Issues of Persistence Service Integration in Enterprise Systems

Ádám Zoltán Végh, Vilmos Szűcs, Miklós Kasza, Vilmos Bilicki
Department of Software Engineering
University of Szeged
Szeged, Hungary
{azvegh,vilo,kaszam,bilickiv}@inf.u-szeged.hu

Abstract—**The increasing spread of smart sensors and multi-functional mobile devices extends the problem space of the integration issues appearing in information systems. The method of integrating different data sources (data providers, sensor devices, data hubs, etc.) highly affects system performance but development performance has to be considered, as well. One major topic where automation can help in both areas is persistence. A well-designed persistence layer service can be used by a diverse set of various enterprise applications. The application of the Service Oriented Architecture (SOA) paradigm can help in engineering such systems. Some methods available in a SOA-based system approach the problem at high level. This paper describes a well-maintainable solution at low level, on data model and data access levels. The main challenge this paper addresses is: how to increase the efficiency of integration in Java Enterprise Edition systems in cases when the data model and the interfaces of data access layer change frequently during development and even in maintenance phases. Based on real-life experience gathered during the execution of several telemedicine projects, our paper presents a solution for publishing a data model in the form of transferable objects where the developers do not have to care about the implementation of the assemblers dealing with transferable objects. The benefits and drawbacks have been identified with regards to performance and maintenance costs.**

Keywords-integration; persistence; data access; serialization; maintenance; code generation; Hibernate

I. INTRODUCTION

The heterogeneity of information systems leads to complex integration processes involving different applications and services. A major portion of integration problems cover domain model integration. When the domain entities change frequently, integration costs can increase significantly. This variability is definitely typical in the field of telemedicine, where a diverse set of sensors, data sources and systems have to work together. This increases the need of a cost-effective and sustainable model that supports a wide range of services and devices.

There are several well-known architectural patterns that help with building simple web applications. Specifically, a common solution, the *Data Access Object* (DAO) pattern is used often for decoupling persistence providers from business logic. However, it can be hard to define a really usable, separated persistence service that can be utilized by several other services – especially in the case of complex enterprise systems providing integrated services.

One of the popular enterprise platforms, the Java EE [1][2][3] platform enables the access of databases via Java Persistence API (JPA [4]) implementations. A typical JPA implementation serves as an object-relational persistence mechanism. It also has to utilize proxies that enable the lazy loading of referenced objects and collections (those not loaded along with the entities on the owning side of these relationships). The main problem of this object-relational persistence mechanism is that the objects managed by the JPA are not serializable. Neither are the container proxies (e.g. proxies in Hibernate [5]). Furthermore, these classes could not be published in any other way either. The proxies describe typical associations, aggregations, and inheritance, which is totally useless information after publishing an object, or these relations could not be resolved in a default way. The utilization *Data Transfer Objects* (DTOs [6][7]) can solve these problems.

A major disadvantage of applying a DTO-based approach is that it needs manually implemented transformations between the DTOs and the managed or entity objects. The implementation of such transformations includes a lot of repetitive tasks and thus infers the possibility of human errors and as such, productivity reduction. During the development of a real commercial system the following question emerges: how to solve the DTO-related development problems smarter and faster in a more maintainable way? Hibernate is widely used in persistence layers, but it does not include any acceptable toolkits that support proxy serialization.

An additional issue concerns integration middleware technologies. As almost every single modern enterprise system employs a middleware product at its heart, these technologies also affect the maintainability of an integrated system. Developers usually have to deal with middleware-related communication rules and interfaces regardless of the underlying transport mechanism, should it be an Enterprise Service Bus (ESB) or a web service-based solution (or anything else dealing with distribution issues). For example, in the case of an ESB-based system, each invocation is an assembled ESB message, published through the bus. The integration solution presented in this paper results in a model addressing the aforementioned issues and is based on design patterns. Metrics-based evaluation (described later in the paper) points out the productivity and maintenance-related benefits of the solution.

The paper is organized as follows: Section II gives a detailed overview of the integration issues in the case of a typical persistence layer in an enterprise system. Section III describes the designed model and extensions for an integrated system. In Section IV a short example of data flow and transformations is introduced. This covers the automated persistence layer access. Section V aims at presenting the comparison and evaluation of the presented model and an already existing DTO-based solution. Finally, Section VI contains a summary of the results and a conclusion on the solution presented in this paper.

II. PERSISTENCE LAYER INTEGRATION ISSUES

A major advantage of Hibernate is the availability of a lazy initialization mechanism [8]. Using this type of object-collection handling, developers can save valuable time, typically for run-time. Hibernate proxies allow late binding during data access, i.e., the proxies are not resolved unless the appropriate accessor (getter) methods are called, and thus the time while the query would run is saved. As soon as the content of a managed object’s collection gets needed for some operations, the proxy cares about running the proper query, de-serializes the results and returns them as a managed collection. However, a disadvantage appears when the managed object is serialized, since the proxies cannot be resolved. Traditional Hibernate proxies are not serializable. After serialization and de-serialization of entities, the proxies are trimmed and the lazy associations and aggregations are ignored. Afterwards, when these trimmed entities are returned to the persistence layer, Hibernate detects the ignored associations, and assumes these aggregations as deleted references: if cascading is set, the connected objects are removed and SQL DELETE commands are performed. Figure 1 explains the life cycle of object management. Integration can work only in the *Detached* state.

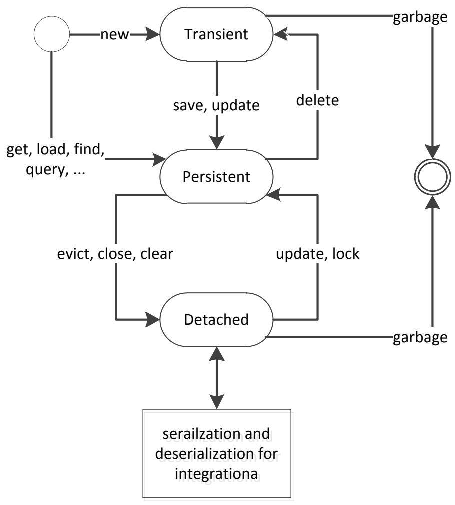

Figure 1. Life cycle of a managed object

In an integrated enterprise system, the DTO design pattern is applied widely in combination with Data Access Objects (DAOs) to overcome the issues regarding proxy serialization. The DTO/DAO model allows the sending and receiving of unmanaged objects and the serialization and de-serialization of them. The DAO-based data access layer often follows the System Façade design pattern. The conversion between the managed objects and the DTOs is provided by DTO assemblers. The DTO assemblers may follow various design patterns. DTO definitions are placed in separate classes. A key problem of the traditional solution is the cost of maintenance. When a DTO structure changes during development or maintenance (e.g. changes caused by third-party developers), maintenance costs increase. A major reason of this increase is that modifications are propagated across code.

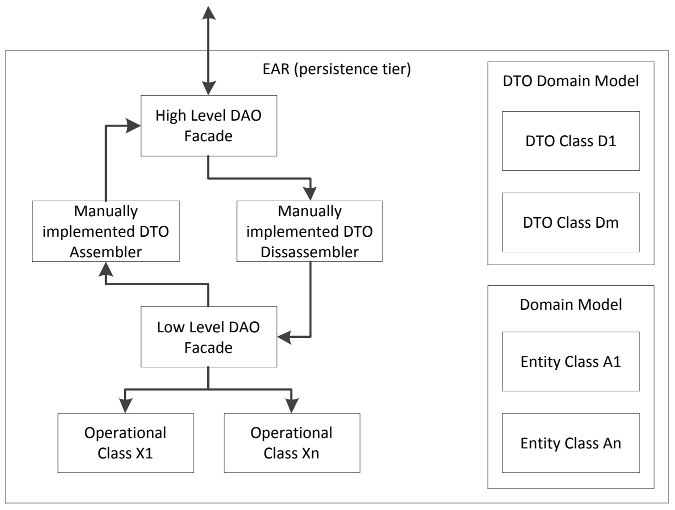

Figure 2. A classical type of DTO model

Data Transfer Objects are accessed by external systems through the DAO layer. Figure 2. explains the DTO transformation and the DAO: the DAO layer is hidden behind a System Façade, but this is not crucial in the model. In enterprise systems, the data provider tier (often the persistence tier) can publish DAOs even through a JNDI directory, or via any other suitable mechanisms. The client or the integration layer is usually responsible for using the DAO, receiving the DTOs, updating and sending them back to the data provider layer. As it was mentioned above, on the consumer side developers also have to deal with the DAO access mechanism at each invocation targeted at the data access layer. This mechanism is based on the integration middleware applied in the system.

III. ARCHITECTURE DESIGN

In this section, our solution is presented that allowed us to publish DTOs to the client side via DAOs in a way where the DAO interface definitions are the same as on the data provider side. Moreover, the DAO implementation automatically creates a huge part of code for the data consumer side (the client side). The presented model hides the different integration layers like ESB, which serialize the objects in some manner. The automatic DTO conversion is not our own development, however the client side DAO generation and the whole system integration presents a new way.

One of the aforementioned problems is the DTO assembling and disassembling. In most cases, the assembler and disassembler codes are written manually and thus the DTO and the entity stored by JPA or Hibernate are nearly the same. Writing the assemblers and disassemblers in such a repetitive way causes code that is hard to maintain. Our methodology and architectural solution uses the Gilead framework to solve this maintenance and reimplementation task. The DTO transformation is executed at the data provider side where the Hibernate proxies are converted to Gilead proxies. This transformation solves the lazy proxy problem because the Gilead proxies are serializable. After the objects are detached, sent and returned, the proxies are restored by Gilead. Finally the returned objects are merged to the managed object instance. Figure 3. represents the high level layout of the Gilead based system.

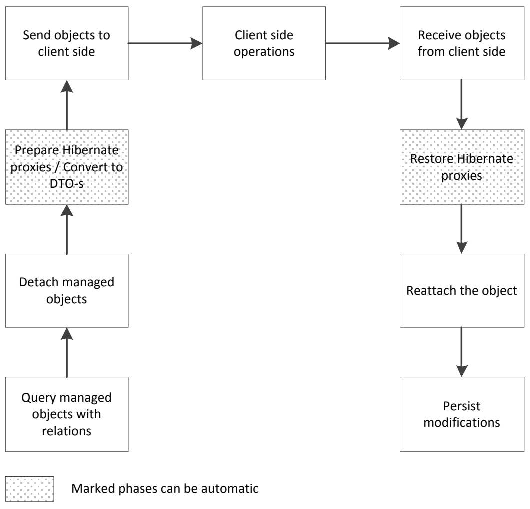
Send objects to client side
Client side operations
Receive objects from client side
Prepare Hibernate proxies / Convert to DTO-s
Restore Hibernate proxies
Detach managed objects
Reattach the object
Query managed objects with relations
Persist modifications
Marked phases can be automatic

Figure 3. Lifecycle of a call

The mechanism of proxy replacement seemed to be simple: the use of the existing Gilead framework provided a solution for the problem. This enabled us to reduce the costs of assemblers and dissemblers written manually by developers where errors occurred very frequently due to careless modifications and redundant work. The other problem, the production of automatized DAOs, seems to be more complex. For example, a data provider layer can publish DAO interfaces through remote procedure calls or web services, but the clients in these cases have to resolve and call the appropriate DAO method manually and they need manual maintenance as well. As the DAO interfaces are available on the data consumer side regardless of the actual type of remote invocation mechanism used, they can be applied to an automatic service invocation delegation approach. We applied the following model for the DAO interface implementation on the client side.

A module based on the Abstract Factory design pattern is responsible for producing DAOs and for caching them. The DAO production is carried out with the help of Javassist [9]. The essence of generation is a method expecting a DAO interface and providing a DAO instance in return. It explores the DAO interface with the help of the Java Reflection API, and then – using the Javassist API and the underlying runtime build technique – it creates the appropriate Java code on the client side. Afterwards, the Java code is compiled and a new instance is returned. The factory is responsible for storing this instance in a cache in order to avoid unnecessary reconversion, since recompilation appears to be a time consuming process. This way, it should be performed only once at the server startup. The developed solution has only been tested with stateless DAOs like most of the solutions used in the integration (see Figure 4.).

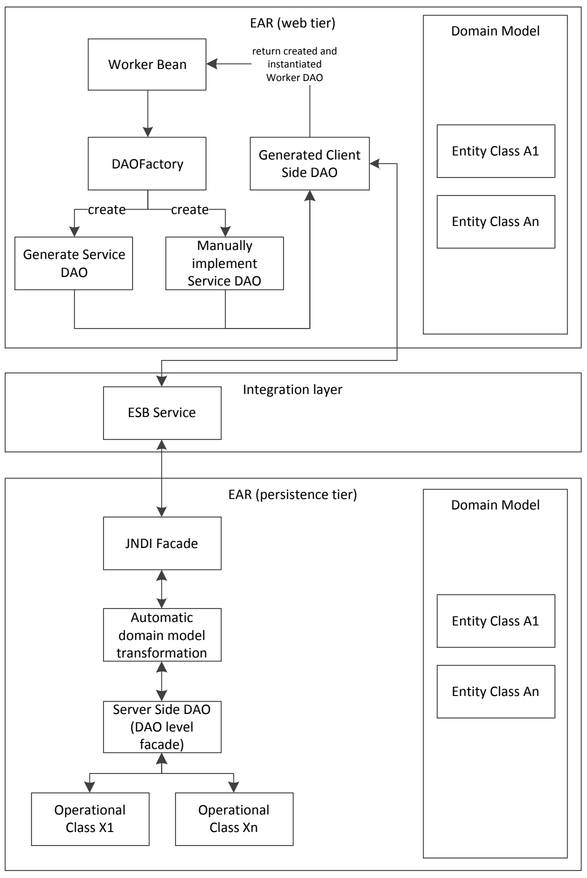

Figure 4. Architecture

By using this approach, we managed to reduce the redundant reimplementation of interfaces on the client side significantly and also to detect human errors occurring during development. The implementation we presented is applied in two real telemedicine projects: the mid-term DEAK-Medistance [10] and the long-term ProSeniis [11] projects. Both projects aim at sensor integration, data collection and storage. The DEAK-Medistance project has approximately 50 user screens, 65k lines of code and 41 entities. In the ProSeniis project there are approximately 150 user screens, 170k lines of code and 155 entities. Basically, the implementation of both projects can be divided into two parts: first, the solution for resolving classical unserializable proxies on the data provider side has been detected; second, we succeeded in the automatization of DAO implementation production on the caller (client) side using DAO interfaces.

IV. Flow of an Invocation

Considering a separated persistence service in an ESB-based, integrated enterprise system, this persistence service provides a data access layer via Enterprise Java Beans-based actual DAO instances registered in a JNDI directory provided by the integration middleware. When a data consumer service wants to interact with the persistence service, it asks the DAO factory for an appropriate DAO instance that implements a specific DAO interface. The factory inspects the methods defined by the interface with the help of the Java Reflection API and afterwards it constructs a Java class by extending the interface and its methods. Each method body assembles an ESB message with three parameters:

- The first parameter defines the *name of the concrete enterprise DAO bean*, which can be automatically prepared from the name of the interface.
- The second parameter defines the *name of the method that will be invoked*.
- Finally, the third parameter holds the *parameters of the invocation*.

As the factory returns the generated DAO instance, the data consumer service can simply invoke any of the methods. The body of the invoked method builds the ESB message and passes it to the bus.

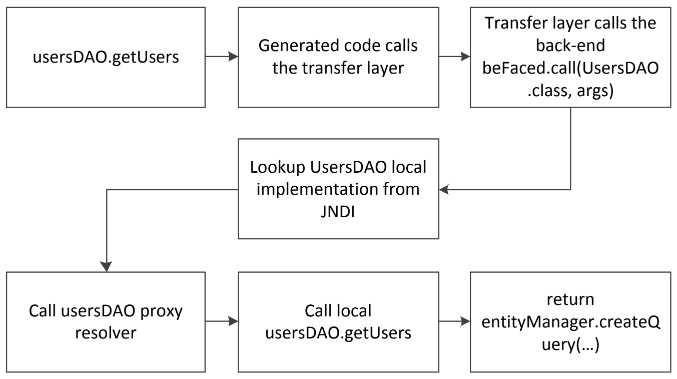
usersDAO.getUsers
Generated code calls the transfer layer
Transfer layer calls the back-end beFaced.call(UsersDAO .class, args)
Lookup UsersDAO local implementation from JNDI
Call usersDAO proxy resolver
Call local usersDAO.getUsers
return entityManager.createQuery(...)

Figure 5. Flow of an invocation

The bus then transmits the ESB message to the persistence service, where a Session Façade interface is responsible for receiving all incoming messages. The receiver looks up the requested DAO (defined by the first parameter) in its JNDI directory. Afterwards, it gets a reference for the concrete DAO instance and then it tries to find the target method based on the second and third parameters of the incoming message using reflection again. If the corresponding method is found, the invocation is delegated to the DAO by passing the parameters defined in the third argument of the ESB message. The DAO uses the Gilead framework to clone and merge proxies. The result is returned in the same way via the bus (Figure 5).

As it can be seen, our approach simplifies the interaction between data consumer and data provider services in the system. Moreover, it hides the integration middleware-specific invocation delegation and thus developers dealing with the client side code can concentrate on the business logic – similarly to the method they use while applying the DAO objects in a simple 3-tier application. This integration model can be adapted for other types of interactions; it is not limited to persistence services.

V. Evaluation of Performance and Maintainability

Performance and maintainability cannot be defined in a general way. This fact is true from another aspect as well, as during system integration a solution that is well maintainable, general and includes a lot of automatization can mean several limitations regarding speed and usability. Furthermore, the model that performed well in theory and in limited-scale experiments was subject to vertical profiling to make the measurement of the suspected performance decrease possible and to be able to compare the performance decrease with the maintenance advantages.

The experiment was performed on a developer workstation (Intel Core I5 CPU, 8GB RAM, Java 1.6.0_18). During the experiment, manual and automatic tests were administered. Logging was carried out with the Test & Performance Tools Platform (TPTP) 4.7 profiler; the platform was a JBoss 4.2.2.GA application server instance. TPTP ran as follows: the TPTP agent ran individually and remotely connected to a starting JBoss. The monitoring results were visualized with the help of Eclipse Helios Performance plugin, which was connected to the separately running TPTP agent.

A. *Comparsion of performance*

During performance measurement metrics were primarily assigned to running time. The consumer-producer call time and the time spent in the generated code were also measured. A real telemedicine application of the classical DAO and the latest re-written version of the application which showed the potentials of novelty presented system were compared.

TABLE I. PERFORMANCE METRICS (SMALLER VALUES ARE BETTER)

Metric	Classic DTO model	Presented model
CGT	0 ms	727 ms
TCMC	51 ms	56 ms
BTMC	3 ms	10 ms
CCS	15	18

The measured values are presented in Table 1. The following metrics were used to measure the absolute performance in a discrete way:

- CGT: *Average Time of Code Generation* per class – the time while the code generation runs. Since the presented model requires the automatic generation of proxy classes, this is an additional one-time cost of the model. This metric is measured in milliseconds.

- TCMC: *Total Cost of Method Call* on client side. Average total time spent with calling methods provided by the service layer and called by the client. The time spent in the service methods is *also* included. This metric is measured in milliseconds.
- BTMC: *Base Time of assembling Method Call*. Average total time spent with calling methods provided by the service layer and called by the client. The time spent in the service methods is *not* included. This metric is measured in milliseconds.
- CCS: *Size of the Call Stack*. Total size of the stack on the client while calling a service method. This metric is measured in stack size.

It can be seen that our solution impacts performance in a negative way. However, this impact is quite low (some milliseconds), considering a simple call.

B. Comparsion of maintainability

The notion of maintainability can be divided into two parts: understandability and modifiability. As a rule of thumb, these characteristics are usually measured by implementing static metrics such as NC (Number of Classes), NA (Number of Attributes), DIT (Depth of Inheritance Tree), etc. Besides these product metrics it is advisable to measure process metrics as well. Process measurement involves simple metrics, such as the number of code lines that should be changed to provide a specific maintenance operation.

The following metrics were used for measuring maintainability:

- ISCL: *Interface Signature Change by Lines of code*. This metric measures the average number of code lines that need to be changed after modifying the signature of an interface method.
- AIL: *Addition of new DAO interface*. This metric measures the number of code lines that need to be changed after the introduction of a new DAO interface.
- DACL: *Change of a domain class attribute set*. This metric measures the average number of code lines that need to be changed after modifying the attribute set of a domain entity.
- DCML: *Modified lines when adding a new class in the domain model*. This metric measures the number of code lines that need to be changed when adding a new class in the domain model (excluding the class definition itself).
- DACC: *Change of the domain class Attribute set*. This metric measures the number of positions in code that need to be changed after modifying the attribute set of a DAO class (including the propagation of the changes).
- DCMC: Count of modifications when adding a new class in the domain model (except the class definition itself). This metric measure the number of positions in code that need to be changed after the introduction of a new container entity.

TABLE II. MAINTENANCE METRICS (SMALLER VALUES ARE BETTER)

Metric	Classic DTO model	Presented model
ISCL	8	5
AIL	8	4
DACL	18	6
DCML	approx. 2x class size	0
DACC	4	1
DCMC	3	0

As it can be seen in Table II, the measurement revealed that since code developers does not have to deal with DTO codebase maintenance, significant time can be saved. This is true only in cases when the DTO matches a managed object definition or its subset.

VI. CONCLUSION

The generation of consumer-side DAO implementation significantly reduces development and maintenance costs. The presented solution enables various types of simplified service invocation delegation in the integration layer. If we want to enable a new integration layer, our only task is to implement the specific generator and the connector at the data producer side. We can also state that automated DTO transformation means some decrease in performance. Tasks done by Gilead proved to be the bottleneck. As a future development, using own assemblers and disassemblers instead of Gilead could be a solution. Regarding maintainability, the code structure and costs of maintenance have improved to a big extent. Less maintenance of code means fewer human errors, which is critical in rapid development cycles. Saving the development costs of the DTO assemblers can mean lower profit. A bigger profit occurs in the reusability of the DTOs on the client side. The generators of the designed DAOs can be reused and extended for any arbitrary serial integrated layer.

Employing DTOs for divided domain model, the managed entities can pose certain limitations. The current solution can be used successfully only in cases where there is no need to employ DTOs with aggregated data.

ACKNOWLEDGMENT

The work presented in this paper was partly funded by the National Innovation Office, Hungary (project No. OM-00191/2008-AALAMSRK ‘ProSeniis’), Telenor 19/55 1I066, Nokia Komarom 19/55 1C117 and GOP-1.1.2-07/1-2008-0007.

REFERENCES

[1] Beth Stearns, Inderjeet Singh, and Mark Johnson, *Designing enterprise applications with the J2EE platform*, 2nd ed.

[2] “Java 2 Platform, Enterprise Edition (J2EE) Overview.” [Online]. Available: http://java.sun.com/j2ee/overview.html. [Accessed: 31-Mar-2011].
[3] Adam Bien, *Real World Java EE Patterns Rethinking Best Practices*. 2009.
[4] Merrick Schincariol, and Michael Keith, *Pro JPA 2: Mastering the Java(TM) Persistence API (Expert’s Voice in Java Technology)*. 2009.
[5] Christian Bauer, and Gavin King, *Hibernate in Action*. Manning Publications Co.
[6] Alexandar Pantaleev, and Atamas Rountev, “Identifying data transfer objects in EJB applications,” in *Proceedings of the 5th International Workshop on Dynamic Analysis*, 2007, p. 5.
[7] Martin Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional.
[8] “Chapter 19. Improving performance.” [Online]. Available: http://docs.jboss.org/hibernate/core/3.3/reference/en/html/performance.html. [Accessed: 31-Mar-2011].
[9] Shigeru Chiba, “Javassist - A Reection-based Programming Wizard for Java,” *Proceedings of OOPSLA’98*, 1998.
[10] “Főoldal | medistance.hu HU.” [Online]. Available: http://medistance.hu/. [Accessed: 31-Mar-2011].
[11] “ProSeniis.” [Online]. Available: http://www.proseniis.hu. [Accessed: 31-Mar-2011].