DAO Dispatcher Pattern: A Robust Design of the Data Access Layer

Pavel Micka
Faculty of Electrical Engineering
Czech Technical University in Prague
Technicka 2, Prague, Czech Republic
mickapa1@fel.cvut.cz

Zdenek Kouba
Faculty of Electrical Engineering
Czech Technical University in Prague
Technicka 2, Prague, Czech Republic
kouba@fel.cvut.cz

***Abstract*—Designing modern software has to respect the necessary requirement of easy maintainability of the software in the future. The structure of the developed software must be logical and easy to comprehend. This is why software designers tend to reusing well-established software design patterns. This paper deals with a novel design pattern aimed at accessing data typically stored in a database. Data access is a cornerstone of all modern enterprise computer systems. Hence, it is crucial to design it with many aspects in mind – testability, reusability, replaceability and many others. Not respecting these principles may cause defective architecture of the upper layer of the product, or even make it impossible to deliver the product in time and/or in required quality. This paper compares several widely used data access designs and presents a novel, robust, cheap to adopt and evolutionary approach convenient for strongly typed object oriented programming languages. The proposed approach makes it possible to exchange different data access implementations or enhance the existing ones even in runtime of the program.**

***Keywords*—data-access; software design; pattern; object-oriented; architecture; software evolution**

I. INTRODUCTION

Software design pattern can be understood as a well-established and reusable technique of designing certain software artifacts that are frequently present in various particular forms in a number of software projects. This paper introduces a novel software pattern aimed at accessing database objects. Its basic idea is motivated by the work of other authors that is briefly surveyed in section III.

Modern computer systems have to deal with increasing volume of data. According to the Moore's law [1], the number of transistors in integrated circuits doubles approximately every 18 months and as the computational capacity grows, grows also the volume of data processed. Hence, the systems and their storage engines (databases), became also increasingly complex in past decades.

To deal with the complexity of application (business) logic, object oriented programming was introduced. Nevertheless, the data itself is usually stored in conventional relational databases, which creates impedance mismatch between the data storage and the program itself. Object-relational technologies and frameworks, such as Java persistence API [2], were developed in order to minimize the differences and provide transparent persistence to the programmer.

Such frameworks help to separate the principal concern of business objects behavior (business logic) from the infrastructural concern of how business object's data is retrieved/stored from/to the database and make business objects free from this infrastructural aspect by delegating it to specialized data access objects (DAO). Thus, DAOs intermediate information exchange between business objects and the database. To facilitate the replacement of the particular mapping technology and to encapsulate database queries, data access objects layer pattern was devised. There are many possible implementations that differ mainly in their reusability, testability, architecture/design purity and by the means they provide to support software evolution.

II. BASIC PRINCIPLES

In order to compare various implementations/designs, we use the following criteria, which describe their conformity with the object oriented paradigm and applicability in non-trivial and evolving software systems. Although these principles are well known within the software engineering community, we will describe them in next paragraphs in order to avoid possible misunderstandings stemming from different definitions.

Encapsulation – the data access module should be well encapsulated to hide implementations details (see the Protected Variations GRASP pattern). Minor changes in implementations should never affect interface of DAO module.

Do not repeat yourself (DRY principle) [3] – the code of the module itself as well as code needed for the usage of the module should not be duplicated (or even multiplicated). This constraint reduces the number of scripts needed to test the application and reduces the possibility of regression defects caused by modifying only one of the copies of the respective code.

You aint gonna need it (YAGNI principle) [4] – the user (programmer) should never be forced to create classes or structures, which he doesnt need at the moment. Also the module itself should fit the actual needs of the programmer, not needs of some feature, which may not be implemented yet. The YAGNI principle reduces code bloat and hence saves money, which would be otherwise spent to create, debug and test superficial features.

Single responsibility principle – every class/structure of the program should have only one responsibility. Hence, if

 ISBN: 978-1-61208-276-9

```
@Entity
@NamedQueries(
    {@NamedQuery(
        name =Book.Q_FIND_BY_TITLE,
        query = "SELECT b FROM Book b
            WHERE b.title = :title")})
public class Book {<CODE>}
```

Fig. 1. JPA Named Query code example

properly encapsulated, it can be easily replaced by another implementation. As the code is focused and has only limited set of dependencies, classes respecting this principle are easier to test.

Reusability – the generic DAO functionality should be reusable, project independent and possibly modularized. Reusability reduces costs of the module, because the generic core code is written and tested only once and developers shared by several projects have to be familiar with only one DAO implementation.

Testability – the testability criterion states that the DAO functionality should be controllable by external testing scripts, its behaviour should be observable and the number of scripts needed for its testing should be minimized.

III. Conventional approaches

A. Generated queries (no DAO)

The most straightforward implementation of data access is not to use the data access layer at all and hardcode the functionality into business objects/service layer. As an example may serve *Java Persistence API Named queries*[5].

The named queries are Strings written in JPA query language, which are passed to the framework as class annotations (metadata) as shown in Figure 1. The programmer invokes these queries by their name. The named queries are usually generated by integrated development environment and do not posses any means for structural parameterization (i.e. name of a columns passed as a query parameter).

Thanks to its support by development environments, named queries are convenient for rapid development of a product prototype.

Nevertheless, they are enormously inappropriate for usage in production. The main disadvantage stems from the above-mentioned fact that their structure cannot be parametrized. This means that for every entity and its every property a new named query has to be created, what results in massive code duplication and additional testing expenses. In addition, all queries are bound to entities, so they are not reusable at all in other non-related projects. Such a design violates encapsulation and single responsibility principle, because the data-access technology is invoked directly from business logic. This makes business logic dependent on the data access technology, although it should be technology agnostic, and when the data access implementation is changed, the business logic will have to be reprogrammed and retested as well.

B. Simple data access object

To encapsulate the technology used, data access objects may be introduced. In their simplest form [6][7] there is one DAO for every business object in the domain that provides all the functionality needed. This design can be seen as encapsulation of generated queries.

Although it solves the main architectural drawback of generated queries, there exists one DAO class per each business object class and it causes immense code duplication, which makes the objects hard to test and maintain. This is why this approach is not suitable for practical usage and the scientific community gone on in investigating more sophisticated methods.

C. Generic data access object

The above mentioned code duplication can be resolved using generic data access object (Figure 2) that contains methods common for all entities, such as *findById*, *remove*, *getAll*, in their generic form (i.e. property and names are passed as parameters when necessary).

GenericDAO class is highly cohesive and radically reduces code redundancy and thus improves testability as opposed to the generated queries design. When combined with templating features of the given programming language (e.g. templates, generics), then the class also provides type safe access to the underlying repository.

Generic DAO still possess some design drawbacks. First of all, there is a question, where to place specific DAO functionality. For example, let us have the query looking for all books that are currently in the borrowed state. One option might be to place all specific queries into GenericDAO class, which will result in creating poorly cohesive class with responsibilities over several entities. The second solution might be to create a new specific DAO (e.g. BookDAO) for all persistent business objects, when needed. Although this second option is better, it still does not satisfy another requirement: the data access should support software evolution. Let us suppose that software, which has been developed for a long time, uses GenericDAO in conjunction with specific DAOs. Then a new requirement appears, which implies a specific functionality of getById method for the Book entity. Again there are two options, how to realize the new behaviour. The first one requires sub-classing of the GenericDAO and overriding the getById method so that it behaves differently for Book entity (testing the type of the entity by *instanceof* operator). The

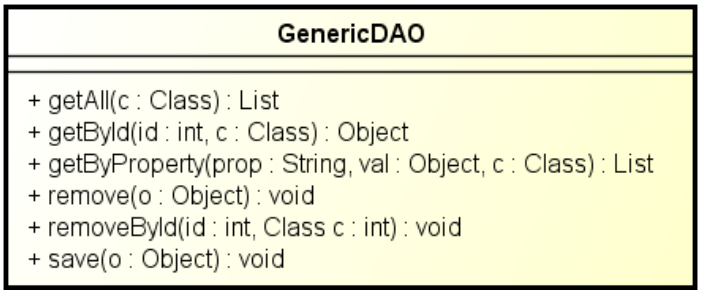

Fig. 2. UML Class diagram of GenericDAO class

 ISBN: 978-1-61208-276-9

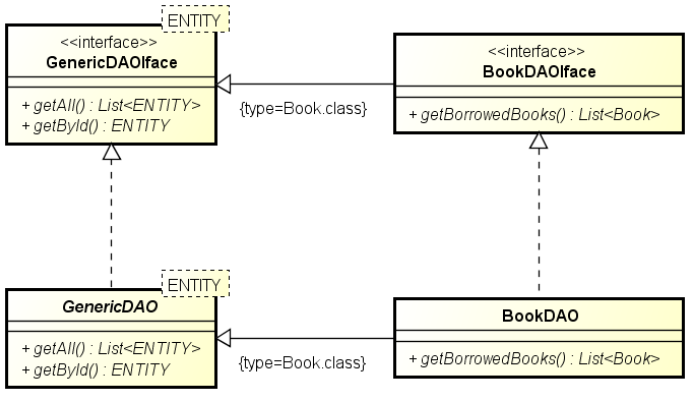

Fig. 3. UML Class diagram of Generic superclass DAO

second option is to put this modified method into specific DAO. While the first approach will later or soon result in spaghetti code — code with enormously tangled structure, usually massively using branching and loop statements — when more modifying functionality will be added, the second approach requires rewriting all calls of the respective method of GenericDAO to specific DAOs one. Thus, none of these two options is satisfactory.

D. Generic superclass

To make possible the code evolution, GenericDAO (see Figure 3) can be modelled as a common (abstract) superclass of all DAO objects. Rosko [8] presents a very similar approach, but he isusing factory to instantiate particular DAOs. Because there will be a mandatory implementation of a specific DAO for every entity, the situation described above will never happen. Software evolution is well supported, because the programmer can consistently override the generic implementation in the respective specific subclass, easily add new specific data access methods and last but not least, the implementation can be easily protected by interfaces and reused in other projects.

Although the generic superclass DAO solves most of the design flaws of the previously discussed implementations, it creates a new one. According to our experience with development of enterprise systems, for the most of entities the generic method implementation is sufficient and also many entities do not require any additional specific methods. And as the specific DAO classes are mandatory, the design results in many classes with empty specification, which is prepared only for possible future changes. This is premature generality, which strongly violates the YAGNI principle.

IV. DAO Dispatcher pattern

To overcome violation of YAGNI, we introduce a new DAO Dispatcher pattern, which combines benefits of both simple Generic DAO and Generic superclass DAO.

A. The overall structure

The pattern (see Figure 5) uses internally Generic DAO class mentioned in the previous chapter to handle all generic requests at one place. If necessary, additional data access methods can be defined in specific DAO classes derived w.r.t. inheritance from the more generic one. If present, the specific data access object mandatorily implements all generic methods, forwarding the call to the respective method of the generic DAO class by default. The signatures of the corresponding methods of specific and generic DAOs are identical except the following point. As the generic DAO class processes data objects of various types, its methods have to have the class parameter that determines the exact type of the processed data. This class parameter is not necessary in case of specific DAO classes. In this case, the type of processed data is implicitly determined by the type of the specific DAO class itself. As a common facade for all generic calls, a new class GenericDAODispatcher was introduced.

B. Registry/DAO Dispatcher class

The registry object is the core of this pattern. It implements the GenericDAO interface and all generic calls should be always made though the registry object. When no specific data access object is registered, it simply delegates the call to the GenericDAO, otherwise the DAO specific to the given class is called.

This mediator makes it possible to introduce the specific functionality without any changes to the code (only the project configuration) just in time, or even to hotswap DAO implementations at runtime.

C. Abstract specific DAO

The abstract specific DAO is a common ancessor of all specific DAO implementations. As it was stated in the previous chapter, usually, the generic functionality is sufficient for most use cases. This is why the default functionality of the specific DAO just routes the query to the generic DAO implementation.

When need for a new DAO functionality occurs, the programmer subclasses the abstract specific DAO and creates only the new method — writes only what he needs. The modification of the generic behaviour is analogous and requires only overriding of the respective method.

D. User interaction

From the user's point of view, there are four major types of interaction with the framework. The interactions are shown as UML transactional diagram in Figure 4.

The first interaction type depicts a call of *getAll* method on a specific DAO type – BookDAO. As it was already stated, the programmer typically does not need to override the existing generic functionality, but wants to extend it. Hence, when the dispatcher is called and the call is delegated to the BookDAO, it only propagates the call further to the generic DAO implementation.

On the contrary, when the getAll functionality is overriden in the specific DAO, than only the delegation from dispatcher is made and the call is executed by the specific method itself (second interaction type).

When the programmer does not specify any specific DAO for the Book entity, than the dispatcher calls directly the GenericDAO in order to provide the default common functionality. This interaction type, third in the image, is predominant for

 ISBN: 978-1-61208-276-9

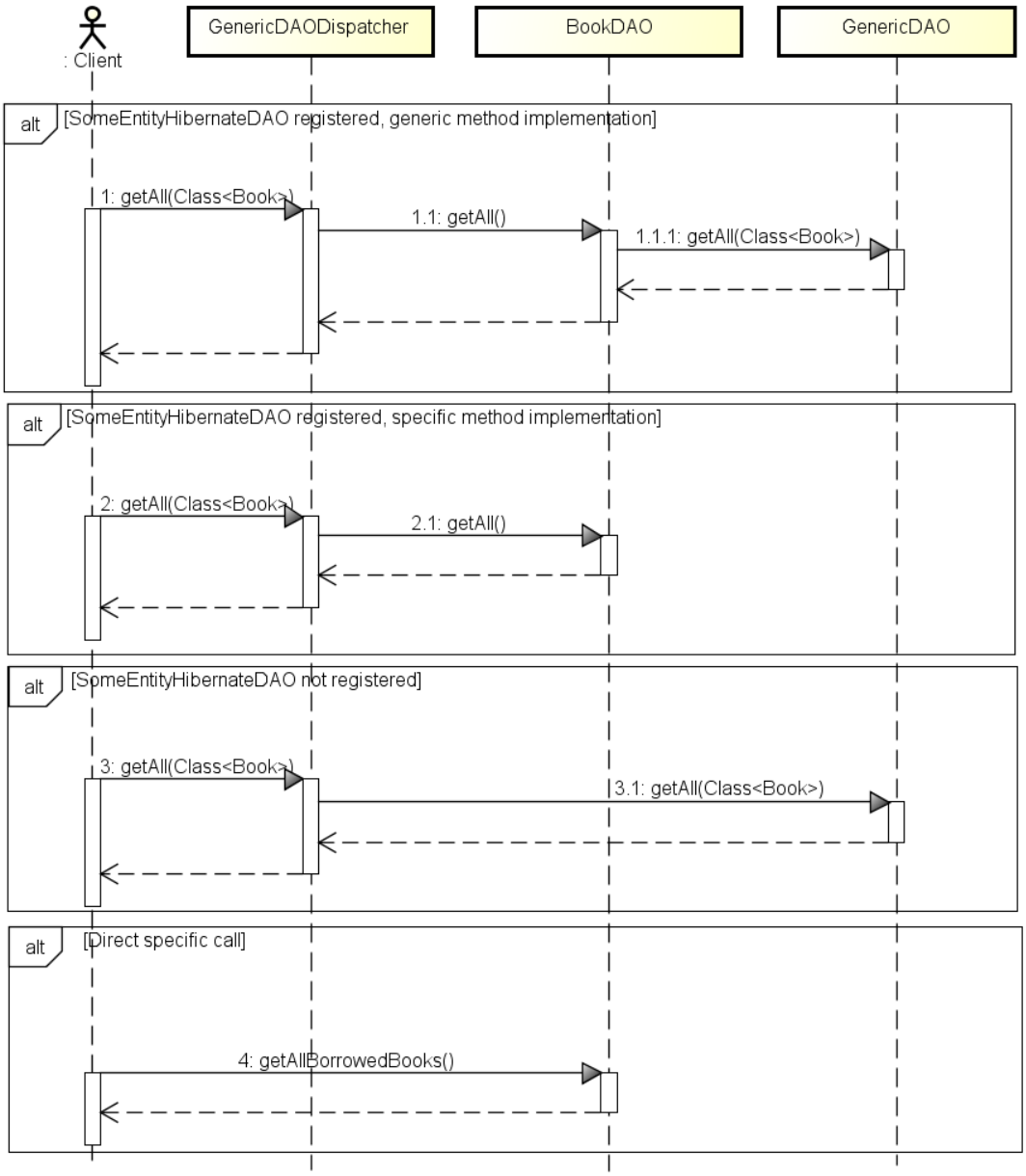

Fig. 4. UML diagram of DAO Dispatcher pattern with Hibernate (JPA) data access implementation

new or not fully fledged applications, where data handling does not have any exceptions from the general flow.

The fourth interaction shows direct invocation of a method specific to the given entity. This method cannot be called through the dispatcher, because the generic interface does not contain its contract and there is naturally no generic implementation in the GenericDAO class. For this reasons the specific functionality calls are always made directly.

E. Advantages of the pattern

The above described structure of the pattern in conjunction with the designed interaction flow provide significant benefits for the end programmer (programmer which creates a system with DAO Dispatcher pattern already implemented as a submodule).

Namely the programmer does not need to write and test the generic DAO functionality, which is already embedded in the submodule.

Also he does not need to prematurely determine, whether the given entity will need any special handling when being stored or retrieved from the database. The framework allows the programmer to make this decision just in time – when it is really needed.

Last but not least, the pattern structure is highly dynamic and flexible. The overriding functionality can be easily plugged-in using configuration of the application, because this change does not require any modifications of the source code

 ISBN: 978-1-61208-276-9

of the project itself. The behaviour of the application can be modified/extended even at runtime.

V. Related patterns

Our novel DAO Dispatcher design pattern uses and extends several commonly known patterns and principles already described by other works. To allow the reader to understand our approach in detail, this section lists these patterns/principals and depicts their usage, similarities and how they relate to DAO Dispatcher classes.

Singleton [9] – all presented classes in the DAO Dispatcher pattern are singletons by their nature. It means that there exists at most one instance of each of them. The reason is that they are either stateless or their state has the application global scope. An example is the Dispatcher class fulfilling the role of a registry in terms of the Registry pattern described bellow.

Ports and adapters (Hexagonal architecture) [10] – In a nutshell, the hexagonal architecture dictates a design of a component in such a way that it communicates with external entities through an API consisting of technology specific ports that are easily adaptable. This makes the core of the component independent on the specific technologies used by the given project and thus the component core is easily portable to other environments.

In particular, DAO Dispatcher pattern as a whole can be described as a single module with clearly defined boundary (interface/ports), which can be accessed through technology specific adapters, when needed. The Dispatcher pattern API also provides means for setting the implementation of the data source (eg. *JPA EntityManagerFactory*), which can be easily exchanged by a mock implementation for testing purposes.

For example: while the core of the module is stable and provides means for direct (binary) calls, in some cases it might be useful to create a serializing adapter, which will transform the input/output objects into JSON, XML or to any other transport format and back. Because the adapting functionality is located externally from the core, it is still possible to test it directly using ordinary unit tests.

Registry – The dispatcher class is an exact realization of the registry pattern as described by Martin Fowler in [11]. The fundamental principle of this pattern is an associative container enabling service providers to register their services in this container using an (typically unique) identifier. Later on, the clients may look up and use the registered services using these identifiers.

Such an architecture is very flexible. From the perspective of the proposed DAO Dispatcher pattern, it is important that the registry allows for on-the-fly inferencing of the appropriate Specific DAO implementation.

For example: if the DAO object for the Novel entity is requested but not available then the more generic Book DAO object shall be used rather than falling back to the purely generic DAO.

Inversion of Control [12] – In conventional programming, the programmer defines the control flow from the beginning to the end himself. However, if he applies a generic framework to the specific problem domain, he usually designs and implements a plugin to that framework. In such a case, he cannot influence the control flow that is determined by the framework itself. Programmer only fills in additional or overriding functionality using pre-prepared join points. In other words, the code of the programmer has the role of a library, while the control flow (in our case of the query evaluation call) is controlled by the framework (DAO Dispatcher pattern).

VI. Application

In typical software systems, the maintenance and enhancement expenses outweigh the costs of development [13], hence it is crucial to use sufficiently robust components during its construction. The pattern is in particular convenient for applications in enterprise systems, which usually evolve continuously and require means for specialization of generic use cases (and respective data access procedures).

Since, as was already described, the DAO layer forms a well encapsulated module, it can be easily interchanged with another implementation. This might be very useful property, when developing a generic system, which will be used by many different customers, each of whom can use completely different data source.

VII. Future work

Although the pattern is intended to be used in strongly typed languages, some dynamic properties might be also employed in future. Mainly, the DAO Dispatcher (registry) class code is in its static form highly duplicate, because each call of the registry only has to delegate the functionality to the appropriate implementation. However, this duplication is well hidden from the user of the module, it would be convenient to use reflection abilities of the host language in order to simplify the registry implementation and reduce the number of lines of code needed to extend the core module functionality.

The extensibility of the core of the module can be also improved by application of the visitor pattern [9]. Each visitor, accepted by the registry, will provide new generic functionality of the core module and, when needed, also overriding functionalities for specific DAO implementations.

VIII. Conclusion

This paper proposed a novel approach to robust data access design, which overcomes imperfections of common implementations. Mainly it is well testable, reusable, honoures the single responsibility and YAGNI principles and last but not least it supports software evolution.

Because the main logic of the module is well hidden behind a facade, the programmer working with it can be familiar only with the general principle, generic DAO interface and the Abstract specific DAO class. This makes the implementation easy to use and cheap to adopt.

However the reference implementation written in Java, using Spring framework [14] for dependency injection, it provides sufficient means for porting the code to other programming languages and clearly proves that the pattern can be easily implemented in strongly typed language, some language specific improvements might be also employed. The reference implementation can be found at https://kbss.felk.cvut.cz/web/portal/dao-dispatcher.

 ISBN: 978-1-61208-276-9

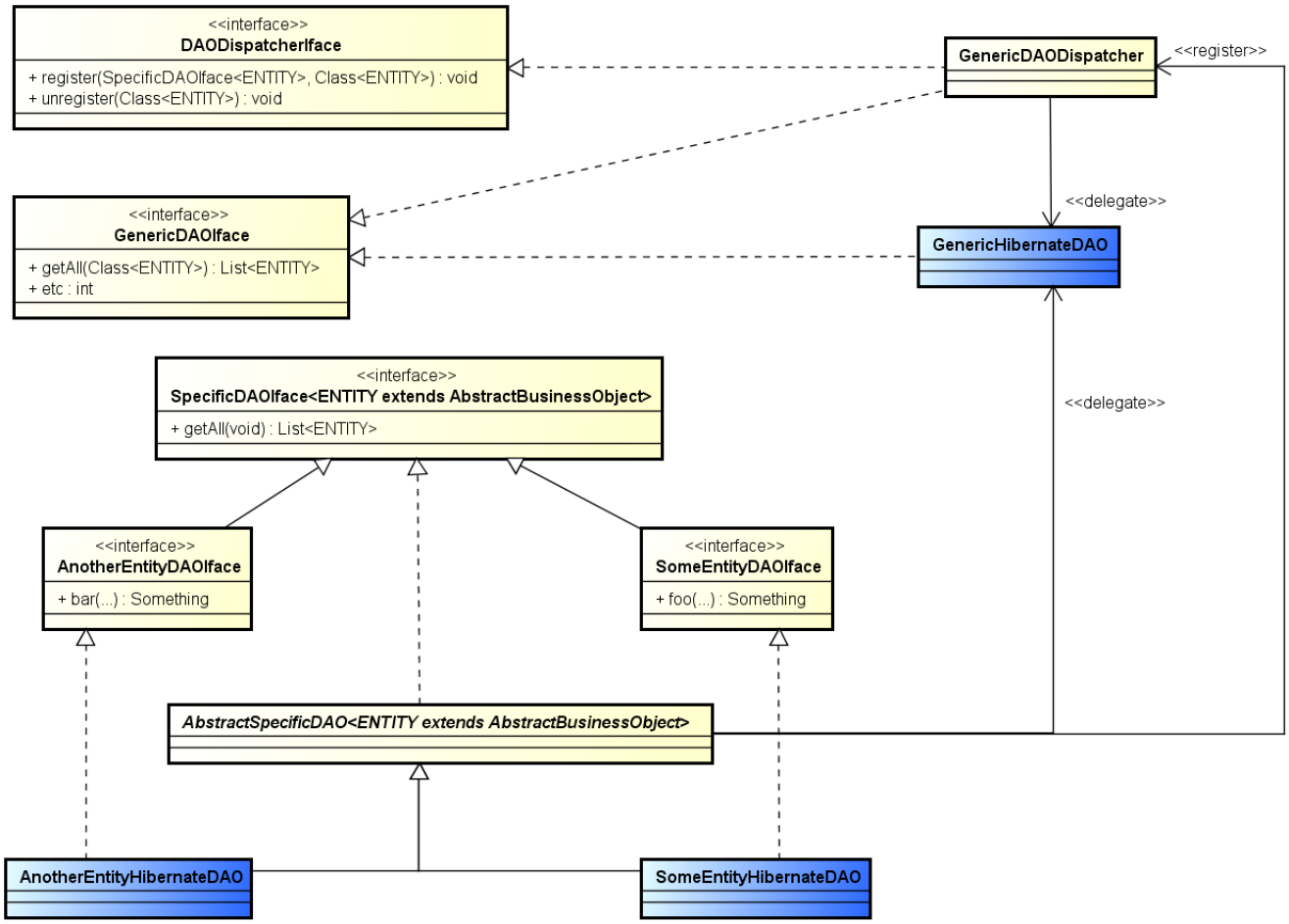

Fig. 5. UML Class diagram of DAO Dispatcher pattern with Hibernate (JPA) data access implementation

ACKNOWLEDGMENT

This work has been supported by the grant by the grant of the Czech Technical University in Prague No. SGS13/204/OHK3/3T/13 — Effective solving of engineering problems using semantic technologies.

REFERENCES

[1] G. Moore, Cramming More Components Onto Integrated Circuits. McGraw-Hill, 1965.

[2] Oracle. (2013) Java persistence api. Retrieved: 12/03/2013. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[3] A. Hunt and D. Thomas, The Pragmatic Programmer: From Journeyman to Master. Pearson Education, 1999.

[4] C. Zannier, H. Erdogmus, and L. Lindstrom, Extreme Programming and Agile Methods - XP/Agile Universe 2004, ser. 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004. Proceedings. Springer, 2004.

[5] Oracle. (2013) Oracle fusion middleware kodo developers guide for jpa/jdo, chapter 10. jpa query. Retrieved: 12/03/2013. [Online]. Available: http://docs.oracle.com/html/E24396_01/ejb3_overview_query.html#ejb3_overview_query_named

[6] M. Berger. (2005) Data access object pattern. Retrieved: 11/03/2013. [Online]. Available: http://max.berger.name/research/silenus/print/dao.pdf

[7] D. Matic, D. Burotac, and H. Kegalj, “Data access architecture in object oriented applications using design patterns,” Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference, 2004. MELECON 2004., vol. 2, pp. 595–598, 2004. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1347000

[8] Z. Rosko and M. Konecki, “Dynamic data access object design pattern,” Information and intelligent systems CECIIS 2008 : 19th International conference, 2008. [Online]. Available: http://www.ceciis.foi.hr/app/index.php/ceciis/2008/paper/view/41

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, ser. Addison-Wesley Professional Computing Series. Pearson Education, 2004.

[10] A. Cockburn. (2005) The pattern: Ports and adapters (”object structural”). Retrieved: 11/03/2013. [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture

[11] M. Fowler, Patterns of Enterprise Application Architecture, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[12] ——. (2005) Inversionofcontrol. Retrieved: 11/03/2013. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html

[13] R. L. Glass, Ed., Frequently Forgotten Fundamental Facts about Software Engineering, ser. IEEE Software, IEEE, May/June 2001.

[14] SpringSource. (2013) Spring framework. Retrieved: 12/03/2013. [Online]. Available: http://www.springsource.org/spring-framework

 ISBN: 978-1-61208-276-9