Agent-Based User Interface Generation from Combined Task, Context and Domain Models

Vi Tran[1], Manuel Kolp[1], Jean Vanderdonckt[1], Yves Wautelet[1], and Stéphane Faulkner[2]

[1] Louvain School of Management-PRISME, Université catholique de Louvain, Louvain-la-Neuve, Belgium
{Vi.Tran,Manuel.Kolp,Jean.Vanderdonckt, Yves.Wautelet}@uclouvain.be
[2] Louvain School of Management-PRISME, University of Namur, Louvain-la-Neuve, Belgium
Stephane.Faulkner@fundp.ac.be

Abstract. User interfaces (UI) for data systems has been a technical and human interaction research question for a long time. Today these user interfaces require dynamic automation and run-time generation to properly deal with on a large-scale. This paper presents an agent-based framework, i.e., a methodological process, a meta-model and a computer software to drive the automatic database user interface design and code behind generation from the task model, context model and domain model combined together. This includes both the user interface and the basic functions of the database application.

Keywords: Task Model, Domain Model, Context Model, Automatic Generation, User Interface, Agent Software.

1 Introduction

Automatic user interface generation has been investigated by numerous Human Computer Interaction (HCI) research works due to its capability and importance in the UI development process providing benefits such as low-cost and rapid implementation. There are currently numerous and various approaches using different input materials to automate UI generation such as techniques based on architectural designs, design patterns, use-case scenarios, declarative models[2, 3, 4].

In order to generate user interface, models are usually not used independently from other ones. For instance, TOOL[4] combines the task and user models; GOLIATH [2] combines the application, presentation and dialogue models together to generate the user interface; other research generate automatically the user interface from combining the domain and use case models [5]. Combining models is, as a matter of fact, an important concept in UI generation: the different models describe different aspects of the user interface [7] and serve specific purposes at different stages of the design process.

This research hence proposes a framework, i.e., a methodological process, a meta model and a software to drive the automatic database user interface design and code behind generation from the task, domain and context models combined together.

Our framework is based and supported by declarative technologies. More specifically, we will adopt the agent paradigm (models, language, methods, …) to analyze

D. England et al. (Eds.): TAMODIA 2009, LNCS 5963, pp. 146–161, 2010.

task, context and domain models and generate the user interface specifications and application code.

The rest of this paper is organized as follows: we present, in Section 2, the motivation of this research. In Section 3, we proposes our automatic UI and code generation process taken together the task, context and domain. Section 4 explains the roles of the main agents that participate in this process such as the query analyzer, the UI designer, the code generator for both the data reviewing and editing. Finally, we propose some conclusions and plan for future work.

2 Motivation

This section motivates use of the Task, Context and Domain models to generate the UI and Application codes and presents what is an agent systems.

2.1 Multi-model Analysis

Two main reasons justify the automatic generation of user interfaces from task, domain and context models. The first one is the commonality of these models: in order to develop a software system, the developer usually starts with building the task, domain and context models so that it is convenient to generate UI on the basis of these existing resources. The second one belongs to the nature of the models itself; indeed they are inherently part of the UI design, more precisely:

- The task model expresses how a end user may want to interact with a system in order to reach a given goal; this expression is intended to be independent of any particular implementation or technology. This explains why a same set of models could initiate several different user interfaces. Therefore, the task model is used to specify a generic user interface.
- The domain model defines the aspects of the application which can be adapted or which are otherwise required for the running of the system. Therefore, the domain model is used to specify the control of this user interface – at this level the user interface is specified in more detail.
- The context model describes the user abilities, the environment in which the user works and the platform that can be used for interacting with the system [11]. Therefore, the context model is used to influence the design and to select among alternative solutions in the design space.

2.2 Forward Engineering: Code Generation

Most of the research works have only focused on the user interface generation and omits the importance of the application code generation. To produce an accurate application at low cost, the developers expect that both the user interface and the application code are automatically generated. The application code is generated to perform the generic tasks of the database application, especially, those tasks related to editing, inserting, deleting and searching data.

The present research focuses on the application code generation for the tasks of a data-oriented application in the following context:

- When the demand for data manipulation is very high. Data manipulations are regularly repeated in most database applications especially for common tasks such as those listed above.
- When functions are easily performed through receiving data from the user, displaying data to the user, executing the different SQL select, insert, update and delete queries for the different user requirements.

2.3 Agent Systems: A Definition

An agent is a declarative entity defined as "a computer system, situated in some environment that is capable of flexible autonomic action in order to meet its design objective" [8]. The declarative nature of an agent can be characterized by its autonomy, flexibility and situateness [9]. The autonomy of agents reflects their social and decentralized nature. The flexible way in which agents operate to accomplish their goals is particularly suited to these business systems which are normally expected to function in an ever-changing environment. The situateness implicitly supposes that agents are not useful as stand-alone entities. As a matter of fact, they are most of the time structured in a multi-agent system (MAS).

The global behavior of a MAS derives from the interaction between the constituent agents: they cooperate, coordinate or negotiate with one another. A multi-agent system is then considered a society of autonomous, collaborative, and goal-driven software components, the agents, like a social organization. The agents intelligently and automatically behave and collaborate to fulfill the objectives of a declarative structure [6]. Each role that an agent has the ability to play has a well defined set of responsibilities (goals) achieved by means of an agent's own abilities, as well as its interaction capabilities.

3 Engineering UI from Domain, Context and Task Models

Our process is organized around the components that will be overviewed in this section and described in detailed in Section 4. The different components play different roles; they interact one another by sending the events and transferring the information. Besides, the components use the resources database, mapping rules base [15], layout-knowledge base and so on to achieve their goals. The pro-active and adaptable aspects of the components motivate the adoption the agent paradigm [1] to analyze the models and generate the UI and code. Each component is described as an agent and the various components interact with each other to compose a MAS architecture.

Fig. 1 depicts the main components of our UI and code generation architecture. The *Model analyst* agent uses the *task-*, *database-* knowledge bases and the database itself to analyze the *task* and *domain models* to derive sub-tasks, domain objects and their attributes; the *context model* is also loaded by the *Model analyst* agent. The *Function analyst* agent uses the *Function description* base to define the basic functions of the application. The loaded tasks have to be manually linked to the attributes of the domain objects and to the function defined by the system. From these linked objects, the *UI creator* agent automatically creates the user interface objects based on the *mapping rules*. Once the UI objects have been created, the *code generator* agent generates the code that will implement the UI. Specially, as already pointed out the idea is to not only generate the user interface code, but also the application code behind needed to perform these pre-determined tasks.

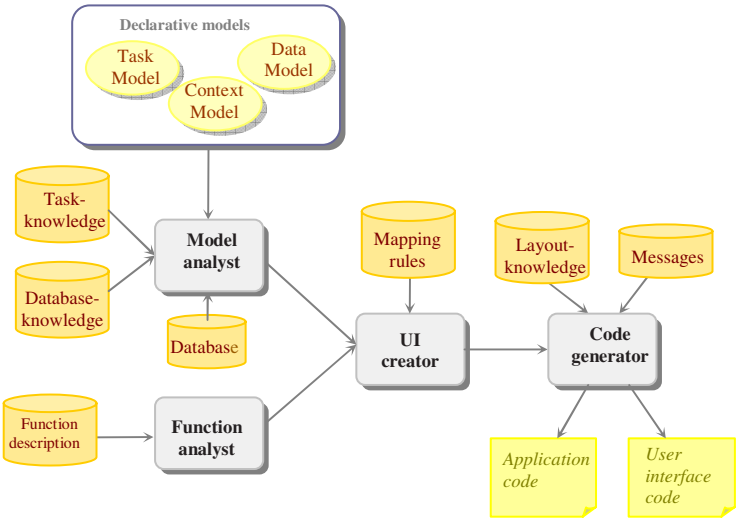

Fig. 1. Main components of user interface generation Architecture

The *model analyst* agent is used to load the task, domain and context models.

In order to obtain the desired behavior of a database application task, the *Function analyst* agent defines the basic functions of an application by using the function description base.

Once the tasks in the task model have been linked to the attributes of the domain objects in the domain model, Concrete Interaction Objects (CIOs) [16] are created based on the attributes characteristics and the relationships between the domain objects by the *UI creator* agent. These characteristics are for instance the data types, data length, is-key flag. Once the CIOs have been created, they are transformed into Final Interaction Objects [16] (FIOs). A FIO is described as a user interface control unit in a concrete platform.

The goal of determining Abstract Interaction Objects [16] (AIOs) is to create CIOs for multi-devices, multi-platforms … This paper focuses on the UI generation for the desktop and laptop devices. Therefore, in our process, the AIOs are not considered.

Finally, the *code generator* agent uses the *Layout-knowledge* base to generate the user interface code based on the FIOs and uses the *Message* base to generate the application code based on the defined functions. The application code is generated to perform the tasks linked to the functions which are defined by the *Function analyst* agent.

In summary, the components of our UI and Code Generation Architecture are:

- The Database used to obtain the information on and of domain model.
- The Task-knowledge base that describes the rules of the task model.
- The Mapping rules base that describes the rules for specifying the CIOs from domain objects and the relationships between these objects and for transforming the CIOs to the FIOs. These rules are described in [15].

- The **Database-knowledge base** that describes generic aspects of the database tasks, the advantages of the syntax and the structure of a query.
- The **Layout-knowledge base** that contains the syntactic design guidelines for controls, windows and other widgets layouts. It also describes the semantic rules from which the control types are defined.
- The **Messages base** that contains the generic messages such as errors, warnings, information to users messages and so on.
- The **Function description base** that describes the basic functions of a database application. For instance, in order to insert the data into a database it has to create a function Insert() which is used to get the data from end user and to input them into the database.

Based on the three classes of user, platforms and environment in the context model [11], we decide to choose the class "user" to support the generation of the user interface. Class user describes the users' characteristics.

This paper aims at reusing the existing resources. The domain model is automatically loaded from a concrete database; the task model is automatically loaded from a XML file which is created by the designer by using the ConcurTaskTreeEnvironment (CTTE) [14] at the analyst level. The user interface is automatically generated based on the links made between the tasks and the attributes of the domain's object. These links are made by the developer.

4 User Interface Generation

This section details the role of the agents and how they communicate with each other to achieve their goals.

4.1 User Interface Generation Social Model

The process proposed in Section 3 is detailed in Fig. 2 using i*. In a few words, i* is an organizational modeling framework [10] proposing concepts such as actor (actors can be agents, positions or roles), as well as social dependencies among actors, including goal, softgoal, task and resource dependencies. Each node represents an actor and each link between two actors indicates a dependency (called dependum) between two actors: the depender and the dependee. The depender is the depending actor, and the dependee, the actor who is depended upon. The type of the dependency may be a goal, a softgoal, a task, or a resource. Goal dependencies represent delegation of responsibility for fulfilling a goal; softgoal dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely; task dependencies are used in situations where the dependee is required to perform a given activity; and resource dependencies require the dependee to provide a resource to the depender. As partially shown in Fig. 2, actors are represented as circles; dependums (goals, softgoals, tasks and resources) are respectively represented as ovals, clouds, hexagons and rectangles; and dependencies have the form depender → dependum → dependee.

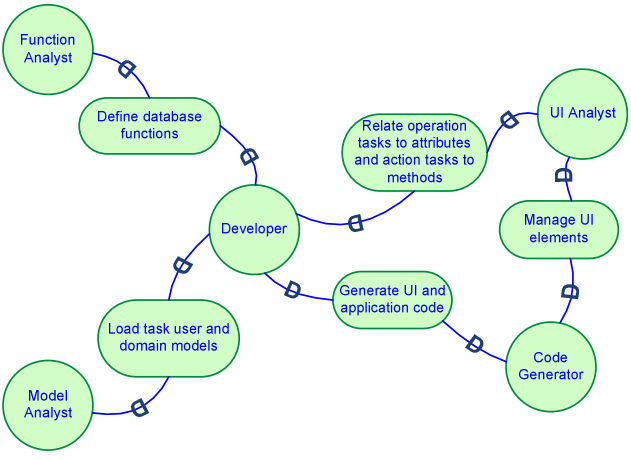

Fig. 2. i* Social Diagram – Processing code generation from task, context and domain models

The ***Developer*** depends on the ***Function Analyst*** agent to define the basic functions of the database application and on the ***Model Analyst*** agent to load the task, user and domain models. The ***UI Analyst*** agent depends on the ***Developer*** to make the links between the tasks and the attributes of the domain's objects. the ***Code Generator*** agent depends on the ***UI Analyst*** to create the UI objects. Finally, the ***Developer*** depends on the ***Code Generator*** agent to generate the user interface code and application code.

<<Agent >> Model Analyst

Plan
LoadTaskModel()
LoadUserModel()
LoadDomainModel()

Belief
XML file
TXT file
Database

Method
Private TaskModel LoadTaskModels(XMLfile f)
Private UserModel LoadUserModel(TXTfile f)
Private DataModel LoadDomainModel(String DbName, String UserName, String Password)

<<Agent >> Function Analyst

Plan
DefineFunctions()

Belief
Function description base

Method
Private void DefineFunctions()

<<Agent >> Developer

Plan
MakeLinkForOperationTask()
MakeLinkForActionTask ()

Belief

Method
Private void MakeOperationLinks()
Private void MakeActionLinks()

<<Agent>> UI Analyst

Plan
CreateCIO()
TransformCIOstoFIOs()

Belief
Mapping rules base

Method
Private Object CreateCIOs()
Private void TransformCIOstoFIOs (Object CIOs)

<<Agent >> Code Generator

Plan
GenerateUICode()
GenerateApplicationCode()

Belief
Layout-knowledge base

Method
Private void SpecifyLanguage ()
Private File GenUICode(Object CIOs)
Private File GenAppCode(Object Methods)

Fig. 3. The Agent Structures

4.2 User Interface Generation Structure

Fig. 3 depicts the structure of the agents in our process. Each agent is structured by its name, plans, the beliefs and methods. These agents are the *Model Analyst*, *Function Analyst*, *Developer*, *UI Analyst* and *Code generator* agents. They will be discussed in more detail below.

4.2.1 Function Analyst Agent

The **Function Analyst** agent uses the *DefineFunctions* plan to define the basic functions which are used to perform the generic tasks of a database application such as add a new record, delete a record, …. These functions are described in Table 1.

Table 1. Defined Functions

Function	Description
Display()	Used to select the data stored in the database and to displays this data to the user
AddNew()	Used to insert a data record into the database
Update()	Used to modify the data of an object in the database
Delete()	Used to delete the data records of an object in the database
Search()	Used to filter the data records based on the some search condition which are determined by the user
Review()	Used to review the data records by displaying the first, next, previous and last record

In order to specify these functions, we design them as design patterns and follow the design pattern framework proposed in [13, 12] for reusability purposes (the generic identified patterns can be reused by others, for instance, patterns objects, attributes and controls can be reused by functions such as Search, AddNew or Display), object-oriented features compliant with the agent paradigm and implementation traceability.

Each function is structured by its name and methods. Functions typically use objects. One object has at least one attribute; one attribute can be associated with more than one control and one control can be associated with more than one attribute. The number of attribute of each object depends on the number of the attributes used to generate the user interface.

Fig. 4 depicts the structure of the functions. These functions use the *GetValuesFromUser()* method to receive data from end-users, the *DisplayResult()* method to display data; the *CreateSQLString()* method to create the SQL strings based on the goals of the functions and the *ExecuteSQL()* method to change data in the database.

One object can have more than one attribute. These attributes are the attributes which are chosen to generate the user interface; they can be a number or all of the original attributes of this object.

One attribute associates with one control which is used to display the attribute's value to or receive the attribute's value from the end-user. One attribute is structured by the name, the data type and the key attribute. For example: attribute Manager{Attribute name: Employee name, Data type: Text, Key attribute: No}.

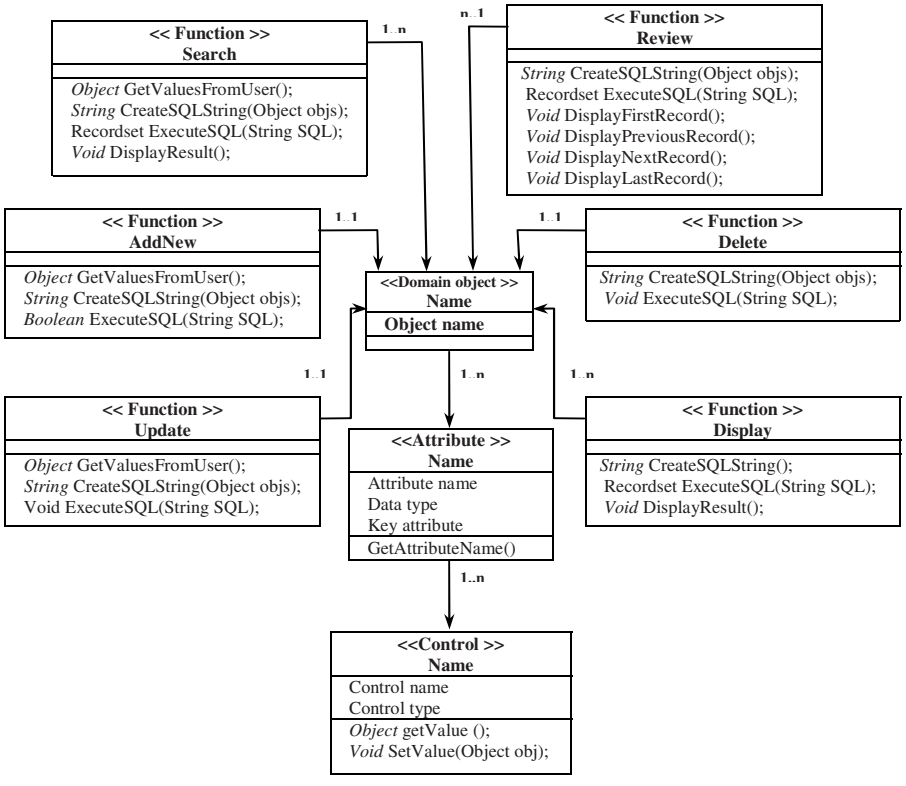

Fig. 4. The Function Structures

One control is structured by the control name, the control type. The getValue() method is used to get data from the end-user and the setValue() is used to display data to the end-user. For example: control Manager{Control name: cbManager, Control type: Combobox}.

In order to obtain the goal, each function has to execute its methods. Each method is executed by the different purposes.

4.2.2 Model Analyst Agent

This agent uses the *LoadTaskModel* plan to load the task model from the XML file, the *LoadDomainModel* to load the domain model from the database and the *LoadUserModel* to load the context model from the TXT file.

The defined categories of tasks at the design level are the action and operation categories. While the categories of tasks, read from the XML file, are abstraction, interaction, cooperation, application and user [13], these categories are defined at the analyst level. At the analyst level, the defined tasks represent different kinds of information including unnecessary ones for UI generation. For instance, the user task is a cognitive task that the end user selects as a strategy to solve a problem or checks the

result; typically this type of task is not used to generate the user interface. Therefore, the User task is mapped to none. (See Tab. 2).

- An **Action task** is a task used to describe the end-user command to the system such as close a dialog, search information, open a dialog and so on.
- An **Operation task** is a task which is used to describe the display of information to end-user or the reception of the information from the end-user.

Table 2. Tasks are mapped as follows

Task mapping	
Analyst level	Design level
Abstraction task	Action task
Interaction task (Type: Control type)	Action
Interaction task (Type: Edit\Monitoring\ selection)	Operation task
Cooperation task	Action task
Application task	Operation task
User task	None

Fig. 5 depicts an example of the mapping between tasks in ConcurTaskTrees and in our process considering a typical AccessStudent Data task. Task *Verify* is not focused; task *AccesstudentData* is automatically mapped to action task; task ShowResults is automatically mapped to operation task; and other tasks are mapped to action and operation tasks based on the type of each task. For example: the task type of *SubmitRequest* is **Control type** so that it is mapped to action task; the task types of *EnterName* and *EnterDepartment* are **Edit** and **Selection** then they are mapped to operation tasks.

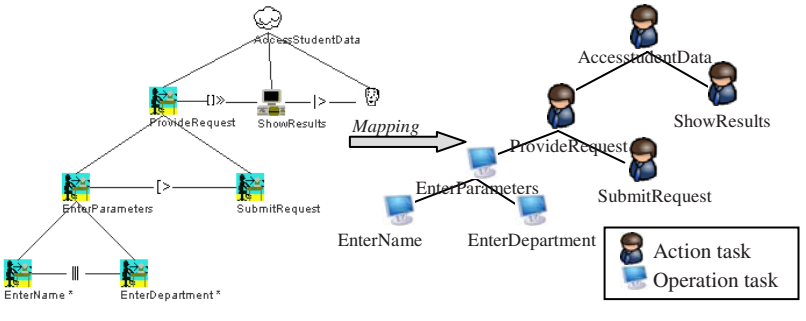

Fig. 5. Example of Mapping: an AccessStudentData task.

The *LoadUserModel* plan is used to analyze the information in the user model to classify the users into three different classes base on their ability to use the software. The analyzed information is the characteristics of the users such as the experience, skill, knowledge, behavior so on. The three classes of the user model are named “Simple”,

"Mean" and "Complex" corresponding to three ability levels for using the software. Based on these user classes, the designer will design a complex, medium or simple user interface.

Finally, the *LoadDomainModel* plan is used to load the domain model from a concrete database which is determined by the developer. The *LoadDomainModel* plan executes the SQL queries to obtain the information of the domain objects (table names), their attributes (column names), aspects of these attributes (column attributes) and relationships between these objects.

4.2.3 Developer

The **Developer** uses the *MakeLinkForOperationTask* and the *MakeLinkForActionTask* plans to make the links between task model and domain model. The *MakeLinkForOperationTask* plan is used to make the associations between leaf operation tasks and attributes of domain's objects. All of the leaf operation task has to be linked to at least one attribute of the domain's. The *MakeLinkForActionTask* plan is used to make the links between the action tasks and the defined functions. One action task is linked to only one defined function (Search Employee task in the Fig. 6). The action tasks not linked to the defined function are linked to default function (Exit task in the Fig. 6).

As depicted in the Fig. 6, one operation task can be linked to more than one attribute of domain's object and one attribute of domain's object can be linked to more than one operation task.

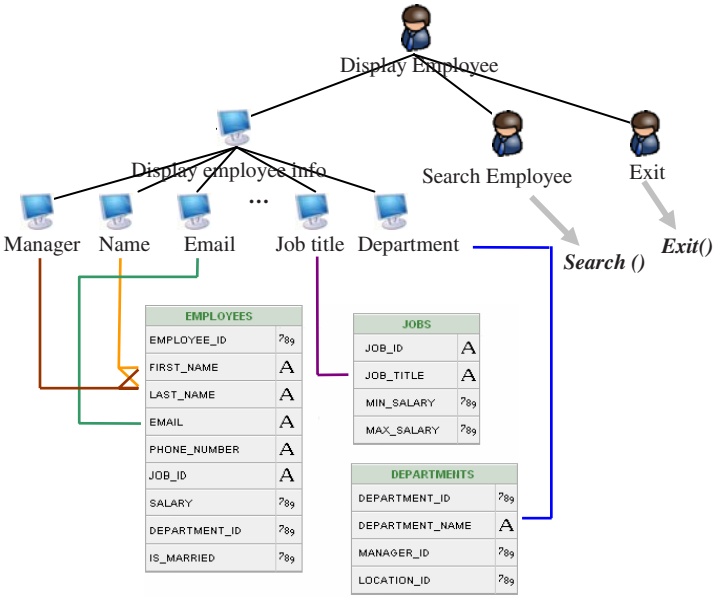

Fig. 6. Making the links between the tasks and the attributes of domain's objects, defined functions

4.2.4 UI Analyst Agent

There are two important UI objects are used in our process. These objects are Concrete Interaction Object and Final Interface Object.

A ***Concrete Interaction Object*** (CIO) is a graphical object for entering and displaying the data that the user can see, feel and manipulate [15]. A CIO is synonymous to a control, a physical interactor, a widget or a presentation object such as text-field, combo-box, check-box, button … A CIO in our process is defined by its label, control type, editable attributes as follows:

Concrete Interaction Object
Label: The label of the CIO; it will be used to label the control
Control type: The control type which is used to communicate between the user and computer's system
Editable: *Yes* if this control can be edited by end-user; otherwise *No*

The **Final Interface Object** (FIO) represents the operational interface object that is running on a special computing platform either by interpretation (e.g., through a web browser) or by execution. The FIO is determined based on the CIO in a certain language, on a certain platform and so on. A FIO defined as follows:

Final Interaction Object
Label: The label of the control
Control type: The control type is specified in certain platform
Editable: *Yes* if this control can be edited by end-user; otherwise *No*
Position: The position (X, Y) of control in a form or in the screen
Size: The dimension of the control, it contains width and height

Since the links are made by the **Developer**, the **UI Analyst** agent uses the *CreateCIO* and *TransformCIOstoFIOs* plans to analyze the linked elements to create the UI objects. These elements are the tasks, the attributes, the domain's objects and the defined functions.

The *CreateCIO* plan is used to create CIOs. The CIOs are determined based on the features of the attribute such as data type (Text/Number/Date), the primary key, ect and the relationships between the objects. For example: When one task derives from the attributes of domain's objects then the control type of the CIO created for this task is **Text field** or **Text box** if the data type is Text; **Number field** if the data type is Number; **Check box/Radio** if the data type is Boolean; Date picker if the data type is Date. Another example, if a task is linked to a defined function then the control type of CIO of this task is Button or Menu. The label of CIO is the name of the task.

Besides, based on the information in the user model, the system selects a correct control type among the possible control types. In other words, if there is more than one control type determined for one CIO then our software chooses one of them for this CIO based on the user's preference. For example: we have two ways to open a student's name dialog; one is to click on students menu then choose the student's name menu, one is click on a student's name button. If the user does not have computer experience it is better if the UI is a button. He/She cans understand exactly, quickly what he/she needs to act to open a dialog to see the student's name.

The *TransformCIOstoFIOs* plan is used to transform the CIOs to FIOs. The FIOs are specified based on the attributes of the CIOs and the programming language determined by the developer. For each CIO, a correlative concrete control is created. As discussed, a CIO is defined by the attributes *Name, control type, editable, position* and *size.*

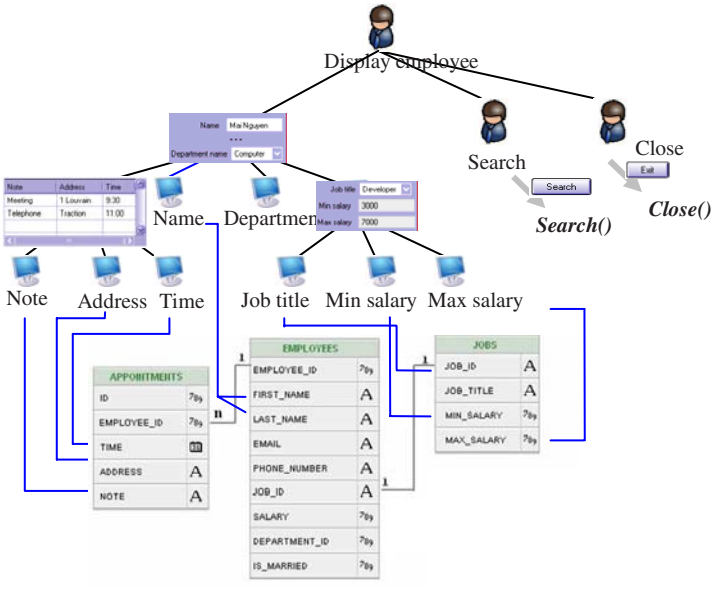

Fig. 7. An example for creating the FIOs

Fig. 7 depicts how the FOIs are created based on the links between the tasks and the attributes, the features of the attributes of domain model and the relationships between the domain's object.

4.2.5 Code Generator Agent

The **Code Generator** agent uses the *SpecifyLanguage* plan to specify the language used to perform the user interface determined above, the *GenerateUICode* plan to generate the code based on the determined FIOs and the *GenerateApplicationCode* to perform the function determined in the *FunctionAnalyst* agent.

4.3 User Interface Generation Workflow

Fig. 8 models the plan diagram depicting the control flow from the Function Analyst agent defining the basic functions to the Code Generator agent creating the user interface. We summarize below some specific points of the generation workflow depicting the relationships between plans and events.

Once the *DefineFunction* plan defines the functions, it sends the *loadModel* event to the *LoadTaskModel, LoadUserModel* and *LoadDomainModel* plans to load task, user and domain models. The *MakeLinkForOperationTask* plan makes links between the operation tasks and the attributes of domain's objects when it received the LinkOTask event. The *MakeLinkForActionTask* plan makes links between the action tasks and the defined functions when it received the *LinkATask* event. The CreateCIOs plan creates the CIOs based on links between the task and components of the domain model when the *CreateCIO* event is detected by this plan. These CIOs are transformed to FIOs by the *TransformCIOstoFIOs* plan. The *GenerateUICode* plan is triggered once it has handled the *GenUICode* events to generate the UI code which performs the user interface. The *GenerateApplicationCode* plan will generate the code to perform methods of our application when this plan receives *GenAppCode* event.

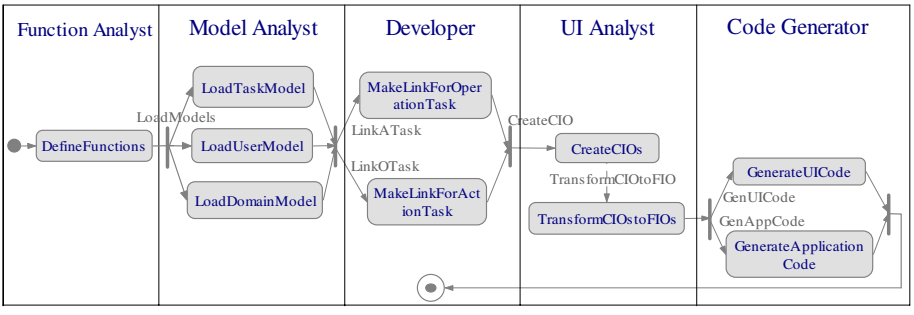

Fig. 8. User interface generation workflow

Since the tasks are associated with the attributes of the domain's objects and the defined functions, the user interface is automatically generated based on the features of the attributes and the relationships between the domain's objects.

5 Code Generator

The Code Generator is made to help developer to easily create the user interface from the task, user and domain models. Especially, Code Generator also automatically generates the application code for performing the defined functions which are discussed in the section 4.2.1. The task model is load from a XML file which is made by designer by using the CTTE [14]. The domain model is load from a concrete database which is determined by the developer. And finally, the user model is load from a text file.

As depicted in the Fig. 9, once the developer chooses a task from the list of tasks then the sub-tasks of this task is displayed in the left of screen. The sub-task are displayed are all of the sub-tasks at level 1 and all of the *operation* tasks at other levels. The domain's objects are displayed in the right of the screen. The UI objects are described in the table in the bottom of the screen.

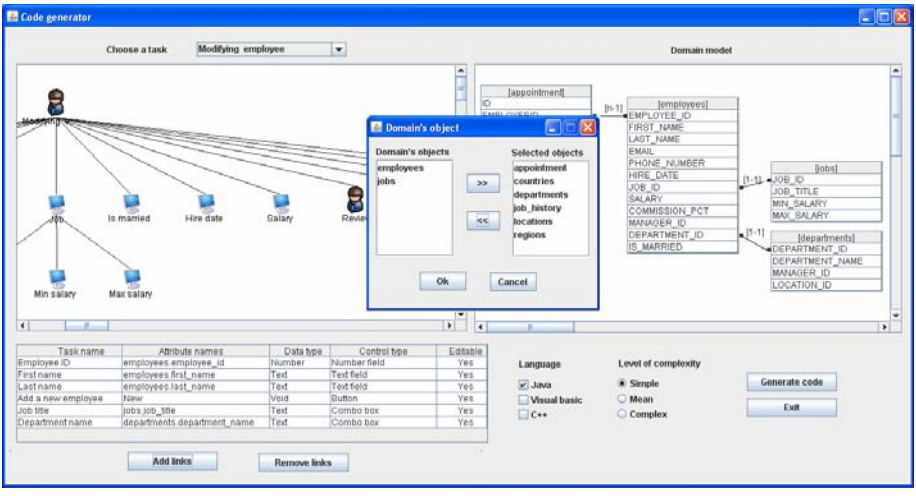

Fig. 9. Code generator

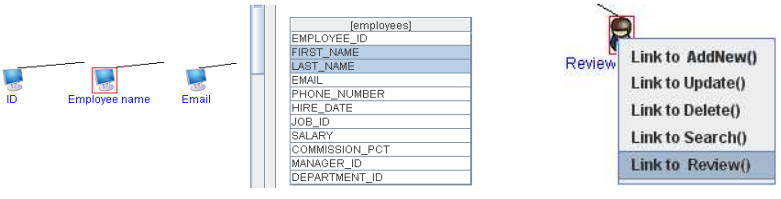

Fig. 10. Associating operation task *Employee name* to attributes First_Name, Last_Name and action task *Review Employees* to Review() function

The link between a leaf operation task and the attributes of the domain's objects is made by focusing the task and selecting the relevant attributes then putting on "Add link" button. The link between a leaf action task and a defined function is made by focusing the task and selecting the relevant function from a list of functions. Each action task is linked to only one function and each function is linked to only one task. (See Fig. 10).

The links are removed by selecting these links in the table then putting on "Remove links" button.

The generated code is generated when the developer puts on the "Generate code" button. The generated code can be compiled and ran immediately. The forms in the Fig. 11 are the result of the using the Code Generator. The selected language in this case is java. Beside generating the UI code, the code generator also generates the application code automatically for creating, deleting, reviewing the information of the employees.

The generated user interface is not useful as the user interface generated in a traditional manner are for the following reasons:

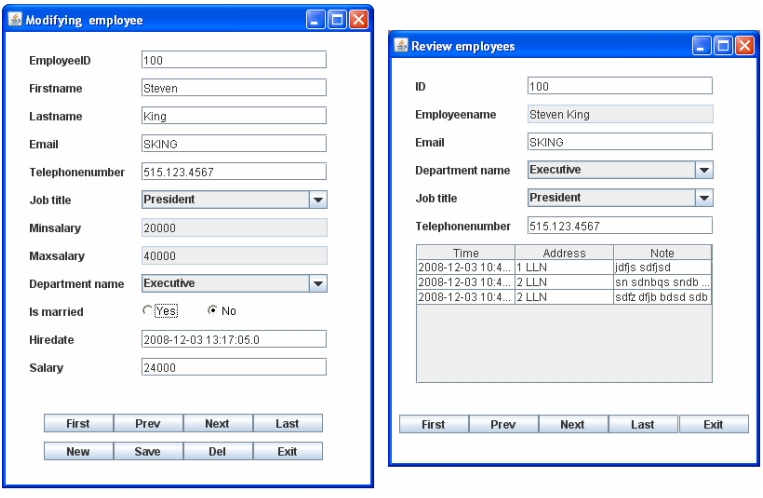

Fig. 11. Forms generated

- The controls are automatically designed in the order of creating the links between components of the task and domain models. They are not arranged reasonably.
- In the case of selecting alternative solutions, a traditional manner works better than an automatic manner since the developer can decide the best solution with his experience, knowledge, domain …

6 Conclusions

To be efficient, data-intensive systems that are an important component of today's software applications need effective human-computer interaction. User interfaces for such data systems has been a recurrent research issue and nowadays these UI have to support automatic generation to adequately be dealt with.

We have proposed here a framework whose purpose is to drive the automatic database user interface design and code behind generation from the task, user and domain model combined together.

This framework has aimed at offering a low cost, short time-to-implementation and efficient development environment from the business user side. Indeed, the objective is not to provide a tool for supporting the development of the database applications to not only the developers but also to support non-IT end-user. Because this tool is easy to use and the generated code cans run immediately.

References

1. Do, T.: A framework for multi-agent systems detailed desgn. Ph.D. Thesis, Université Catholique de Louvain, Institut d'Administration et de Gestion (IAG), Louvain La Neuve, Belgique (July 2005)

2. Julien, D., Ziane, M., Guessoum, Z.: GOLIATH: An extensible model-based environment to develop user interfaces. In: Proceedings of the Fourth International Conference on Computer Aided Design for User Interfaces (2004)
3. Da Silva, P.P., Griffiths, T., Paton, N.: Generating user interface code in a model based user interface development environment. In: Proc. of Advanced Visual Interfaces (AVI 2000), New York, pp. 155–160 (2000)
4. Mahfoudhi, A., Abed, M., Abid, M.: Towards a User Interface Generation Approach Based on Object Oriented Design and Task Model. In: TAMODIA 2005: 4th International Workshop on TAsk MOdels and DIAgrams for user interface design For Work and Beyond Gdansk, Poland, September 26-27 (2005)
5. Rosado da Cruz, A.M., Pascoal de Faria, J.: Automatic Generation of User Interfaces from Domain and Use Case Models. In: Quality of Information and Communications Technology, QUATIC 2007, pp. 208–212 (2007)
6. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Organizational abstractions for the analysis and design of multi-agent systems. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 235–251. Springer, Heidelberg (2001)
7. Pribeanu, C.: Tool Support for Handling Mapping Rules from Domain to Task Models. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 16–23. Springer, Heidelberg (2007)
8. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. The Knowledge Engineering Review 10(2), 115–152 (1995)
9. Wooldridge, M., Jennings, N.R.: Special Issue on Intelligent Agents and Multi-Agent Systems. Applied Artificial Intelligence Journal 9(4), 74–86 (1996)
10. Yu, E.: Modelling Strategic Relationships for Process Reengineering. Ph.D Thesis, Dept of Computer Science, University of Toronto, Canada (1995)
11. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vander-donckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers 15(3), 289–308 (2003)
12. Forbrig, P., Lämmel, R.: Programming with Patterns. In: Proceedings TOOLS-USA 2000. IEEE, Los Alamitos (2000)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
14. Paternò, F., Mori, G., Galiberti, R.: CTTE: an environment for analysis and development of task models of cooperative applications. In: CHI 2001 Extended Abstracts on Human Factors in Computer Systems, Seattle, pp. 21–22. ACM Press, New York (2001)
15. Tran, V., Vanderdonckt, J., Kolp, M., Faulkner, S.: Generating User Interface from Task, User and Domain Models. In: Proceedings of the Second International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, Centric 2009 (2009)
16. Vanderdonckt, J., Bodart, F.: Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In: Proc. of the ACM Conf. on Human Factors in Computing Systems INTERCHI 1993, Amsterdam, April 24-29, pp. 424–429. ACM Press, New York (1993)