Metadata of the chapter that will be visualized in SpringerLink

Book Title	Advanced Information Systems Engineering Workshops	
Series Title		
Chapter Title	Adapting Domain-Specific Interfaces Using Invariants Mechanisms	
Copyright Year	2021	
Copyright HolderName	Springer Nature Switzerland AG	
Corresponding Author	Family Name	**Ulitin**
	Particle	
	Given Name	**Boris**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	HSE University
	Address	Moscow, Russia
	Email	bulitin@hse.ru
	ORCID	http://orcid.org/0000-0003-3774-2457
Author	Family Name	**Babkina**
	Particle	
	Given Name	**Tatiana**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	HSE University
	Address	Moscow, Russia
	Email	tbabkina@hse.ru
	ORCID	http://orcid.org/0000-0003-2892-8831

Abstract	The process of implementing agile technologies in an enterprise is an example of a highly demanding and challenging topic both for practitioners and academy. Speaking about agility, we basically mean various kinds of automation. A higher degree of automation results in a more agile enterprise. However, in practice, even in the case of complete automation of the enterprise, there remains a need for user interaction with various software systems. To increase the efficiency and simplify such interactions, it is necessary to develop mechanisms, which enable adaptation of user domain-specific interfaces for actual conditions of human-machine interaction. We consider for this purpose the use of an invariant formalism based on the allocation of stable, unchanging object data structures and interface structures. Evaluation of the effectiveness of the approach proposed is carried out with the example of the analytical system of the university admissions committee.

Keywords (separated by '-')	Organizational agility - Domain-specific interface - GUI - Object-relational schema - UML - Transformation - Invariants

Adapting Domain-Specific Interfaces Using Invariants Mechanisms

Boris Ulitin(✉) and Tatiana Babkina

HSE University, Moscow, Russia
{bulitin,tbabkina}@hse.ru

AQ1

Abstract. The process of implementing agile technologies in an enterprise is an example of a highly demanding and challenging topic both for practitioners and academy. Speaking about agility, we basically mean various kinds of automation. A higher degree of automation results in a more agile enterprise. However, in practice, even in the case of complete automation of the enterprise, there remains a need for user interaction with various software systems. To increase the efficiency and simplify such interactions, it is necessary to develop mechanisms, which enable adaptation of user domain-specific interfaces for actual conditions of human-machine interaction. We consider for this purpose the use of an invariant formalism based on the allocation of stable, unchanging object data structures and interface structures. Evaluation of the effectiveness of the approach proposed is carried out with the example of the analytical system of the university admissions committee.

Keywords: Organizational agility · Domain-specific interface · GUI · Object-relational schema · UML · Transformation · Invariants

1 Introduction

Engineering of digital transformations is an urgent problem for modern enterprises, especially within the Industry 4.0 paradigm [1]. This process is particularly important in the case of the 'agilisation' [18] of enterprise processes through the application of different smart components and artificial intelligence systems. As a result, the degree of autonomy of the enterprise components increases as well as the intensity of their interaction [1].

At the same time, speaking about agility, we primarily mean by this process various kinds of automation and 'smartisation'. However, as practice shows, even in the case of complete automation and agility of production, the need for user interaction with different systems remains [2]. First of all, such interaction is necessary for the accumulation of the knowledge base used during technological cycle. In addition, users may need various sorts of *ad hoc* analytical information from the system. Finally, there are situations that are not foreseen in the algorithms of behavior for the system and require the direct control by the end-user.

To increase the efficiency and simplify such interactions, it is necessary to develop mechanisms, which enable continuous adaptation of user domain-based interfaces for

A. Polyvyanyy and S. Rinderle-Ma (Eds.): CAiSE 2021 Workshops, LNBIP 423, pp. 1–12, 2021.
https://doi.org/10.1007/978-3-030-79022-6_8

actual conditions of human-machine interaction. Existing works [8, 10] pay more attention to automation of optimization algorithms rather than to processes of interaction with systems. The work [9] exemplifies a common opinion that the user interface the user interface is developed immediately as full and general as possible and subsequently it can only be changed manually. Such a statement deprives the user interface of possible flexibility and leads to cases when the user interface begins to contradict the conceptual model of the domain.

In previous works we substantiated that any interface can be interpreted as a kind of domain-specific language (DSL) [16], and an object-oriented paradigm naturally represents the interface, since it reflects elements of the conceptual model of the domain. Thus, when solving the problem of adapting an interface, models and methods common to the DSL approach can be applied. This leads us to the idea that the interface also has a model-oriented nature, which means that it is possible to automate the process of its incremental modification during the evolution of a conceptual domain model.

Within a generic framework of DSL development this article aims at proposing a new approach to automation of development of adaptive domain-specific interfaces using transformations between the UML-model and Object-Relational Schema restrained by a certain invariant formalism. In our case the UML-model is used to describe the component structure of the user interface through which an Object-Relational Schema containing domain data is filled. Invariants represent stable structures, the correspondences between which are established at the level of both models [13]. A certain transformation is used to shift between identified invariants of both models and to adapt the interface components to changes in the domain model. This approach allows us to fully automate the interface designing that results in reducing the development time of the software system as a whole. In addition, due to the formal model of invariants [14], the interface structure can always be consistently adapted to changes in the domain model.

In our research we evaluated the approach proposed within the task of incremental modification and adaptation of reporting interfaces in an analytical software system for support of admission process in a higher education institution.

Compared to previous works [16, 17], this article contains a practical implementation of the idea of using invariants for the domain-specific interfaces evolution and presents results as follows. In Sect. 2 we observe main aspects of representation of user interfaces by UML class-diagram and show correspondence between the UML class-diagram and the Object-Relational Schema. Section 3 contains the description of the proposed approach, specifies the invariants of the Object-Relational Schema and the interface model and determines the transformations between them. Section 4 is devoted to the application of proposed approach. We cosect.clude the article with the analysis of the proposed approach and further research steps.

2 Key Concepts of the Research

2.1 Object-Oriented Definition of User Interface

Object-oriented representation of the Graphical user interface (GUI) in general settles as part of the object-oriented analysis and design (OOAD) approach. It's a structured method for analyzing, designing a system by applying the object-oriented concepts,

and develop a set of graphical system models during the development life cycle of the software [3].

From this point of view, any system is represented as a set of interconnected components (objects), characterized by their attributes and behavior.

According to this idea, we separate the system into *objects*, each of which has.

- an identity (id) which distinguishes it from other objects in the system;
- a state that determines the attributes of an object as well as each attribute values;
- behavior that represents available activities performed by an object in terms of changes in its state.

Objects containing the same attributes and/or exhibit common behavior are organized into *classes*, which contain a set of attributes for the objects that are to be instantiated from the class and operations that portray the behavior of the objects of the class.

Therefore, we can formalize any class as a combination (O, R) of some objects and relations between them, where each object is a set of its attributes (one of the attributes is unique and is considered an identifier) and operations $o_i = (Attr_i, Opp_i) = \left(\{attr_{i_1}, attr_{i_2}, \dots, attr_{i_M}\}, \{opp_{i_1}, opp_{i_2}, \dots, opp_{i_K}\}\right), M, K \in \mathbb{N}, i = 1, N)$.

Thus, the system is represented as a superposition of many objects of various classes. Being an integral part of the system, GUI can also be represented as a combination of the number of related objects of various classes. It is important to note that attributes may be elementary or complex [3]. Complex is an attribute that is an instance of a class (object). Elementary is an attribute containing a constant value that is not an object. Given this classification of attributes, we can argue that the system is a hierarchy of interconnected objects. At the lower level of such a hierarchy there are objects containing only elementary attributes, and at the top - the system itself. Such basic lower level objects that cannot be dissected into smaller components, make up system object invariants. Such object invariants make it possible to describe the structure as a superposition of low-level invariants. Functionals used to construct structures from object invariants form an operational invariant. These types of invariants are described in more detail in Sect. 2.2.

In this case, each object and, as a consequence, the GUI (as well as the system in general), represents a combination of many object invariants that do not change during the work on the system and can be used to automate the development process. Assuming that the GUI is tied to a certain data set, we can identify the invariants at the data level as well as at the level of the GUI. As a result, the construction of the GUI becomes nothing else than the identification of the invariant at the data level with the subsequent search and display of its equivalent at the interface level.

To better understand the mechanisms of automation through the use of invariants, it is necessary to precise their definition and develop a classification of invariants.

2.2 Definition and Classification of Invariants

First of all, it should be noted that there are at least 4 options for determining invariants depending on the context of use. This paper uses three classic forms of invariants: object-oriented (object), inductive and operational [13, 14].

Declarations of *object invariants* can appear in every class. The invariants that pertain to an object o are those declared in the classes between ***object*** – the root of the single-inheritance hierarchy – and ***type***(o) – the allocated type of o. Each object o has a special field *inv*, whose value names a class in the range from ***object*** to ***type***(o), and which represents the most refined subclass whose invariant can be relied upon for this object. More precisely, for any object o and class T and in any execution state of the program, if $o.inv$ is a subclass of T, which we denote by $o.inv \leq T$, then all object invariants declared in class T are known to hold for o. The object invariants declared in other classes may or may not hold for o, so they cannot be relied upon.

Inductive invariants describe the connection between components of two (or more) sets of objects and are denoted with inv_τ. The formal definition of the inductive invariants was demonstrated in [17]. An inductive invariant means, that there is a strong correspondence between elements of two sets of objects, which are connected during some relation (transformation).

In order to guarantee that the obtained set of interface invariants will be consistent and display the corresponding set of data model invariants, the third type of invariants is used. Interpreting each modification of an interface as its transformation, it can be argued that its sequential refinement is a consistent application of the transformation function. From this point of view, the process of interface development consists of the execution of the operational invariants and can be associated with a subset $\mathcal{O}.F$ of $(\Sigma, \mathrm{F})^{\omega}$ of (in)finite sequences of states, which were identified and analyzed in our previous work [17].

Informally, $\mathcal{O}$ consists of those sequences of states that begin with an initial state that satisfies $\mathcal{J}$ (*Neg* negative application condition in our case) and in which each state has a successor in accordance with the transition $\mathcal{W}$ (*Rule* transformation rule in our case). From this point of view, $\mathcal{O}$ can be interpreted as a transition system between various states, that satisfies $\mathcal{J}$. The set $\mathcal{O}$ is nonempty because $\mathcal{J}$ is satisfiable and $\mathcal{W}$ includes stuttering steps.

Once the computations of a transition system are built, *next* and *inv* specifications are defined as expected:

$$next_{\mathcal{O}}.(p,q).F \triangleq \forall \sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p, \sigma_i \Longrightarrow q, \sigma_{i+1} \tag{1}$$

$$inv_{\mathcal{O}}.p.F \triangleq \forall \sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p, \sigma_i \tag{2}$$

Informally, $next_{\mathcal{O}}.(p, q)$ means that, in any modification of the system, any state that satisfies p is immediately followed by a state that satisfies q. Although modifications include stuttering steps, $next_{\mathcal{O}}.(p, q)$ does *not* imply that $\left[p \Rightarrow q\right]$. In the same way, $inv_{\mathcal{O}}p$ means that any state of any modification of a system satisfies p. Naturally, $next_{\mathcal{O}}$ and $inv_{\mathcal{O}}$ are related in a way similar to the relationship between $next_\tau$ and inv_τ, namely: $inv_{\mathcal{O}}.p.F \equiv next_{\mathcal{O}}.(p, q).F \wedge [\mathcal{J}.F \Rightarrow p]$.

The only thing is to determine the invariants at the level of the data model and interface and establish a correspondence between them. For this, it is necessary to compare the elements of the Object-Relational Schema and the object-oriented interface model.

The Object-Relational Schema looks effective in this case, since any interface involves some interaction with domain data. Taking into account the object nature of the Object-Relational Schema, this interaction can be easier to organize by establishing a correspondence between data objects and the interface.

2.3 Structural Description of UML Class Diagram and Object-Relational Schemas

Although the UML class diagrams [3] have many elements to model, we have selected the more commonly used for a database schema design, which are [4]: (1) Classes (C); (2) Attributes: single (SA), composed (CA), multivalued (MA); (3) Operations (Op); (4) Relationships (Rel) between classes: aggregation (AG), composition (CM), n-ary association (NAS), binary association (BAS), association class (AC), generalization-specialization (GS).

In defining the individual components of this diagram, we will adhere to the concepts described by M.F. Golobisky and A. Vecchietti [11].

Since we defined class C earlier as set of identifier, state and behavior, here we only decompose its definition as follows: $C = (id, C, SA, CA, MA, O, R)$, where: id is the name assigned to the class; C is a finite set of classes due to the fact that a class can be composed of other classes; SA, CA, MA are finite sets of single, composed and multivalued attributes correspondingly; Op is a finite set of operations and Rel is a finite set of relationships where the class is participating. The definition of various types of relations and their formalization are presented in more detail in [11].

After the structure of the UML class diagram is determined, we need to consider in more detail the structure of the Object-Relational Schema. Components of this model are mainly defined in the SQL standard [12]. Only the most relevant parts of this standard for this paper are considered, which are: row type, collection types (arrays and multisets) and a reference type, since they are responsible for relational object (tables) structure definition.

A row type is defined as a set of pairs $RT = (F_1: T_1, F_2: T_2, \ldots, F_n: T_n)$ where F_i is the name of a field in the row type and T_i is a built-in data type (int, float, etc.). A Reference Type called $Ref(T)$ is also a data type which contains the reference values (OID) of T. T is a row in a typed table [6].

Upon initial consideration, both models have an object-oriented nature, that results in the opportunity to establish a correspondence between them at the level of objects.

3 Description of the Approach Proposed

In our research we prove that the object-oriented model of the interface (in terms of the UML Class Diagram) can be derived through the transformations from the Object-Relational Schemas. For this, it is vital to establish a correspondence between the components of these models.

There are three layers involved in the transformation from UML class diagrams into persistent objects of the Object-Relational Schema (Fig. 1). The first one corresponds to the UML classes and relationships. The second object-relational layer contains the objects proposed by the SQL designed to implement classes and relationships. Finally, the third layer is composed of typed tables with keys, constraints, triggers (relational approach) and other elements like object identifiers (OID).

To complete the transformations, it is necessary to define several functions to translate the components from one layer to the other, using a mapping function. This function is a

binary relation $f: A \rightarrow B$, where A and B are invariants from mapping models. The most important property of this function in conjunction with invariants is that each element in A maps to exactly one element in B [7].

In the case of UML classes and the Object-Relational Schema the following correspondence can be used (Table 1). The relationships cannot be transformed in the same way than the other elements of the class. Such limitations are caused by several alternatives mapping functions can be applied according to the characteristics of the relationships and the classes involved on it.

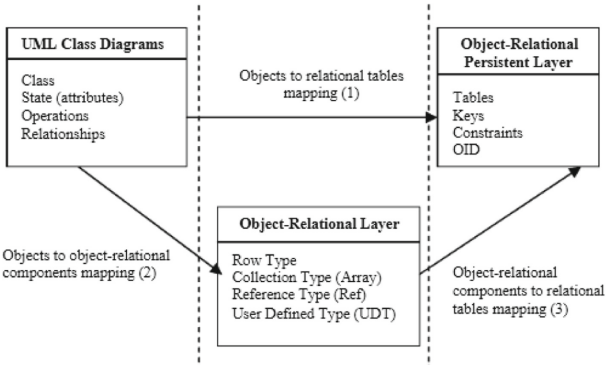

Fig. 1. Layers involved in mapping objects into the Object-Relational Schema

Since an association class is both an association and a class, the mapping function is like the one defined for a class plus the references to the classes to which the association class is linked: $f: AC = (id, Ref(C_1), Ref(C_2), \ldots, Ref(C_n),\ SA, CA, MA, Op) \rightarrow T = (\text{'}name\text{'}, Ref(C_1), Ref(C_2), \ldots, Ref(C_n), BIT, RT, AT, T, MM)$ where the T defined for the association class contains the references to the classes related by the association plus the built-in (BIT) data types, row types, array types, Ts, and member methods of the association class.

After we have established a correspondence between the components of the UML Class diagram and the Object-Relational Schema, the only thing is to formalize the transformation between them in the form of invariants.

For this purpose, we use two types of previously reviewed invariants: object and inductive. First of all, we reveal object invariants at the level of the data model and the interface. Then, we define a function that will analyze the invariants in the data model and find the corresponding interface-level invariant for them. Such invariant is added to the set of previously found interface components and is displayed on the screen.

To identify the object invariants of both levels (Class Diagram and Object-Relational Schema) Table 1 can be used. On the Class side, attributes are invariants, which can also be simpler classes. Following this logic, we can argue that in the case of an interface, the invariant will be classes that describe its individual components and do not contain references to other classes.

Table 1. Proposed correspondence of elements of classes and tables.

Class	Table
Single attribute	Built-in type
Composed attribute	Row type
Multivalued attribute	Array type, multiset type
Operation	Member method

In the case of the Object-Relational Schema, the invariants are columns and their types, which are fully consistent with class attributes. These objects are object invariants. The transformation function between interface elements and table columns described above is an inductive invariant. Using this function, we can design the interface based on an Object-Relational Schema in the form of a complete invariant scheme.

Such a definition of the interface development process as a sequential application of the transformation function defined by inductive invariants allows us to state that the final state of the interface will fully correspond to the data model. That provides the ability to automate the process of developing interfaces and their adaptation in the case of a change in the structure of relational tables, since it is tied not to the structure of the table as a whole, but to separated invariants (columns).

As a result, the process of the interface development in form of composition of the invariants can be described in pseudo-code as follows (Fig. 2).

```
...
foreach column_invariant do
  foreach interface_invariant do
    if mapping_function(column_invariant) == interface_invariant then
      add interface_invariant onto GUI
...
```

Fig. 2. Pseudo-code description of algorithm within the approach proposed

In this case we sort through all the invariants of the columns of the Object-Relational schema and look for a correspondence among the invariants of the interface. If a match is found, the corresponding interface invariant is added to the display. Otherwise, the transition to the next interface invariant occurs.

4 Application to the Admissions Committee Domain

The application process in higher education organizations should be as simple and fast as possible and meet all the criteria for agility. The information system, which supports that process, contains about 50 relational tables (containing from 4 to 30 attributes) connected by more than 80 relationships.

The set of necessary analytical indicators expands every year according to the modification of the admission rules, that results in a change of the data set collected during the admission process.

In these conditions, it is a critical need for continuous refinement of not only the Object-Relational Schema of data collected in the admission process, but also user interfaces. In accordance with the previously described approach, it is necessary to distinguish the invariants at the level of the Object-Relational Schema, associate them with the invariants at the interface level and implement the mapping function between them.

The invariants that stand out at the level of the Object-Relational Schema were discussed in the previous section, so here we focus only on their practical implementation. It is important to note that although we have identified the table column as an invariant (namely, its type), the specific implementation of this invariant differs depending on the database management system (DBMS) chosen.

In our case, the DBMS MS SQL Server is used, therefore we will consider the main data types specific to this environment. Based on the data schema described in Fig. 1, we can identify the main types of data necessary for the task: int, float, date, etc.

After the invariants are defined at the level of the Object-Relational Schema, it is necessary to distinguish the corresponding invariants at the interface level. As noted earlier, an interface is a collection of interconnected components, each of which is tied to a corresponding data block. For the convenience of the user, it seems useful that the invariant for each individual data type contains not only the value of the corresponding data type, but also the name of the column from which this data is extracted. In this way, the user can see not only the value, but also understand which name column of which table it refers to. As a result, editing values and their processing become more understandable.

Based on this idea, a set of the object invariants was developed at the interface level in accordance with the relational invariant (Fig. 3 demonstrates the example of int invariant). These invariants are fully consistent with the previously identified structure and formalization of invariants.

Fig. 3. A visual invariant at the interface layer for the int type relational invariant

The invariant system developed is complete and fully consistent with the invariants identified at the level of the Object-Relational Schema. Also, this invariant system is stable, because it is not tied to the structure of tables of the real-time model, but only to data types, which makes it easily scalable.

After the system of invariants is developed, it remains only to determine the correspondence function between them. This function determines which invariant of the Object-Relational Schema was input, and finds the corresponding interface level invariant. Taking into account that the development of the system is carried out in Java (and patterns are written in Java FX), the part of the function may look as follows (Fig. 4).

```
switch (typePattern) {
   case "boolean":
    loader.setLocation(getClass().getResource("BoolInputPattern.fxml"));
    ...
    break;
   case "date":
    loader.setLocation(getClass().getResource("DateInputPattern.fxml"));
    ...
    break;
    ...
}
```

Fig. 4. A part of the transformation function between invariants

An important result of such a structure is its adaptability. Due to the interconnection of invariants, the interface is built dynamically in real time, without the need for its design and manual programming.

In our case, the interface is a table that displays data from the corresponding database table, as well as fields for interacting with them (adding, editing). Both interface elements are built dynamically as a result of reading the structure of the corresponding table, which the user can select preliminarily (Fig. 5). Using this interface, the user can view, and also add and process data for all applicants in the database. The left part of the interface is a table showing all the data on applicants, while the right panel serves to interact with the content.

We developed a block that allows the end user to make changes to the structure of the database tables. This interface supports the following operations: creating a new table, creating a table based on existing ones (with the ability to select columns), dividing the table into several related ones, editing the structure of existing tables, etc. In the first case, the user determines the name of the table being created, as well as a list of its columns. In the second case, when creating a table, one or several existing tables are taken as the basis, from which columns are selected, the equivalents of which should be created in the new one. It is important to note that in this case the data from the linked columns of the original tables is also copied to the new one, thus the transfer of the data as well as the structure is achieved.

In the case of editing the table structure, we can change its name, as well as change the columns by adding or removing them, changing the type or name. To demonstrate the adaptability of the interface we created, let us add a new column to the table with a list of applicants: the registration code (Fig. 5).

This code contains the encoded representation of information about the applicant. It is calculated on the basis of the general data of the applicant, the number of his competitive points, etc.

After the new attribute is added, we verify that the original interface has adapted to the new table structure (Fig. 6). As we can see, the structure of the GUI has adapted to the changed structure of the database table, as well as all input fields of the interface also now correspond to the new structure. Moreover, we can interact with the modified

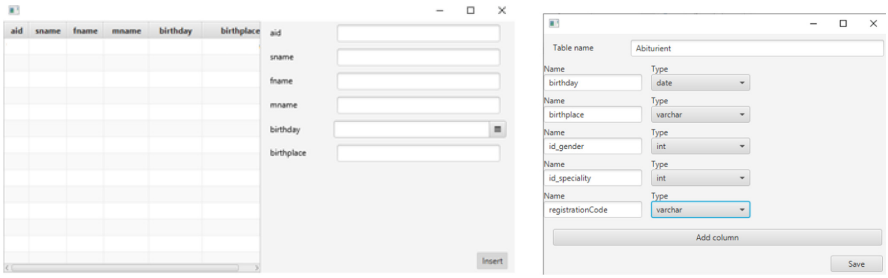

Fig. 5. A part of the GUI, responsible for Object-Relational Schema adaptation

table, for example, enter a value for the applicant in the added column. As a result of the input, we obtain a new state of the interface (Fig. 6).

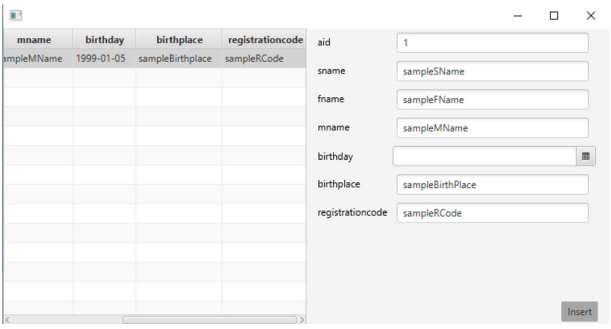

Fig. 6. The part of the GUI after update of data in added column

As a result of the application of our approach the interface is fully adapted to the new structure. Most importantly, we did not manually make any changes to the program code or to the structure of relational tables. All changes are made through the interface. This allows users to make the system fully adaptable and customizable, without the need for special skills in programming and designing Object-Relational Schemas. Such flexibility is achieved by introducing into the software prototype mechanisms of invariants that are unchanged with any change in the system as a whole. As a result, any change in both the structure of tables and the interface is described in terms of invariants. And the interface itself is built dynamically based on an updated set of table invariants. This significantly reduces the time required to adapt a data-based system and allows end users to be involved in this process.

5 Conclusion

Lean adoption is a trend in many domains, from enterprises to small academic projects [18]. Since many enterprises are automated and use various software systems, agility

results in the need for a mechanism to adapt the interfaces of these systems to the requirements of end users to improve efficiency and simplify human-cyber interaction.

This article examines the process of adapting interfaces by introducing invariant mechanisms. The approach is based on the idea that any user interface in accordance with an object-oriented paradigm can be represented as a composition of objects corresponding to individual components of the system. Each such object is an object invariant, since it is preserved when any changes are made to the system and interface. This is possible, since any interface can be considered as a special kind of domain-specific language. As a result, all methods specific to the adaptation of DSL are applicable to it.

This ability to automatically adapt the interface is important in the context of enterprise digitalization and agility. It is especially important in the case of the introduction of different smart systems and the organization of interaction with the end user [2].

On the other hand, any interface is tied to a certain set of data that can be described as an Object-Relational Schema. At the level of the Object-Relational Schema, we can also distinguish invariants that are unchanged during its evolution. Such invariants are table columns and their types.

By mapping these invariants with each other, it is possible to automate the interface development by means of a correspondence function that reads the invariant of the Object-Relational Schema and displays the corresponding interface level invariant. As a result, the need to manually create an interface disappears, since it is built as a composition of individual invariants in real time. Such automation is achieved due to the fact that the interface is constructed as a superposition of object invariants. As a result, the interface structure is dynamically built in real time based on the data structure.

In comparison with existing solutions [5, 8, 10], the approach proposed allows developers to fully automate the process of making changes to the Object-Relational Schema. This extends the adaptation capabilities described in [8], since our approach does not require the preservation of the data structure used by the interface, but allows them to be changed in real time. Our case shows that the need to interact with developers to change the data structure is completely eliminated. Furthermore, the proposed approach allows users to implement all the changes without programming skills, since all work is carried out through the interface.

In further research it is planned to expand the set of supported functions for managing the Object-Relational Schema, making it more complete. This will fully ensure the entire cycle of work with the Object-Relational Schema via the interface, without the need for preliminary database creation, even in the initial version.

References

1. Heavin, C., Power, D.J.: Challenges for digital transformation – towards a conceptual decision support guide for managers. J. Decis. Syst. **27**(1), 38–45 (2018)
2. Ruffolo, M., Sidhu, I., Guadagno, L.: Semantic enterprise technologies. In: Proceedings of the First International Conference on Industrial Results of Semantic Technologies, vol. 293, pp. 70–84 (2007)
3. Hayat, S.A.E., Toufik, F., Bahaj, M.: UML/OCL based design and the transition towards temporal object relational database with bitemporal data. J. King Saud Univ. Comput. Inf. Sci. **32**(4), 398–407 (2020)

4. Bashir, R.S., Lee, S.P., Khan, S.U.R., Chang, V., Farid, S.: UML models consistency management: guidelines for software quality manager. Int. J. Inf. Manag. **36**(6), 883–899 (2016)
5. Lazareva, O.F., McInnerney, J., Williams, T.: Implicit relational learning in a multiple-object tracking task. Behav. Proc. **152**, 26–36 (2018)
6. Wu, Y., Mu, T., Liatsis, P., Goulermas, J.Y.: Computation of heterogeneous object co-embeddings from relational measurements. Pattern Recogn. **65**, 146–163 (2017)
7. Torres, A., Galante, R., Pimenta, M.S., Martins, A.J.B.: Twenty years of object-relational mapping: a survey on patterns, solutions, and their implications on application design. Inf. Softw. Technol. **82**, 1–18 (2017)
8. Wang, N., Wang, D., Zhang, Y.: Design of an adaptive examination system based on artificial intelligence recognition model. Mech. Syst. Signal Process. **142**, 1–14 (2020)
9. Konyrbaev, N.B., Ibadulla, S.I., Diveev, A.I.: Evolutional methods for creating artificial intelligence of robotic technical systems. Procedia Comput. Sci. **150**, 709–715 (2019)
10. Leung, Y.: Artificial Intelligence and Expert Systems. In: International Encyclopedia of Human Geography, 2nd Edn., pp. 209–215 (2020)
11. Golobisky, M.F., Vecchietti, A.: Mapping UML class diagrams into object-relational schemas. In: Proceedings of Argentine Symposium on Software Engineering, pp. 65–79 (2005)
12. Köhler, H., Link, S.: SQL schema design: foundations, normal forms, and normalization. Inf. Syst. **76**, 88–113 (2018)
13. Chen, Y., Tang, Z.: Vector invariant fields of finite classical groups. J. Algebra **534**, 129–144 (2019)
14. Carvalho, J.F., Pequito, S., Aguiar, A.P., Kar, S., Johansson, K.H.: Composability and controllability of structural linear time-invariant systems: distributed verification. Automatica **78**, 123–134 (2017)
15. SQL Standard 2016 (ISO/IEC 9075-1:2016). https://www.iso.org/committee/45342/x/catalogue/p/1/u/0/w/0/d/0
16. Ulitin, B., Babkin, E., Babkina, T.: A projection-based approach for development of domain-specific languages. In: Zdravkovic, J., Grabis, J., Nurcan, S., Stirna, J. (eds.) BIR 2018. LNBIP, vol. 330, pp. 219–234. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99951-7_15
17. Ulitin, B., Babkin, E., Babkina, T., Vizgunov, A.: Automated formal verification of model transformations using the invariants mechanism. In: Pańkowska, M., Sandkuhl, K. (eds.) BIR 2019. LNBIP, vol. 365, pp. 59–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31143-8_5
18. Küpper, S., Kuhrmann, M., Wiatrok, M., Andelfinger, U., Rausch, A.: Is there a blueprint for building an agile culture? In: Projektmanagement und Vorgehensmodelle 2017-Die Spannung zwischen dem Prozess und den Mensch im Projekt (2017)

Author Queries

Chapter 8

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author and email address has been identified as per the information available in the Copyright form.	