support for schema evolution requires more than applying schematic changes to the database. Existing data needs to be migrated. DAMPF provides a mechanism for both and supports an even more complicated scenario, which is illustrated in Example 2.

**Example 2** *Imagine an application for the management of students. Figure 1.2 depicts the example. In a first iteration students have an attribute* `year`*, storing the year they started to study. In the following iteration this field is removed and another attribute, pointing to the exam regulations the students study, is added. In a third iteration the year of study is added again, additionally to the attribute, pointing to students' exam regulations.*

In DAMPF the data from the first iteration will not get lost due to the second iteration, although the attribute for this data is removed temporarily from the class schema. The data from the first iteration will be available in the third iteration again! This is, because DAMPF does not remove attributes from the relational schema by default. A removed attribute will just not be fetched from the database. Hence, schema evolution is supported in a novel way, utilizing the abilities of the transformation between the relational and class schemata.

**Distribution in Heterogeneous Environments.** Nowadays software runs everywhere: in cars, PDAs, fridges, mobile phones and of course still on classic computers. In this context the distribution of software across multiple processing units is of very high importance. Processing units of mobile devices are by nature less powerful than classic, stationary processing units. In order to accomplish complex, cost-intensive tasks, mobile processing units use services, provided by more powerful units. Current technologies for this purpose are Enterprise Service Buses and Application Servers. In general the applied architecture is of client-server nature. Mobile devices are clients, using services, provided by servers. A direct consequence of this architecture is, that clients need to communicate with their servers, viz. they need a connection. Often this connection is unstable, due to the properties of the environment the mobile devices is currently located. This leads to the requirement to store important data

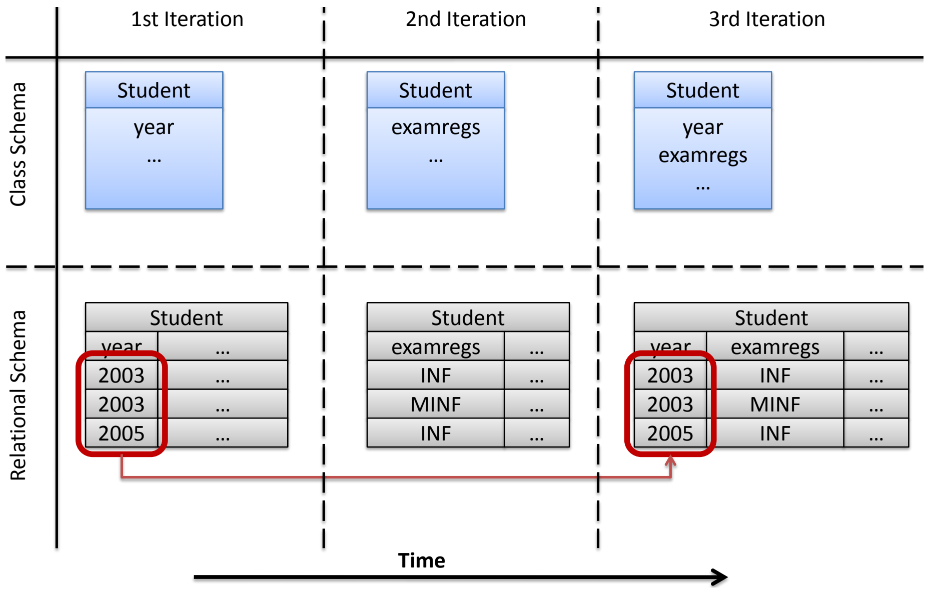


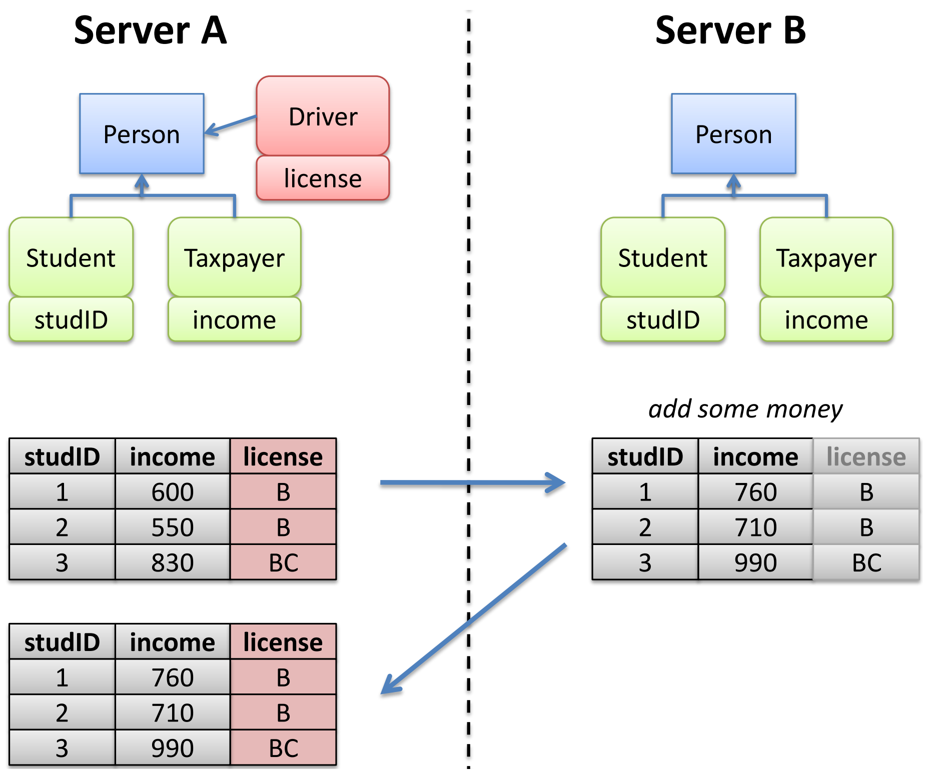
Figure 1.2: Schema Evolution with Reuse of Data.

on mobile devices, until the connection to the server is stable. Though mobile devices are less powerful than servers, they tend to get more and more powerful. In the past only imperative programming languages close to hardware were used to develop software for mobile devices. But the trend is to use object-oriented technologies for this purpose. Mobile phones are a prominent example. Sun offers a separate edition of the object-oriented programming language Java for mobile devices, called J2ME. Google developed an object-oriented framework and operating system for mobile phones, called Android[1]. Besides the power of mobile processors, the available memory on mobile devices grows, too. Nowadays mobile phones provide about half a gigabyte of internal memory and are extensible, to support 8 or even more gigabyte. Hence, there is the possibility to use database management systems on mobile devices. Notably, storing information temporarily on mobile devices leads to the well known problem of *offline database synchronization*. This thesis does not focus on this problem, but leaves it to future work. As mobile software is developed using object-oriented programming languages and databases can be used, the requirement for object-relational mappers on mobiles devices emerges. Those mappers need to consider the special properties, which are not present in stationary devices.

The same holds for classically distributed systems. Usually services are not provided by a single server, but multiple servers provide them and work together in order to achieve complex tasks. Each server needs to persist its data, as network connections cannot be considered stable by default. Loss of data leads to exhaustive costs in industrial settings. Imagine a system, responsible for financial transactions of a bank or a system, responsible for order processing of a trading company.

Vast amounts of servers of different companies providing lots and lots of

[1]http://www.android.com/



| studID | income | license |
|---|---|---|
| 1 | 600 | B |
| 2 | 550 | B |
| 3 | 830 | BC |

| studID | income | license |
|---|---|---|
| 1 | 760 | B |
| 2 | 710 | B |
| 3 | 990 | BC |

| studID | income | license |
|---|---|---|
| 1 | 760 | B |
| 2 | 710 | B |
| 3 | 990 | BC |

Figure 1.3: Example of Domain Object Distribution Between Heterogeneous Servers.

services currently exist. In order to utilize the offered functionality systems of different companies, developed using different technologies need to work together. The Open Service Process Platform[31] provides an environment, which enables developers to utilize and compose services. Each server needs its own persistency mechanism in order to store and thereby to avoid the loss of its data. Additionally servers need to exchange data, if they are working together on the same objective.

The transport of data between servers, which were developed independently from each other, poses the requirement of a translator between the servers. Each server (or cluster) has its own domain model, representing its understanding of the world. The translator needs to transform domain objects from the domain model of one server to domain objects, which conform to the domain model of the other server. Though DAMPF does not provide such a translation mechanism, it gives servers the ability to work with domain objects they do not fully understand. DAMPF is able to send and receive domain objects. Whenever a domain object is received, which is not compatible with the current servers understanding, only those parts of the domain object, which can be understood by the current server, will be used to create a domain object for it. Data, which is not understood, will not be deleted, but just not used by the current server. Thus, when the server sends back an incompatible, but anyway used, domain object to the sender, all data will be available again.

**Example 3** *Figure 1.3 depicts an example for such a scenario. The domain model of server A, depicted at the left top, describes persons with three roles:* `Student`*,* `Taxpayer` *and* `Driver`*. Below the model, there are 3 exemplary persons, having student id, income and driving license. Server B does not know about the* `Driver` *role, but about* `Student` *and* `Taxpayer`*. Its task is to add some money to the persons he receives and to send the data back, to where it came from. Is depicted in the figure, the driving license data is not lost due to the processing of Server B. Server B just uses the data it understands, but leaves the remaining untouched. Indeed Server B will not know that the received domain object does not fit to its domain model, because DAMPF hides these adaptations from the servers.*

The ability to process domain objects in a distributed environment, without the need, to explicitly adapt all objects, eases the development of such systems considerably.

**Context-Based Security.** An important aspect of domain objects for their transformation to their relational representatives is that they contain data. In combination with the ability to distribute domain objects, security requirements emerge. For example governmental systems stockpile vast amounts of highly confidential data. This data cannot simply be send from one device to another. Depending on the context, different subsets of the whole data should be accessible. Or in other words, parts of the data need to be hidden in some systems.

**Example 4** *A central governmental system contains data about all citizens of the country. Data from health insurance companies, tax offices and many more systems is collected. The central systems should also offer data to these other systems. It is important, that a tax office, requesting the data of some citizen, does not see any health-related information of that citizen. People do not want to expose their meticulous illnesses to somebody else, than their doctor. Some data is relevant to more than one company. The tax office and health insurance companies need to know the age, name and sex of a citizen.*

Hence, data is context-sensitive. Depending on the authority accessing the data, a different subset of it should be exposed. Different approaches exist, to realize this context-based data selection, but none is available as part of an object(-role)-relational mapper. Using the distribution feature of DAMPF along with role-based programming, it is possible to realize context-based security.

Domain objects, which are subject to many contexts, have a core comprising those attributes, shared by all contexts. All other attributes are modeled in roles. A central system contains the definition of all roles. The particular systems only contain the definitions of those roles, they are interested in. The distribution of a domain object from one system to another, as described in the preceding paragraph, leads to a transformation of the domain object, which can be understood by that system, but does not lead to loss of data. Because the receiving system does not know about roles, which are not relevant to it, the transformed domain object only exposes the data, which is of interest to that system, but hides all information, which must not be accessible.

**Example 5** *The central governmental system from Example 4 contains the class* `Person`*, with the attributes* `name`*,* `age` *and* `sex`*. It furthermore contains*

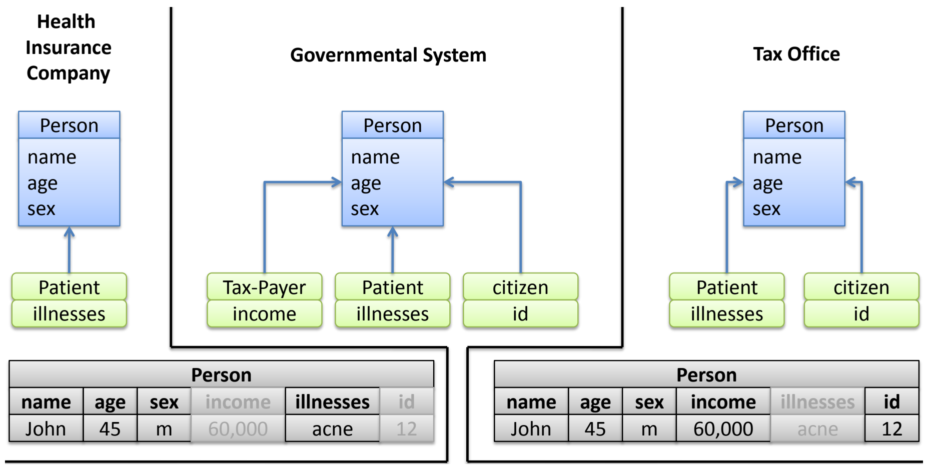


Figure 1.4: Context-Based Security as a Direct Consequence of Transformative Domain-Object Distribution.

*three roles, playable by persons:* ***`Citizen`***, ***`Patient`*** *and* ***`Tax-Payer`***. *A health insurance system contains only the class* ***`Person`*** *and the role* ***`Tax-Payer`***. *Tax office systems contain class* ***`Person`*** *and the roles* ***`Citizen`***, *as well as* ***`Tax-Payer`***. *The tax office requests the domain object of John Smith, a 45 years old male citizen with an income of $120,000 a year, who had acne as a child. A clerk thoroughly processes the tax assessment of John Smith, and thereby cannot see the illnesses of John, but the data of John as a tax payer and citizen. After the clerk finished his work, the domain object is send back to the central system. Now, the insurance company is requesting the domain object of John Smith, too. A clerk of that company does not see any information about John as a tax payer or citizen, but as a patient. Figure 1.4 depicts this scenario.*

In summary, the distribution feature of DAMPF allows to realize context-based security without any extra effort. Systems using DAMPF will always only see that information, which is in accord with their domain model. These systems will even not know that they only see part of the whole object, because DAMPF hides this information from them. In consequence, the distribution feature of DAMPF realizes context-sensitive data selection.

The thesis is structured as follows. Chapters 2 and 3 comprise background knowledge. Experienced readers can skip them. Chapter 2 provides an overview of the current state of the art of roles as an extension to the object-oriented paradigm. Chapter 3 explains fundamental concepts about the relational model and the transformation of domain objects into their relational representatives. The concepts of DAMPF are elaborated in Chapter 4. The actual implementation is subject of Chapter 5, which includes a description of the normalization approach, too. Finally, the thesis concludes with Chapter 6.

# Chapter 2

# Dynamic Collaborations with Roles

## 2.1 A Conceptual View on Roles

Object-Oriented systems consist of objects, which **collaborate** with each other. That is objects send messages to each other and react on received messages. Two different shapes of Object-Orientation exist: class-based and prototype-based approaches. Nowadays class-based approaches form the mainstream.

In class-based approaches objects are derived from classes. A class thus describes a set of objects, which are similar in their properties. Properties are both attributes and methods. The derivation of an object from a class is called *instantiation*. If attributes are designed mandatory, a value has to be assigned to them during instantiation. This is done in a special method, called the *constructor*, which is executed at instantiation time of the object. This way no object can exist, which has mandatory attributes without a value. The values of the attributes of an object form **the state** of the object.

Methods encapsulate behavior, viz. how an object reacts on a received message. The set of methods, defined in a class, can be seen as the set of valid messages for objects derived from that class.

In order to foster reuse and modular design, classes can be structured in **inheritance-hierarchies**. A class is able to refine another class, called its superclass, in that it overrides or extends methods of the superclass. The most powerful feature of class-based object-orientation is *polymorphism*, which enables the developer to assign objects of any subclass of some class X, to a variable typed with X. This enables an important principle called 'divide-and-conquer': Big problems are divided into small pieces, which are easier to solve, and the partial solutions are composed to an overall solution. For each category of subproblems a separate subclass can be written, containing a specialized algorithm, which fits best for the corresponding category.

Inheritance in class-based object-oriented system is static. That is, inheritance is defined between classes, not objects. If some class A inherits from another class B, this dependency holds for all instances of A. Polymorphism is only partially dynamic, as it is about substitutability of objects at runtime, but the possible substitutions are limited by the static hierarchy of types.

### 2.1.1 Motivating the Concept of Roles

Imagine a University Management System. Two important concepts in such a system are `Student` and teaching assistant, in short `TA`. Both refine the more abstract concept `Person`. In class-based object-oriented languages a common design for these concepts consists of one class per concept, where `Student` and `TA` inherit from `Person` as depicted in Figure 2.1.

Now imagine a person, who starts studying at a university and, after a year, starts to work as a teaching assistant in parallel. The proposed design is not able to express this situation, because an instance must not be of more than one type. A design supporting this scenario needs **multiple inheritance**, i.e. a class `StudentTA` needs to be introducing inheriting from both `Student` and `TA`.

This might look like a suitable solution on the first sight. The problem of this solution is, that it does not scale. Imagine further concepts like `StudentWorker`, `StudentRepresentative` and so on. A person could be a student, teaching assistant, student worker and student representative at the same time. Or

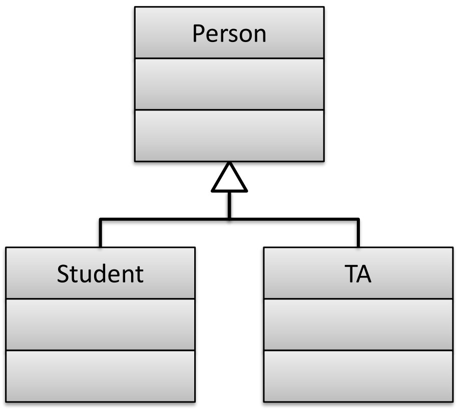


Figure 2.1: Using Inheritance to Model Students and Teaching Assistants (TA).

he could be a student and a student worker at the same time. In the worst case every possible combination the subclasses of class `Person` is needed as a separate subclass. This swiftly leads to lots of additional classes. Thus multiple inheritance is not a feasible solution.

So the solution using multiple inheritance is only suitable for a small number of classes? No! A further problem of using the inheritance solutions reveals by slightly changing the scenario. Imagine a student being a teaching assistant for two different courses, viz. being a teaching assistant twice. There is no clean solution to model this scenario using inheritance.

Furthermore using inheritance does not allow a student to be a teaching assistant only for some time. That is because the instance expressing the student, being a teaching assistant at the same time, is an instance of class `StudentTA`. If the student finished his job as teaching assistant, he cannot change its type at runtime to `Student`. A solution would be, to delete the instance of `StudentTA` and create a new instance of `Student`, but this is only a workaround on the level of implementation. On a conceptual level the instance representing the student must not be deleted, unless the student finished his studies or died.

As the previous examples showed, it is not possible to model the scenario using inheritance. Besides inheritance the object-oriented paradigm offers delegation. Delegation works on the level of objects and denotes, that an object $o_1$ referencing an object $o_2$ is able to call a method provided by $o_2$. A solution for the above described scenario using delegation is depicted in Figure 2.2.

This design is able to express the scenarios described above. For example

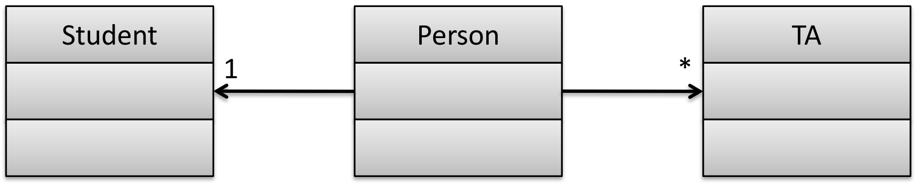


Figure 2.2: Using Delegation to Model Students and Teaching Assistants.

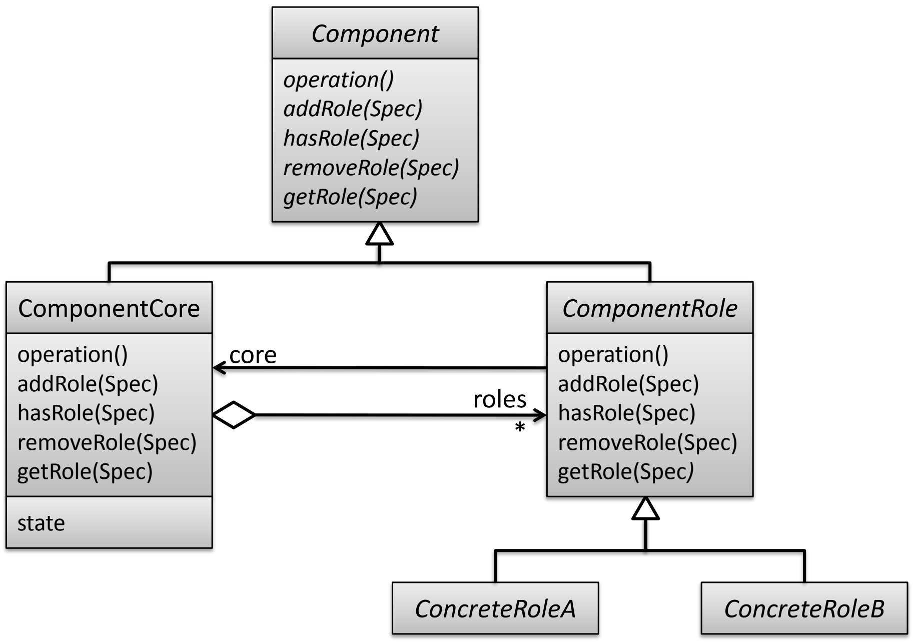


Figure 2.3: Role Object Pattern, redrawn after [6].

to model a student working as a teaching assistant twice, at the same time, an instance of class `Person` references and instance of class `Student` and two instances of class `TA`. It is also possible to express a student, being a teaching assistant only for some time. Therefore code in class `Person` is needed, which manages the references of persons. If a student stops being a teaching assistant, the reference to the corresponding instance of `TA` and the instance itself is simply deleted.

But how does this solution scale? Every further class, like `StudentWorker`, is introduced as a separate class and connected to class `Person`. For each new class management methods need to be introduced in class `Person`, offering the functionality to add and remove instances of the new class.

In 1997, Bäumer et al. introduced the Role Object Pattern[6], which is a delegation-based solution for exactly the problems described above. The pattern, as depicted in Figure 2.3, structures and refines the solution showed in Figure 2.2. First the management code is extracted to a separate subclass, which is named `ComponentCore`. Next a further subclass, called `ComponentRole`, of `Person` is introduced. This class serves as superclass of all subclasses like `Student` and `TA`. Due to the inheritance relations the management methods are present in all classes. Their actual implementation is defined in `ComponentCore`, the implementations in `ComponentRole` simply forward the calls to the corresponding core instance.

The pattern allows dynamically adding and removing classes like `TA` at runtime. Those classes can evolve independent from each other and the combinatorial explosion of classes through multiple inheritance is avoided. Nevertheless this solution does not scale. Bäumer et al. mention a set of drawbacks of their pattern. Clients using instances of classes like `Person` get complex very soon, because they need to be aware of all the subclasses and manage their instances (using the management methods). If there are constraints like: "a person, which is a student, must not be a professor at the same time.", these need to be expressed in client code, which thereby gets even more complex. The type system cannot be used to enforce these constraints. A final drawback of the pattern and of solutions using delegation is that the conceptual unity gets lost. Imagine the very simple scenario of a student. The student is express by two instances, one core object and one role object. Thus two instances are used to express a single conceptual entity, i.e. the student. This problem is called *object schizophrenia*[33] and can be seen as the reason, why a new concept needs to be added to the class-based object-oriented paradigm: **the role**.

### 2.1.2 The Concept of Roles at a Glance

As described in the former Subsection, the Object-orientated paradigm is missing an important concept. Object-orientation requires each instance to have a single type for its whole lifetime. A concept enabling instances to dynamically change their type at runtime is missing. That is a concept describing collaborations of objects. This drawback has already been formulated in 1987 by James Rumbaugh: "class-based object-oriented implementations of object collaborations hide the semantic information of collaborations but expose their implementation details"[58]. The concept of roles provides this feature.

Roles describe how the behavior and structure of an object differs in a particular collaboration.

Remind the scenarios of the last Subsection, viz. a person starting to study at a university and thus becoming a student. After some time that person starts to work as teaching assistant, but just for half a year. After 2 years of being a student, the person gets elected by other students and becomes a student representative. Finally, the person finishes to study and starts to work for a company as an employee. Note, that the scenario describes, what the person does, i.e. being a student, a teaching assistant and so on.

All these activities are roles. In real life they are described as roles, too. A person plays the role of an employee, a student representative and so on. It is inherent to the concept of roles that they are played just for some time. For example the teaching assistant role is being played just as long as exercises are offered in a semester. Thus roles can be **active** or **inactive**. It is furthermore possible, that a role is inactive for some time, for example the car driver role. This role is active, as long as the person drives a car, will be inactive when the person does not drive a car and will be active again, if the person starts to drive a car again.

Talking about playing the *same* role again, leads to the notion of role classes and roles instances. Like classes, defining a set of objects, which have the same properties, role classes define a set of roles instances, having the same properties. The role class `Student` defines student role instances, all having the same properties, like the attribute `matriculation number`. But how are role instances created? To answer this question first two more properties of the role concept need to be examined: rigidity and the property of being founded.

**Rigidity** is a property of types, denoting that a type is able to exist on its own. Instances of type `Book` for example are able to exist on their own. Instances of type `Reader` cannot, because if there is a reader, there always is a person, who `is` that reader. Types like `Reader` are called non-rigid types. Role types are non-rigid, too. Looking at the concept of roles, this non-rigidity excels in the requirement of a **player**. There must not be a role without a player. Remind the scenario from above. If there is a student, there is a person, being that student, too. This person is the player for role type `Student`.

The second important property of role types is **foundation**. Types in general are founded, if it does not make sense, to talk about them, without talking about other types. In essence types are founded, if their instances always take part in a collaboration. Remind the example from the last paragraph. The type `Reader` is founded, because it does not make sense to talk about readers, without talking about what they read, i.e. books. Readers always take part in a collaboration, viz. a reader collaborates with books. Thus founded types cannot exist alone, but *depend* on other types. This dependency is a constraint over the instances, saying that there must not be an instance of this type, which is not connected to another instance. Multiple qualities of foundation exist, that is various types of these constraints can be identified. A founded type may require its instances to collaborate with exactly one, at least one, at most N, between N and M or exactly M other instances. A further restriction could be that the instances have to be of different types. Up to now these qualities have not been investigated in detail. With regard to role types, which in general are founded, i.e. cannot exist on their own, the property foundation excels as the requirement of roles to be part of a **context**. For example in the scenario of the last Subsection the role type `Student` is founded, because it does not make sense to talk about students, without talking about what they do. Students interplay with other students, teaching assistants, professors and so on. The context of this collaboration is the university.

In general a context describes a collaboration, i.e. the interplay of a set of instances, and a boundary for it. The university from the former example defines a boundary for collaborating students, professors and so on. The role type `Employee` has another context, the company employing the person. Imagine a person called Mary, studying at the University of Technology Dresden and, to earn money, working for two local companies. This scenario is depicted in Figure 2.4. The example clarifies the need for context types and context instances. In analogy to classes and objects or role types and role instances, context types define a set of context instances, having the same properties in common. The context type `University` describes context instances like `tud`[1] and `htwk`[2]. These context instances have properties, like the attribute `name` or the method `enroll()`, in common.

Thus roles are bound to exactly one player and one context instance at a time. That is a role instance cannot be played by another player and is not able to be part of another context instance at the same time. This leads to the different states of role instances and the answer to the question, how role instances are created.

The possible states of role instances are depicted in Figure 2.5, which can be read like a state chart. All states are relative to a given context. The state `null` means, that the role does not exist at all in the given context. If it exists, the first possible state is called `unbound`, meaning, that the role instance is not

[1]tud = University of Technology Dresden

[2]htwk = College for Techniques, Economics and Culture Dresden

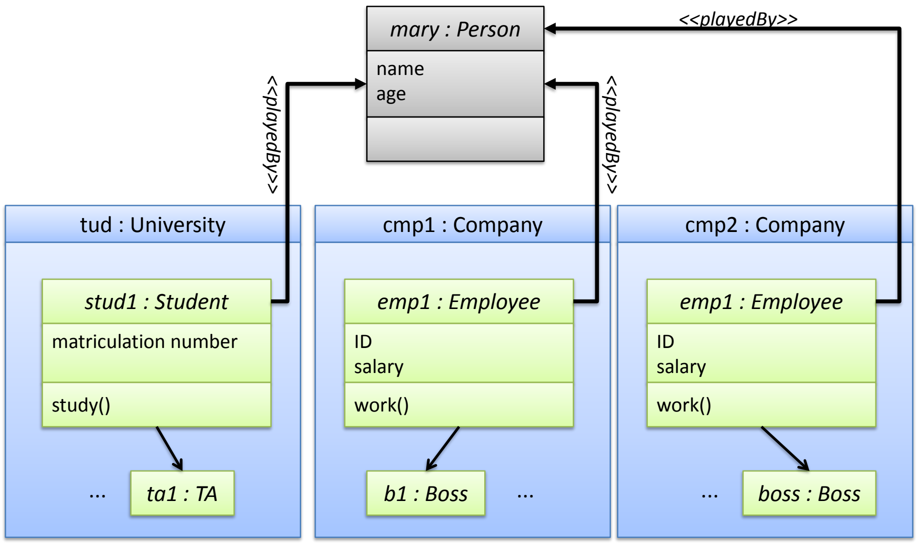


Figure 2.4: Example of Mary, Playing Several Roles.

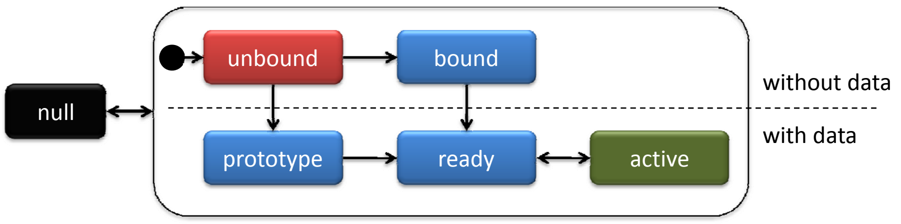


Figure 2.5: Possible States of Roles, Relative to a Given Context.

bound to a player and has no data. If the role gets bound to a player it reaches state `bound`. If it gets filled with data, it reaches state `prototype`. The name typifies the use of this state. Roles in prototype state are used such, meaning that they are likely to be cloned and these clones are bound to various players. If a prototype role is bound to a player or a bound role is filled with data, the `ready` state is reached. This state typifies a role, which is *ready* to be actively played. If such a role is activated, it gets into state `active`.

With regard to Figure 2.5 a role instance is created by binding it to a given context. It does not have to be filled with data or to be bound to a player. From a conceptual point of view the first three states, i.e. `null`, `unbound` and `prototype`, disqualify the instance to be a role. As long as the instance is not bound to a player it is conceptually not a role instance. But why do we need the three contradicting states? The prototype state fosters reuse on instance level. That is, role instances are easier to create based on the state of already existing roles. Additionally the prototype state is a conceptual necessity. Imagine a garden plot union and their executive. One of the residents is elected to be the executive. What happens to the executive role, if the person, being that executive, migrates or dies? It is unbound, as it does not have a player anymore, but it is filled with data, thus it is in prototype state. The residents will elect a new executive and in the meantime will talk about that role, making the assumption, that there *will be* a player for it. Thus a role is allowed to be unbound, as long as there will be a player for it in the future. This qualifies the existence of the unbound state, too.

As already mentioned, roles require a player and have, like their player, an own state. That is, a role class is described in a similar way like classes, meaning that they describe a set of attributes and a set of methods, too. But role classes contain a further construct, that of *delta methods*. It is the nature of roles, to describe how some player acts or reacts in the context the role is bound to. This includes, that a role describes how the behavior of the player changes, if he acts or reacts in that context. Imagine a married soldier. If his wife asks him for a coffee, he will answer "Sure, honey." and after a minute go and get the coffee for her. If his direct superior asks him for a coffee, he should behave in another way, if he likes his job. The person behind the soldier or husband may be even lazy and would naturally not react on such a question at all. The roles soldier and husband define how the behavior of the person changes in the context of the army or family.

But roles do not just change the behavior of their players in a given context. They change the way players look like, that is there public structure, too. In a collaboration objects communicate with each other. How they do so, is described by their roles. Thus, what they see of each other is described by their roles, too. And what they see could be something completely different, from what the object originally looked like. Remind the married soldier. In the army he looks like bad medicine, always running at the very front line. For his wife he looks like the nice guy, she fell in love with, or the guy, who always forgets about washing the dishes. Thus, in different contexts a different set of properties is visible and the properties of the player can be hidden in some contexts.

This leads to the question of accessibility, viz. who is able to see which roles. As mentioned, roles are bound to a context instance and a player. In principle a role can only be used by other roles, which are bound to the same context. This is because the context defines the collaboration. If the role needs to be used by another role, which is bound to another context instance, the collaboration is incomplete. The essence of a collaboration is to describe *all* interactions of the role instances it contains. But there are scenarios in which role instances need to communicate across the boundaries of their contexts. Remind the married soldier. As a soldier he earns money and as a citizen he has to pay taxes. Thus the taxman is highly interested in details about the soldiers pay and, due to the rules passed by the government, the fact, that he is married. Thus there is a collaboration between the soldier role, bound to the army context, the husband role, bound to the family context and the tax collector, bound to the taxman context. It is important to note, that this collaboration is of temporal nature. Unlike the family context, which sustains at least until the husband dies, this collaboration exists just for a short, delimited time. Of course the taxman wants to collect the taxes every year, but this leads to a new temporal collaboration each time. The role instances participating in the collaboration may change each time. Just imagine that the soldier gets father and hence, stops his carrier as a soldier, to earn his money in a safer way. Due to the possibility of varying participants, such temporal collaborations have no fixed boundary.

In order to describe such temporal collaborations a new concept is needed: **transclusion**. Transclusion is a property of compositions. Usually all parts of a composition are consistent in themselves. If they are composed in a safe way, the result is consistent, too. If a part is decomposed into smaller parts, the resulting fragments are not necessarily consistent. Even a safe composition mechanism cannot ensure that a composition of such fragments will be consistent. Transclusion denotes that a composition of inconsistent fragments leads to a consistent result. In terms of temporal collaborations this means that inconsistent fragments of contexts can be joined to a new collaboration, which in turn is consistent. In the taxman example the soldier role is ripped out of the army context, leading to an inconsistent state of itself. The role may require a direct superior, which in the new temporal collaboration does not exist. A transclusive composition of partial contexts changes the external dependencies of roles in order to construct a consistent temporal collaboration. In essence, such a composition consists of 3 steps: **select, project, join**. First parts of contexts are *selected*. The individual roles needed for the temporal collaboration are *projected*. Finally the projections are *joined* and now form a consistent composition.

In summary, roles are instances of non-rigid, founded types. The property of being founded excels in their need for a context. Their property of being non-rigid excels in the requirement of a player or, in other words, the fact, that roles do not have an own identity. Roles are merged with their players and form a conceptual unit, which is, what delegation-based approaches cannot achieve. Roles enable their players to change their type at runtime, which is what inheritance-based solutions cannot achieve.

### 2.1.3 History and State of the Art of Conceptual Roles

Roles are a very new concept, since a long time. Many researchers developed their own understanding of the role concept. Some, like Friedrich Steimann, tried to unify these insights[63]. Others combined preliminary definitions into an own classification, like Graversen[29]. But, up to now, no common sense is achieved. In the following, the most important viewpoints on the role concept are presented.

#### T.A. Halpin - Object Role Modeling

In 1989 Terence Aidan Halpin introduced, what is known today as Object-Role Modeling[32]. The goal of the approach is, to overcome limitations of object-oriented (OO) and entity-relationship (ER) modeling. In Europe Object-Role Modeling (ORM) is often called Natural language Information Analysis Method (NIAM). The approach focuses on data modeling, because it stems from the domain of information system design.

The key difference to OO (data) modeling is that not just objects are modeled, but the roles they play, too. Halpin sees objects as entities or values, where entities are those parts of a system, representing and structuring data. In ER modeling an entity has a set of attributes. In contrast, attributes are not explicitly used in ORM. Instead relationships are preferred, which relate entities to each other. Imagine yourself, reading this master thesis. To model such a scenario to entities are needed. The first is you, the second the thesis. In OO and ER modeling an attribute would be used, to model that you read this thesis. In ORM/NIAM a relationship *reads* is used, i.e. `Person` *reads* `Thesis`.

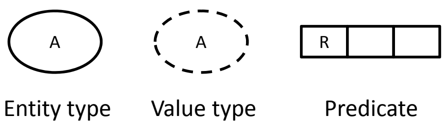


Figure 2.6: Basic Graphical Elements of ORM/NIAM.

The key benefit of using relationships instead of attributes is that both sides of the relationship are able to evolve independently. Imagine an entity `Person`, working for a company. In a first design only the name of the company is taken into consideration, thus it is modeled as an attribute of `Person`. Later it is decided, that the company's turnaround and the total number of employees have to be modeled, too. Thus the attribute needs to evolve to an entity, containing name, turnaround and total number of employees, and a relationship between `Person` and `Company` is introduced. In ORM/NIAM the entity `Company` and the relationship *works for* already exist. To apply the new requirements only the entity `Company` needs to be adjusted - independently from `Person`.

The second important concept in ORM/NIAM is, besides objects, the role. Halpin sees roles as parts of relationships. Remind the two example relations from the last paragraph: `Person` *reads* `Thesis` and `Person` *works for* `Company`. Both relations are binary, viz. they have two ends. They define, that two tuples (or instances in terms of OO) relate to each other. The *reads* relationship defines, that persons read theses. Halpin sees the ends of relationships as slots for players in the relationship. You, for example, play the role of the *reader* and this copy of the thesis plays the role of *being read*. Relationships are not limited to be binary. They can be n-ary in general. A relationship it the sense of Halpin defines a context, almost like introduced in 2.1.2. That is, it defines a collaboration and a boundary. The difference is that the boundary is not defined as a separate entity. Imagine the university context. A possible relationship defining such a context could be `Person` *studies* and *participates in* `Course` *held by* `Professor`. This relationship is complex, viz. it is composed out of simpler relationships. The simple relationship *studies* is unary, denoting that some person is a Student. The other two relationships denote that students take part in courses, which are held by professors. But the boundary, that is the university itself, is missing. To overcome this limitation each relationship needs to be extended by a further slot, binding the collaboration to an explicit boundary. It is interesting to note, that this leads to the aspect, that the boundary of the context is a role and itself participates in the collaboration it delimits.

ORM/NIAM is a modeling method for system analysis. At this stage of development (and in general) communication between all participants of the project is very important. Graphical abstractions help to communicate complex issues. Therefore ORM/NIAM provides a graphical notation. The basic elements are entity types, which are drawn as named ellipses, value types, drawn as named, dotted ellipses and predicates, drawn as a set of boxes. Figure 2.6 depicts these basic elements.

The term predicate in the graphical notion is just an interpretation of relationships. Predicates, like those from predicate logic, relate facts to each other. Halpins relationships do the same, hence they are called predicates. Remind the relationship *works for*. In predicate logic the same would be expressed as the predicate `worksFor/2`. A predicate consists of several roles, where each role is depicted by a separate box. Each role has to be bound to a player, which is of an entity or value type, using a solid line. Figure 2.7 shows the graphical representation of the two former examples, viz. the *works for* and *reads* relationships. The example shows one of the biggest benefits of ORM/NIAM, its closeness to natural language. That is, what the N in NIAM is for. The figure reads as follows: a person *has* a name, a person works for a company, a company employs a person and a person reads a thesis. As can be seen, this is very near to plain text, but enriched with metadata for each word.

A lot more elements exist. For example a noticeable set of constraints. To better understand them, a closer look at predicates is necessary. As already mentioned, the terms relationship and predicate are used interchangeably. It is a well known principle from database design to realize 1:N relationships like entities as separate relations, viz. tables. In terms of OO such relationships are realized as classes. Instances of such *relationship classes* describe collaborations. Halpins constraints on and between relationships require the understanding of them as relations.

Furthermore two important concepts of the relational model are needed: primary and foreign key. A relation consists of multiple attributes, whereof a subset is called the primary key. Attributes, belonging to the primary key, are unique for all tuples, i.e. instances in terms of OO. That is, if there is a tuple, having some value $v_1$ to $v_k$ for the attributes of the primary key, there cannot be another tuple having any of the values $v_1$ to $v_k$ for the same attributes. This uniqueness allows using the primary key to reference the respective tuple. The origins of such references are again attributes. The relation containing such an attribute defines a foreign key, which describes origin and target of a reference.

An internal uniqueness constraint defines for a relationship, which attributes or roles, in the sense of Halpin, are unique for the whole relationship. It is depicted by an arrow tipped bar. Imagine a relationship `Person` *lives in* `Country`. If an internal uniqueness constraint is put on the first slot of this relationship, the requirement that a person lives in at most one country is modeled. This is because there can be only one tuple or instance of this relationship for the same person. It is possible to apply multiple internal uniqueness constraints to a single relationship. At most one of these can be marked primary, meaning that the slots comprised by this constraint form the primary key of the relationship.

Uniqueness constraints can also be applied between two relationships. Those constraints are termed external and base on the join of the relationships. They

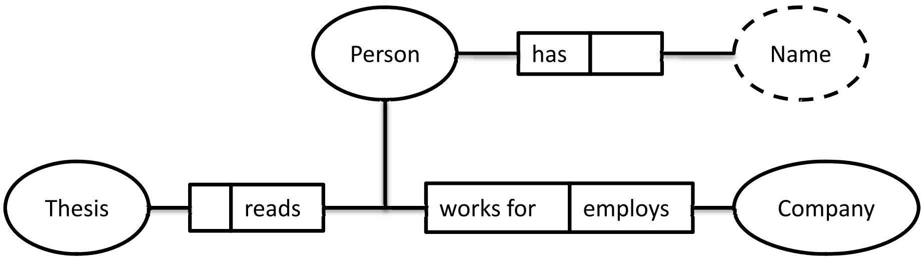


Figure 2.7: Example Application of Basic ORM/NIAM Elements.

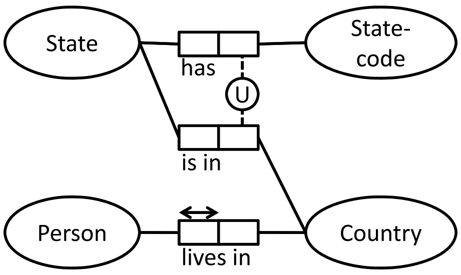


Figure 2.8: Example for Uniqueness Constraints of ORM/NIAM.

are depicted by a circled "U", connected with dotted lines to the corresponding slots. Again relationships need to be understood as relations, which can be joined, based on their primary and foreign keys. An external uniqueness constraint between slots $s_1$ to $s_k$ of relationship A and slots $s_m$ to $s_n$ of relationship B defines, that in the join of A $\bowtie$ B all these slots are unique. Imagine two relationships: `State` *has* `Statecode` and `State` *is in* `Country` and an external uniqueness constraint between `Statecode` and `Country`. The consequence is that each combination of statecode and country relates to at most one state. Or in other words, each state has at most one statecode per country. Again it is possible to apply more than one such constraint and to allow for further composition (joins) of relationships, one of them can be marked primary. Thus, in the example above, a state could be references by its statecode and country, if the external uniqueness constraint is marked primary. Figure 2.8 depicts the example of this and the last paragraph.

The uniqueness constraints allowed defining, how relationships are built up and composed. Halpin even goes a step further and allows reifying, viz. objectifying, relationships to entities. Using the feature of reification, relationships, viz. collaborations, are able to play roles, too. Furthermore roles can be defined mandatory. This constraint is depicted as a black dot on the line between an entity and a slot of a relationship at the end of the entity. The meaning of *mandatory* is that no entity is allowed to exist, without playing this role. For entities of value type this enables to model mandatory attributes. Entities of object type are forced by this constraint to participate in collaborations. Students for example have to study at a university, in order to be students. For complex scenarios disjunctive mandatory roles can be defined. They are depicted as circled black dots on a line, connecting two or more roles. There meaning is that players of these roles have to play at least one of them. Imagine students, which are allowed to choose between an oral exam and a written exam. This can be modeled with two relationships: `Student` *does oral exam in* `Subject` and `Student` *does written exam in* `Subject`, where the Subject-slots are connected by a disjunctive mandatory role constraint. Figure 2.9 depicts the examples of this paragraph.

Another important constraint of ORM/NIAM is the subtype constraint, which is lent from object-orientation. For example, the object type `Woman` is a (proper) subtype of `Person`. Subtyping is depicted by a solid arrow. In ORM/NIAM subtyping is applied to entities only. There is no subtyping for relationships or roles.

Furthermore ORM/NIAM provides set comparison constraints, which are applied between relationships, which have the same host object types. Imagine persons, which work for companies, but also buy goods from that company. Two distinct relationships model this scenario: `Person` *works for* `Company` *as* `Employee` and `Person` *buys* `Product` *from* `Company`. Set comparison constraints are allowed to be defined between the roles of `Person` and `Company`, but no for `Product` or `Employee`. For the example above a *subset constraint* can be applied between the Person roles, from the *...works at..as..* to the *...buys..at..* relationship. This models the set of persons working for companies as a subset of persons buying products at companies. The subset constraint is depicted by a dotted arrow, where the source is defined as the subset. If the arrow has heads on both ends, it depicts an equality constraint. That is, both sets are equal. Imagine a further relationship `Person` *uses* `Product`. Each person, who bought a product, does also use it. Finally Halpin defined an *exclusion constraint.* It is depicted as a crossed circle on a dotted line between two roles. It defines, that in any case either the first role or the second role is allowed to be played, but never both together. A further relationship in the example above is a unary one: `Person` *is tenured.* An exclusion constraint can be applied between the *is tenured* and *works as..for..* relationship, meaning, that a person is either tenured or works for a company. Figure 2.10 depicts the examples of this paragraph. Note the similarities to Riehles constraints. Halpins *equivalence constraint* indeed is Riehles *role-equivalence.* Halpins *exclusion constraint* is Riehles *role-prohibition.* Finally Halpins *subset constraint* is Riehles *role-implication.*

Halpin defined a lot more constraints: a set of frequency constraints, and a set of ring constraints. The former are applied to sequences of roles, like set comparison constraints, and define how often each role has to be played. Ring constraints are applied to pairs of roles played by the same host type and qualify the playing relation as reflexive, transitive and so on.

ORM/NIAM has been used for a very long time. Since its initial publication many extensions have been developed. Microsoft supports the approach, leading to high-quality ORM/NIAM tools. In consequence, unfortunately, these tools are commercial. Nevertheless ORM/NIAM provides a powerful construction kit for complex role models.

### First-order Relationships

Besides work on roles as first-class citizens of programming languages, there is a community focusing on relationships, viz. collaborations as first-class citizens.

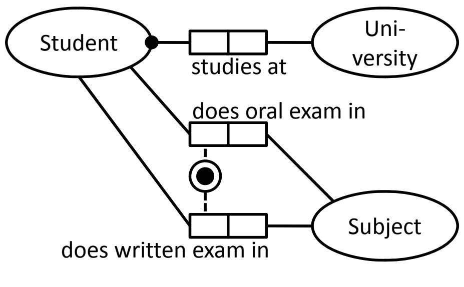


Figure 2.9: Example of Mandatory Role Constraints.

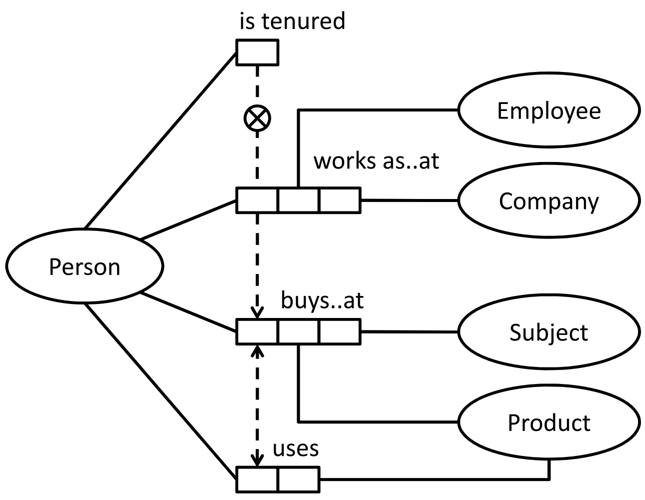


Figure 2.10: Example for Set Constraints of ORM/NIAM.

**Bierman, Wren and Balzer. RelJ Extended by Member Interposition and Relationship Invariants.** Bierman and Wren[9] and Stephanie Balzer et al.[5] examined, among further researchers, relationships in object-oriented programming. Like Halpin, they understand relationships as collaborations of objects and argue that object-oriented languages are unable to express them directly. Hence, they call for relationships as a first-class programming language construct, like classes, for collaborations. The argument is that first-order relationships allow to describe collaborations declaratively. Without such a construct, collaborations are described imperatively and are thus scattered. Furthermore first-order relationships explicitly define a scope, viz. a context in our terminology, for collaborations. Plain object-oriented programs need to be thoroughly analyzed, in order to find the scope of the imperatively described collaborations. Finally, first-order relationships allow defining constraints on collaborations, which are distributed over the participants in object-oriented programs and thus are likely to get inconsistent, if one of the participants evolves. For these reasons, Bierman and Wren[9] introduced a new language, called RelJ, providing first-order relationships. Stephanie Balzer et al.[5] provided two further, important concepts: member interposition and relationship invariants.

Member interposition deals with the problem that some properties of objects only apply, if they participate in a collaboration. In our terminology, roles provide properties, which thereby are only available inside the context instance they are bound to. Member interposition denotes that properties, defined in the relationship, are woven into the participants at runtime. The need for member interposition arises, because no explicit notion of roles exists. Balzer distinguishes between interposed and non-interposed members of relationships. A non-interposed member of a relationship is a member, which is bound to the relationship itself. In our terminology, contexts have properties, too. Imagine a relationship `Person` *works for* `Company`. The relationship has two members: `salary` and `employeeDiscount`. The salary is an interposed member, as it is tied to each person participating in this relationship. The discount for employees is a non-interposed member, because it is tied to the relationship itself. Interposed members are also referred to as *participant-level* members, whereas non-interposed members are referred to as *relationship-level* members. In comparison, ORM does not provide means to define participant-level members.

Relationship invariants can be seen as consistency constraints. Stephanie Balzer defines four types of invariants: intra-relationship, inter-relationship, value-based and structural invariants. The first two kinds define the scope in that they are valid. Intra-relationship invariants are valid for a single relationship. This conforms to Halpins internal constraints. Inter-relationship invariants have a scope of multiple relationships, conforming to Halpins external constraints. The last two kinds differ in whether invariants use values for their definition or are purely based on structure. The first and last two kinds form two orthogonal dimensions. Structural intra-relationship invariants are able to express multiplicity constraints. Remind the example of Figure 2.9. A student has to study at a university in order to be a student. This mandatory role constraint of ORM/NIAM is a structural intra-relationship constraint of Balzer. Like ORM/NIAM allows defining multiplicities for such constraints, Balzer does, too. The second part of the example in Figure 2.9, that is students may choose between oral or written exams, is represented by Balzers structural inter-relationship invariants, because it defines a constraint over two relationships. Again Balzers definition is as powerful as Halpins, though Balzers notation is more complex. So is ORM/NIAM in regard to constraints as powerful as Balzers extensions to RelJ? No, it is not, because of the value-based invariants. They use values of the participant's members to define constraints. Imagine the participant-level member `semester` of a relationship *assists* between `Student` and `Course`. The relationship may require the student to be in its third semester, in order to assist a course. This cannot be expressed in ORM/NIAM, but with a value-based intra-relationship invariant. Finally value-based inter-relationship invariants even allow composing relationships, based on the values of themselves or their participants. Thus Balzers extensions to RelJ provide the most powerful mechanisms to define constraints on collaborations.

**The Difference between Tuples and Relationships.** Stephen Nelson et al.[47] provide a further viewpoint on first-order relationships. They decompose systems into three levels[48]. The lowest level comprises plain objects, as known from the object-oriented paradigm. The second level defines tuples, viz. linked objects. Noticeably these tuples have state. The third and topmost level defines relationships as sets of tuples, which are allowed to be instantiated multiple times. In comparison to Balzers extensions to RelJ, the state of tuples represents interposed members. Non-interposed members are not part of Nelsons model. Nelson claims, that RelJ does not allow to instantiate relations multiple times. [49].

### Roles in Modeling

The large modeling community also investigated the role concept. In the following three important approaches will be presented: Smolanders GOPPR metametamodel, Odells powertypes in the UML and the Object-Oriented role analysis and modeling approach by Reenskaugs and Anderson.

**A Metametamodel Supporting Roles - GOPRR.** In 1991 Smolander published the **OPRR** metametamodel [61]. The OPRR metametamodel has four concepts: Objects, Properties, Relationships and Roles. As OPRR is targeting Information Systems (IS) Design, that is data modeling, it does not contain any means for behavior. Following Smolander an information system comprises objects, having properties, relationships and roles, both having properties, too.

The terms **object** and **property** are used like in object-orientation. That is a property is of a type and has a value. **Relationships** connect objects with each other. Moreover relationships describe more than which objects are connected to which other objects - they describe how they are connected. Take a person driving a car as an example. The person and the car are objects, having attributes like `name`, `manufacturer` and so on. They are connected in many ways. One connection is the car ownership. This ownership is modeled as a relationship between person and car. But the relationship describes more, than just the fact, the person and car are connected. It contains information, like the date of purchase and number of accidents. This is why relationships in OPRR have properties, too. They are handled as first class citizens, because they are a standalone concept of the metametamodel. Hence, relationships in OPRR are what the community of first-class relationships aims for.

Besides objects, properties and relationships, OPRR also contains **roles** as first-class citizens. They are described as *the ends of relationships*. Remind the example of the last paragraph. The car-ownership relationship has two roles: that of the car-owner, which is bound to the person object and that of the bought-car, which is bound to the car. OPRR has only binary relationships, viz. those having two ends. Thus each relationship contains, besides properties, two roles. These roles are played by the objects, which are connected to each other via the relationship. Interestingly roles have properties, too. This empowers the designer to put the information exactly where it belongs. Remind the properties *date of purchase* and *number of accidents*. The first property correctly belongs to the relationship. But the second one belongs to the car playing the role of a bought-car. Imagine a further property *owns car since*. This property belongs to persons playing the role of a car-owner.

These four concepts form a powerful metametamodel, enabling method engineers to build very expressive metamodels for their methods. But, nevertheless, OPRR was not feasible for complex methods. This changed in 1993, when Smolander introduced GOPRR. [62]. **GOPRR** is an extension to OPRR, adding a fifth concept: the **graph**. Graphs can be seen as collections of objects, relationships and roles. Hence it was possible to build modular metamodels for complex methods. Furthermore GOPRR allows for n-ary relationships, that is *role models*.

In 1995 Steven Kelly[38] pointed out, that the relationship concept in GOPRR needs to be split into two. This is because a relationship comprises two distinct aspects. The first aspect describes the connection of objects, viz. which object is connected with which other object. Kelly calls these connections **bindings**. The second aspect of a relationship is that it contains properties. Kelly claims that binding and property aspects need to be separated. The property aspect remains in the relationship concept, but the binding aspect is lifted to the graph concept. That is graphs define which objects are bound to which other objects via which relationship. Relationships themselves just describe their data, viz. their properties. The benefit of lifting the binding aspect to the graph is that it is now possible to globally define how the comprised objects collaborate. Graphs hence represent objects, roles, relationships and how each of them is connected. Kelly differentiates between type-level and instance-level bindings. A type-level binding defines, which role types can be played by which object types (that is classes). Instance-level bindings are concrete collaborations, viz. objects playing role-instances in a concrete relationship instance.

The clue of defining each kind of binding in graphs is that it is possible to define global constraints on the collaborations. GOPRR supports two kinds of constraints: multiplicities for roles and connectivity constraints for objects. The first type of constraint allows limiting the number of role instances in a relationship. Remind the car-ownership relation. It is usual, that only one person plays the role of a car-owner for the same car at the same time. This can be modeled by a multiplicity of 1 for the car-owner role. The second type of constraint allows defining inter-relationship constraints. They are limited to numeric constraints (i.e. multiplicities) and are bound to objects, instead of roles. Imagine a person being president of a country. There must not be more than one person at a time, playing this role. Using connectivity constraints this is easy to model.

The GOPRR metametamodel is near to the first-class relationships community, as it has relationships as first-class relationships. Nevertheless its abilities in expressing constraints on roles are less powerful than those of Stephanie Balzers extension to RelJ. ORM/NIAM is shown to be an instance of GOPRR[38]. Notably GOPRR aims at data modeling and does not contain any concept to model behavior.

**Modeling Roles in The Unified Modeling Language Using Powertypes.** The Unified Modeling Language (UML)[51] is a collection of many types of models. It provides a powerful metametamodel and lots of metamodels for all its models. A well known type of model is the class diagram. In a class diagram a set of classes can be modeled, each having attributes and methods. These classes can be related to each other in different ways. In the context of this thesis, only associations between classes are of interest. The ends of these associations are, like in GOPPR, called roles, too. Associations can be seen like relationships in GOPPR. The key difference is that relations do not have properties, or attributes in terms of the UML. Though, it is possible to define so called association classes, these are - as their name implies - classes and not relationships. Roles as ends of associations do not have attributes, too. In contrast to associations and association classes, there is no concept to enrich roles with attributes. Though the UML separates the binding aspect (association) from the property aspect (association classes), it is much less powerful than GOPPR. The reason is that the binding aspect stays in the middle and is not lifted up to the model itself. It is hence more complicated to define global constraints. The Object Constraint Language (OCL)[50] provides means to define such constraints. Nevertheless, they are based on objects and not on the roles, these objects play. Indeed roles are used for navigational means only. Roles in the UML cannot be used for much more than that, because they do not have attributes or methods - they just have a name.

Notably the UML offers another concept, which is nearer to our understanding of roles, than the concept of roles as defined by the UML. Powertypes[52] provide a further classification technique, besides inheritance. Imagine the class `Person`. Typical subclasses are `Man` and `Woman`. These two subclasses form a partition, because they specialize `Person` in regard to the gender. Though, it is possible to define further subclasses, like for example `Child`, `Adult` and `Senior`, the problem is that subclasses of a different facet of the superclass are mixed with each other. The problem can be solved elegantly with powertypes. Imagine a class `PersonKind`, which is a powertype of `Person`. Instances of this class are for example man, woman, child, senior and so on. That is instances of this class are objects, which can be seen as classes for instances of Person. Due to their object *and* class nature, instances of powertypes are also called **clabjects**.

But how are powertypes related to roles? The key similarity is dynamic typing. Roles can be played for some time, viz. they are bound to a player for some time and hence change the type of the player for some time. Powertypes can be created are runtime and form an additional class for instances. Imagine the role `Student`, bound to class `Person`. It is possible to define a powertype `PersonKind` and create a student-instance of it. Instances of class `Person` can be bound to the student instance of the powertype, which is nothing less, than that they start to play the student-role. The key difference between powertypes and roles is that instances of powertypes do not have attributes. Powertypes have attributes, of course. But these attributes belong to the clabject. It is not possible to define clabjects, which modify their instances.

**Object-Oriented Role Analysis and Modeling - The OOram Software Engineering Method.** In 1995 Trygve Reenskaug publishes his book about the OOram Software Engineering Method[56]. The book describes a methodology for the whole life cycle of software. Reenskaug's understanding of roles emerged in joint work with Egil P. Anderson, one of his PhD-students. Andersons PhD-thesis[1] contains a detailed description of the understanding of roles as used in OOram.

Their main motivation to introduce roles stems from problems to model activities. An activity describes a set of objects, which interact with each other to achieve a given objective. Activities, when modeled using classes, crosscut each other in the classes. That is multiple activities utilizing objects of the same classes are tangled in these classes and scattered across them.

**Example** Imagine a company selling products. The company employs salesmen, who sell products to customers. A salesman has to report to his superior. Figure 2.11 depicts a model, showing the classes and activities of this scenario.

Anderson states, that this crosscutting of activities raises the complexity of modeling to a degree, unacceptable for large-scale models. The proposed solution is to use composable role models. A *role model* describes an activity using *roles*, which are defined as named elements comprising a set of *role paths*. The connection between roles is established using these role paths, which point to another role and define a set of methods, which can be sent to this role. That is, roles define an interface. Notably Andersen's roles do not have state. He claims that information modeling should be strictly separated from interaction modeling. Figure 2.12 depicts the example described above using role models. Circles or ellipses denote roles, lines denote role paths and a black dot at an end of a line depicts that methods are send to the role, the dot is attached to,

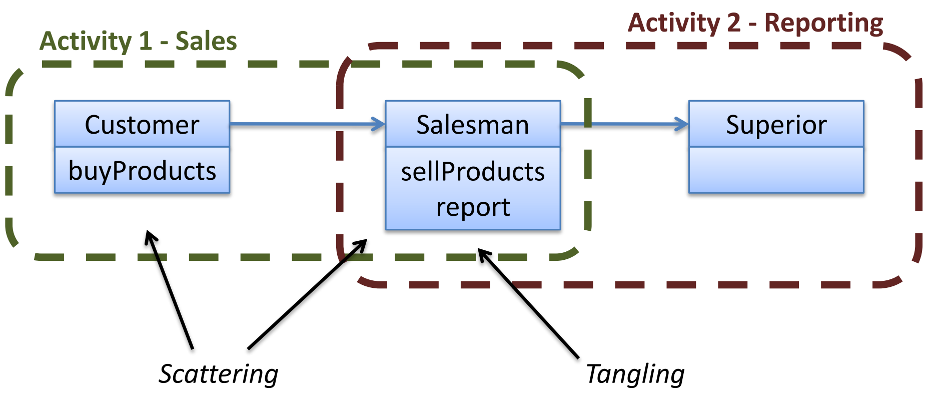


Figure 2.11: Example of Scattering and Tangling Activities in Classes.

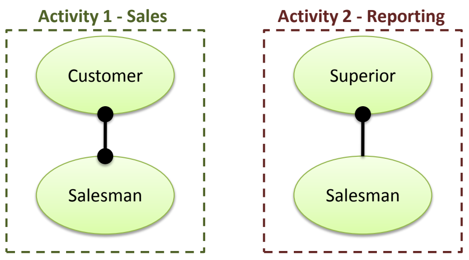


Figure 2.12: Example of Activities Modeled using Role Models.

using the corresponding the role path.

The main objective of Anderson's PhD-thesis is to find methods to handle the complexity of large-scale models. But role models swiftly get very complex. As solution Anderson provides operators to compose role models from smaller role models. He defines composition using the role paths. Figure 2.13 depicts the composition of the role models shown in Figure 2.12.

Anderson's understanding of roles is that they are played by objects. He is able to express, that an object plays multiple roles, even of the same type.

**Example** Remind the last example of customers, salesmen and superiors. John is a salesman, selling products to Mary and Peter. Mary is a salesman, too,

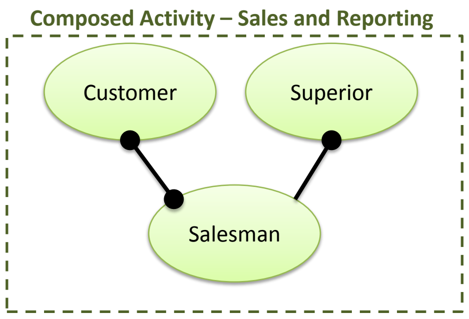


Figure 2.13: Composition of Role Models Shown in Figure 2.12.

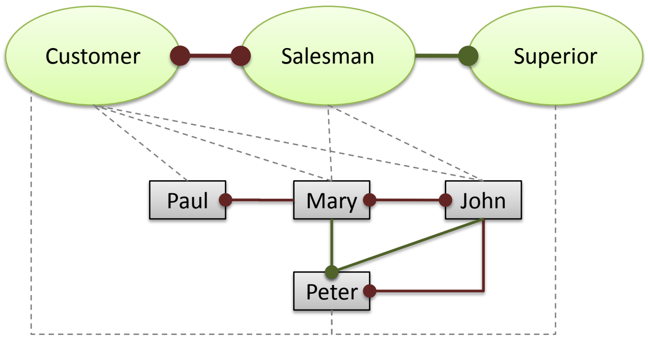


Figure 2.14: Runtime Model of Objects Playing Multiple Roles Simultaneously.

selling products to John and Paul. Hence Mary and John are both, salesman and customers. Both report to Peter, who is their superior. As peter buys products from John, he is a customer, too. Figure 2.14 depicts this scenario. On the top there is the composed role model. Below objects are shown in rectangles, which are connected to the corresponding roles they play.

In order to support the dynamic characteristics of roles, Anderson investigated Labeled Transition Systems (LTS) and developed statespaces to overcome their limitations. He shows in his dissertation, that both can be transformed into each other without loss of information.

Statespaces consist of states and transitions, like most types of state charts. Special for statespaces is the classification of states into basic, abstract and composite states. *Basic states* simply have a name and nothing more. *Abstract states* are comparable to super states in UML state charts, viz. they have a name and consist of a set of states, whereof one is marked as initial state. Notably there are no transitions between the states in an abstract state. Transitions are defined globally, that is for the whole statespace. Nevertheless, states in an abstract state are connectible by transitions anyway, as all states are defined globally. *Composite states* do not have a name, but also comprise a set of states. Abstract states are allowed to contain further abstract states, but composite states must not comprise further composite states. The name of a composite state is implicitly available by merging the names of the states it composes.

Notably the description of the dynamic behavior of roles using statespaces leads to partitioning of roles. Depending on the current state of a role, it offers another interface. But providing runtime-context dependent interfaces is what roles offer naturally. Roles are hence decomposed into smaller parts. Andersons work on dynamic behavior descriptions does not focus on expressing who plays which role at which time, but on behavior modeling in general.

## 2.2 Roles on the Level of Implementation

A lot of work has been done on investigated roles conceptually. But there are approaches, trying to provide the role concept in programming languages, too. In the following four of them will be presented: powerJava, Rava, DOOR, EpsilonJ and ObjectTeams.

**PowerJava - Boella et al.** Guido Boella et al.[4] have a quite similar understanding of roles, compared to our view described in Subsection 2.1.2. Roles are bound to their players, have state and behavior and need to be bound to an institution, which we call context. An important difference lies in Boella et. al.s original field of research: multi-agent systems. The language introduced by Boella et al. is called **powerJava**[3] and is an extension to the Java programming language.

One of the biggest differences of powerJava's roles to the understanding presented in this thesis is, that classes in powerJava are required to offer some methods in order to play a role. Additionally, roles provide methods to objects, like in our approach.

Though powerJava is under development for more than five years, it seems not to have an active community, because it is still far from stable and no version of it is provided over the World Wide Web.

**Rava - He et al.** Rava[35] is an extension to the Java language supporting roles on the level of implementation. It is based on the Role-Object Pattern[6], but extends it using the Mediator Pattern[26]. Following Chengwan He at al. the main limitation of the Role-Object Pattern is its opacity for the programmers, which need to have in-depth knowledge about the pattern. Rava provides a set of new keywords, which abstract from the pattern itself.

A role is defined using the keyword `ROLE`, followed by the name of the role and another keyword `roleOf`, which precedes the set of class names, the role can be bound to. In methods of roles the keyword `@core` can be used, to refer to methods and attributes of the player instance. To invoke a role method the keyword `@INVOKEROLE` is provided, taking the name of the role, as well as the name of the method and the arguments.

These 4 keywords are translated into usual Java code, compilable by every Java compiler and running on all Java Virtual Machines.

Like it is the case for powerJava, no version is available for download over the World Wide Web. It is hence likely, that no stable implementation exists.

**DOOR - Wong et al.** DOOR is a dynamic object-oriented data base management system supporting roles[67, 66]. DOOR can also be seen as an object-oriented programming language, which has a focus on databases. The data model of DOOR defines class and role types, as well as objects and roles as instances of class types and respectively role types. Class and role types may form an inheritance hierarchy, where class and role type hierarchies are orthogonal to each other, i.e. a role type may not specialize a class type and vice versa. To link both hierarchies to each other Wong et al. defined a *playedBy* relation, which is defined in role types, as a partial function playedBy(RT) = RT → CT, where RT is a role type and CT is a *set of* class types. Thus DOOR supports multiple qualifying player types. DOOR has been developed, with persistency in mind. Nevertheless, the relational schema is not used therefore, but serialization. Objects, roles instance, and definitions and so on all are saved as binary data streams.

Unfortunately, since 1999, DOOR seems not to be further developed. No version is available over the World Wide Web.

**EpsilonJ - Tamai et al.** EpsilonJ[46], is a language extension to the Java programming language. EpsilonJ is based on a language independent model, called Epsilon[64]. The model describes a view on roles, which is close to the understanding of roles presented in this thesis. That is, context, role and players exist. In Epsilon, contexts comprise roles, which can be bound to objects outside of the context, which are called players. Notably, roles define a required interface, which has to be provided by objects, in order to qualify as player. The binding of roles and players needs to be expressed explicitly, using the methods `bind` or `newBind`. The life cycle of roles needs to be managed by the developer.

EpsilonJ does not have a very active community, so it only slowly develops and still is in a early phase of research. No version is available over the World Wide Web, but is send by mail on request.

EpsilonJ provides additional keywords to Java, like `context`, `player` and `requires`. The EpsilonJ *compiler* transforms EpsilonJ source code to Java source code. The version I requested for testing purposes was hard to use, because most transformations did not work.

**ObjectTeams - Herrmann et al.** ObjectTeams is a language extension to the Java programming language, which does not require using a modified virtual machine and does not rely on source-code transformations. Instead the source code, which uses ObjectTeams-specific keywords, is compiled directly to bytecode, runnable in any Java virtual machine. The only requirement is, to provide the ObjectTeams runtime to the virtual machine, which is a compact Java archive.

The most important concepts of ObjectTeams 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**, which are usual Java classes, marked with the keyword `team`.

Inner classes of teams are by definition **roles**, as long as they are not marked as teams. Hence nesting of teams is supported. A special keyword `role` does not exist, due to their implicit declaration. As roles are rigid, they need to be bound to a 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. Notably a player class is used, to express the binding. Thus, players of the role are limited to be instances of that class. Moreover, only one class can be provided, comparable to Java's single inheritance. Role instances are bound to their *player instance.* That is, roles cannot migrate from one player to another.

The semantics of roles is to change structure and behavior of their player. In ObjectTeams roles can be either **active** or not. Only active roles affect their players. Roles can be activated over their team. Either directly by invoking the method `activate()` or indirectly by specifying constraints under which the team shall be active, so-called guards, which will be introduced later on. 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 events, is not planned.

When an object starts to play a role and when it stops doing so, cannot directly be expressed in ObjectTeams. But the mechanisms of team activation and guard predicates enable to realize this behavior. Teams can be either active or inactive. The activation of a team leads to the activation of all its roles. Only active roles affect their players. In order to activate only a subset of the roles of a team guard predicates can be defined. These predicates can be defined on different levels - on the level of teams, roles or bindings. Team level guard predicates constrain, when a team can be activated at all. The predicates can rely on instance attributes of the team. Imagine the team `Car`, which has an attribute `numberOfGears`. A car should only be activated, when it has all 4 gears. A team-level guard predicate for this condition is denoted by `when(numberOfGears >= 4)` after the team declaration. Role level guard predicates allow to restrict the activation of roles. Whenever a team is activated only those roles of it are activated, whose guard predicates evaluate to true. Role-level guard predicates are allowed to rely on the roles attributes. Imagine the team `University` with the role type `Student`. Students can go on holiday for a semester, expressed by a boolean attribute *onHoliday.* Activation of the university, for example when the semester starts, should only active those students, which are not on holiday. To express this condition `when(!onHoliday)` needs to be added to the role declaration. Notably role-level guard predicates are not allowed to rely on attributes of the players. In order to access player attributes the additional keyword `base` needs to be added. Imagine the university with students. Student is a role played by persons, which have an age. If one wants to express, that students must not be older than 40 years, he can do so by using the following base guard predicate: `base when(base.age <= 40)`. Finally, there are binding level guard predicates. These enable to restrict the activation of single callins. That is the role itself gets activated, but not all of its callins are effective. In order to model dynamic role playing, role-level base guard predicates suffice.

ObjectTeams offers a set of reflective methods, enabling the developer to fetch, for example, all active roles. To model dynamic role playing, only one of these methods is needed: the method to check if a player plays some role of a given role type, called `hasRole(player, roletype)`. Combining all these language constructs leads to the Object-Registration pattern[3]. Roles are constrained by a base guard predicate, whose condition is, that the player of this potential role, does not already play a role of this type, that is `base when( Team.this.hasRole( base , Role.class )`. This way these roles will never be played without being explicitly forced. This can be done by providing methods in the team, which force role instantiation.

ObjectTeams is under active development since more than 7 years. Thus, many problems have already been addressed. OT/J is the most mature language currently available for the purpose of using roles on the level of implementation. For this reason it has been chosen as the first role language to be supported by DAMPF.

**Summary.** A lot of approaches exist, which tried to support roles on the level of implementation, without extending the base language. Michael Pradel et al. showed[55], how to support roles in Scala, Gottlob et al. showed how to integrate roles into Smalltalk[27], Dirk Riehle introduced the Role Object Pattern[6], which offers a method to use roles in all class-based object-oriented languages, to name just a few. Bettini et al. propose a further language extension to Java, called Dec-Java[8], using the Decorator pattern[26].

More approaches exist, but those already mentioned suffice, to give a good understanding about the current state of the art of the role community.

[3]http://trac.objectteams.org/ot/wiki/OtPatterns/ObjectRegistration

# Chapter 3

# Object-Relational Mapping

## 3.1 Basic Terminology of the Relational Model

This section will introduce the basic terminology of the relational model, introduced in 1969 by E. F. Codd[14]. A *relation schema* RS is comprised of a set of *attributes* S = {A,B,C} and a set of *constraints* $\Sigma = \{C_1, C_2, ..., C_n\}$. A *relation* R is an instance of a relation schema RS. It is comprised of *tuples*, which are sets of values. A tuple t of a relation R, which is an instance of RS, contains exactly as much values, as attributes are defined for RS. There is **no order** defined for the values of t and/or the attributes of RS. The term R(A,B,C) stands for a relation R, which is an instance of a relation schema RS, comprised of the union of the attribute sets A, B and C, where A, B and C are disjoint. An *A-tuple* is a tuple, containing as much values as A contains attributes. The *X-value* of a tuple is the value of the tuple according to the attribute X.

The simplest form of a constraint is the *functional dependency*, introduced by W.W. Armstrong[2]. In a relation R(A,B,C) a functional dependency FD = A→B, where A and B are sets of attributes, states, that if a tuple $(\alpha,\beta,\gamma)$ exists, than every other tuple, having $\alpha$ as its first value, must have $\beta$ as its second value. Or in other words: if two tuples agree in their A-value, than they agree in their B-value, too. The value $\gamma$ is not constrained by the functional dependency A→B. Further types of constraints are introduced in the next section.

The attributes of a relation schema can be distinguished into *prime* and *non-prime attributes.* A prime-attribute is unique for a relation. That is, if A is a prime-attribute of the relation schema RS, than every A-value of every tuple in each relation R, being an instance of RS, differs. Relation schemata may contain multiple prime attributes. All combinations of these prime attributes form *candidate keys.* A superset of a candidate key is called a *super key.* From the set of candidate keys a single primary key has to be chosen, which is later on used to address single attributes in so called *foreign keys.* A foreign key is an inter-relational constraint defined between 2 relations. If relation A has the primary key $P_1$ and relation B has an attribute set AS, with the same number of attributes as $P_1$, a foreign key can be defined from B to A, meaning that the AS-values of B point to tuples of A using the primary key of A as index.

These concepts of the relational model, summarized in table 3.1, form the basis for the following sections. Of course, more concepts exist. Nevertheless, they are not needed for the understanding of this thesis.

| **Concept** | **Description** |
|---|---|
| relation schema | a set of attributes and a set of constraints |
| relation | an instance of a relation schema; contains tuples |
| X-tuple | set of values, conforming to the attribute set X |
| Y-value | value of a tuple at the position of attribute Y |
| functional dependency | if two tuples agree in their A-value, than they agree in their B-value, too |
| prime attribute | an attribute whereof all values are unique |
| candidate key | each prime attribute combination of a relation |
| super key | a superset of a single candidate key |
| foreign key | constraint, wiring tuples of relations |

Table 3.1: Summary of Basic Relational Concepts.

## 3.2 Developing Fragrant Databases

Edgar Frank Codd, the inventor of the relational model[14], introduced principles for the construction of good smelling databases. His aim was to help the database engineer, to avoid common unwanted characteristics, such as insert, update and delete anomalies. Furthermore redundancy shall be omitted. The process of giving a database schema a good smell is called **normalization**.

In 1970 E.F. Codd introduced the **first normal form**[15]. The essence of this form is that a database schema is comprised solely by flat relations, i.e. no attribute is a relation itself. Figure 3.1 depicts this requirement using the relation `Student` as an example.

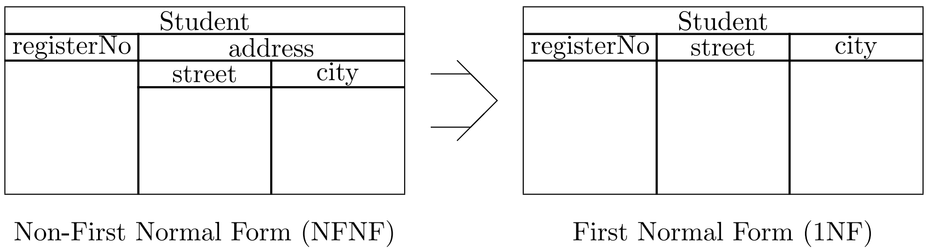


Figure 3.1: Requirement for First Normal Form.

In 1971 E.F. Codd introduced the **second and third normal form**[16]. The essence of the second normal form is that no non-prime attribute provides a fact about only part of the whole key. Imagine an enrollment of a student to a course. A suitable relation for such enrollments consists of the attributes `courseID`, `studentID`, `registerNo`, `name` and `timestamp`, where the last attribute is used to store the time at which the student enrolled himself to the course. The candidate key of this relation is comprised of the attributes `courseID` and `studentID`. Figure 3.2 depicts this relation.

| Enrollment | | | | |
|---|---|---|---|---|
| **courseID** | **studentID** | registerNo | name | timestamp |
| | | | | |

Figure 3.2: Example Relation Violating the Rules of 2NF. Contained Functional Dependencies: registerNo→studentID; name → registerNo; timestamp → {courseID,studentID}.

Examining the attributes reveals that `timestamp` is directly functional dependent on the candidate key, but the attributes `registerNo` and `name` only depend on `studentID`. Furthermore `name` only transitively depends on `studentID`, as it directly depends on `registerNo`, which depends on `studentID`. The functional dependency registerNo→studentID violates the requirement of the second normal form (2NF). In order to transform this relation to fulfill the requirements of the 2NF, the relation needs to be split into the relations `Enrollment` and `Student`. Figure 3.3 depicts a redesign of Figure 3.2, which is in 2NF.

The essence of the third normal form is, that each non-prime attribute "must provide a fact about the key, the whole key and nothing but the key."[39, p.120].

| Enrollment | | |
|---|---|---|
| **courseID** | **studentID** | timestamp |
| | | |

| Student | | |
|---|---|---|
| **studentID** | registerNo | name |
| | | |

Figure 3.3: Transformed Relation From Figure 3.2 Supporting 2NF.

It thus goes further than 2NF in that it prohibits transitive functional dependencies, i.e. functional dependencies that have attributes as their result, which are the value of another functional dependency which has the key or other attributes as their result. 3NF requires all non-prime attributes to be directly, i.e. non-transitively, dependent on the whole key. Another viewpoint on 3NF has been provided by Zaniolo in 1982[68]. He claims, that a relational schema is in 3NF if for all functional dependencies X → Z, where X and Z are sets of attributes, at least one of the following three requirements is fulfilled:

1. Z contains X, i.e. the dependency is trivial
2. X is a super key, i.e. a superset of the candidate key
3. X is a prime attribute.

Recall Figure 3.3, which depicts two relation schemata, which are in 2NF. The relation `Student` still contains two functional dependencies: registerNo → studentID and name → registerNo. The second dependency only transitively depends on the key (that is `studentID`). Thus this relation violates the rules of 3NF. Figure 3.4 shows a transformation of relation `Student` into two relations, which fulfill the requirements of 3NF.

| StudentRegisterNo | |
|---|---|
| **studentID** | registerNo |
| | |

| StudentName | |
|---|---|
| **studentID** | name |
| | |

Figure 3.4: Transformed Relation `Student` From Figure 3.3 Supporting 3NF.

In 1974 Raymond F. Boyce and E.F. Codd introduced the **Boyce-Codd normal form**[17]. This normal form further restricts 3NF in that it requires all attributes, including prime attributes, to provide a fact about the whole key. Another way to define BCNF is, to eliminate the third requirement of Zaniolos requirements for 3NF. Imagine a relation `Student`, comprised of the attributes `studentID`, `name` and `courseOfStudy`. A student is meant to study only one course of study. This relation contains two functional dependencies: studentID→name and studentID→courseOfStudy. The relation is in 3NF, as it does not contain any non-prime attributes, which provide a fact about only part of the key. Indeed the relation does not contain any functional dependencies having a non-prime attribute as its source, i.e. left-hand side. But the relation is not in BCNF, as it contains prime attributes, which do not provide a fact about the whole key. Namely both dependencies violate BCNF. Figure 3.5 shows a transformation to BCNF for this example.

A further normalization step has been introduced in 1977 by Ronald Fagin as the **forth normal form**[23]. Fagin introduced multivalued dependencies

| Student | | |
|---|---|---|
| **studentID** | name | COS* |
| 100 | INF | hans |
| 200 | IST | alex |

non-BCNF, but 3NF

| Student | |
|---|---|
| **studentID** | name |
| 100 | hans |
| 200 | alex |

| StudiesIn | |
|---|---|
| **studentID** | COS |
| 100 | INF |
| 200 | IST |

Boyce-Codd Normal Form

Figure 3.5: Transformation of Relation `Student` into two Relations supporting BCNF. *COS=courseOfStudy.

as generalization of functional dependencies. A functional dependency A → B implies, that if two tuples agree in the value of A, then they agree in the value of B, too. Multivalued dependencies take sets into consideration. See definition 1.

**Definition 1 (Multivalued Dependency (MVD))** *If in a relation R(A,B)* a *and* a' *denote tuples of A and* b *and* b' *denote tuples of B, then a multivalued dependency A →→ B implies, that if (a,b) ϵ R(A,B) and (a,b') ϵ R(A,B), then (a',b) ϵ R(A,B) and (a',b') ϵ R(A,B).*

Fagin defined a relation schema to be in forth normal form, if all multivalued dependency of it have the whole key as their result. Recall the relation `Student` from the left hand side of Figure 3.5. Now imagine that a student may study multiple courses of studies. This constraint eliminates the functional dependency studentID→courseOfStudy. But this relation contains two multivalued dependencies, where one is a functional dependency: name→studentID and studentID→→courseOfStudy. This relation is not in 4NF (and not in BCNF, due to the first MVD), as it contains multivalued dependencies, which do not have the whole key (indeed not even part of the key) as its result. Figure 3.6 depicts a valid transformation of this relation into two relations fulfilling the requirements of 4NF. In fact it is the same transformation as in Figure 3.5.

| Student | | |
|---|---|---|
| **studentID** | **COS*** | name |
| 100 | INF | hans |
| 100 | IST | hans |

non-4NF

| Student | |
|---|---|
| **studentID** | name |
| 100 | hans |

| StudiesIn | |
|---|---|
| **studentID** | COS |
| 100 | INF |
| 100 | MINF |

Fourth Normal Form (4NF)

Figure 3.6: Transformation of Relation `Student` into two Relations supporting 4NF. *COS=courseOfStudy.

In 1979 Ronald Fagin introduced the **projection-join normal form**[24] (PJ/NF), which is more restrictive than the fourth normal form and thus can be seen as the fifth normal form. The name PJ/NF emphasizes the normalization process, which uses solely the projection and join operators. In order to define the PJ/NF, Fagin introduced the concept of the *join dependency*, which is a generalization of multivalued dependencies. See definition 2. In simple words:

a join dependency *{A,B} exists, if the relation R(A,B) can be decomposed into two relations without loss of information. A relational schema is in PJ/NF, i.e. 5NF, if every join dependency is implied by the candidate key. The aim of 5NF is to decompose relations until no further decomposition, except it is based on keys, is possible[24, p.157]. In most cases 5NF does not differ from 4NF, except if some symmetric constraint on the data exists.

**Definition 2 (Join Dependency)** *A set A of columns, is join-dependent on another set B of columns, if the relation R(A,B) is the result of the join R'(A) * R"(B), where A and B share at least one attribute, i.e. A / B ≠ ∅.*

**Definition 3 (Trivial Join Dependency)** *If A and B are sets of attributes of the relation R and A∪B contains all attributes of R, than a join dependency *{A,B} is called trivial, if either A or B contain all attributes of R.*

A "bizarre" example of a relation schema, which is in 4NF, but not in PJ/NF was presented in [24, p.157]. A relation schema, which is comprised by three attributes A, P and C (A for agent, P for product and C for company) and obeys the join dependency $\sigma$ = *{AP,AC,PC} is in 4NF. The join dependency $\sigma$ says, that if:

1. agent a sells product p for company c',
2. agent a sells product p' for company c and
3. agent a' sells product p for company c, **than**
4. agent a sells product p for company c.

Now, additionally the constraint, that an agent must not work for two companies, whose products overlap, is assumed. This constraint violates the join dependency, because now, if some agent a sells product p for company c' and an agent a' sells product p for company c, than companies c and c' share product p and as agent a works for company c' he cannot work for company c, i.e. the above statements 2 and 4 cannot occur. As shown, no non-trivial join dependencies exist[1], hence the relation schema is in 4NF. But it is not in 5NF, as the join dependency $\sigma$, although instances of it can never occur, is not logically implied by the trivial functional dependency APC→APC.

In 1981 Ronald Fagin introduced the **Domain/key normal form**[25]. To clearly define this normal form two new types of dependencies need to be introduced: domain dependency (see 4) and key dependency (see 5).

**Definition 4 (Domain Dependency)** *A domain dependency IN(A,S) holds for a relation R(X), if A is an attribute, A ∊ X, S is a set of values (e.g. all strings of length 5, comprised solely by the big letters A-Z) and no tuple of R exists, which has an A-entry (i.e. the tuples value for attribute A) having a value not in S.*

**Definition 5 (Key Dependency)** *A key dependency KEY(K) holds for a relation R(X), if K is a subset of X and K is a key, i.e. the functional dependency K → X holds.*

[1]as the join dependency $\sigma$ cannot be fulfilled

Intuitively, a relation schema is in DK/NF, if it is free from any insertion and deletion anomalies. To properly define DK/NF, R. Fagin introduced a definition if insertion and deletion anomaly using the concepts of domain and key dependency. He defined a tuple t to be *compatible* with a relation R, if it does not violate any domain or key dependencies of R and if the tuple has the same attributes as R. He notably omitted other dependencies, e.g. functional dependencies. If there is a relation R, comprised by a set of tuples, which don't violate any dependencies, and there is a tuple t, which is compatible with R, than an insertion anomaly as defined by Fagin occurs, when the insertion of t into R, i.e. {t}∪R, results in a new relation R', which violates some dependency. Or, in other words, an insertion anomaly occurs if a newly inserted tuple t, which is seemingly valid, i.e. is compatible with R, leads to a new relation R'=t∪R, which violates some dependency. A deletion anomaly occurs, if a tuple t is deleted from a relation R and the resulting relation R' violates some dependency.

In 2002 Chris Date, Hugh Darwen and Nikos Lorentzos introduced the **sixth normal form**[20]. They focused on temporal data, i.e. data which is enriched with the information of when it has (or will be) valid. To temporalize a relation they add an interval attribute to it. If a table is in fifth normal form, that is, all nontrivial join dependencies are implied by a key, relations are further reducible. In general, if you have a relation R(A,B,C), where A is key of the relation and the join dependency *{AB,AC} is satisfied by R, than R is reducible to R(A,B) and R(A,C). Nevertheless R(A,B,C) is in fifth normal form as it does not contain any nontrivial join dependencies, which are not implied by a key. Temporalizing such a relation, i.e. adding an interval attribute, leads to the problem, that it is unclear, to which attribute the interval is related. That is, if you enrich R(A,B,C) with an interval attribute I to R(A,B,C,I), it is not clear, onto which attributes I relies. Imagine a relation `Student`, comprised of the attributes `registerNo`, `surname` and `courseOfStudy`. The attribute `registerNo` is meant to be the key of this relation. Two join dependencies exist: `*{{registerNo,surname}, {registerNo,courseOfStudy}}`. The relation is in fifth normal form, but further reducible. If this relation is enriched with an interval attribute, that is, the relation is to contain the information during which interval a given student had which surname and which course of study, it will be unclear, if that interval relates to the students surname or to its course of study. Therefore Date et. al. introduced the sixth normal form, which requires a relation to contain no non-trivial join dependencies at all or, in other words, the relation is irreducible. For the given examples it follows, that R needs to be decomposed into two relations $R_1$`(registerNo, surname)` and $R_2$`( registerNo, courseOfStudy)`.

Table 3.2 gives a short overview of all normal forms presented in this section.

As mentioned at the beginning of this Section, the aim of normalization is to design relation schemata, which do not suffer from insert, delete or update anomalies. From this point of view, 5NF is the *ultimate* normal form, if no temporal data has to be taken into consideration. Else 6NF would be the best choice.

It is important to mention, that the normalization of relation schemata to avoid such anomalies is only necessary if humans interact directly with the database. In case a persistence management layer is used by software written on top of it, these anomalies are avoidable by implementing a persistence manager, which takes care of the specific design of the database.

| **Normal Form** | **Meaning** |
|---|---|
| NFNF | relation contains attributes, which are non-atomic, i.e. relations themselves |
| 1NF | all attributes are atomic |
| 2NF | no non-prime attribute provides a fact about only part of the key |
| 3NF | each non-prime attribute must provide a fact about the key, the whole key and nothing but the key. |
| BCNF | each attribute, including prime attributes, must provide a fact about the key, the whole key and nothing but the key |
| 4NF | all multivalued dependencies have a key, the whole key and nothing by the key as result |
| 5NF (PJ/NF) | every nontrivial join dependency is implied by a key |
| DK/NF | the relation is free of all insertion and deletion anomalies |
| 6NF | the relation is free of any nontrivial (see definition 3) join dependencies, i.e. is irreducible |

Table 3.2: Overview of Normal Forms Introduced in this Section.

Though these anomalies thus are not important in the context of persistence managers, performance is. How long does it take to insert a new business object into the database? How long does it take to update it? How long does it take to delete it? And how long does it take to fetch it from the database? These four questions can be transformed into requirements for a persistence manager:

1. **c**reation, i.e. insertion, should be as fast as possible,
2. **r**ead access should be as fast as possible,
3. **u**pdates should be as fast as possible,
4. **d**eletion should be as fast as possible.

The third requirement, updates, competes against the others. This is, because changes, which do not affect the whole business object, will be updateable faster, as only those relations need to be updated, which hold the according parts of the business object. On the other hand side, if a business object is scattered across multiple relations, it will take longer to read it from the database, than it would take, if it is mapped to a single relation. The reason therefore is, that more joins need to be executed by the database management system. The creation of a business object, as well as its deletion, will take longer, too. This is because multiple inserts or deletes need to be executed. If the business object is mapped to a single relation, only one insert or delete would be necessary.

These performance tradeoffs, with regard to normalization, show, that it is not necessarily the case, that a higher, i.e. stronger, normal form is always the better choice. The stronger the normal form, the more relations are introduced for the same business object. In case of a web application, which is less often updated than viewed, a weaker normal form should be preferred.

## 3.3 Classes And Relations - Object Relational Mappers

In order to persist the business objects state of an application in a database, the structural description of them needs to be transformed into a relational form. In order to restore an application with such a persistence mechanism, the relational structure needs to be transformed back into the original class schema. Interestingly, object-orientation and the relational schema are not completely different from each other. They have a set of similarities and differences. The differences have been examined in detail for a long time and therefore even have a dedicated name: the impedance mismatch[22].

Five key differences between both paradigms can be identified. First, attributes of relations always have a primitive type, whereas attributes of classes are usually references to other objects. Second, references in the object-oriented paradigm are in general unidirectional. Ternary or n-ary relationships are not directly supported. In contrast, the relational model supports n-ary relationships in a straight forward way, using foreign key constraints[18]. Third, the unidirectional relationships of classes are *explicit*. Relationships in the relational model are only *implicit*. But, using foreign key constraints, it is possible to express them explicitly, too. The ability to query the data in combination with bidirectional relationships allows easier data access in the relational model. Object-oriented relationships only allow querying objects in one way, but not in the other. Central to this difference is that the relational model is set-oriented, whereas the object-oriented paradigm is not. Forth, both paradigms handle n:m relationships in a different way. In the object-oriented paradigm collection types at both ends can be used. In the relation model a separate collection relation is needed. Notably, the object-oriented paradigm does not directly point to all elements of a collection, but to a collection object. Hence, both paradigms have in common, that they do not support collections in a direct manner, but they differ in how they do not support them. Fifth, references in the object-oriented paradigm originate from an attribute, but refer to a whole object. References in the relation schema reference only part of the referenced relation that is the primary key. This is, because the identity of objects is an implicit property, which needs special care in the object-oriented paradigm. The relational schema uses primary keys to explicitly express identity.

The object-oriented paradigm and the relation schema have in common, that both base on a single central concept: the class or the relation. Both comprise a set of typed attributes. Notably both are flat, that is classes and relations are not nested. Admittedly, relations may be nested, if they are in non-first normal form. Nevertheless, usual database schemata should adhere at least to the first normal form. Inner classes do not break the flatness property, too, because an inner class can be seen as a separate structure. Most remarkably, both paradigms do not consider relationships between their concepts as first class citizens.

In the following the transformation between classes and relations is discussed and existing solutions are described.

### 3.3.1 Transformations Between Classes and Relations

Classes describe, like relations, a set of attributes. Additionally they describe methods, which contain behavior. In the context of business object persistency, the behavior is not of interest. The instances of classes, objects, contain values for these attributes, like tuples of relations. Hence, in the simplest case a class can be directly transformed into a relation. The relation only has to have the same attributes as described by the class.

The object-oriented paradigm enables the programmer to define *abstract* and *concrete* classes. Objects can only be instantiated from concrete classes. Abstract classes cannot be instantiated. Hence, they do not directly need to be persisted. Only if there is a concrete subclass of them, their attributes will be persisted.

Of course the transformation is much more complex than described in the last two paragraphs, which is due to further properties of classes. First, classes may not only contain attributes of some data type, but also attributes, which are references to other objects. Second, They may also contain attributes, which have a collection type. Third, classes may inherit attributes from other classes. Forth, classes may have static attributes. Finally, classes may be nested. These 5 properties of classes need special care as described in the following.

#### Handling Object References

If an attribute of a class references another class, then the attribute cannot directly be transformed into a relational attribute. The problem lies in the data type of the attribute. An attribute in the relational model needs to have a simple data type, like `INTEGER` or `VARCHAR`. There are two ways to solve this problem.

First, the referenced relation is nested into the referencing relations. As a consequence the relation schema will be in NFNF, due to the attribute, which is a relation itself. The resulting relation can easily be transformed into 1NF, by flatting the relation and renaming the attributes, if necessary. But if multiple classes reference the same class X, the resulting relations will contain the attributes of the transformed class X multiple times, leading to a waste of data space.

Second, the type of the primary key of the referenced relation is used as type of the referencing attribute. Additionally a foreign key constraint is defined, wiring the two relations together using the referencing attribute and the primary key. This way the attributes of the classes occur only once in the relations after the transformation. If the relation of the referenced class has multiple attributes as its primary key, then the attribute of the referencing class will be transformed into multiple attributes, too. Note, that it is possible for an object of a class referencing another class, that the reference is *null*, i.e. no other object is referenced. Hence, the attribute(s) forming the source of the foreign key need to allow null-values, too.

In comparison to the first approach the data usage is lower, as every attribute exists only once in the whole schema. The cost of this benefit is additional time, which is needed when the tuples have to be transformed back into objects. The additional time is due to the needed join of the tuples. For applications which often read data, but rarely store it, i.e. OLAP[2] applications, the first approach is more beneficial, because time is more important than space. For application which store data at least as often than they read it, i.e. OLTP[3] applications, the second approach is more beneficial, because space is more likely to get a problem, than time. Because the requirements of OLTP and OLAP applications contradict, the approach can only be optimized for one of them. DAMPF uses the second approach.

### Handling Collections

Classes may have attributes, whose values are collections of some type, i.e. are more than a single value. The relational model does not provide collection types. Hence, a direct transformation is not possible. There are three ways, to solve this problem.

First, the attribute of collection type is transformed into an attribute of a *string* type, i.e. `VARCHAR`. All values of the collection are transformed into their string representatives and joined by comma, like it is known from comma separated value (.csv) files. This approach has multiple problems:

- **Complex Transformation.** Not every value has an easily reconstructable string representative. Imagine a value, which is a reference to another object. It is possible to transform it into a string, containing all data, which is needed to reconstruct the object later on, but it is a complex and time consuming task.

- **Loss of Collection Type.** The type of the collection is lost by transforming the collection value into comma separated values. Imagine a *set* of integer values and a *list* of float values. Sets must not contain duplicates, lists are allowed to have duplicates. In order to detect the type of a collection solely by looking at its values rules are needed. One of these rules would be, that if every value only exists once, i.e. no duplicates exist, the type of the collection is *set*. If the list of float values does not contain any duplicates (it is only allowed to have them, but does not need to have them) the reconstructed collection type will be set and not list. The only way to reconstruct the correct collection is to add a string representative of the collection type to the list of comma separated values.

- **Separator Character.** If the collection comprises string values, than these values may contain the separator character. Many approaches exist, to deal with this problem. The most common one is, to preprocess each string value before it is added to the list of comma separated values and to put a special character in front of each occurrence of the separator character. Nevertheless these approaches are well known to be error-prone.

Although using this approach has the aforementioned problems, it should be taken into consideration as it requires the fewest data space. Ordered collections are supported, too, as the order of the values is implicitly contained in the order of the comma separated values.

[2]OLAP = Online Analytical Processing
[3]OLTP = Online Transaction Processing

Second, the collection attribute can be transformed into many attributes, which hold each value of the collection values. This approach is only feasible, if a maximum number of values per collection is known and if this maximum is relatively small. If both criteria fit, than this approach allows for collection values of different type, for example a collection value containing string and float values. Ordered collections are not supported by a relational database management system (RDBMS) in general, as the attributes of a relation have no defined order. Nevertheless, most RDBMS provide ordered attributes, as can be seen by SQL-Extensions of `ALTER TABLE`, supporting to add attributes at a specific position.

Third, the collection attribute can be transformed into a separate relation. The attribute will not exist as attribute in the relation of the transformed class. The separate relation contains each collection value as a tuple and has an attribute, having the type of the primary key of the relation of the transformed class. Additionally a foreign key is defined, wiring the collection relation with the relation of the transformed class. Having each collection value as a separate tuple provides the ability to use the comparison and selection features of the relational database management system, like for example `DISTINCT`. It is furthermore easily possible to support ordered collections by adding an attribute for the position to the collection relation. Support for collections with values of different type requires more than one collection relation. For each type, which is present in the collection, a separate relation is to be created. The foreign keys are defined from the collection relations to the relation of the transformed class, hence the do not pose a problem. If it is not possible to foresee every possible type in the collection, this approach reaches its borders. Nevertheless using collections with values of different, unforeseeable types is a bad programming practice. Hence not supporting them is only a minor drawback of the approach.

In comparison all three approaches support ordered collections and collections with values of different types. The third approach requires these types to be predictable. The fewest data usage is achieved by applying the first approach, whose biggest drawback is the time needed to handle all the problems mentioned before. The second approach can only be applied, if the collections have a relatively small limited number of values. Although the third approach requires the most data space, it can provide the fastest and cleanest way to transform collection attributes. The time needed to join the collection values with the tuple they belong to is less, than the time needed to preprocess each tuple and to reconstruct each value from its string representative. Hence the third approach is the most beneficial.

### Handling Inheritance

One of the biggest features of the object-oriented paradigm is inheritance. For the persistence aspect the inheritance of attributes is of interest. Classes inherit all non-private attributes of their superclasses. The relational model does not provide inheritance between relations, it is hence necessary to use other mechanisms to emulate inheritance. This problem is well known and over time three approaches emerged.

1. **Single Class.** For each inheritance hierarchy a single table is created, containing all attributes of the hierarchies' classes. Additionally an attribute to assign the tupels to their originating class is needed, which

usually contains the name of the class as string value. This approach leads to very high data redundancy, but offers fast restore times.

2. **Table per Class.** A class, which has superclasses, is transformed into a single relation, having all attributes of the original class and all non-private attributes of all direct and indirect superclasses. This approach offers fast restore times, too, as no joins are needed. It furthermore eliminates the data redundancy, but introduces schema redundancy, as non-primitive attributes of superclasses exist at least twice - in the subclass and in the superclass. Hence, this approach requires less data space than the first one, but more than the last one.

3. **Joinable Table per Class.** For each class a separate relation is created, containing only the attributes of the class itself and additional attributes, which point to the relations of the superclasses. Benefit and drawback of this approach are the opposite of the first two approaches, that is this approach requires the fewest data space, but therefore more time to restore the objects, because the tupels need to be joined.

In comparison the first approach is the least beneficial approach, due to its data and schema redundancies. The second approach provides fast restore times, whereas the third approach provides no redundancies and hence less data usage. Depending on the type of application, i.e. OLAP versus OLTP, a different approach should be used. OLAP application focus on fast restore times, hence the second approach is more beneficial. OLTP applications focus both on fast restore times and low data usage. Hence the third approach is feasible.

### Handling Static Attributes

Classes have two types of attributes: class- and instance-level attributes. Attributes on instance-level are tied to instances, i.e. objects. Class-level attributes are tied to the class itself, viz. they are the same for all objects. Of course they can be persisted just like instance-level attributes, but that is not necessarily optimal.

Storing static attributes like instance-level attributes leads to data redundancy, because for each object the same values are stored over and over again. There are two approaches, which address this problem. First, a separate relation can be introduced, which stores the values of static attributes. This relation needs as many attributes, as static attributes exist. If for each class such an additional relation for static attributes is created, these relations will always contain no more than a single tuple. Thus it is worth to think about a single relation for all static attributes of all classes, which is the second approach. The problem with this approach is, that the amount of all static attributes may be very high, leading to a relation with lots of attributes and sparse data, as each tuple only contains the values for the attributes of the class they belong to. It is possible to lower the amount of needed attributes by transforming the values to their string representatives, as described as a possible solution for collection attributes, but leading to the same problems.

If an application comprises many static attributes the first approach is to be preferred. Else the second approach is feasible. For classes, of which only few objects will exist, no separate relation for static attributes is feasible, too.

### Handling Nested Classes

The object-oriented paradigm allows nesting classes, leading to so called *inner-classes*. Inner classes can be transformed just like usual classes, with a single difference. They need to have an additional attribute, pointing to their enclosing instance, viz. the instance of their enclosing class.

Another possibility is to transform a nested class to an attribute, leading to a relation in NFNF. To transform such a relation to 1NF this attribute needs to be flattened. For small nested classes, which do not contain further nested classes, this approach is feasible, but the bigger these inner classes get and the deeper the nesting is, the bigger the resulting relation will be. Because RDBMS define a maximum number of attributes for relations, this approach is not always feasible. It furthermore leads to sparse data, as not every instance of a class with a nested class needs to have an instance of this nested class.

In comparison the first approach offers less data usage, but requires more time to restore objects than the second approach. For deeply nested classes with many attributes the resulting relations of the second approach get very huge, leading to performance penalties in terms of time, too. Hence, for applications which only have a few inner classes, whereof most only have few attributes, the second approach is more beneficial. Else the first approach is to be preferred.

### Transforming Relations to Classes

The last Subsubsections described in detail how classes and objects can be transformed into relations and tuples. But an Object Relational Mapper needs to be able to restore objects from tuples in the database, too. Notably, there is no need to reconstruct the class schema from relations, because applications store this information in files anyway. Depending on the approaches for transforming classes to relations, as described before, simple and fast queries can be used to load the required data of an object. If classes are split into many relations, more complex queries, which join the tuples and apply projections on them, are needed.

The complexity in restoring objects is implementation specific. Usually object-oriented programming languages use constructors, to create an instance of a class. Constructors are used to initialize the objects appropriately, which does not mean, that they initialize all attributes. Furthermore classes may have multiple constructors, initializing the objects in different ways by requiring different arguments as input. These arguments are not necessarily directly connected with the attributes of the class. Imagine an attribute `date` and a constructor taking the arguments `day`, `month` and `year`. In the constructor the arguments are used to create a date value, but each argument alone is not directly connected with the attribute of the class.

Hence all approaches for object reconstruction require the introduction of a special constructor, which differs from all existing ones. This constructor is used to create an empty object of the class, i.e. an object whose attributes are not initialized. Additionally means to initialize the attributes are needed, which leads to another problem: private attributes.

The problem with private attributes is, that they can only be altered inside of methods of the class they belong to. Hence for all attributes a single or separate method needs to be introduced, initializing the attributes and taking the value to be used as an argument. Of course these methods can be merged into the special constructor, so only one method, that is the constructor, needs to be introduced.

The introduction of such a special constructor is necessary, if all kinds of objects need to be restorable. Other approaches, which allow for object recreation, make assumptions about the application, narrowing the creative freedom of programmers and designers.

### 3.3.2 Existing Technologies

The Java Persistence API (JPA [30]), part of the EJB 3.0 specification from JSR 220, forms a standard for object relational mappers in the context of Java. Many implementations of this standard exist. For example Hibernate[4], EclipseLink[5], TopLink Essentials[6] and OpenJPA[7].

The most important concepts in JPA are *Entity*, *EntityManager* and *PersistenceContext*. Entities are domain objects, subject to persistency, like in DAMPF. In order to persist objects, the EntityManager is to be used. It provides a method `persist`, which takes the domain object, to be persisted, as an argument. Furthermore it provides methods to search and restore domain objects from the database. Besides finding entities by their primary key, using method `find`, complex SQL-like queries can be used. These queries allow to conjunctively or disjunctively connect attribute-value pairs. Because the JPA is in productive use, most implementations offer functionality for very sophisticated queries. Range checks, for example, are just one of the many features. With regard to the conceptual architecture of DAMPF, presented in Section 4.3, the EntityManager provides the runtime utilities *Store* and *Search and Restore*. Further transactional features, which are out of scope for this thesis, are provided by the EntityManager. The PersistenceContext defines a boundary for sets of domain objects. It enables the developer to group objects, which are subject to persistency, together. An EntityManager is directly connected to a PersistenceContext. That is, it is responsible for the management of all objects, belonging to the corresponding PersistenceContext.

Notably the JPA was designed for applications, running in a container. That is enterprise applications, which run in application servers, or web applications, which run in web servers. The reason for it is, that JPA implementations need to inject dependencies to the code, but use another approach than DAMPF. The injection is done by the container, not by the JPA implementation. Thus JPA is not directly usable in standalone applications. In order to use JPA in standalone applications anyway, the developer needs to take care about the injections himself. Indeed, there is no injection anymore at all, but the developer directly uses the API of the JPA to get instances of, for example, an EntityManager. This leads to very low transparency for the users of this persistency solution.

In summary, the JPA is a sophisticated persistency solution, providing a very expressive search feature, but its reasonable application is limited to enterprise and web applications.

[4]http://www.hibernate.org
[5]http://www.eclipse.org/eclipselink
[6]http://www.oracle.com/technology/jpa
[7]http://openjpa.apache.org

# Chapter 4

# The Concept of DAMPF

To realize the main features of DAMPF a novel approach to persistency of object-oriented systems supporting roles as first-class citizens is needed. The three main features of DAMPF are its support for schema evolution, the support for distribution of domain objects in heterogeneous environments and its support for context-based security.

The first main feature focuses on the evolution of application schemata, which are bound to a database schema, due to the application of an object-relational mapper. Current mappers do not allow schema changes in the object-oriented application, without losing the data in the database. This is, because changes to the application require the mapper to recreate the database schema, which is usually done by removing the whole schema and creating a new one. DAMPF will support changes to the application schema without loss of data in the database. Moreover, DAMPF will support to reuse data from earlier versions of the application. That is, for example data of classes, which formerly existed and are reintroduced, will be available again, too.

Distributed systems are designed to work together on a common task. To do so, they need to exchange the domain objects they are working on. Crucial to this distribution of domain objects is the serialization and deserialization of domain objects. Current approaches base on complex transformations. DAMPF will not support another transformation mechanism, but means for distributed systems to use domain objects they do not fully understand. A domain object adheres to the domain model of its originating system. Other systems are likely to have different domain models. But distributed systems are developed to work together and hence, are likely to have at least similar domain models. Thus, those parts of the domain object, which adhere to the domain model of a system, should be usable for that system. DAMPF will simply hide those parts of the domain object, which to not adhere to the domain model. Importantly, the hidden parts do not lead to loss of data, as the data is just hidden, but not deleted.

The tasks of distributed systems might incorporate domain objects, containing confidential data. To ensure, that only systems see data, which they

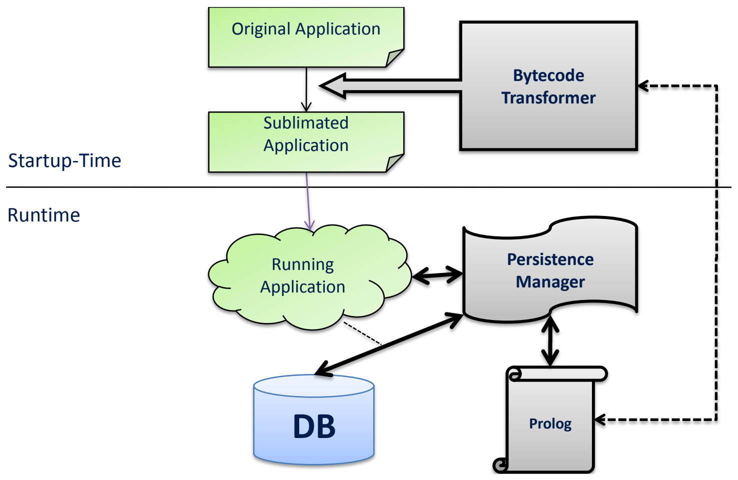


Figure 4.1: Coarse-grain Overview of Architectural Parts and Their Connection.

are granted to see, highly sophisticated, but also complicated, mechanisms exist. The complexity stems from the requirement of considering the context in regard to access rights. DAMPF will support context-sensitive security, by its ability to hide parts of domain objects. In order to ensure, that a system does not see confidential data, only its domain model needs to be designed accordingly.

Existing approaches do not support schema evolution, distribution in heterogeneous environments or context-based security. In the following, first a coarse-grain overview of the conceptual parts is given, followed by an explanation on the steps accomplished by DAMPF from the applications startup until runtime.

The main conceptual parts of DAMPF are depicted in Figure 4.1. The original application is transformed by a **bytecode transformer**, so it exposes its implicit dataflow in form of an explicit event stream. This transformer also inspects the application and writes out the applications schema to a Prolog fact base, which is dedicated to schema information. This fact base is from now on referred to as *schema fact base.* Schema changes are identified and noted by the transformer in that fact base, too. All this happens at startup time. When the application started up, the running application communicates with the **Persistence Manager**, which is the entry point into the *runtime utilities* of DAMPF. In fact, the Persistence Manager is listening to the event stream of the application, formerly exposed by the bytecode transformer. All events are traced, that is they are logged into a Prolog fact base. As this information is collected at runtime, it is noted in the so-called *runtime fact base.* The runtime utilities furthermore provide means to store, search and restore domain objects. All these activities force the Persistence Manager to connect and communicate with the database. In case the Persistence Manager connects to the database the first time, he creates the complete database schema. Else only changes are applied. Changes are handled at runtime, whereby applications can be fully dynamic, that is they can be changed at runtime. Finally, sending and receiving objects is provided by the Persistence Manager. The three parts on the right of Figure 4.1 are the major architectural parts of DAMPF: the transformer, the manager and the fact bases. The transformer focuses on sublimating the original application and the extraction and comparison of the applications schema. The manager provides functionality to trace the running application and to react accordingly. Notably, two Prolog fact bases exist: a schema fact base, containing a description of the applications schema and changes to it, and a runtime fact base, comprising knowledge about the application collected at runtime. The following Section describes the basic steps of the approach, which are derived from the main conceptual parts.

## 4.1 The Five Steps of DAMPF

In order to realize these features, DAMPF follows a set of steps, which will be described in the following.

1. **Sublimate** - Transparently adjust applications to make their data flow explicit.

2. **Compare** - Collect static information for comparison with older versions

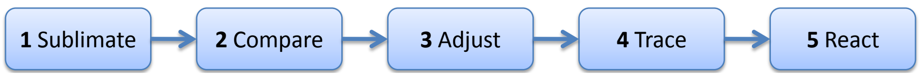


Figure 4.2: The Five Steps of DAMPF.

or remote applications.

3. **Adjust** - React accordingly on changes in the current version.
4. **Trace** - Collect runtime information and derive further information from it.
5. **React** - Provide runtime utilities, which base on the explicit data flow and react accordingly.

Figure 4.2 depicts these five steps.

**Sublimate.** Persisting the domain objects of applications means to store a snapshot of the current application in the database. To derive such a snapshot all domain objects need to unfold their state. There are two approaches to derive snapshots.

First a *one-shot approach* is possible. That is, at one point in time, at which the system shall be persisted, all the information is collected. This requires a way to find all domain objects. Usually a central data structure is used therefore. Some programming languages, like Java, provide tooling for this purpose. For Java the JVMTI, that is the Java Virtual Machine Tool Interface, can be used. Using the JVMTI allows to access the state of a currently running virtual machine, including the states of all objects, which are currently in the heap. As all objects are found this way, a mechanism to distinct between domain objects and system objects is needed. The easiest approach therefore is, to use annotations. Domain objects are marked with annotations like `@Entity`. The tool searches for this annotation, to identify domain objects. Objects, which are directly related to a domain object, need to be persisted, too. A more detailed explanation can be found in Chapter 5. When all domain objects have been found and their state has been unfolded the transformation to relations, as described in Section 3.3 and 4.2, follows.

The *second approach* to derive snapshots is to trace the state of domain objects. In order to get a trace, the classes of domain objects need to be sublimated. They are modified in a way that they unfold their implicit data flow. In Java this can be done by so-called bytecode modifiers. The modification can take place as an additional step after compilation, using post-processors, or at load-time of the class. Post-processing modifies the application before it has been deployed. Its benefit is that customers do not get in touch with the modifications at all. On the other hand side, the shipped code is fixed, that is changes to the post-processor have no effect on the application, after the application has been shipped to the customer. The benefits and drawbacks of load-time weavers are exactly the other way around. Customers potentially see the use of the load-time weaver, for example as an additional startup parameter. But load-time weavers are not part of the application and thus, can be updated separately. Another benefit of load-time weaving is, that it can be combined with other weavers. If a customer is already using a set of weavers, the order of them is important. If one of the weavers removes statements from a method body, which would be changed by another weaver, a different final application would result, with regard to the application order of the weavers.

Regardless if post-processing or load-time weaving is used, the changes to domain classes are the same. The only difference is the time, at which these classes are changed. All class-based object-oriented programming languages define classes with constructors, attributes and methods. In order to unfold the dataflow, which is implicitly contained in a class definition, constructors and methods need to be adjusted. The invocation of a constructor reveals the creation of a new object. The invocation of a method potentially changes a set of attributes. Which attributes are changed can be identified by analyzing the bytecode of the method. Thus at the end of constructors an event is to be fired, signaling the creation of a new object. At the end of each method an event is to fired, signaling the values of the changed attributes. By using events to unfold the implicit data flow, the problem of encapsulation is avoided. The basic principle of encapsulation is not violated, as it extends the principle of encapsulation with so-called friendly classes. Friendly classes in this context are classes, which are listeners to this event stream. Notably the listeners are bound to the domain objects by code in the domain classes. The listeners don not look into the domain objects, but the domain objects tell them their inner state.

Changes to roles highly dependent on how the language (extension) for them is implemented. All implementations should have in common, that there is some method to create a new role, there is a method signaling the destruction of a role and that there are methods that are used to bind the role to its context and player. Furthermore roles have methods changing their state. Notably role methods may change the state of their player, too.

Thus, in general three relevant types of methods in roles can be identified: creators, destructors, binders and usual methods, which have no effect on the roles life cycle. Creators need to be adjusted in the same way, like constructors of classes are, that is they need to signal the creation of a new role. Destructors need to signal the end of a roles life cycle. Binders need to fire an event, signaling which role has been bound to which context or player. Life-cycle independent methods potentially change the roles state and hence need to be sublimated, like methods of classes, viz. signaling the values of changed attributes.

The last two paragraphs described how classes and role types are adjusted to unfold their implicit data flow. In summary, the firing of events, related to the life cycle of objects and roles or to value changes of attributes, is incorporated into the methods. The listener for these events is fixed in the modified code and will be described in more detail in this Section under the paragraph of **React**.

**Compare.** At the same time the application is sublimated, the schema of the application is extracted to a fact base. That is, either immediately after the compilation (post-processing) or at class load-time (load-time weaver). The schema will be extracted completely, if no old fact base, which has formerly been extracted, exists.

All structural information of the applications domain objects will be written into a Prolog fact base. For example, the existence of each class, role, context and attribute will be noted as a separate fact. The relations between classes, that is inheritance hierarchies and references between classes, will be noted, too.

If, at an application startup, a formerly extracted schema already exists, the facts are compared. That is, for every fact, which is extracted from the new schema, its existence will be checked. If the fact already existed, there has been no change to this part of the structure. If the fact does not exist, it has either been changed or removed. It is not always possible to determine, whether the fact has been removed or changed. Facts of attributes can be changed in three ways: the type, name or both. If only the type has been changed, the change can be identified by the name of the attribute, which needs to be unique for the class. This does not work, if the name has been changed. Although names of attributes need to be unique, the intention of the developer cannot be reconstructed. It might be, that the developer removed the old attribute and added a new one, of the same type. Or he removed an old attribute and added a new attribute with a different type, but the same name. The same holds for facts about the existence of classes. If a class has been renamed and is not related to any other class, it is impossible to determine, whether the developer removed the old class and added a new one, or if he just renamed the already existing class. Thus changes to facts will in general be recognized as removals and additions. By default the name will be used to identify changes.

The information derived from the comparison needs to be noted, too. This is, because the database schema, which already exists in that case, needs to be adjusted. As all changes are removals, additions or combinations of both, the fact base is extended with a set of facts, representing the changes in this way. Having this knowledge in the fact base enables to react on changes at runtime in a flexible way. For example, knowing, that an attribute has been removed, allows to avoid removing the attribute from the database and thereby removing all the values of it, but just not to fetch it from the database. That is, when the domain objects are restored from the database, this attribute will not be contained in the projection of the corresponding query. The next paragraph, elaborating the **Adjust**-Step, will present ways to react on changes.

Every time an application is started, the formerly described comparison takes place. Thus, if the application is started for the third time or above, the already existing fact base contains facts about former changes, too. This information must not be removed. The fact base will contain all changes to the application, beginning at its first version. This leads to a stacking of change information, which is beneficial in terms of knowledge, but disadvantageous in terms of space. One of the features of DAMPF is, that it can be used for applications under development. Thus, it will be common, that an application is started the hundredth time with changes. To avoid crowding of the fact base, the developer needs to get the ability to tell DAMPF, that at the next startup all already existing changes shall be fixated and removed from the fact base. An earlier removed field, for example, will be removed from the database, if the developer decides to fixate all changes. To enable the developer to decide on this fixation, he needs to mark the current application as a milestone release. If a version is marked as milestone, developers start a new iteration, which usually does not rely on old data, viz. the application is unlikely to reuse attributes of earlier versions.

| **Cause** | **Effect** |
|---|---|
| Added attribute | add corresponding attributes and possibly required relations to the schema |
| Removed attribute | *leave* the attribute in the schema |
| Added class, role or context | add corresponding relations to the schema |
| Removed class, role or context | *leave* corresponding relations in the schema |
| Role changed context or player | in general a runtime change, add corresponding attribute to new context, but *leave* attribute in the old context |
| Added reference | add corresponding foreign key to the schema (is always connected to an added attribute) |
| Removed reference | remove corresponding foreign key from the schema (always connected to a removed attribute) |

Table 4.1: Effect of Application-Schema Changes to the Database.

**Adjust.** When the application is started for the first time, no schema fact base exists, but will be created. The adjustments, which are subject to this step, are not performed in this case. If the application is started for the second time or above, the already existing schema fact base is used along with the applications schema to identify changes. If schema changes have been found, DAMPF needs to handle them. To do so, the database schema needs to be adjusted and when storing and restoring domain objects the changes need to be considered, too. The adjustments highly depend on the type of the identified change. In the following the database adjustments in accordance to the identified changes are examined, followed by a description on how changes are considered, when storing and restoring domain objects. Table 4.1 shows the database adjustments as effect on application-schema changes at a glance.

If an attribute is added to a class or role, a corresponding attribute needs to be added to the database schema. This attribute is *nullable* by default. Developers can provide default values by setting the value in-place, viz. not in the constructor, but where the instance attribute is defined. If the added attribute is a reference to another class, an attribute pointing to the primary key of the corresponding object of the other class is to be added. If the added attribute is a collection, a collection relation and an attribute pointing to it is to be added. In general the addition of an attribute leads to relational constructs as described in Chapter 3 and Section 4.2.

If an attribute is removed from a class or role, the corresponding attribute remains in the database schema. It is not deleted, except the developer marks the current version as milestone release. In such a case, the corresponding attributes and connected collection relations need to be removed from the database schema.

The *addition* of classes, contexts and roles leads to the addition of corresponding relation schemata. The transformation of roles and contexts will be done, as described in Section 4.2. The *removal* of them does not lead to their deletion from the database schema. They will be kept in the database, like attributes, until the developer marks the application version as milestone release.

Changes, which are more complex in their handling, are changes to the bindings between roles and contexts. In general roles are able to change their context. Thus, conceptually these changes are runtime changes, which should be considered while sublimating the application, so they are fired as life-cycle events. Because ObjectTeams, which is used for the references implementation, only supports fixed players and contexts, the reference implementation of this thesis can consider such changes from a schematic point of view. The change of a unary binding of a role to its context is recognizable as removal of the role from one context and addition of the role to another context. Developers need to provide the information, if the role already existed in the former context, or if a new role is added. The easiest way for the developer to provide this information is to annotate the role class.

Roles might also change their player. Again, ObjectTeams fixes a role to a single player, so the reference implementation can see player changes as schema changes. In general the change of a player should be seen as a life cycle event. If another player class is bound to the role in the new version, the relation schema of the role needs to be updated. As old roles rely on old players and the developer might want to rollback its changes in the future, a new attribute pointing to the primary key of the new player is added, along with a foreign key for this connection. The old foreign key stays untouched! If the developer marks the current application version as milestone release, all former foreign keys and attributes pointing to the old players are removed from the database. Whether a role is actively played by an object is not relevant to the schema, because starting or stopping to play a role, as well as playing a role or not, is information available at runtime only.

Finally, references can be added or removed. The relational counterpart of references is the foreign key constraint. Hence, the addition of a reference leads to the addition of a corresponding foreign key to the database schema. Unlike for classes and attributes, the removal of a reference does lead to the removal of the corresponding foreign key!

**Example 6** *Possibly the most important domain concept of a university management system is the student. In a first version of such a system a class* `Student` *with the attributes* `name` *and* `height` *exists. In a future version of the system the* `height` *attribute has been removed, as it is of no interest for the university. Furthermore an attribute* `country` *is added, which points to an instance of the newly added class* `Country`*, which has a single attribute* `name`*. The first time the application is started, the original schema is noted in the schema fact base and a database schema with a relation* `Student`*, having* <u>*three*</u> *attributes,* `id`*,* `name` *and* `height`*, is created. When the application is started, after the above explained changes have been applied, a set of changes is identified. First, the removal of the* `height` *attribute. Second, the addition of the* `country` *attribute. Third, the addition of the* `Country` *class, along with its attribute. And finally, the addition of the reference from* `Student` *to* `Country`*. The adjustments to the database are the addition of a new attribute* `country` *to the* `Student` *relation, the addition of a new relation* `Country`*, along with two attributes,* `id` *and* `name`*, and the addition of a foreign key constraint, expressing that the attribute* `country` *in relation* `Student` *points to the* `id` *attribute of the* `Country` *relation. Figure 4.3 depicts this example.*

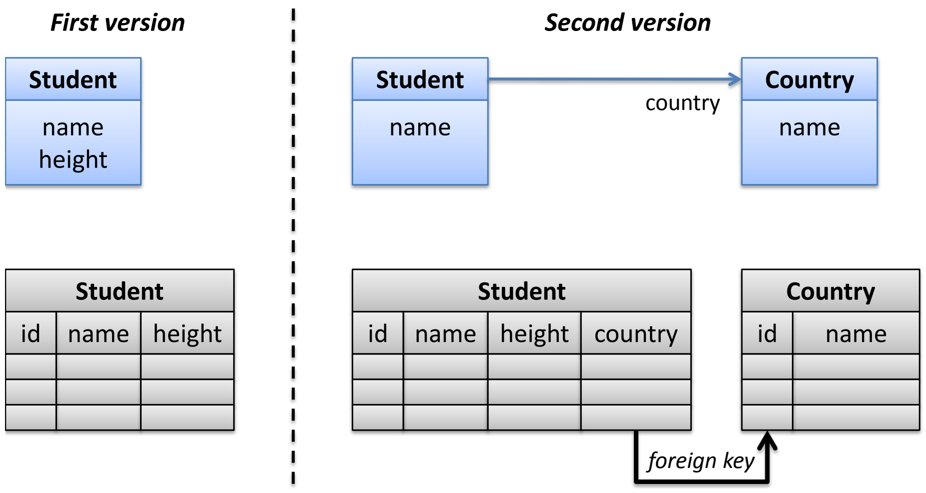


Figure 4.3: Example of Database Adjustments, due to Schema Changes in the Application.

Besides changes to the database schema, DAMPF needs to take care of schema changes when storing and restoring domain objects, too. Both activities are subject of the runtime utilities, which are examined in more detail in the paragraph for step **React**. In short, storing boils down to the insertion of one or more tuples to the database, restoring to fetching one or more tuples and joining them.

If attributes of a domain object have been removed from the application schema, they still exist in the database schema. Storing such a domain object needs to explicitly name all attributes, in order to leave out the removed one. Alternatively, the value for the removed attribute needs to be stated as *null*. This does not pose a problem, as all attributes, derived from class attributes, are nullable. Of course, in object-oriented programs it is possible to ensure, that an instance attribute must not be unset after object instantiation, it is always possible, that in an upcoming version of the application, this attribute will be null for some objects. In general attributes referencing objects, are by default *null*. Because even common data types like integers and booleans are often represented as objects, all attributes need to allow *null* as a value. When restoring a domain object, in the context of removed attributes, only the still existing attributes need to be fetched from the database. These principles for the handling of removed attributes are valid for classes, contexts and roles.

The addition of attributes needs to be considered, too. Storing a corresponding domain object requires to store the value of the new attribute, too. Restoring needs to take care of default values, viz. the domain object is restored with *null* as the value for the new attribute, but in the corresponding class, context or role the attribute may have a default value, which needs to be set.

The addition of separate classes will be handled, just like for all classes at the first application startup. The same holds for contexts and roles. The change of role—context and role—player bindings needs to be considered, in that the corresponding insertions and selections leave out the old binding-attribute, but use the new binding-attribute.

In summary, database adjustments, along with runtime adaptations are required, to handle schema changes of an application. Renaming poses a problem, in that it requires the developer to provide information about his intend. Either he renamed the attribute, class, context or role. Or he removed the old and added a new one. Additions and removals do not pose any problems for DAMPF.

**Trace.** This step denotes an important part of the runtime utilities, which are presented as a whole in the last paragraph of this Section. As described in the **Sublimation**-step, the implicit dataflow is transformed into an explicit event stream. This event stream exposes the state of the domain objects as well as their life cycle. This information is highly important for DAMPF. Hence, it is logged. As shown in step **Compare**, schema information is noted in a Prolog fact base. The same is done with the event stream. This way static and runtime information is available for further processing.

Not every value-change event needs to be logged separately. More interesting is the current state of an instance, viz. all values of the instance. This is where Prolog provides its first benefit for DAMPF. As the structure of domain objects is available in Prolog and the value-change events, too, it is possible to derive the current state of the domain object in Prolog.

A further benefit of having static and runtime information as Prolog fact base is, that it allows for **completely automatic normalization**. In order to normalize a schema, its functional dependencies need to be identified. Therefore more than just the schema is needed, but runtime information, viz. actual data, too. As will be shown in Section 5.4, normalization of real-world schemata is too time-consuming for nowadays desktop computers. The reason therefore is that classes, having 20 or more attributes, have an enormous amount of possible functional dependencies. Optimizations to the normalization algorithm allow deriving the normalized schema with limited memory space and in shorter time, but even single classes, with lots of attributes, consume time in the range of minutes, which is not feasible for DAMPF to normalize on the fly.

Normalizing a schema leads to splitting of classes and, consequently, to the need to join them later on. In Prolog rules can be defined, that accomplish this task. The ability to split and join tuples as well as classes in Prolog, allows handling domain objects, which are part of an inheritance hierarchy, separately. This helps with handling the value-change events, fired at the creation of an object and enables to postpone the decision, how to persist inheritance hierarchies.

Summarizing, the use of Prolog to trace runtime as well as schema information, enables DAMPF to derive further information from the application, which is not available otherwise.

**React.** This final step is, like the former, related to the runtime of applications in that it provides runtime utilities. It contains the listener of the event stream from step **Sublimate**. Besides forwarding the events to the trace utility, described in the last step, it provides utilities to store and restore domain objects.

When an object is to be stored, can be defined in multiple ways. The developer can **choose** from a set of **persistence strategies**. Those range from very course-grain to fine-grained levels:

1. Application Level - all objects are stored, when the application is shut down
2. Instance Level - all objects are stored, whenever a new object is instantiated
3. Value Level - single objects are stored, whenever a value of them changes
4. Manual - the developer proactively invokes a method provided by DAMPF

Notably, the Java Persistence API only provides the last persistence strategy. That is, the developer needs to manually incorporate the logic, when to persist which domain objects. DAMPF offers application developers the choice.

With application level persistency all objects are only stored, when the application shuts down. This leads to the lowest runtime-penalties, because there is no database interaction while the application is running. In contrast, this strategy does not support multiple users to work concurrently on the same data. Furthermore no transactional features can be provided, which would be helpful if the system crashes.

Instance level persistency lowers the drawbacks of the application level strategy, but lowers the benefits, too. This strategy persists all objects, whenever a new object is instantiated, or more specifically, when the instantiation of the object finished. Using this strategy, multiple users can work concurrently with the same data. They see course-grain changes of each other. Also course-grain transactional features are supported. Changes, which have been done since the last storing, can be undone, using the trace log, as described in the last step. On the contrary the applications performance is lowered, because now the application will interact with the database while the application is running.

Finally, value level persistency is the most fine-grained level one can think of. Whenever a value is changed, this change will be reflected in the database. Of course this leads to massive performance penalties, but offers the best multi-user support and fine-grained transactional security.

If all these strategies do not fit the developers' needs, he can decide to manually invoke the persistence mechanism. All he needs to do is to invoke the `store`-method of the runtime utilities. DAMPF will provide a mechanism to inject this method. The former strategies simply weave this call into the appropriate places, for example, into the handler of value changes, to achieve value level persistency.

To restore objects a search mechanism is required. DAMPF focuses on domain objects, not applications as a whole. Thus, it is not possible, to restore all domain objects and hook them into the application. The same holds for well-known object relational mappers, like those following the JPA. In order to search and restore domain objects, a set of search criteria can be send to the runtime utilities of DAMPF. Search criteria consist of attribute-value pairs, which can be connected conjunctively (and) or disjunctively (or).

**Example 7** *To search, for example, for an instance of class `Person`, having the attributes `firstname` and `surname`, representing the person 'John Doe', two* *attribute-value pairs [firstname,'John'] and [surname,'Doe'] are connected conjunctively and send as an argument to the search method. If such an object has been persisted formerly, it will be found.*

Because the focus of this thesis is on schema evolution and support for distribution in role-based systems, the search criteria API provides only limited expressiveness. Further development, to support for example value ranges, is future work.

**Distribution and Security.** The distribution and security features crosscut the five steps presented before. The composition of multiple, distributed applications to work on the some global task, usually requires all these applications to have a common domain model or complex conversations of the shared data. Web services base on conversations. They are described by the web service definition language (WSDL), which particularly defines the structure of messages a web service understands or will give as an answer. These messages form the domain objects send to and from the web service. In order to invoke a web service the corresponding message needs to be constructed, which requires converting the domain object into the requested message format. Another approach to distributed processing is to use a classic middleware layer. In essence, different clients work on the same data and share a domain model provided by the middleware layer. In contrast to web services, this approach leads to tight coupling of clients. Or in other words, developing clients independently from each other is much more complicated.

Using the approach to persistency of DAMPF in combination with the dynamic properties of roles, a novel approach for distribution emerges. Applications, which shall be composed to work on the same global task, need to share only a single concept, that of a task. The only property of such a task is, that it has an identity. Everything else is realized through roles, which are played by that task. If two applications have the same role definitions, they will both understand this part of the domain object. A complex domain object, playing many roles, will be understood by every participating application. Complex conversions are not needed, because each application will understand those part of the domain object, defined by the roles of the applications domain model. This way security can be ensured, too. Confidential data is part of roles, whose definition will only be available in those applications, which shall have access to it. Most importantly, the data which cannot be seen by an application, will not be removed by DAMPF, but just be hidden. This way an application can work on parts of a complex domain object, without understanding the entire object. Those parts, which were not understood, remain unchanged and are available in the next application in the processing chain, which knows about these parts. The feature of leaving parts of a domain object untouched is essential to the schema evolution feature of DAMPF. Remind that the removal of attributes from the class schema does not lead to their removal from the relation schema.

The distribution of domain objects poses another problem. The objects to be send need to be serialized and, when received, deserialized again. Many approaches for how to serialize objects exist. DAMPF provides a novel way. Each object of an application is, in consequence of the **Trace** step, available as a set of Prolog facts. These facts are textual and do not need to be read in a specific order. Hence they already denote a serialized form. DAMPF

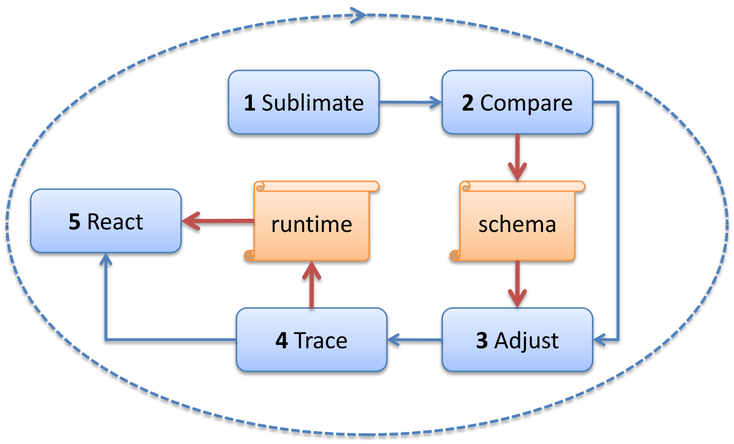


Figure 4.4: Cycling Steps of DAMPF with Fact-Base Connection.

will provide injectable methods to send and receive domain objects. Whenever these methods are used, the object identifier is used to fetch the corresponding facts from the Prolog fact bases, and to the shipment of them. Furthermore encryption technologies can be used, for a secure transfer.

**Summary.** The first three steps all affect application startup time, viz. the application is sublimated at startup time, thereby transforming the implicit dataflow into an explicit event stream, followed by the schema comparison and database adjustments. The last two steps focus on runtime utilities, used to trace the application and to **store**, as well as **restore**, domain objects. The steps **Sublimate** and **Compare** lead to two fact bases: the runtime fact base and the schema fact base. The third step relies on the schema fact base for the database adjustments. The last two steps base mainly on the runtime fact base. Figure 4.4 depicts the dependencies. It furthermore shows the cycling nature of the five steps. Each time an application is started, a new cycle begins. When the application is running, DAMPF will be in the last two steps, that is **Trace** and **React**, until the application is shut down and a new cycle is waiting to begin. When domain objects are stored, is up to the application developer, who can choose from differently coarse- or fine-grained persistency strategies. DAMPF provides means to enable the developer to define, when domain objects are restored.

## 4.2 Role-Relational Mapping

Chapter 3 investigated how classes and objects can be transformed into their relational counterparts. The transformation of roles, contexts and bindings between roles, contexts and players into the relational world belongs to the contributions of this thesis, because it has not been investigated, yet. This Section elaborates on these transformations.

The transformation of classes and objects to relations and tuples has been investigated for a very long time, but not the transformation of roles and related concepts into relational concepts. The dynamic nature of roles as extension to objects puts additional requirements on such a transformation.

The research field of role-relational mapping is very new and has been investigated, up to now, in only one other work than this thesis. Olaf Otto extends EclipseLink, an implementation of the JPA, to support the role concepts of ObjectTeams in his thesis[54]. The thesis focuses solely on ObjectTeams and does not investigate other approaches for role oriented programming. Because ObjectTeams does not provide a direct mechanism for an object to start or stop playing a role, viz. a pattern is needed to implement these tasks, this thesis does not examine the dynamic characteristics of roles. The focus of his thesis is on the aspectual character of roles in ObjectTeams. The result of the thesis is an extension to EclipseLink, supporting the concepts of ObjectTeams, although not all features of ObjectTeams can be used, which is due to bugs in EclipseLink, for example [53].

As described in Chapter 2 there are two important concepts related to roles. First, roles are bound to a *context*. Or, in other words, a context provides a boundary for the interaction of roles. Second, roles are bound to a player, viz. an object, playing this role. Roles have at most one player at a time and belong analogously to one context at a time. On the contrary, contexts comprise many roles and players can play more than one role at a time.

Objects are described by the class, whose instance they are. Role types form the description of role instances. Thus the transformation of roles to relational concepts divides into the transformation of role types and role instances. As classes are transformed to relations and objects to tuples, the same can be done with role types and role instances.

Depending on the specific language or language extension used, role types are realized as classes. Rava [35] for example realizes roles as usual classes. ObjectTeams [36] defines role types as inner classes, whose outer class is a context. This way the context-dependency is modeled using standard mechanisms of the language, namely enclosing instances. An important difference between role types and classes is, that role types describe, besides behavioral changes, how the structure of their player changes. Usually this is done by defining a completely new structure in the role type. Callers using the player through a role instance of that type will only see this structure. No current language (extension) offers the feature to define changes of the players' attributes declaratively. ObjectTeams provides a usable mechanism, but is limited to method signatures. It is possible to declare a method in a role type, which represents an attribute in the player, but it is not possible to declare an attribute in a role type which refers to an attribute of the player. The structural changes do not need to be taken into consideration for the transformation, because they do not change the structure of the player in a sustainable way. Indeed they do not change the structure of the player at all, but provide another interface.

Three different approaches can be used to transform roles, contexts and players to relations. First, all are transformed into completely separate relations. Second, role-player combinations can be transformed into a single, that is a combined, relation. Third, the transformation of the second approach is used, but the resulting relations are normalized to NF3.

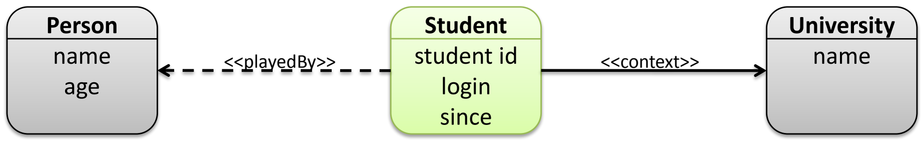
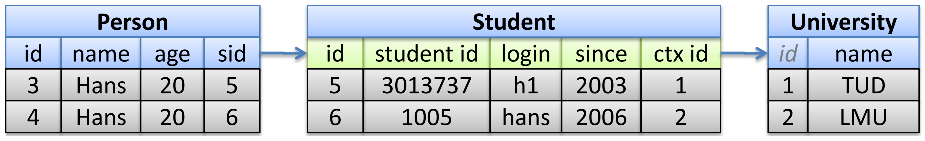


Figure 4.5: Example Scenario in Class-Role Representation.



| Person | | | |
|---|---|---|---|
| id | name | age | sid |
| 3 | Hans | 20 | 5 |
| 4 | Hans | 20 | 6 |

| Student | | | | |
|---|---|---|---|---|
| id | student id | login | since | ctx id |
| 5 | 3013737 | h1 | 2003 | 1 |
| 6 | 1005 | hans | 2006 | 2 |

| University | |
|---|---|
| id | name |
| 1 | TUD |
| 2 | LMU |

Figure 4.6: *Complete-Separation* Transformation of Example Scenario into Relational Representation with Exemplary Data.

### 4.2.1 Complete Separation of Roles, Contexts and Players

Role types can be transformed like classes into relations, whereas each attribute of the role type becomes an attribute of the corresponding relation. Additionally references to the current enclosing context and to the current player need to be represented as attributes. The context and the player, which both are represented as classes, are transformed to separate relations. The transformation of role instances reveals another required attribute: *a role identifier*. Conceptually roles do not have an own identity, because they are non-rigid. That is a role shares the identity with its player. Nevertheless, for a relational representation of roles an identifier is required to express, to which player the role belongs at the point in time the application has been persisted. Hence, the role identifier does not compromise the formal properties of the role concept. Note that objects suffer from a similar problem. That is, they do not have an explicit identity. Hence all object relational mappers introduce a primary key into their relational counterparts.

**Example 8** *Imagine the role `Student`, which is played by instances of class `Person` in the context of the `University`. Persons have a name and an age. Students have a student id, a login and an attribute `since`, expressing the year they started to study. Universities have a name. Figure 4.5 depicts this scenario as class-role diagram. Figure 4.6 depicts its transformation, according to the complete-separation approach. It furthermore contains exemplary data. Hans studies at 2 universities. Since 2003 he studies at the TUD and since 2006 at the LMU.*

As you can see in Example 8 the role identifier is not used as target of a foreign key, that is, it is not used to reference the role instance. The reason for the identifier to exist is that other role types might exist in the future, which reference this role.

### 4.2.2 Class-Role Relations

An alternative way of transforming roles to relations is, to add their attributes to their players. In general role types are not bound to a specific player class, that is roles may be played by any object. Usually this freedom is limited to a set of role types, whose instances are allowed to play the role. But this limits the dynamics of roles in that only anticipated players can be used. It thus gets more complex to support unanticipated changes. Some languages, like ObjectTeams, even limit role types to be played by objects of only one player class. But in return ObjectTeams provides powerful, expressive declarations to define how the role changes the behavior of its player.

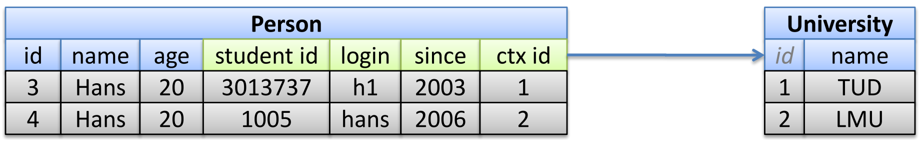
In consequence the attributes of a role type are added to all relations of its transformed player classes. Figure 4.7 depicts this transformation, based on Example 8. Depending on the limitation of the language this leads to semantic redundancy. Additionally each player relation needs an attribute pointing to the enclosing context of the role. Because players may play many roles this swiftly leads to very big relations. It furthermore leads to relations with sparse data, because the primary use of the relation is to store instances of the transformed player class, which will rarely play all roles the player might play.

The first approach, the complete separation of roles, contexts and players is hence more beneficial. A way to avoid the redundancy of this approach is to normalize the resulting relations to NF3 as will be described in the next Subsection.

### 4.2.3 Normalized Class-Role Relations

As described in the last Subsection, the attributes of roles can be added to relations of their players, leading to a considerable schema redundancy. The typical approach of a database engineer, to get rid of this redundancy, is to normalize the relations to NF3. The consequences of NF3 are, as described in Section 3.2, no schema and no data redundancy. DAMPF provides a tool to automatically normalize relations, so the user does not need to do this complex task manually. The tool contains all required steps to normalize in a fully automatic way. That is, functional dependencies are discovered by inferring them from the runtime fact base, an optimal primary key is inferred, functional dependencies violating normal forms are identified and the relations, which do not adhere to a given normal form, are split accordingly. The Prolog rules, realizing this functionality are listed in the appendix. Section 5.4 elaborates on normalization.

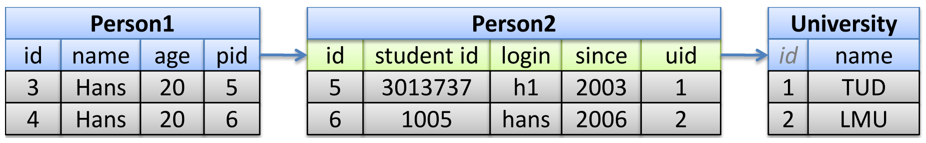
**Example 9** *Remind Example 8. Using the normalized class-role relations approach in a first step two relations will result: one for* `University` *and another one for* `Person` *and* `Role`*, as depicted in Figure 4.7. Due to normalization the* `Person` *relation is split into the relations* `Person1` *and* `Person2`*, as depicted in*



| Person | | | | | | |
|---|---|---|---|---|---|---|
| id | name | age | student id | login | since | ctx id |
| 3 | Hans | 20 | 3013737 | h1 | 2003 | 1 |
| 4 | Hans | 20 | 1005 | hans | 2006 | 2 |

| University | |
|---|---|
| id | name |
| 1 | TUD |
| 2 | LMU |

Figure 4.7: Transformation to *Class-Role Relations* of Example Scenario into Relational Representation with Exemplary Data.



**Person1**

| id | name | age | pid |
|---|---|---|---|
| 3 | Hans | 20 | 5 |
| 4 | Hans | 20 | 6 |

**Person2**

| id | student id | login | since | uid |
|---|---|---|---|---|
| 5 | 3013737 | h1 | 2003 | 1 |
| 6 | 1005 | hans | 2006 | 2 |

**University**

| id | name |
|---|---|
| 1 | TUD |
| 2 | LMU |

Figure 4.8: *Normalized Class-Role Relations* Transformation of Example Scenario.

*Figure 4.8. Notably, the same relations as for the complete-separation transformation result, which is not necessarily the case.*

## 4.3 Conceptual Architecture of DAMPF

The last Section elaborated in detail all concepts of DAMPF. This Section focuses on the architecture from a conceptual viewpoint. This Section presents a fine-grained, detailed overview of the architecture.

Figure 4.9 depicts all major parts of the conceptual architecture. Solid lines denote activities, dotted lines information usage. For example the part *runtime utilities* actively change the runtime fact base and use the information from both fact bases. Each part will be described in the next paragraphs.

Two categories can be identified: startup and runtime utilities. The original application is passed to the startup utilities, which sublimate the application and extract, as well as compare the schema from the application, as described for the step **Compare**.

The sublimation of an application can be done using a post-processor, which is run every time an application has been compiled, or a load-time weaver, which sublimates the application each time it is started. The current usual way to distribute applications to customers is by enabling the customer to download software from the World Wide Web. But there are more modern ways of software distribution already available. For example Java WebStart[1]. Applications are deployed from servers to the customers' desktop computer. Because many software companies exist, the customer has the choice, which software to use. In the near future the customer might be able to compose its software from parts developed by companies from all over the world. In such a scenario, the post-processing approach cannot be used anymore, because it requires a single developer team to provide the software. Therefore DAMPF provides a **load-time weaver**, which works independently from software vendors.

The second category of the architectural parts comprises runtime utilities. These comprise tracing of the applications event stream and utilities to store, search and restore domain objects. They correspond to step `Trace` and `React`. Storing domain objects requires information from the schema and runtime fact bases, to transform them into their relational counterpart. The transformation is done completely in Prolog. Tracing the event stream implicitly prepares the transformation. This is, because it does not just log all life-cycle and value-change events, but already derives the full state of the domain objects. A value-change event leads to the update of a fact in the runtime fact base, representing

[1]http://java.sun.com/javase/technologies/desktop/javawebstart/

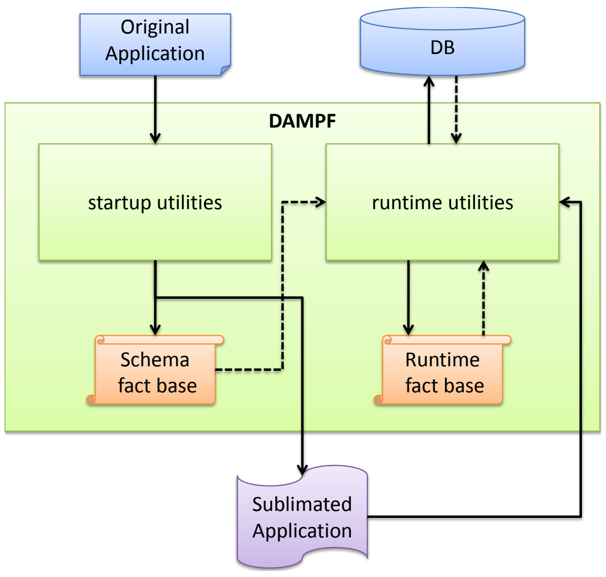


Figure 4.9: Conceptual Architecture of DAMPF.

the state of the domain object, which was effected by that event. To store the object, this fact is used. The retrieval of an objects state is not done using the *one-shot* approach, leading to few, but big performance penalties in terms of delayed execution. Instead the runtime penalties are spread over time, as the value-change and life-cycle events are. Having many small penalties, spread over time, is better than having few, big penalties, because the user is more likely to notice the second.

To search and restore domain objects, mainly the schema fact base is needed. This is because database queries need to be build, to fetch the data of the domain objects, which shall be restored. Importantly, schema changes need to be considered at this point. For example an attribute, which has been removed from the class schema, still remains in the database and thus, must explicitly **not** be fetched by corresponding queries. The runtime fact base can be used for performance optimizations, viz. as a cache in front of the database. This requires a sophisticated cache policy, which is out of scope of this thesis. Searching partially relies on the runtime fact base. How much this fact base is used, depends on the persistence strategy chosen in step **React**. Application-level persistency enables to use the runtime fact base as only source of information, because changes from other users, will not be available until the application is restarted. At application startup the runtime fact base is rebuild, thereby querying the database. Starting from instance-level persistency, down to manual persistency requires consulting the database in order to get up-to-date results. To validate search queries the schema fact base is used, because it contains the information, which attributes exist and of which type they are.

In Summary, DAMPF comprises four distinct architectural parts: the startup and runtime utilities, as well as the schema and runtime fact bases. Because the fact bases are Prolog programs, they are able to provide more information, than the information, which has been written into them.

# Chapter 5

# Implementation

This chapter elaborates on the actual implementation, based on the conceptual reflection presented in the Chapter 4. The concepts do not base on a particular programming language. Any class-based object-oriented language, extended with the roles concept, can be used as target platform, to realize DAMPF. The reference implementation presented here is based on the Java programming language and ObjectTeams/Java as extension, supporting roles as first-class constructs. Hence almost all of the following implementation details are specific to Java and ObjectTeams and might differ for other languages.

Figure 5.1 depicts a detailed overview of the architecture of DAMPF. The most course-grain parts of the architecture are the startup and runtime utilities, as well as the fact bases, which will be explained in the following.

## 5.1 Startup Utilities

The startup utilities, Sublimate, Compare and Adjust, are merged in a single implementation artifact: a so-called *Java agent*. Java agents are a technology available since Java 1.5. They support to intercept the classloading process and

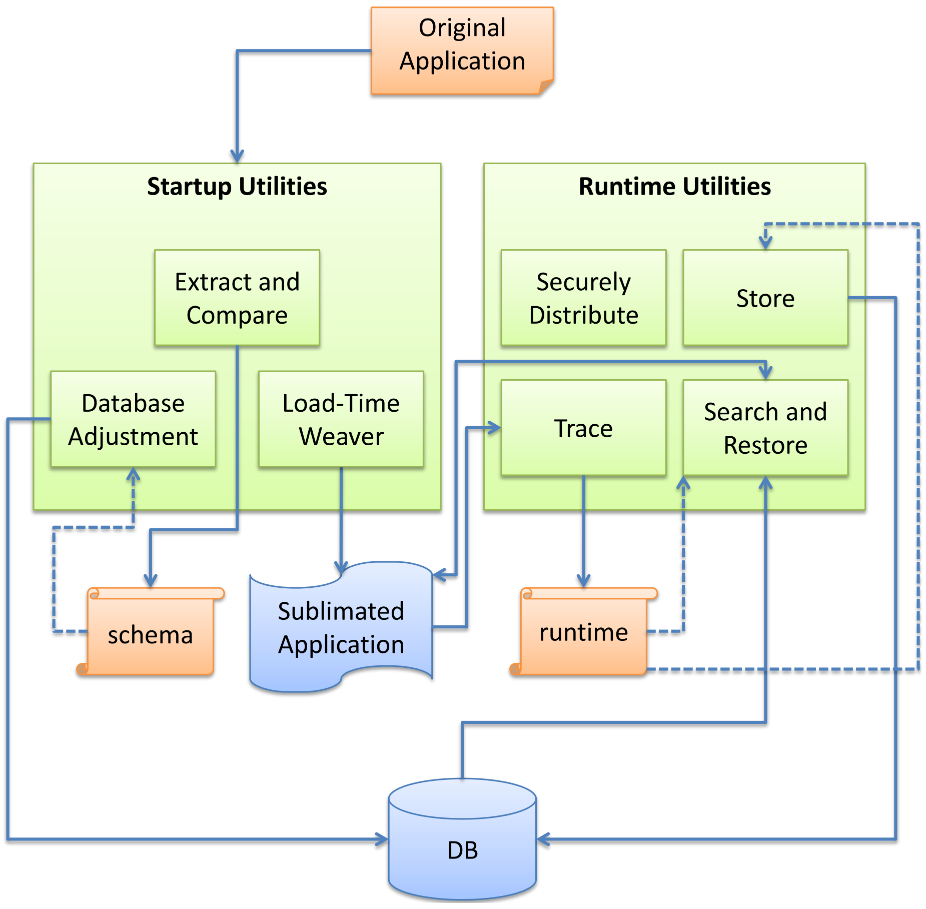


Figure 5.1: Detailed Architecture of DAMPF.

thereby to transform a class, before it is loaded. This ability fits the needs for step one, Sublimate, perfectly. Whenever a class is loaded, the Java agent is involved and is able to decide, whether and how to transform the class to be loaded. Additionally, agents can execute further code, whenever a class is loaded. This ability is used to realize step two and three, Compare and Adjust. Whenever a class is loaded, the schema fact base gets updated and, if the facts to be added already exist, compared. Based on the updated schema fact-base the Java agent also initiates the database adjustments.

Java agents have a simple structure. They are implemented as classes, which have a static `premain`-method. This method has two parameters. A String-attribute to fetch startup arguments for the agent and an instance of class `java.lang.Instrumentation`, which offers methods to add so-called transformers amongst others. The `premain`-method is executed once, before the application starts up, that is before the applications `main`-method is executed. What happens, when a class is loaded, is defined by transformers. They need to implement the interface `ClassFileTransformer` in package `java.lang.instrument`. This interface offers a single, but important method: `transform`. It takes a class loader, the current class name, an instance of type Class, a protection domain and a byte array, representing the class file, as arguments. Its purpose is to transform the byte array and to return it, when finished. Every time a class is loaded, this method will be executed.

To realize the actual transformation, third-party libraries can be used. Without them, the developer needs to manually adjust the byte array. Three different libraries exist for that purpose: Apaches BCEL[1], ASM[2] of the OW2 consortium and Shigeru Chibas Javassist[13]. All three are powerful tools for bytecode transformations. One is as powerful as the other. For the reference implementation, Javassist has been chosen, because it is most comprehensively documented.

The first task of the transformer in order to sublimate the application is, to identify, if the current class is an entity or not. As system classes do not differ from domain classes the annotation `@Entity` is provided to the application developer, to signal, that a class belongs to the domain model of the application. The transformer checks, whether this annotation exists or not. But even in small systems this leads to considerable performance penalties. The reason is that at application startup all classes, needed to run the application are loaded. This includes classes of the Java Runtime Environment and possibly third-party libraries used by the application. Checking a single class for the annotation consumes only little time. But checking thousands of classes requires seconds, thus slowing down the application startup perceptibly for the user. Excluding classes from the Java Runtime Environment only lowers the penalty for small systems, which do not use third party libraries. Because it cannot be foreseen, which libraries the application developer will use, a white list, defining those packages, which shall be checked, is used, to narrow the annotation check to application classes only. Though this test requires some time, too, it is much faster than reflecting a class, to test for an annotation. It boils down to a simple string comparison.

Classes, marked as entities, may reference other classes, not marked as entities, in different ways. Either they have an attribute, pointing to an instance

[1]http://jakarta.apache.org/bcel/
[2]http://asm.ow2.org/

of another class or they inherit from another class. Additionally, roles reference their player and context. In some of these cases, the referenced class needs to be considered as an entity, too, although it is not marked as such. Attributes, pointing to instances of other classes, do not require the other class, to be considered an entity. This is, because the referenced class potentially is a system class. If the referenced class belongs to the domain, it should be marked as such. Contrarily, inheritance requires the referenced class to be considered an entity, even if it is a system class. The reason lies in the requirement to be able to restore the object afterwards. If the class of an object is a subclass, the objects state comprises values of the superclasses attributes. These need to be persisted as well. The players and contexts of roles belong to the domain model and are marked as such.

To identify superclasses of classes, marked as entity, the class loading order needs to be examined. Creating an instance of a subclass, that is invoking the constructor of the subclass, leads to the invocation of the superclasses constructor. Notably the superclass constructor is called by the subclass constructor at its very beginning. Thus, first the superclass constructor followed by the subclass constructor is executed. In contrast, the classes are loaded the other way around. That is, first the subclass is loaded, followed by the superclass. To identify, that the current class needs to be considered as an entity, although it is not marked as such, the transformer uses a map, storing the superclasses of already loaded classes. Thus, whenever a class is loaded, which has a superclass, this fact is stored in that map. As classes, related by inheritance are always loaded in the same order, that is subclasses prior to superclasses, this map suffices.

**Example 10** *Clerks are, of course, humans, which in turn are mammals. For this reason `Mammal` is a superclass of `Human`, which in turn is a superclass of `Clerk`. Figure 5.2 depicts this example. When an instance of class `Clerk` is created, first the constructor of `Mammal` is executed, followed by the execution of the constructor of class `Human` and finally the constructor of `Clerk`. Hence, visually in accordance to the figure, instantiation goes top-down, that is from superclasses to subclasses. Class loading works exactly the other way around. First class `Clerk` is loaded, followed by class `Human` and, finally, class `Mammal`.*

As can be seen in Example 10, the classes of an inheritance hierarchy are loaded bottom up, that is starting at subclasses going up the ladder of superclasses. The reason lies in the way the virtual machine identifies classes to be loaded. The instantiation of a subclass only mentions only that subclass. Hence first the subclass is loaded. After this class has been loaded, the virtual machine knows that there is a superclass and starts to load it. Loading the superclass possibly reveals that there are further superclasses, which in turn are loaded, too. In summary, first all classes of an inheritance hierarchy are loaded bottom up and then the constructors are executed top down.

A reference to the runtime utilities is injected into each identified entity. This reference forms a listener for the events fired by the sublimated application. Indeed it is not a real listener, as the entities will send messages to this reference, to signal value changes and life cycle events. Additionally an integer field for an object identifier is added to each entity. This identifier is required internally of DAMPF to relate objects to each other.

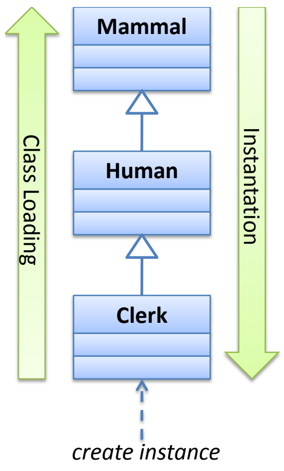


Figure 5.2: Class Loading and Instantiation of Subclasses.

If a class has been identified as an entity, its schema is introspected prior to the sublimation. Inheritance relations are directly available using the Java Reflection API or the reflection API of the chosen byte code modifier library. The same holds for attributes, except they are of collection type. Java allows defining arrays and provides a collection API. Both types need special care, as has been described in Subsection 3.3.1 in Chapter 3. Arrays are easier to handle than collections. Indeed they can be considered as constrained collections, as will be shown later. Java Collections are more than sets of values, because they provide further properties. For example, the collection type `Set` prohibits duplicate values, whereas type `List` allows for them. Prior to Java version 5, collections could contain values of any type. Newer versions still support this type of collections. It is for example possible, to put a string, an integer and some user-defined object into a collection. Generics constrain the possible types of the values of a collection and are available since Java version 5. Arrays are special collections, because they can be considered as lists, viz. allowing duplicates, constrained to allow only a single type for its values. Mixing values of different, unrelated types in a collection is error-prone and can be seen as a bad programming habit. DAMPF supports generic collections and arrays. If an attribute of such a type is identified, it is handled, like described in Subsection 3.3.1.

But to identify the generic type of a collection, a little obstacle of Java is in the way. Although in source code the type description contains the generic parameter, this parameter is hidden in the bytecode produced by the compiler. Fortunately an extension to the Java Language Specification[43] describes where the compiler puts this information. Generic parameters can be extracted from signature attributes in the class pool.

Sometimes domain objects have attributes, which should explicitly not be persisted. Usually this is due to bad programming habits, leading to interweaved domain and system code. To enable the application developer to exclude certain attributes from persistency, the annotation `@Ignore` is provided.

Each time a class is loaded and has been identified as an entity, its structure is written into the schema fact base. The existence of the class, all attributes,

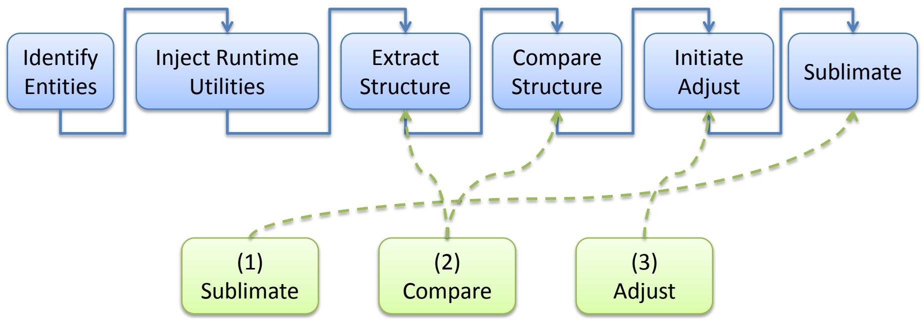


Figure 5.3: Process of Startup Utilities and Connection to Conceptual Architecture.

except those marked with `@Ignore`, references to other classes, the subclass relation and if the class is a role or context is examined and noted. The first part of Section 5.2 presents all resulting predicates in the schema fact base.

If a predicate for the structural element already exists in the schema fact base, it stays as it is. A change is identified, if some of the arguments of the predicate differ. For example, attributes are identified by their name and class and can differ in their type or position. Each identified change is written to the schema fact base.

If schema changes have been identified, which require changes to the database, the handling of them is initiated by the Java agent, too. The application of the changes is accomplished by the runtime utilities.

Finally, the currently processed class is transformed in order to sublimate it. All constructors are enhanced with calls to the runtime utilities, signaling the creation of a new object. All other methods are analyzed in regard to the attributes they use. At the end of each method a call to the runtime utilities is added, signaling which attributes have been changed along with the new values of them.

Lifter methods of roles are adjusted, too, signaling the lifting of a new role. It is important to note, that the lifting of a role is not necessarily the creation of a new role. Lifting may also lead to the reuse of an already existing role! The creation of roles involves the execution of their constructor, which is adjusted like a usual constructor, but signaling the creation of a new role. The same holds for contexts. Some methods, which are internal to ObjectTeams, are excluded from sublimation. These methods can be identified by their name, because all begin with `_OT$`.

In summary the Java agent first identifies, whether the current class is an entity, injects the runtime utilities, inspects the structure of the class and writes it to schema fact base, compares with already existing structure and writes changes to schema fact base, too, initiates adjustments to the database and finally sublimates classes to signal events. Figure 5.3 depicts this process and how it is connected with the conceptual architecture.

## 5.2 Schema and Runtime Fact-Base

Two fact bases are used to store information about the application. The schema fact base contains all structural information. Runtime data, like the current states of domain objects, is contained in the runtime fact base. Both are explained in detail in the following.

**Schema Fact-Base.** The fact base is based on the programming language Prolog, which is a declarative, logic programming language. The fundament of Prolog programs are horn clauses, that is clauses, with at most one positive literal. Horn clauses can be represented as logical implications. Their typical form is (a ∧ b ∧ c ∧ ...) → z. Robert Kowalski proposed to see horn clauses as procedures[41] and thereby laid the foundation for logic programming. In [42], Kowalski gives a good summary on the history of Prolog.

A Prolog program is a set of horn clauses. Syntactically the implication is drawn the other way around, viz. z ← (a ∧ b ...). The logical and-operator is represented by a comma, the implication ← is represented by `:-`. In consequence, a typical Prolog clause has the form `head :- body`. Such clauses are called *rules.* Rules with an empty body are called *facts* and are written `fact.`. To derive further information from facts, rules can be used.

The benefit of using Prolog for the schema and runtime fact-bases is exactly this ability, to derive further information using rules. Other languages, providing the same benefit exist: Datalog and Frame-Logic.

From a syntactical point of view, Datalog[12] is a subset of Prolog. Nevertheless Prolog programs differ from Datalog programs in their semantics. The order of rules and facts is of great importance in Prolog, but of no importance in Datalog. Furthermore Prolog programs cannot guarantee termination, but Datalog programs can. Though this is very beneficial, Datalog has some drawbacks, too. Most notably, Datalog programs need to be stratified. A set of horn clauses is called stratified, if a function exists, which maps all predicates to numbers, adhering to the following rules. Each predicate, which positively depends on another predicate, must be mapped to a bigger or equal number, than the referenced predicate. If a predicate negatively depends on another predicate, it needs to have a bigger number, than the referenced predicate. The guaranteed termination is one of the consequences of the stratification.

Frame-Logic looks more powerful than Prolog. It allows using object-oriented concepts in a logic-based environment[40]. Nevertheless, frame-logic programs can be transformed into Prolog programs and even into Datalog programs. Hence, frame-logic is not more powerful than Prolog, but more convenient to use in DAMPF.

In comparison, using Datalog instead of Prolog leads to more implementation effort. It is time-consuming to stratify clauses, which is required in Datalog. The benefit of knowing that stratified rules terminate definitely is not worth the effort. Furthermore no implementation of Datalog is publicly available, which can be used in a Java environment. Frame-Logic would be a good choice for the fact-bases, too, but suffers the same problem as Datalog. No implementation usable in a Java environment exists.

The facts of the schema fact-based describe the applications schema and are listed in Table 5.1. The existence of each class is express by `isClass/1`. An attribute *a*, belonging to class *C*, of type *t* is represented by the predicate

| | |
|---|---|
| `isClass(C)` | class `C` exists |
| `isRole(R,C,P)` | role `R` exists in context `C` and is bound to player `P` |
| `isContext(C)` | context `C` exists |
| `hasAttribute(C,a,t,p)` | class `C` has attribute `a` of type `t` at position `p`. |
| `hasStaticAttribute/4` | same as `hasAttribute/4`, but for static attributes |
| `references(C,a,D,b)` | class `C`'s attribute `a` references class `D`'s attribute `b` (usually unknown) |
| `subclasses(A,B)` | class `A` is a direct subclass of class `B` |

Table 5.1: Predicates Used in Schema Fact Base.

`hasAttribute(C,a,t,Position)`. The position of the attribute is required internally, in order to identify attribute-value pairs in combination with the runtime fact-base. The visibility of an attribute is not relevant to DAMPF, because their values are to be stored regardless of their visibility. Static attributes are expressed like instance attributes, but using another predicate: `hasStaticAttribute(C,a,t,Position)`. To denote, that class `Super` is the superclass of class `Sub`, the predicate `subclasses(Sub,Super)` is used. To determine transitively, if a class is a subclass of another class, the predicate `superClass(Sub,Super)` was implemented.

```
superClass(Sub,Super) :- subclasses(Sub,Super).
superClass(Sub,Super) :- subclasses(Sub,X),
                         superClass(X,Super).
```

The references between classes, viz. attributes pointing to instances of classes, are expressed using the `references/4` predicate. To denote for example, that class $C_1$ has an attribute $a_1$ of type $C_2$, thus pointing to an instance of $C_2$, the predicate is `references(`$C_1$`,`$a_1$`,`$C_2$`,'')`. The fourth argument can be used to define a specific target attribute. In order to express arrays or generic collections, the type name of the attribute needs to contain this additional information.

The existence of roles and contexts is represented by `isRole` and `isContext`. If roles and contexts are realized as classes, this fact will be reflected by an additional `isClass/1` in the schema fact base. The second argument of `isRole/2` points to the player type of the role and is an optimization for the ObjectTeams implementation. The current player is referenced in the runtime fact base and the current context as well. Because ObjectTeams only supports a single player type for a role, DAMPF uses this simplification to optimize the derivation of further information.

Additionally schema changes are noted in the schema fact base. Table 5.2 lists all predicates, expressing such changes.

For classes, roles and contexts predicates, expressing that they were added or removed exist. These are the six predicates `AddedClass`, `RemovedClass`, `AddedRole`, `RemovedRole`, `AddedContext` and `RemovedContext`. Renamings cannot be detected, because the depend on the intend of the developer. Renaming a class can also be seen as removal of the old class and addition of the new one. Based on the structure of the classes, the old and the new class can be compared, which allows to infer a similarity level. Using heuristics it is possible to decide, whether a rename has happened or not. Nevertheless, using heuristics leads to false positives. If, for example, the developer removed a class, and created a new one, which has a very similar structure, but completely different semantics, the heuristic-based decision was false. In consequence the data of the old class is used for the new. Thus data from completely different objects will be used, leading to problems in the application, which cannot be anticipated. Hence, using heuristics is not a possibility in this approach. In consequence, renamings cannot be detected.

Classes, roles and contexts can change their superclass, too. It might happen that a class, which formerly had a superclass, now has none. Or a class, which formerly had no superclass, now has one. Finally, a class, which formerly had a superclass, might now have another superclass. The third type of changes to the inheritance hierarchy includes special cases. A new class can be introduced between two formerly connected ones. That is, a class A, having a superclass B, gets a new superclass C, which has B as superclass. In contrast, a class might be removed from a hierarchy, so that its subclass afterwards has the superclass of its former superclass. In other words, a class A, having a superclass B, which has a superclass C, is changed in a way, that it now has superclass C. B is either completely removed from the hierarchy, that is it does not have C any longer as its superclass, or it simply loses its state as superclass of A. Further changes to inheritance hierarchy can be identified, but are left for future work. In the schema fact base, changes to the inheritance hierarchy are expressed by the

| | |
|---|---|
| `AddedClass(C)` | class `C` has been added |
| `RemovedClass(C)` | class `C` has been removed |
| `AddedRole(R,C,P)` | role `R` has been added to context `C` and is bound to player `P` |
| `RemovedRole(R,C,P)` | role `R`, bound to player `P` has been removed from context `C` |
| `AddedContext(C)` | context `C` has been added |
| `RemovedContext(C)` | context `C` has been removed |
| `AttachedSuperclass(C,SC)` | class `C` now has superclass `SC` |
| `DetachedSuperclass(C,SC)` | class `C` no longer has superclass `SC` |
| `AddedAttribute(C,N,T,P)` | attribute, with name `N` in class `C`, with type `T` at position `P` has been added |
| `RemovedAttribute(C,N,T,P)` | attribute, with name `N` in class `C`, with type `T` at position `P` has been removed |
| `ChangedAttribute(C,N,T,P)` | attribute, with name `N` in class `C`, with type `T` now has position `P` |
| `ChangedPlayer(R,PC)` | role `R` now is bound to players of type `PC` |
| `AddedReference(SC,SA,TC,TA)` | rule, inferring added references |
| `RemovedReference(SC,SA,TC,TA)` | rule, inferring removed references |

Table 5.2: Predicates Denoting Schema Changes.

predicates `DetachedSuperclass`, `AttachedSuperclass`. The special cases do not need separate predicates, as they can be inferred from the two predicates.

Attributes can be added, removed and might change their name, type or position. To note an addition or removal of an attribute the predicates `Added-Attribute` and `RemovedAttribute` are used. Changes to attributes cannot always be identified. This is due to the same reasons, like for classes, contexts and roles. If an attribute has been changed or not, depends on the developers intend. If the name of an attribute changed, but type and position remain unchanged, it is likely, that the developer really renamed the attribute. Nevertheless, it is possible, that the developer removed an attribute and added a new one at the same position, with the same type. The same holds, if only the type has been changed, but not the name and position. If only the position has changed, the developer removed the attributed and added it at another position. It is still the same attribute. Thus, changes to the position can be identified. The predicate `ChangedAttribute` is used for that purpose.

The references between classes, context and roles might change, too. As described in Chapter 3, references boil down to foreign key constraints in the database schema. The removal of a reference implicitly contains the removal of an attribute, namely the source attribute of the reference. The removal of the attribute is noted by `RemovedAttribute`, as mentioned in the last paragraph. The removal of the reference can be inferred from these facts, by the Prolog rule `RemovedReference`. A newly added reference can also be inferred, as it is a direct consequence of a newly added attribute, whose type is a domain class. The Prolog rule `AddedReference` is used for that purpose. Changes to references are to be seen as removals of the old and addition of the new reference. Special to references is solely the connection between two classes, roles or contexts. All other changes are changes to the involved attributes. The predicates inferring addition and removal of references are presented in the following:

```
RemovedReference(SrcClass, SrcAttr, TgtClass, TgtAttr) :-
   references(SrcClass, SrcAttr, TgtClass, TgtAttr),
   RemovedAttribute(SrcClass, SrcAttr, _, _).

AddedReference(SrcClass, SrcAttr, TgtClass, _) :-
   AddedAttribute(SrcClass,SrcAttr,TgtClass,_).
```

Finally, changes to the bindings of roles need to be considered. The programming language extension used for the implementation of the prototype, that is ObjectTeams, fixates the player and context of a role statically. In the general case, these changes happen at runtime. A role might change the type of its players, which is denoted by `ChangedPlayer`. The binding of a role to its context cannot be removed, without adding the role to another context, because a role needs to be bound to a context at any time. If a role has been moved from one context to another cannot be detected, due to the same reasons like renamings of them cannot. Hence, only the predicates `AddedRole` and `RemovedRole` are used.

These changes need to be considered at the applications startup and at runtime. To allow the application to change its schema at runtime, database adjustments need to be performable at runtime, too. For this reason, the handling of database adjustments is part of the runtime utilities. Furthermore, these changes need to be considered, when storing, searching and restoring do-

| | |
|---|---|
| `instanceof(C,S)` | an instance of class `C` with state S exists |
| `sameInstance(`$C_1$`,`$C_2$`,`$ID_1$`,`$ID_2$ | the instances with $ID_1$ and $ID_2$ of classes $C_1$ and $C_2$ are the same |
| `contextState(C,ID,S)` | the context of type `C` with the id `ID` is either active or not |

Table 5.3: Predicates Used in Runtime Fact Base.

main objects. Finally the send and receive feature needs to take care of the changes. Because all of these features are used at runtime, they are handled by the runtime utilities, too.

**Runtime Fact-Base.** The sublimated application signals value changes as well as life cycle events. Both events affect the runtime fact base. The fact base contains the current state of the domain objects. Table 5.3 summarizes the predicates, which are used in the runtime fact-base.

The predicate `instanceof(C,S)` expresses, that an instance of class C exists and that it has the state S. The state is represented as a comma separated string of values. Change events are directly reflected in that string.

**Example 11** *Students have an identifier and a name and can be represented as class* `Student`*, with the attributes* `studentid` *and* `name`*. Furthermore the startup utilities will add an object id and create the following predicates in the schema fact base:*

```
isClass('Student').
hasAttribute('Student','studentid','int',0).
hasAttribute('Student','name','java.lang.String',1).
hasAttribute('Student','__DAMPF__oid','int',2).
```

*The instantiation of a student with student id 300, whose name is John, leads to a sequence of events. First a* ***new object event*** *is fired, followed by multiple* ***value change events****, signaling the value change of each attribute. The runtime fact base emerges in the following sequence:*

```
instanceof('Student',[-,-,1]).
instanceof('Student',[300,-,1]).
instanceof('Student',[300,'John',1]).
```

*The* ***new object event*** *leads to the first predicate. The object id is already set, all other values are undefined. The following* ***value change events*** *for the student identifier and the name lead to the subsequent predicates in that order.*

The object id is used to index the predicates of the runtime fact base and to relate instances to each other. Values of attributes, referencing objects, will be represented using the object id of the referenced objects. In order to access the object id the following rule is used:

```
getInstanceof(Class,OID,Values) :-
   instanceof(Class,Values),
   hasAttribute(Class,'__DAMPF_oid__',_,POS),
   nth0(POS,Values,OID).
```

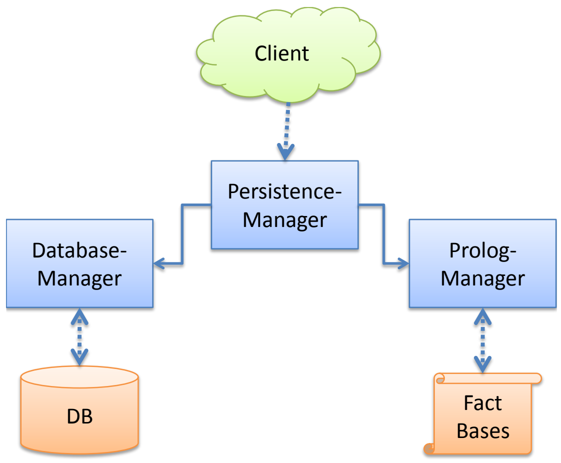


Figure 5.4: Architecture of Runtime Utilities.

The `nth0` predicate is provided by Prolog and allows accessing the $n^{th}$ value, starting at 0, in a list.

To keep track on the state of objects, whose classes are part of an inheritance hierarchy, the partial states are traced. For each class up the inheritance hierarchy a separate state is logged. Additionally the relation of all these states is expressed using the `sameInstance` predicate.

Because ObjectTeams does not allow roles to switch players or contexts at runtime, these features have not been implemented. Nevertheless, ObjectTeams supports to activate and deactivate contexts. Hence the current state needs to be tracked. For this purpose, the `contextState` predicate can be used. It takes three arguments: the context' class name, the object id of the context and a boolean value, expressing whether the context is currently active or not.

## 5.3 Runtime Utilities

In order to trace and react on the events fired by the sublimated application runtime utilities are required. Furthermore search and restore, as well as store functionality needs to be provided. Finally, the distribution and thereby security feature is realized as part of the runtime utilities. Central to these utilities is a single class — the `PersistenceManager`.

The PersistenceManager needs to communicate with the database and the Prolog fact bases. In order to clearly separate these access layers two classes have been extracted from the `PersistenceManager`: `DatabaseManager` and `PrologManager`. Figure 5.4 depicts this architecture. The DatabaseManager is responsible for the communication with the database and the PrologManager for communicating with the fact bases.

**Database Adjustments.** As mentioned earlier, required adjustments to the database are triggered by the startup utilities, but accomplished by the runtime utilities. The `DatabaseManager` provides a dedicated entry method for that purpose: `processSchema()`. It is not directly accessible, but through the

| **name** | **type** | **cause** |
|---|---|---|
| `fireNewInstance` | life cycle | instantiation |
| `fireInstanceRemoved` | life cycle | explicit remove |
| `fireNewRole` | life cycle | instantiation |
| `fireRoleLifted` | life cycle | lifting |
| `fireContextActivated` | life cycle | de/activation |
| `fireValueChange` | - | method execution |

Table 5.4: Events Fired by the Sublimated Application.

`PersistenceManager`. Depending on the chosen persistence strategy, the persistence manager decides, when `processSchema()` is called. Manual, value and instance level persistency leads to its execution whenever the user manually triggers to persist the domain objects. This is because the applications schema might change at runtime, due to the dynamic features of roles. Admittedly, ObjectTeams does not allow runtime changes to the schema, other languages might do. Application level persistency, that is persisting every time the application is shut down, does not need to keep track on changes at runtime. Hence, the schema changes are applied once at application startup.

The database adjustments mainly lead to the creation of new attributes or relations. Removed attributes or classes do not lead to their removal in the database schema by default. Only, if the application developer marked the application to be a milestone release, those elements are removed. But sometimes attributes cannot be deleted directly. If there is an index or foreign key pointing to them, these elements need to be removed first. In case many attributes have to be removed, their order is important. Because the whole schema is available in Prolog, the order of classes along their keys can be resolved by a Prolog rule, which is called `resolveFKs`. This rule and all required helping rules can be found in the appendix.

The reference resolution using `resolveFKs` is also important for persisting domain objects. If multiple domain objects have to be stored, they need to be stored in the right order. But before any object can be stored, its state needs to be revealed, which is the task of step 4, **Trace**.

**The Event Stream.** The functionality of **Trace** is merged into the `PersistenceManager`. At sublimation a reference to that manager is injected into each entity class. Firing life cycle or value change events boils down to calling the appropriate method of the `PersistenceManager`. The events are summarized in Table 5.4.

The `newInstance` event first leads to the creation of a new object id. The `PrologManager` is than asked, to add a new `instanceof` fact for the corresponding class with the newly created object id to the runtime fact base. If the class, which fired the event, is a not the runtime class of the object, it is a superclass and the `PrologManager` is forced to add a `sameInstance` fact to the runtime fact base, too. If the runtime class and the class, which fired the event, are the same, the PrologManager needs to check if there are `sameInstance` facts and properly update them, like shown in Example 16.

If `fireInstanceRemoved` is executed, the corresponding `instanceof` and `sameInstance` facts are removed from the runtime fact base. Java does not support to explicitly delete objects, instead the garbage collector decides, whether an object is to be deleted or not. To enable the application developer to explicitly remove objects, the annotation `@Remove` is provided, which is to be used to mark a method of the domain class, the developer wants to use for that purpose.

The creation of a new role fires a `newRole` event. The PersistenceManager reacts almost like a new object is created, but fetches the current player and context, too. The resulting `instanceof` points to them by their object id. In case of a `roleLifted` event, the runtime fact base needs not to be changed, but its recognition might be useful for future work. The activation or deactivation of a context leads to the execution of `fireContextActivated`, which in turn is reflected by the change of the `contextState` fact in the runtime fact base.

Finally, the change of an attributes value leads to the execution of `fireValueChanged`. Depending on whether the attribute is a collection or not, different actions follow. If the attribute is not a collection, the new value is send to Prolog, which exchanges the old against the new value. Else, the values of the collection are compared, to identify, which values have been removed and which have been added. Again, Prolog is used for that purpose.

**Storing Domain Objects.** Essential to DAMPF is the persistency mechanism. The `PersistenceManager` provides a method `persist()` for that purpose. If the application developer decided on manual persistency he needs to invoke this method on his own. But he shall not need to call this method directly, because this would lead to a very tight coupling of DAMPF to the application. Instead an annotation `@Store` is provided. The application developer adds this annotation to one if his methods, if he wants to use that method to trigger persistency.

The method `persist()` delegates to the `DatabaseManager`, which in turn uses the methods `processSchema()` and `processRuntime()`. As already mentioned, in case of application level persistency, `processSchema()` is omitted at this place, but executed once at application startup. The execution of `processRuntime()` first uses the PrologManager to get the current states of instances. Next the ordered list of classes is retrieved. Using this order the instance are either added to the database or updated in the database. Whether an instance needs to be updated or added is examined by the `PrologManager`, which in turn uses the `DatabaseManager` to get the instance states from the database. An important aspect of adding or updating the representatives of objects in the database is, to take care of schema changes. The `INSERT` and `UPDATE` queries are created in accordance to the change information, available in the schema fact base.

**Search and Restore.** Besides storing domain objects to the database, they need to be restoreable, too. Developers need to express, which domain objects they want to be restored, but they do not know about the object id, which is internal to DAMPF. Therefore a search criteria API is provided. The prototype only contains a very simple implementation. To search and restore a domain object, the PersistenceManager offers a method `getObject`, taking the class name and a set of search criteria as arguments. A search criterion is a Map, whose keys and values are strings. The keys represented attribute names, the values represent accordingly the expected values for the attribute. If the map contains multiple keys, DAMPF interprets them as conjunctively connected. If multiple maps are sent to `getObject`, viz. the list comprises multiple maps, the criteria are connected disjunctively.

**Example 12** *Persons have a first name and a surname and thus can be represented by class `Person` with the attributes `firstname` and `surname`. To search and restore for persons, whose surname is Meier, the method `getObject()` is to be invoked with 'Person' as first argument and the map `['surname' = 'Meier']` as second argument. To search for John Smith, the maps needs to be `['firstname' = 'John', 'surname' = 'Smith']`. To search for persons who either have John as first name or Smith as surname, the following list of maps needs to be send to `getObject: ['firstname' = 'John'], ['surname' = 'Smith']`.*

A considerable part of the work to restore found objects is done in Prolog. All required tuples are fetched from the database and are added to or updated in the runtime fact base in such a way that single `instanceof` facts for each domain object result. Multiple `instanceof` and `sameInstance` facts result, in case the objects class is part of an inheritance hierarchy. The creation of an object in general is a simple task. It does not need any more than calling a constructor. But calling one of the provided constructors affects the state of the object to be created. Furthermore it is likely, that all provided constructors have arguments, viz. the default constructor with no arguments is not available. Finally, after instantiation, the values cannot freely be set. Private attributes prohibit setting them from outside the class. The solution of DAMPF is to add creators and initializers to each entity, when it is sublimated.

Creator methods are static. Their purpose is to create a new object using one of the provided constructors. After the object has been created, the initializer is triggered. Initializers set values according to their arguments and call their super-method if there is a superclass. The `PersistenceManager` gets the expected state from the `PrologManager` and sends it to the creator, who in turn sends it to the initializer. Because both, creators and initializers, are woven into each entity class, they are allowed to set private attributes.

Retrieving the data for domain objects from the database needs to consider changes to the schema. Notably, the used SELECT statements need to by adjusted, according to the changes identified, like shown in Example 13.

**Example 13** *The central domain concept of a system for a medical practice is the patient. In the first version of the system, patients have a `name`, an `employer` and a health insurance company, denoted by `hic`. In the second version, the employer is removed from `Patient`, but an attribute `illnesses` is added, pointing to a set of illnesses. In addition class `Illness` is added, which has a `name` and a `description`.*

*The relational schema for the first version consists of a relation `patient` with four attributes `id`, `name`, `employer` and `hic`. To fetch patient John Smith in the first version, the following SELECT-query suffices:*

```
SELECT id, name, employer, hic
  FROM Patient
 WHERE name LIKE 'John Smith'
```

*The schema for the second version has two additional relations: first, the relation* `Patient_Illnesses`*, with the attributes* `patient_id` *and* `illness_id`*. Second the relation* `Illnesses`*, with the attributes* `id`*,* `name` *and* `description`*. Furthermore two foreign key constraints are added to the schema. The first connects* `Patient_Illnesses` *with* `Patient`*, using the* `patient_id` *and the* `id` *of* `Patient`*, the second connects* `Patient_Illnesses` *with* `Illness`*, using the* `illness_id` *and the* `id` *attribute of* `Illness`*. Notably, the attribute* `employer` *of* `Patient` *remains in the database!*

*To fetch the patient John Smith in the second version, the SELECT-query needs to be adjusted. The new query looks as follows:*

```
SELECT p.id, p.name, p.hic, i.id, i.name as i_name, i.description
  FROM patient p, patient_illness pi, illness i
 WHERE pi.patient_id = p.id and
       pi.illness_id = i.id and
       p.name LIKE 'John Smith'.
```

**Send and Receive.** The ability to distribute domain objects is one of the main features of DAMPF. From a technical point of view, lots of technologies for distribution exist. For example plain sockets can be used, to create a channel between to processing units. But usually the infrastructure of application containers, like application servers, is used. Typical Java Application Servers provide various means to distribute objects. OSGi [3] is a well known application container, too. The extension R-OSGi[4] offers facilities specifically for distribution. The focus of DAMPF is not to provide a new technology to send and receive data, but a new concept on how to distribute domain objects.

What DAMPF provides, is a novel serialization technique, which is inherent to the whole approach. Domain objects are represented as a set of Prolog facts. These facts are textual and do not depend on a specific order. To send a domain object all its Prolog facts are packaged. The transfer of such a package boils to transferring streams of characters.

To provide the application developer means to use the transmission technique of their choice, DAMPF provides two annotations, `@Sender` and `@Receiver`, which are to be applied on methods of the application. Methods marked with `@Sender` need to have at least one argument, representing the domain object to be send. DAMPF injects code, which transforms the argument into its string representative. If the application developer wants to, for each domain object a separate method can be implemented, but a single sender method suffices, too. Receiver methods follow the same principle. They need to have an argument of string type, representing the textual representation of a domain object. DAMPF injects code, to insert the domain object into the receiving application. The actual transmission code uses these sender and receiver methods to serialize and deserialize the domain objects. Notably, the transmission code only covers the transmission of the character streams. How domain objects are serialized, deserialized and inserted into the running, receiving application is accomplished by DAMPF.

Special about the receiver methods is, that they handle domain objects, which differ from the domain model of the current application. They insert

[3]http://www.osgi.org
[4]http://r-osgi.sf.net

all information into the Prolog fact bases. Those parts, which conform to the current domain model, are inserted into the database, too. When such a domain object is send again, the complete data set of the domain object is updated by the data from the database and serialized. This allows systems, with partially different domain models to work together and takes the burden of complex conversions from the developers.

Context-based security is an implicitly available feature. Systems, which shall not see the whole data, will only see the data, which corresponds to their domain model. Thus, to realize context-based security, the systems domain models need to be created accordingly. Complex code, to hide parts of the data, gets superseded by DAMPF.

## 5.4 Normalization

As examined in Section 3.2, a relational database schema can be classified by the normal form it adheres. An important discipline in database technology is, to transform relational schemata into higher order normal forms, which is called normalization. Most schemata, which are in practical use, adhere to at most the third normal form. Higher normal forms still have an academic status.

Automatic decomposition of relation schemata into normal forms has been thoroughly investigated. Ceri and Gottlob[11] present an approach for the decomposition of relational schemata into BCNF, which is based on algorithms, specialized on the parts of the overall algorithm. They base on Bernstein's approach[7] for the decomposition into 3NF, use Lucchesi's and Osborne's algorithm[44] to identify the keys of relations and finally decomposition into BCNF using Tsou and Fischer's approach[65]. Diederich and Milton present another approach[21] to decompose relations into 3NF. Furthermore Grahne and Räihä developed an algorithm[28] for decomposition into 4NF, which considers multivalued dependencies, too. Many more approaches exist.

Notably, approaches focusing on decomposition of relational schemata base on knowing all functional dependencies. Approaches to automatically derive these dependencies are a well discussed research topic, too. The approaches of Huhtala et al.[37], Savnik and Flach[59] and Mannila and Räihä[45] are just some of them.

Central to the first three normal forms is the resolution of functional dependencies[2]. As described in Chapter 3, a functional dependency is a relation between data of different attribute sets of the same relation. If for all tuples of a relation, the values of the attribute set A always occur with the same values of attribute set B, a functional dependency A→B exists. Hence, both runtime data and the schema description are required to identify such dependencies. Because functional dependencies are defined between attribute sets, which are allowed to overlap, vast amounts of potential functional dependencies exist. Example 14 shows, how many dependencies exist in a relation with four attributes.

**Example 14** *A relation with four attributes* `a`, `b`, `c` *and* `d` *comprises 15 attribute sets:* `a`, `b`, `c`, `d`, `ab`, `ac`, `ad`, `bc`, `bd`, `cd`, `abc`, `abd`, `acd`, `bcd` *and* `abcd`. *The number of functional dependencies accords to the number of elements of the Cartesian product of this set, viz. 15 times 15, thus 225.*

In general the number of different attribute sets of a relation is based on the number of elements of the power set of all attributes, which is defined as $\mid \wp(AS) \mid = 2^{|AS|}$. The definition of the power set includes the empty set, which is not of interest for functional dependencies. Hence, the number of possible functional dependencies of a relation with n attributes is: $(2^n - 1)^2 = 2^{2n} - 2^{n+1} + 1$. In consequence, a relation with 5 attributes leads to $2^{10} - 2^6 + 1 = 961$, 6 attributes lead to approximately 4,000 and 10 attributes to more than a million possible dependencies. Each possible dependency needs to be checked in regard to the values of the attribute set. Current personal computers require minutes to accomplish this task, even with small sets of data.

Many combinations of attribute sets do not need to be investigated. A set of general rules to identify functional dependencies exist. First, every attribute set obviously depends on itself. Hence, all combinations AS→AS do not need to be investigated. Second, if an attribute is unique, that is all values differ from each other, it will always be the source of a functional dependency. This is, because for every value of the unique attribute only one corresponding value set exists. Third, static attributes, that is all values are the same, will always be the target of a functional dependency. This is, because it does not matter which attribute set is the source of such a functional dependency, as all value sets will imply the same value. Forth and finally, if a functional dependency $AS_1 \rightarrow AS_2$ exists, each superset of $AS_1$ will imply $AS_2$, too. Formally, the following functional dependencies exist for a relation `R` by definition:

1. $A \rightarrow A \mid A \in \wp(R)$
2. $A \rightarrow B \mid A,B \in \wp(R) \land \text{unique}(A)$
3. $A \rightarrow B \mid A,B \in \wp(R) \land \text{static}(B)$
4. $A \rightarrow B \mid A,B \in \wp(R) \land C \rightarrow B \land C \subset A$

The forth rule can be applied to the former three and thereby further narrows the amount of possible functional dependencies. The first rule covers $2^n - 1$ dependencies for a relation with n attributes. The second rule covers $2^n - 1$ dependencies per unique attribute set. The same holds for the third rule, but in regard to the number of static attribute sets. Interestingly, the combination of the first and forth rule covers all trivial functional dependencies.

Nevertheless, although using these rules considerably reduces the number of possible functional dependencies, investigating the remaining ones still requires time in the magnitude of minutes.

To check, whether a possible functional dependency qualifies or not, each value combination in accordance to the attribute sets of the dependency needs to be examined. DAMPF is using Prolog to identify and check these dependencies. To check a dependency, the values are seen as a function. If the range of this function only contains single values, the dependency qualifies. Possible dependencies are declared straight forward in Prolog, using the four general rules along with the definition of the powerset.

To transform a relation into first normal form, every attribute needs to be flattened. Thus, if an attribute is a relation itself, the attribute is to be transformed into the set of its attributes. Knowing which functional dependencies exist, allows transforming a relation into the second normal form. As described in Section 3.2, the second normal form requires that no non-prime attribute is

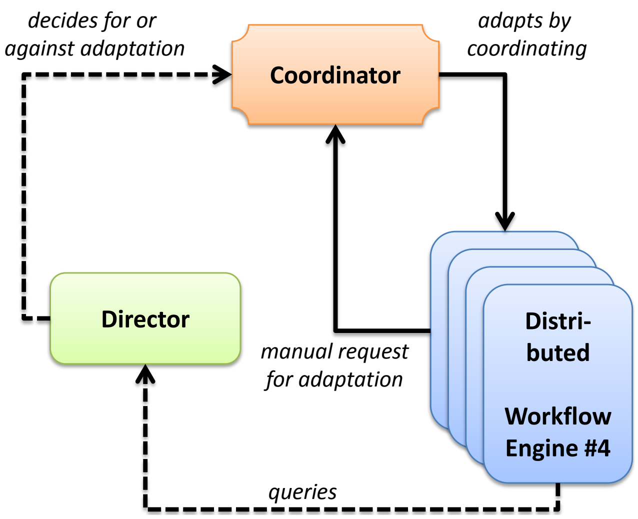

decides for or against adaptation
Coordinator
adapts by coordinating
Director
manual request for adaptation
Distri-buted Workflow Engine #4
queries

Figure 5.5: Dynamic Adaptation of Workflows in OSPP.

functionally dependent on only part of the primary key. It does not pose a violation, if a dependency only transitively provides a fact about the key. Hence the identified functional dependencies need to be checked in this regard. To transform a relation into third normal form, it needs to be checked, that all functional dependencies starting from non-prime attributes reference the whole key and nothing but the key. Both checks can be implemented straight forward in Prolog.

Finally, normalization requires relations to be split if necessary. The identified dependencies define how to split them. Each violating dependency is potentially only a subordinate dependency. The highest violating dependency defines, that the union of its attribute sets should be extracted as a separate relation.

Unfortunately the identification of dependencies is too time-consuming. Due to this reason, normalization is not integrated into DAMPF, but available as a separate set of Prolog rules, which are listed in the appendix.

## 5.5 Evaluation

In order to evaluate the usability and practicability of DAMPF, a system, which needs to persist domain objects in a distributed environment, is required. Furthermore, this system should utilize roles as extension to the object-oriented paradigm. Because the prototype has been implemented using ObjectTeams, the system needs to base on ObjectTeams, too.

A medium sized system of approximately 80.000 lines of code, fulfilling these requirements is the Open Service Process Platform[31] (OSPP), which is developed at the University of Technology Dresden. Therefore OSPP has been chosen, to evaluate the prototype of DAMPF.

OSPP is a distributed, multi-purpose workflow engine and provides an infrastructure for service oriented architectures. Its key features are its support for arbitrary extensions, the integrated solution for the design and execution of workflows and, most importantly, its adaptivity. OSPP allows designing workflows, which can be adapted at runtime. The requirement for runtime adaptivity emerges from the nature of distributed systems. A distributed workflow, viz. a workflow, whose tasks are performed by more than one workflow engine, can easily fail, for example due to non-availability of an engine. To avoid the failing of distributed workflows, the workflow needs to be able to adapt to changes in the environment, like breakdowns of engines. OSPP provides a sophisticated approach for runtime adaptivity, based on so-called directors and coordinators. The coordinator is able to adapt running processes, if necessary. The director decides, whether an adaptation is needed or not. Figure 5.5 depicts the adaptation approach of OSPP. Different, exchangeable implementations of the directors exist. The BDI-director[10] is based on the concept: **b**elieve, **d**esire and **i**ntend, known from artificial intelligence. The decision for or against adaptations is derived using case-based reasoning. The Semantic-BDI director[60] bases on the BDI concept, too, but uses reasoning over ontologies, to derive decisions. Finally, a less powerful statechart-based director[34] exists, which derives decisions using statecharts.

Each workflow engine needs to persist the tasks it is currently executing, to ensure that their data is not lost, due to system breakdowns. As each OSPP engine can be used to design new processes, viz. distributed workflows, at runtime, these process definitions need to be persisted, too. Special for OSPP is that tasks are realized using roles. Tasks play roles of five different contexts, namely from a behavioral, a functional, an informational, an organizational and an operational context. Admittedly, the behavioral context does not comprise roles, but the individuals, which are refined by the roles from the other contexts. The individuals are parts of petri nets, namely transitions, edges, places, tokens and the petri net itself. The informational context contains roles, dedicated to collect or reference data. The organizational context comprises roles to denote the type of organizational unit, like for example `Human` or `Computer`. The functional context provides means to distinguish between functions and conditions. Finally, the operational context focuses on information, required to execute operations. For example a `WebService` role contains information about an invokable webservice.

Integrating DAMPF into OSPP revealed a set of requirements, which are special to systems having a size like OSPP.

First, sublimating classes, when they are loaded, leads to valuable performance penalties in terms of delayed execution. During system startup vast amounts of classes are loaded, whereof most are system classes. Checking each class for the annotation @Entity consumes too much time. A whitelist of package names is used, to overcome this problem. Each time a class is loaded, the only check done is, whether it is in a package denoted in the whitelist. Only if this check evaluates to true, further processing is done.

Second, exposing the state of each object in the runtime fact base swiftly leads to an oversized fact base. A few hundred objects do not pose a problem, but a few thousands do. Especially in regard to instances of classes, which are at the bottom of an inheritance hierarchy. The full state of such instances is derived at runtime, which gets time-consuming in an oversized fact base. DAMPF does not need the state of all currently existing objects in the runtime fact base, but only those, which are currently, actively in use. Due to limited

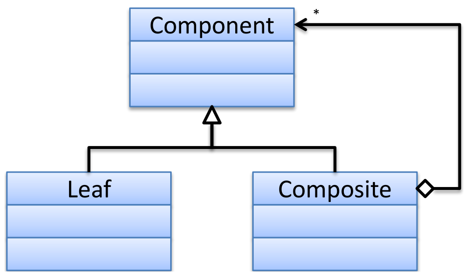


Figure 5.6: Composite Design Pattern, redrawn from [26]

time, this feature has been postponed to future work.

Furthermore the existence of cycles in the schema of applications revealed to be common. Most of them are intra-class-cycles, viz. classes, which have an attribute of their own type. This way object chains are designed. Structural design patterns, like composite or decorator[26], usually contain cycles, too. The composite class in the composite pattern, like depicted in Figure 5.6, points to its superclass, forming an indirect cycle. The same situation shows up in the decorator pattern. In general, designing object chains or trees leads to cycles in the schema. These cycles do not pose a problem on the creation of the relational schema or on the mapping in general, but on the way objects are inserted into the corresponding relations. To properly insert objects, which reference objects of the same type, the order of their insertion is important. Prior to all other objects the top-most object, viz. the root, in the object tree or chain needs to be inserted. In the next step all direct ascendants of the root can be inserted and so on. This ordering can be achieved by sorting on the self-referencing attributes. Additionally bidirectional associations lead to cycles. This is, because the first class references the second, which references the first in turn. In contrast to object chains and trees, such a situation cannot be handled easily. The reason is that no order can be derived. In some cases the first class might be instantiated prior to the second, in some other cases the second might be instantiated prior to the first. To handle such situations, the cyclic attributes need to be marked, so no foreign keys are introduced for them. The realization of this feature has been postponed to future work.

In the current implementation of OSPP another persistence mechanism is used: XStream[5], which provides means to translate between Java objects and XML files. The application of XStream requires using temporary objects, which is a common programming technique. Each time OSPP is started a dummy master process is created, along with a dummy state machine and many more objects. These helper objects have not to be persisted, as they are created at each startup. Using DAMPF as persistency mechanism, these helper objects would be persisted, too, which is not intended. Hence, programmers have to be able to explicitly exclude certain instances from being persisted. This feature has been postponed for future work.

The prototype of DAMPF has been improved considerably, due to its integration into OSPP. Interestingly the performance penalties in terms of delayed execution are low. Admittedly, the first time DAMPF stores the applications'

[5]http://xstream.codehaus.org/

domain objects, a perceptible performance penalty occurs. This is due to the need to create the database schema. In contrast to other approaches, DAMPF creates foreign keys, too. In OSPP almost 70 relations and more than 70 foreign keys need to be created in the simplest case. Once the schema has been created, only small changes to it will follow. Handling small schema changes, for example adding or removing an attribute, does not lead to noticeable performance penalties. Tracing the state of objects as well as extracting and comparing the schema of classes revealed to be faster than perceptible, too. The number of facts in the schema fact base exceeded 480, whereof most (273) are `hasAttribute/4`, leading to a file size of about 58kb. The number of facts in the runtime fact base showed up to be 34 in the smallest example run, viz. only one very simple process was defined, leading to only 4kb. It is important to note, that the runtime fact base swiftly grows to several hundred kilobytes or more, if many complex processes are defined.

Using the monitoring and management utilities of Java almost side-effect free insights into the memory usage of OSPP can be obtained. Running OSPP without DAMPF utilizes in the simplest case approximately 60mb of memory, whereof 15mb are heap-memory and 45mb are non-heap memory. If DAMPF is incorporated, the total memory usage does not grow, but has another distribution. DAMPF grows the heap-memory by 5mb, but shrinks the non-heap memory by 5mb, too. Storing domain objects does not lead to a valuable change to the size of used memory, if XStream is used. Using DAMPF the heap grew by 2mb, whereas non-heap memory grew by 3mb when storing a simple process. Storing multiple more complex processes, the heap grew to 30mb and the non-heap space to 45mb. Thus DAMPF does lead to considerable performance penalties in terms of higher memory usage, as in complex scenarios approximately 25% more memory is utilized. Notably, OSPP with DAMPF exceeds, at least in complex scenarios, 64mb of required memory. This leads to `OutOfMemory` exceptions, if the corresponding virtual machine is started using the default for memory size, which is 64mb. Large systems should be run using `-Xmx256m`, viz. 256mb of runtime memory.

The prototype of DAMPF comprises only about 4.000 lines of Java code and 400 lines of Prolog code, which stem from 108 predicates. The reason for such a small amount of code lies in power of Prolog. If DAMPF did not have used Prolog, but solely Java, several ten thousands lines of code would have been needed. In consequence, DAMPF has a quite small footprint of about 100kb. Admittedly, DAMPF requires a set of third party libraries. First, the Java Prolog API of SWI, which comprises 128kb. Second, Javassist, which roughly requires 600kb. Furthermore a JDBC driver for the corresponding database is needed. The driver for MySQL takes about 700kb. Thus, altogether DAMPF's footprint is approximately 1.5mb. Finally, SWI-Prolog needs to be installed on the system, the application shall be run, which requires further 22mb.

In conclusion, DAMPF has successfully been integrated into OSPP. The performance penalties are, except for the first time a system is persisted, very low. The integration evaluates the approach of DAMPF to be usable in productive environments. To be utilized productively, the current prototype needs further work, as already pointed out.

# Chapter 6

# Conclusion

DAMPF is a novel approach to object-relational mapping, supporting and utilizing the dynamic properties of roles as an extension to the object-oriented paradigm. The main features are support for schema evolution, distribution of domain objects in heterogeneous environments and context-based security.

The core principle of DAMPF can be explained in five steps: *Sublimate, Compare, Adjust, Trace* and *React*.

Original applications are **sublimated** by a bytecode modifying Java agent, which reveals the implicit data flow of the application as an explicit event stream. Sublimated applications fire events, signaling the creation or destruction of objects, roles and contexts, as well as the binding of a role to its player or context. Furthermore every method signals those values of attributes, which have changed. Besides exposing the data flow, further adjustments are made to the application. Every class, context and role gets a new integer attribute, representing an object identifier. In the object-oriented paradigm the identity of objects is an implicit property. To process objects in Prolog this identity needs to be made explicit. Finally code, injecting an instance of the Persistence Manager, which represents the entry point to the runtime utilities, is added.

In the next step the schema is extracted from the application and **compared** with older versions of the schema. Special about this schema extraction is, that the schema is represented as a set of Prolog facts. The existence of a class is denoted by the fact `isClass/1`, taking the name of the class as argument. If a class is marked as a context, an additional `isContext/1` fact is added. Roles are denoted by `isRole/2`, which in the reference implementation takes the player class as an argument, too. This is, because ObjectTeams, which has been chosen as language extension supporting roles, requires a role to have players of only one class. In general a role might change its player freely. ObjectTeams has been chosen, because it is currently the only mature language extension supporting roles. All alternatives are in a stage of early research. The structure of classes, contexts and roles comprises their attributes, too. Those are denoted by `hasAttribute/4`, which takes the name, type, position and the class, the attribute belongs to, as arguments. Although attributes are unordered in the object-oriented paradigm and in the relation schema, the position argument of attributes is needed for internal processing reasons. Static attributes are express in the same way, but using `isStaticAttribute/4`. In plain object-oriented languages, all attributes are pointers to objects. Thus, classes reference each other using attributes. To express such references, the predicate `references/4` is used. It takes the source class and attribute, as well as the target class and attribute as arguments. Finally, inheritance hierarchies are denoted by `subclasses/2` predicates, which relate two classes in the hierarchy. A Prolog rule is used, to retrieve the transitive closure of the inheritance hierarchy.

Besides extracting these facts from the application, they are compared with already existing facts. If a fact already exists, the corresponding structure of the application has not changed. If a fact partially differs, it depends on the arguments, which differ. In general attributes, classes, roles and contexts can be identified by their name. If the developer changes a name, his intent, viz. that he renamed something, cannot be inferred. Either he removed the old entity and added a new one or he just changed the name of the old one. Changes are in general interpreted as combined removals and additions. Chapter 5 explains the identification of changes in detail.

In case the comparison identified changes, the database schema needs to be

**adjusted** at startup time, too. Additional classes, roles, contexts or attributes lead to additional relations or attributes in the database schema. Importantly, the removal of them does not lead to removals in the database schema. This enables developers to reuse data from earlier stages of development. To fixate all changes, and force the removals in the database schema, the developer needs to mark the application as milestone release. Notably, more than changes to the database schema are require by a changed application. When domain objects are stored or restored, these changes need to be taken into consideration, too. This is done in the last step, viz. *React*.

Sublimation, comparison and adjustment happen at startup time. The last two steps happen at runtime. While the application is running, it fires life cycle and value change events, due to its sublimation. The events are **traced** in order to defer further information. Like the applications schema, this runtime information is represented as a set of Prolog facts, too. But, in contrast to the schema facts, not every event leads to a new runtime fact. The main purpose of the runtime fact base is, to defer the current states of all objects. Whenever a create event is fired, a new `instanceof/2` predicate is added. This predicate represents an object and takes the name of the class as first argument and contains the current state as second argument. The current state is a Prolog list. The creation of an object leads to the creation of a new unique object identifier, which is the only value available immediately after object instantiation. Value change events lead to updates of that predicate. Because the events are fired by the object, whose values changed, private attributes do not pose a problem. The destruction of an object leads to the removal of the corresponding fact from the fact base. Inheritance is handled in a special way, which is described in detail in Section 5.2. Special to ObjectTeams is the notion of active and inactive contexts. In the runtime fact base this information is noted by the predicate `contextState/3`, taking the name of the context, its object identifier and state, viz. active or inactive, as arguments.

The last step describes the remaining runtime utilities of DAMPF. These include the features to store, search and restore and to distribute domain objects. Storing domain objects boils down to process the Prolog facts. If the object already exists in the database, DAMPF checks, if an update is required. Else the object is inserted. The transformation of Prolog facts to database data manipulation queries, that is INSERT or UPDATE, is straight forward. To restore objects, a search facility is provided. To search for a domain object, the developer expresses expected attribute-value pairs and connects them conjunctively or disjunctively. Restoring an object boils down to a SELECT query, whose result set is transformed into Prolog facts. The final part of the runtime utilities is a facility to distribute domain objects. Notably, not a new transmission technique is provided, by a new way to distribute domain objects. One of the mature problems in distribution, namely serialization, is a feature inherent to the approach of DAMPF. Each object is represented as a set of Prolog facts. Because these facts are strings and do not require a specific order, they are already in a serialized form. To distribute domain objects their Prolog predicates are packaged. How to transmit the string package can be freely chosen by the developer.

These five steps lead to three main architectural parts: a set of startup utilities, a set of runtime utilities and the Prolog fact bases. Sublimation, schema extraction and comparison, as well as database adjustments belong to the startup utilities. The first two are realized as a bytecode transforming Java agent. Database adjustments are realized as part of the runtime utilities, to support schema changes at runtime, too. Storing, searching, restoring and distributing domain objects, as well as tracing the running application are features of the runtime utilities. All of them need to communicate with the database and the Prolog fact bases. The main entry point to the runtime utilities is the `PersistenceManager`. Communication with the database and Prolog is extracted into separate units: the `DatabaseManager` and the `PrologManager`. Both use a dedicated `Communicator` interface to support multiple database and Prolog implementations. The reference implementation has been developed against MySQL and Microsoft SQL Server as database implementations and SWI Prolog as Prolog implementation. The fact bases are split into a schema and a runtime fact base. The rules to defer further information from these facts are in a third fact base. All these parts are almost invisible to the developer, which is only required to add the Java agent in form of a startup parameter. Startup and runtime utilities, as well as the fact bases, are transparently woven into the application by this agent.

Special about DAMPF is its support for **schema evolution**, which is mainly due to the compare and the adjustment step. The development of applications usually runs through many iterations. In each iteration the schema potentially changes. Current object-relational mappers require to completely recreate the database, whenever such a change happened. DAMPF does not! Changes to the applications schema only lead to a performance penalty in terms of delayed execution at startup time, because the database schema is adjusted accordingly, which includes the migration of data from old to new relations, if necessary. DAMPF even supports to reuse data from former iterations than the last. This is, because a removal in the class schema does not lead to a removal in the database schema by default.

A further unique feature is the novel approach to serialize domain objects in order to **distribute** them. Though a lot of approaches for the distribution of objects exist, none is combined with an object-relational mapper. Serialization in such approaches is dedicated to the purpose of distribution. In DAMPF, all objects are already serialized, as they are represented as Prolog facts. The distribution feature furthermore eases the handling of domain objects, which do not adhere to the current domain model. An application will always only see those parts of the domain object, which adhere to the domain model. Those parts, which cannot be interpreted, are hidden. Notably, hidden does not mean ignored! A system can process a domain object, it does not fully understand, and send it to the next application. The data, which was not understood by that application, remains unchanged and is not lost!

The distribution feature implicitly provides a novel mechanism to context-based security. Security constraints are implicitly expressed by the domain models of applications. If a system shall not understand certain parts of a domain object it processes, the corresponding elements need to be left out in its domain model. Optimally domain objects should be realized by a single class, which does not provide anything else than identity. Everything else should be modeled using roles, played by objects of this class. This way domain objects can be reconstructed easier.

Another contribution of this thesis is the investigation on how to map roles and contexts to relations. Three different approaches have been presented in

Section 4.2.

In summary, three features, not supported by any other object-relational mapper, are realized, based on a shared approach. Developers of applications are allowed to use object-relational mapping during development, which was impossible or at least cumbersome before. Distribution and security, which both were formerly developed independently from persistency, are packaged into an overall solution: DAMPF.

## 6.1 Future Work

The concepts of DAMPF have been realized in a prototype, based on Java and ObjectTeams. The search criteria API has been kept simple and should be improved in the future. Optimizations to the approach, like special caching techniques, are future work, too. Section 5.5 points to some more technical tasks of future work. Furthermore the notion of roles needs to be investigated in more detail. Chapter 2 provided an overview of the current *role* community and our understanding of roles.

Central to most understandings is, that only the players provide identity and roles do not. But how do roles and players depend on each other? In short, players may exist without roles, but roles cannot. The only exception is a role instance, which is *temporarily* unbound.

**Example 15** *John Smith is the executive of a company, which unexpectedly died, due to a car accident. In other words, the person John plays the role* `executive` *in a* `company` *context. When he died, the person does not exist any longer as such, but the role sustains. After a few days a new executive is elected, who carries over the role.*

Example 15 shows, that there are situations, in which an identity is deleted, but the role continues to exist. In contrast, there are situations, in which role instances *existentially* depend on their players. For example a student role instance. If the person, being the student, dies, the student role instance ceases to exist, too. This is because there will never be another person, playing this student role. Thus, whether a role instance existentially depends on its player or not, depends on possible players in the future. Hence, a role-based system requires models, defining **the life-cycle of roles** and thereby predicting the future.

**Modeling Interaction of Roles - Role Life Cycle Charts.** In [57] Dirk Riehle and Thomas Gross define a set of constraints for roles. The key idea is to describe, which roles communicate with which other roles and which roles can or must be played simultaneously and which must not. Figure 6.1 shows an example model utilizing all five role constraints, introduced by Riehle and Gross. An arrow from role $r_1$ to role $r_2$ denotes a direct use-relationship, viz. $r_1$ uses $r_2$ in terms of sending messages to it. A plain line between two roles denotes a bidirectional use-relationship, meaning that both roles send messages to each other. At the ends of these two types of relationships are multiplicities, defining the cardinality of the corresponding role. Figure 6.1 models a professor sending messages to many students and the interplay between students and

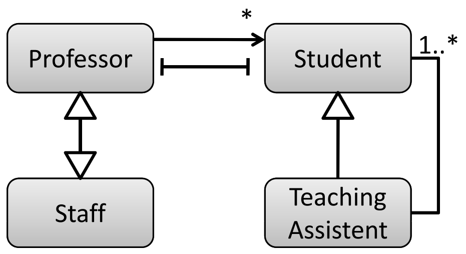


Figure 6.1: Role Life-Cycle Constraints Introduced by Riehle and Gross in [57]

teaching assistants. An arrow with a white arrowhead from role $r_1$ to role $r_2$ denotes that a player, playing $r_1$ has to play $r_2$, too. This relationship is called *role-implication.* An arrow with a white arrowhead at both ends is shorthand for two contrary role-implications and is called *role-equivalence.* Figure 6.1 depicts, that each player, playing the role of a teaching assistant, has to play the role of a student, too. Furthermore all players either are professor and staff at the same time, or they are neither of it. Finally a line with a block at each end between roles $r_1$ and $r_2$ denotes, that if $r_1$ is played, $r_2$ must not be played and vice versa. This last constraint is called *role-prohibition.* The example models the exclusion of being professor and student at the same time.

But these five constraints do not suffice to fully describe a life-cycle. They enable the developer to model, which roles interact with each other and valid states of role-player bindings. The problem is that the dynamic nature of roles is not fully explored. Valid states refer to fixed time. Riehle's constraints just describe, which roles are allowed to be played *at the same time.* This provides a way to model, which roles are allowed to be played in parallel, but not which are allowed or have to be played in sequence.

Currently no approach is able to model valid sequences of roles. At a first glance UML sequence charts look like a suitable base, because they model the interaction of objects. They precisely define, which object sends which message to which other object and the causality of message transfers, viz. the sequence. The problem is that these sequence charts are of exemplary nature. That is, they do not describe all valid sequences, but a single one. To model all valid sequences a vast amount of sequence charts is necessary. Because they do not provide the possibility of modeling all valid sequences with feasible effort they cannot be used as a base for role life cycle charts.

Further possible base diagrams for role life cycle charts are Life Sequence Charts[19], introduced by Damm and Harel in 2001, UML activity charts and BPMN. Which type of chart qualifies best as base and how life cycle charts finally will look like is future work.

Finally, further application areas of DAMPF need to be discovered, examined and evaluated in the future.

# Appendix A

# Prolog Rules

The implementation of DAMPF includes a variety of Prolog rules, used to derive further information from the schema and runtime fact base. This Chapter lists those rules, which are not covered in Chapter 5.

**Merging Instances Connected By Inheritance.** Instances of subclasses are denoted in the runtime fact base by multiple `instanceof` facts, connected by `sameInstance` facts. Example 16 shows the evolution of the runtime fact base for such a scenario step by step.

**Example 16** *Students are, of course, humans. Special to students is only their identifier. Humans have a name and are mammals, which have a birthday, too. This scenario can be modeled using the class* ***Student****, with the attribute* ***studentid****, subclassing class* ***Human****, which has the attribute* ***name****, subclassing class* ***Mammal****, having the attribute* ***birthday****.*

*As explained in Section 5.1, classes of an inheritance hierarchy are loaded bottom up. The schema fact base will emerge in the following order:*

```
%--class Student is loaded
isClass('Student').
subclasses('Student','Human').
hasAttribute('Student','studentid','int',0).
hasAttribute('Student','__DAMPF_oid__','int',1).
%--class Human is loaded
isClass('Human').
subclasses('Human','Mammal').
hasAttribute('Human','name','java.lang.String',0).
hasAttribute('Human','__DAMPF_oid__','int',2).
%--class Mammal is loaded
isClass('Mammal').
hasAttribute('Human','birthday','java.util.Date',0).
hasAttribute('Human','__DAMPF_oid__','int',2).
```

*The instantiation on the other hand runs top down the inheritance hierarchy. Thus, first an instance of* ***Mammal*** *is logged in the runtime fact base, followed by the instances of class* ***Human*** *and* ***Student****. If student John, with id 300, born at the 19th of May 1994, is created, the runtime fact base emerges as follows. After each comment the whole fact base at this point in time is shown.*

```
%--new object event
instanceof('Mammal',[-,0]).
sameInstance('Mammal','Student',0,-1).

%--change value event
instanceof('Mammal',['19.05.1984',0]).
sameInstance('Mammal','Student',0,-1).

%--new object event
instanceof('Mammal',['19.05.1984',0]).
instanceof('Human',[-,1]).
sameInstance('Mammal','Student',0,-1).
sameInstance('Human','Student',1,-1).

%--change value event
```

```
  instanceof('Mammal',['19.05.1984',0]).
  instanceof('Human',['John',1]).
  sameInstance('Mammal','Student',0,-1).
  sameInstance('Human','Student',1,-1).

  %--new object event
  instanceof('Mammal',['19.05.1984',0]).
  instanceof('Human',['John',1]).
  instanceof('Student',[_,2]).
  sameInstance('Mammal','Student',0,-1).
  sameInstance('Human','Student',1,-1).

  %--bottom of inheritance hierarchy reached
  instanceof('Mammal',['19.05.1984',0]).
  instanceof('Human',['John',1]).
  instanceof('Student',[_,2]).
  sameInstance('Mammal','Student',0,2).
  sameInstance('Human','Student',1,2).

  %--final value change event
  instanceof('Mammal',['19.05.1984',0]).
  instanceof('Human',['John',1]).
  instanceof('Student',[300,2]).
  sameInstance('Mammal','Student',0,2).
  sameInstance('Human','Student',1,2).
  sameInstance('Human','Student',2,2).
```

As can be seen in Example 16, the `sameInstance` predicates first store a -1 for the object id referencing the bottom-most object of the inheritance hierarchy. Not before this last object has been created, the actual value can be set. In order to persist an object, whose state is split in the runtime fact base, a Prolog rule is used to merge the states. This predicate is called `fullInstance`:

```
fullInstance(Class,OID,Values) :-
   getAllSuperInstances(Class,OID,X),
   fullInstanceInner(Class,OID,Values,X).

getAllSuperInstances(Class,OID,X) :-
   findall([SID,Super],
     sameInstance(Super,Class,SID,OID),X).

%fetch usual instanceof's
fullInstanceInner(Class,OID,Values,[]) :-
   getInstanceof(Class,OID,Values),
   \+ sameInstance(Class,_,OID,_).

%fetch instanceofs of superclasses and merge
fullInstanceInner(Class,OID,Values,
                  [[SID,SuperClass]|MoreS]) :-
   getInstanceofWithoutID(SuperClass, SID,V),
   fullInstanceInner(Class,OID,MoreV,MoreS),
   append(MoreV,V,Values).

%catch all instanceof/2 and extract their OID
getInstanceof(Class,OID,Values) :-
```

```
   instanceof(Class,Values),
   hasAttribute(Class,'__DAMPF_oid__',_,POS),
   nth0(POS,Values,OID).

%catch all instanceof/2, extract their OID and
%remove it from the values
getInstanceofWithoutID(Class,OID,ValuesWithoutID) :-
   getInstanceof(Class,OID,Values),
   removeValueFromList(Values,POS,ValuesWithoutID).

removeValueFromList([_|B],0,B).
removeValueFromList([A|B],POS,[A|More]) :-
   X is POS - 1, removeValueFromList(B,X,More).
```

The `fullInstance` predicate uses the `getAllSuperInstances` predicate, to the values and object identifiers of all superinstances. It furthermore uses the predicate `fullInstanceInner`, which recursively merges the values. It merges until the only value-set left, does not belong to a superinstance.

The `fullInstance` predicate bases on complete `sameInstance` facts. During object creation, these predicates are temporarily incomplete, in that their pointer to the bottom-most instance id, is -1. Once the bottom-most instance is created, the Prolog rule `mergeTemp` is used to update the `sameInstance` predicates:

```
mergeTemp :- sameInstance(Src,Tgt,OID,-1),
   instanceof(Src,List1),
   latestInstanceof(Tgt,List2),
   hasAttribute(Src,'__DAMPF_oid__',_,POS),
   hasAttribute(Tgt,'__DAMPF_oid__',_,NewPOS),
   nth0(POS,List1,OID),
   nth0(NewPOS,List2,NewOID),
   retract(sameInstance(Src,Tgt,OID,-1)),
   assertz(sameInstance(Src,Tgt,OID,NewOID)),
   dumpschema,
   mergeTemp.

latestInstanceof(Class,Values) :-
   findall([ID,Values],
           getInstanceof(Class,ID,Values),
           X),
   biggest(X,Values).

biggest([[_,V]],V).
biggest([[A,V1],[B,_]|_],V1) :-
   A > B.
biggest([[A,_],[B,V2]|More],Y) :-
   A < B,
   biggest([[B,V2]|More],Y).
```

The rule `dumpschema` is used, to write out the modified version of the schema fact base. A similar rule exists for the runtime fact base and is called `dumpruntime`.

```
dumpruntime :-
    tell('runtime.pl'),
    listing('instanceof'),
    listing('sameInstance'),
    listing('contextState'),
    told.

dumpschema :-
    tell('schema.pl'),
    listing('isClass'),
    listing('subclasses'),
    listing('isRole'),
    listing('isContext'),
    listing('hasAttribute'),
    listing('hasStaticAttribute'),
    listing('references'),
    listing('AddedClass'),
    listing('RemovedClass'),
    listing('AddedRole'),
    listing('RemovedRole'),
    listing('AddedContext'),
    listing('RemovedContext'),
    listing('AttachedSuperclass'),
    listing('DetachedSuperclass'),
    listing('AddedAttribute'),
    listing('RemovedAttribute'),
    listing('ChangedAttribute'),
    listing('ChangedPlayer'),
    told.
```

**Reference Resolution.** In order to store and restore domain objects, the foreign keys between relations need to be resolved. If a domain object is split over many relations, these relations will depend on each other in terms of foreign key constraints. Inserting the domain object into these relations requires the insertion of those parts, which do not depend on other parts first. The predicate `resolveFKs` is used for the required resolution of foreign keys.

```
resolveFKs(Resolved) :- refsAcyclic,
                        resolve(R,[],[]),
                        flatten(R,Resolved).
```

The predicate `refsAcyclic` checks, if there are any cyclic paths created by the foreign keys, except for those looping in the same relation.

```
refsAcyclic :- findall((A->B),cpath(A,B,[]),
                        Cycles),
               length(Cycles,0).
```

To identify cyclic paths a further predicate, `cpath` is used:

```
cpath(Von,[Von|More],Done) :-
   references(Von,_,Nach,_),
   \+ memberchk(Nach,Done),
   \+ roleToContextRef(Von,Nach),
   cpath(Nach,More,[Nach|Done]).

cpath(Von,[Von,Nach],Done) :-
```

```
        references(Von,_,Nach,_),
        memberchk(Nach,Done).

     %check if role belongs to the context
     roleToContextRef(Role,Context) :-
        isContext(Context),
        isRole(Role,Context).
```

This predicate recursively follows the `references` facts, to identify cycles. Role to context references are excluded, because contexts and roles are likely to form cycles, which do not pose a problem for persistency. Such cycles occur, when a context holds an explicit reference to its roles. The cycles arise, because roles are bound to their context and hence reference it.

The predicate `resolve(Ordered, Unresolved, Done)` investigates, which classes are referenced by which other classes. The result is an ordered list of classes, in accordance to references between those classes. It starts by splitting the classes into referenced and non-referenced ones. The first argument will contain the ordered list of classes, the second argument those classes, which still need to be examined. The last argument contains the classes, which have already been investigated. The predicate checks for each unresolved class, if its referencing class has already been investigated and, if so, puts it into the list *Done*, too. If the currently examined class is not yet referenced, it is moved to the end of the *Unresolved* list. Finally, the predicate checks, if really all classes have been processed and cuts further unification for performance reasons, because one solution suffices.

```
resolve([NonRC|More],[],[]) :-
   allNonReferencedClasses(NonRC),
   allReferencedClasses(RC),
   resolve(More,RC,NonRC).

resolve([Cur|More],[Cur|Rest],Done) :-
   referencedBy(Cur,Done),
   resolve(More,Rest,[Cur|Done]).

resolve(More,[Cur|Rest],Done) :-
   \+ referencedBy(Cur,Done),
   addAtEnd(Cur,Rest,Temp),
   resolve(More,Temp,Done).

resolve([],[],Done) :-
   allClasses(All),
   length(All,L),
   length(Done,L), !.

allClasses(Classes) :-
   findall(Class,isClass(Class),Classes).

allReferencedClasses(ReferencedClasses) :-
   findall(Class,(references(X,_,Class,_),
                X \= Class),
          ReferencedClassesList),
   list_to_set(ReferencedClassesList,
             ReferencedClasses).
```

```
allNonReferencedClasses(NonReferencedClasses) :-
   allReferencedClasses(ReferencedClasses),
   allClasses(AllClasses),
   subtract(AllClasses, ReferencedClasses,
            NonReferencedClassesList),
   list_to_set(NonReferencedClassesList,
               NonReferencedClasses).

referencedBy(Class,List) :-
   findall(X,(references(X,_,Class,_),
              X \= Class,
              \+roleToContextRef(X,Class)),
           AllX),
   subtract(AllX,List,IS),
   length(IS,0).

addAtEnd(Elem,List,NewList) :-
   reverse(List,RList),
   reverse([Elem|RList],NewList).
```

Notably the predicate `referencedBy` transitively checks the `references` facts from the schema fact base. The other predicates, `allNonReferenced-Classes`, `allReferencedClasses`, `addAtEnd` and `allClasses` do what their names pretend.

**Normalization.** Automatic normalization of relational schemata is a well discussed research topic. DAMPF includes a Prolog implementation for this purpose, too.

As normalization is a complex task, different subtasks can be identified. First of all functional dependencies need to be identified. Next these dependencies can be used to normalize the according relations.

To infer functional dependencies, the following Prolog rules are used.

```
allFDs(Class,FDs) :-
   findall((A->B),
           checkForFD(Class,A,B),
           FDs).

allNonTrivialFDs(Class,FDs) :-
   findall((A->B),
           checkForNonTrivialFD(Class,A,B),
           FDs).

checkForFD(Class,A,B) :-
   attributeCombinations(Class,Combis),
   member(A,Combis),
   member(B,Combis),
   A \= B,
   A \= [],
   B \= [],
   functionalDependend(Class,A,B).

checkForNonTrivialFD(Class,A,B) :-
```

```
    attributeCombinations(Class,Combis),
    member(A,Combis),
    member(B,Combis),
    A \= B,
    A \= [],
    B \= [],
    intersection(A,B,[]),
    functionalDependend(Class,A,B).

attributeCombinations(Class,SortedPowerSet) :-
    findall(X,hasAttributeAll(Class,_,X),Pos),
    powerset(SortedPowerSet,Pos).

functionalDependend(Class,SrcPos,TgtPos) :-
    valueBagForSrcTgt(Class,SrcPos,TgtPos,Bag),
    allSubSetsAreSame(Bag).

hasAttributeAll(Class,Attribute,Type,Pos) :-
    allAttributesOfClass(Class,All),
    member([_,Attribute,Type,Pos], All).

powerset(S,List) :-
    sortedSubs(X,List),
    list_to_set(X,S).

rsubs([],_).
rsubs([H|T],L) :-
    member(H,L),
    subtract(L,[H],R),
    rsubs(T,R).

sortedSubs(Y,L) :-
    findall(S,(rsubs(X,L), sort(X,S)),Y).

valueBagForSrcTgt(Class,SrcPos,TgtPos,Bag) :-
    setof([V,TgtVals],
          (valuesAtPositions(Class,SrcPos,V),
           valuesForValues(Class,SrcPos,TgtPos,
                           V,TgtVals)),
          Bag).

valuesAtPositions(Class,Positions,AllVals) :-
    instanceof(Class,Values),
    selectFromList(Values,Positions,AllVals).

valuesForValues(Class,SrcPos,TgtPos,
                SrcVals,TgtVals) :-
    findall(TV,
            valuesForSrcTgt(Class,SrcPos,
                            TgtPos,SrcVals,TV),
            TgtVals).

valuesForSrcTgt(Class,SrcPositions,
                TgtPositions,SrcVals,TgtVals) :-
    instanceof(Class,V),
    selectFromList(V,SrcPositions,SrcVals),
    selectFromList(V,TgtPositions,TgtVals).

selectFromList(_,[],[]) :- !.
selectFromList(List,[A|B],[Val|Rest]) :-
    nth0(A,List,Val),
    selectFromList(List,B,Rest).

allSubSetsAreSame([]) :- !.
allSubSetsAreSame([[_,SubSet]|Rest]) :-
    allValuesTheSame(SubSet),
    allSubSetsAreSame(Rest).

allValuesTheSame(Values) :-
    list_to_set(Values,AsSet),
    length(AsSet,1).
```

The first two predicates, allFDs and allNonTrivialFDs are used in the next step, which derives an optimal primary key for the relations. The criterion for optimality is the maximum number of functional dependencies, defining the attribute set. Because normalization bases on the primary key, the resulting key is persisted in the fact base once it has been inferred. Recalculation of primary keys is done using the updatePK predicate.

```
setPK(Class) :- maxCKByFDCount(Class, CK, _),
                assert(isPK(Class, CK)).

updatePK(Class) :- retract(isPK(Class,_)),
                   setPK(Class).

maxCKByFDCount(Class, CK, Count) :-
    allCKsWithFDCount(Class, CKs),
    aggregate(CKs,AggCKs),
    maxElemFromList(AggCKs,CKn,Count),
    flatten(CKn,CK1),
    list_to_set(CK1,CK),
    !.

allCKsWithFDCount(Class, CKs) :-
    findall([C,A],cksWithFDCount(Class,A,C),CKs).

cksWithFDCount(Class, A, Count) :-
    uniqueAttributeSets(Class,A),
    findall(L,checkForNonTrivialFD(Class,A,L),AllL),
    length(AllL,Count).

uniqueAttributeSets(Class, A) :-
    attributeCombinations(Class,Combis),
    member(A,Combis),
    A \= [],
    allValuesAtPositions(Class,A,Vals),
    list_to_set(Vals,Set),
    length(Set,X),
    length(Vals,X).
```

```
allValuesAtPositions(Class, Positions, AllValues) :-
   findall(AllVals,
           valuesAtPositions(Class,Positions,AllVals),
           AllValues).

%e.g. [[0,a],[0,b],[1,c],[1,d]] becomes
%      [ [0,[a,b]] , [1,[c,d]] ]
aggregate([],[]).
aggregate(Input, Agg) :-
   findall([Key,Values],
           (keyset(Input,KeySet),
            member(Key,KeySet),
            findForKey(Input,Key,Values)),
           Agg).

keyset(List,Set) :-
   keylist(List,KeyList),
   list_to_set(KeyList,Set).

keylist([],[]).
keylist([[K,_]|More],[K|Set]) :-
   keylist(More,Set).

findForKey([],_,[]).
findForKey([[K,V]|More],K,[V|MoreV]) :-
   findForKey(More,K,MoreV), !.
findForKey([[_,_]|More],SK,MoreV) :-
   findForKey(More,SK,MoreV).

%for lists of type [[5,x],[2,y],[7,z]|More]
maxElemFromList([[Count,Item]|More],MaxItem,MaxCount) :-
   maxElemFromList(More,Item,Count,MaxItem,MaxCount),
   !.
maxElemFromList([],MaxItem,MaxCount,MaxItem,MaxCount) :-
   !.
maxElemFromList([[Count,Item]|More], _,
                CurMax, MaxItem, MaxCount) :-
   Count > CurMax,
   maxElemFromList(More,Item,Count,MaxItem,MaxCount).
maxElemFromList([[Count,_]|More], CurItem,
                CurMax, MaxItem, MaxCount) :-
   Count =< CurMax,
   maxElemFromList(More,CurItem,CurMax,MaxItem,MaxCount).
```

Once the primary key has been set using `setPK`, a fact `isPK(Class, Key)` will exist. Based on this information it is possible to check, if a relational schema adheres to NF2 or NF3.

```
isNF2(Class) :-
   findall([A,B],allFDsViolatingNF2(Class,A,B),BadFDs),
   length(BadFDs,0).

isNF3(Class) :-
   findall([A,B],allFDsViolatingNF3(Class,A,B),BadFDs),
   length(BadFDs,0).

%NF2: no non-prime attribute provides a fact about
%      only part of the whole key
allFDsViolatingNF2(Class,A,B) :-
   checkForNonTrivialFD(Class,A,B),
   nonPrimeAttribute(Class,A),
   isPK(Class, PK),
   subset(B, PK),
   \+ B = PK.

nonPrimeAttribute(Class,A) :-
   hasAttributeAll(Class,_,A),
   isPK(Class,PK),
   \+ memberchk(A,PK).

%NF3: all non-prime attribute provide a fact about
%      the whole key and nothing but the key
allFDsViolatingNF3(Class,A,B) :-
   checkForNonTrivialFD(Class,[A],B),
   nonPrimeAttribute(Class,A),
   \+ isPK(Class,B),
   assert(fdViolatingNF3(Class,A,B)).
```

To normalize relational schemata to NF3 they need to be split. Please note that the predicate `allFDsViolatingNF3` asserts a new fact, `fdViolatingNF3`, to the fact base, which is for performance reasons, so violating dependencies only need to be inferred once. Identifying the new relations is done using the transitive closure of those functional dependencies, which violate NF3.

```
extractableRelationsForNF3(Class,AllClosures) :-
   findall(SortedClosure,
          (hasAttributeAll(Class,_,Pos),
           findall(ClosureT,
                   reachableByNF3From(Class,
                      Pos,ClosureT,[]),
                   AllCs),
           AllCs \= [],
           flatten(AllCs,Cs),
           list_to_set(Cs,Closure),
           sort(Closure,SortedClosure)),ASC),
           list_to_set(ASC,AllClosures).

reachableByNF3From(_,A,Temp,Temp) :-
   member(A,Temp).
reachableByNF3From(Class,A,[B|More],Temp) :-
   \+member(A,Temp),
   allDirectFD3s(Class,Closure),
   member([A,B],Closure),
   reachableByNF3From(Class,B,More,[A,B|Temp]).

allDirectFD3s(Class,Closure) :-
   findall([A,B],
          (fdViolatingNF3(Class,A,Tgts),
           member(B,Tgts)),
            ClosureList),
    list_to_set(ClosureList,Closure).
```

Finally, the actual split of the classes is implemented using the predicates `insertRef` and `extractColumns`. They need to be triggered by the client!

```
%insert pk, fk and ref to/for relations.
insertRef(Class,NewClass) :-
    nextPos(Class,LastPos),
    concat('refTo',NewClass,Name),
    assert(hasAttribute(Class,Name,LastPos)),
    nextPos(NewClass,LastNPos),
    concat('refFrom',Class,NName),
    assert(hasAttribute(NewClass,NName,LastNPos)),
    assert(references(Class,Name,NewClass,NName)).

%remove the selected attrs from Class and
%add them to NewClass
extractColumns(Class,Cols,NewClass) :-
    member(Pos,Cols),
    retract(hasAttribute(Class,Name,Pos)),
    nextPos(NewClass,NewPos),
    assert(hasAttribute(NewClass,Name,NewPos)),
    assert(attributeOrigin(NewClass,Name,
                           NewPos,Class,Pos)).

nextPos(Class,0) :-
    findall(NPos,
            hasAttribute(Class,_,NPos),AllNPos),
            length(AllNPos,0).
nextPos(Class,NPos) :-
    findall(NPos,
            hasAttribute(Class,_,NPos),AllNPos),
            maximum(AllNPos,_).

maximum([X],X).
maximum([X|Rest],X) :- maximum(Rest, Max), X > Max, !.
maximum([_|Rest],X) :- maximum(Rest, X).
```

# List of Figures

# Bibliography

[1] E. P. Andersen. *Conceptual Modeling of Objects - A Role Modeling Approach.* PhD thesis, University of Oslo, 1997.

[2] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[3] M. Baldoni, G. Boella, and L. van der Torre. Roles as a Coordination Construct: Introducing powerJava. In *Proceedings of the First International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005)*, pages 9–29, 2006.

[4] M. Baldoni, G. Boella, and L. van der Torre. The interplay between relationships, roles and objects. In *Proceedings of the 3rd International Conference on Fundamentals of Software Engineering (FSEN'09)*, 2009.

[5] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming, ECOOP*, pages 323–346, 2007.

[6] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. Role object. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, pages 15–32. Addison-Wesley, 2000.

[7] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems (TODS)*, 1(4):277–298, 1976.

[8] L. Bettini, S. Capecchi, and B. Venneri. Extending java to dynamic object behaviors. In *Proceedings of the Workshop on Object Oriented Developments 2003 (WOOD)*, pages 33–52, 2003.

[9] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP*, pages 262–286, 2005.

[10] W. Bücke. Reflexive workflows. Master's thesis, University of Technology Dresden, 2008.

[11] S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of the ACM*, 29(6):524–544, 1986.

[12] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[13] S. Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[14] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report*, RJ599, 1969.

[15] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[16] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report*, RJ909, 1971.

[17] E. F. Codd. Recent investigations in relational data base systems. *IBM Research Report*, RJ1385, 1974.

[18] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[19] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 2001.

[20] C. J. Date, H. Darwen, and N. A. Lorentzos. *Temporal data and the relational model*. Morgan Kaufmann, 2002.

[21] J. Diederich and J. Milton. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)*, 13(3):339–365, 1988.

[22] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 5th edition*. Addison-Wesley, 2007.

[23] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2(3):262–278, 1977.

[24] R. Fagin. Normal forms and relational database operators. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference On Management Of Data*, pages 153–160, New York, NY, USA, 1979. ACM.

[25] R. Fagin. A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems*, 6(3):387–415, 1981.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[27] G. Gottlob, M. Schrefl, and B. Rck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[28] G. Grahne and K.-J. Räihä. Database decomposition into fourth normal form. In *VLDB '83: Proceedings of the 9th International Conference on Very Large Data Bases*, pages 186–196, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[29] K. B. Graversen. *The nature of roles—A taxonomic analysis of roles as a language construct*. PhD thesis, IT University of Copenhagen, Denmark, 2006.

[30] E. . E. Group. Jsr 220: Enterprise javabeanstm,version 3.0 - java persistence api. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[31] D. Habich, S. Richly, A. Ruempel, W. Buecke, and S. Preissler. Open service process platform 2.0. In *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*, pages 152–159, Washington, DC, USA, 2008. IEEE Computer Society.

[32] T. A. Halpin. *A Logical Analysis of Information Systems: static aspects of the data-oriented perspective*. PhD thesis, University of Queensland, 1989.

[33] W. Harrison and H. 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.

[34] S. Hartung. Ereignisgesteuerte Einschrnkungen in Workflows basierend auf Zustandsdiagrammen. Master's thesis, University of Technology Dresden, 2009.

[35] C. He, Z. Nie, B. Li, L. Cao, and K. He. Rava: Designing a java extension with dynamic object roles. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 453 – 459. IEEE Computer Society, 2006.

[36] S. Herrmann, C. Hundt, and M. Mosconi. Objectteams/java language definition version 1.0. Technical report, Technical University Berlin, 2007.

[37] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2):100–111, 1999.

[38] S. Kelly. What's in a relationship? on distinguishing property holding and object binding. In *Proceedings of the 3rd International Conference on Information Systems, ISCO3: Towards a Consolidation of Views*, pages 144–159, 1995.

[39] W. Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26(2):120–125, 1983.

[40] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[41] R. Kowalski. Predicate logic as programming language. In *Proceedings of IFIP Congress*, pages 569–574, Stockholm, 1974. North Holland Publishing Co.

[42] R. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[43] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition — Revisions to "The class File Format"*. Prentice Hall, 1999.

[44] C. L. Lucchesi and S. L. Osborn. Candidate keys for relations. *Journal on Computer Systems and Science*, 17(2):270–279, 1978.

[45] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1):83–99, 1994.

[46] S. Monpratarnchai and T. Tetsuo. The design and implementation of a role model based language, EpsilonJ. In *Proceedings of the 5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2008)*, pages 37–40, 2008.

[47] S. Nelson, J. Noble, and D. Pearce. Implementing first class relationships in java. In *Proceedings of the First Workshop on Relationships and Associations in Object-Oriented Languages, RAOOL*, 2007.

[48] S. Nelson, J. Noble, and D. J. Pearce. First class relationships for oo languages. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2008.

[49] S. Nelson, D. J. Pearce, and J. Noble. Implementing relationships using affinity. In *Proceedings of the 2nd Workshop on Relationships and Associations in Object-Oriented Languages, RAOOL*, pages 5–8, New York, NY, USA, 2009. ACM.

[50] Object Management Group. The object constraint language specification, version 2.0. http://www.omg.org/spec/OCL/2.0/PDF/, 2006.

[51] Object Management Group. The unified modelling language specification, version 2.2. http://www.omg.org/spec/UML/2.2/, 2009.

[52] J. J. Odell. Power types. *Journal of Object-Oriented Programming*, 7(2):8–12, May 1994.

[53] O. Otto. Bug 275367 - no support for dynamically enhanced entity types. https://bugs.eclipse.org/bugs/show_bug.cgi?id=275367, 2009.

[54] O. Otto. Entwicklung einer Persistenzlösung für Object Teams auf Basis der Java Persistence API. Master's thesis, Technische Universität Berlin, 2009.

[55] M. Pradel and M. Odersky. Scala Roles - A lightweight approach towards reusable collaborations. In *International Conference on Software and Data Technologies (ICSOFT '08)*, 2008.

[56] T. Reenskaug, P. Wold, and O. Lehne. *Working with objects - The OOram Software Engineering Method.* TASKON, 1995.

[57] D. Riehle and T. Gross. Role model based framework design and integration. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 117–133, New York, NY, USA, 1998. ACM.

[58] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 466–481, 1987.

[59] I. Savnik and P. A. Flach. Bottom-up induction of functional dependencies from relations. In *Proceedings of the Knowledge Discovery in Databases Workshop 1993*, pages 174–185, 1993.

[60] S. Schmidt. Kooperative entscheidungsfindung in progressiv reflexiven workflows. Master's thesis, University of Technology Dresden, 2009.

[61] K. Smolander. OPRR: A model for modelling systems development methods. *Next Generation CASE Tools, K. Lyytinen and V.-P. Tahvanainen (Ed.)*, 1991.

[62] K. Smolander. GOPRR: a proposal for a meta level model. Technical report, University of Jyväskylä, 1993.

[63] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *IEEE Transactions on Data and Knowledge Engineering*, 35(1):83–106, 2000.

[64] T. Tamai, N. Ubayashi, and R. Ichiyama. Objects as actors assuming roles in the environment. In *LNCS Software Engineering for Multi-Agent Systems V: Research Issues and Practical Applications*, pages 185–203, 2007.

[65] D.-M. Tsou and P. C. Fischer. Decomposition of a relation scheme into boyce-codd normal form. *ACM SIGACT News*, 14(3):23–29, 1982.

[66] R. K. Wong. Heterogeneous and multifaceted multimedia objects in door/mm: A role-based approach with views. *Journal of Parallel and Distributed Computing*, 56:251–271, 1999.

[67] R. K. Wong, H. L. Chau, and F. H. Lochovsky. Door: A dynamic object-oriented data model with roles. Technical Report 12, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, May 1996.

[68] C. Zaniolo. A new normal form for the design of relational database schemata. *ACM Transactions on Database Systems*, 7:489–499, 1982.