Evolutionary Database Design: Enhancing Data Abstraction Through Database Modularization to Achieve Graceful Schema Evolution

Gustavo Bartz Guedes[1,2(✉)], Gisele Busichia Baioco[2], and Regina Lúcia de Oliveira Moraes[2]

[1] Federal Institute of São Paulo, Hortolândia, SP 13183-250, Brazil
gubartz@ifsp.edu.br
http://hto.ifsp.edu.br
[2] University of Campinas, Limeira, SP 13484-332, Brazil
{gisele,regina}@ft.unicamp.br
http://www.ft.unicamp.br/

Abstract. Software systems are not immutable through time, especially in modern development methods such as agile ones. Therefore, a software system is constantly evolving. Besides coding, the database schema design also plays a major role. Changes in requirements will probably affect the database schema, which will have to be modified to accommodate them. In a software system, changes to the database schema are costly, due to application's perspective, where data semantics needs to be maintained. This paper presents a process to conduct database schema evolution by extending the database modularization to work in an evolutionary manner. The evolutionary database modularization process is executed during conceptual design, improving the abstraction capacity of generated data schema and results in loosely coupled database elements, organized in database modules. Finally, we present the process execution in an agile project.

Keywords: Evolutionary database design · Schema evolution · Database evolution · Agile methods

1 Introduction

Software systems manipulate data through a database management system, which provides a set of resources that allow the transparent creation, definition and manipulation of databases. An important process in the database design is its definition, made as a schema that describes the metadata about how information is represented and stored.

During an evolution process, new requirements demand the evaluation of the necessary changes and their impact along all levels that compose the software system. Therefore, it is most likely that the database schema also needs to be updated in order to support new functionalities.

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part I, LNCS 9827, pp. 355–369, 2016.
DOI: 10.1007/978-3-319-44403-1_22

Nowadays, the database evolution is mainly performed with the refactoring technique, which consists in small changes to the database schema in order to support new functionalities [1]. However, these changes are costly and may result on inconsistencies and semantic lost, regarding the original database schema. Therefore, it is desirable to produce a high quality data schema at first, rich in abstraction, to facilitate further evolution.

In this way, the choice of the database model has a great impact on schema evolution. Schemas built upon low expressiveness data models make further evolution harder, because they have a poor representation of reality. Therefore, conceptual data models are preferable, since they provide a high-level abstract representation. According to Batini, Ceri and Navathe [2] "the quality of the resulting schemas depends not only on the skill of the database designers, but also on the qualities of the selected data model".

In respect to the software development, Sommerville [3] states that modularity is one of the attributes towards software quality, and if the modularization process is well conducted it results in improved software quality. Thus, modularity assists in software evolution, since each module contains a set of loosely coupled functionalities. In addition, the emergence of the agile methods introduced a new standard in software development, allowing the delivery of independent functional modules in an incremental manner [4–6]. In this way, each increment must be integrated with the already operational software system delivered in previous iterations in order to provide new functionalities. Therefore, modularity can be used at database design process level, in order to produce cohesive and loosely coupled database elements assisting in the schema evolution.

The work of Ferreira and Busichia [7] presents the database modularization process, an extension of the classical database design process. It decomposes a database schema into database modules, according to the application's transactions. In addition, it provides a representation for a database module, which enhances the overall expressiveness of the database design, by representing the conceptual data schema of a module along with its related functionalities.

Database modularization provides a standardized way to conduct the database design process, resulting in loosely coupled and autonomous schemas, providing an evolutionary design approach. It narrows down the scope when assessing the changes and impacts caused by evolution, improving traceability from database and software perspective. A preliminary work towards evolutionary database modularization process was introduced in Guedes, Busichia and Moraes [8].

This paper presents the extension of the database modularization process to support an evolutionary design. The result is a graceful schema evolution amplifying the schema's abstraction to database designers, assisting future changes. Also, we propose a metadata schema to hold the database modules definitions providing the application with a transparent view of the database modules.

The remainder of this paper is organized as follows: related works are presented in Sect. 2; Sect. 3 presents an overview of the database modularization process; the evolutionary database modularization process, including new

phases, is described in Sect. 4; Sect. 5 presents our approach for the evolutionary database modules integration; an application of our approach is presented in Sect. 6; finally Sect. 7 concludes the paper and indicates future works.

2 Related Works

Database schema evolution is not a recent topic and some works date back to the seventies [9,10]. Whereas the software-developing paradigm has changed, from procedural to object oriented, database technologies did not have such a dramatic change and are mostly supported by the relational model [11].

Most works in schema evolution includes database refactoring, co-evolution of the conceptual and physical schemas, impact and prediction analysis of schema changes and schema mapping and versioning. The tight coupling between application and database schema is a common topic of these works.

A database refactoring [1,12] consists in a small change to the schema in order to support new functionalities while it must preserve the behavioral and informational semantics. Thus, after a refactoring, the information, from the application's point of view, must be preserved.

According to Ambler [12], a database refactoring comprises: (i) Changes in the database schema, such as modification in relations, views, stored procedures, triggers; (ii) Data migration to the new schema. This may require data transformation, such as type conversion and update in referential integrity constraints. When multiple applications use the database, a transition period may exist, in which the old and new schemas coexist; (iii) Changes to the application's source code to reflect the schema's modifications.

Cleve et al. [13,14] approach explores the co-evolution of the conceptual and physical layer, by automating generation of the relational schema from its corresponding conceptual schema.

The Hecataeus is a "what-if" analysis tool that simulates the propagation of events caused by schema changes [15]. Hence, focused on the coupling between database elements. Hecataeus maps a relational database schema as well as queries and views into a directed graph, where each node represents a dependency. A set of metrics is provided to evaluate coupling.

The work of Curino et al. [16] presents a set of operators, called Schema Modification Operators (SMO) that modify the physical database schema while preserving previous versions and establishing a mapping among them. This allows different applications to query multiple versions of the database. In that work a prototype tool, the PRISM workbench, is presented in order to support the SMOs.

This paper presents a standardized method to design the database schema in evolutionary and modular way, resulting in a set of cohesive and independent database modules, improving the schema abstraction and reducing the need for database changes. In addition, it provides interoperability between the software and the database, through a catalog that stores the database modules metadata definition.

3 Background

This work is based on the extension of the database modularization design process proposed in Ferreira and Busichia [7]. Therefore, in this section we present the background regarding the phases involved in this process.

3.1 Overview of Database Modularization Design

In this section, we describe the four stages of the "Modularization Design" phase as it is in Ferreira and Busichia [7].

Stage 1 - Partitioning of the conceptual global data schema. The input is the global application schema, which is partitioned in one subschema per subsystem. A subsystem is defined as a group of functionalities.

Stage 2 - Treatment of information sharing. Previous stage may result in overlaps, when a schema element belongs to more than one subschema. In these situations each element is classified according to read or write operations executed by each subsystem. Three classifications are defined as follows:

1. Non-shared: the element belongs to a single subschema and a single subsystem carries out both reading and writing operations on the element.
2. Unidirectional sharing: the element belongs to two or more subschemas and a single subsystem performs the writing operations on the element, while the others carry out only reading operations.
3. Multidirectional sharing: the element belongs to two or more subschemas and two or more subsystems perform writing operations on the element.

Stage 3 - Definition of the database modules. Here a group G_{Si} is created for each subsystem Si containing the elements that it maintains, which are those that perform writing operations. An intersection with all groups is made, in order to handle multidirectional sharing and to define the database modules.

Groups, which intersection with all others is empty, originates a database module per group.

$G_{S1} \cap G_{S2} \cap ... G_{Sn} = \emptyset$, a Module M_n is created for each G_{Sn}.

Groups, which intersection with all others is non-empty, can originate a database module with all elements of the groups.

$G_{S1} \cap G_{S2} \cap ... G_{Sn} \neq \emptyset$, a Module M_n is created with $G_{S1} \cup G_{S2} \cup ... G_{Sn}$.

Another possibility is to create a database module based on the difference between the elements of the intersection result and each subsystem elements group.

$G_{S1} \cap G_{S2} = G_x$, generates module M_x.

G_{S1}-G_x, generates Module M_n.

G_{S2}-G_x, generates Module M_{n+1}.

Stage 4 - Definition of the interface of the database modules. This stage encapsulates subschemas in database modules with public and private procedures, read access and write access respectively. Figure 1 shows a graphical representation of a database module, enhancing abstraction by including both the conceptual schema of the database module as well as the associated procedures.

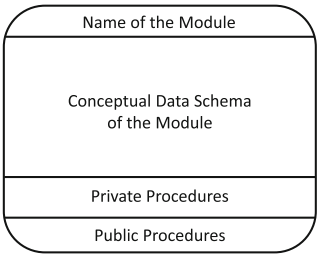

Fig. 1. A module representation [7].

4 Evolutionary Database Modularization Design Process

Our approach towards database evolution is an extension of the database modularization design process along with new data modeling concepts.

The original database modularization process was designed as a top-down database project, which means that a global conceptual schema was upfront designed and partitioned (modularized). On the other hand, our scenario is an incremental database schema design, where subsequent development cycles start from an already implemented schema. For example, this is the case on agile projects.

The evolutionary database modularization design process is conducted with both top-down and bottom-up design strategies. The top-down is applied at each cycle to define new subsystems or associate new requirements and functionalities to the already existing ones. Concurrently, it is necessary to consider the existence of database modules created in previous cycles that can contemplate fully or partially new functionalities, thus characterizing a bottom-up design.

Six phases comprises the evolutionary database modularization design process as showed in Fig. 2. Gray rectangles represent our extension of the original design process.

1. Requirements Collection and Analysis: it consists in identifying and documenting both data and functional requirements.
2. Evolutionary Analysis of Modularization Requirements: in an evolutionary approach, it becomes an iterative process, where all phases are executed at each cycle, resulting in a new increment to the database. Therefore, this phase aims to analyze new requirements taking into account the already implemented databases modules from previous iterations. The output is the Modularization Data Requirements, which indicates how the iteration will affect the subsystems definitions, according to the functional requirements of the previous stage, and the impact on the database modules, resulting in either extending or creating new modules.
3. Iteration's Conceptual Design: the requirements are analyzed in order to produce the iteration's conceptual schema through the use of the entity-relationship model.

4. Evolutionary Database Modularization Design: this phase was extended to perform the database modularization design in an evolutionary manner, considering the stages presented in Sect. 3.1. Thus, we added the "Evolutionary definition of the database modules" in respect to the original modularization process. In addition, the conceptual schema of each database module is adapted to support future evolutions.
5. Logical Design to each Module: in this phase, the conceptual schema of each database module is transformed into a logical schema.
6. Physical Design to each Module: lastly the logical schema of each module is deployed physically into the database, using a specific database management system.

From the second iteration on, we consider the deployed database modules, represented by the dotted arrow in Fig. 2. The next subsections describe, in detail, the new phase called "Evolutionary Analysis of Modularization Requirements" and the extended one called "Evolutionary Database Modularization Design".

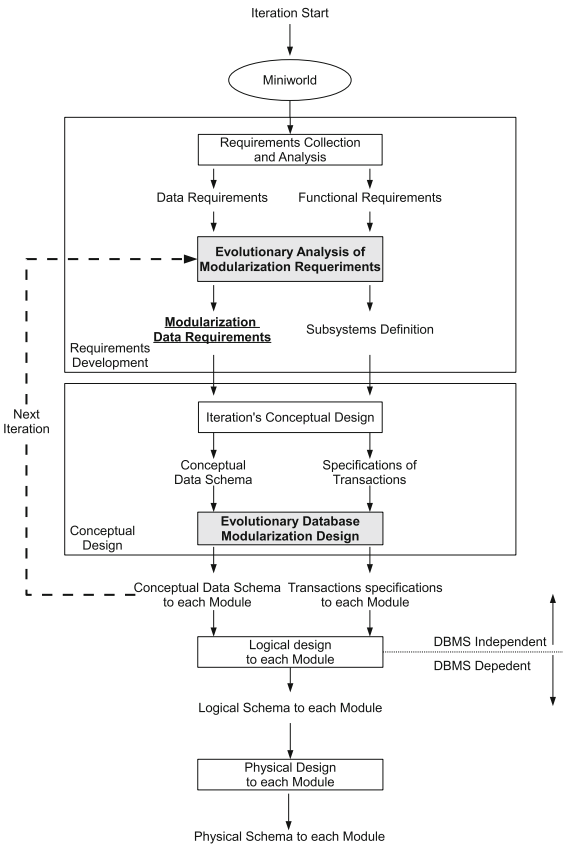

Fig. 2. Evolutionary database modularization design process.

4.1 Evolutionary Analysis of Modularization Requirements

This phase assess the completeness of the database modules in respect to an iteration's functionalities. The activity diagram of Fig. 3 shows the requirements analysis according to the "Evolutionary Analysis of Modularization Requirements" phase as well as the possible outputs.

1. The already existing database modules support the new requirements; therefore, no change is required and the functionalities are implemented into the correspondent database modules.
2. There is no support, from the existing database modules, to the new requirements; thus, new database modules will be created with the correspondent elements and functionalities.
3. If the existing database modules contemplate partially the new requirements, new database modules can be created as well as the extension of the existing ones, in order to support the new requirements.

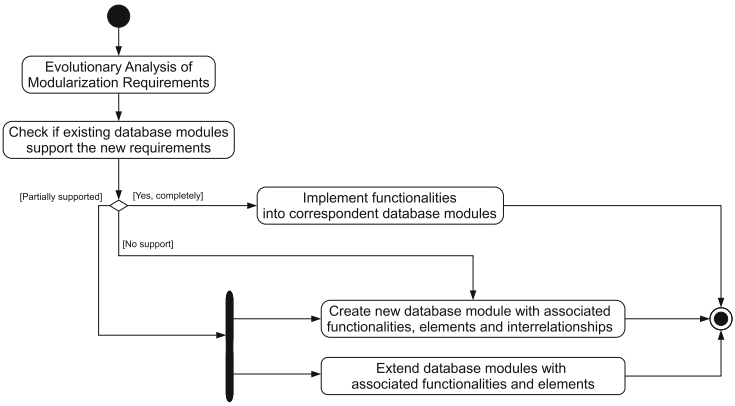

Fig. 3. Activity diagram: evolutionary approach to modularization design.

It is important to note that there are no database modules to assess at the first iteration. Therefore, it results in the creation of the early database modules, which means that the output is the "Create new database module with associated functionalities, elements and interrelationships" of Fig. 3. From the second iteration, this phase receives also the conceptual data schema of database modules generated in previous iterations, represented by the dotted arrow of Fig. 2.

4.2 Evolutionary Database Modularization Design

Our work presents interrelationship and intergeneralization as new semantic data modeling concepts that arise from modules definitions. In order to produce cohesive database modules to support further evolution, we need to consider coupling

that in modularization occurs in interrelationships and generalization hierarchies among modules, a semantic data modeling concept that we named intergeneralization.

An intergeneralization occurs when a future requirement needs to specialize an entity type that belongs to an existing database module. In this scenario, the specific entity type is created in a new module. This avoid unnecessary coupling between the specific and generic entity types and it conforms to the database modularization definition where "a highly detailed degree of data abstraction is required to make a partition of the conceptual data schema" [7]. The dotted generalization hierarchy of Fig. 4, represents an intergeneralization, where the generic entity type E1 and the specific E1' belong to two distinct database modules. A specific entity type (EX) is added to Module X and shares the $A_{E1'}$ attributes domain. Therefore, intergeneralization provides independent evolution of each entity type reducing coupling among the elements. As of interrelationships, they represent the point where two database modules share data. In an evolutionary design there is a high probability of multiplicity change in a future iteration, which would require refactoring. Therefore, interrelationships are multiplicity change safe, since they are mapped as many-to-many relationships, as it is the case of R relationship type in Fig. 4.

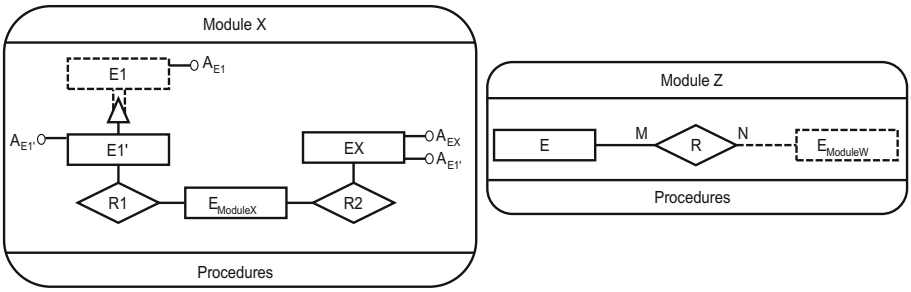

Fig. 4. Intergeneralization representation (left) and interrelationship mapped as many-to-many (right).

With the concepts of interrelationship and intergeneralization presented previously, now we discuss the "Evolutionary Database Modularization Design" stages, that include adaptations and extensions of the original "Modularization Design" stages presented in Sect. 3.1.

Stage 1 - Partitioning of the conceptual database schema. The conceptual schema is still being partitioned in one subschema per subsystem. However, new subsystems and modifications on the already existing ones can occur on future iterations, which can result in a different set of subschemas in respect to the previous iterations.

Stage 2 - Treatment of information sharing. The overlaps of the elements between the subschemas of the stage 1 can change from one iteration to the other.

The introduction of new subsystems and the modification of the already existing ones can change the classification of an element:

1. From Non-shared to Unidirectional Sharing: if a subsystem starts to read data from an element maintained by another subsystem;
2. From Unidirectional to Multidirectional Sharing: if a subsystem starts to write data on an element maintained by another subsystem.

Stage 3 - Evolutionary definition of the database modules. In this stage, the elements may have had their classification changed according to the stage 2, which means intersection operations can generate new interrelationships. Hence, this stage's role was extended and we have added the "Evolutionary" to the original's stage name "Definition of the database modules" (see Sect. 3.1).

According to the database modularization process, there are two options to handle multidirectional sharing. One is to create a database module with the intersected elements; the other is to perform a union among all these elements. Our approach considers only the first, since it will result in loosely coupled modules. First, from one iteration to another the multiplicity of the relationship types can change, however when this occurs in an interrelationship there is no need to refactoring because it is already a many-to-many relationship. The same happens with newly introduced generalization among modules, because they will be mapped as intergeneralizations, hence the specialized entity type is created in the correspondent database module.

This stage defines the database modules following the database modularization requirements, which is the output of the "Evolutionary Analysis of Modularization Requirements" phase, resulting in:

1. Creation of new database modules when:
   (a) a set of elements that were non-shared or unidirectional shared become multidirectional shared;
   (b) new elements are not associated to any previously existing database module.
2. A database module is extended when the newly introduced set of elements were associated to a previously existing database module and have only non-shared or unidirectional sharing.

Stage 4 - Definition of the interface of the database modules. The public and private procedures are associated with each defined database module according to stage 3.

5 Evolutionary Database Modules Integration

In order to support the evolutionary design, we propose an Integration Object Catalog that holds the modularization's process metadata at each iteration, such as subsystems definitions, entity types, relationships types, attributes and the database modules definitions. Figure 5 shows the conceptual schema of catalog,

which is updated to reflect database modules changes. The gray highlighted subschema represents the extension proposed by this work, while the remaining subschema was introduced in Busichia and Ferreira [17,18].

The conceptual schema holds interrelationships metadata in the "Belongs" and "Participates" relationship types. The first relates each database module's elements to its module, while the second holds the relationship types of the application's conceptual schema. Hence, a relationship between two entity types of two distinct database modules is a interrelationship. The former multiplicity of an interrelationship is store to enforce integrity rules, since physically they are many-to-many. Similarly, "Belongs" and "Generalizes" hold intergeneralizations, where the latest indicates if an entity type participates in a generalization hierarchy. If the generic and specific entity types belong to two distinct database modules, then it is characterized as an intergeneralization.

The catalog has a major role in the process and it is used as the input of the "Evolutionary Analysis of Modularization Requirements" phase (see Fig. 2), since it maintains the metadata of the database modularization process. Moreover, it can be used to automate the generation of database modules at each iteration, by verifying the changes in the metadata, such as the conceptual schema, subsystems and the access type to each element. In this way, it is necessary to keep at least the previous instance of catalog data, in order to compare changes and apply them to the database schema, according to the definitions of the evolutionary database modularization design process.

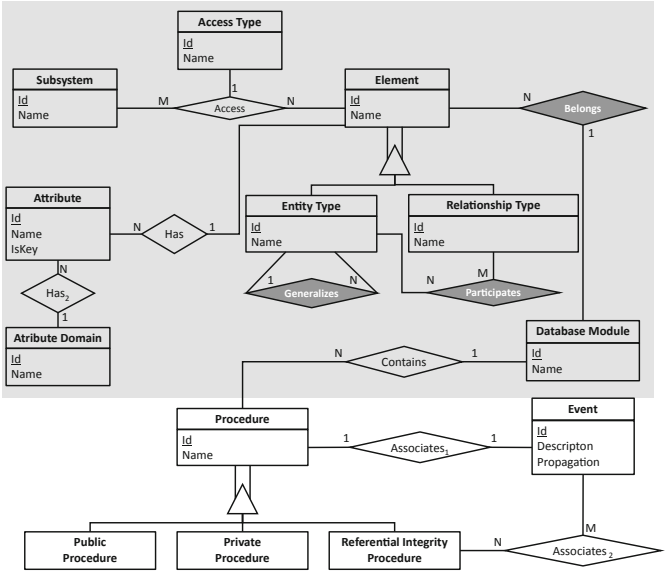

Fig. 5. Conceptual schema of the Integration Object Catalog.

6 Case Study: Applying the Evolutionary Database Modularization in an Agile Project

In this section we use the software specification contained in Ambler [19], a simple karate school system at first, to demonstrate the Evolutionary Database Modularization Design. Table 1 presents a brief description of the requirements.

Table 1. Brief description of the requirements.

Iteration	Description
1	Maintain student contact information
	Enroll student
	Drop student
	Record payment
2	Promote student to higher belt
	Invite student to grading
	Email membership to student
	Print membership for student
3	Schedule grading
	Print certificate
	Put membership on hold
4	Enroll child student
	Offer family membership plan
	Support child belt system
5	Enroll student in Tai Chi
	Support Tai Chi belt system
	Enroll student in cardio kick boxing
	Support the belt order for each style
6	Maintain product information
	Sell product

The first iteration results in two subsystems, one to handle the student functionalities (S1) and one to control payment (S2). These subsystems originate database modules M1 and M2 respectively. Also, we create the Student entity type, instead of creating a generic Person entity type upfront.

Second iteration introduces the belt control system, whose functionalities are grouped in S3 subsystem. When applying the database modularization process it indicates that the existing database modules attend the data requirements partially, since module M1 maintain student data. Next, "treatment of information sharing" indicates that there is only unidirectional sharing between new elements; therefore, new M3 module is created to support the belt system. Iterations 3 and 4 also have similar situations where either a database module is created or an existing one is extended. Figure 6 shows the generated database modules until the fourth iteration.

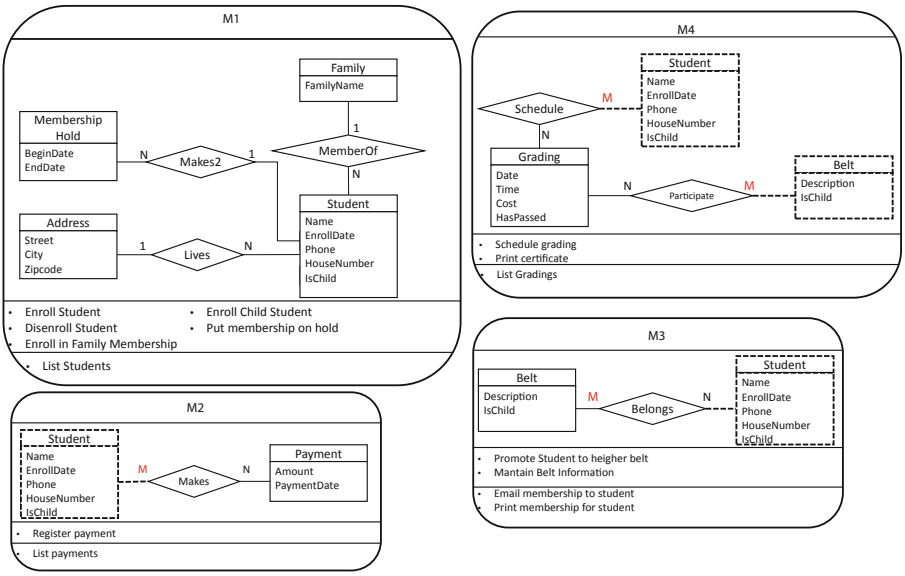

Fig. 6. Generated modules until fourth iteration.

6.1 Fifth Iteration

Initially, the system was design to support only one type of training (karate). However, fifth iteration requirements introduce support for other types of coaching. First, training management functionalities are grouped in a new subsystem, S5. The "Evolutionary Analysis of Modularization Requirements" phase indicates that M1 and M3 modules partially support multi-training functionality. The result of "Evolutionary Database Modularization Design" phase is summarized on Fig. 7.

6.2 Sixth Iteration

The sixth iteration requirements introduce a store functionalities that are grouped into a new subsystem (S6). An intergeneralization is introduced to support both student and general customers buys (Fig. 8). Note that CreditCard attribute are added to both "StudentCustomer" and "Customer" entity types. On the other hand, DiscountIndex attribute is only present in "StudentCustomer" entity type, since it only makes sense to students byers. In this way, "StudentCustomer" and "Customer" entity types evolve separately in further iterations.

Module	Iteration	Element	Element Type	Type of Operation (R- Read / W - Write)						Type of Sharing	Maintained by					
				S1	S2	S3	S4	S5	S6		S1	S2	S3	S4	S5	S6
M1	1	Student	Entity Type	R/W	R	R	R	R	R	Unidirectional						
	6	Lives	**Interrelationship**	R/W						Non-Shared						
	3	MembershiHold	Entity Type	R/W						Non-Shared						
		Makes2	Relationship Type	R/W						Non-Shared						
	4	Family	Entity Type	R/W						Non-Shared						
		MemberOf	Relationship Type	R/W						Non-Shared						
	5	Enroll	**Interrelationship**	R/W				R		Unidirectional						
M2	1	Payment	Entity Type		R/W					Non-Shared						
		Makes2	**Interrelationship**		R/W					Non-Shared						
M3	2	Belt	Entity Type			R/W	R	R		Unidirectional						
		Belongs	**Interrelationship**			R/W				Non-Shared						
M4	3	Grading	Entity Type				R/W			Non-Shared						
		Schedule	**Interrelationship**				R/W			Non-Shared						
		Participate	**Interrelationship**				R/W			Non-Shared						
M5	5	Training	Entity Type					R/W		Non-Shared						
		Has	**Interrelationship**					R/W		Non-Shared						
M6	6	StudentCustomer	**Intergeneralization**	R					R/W	Unidirectional						
		Deliver	**Interrelationship**						R/W	Non-Shared						
		Deliver2	**Interrelationship**						R/W	Non-Shared						
		Makes3	Relationship Type						R/W	Non-Shared						
		Makes4	Relationship Type						R/W	Non-Shared						
		Customer	Entity Type						R/W	Non-Shared						
		Order	Entity Type						R/W	Non-Shared						
		Item	Entity Type						R/W	Non-Shared						
M7	6	Address	Entity Type	R/W					R/W	**Multidirectional**						

Fig. 7. Evolutionary database modularization design phase until sixth iteration.

Figure 8 shows the extension of M1 module with "Lives" relationship type as an interrelationship. Also, a new database module M7 accommodates "Address" entity type, with a multidirectional sharing, since both S1 and S6 subsystems perform writing operations. Intergeneralization avoids over modeling, like it would have been the case if a "Person" entity type was upfront created at the first iteration. It allows independent evolution of entity types that would participate in a same generalization hierarchy. This is achieved by representing most detailed level of data abstraction, creating the specific entity types only.

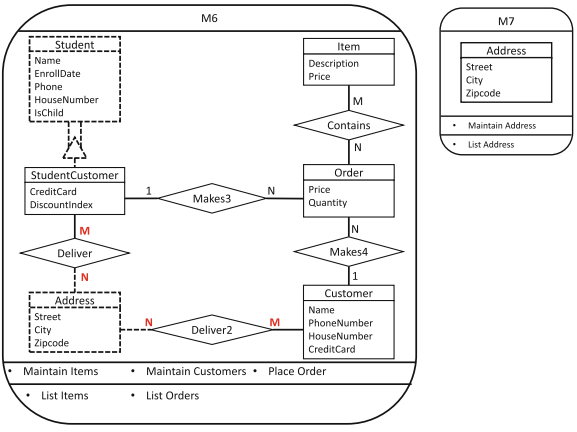

Fig. 8. M6 and M7 modules.

7 Conclusion

Preserve software system integrity through an evolutionary process demands a great deal of effort, especially due to two different paradigms: coding and database, whose schema must evolve to support new functionalities. Therefore, in a constantly changing environment it is ideal to have an evolutionary database design process. Thus, this paper presented the evolutionary database modularization design process.

At first our design increases the abstraction level of the general schema. First, because the process is conducted at conceptual design phase; secondly by introducing two new data modeling concepts, interrelationship and intergeneralization.

Interrelationships, can stand further multiplicity changes without refactoring. That is also the case of intergeneralizations, that evolve independently in each module.

The scope of change assessment is narrowed down within each database module. In addition it improves traceability, since a database module representation includes both the conceptual schema and its related transactions.

We presented an Integration Object Catalog to store database modules definition. Therefore, it is possible to use database modules metadata to assist the evolution process of database modules.

In short, a standardized evolutionary database design method is provided facilitating further schema evolution. Agile projects can benefit from our approach due to its incremental and iterative characteristics.

Future work includes the extension of evolutionary database modularization design process to intra-modules changes, when they occur inside a database module. Further, the creation of a database module definition language. Another issue concerns the coexistence of multiple database schemas for a certain period. In this situation, public and private procedures could encapsulate the transactions, making schema changes transparent to the application. This can be achieved by implementing an integration object layer to handle transactions.

References

1. Ambler, S., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison Wesley, Upper Saddle River (2006)
2. Batini, C., Ceri, S., Navathe, S.B.: Conceptual Database Design: An Entity-Relationship Approach. Addison Wesley, California (1991)
3. Sommerville, I.: Software Engineering. Addison-Wesley, Harlow (2007)
4. Beck, K., et al.: Manifesto for Agile Software Development (2001). http://agilemanifesto.org. Accessed 2 Mar 2015
5. Wells, D.: Extreme Programming: A Gentle Introduction (1999). http://www.extremeprogramming.org. Accessed 2 Mar 2015
6. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Boston (2004)

7. Ferreira, J.E., Busichia; G.: Database modularization design for the construction of flexible information systems. In: Proceedings of the 1999 International Symposium on Database Engineering & Applications (IDEAS 1999), pp. 415–422. IEEE Computer Society, Montreal (1999)
8. Guedes, G.B., Busichia, G., Moraes, R.L.O.: Database modularization applied to system evolution. In: Sixth Latin American Symposium on Dependable Computing (LADC 2013), Rio de Janeiro, Brazil, pp. 81–82 (2013)
9. Navathe, S.B., Fry, J.P.: Restructuring for large databases: three levels of abstraction. ACM Trans. Database Syst. **1**, 138–158 (1976)
10. Sockut, G.H., Goldberg, R.P.: Database reorganization-principles and practice. ACM Comput. Surv. **11**(4), 371–395 (1979)
11. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
12. Ambler, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. Wiley, New York (2003)
13. Cleve, A., Hainaut, J.-L.: Co-transformations in database applications evolution. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 409–421. Springer, Heidelberg (2006)
14. Cleve, A., Brogneaux, A.-F., Hainaut, J.-L.: A conceptual approach to database applications evolution. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 132–145. Springer, Heidelberg (2010)
15. Papastefanatos, G., Anagnostou, F., Vassiliou, Y., Vassiliadis, P.: Hecataeus: a what-if analysis tool for database schema evolution. In: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), pp. 326–328. IEEE Computer Society, Washington (2008)
16. Curino, C.A., Moon, H.J., Ham, M., Zaniolo, C.: The PRISM workwench: database schema evolution without tears. In: Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE 2009), pp. 1523–1526. IEEE Computer Society, Washington (2009)
17. Busichia, G., Ferreira, J.E.: Sharing of heterogeneous database modules by integration objects. In: Eder, J., Rozman, I., Welzer, T. (eds.) Proceedings of the Third East European Conference on Advances in Databases and Information Systems (ADBIS 1999), pp. 1–8. Institute of Informatics, Faculty of Electrical Engineering and Computer Science, Smetanova 17, IS-2000 Maribor, Slovenia (1999)
18. Busichia, G., Ferreira, J.E.: Compartilhamento de Módulos de Bases de Dados Heterogêneas através de Objetos Integradores. In: Simpósio Brasileiro de Banco de Dados (SBBD), pp. 395–409. SBC, Florianpolis (1999)
19. Ambler, S.: Agile/Evolutionary Data Modeling: From Domain Modeling to Physical Modeling (2004). http://www.agiledata.org/essays/agileDataModeling.html. Accessed 2 Mar 2015