Easy Development of Web Applications using WebODRA2 and a Dedicated IDE

Mariusz Trzaska
Chair of Software Engineering
Polish-Japanese Institute of Information Technology
Warsaw, Poland
mtrzaska@pjwstk.edu.pl

Abstract - **The modern Web requires new ways for creating applications. We present our approach combining a web framework with a modern object-oriented database and a dedicated Integrated Development Environment (IDE). It makes it easier to develop web applications by rising the level of abstraction. In contrast to many existing solutions, where the business logic is developed in an object-oriented programming language and data is stored and processed in a relational system, our proposal employs a single programming and query language. Such a solution, together with flexible routing rules, creates a coherent ecosystem and, as an additional benefit, reduces the impedance mismatch. Our research is supported by a working prototype of the IDE and a web framework for our own object-oriented database management system. Furthermore, the created IDE utilizes scaffolding, which can automatically generate web GUIs supporting some useful operations.**

Keywords-Web frameworks; Web tools; Web applications; Object-Oriented Databases; Integration Development Environment; IDE; DSL editors; Scaffolding

I. INTRODUCTION

Web frameworks are commonly utilized in software development. Moreover, it seems that for each popular programming language exists at least a few different proposals. The situation is different in case of prototype technologies. In [1], we have presented our proposal of a web framework dedicated to the object-oriented database ODRA (Object Database for Rapid Application development).

It seems that the most successful frameworks (e.g., Rails, ASP.NET MVC) follow the three-tier architecture: a presentation layer, business logic (a middle tier) and a data tier. Each of them can be developed through a different technology and can utilize incompatible data models.

Typically, the middle tier is developed using an object-oriented programming language such as Java, MS C#, Ruby, etc. However, the object-orientedness is a bit blurry concept. There is no single, well-accepted, specific definition or set of properties, which determine features of an object-oriented programming language. Java and C# are pretty close to each other in that area, but for instance Ruby is based on quite different concepts.

Contrary to implementation of the business logic, the data is usually stored using a relational database system. This causes a negative phenomenon known as impedance mismatch. Our framework has been created not only as an aid for making websites but also as an attempt to remove the fault. Of course, during the years, numerous approaches have been proposed to solve or reduce the problem. In particular, following Trzaska [2], the solution could use a single model both for the business logic and data.

Aside of frameworks, one of the most popular software, widely utilized by programmers, is an Integrated Development Environment (IDE). Various IDEs are on the scene for many years. They provide many different services and are invaluable help during software development. At the basic level they just support a programming language. However, their real power can be experienced when they have dedicated functionalities for particular frameworks. Similarly to the situations with the frameworks, prototype solutions are rarely equipped with an IDE.

In this paper, we would like to employ the idea for a tool aiming at creating web applications. We propose a paradigm, which uses the same high level language for working with data and implementing a business logic. In fact, those two utilizations are indistinguishable.

On the software level, our solution consists of two parts:

- An Integrated development Environment (called ODRA eIDE2) optimized for the ODRA DBMS, SBQL (Stack-Based Query Language) language and WebODRA2;
- A new version of the web framework (presented in [1]) called WebODRA2, which integrates two independent components:
    - The object-oriented DBMS ODRA with SBQL, a powerful programming and query language;
    - A web server.

This approach increases significantly the level of abstraction, which reduces the implementation time, decreases the number of errors and of course, completely eliminates the impedance mismatch. The programmers are able to focus on website's creation using a single, coherent technology.

The main contribution of the paper are the following:

- A new coherent paradigm of creating web application using the same high level programming and query language;
- A working prototype implementation of the approach containing a dedicated IDE, object-oriented database, web server and all the necessary components.

The rest of the paper is organized as follows. To fully understand our motivation and approach, some related solutions are presented in Section II. Section III briefly discusses key concepts of the utilized database and programming/query language. Section IV presents the prototype implementation of the proposed IDE and the web framework. Section V is devoted to the scaffolding mechanism. Section VI concludes.

II. Related Solutions

The related solutions appropriate for this paper could be analyzed from two points of view: web frameworks and IDEs. The next two sections contain their discussion.

A. *Existing Web Frameworks*

There are a lot of different web frameworks that use various approaches; just to name the most popular ones (by platform):

- Java: Apache Struts, Java Server Faces, JBoss Seam, Spring, Grails (Groovy), Play (Scala and Java);
- MS C#: ASP.Net, ASP.NET MVC, Kentico;
- PHP: CakePHP, Symfony, Zend;
- Smalltalk: Seaside [3];
- Ruby: Ruby on Rails, Sinatra.

They differ in details but unfortunately share the same problems related to inconsistent data models for programming languages and data. Even when an object-relational mapper is utilized the impedance mismatch problem is decreased, but not removed. For instance, the Ruby's Active Record requires additional information from a programmer to specify some non-mappable objects like arrays [4].

However, it is also possible to find solutions, where a website is developed using a single model. The next paragraphs contain description of such frameworks.

CouchApp [5] is a technology, which allows for creating applications delivered to the browser from CouchDB [6]. Applications are implemented using JavaScript and HTML5. The general idea is quite similar to our approach because CouchDB is a database management system. However, on contrary to our framework, the DBMS follows the NoSQL philosophy and allows one to store documents in the JSON [7] format. There is also no query language similar to SQL or our SBQL (see Section III). All database queries are performed using dedicated API and JavaScript. The result is also returned as a JSON data.

Every web application needs a GUI. In case of CouchApp a GUI is created as a transformation of returned JSON data into some other format. For instance there are functions, which together with dedicated views, are able to convert the data into HTML, XML, CVS, etc.

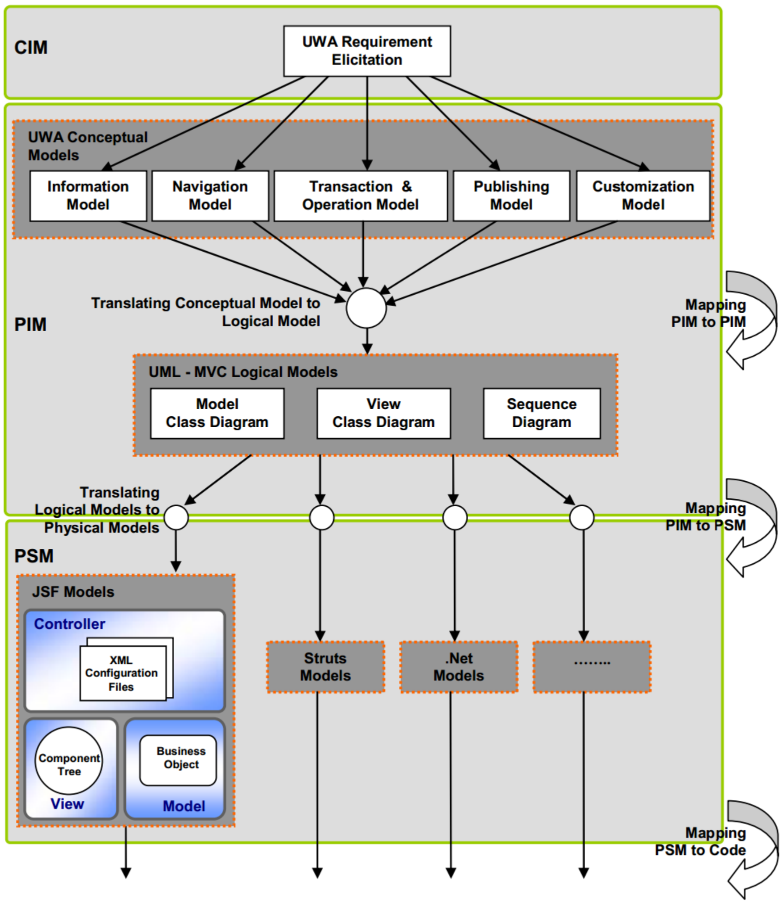
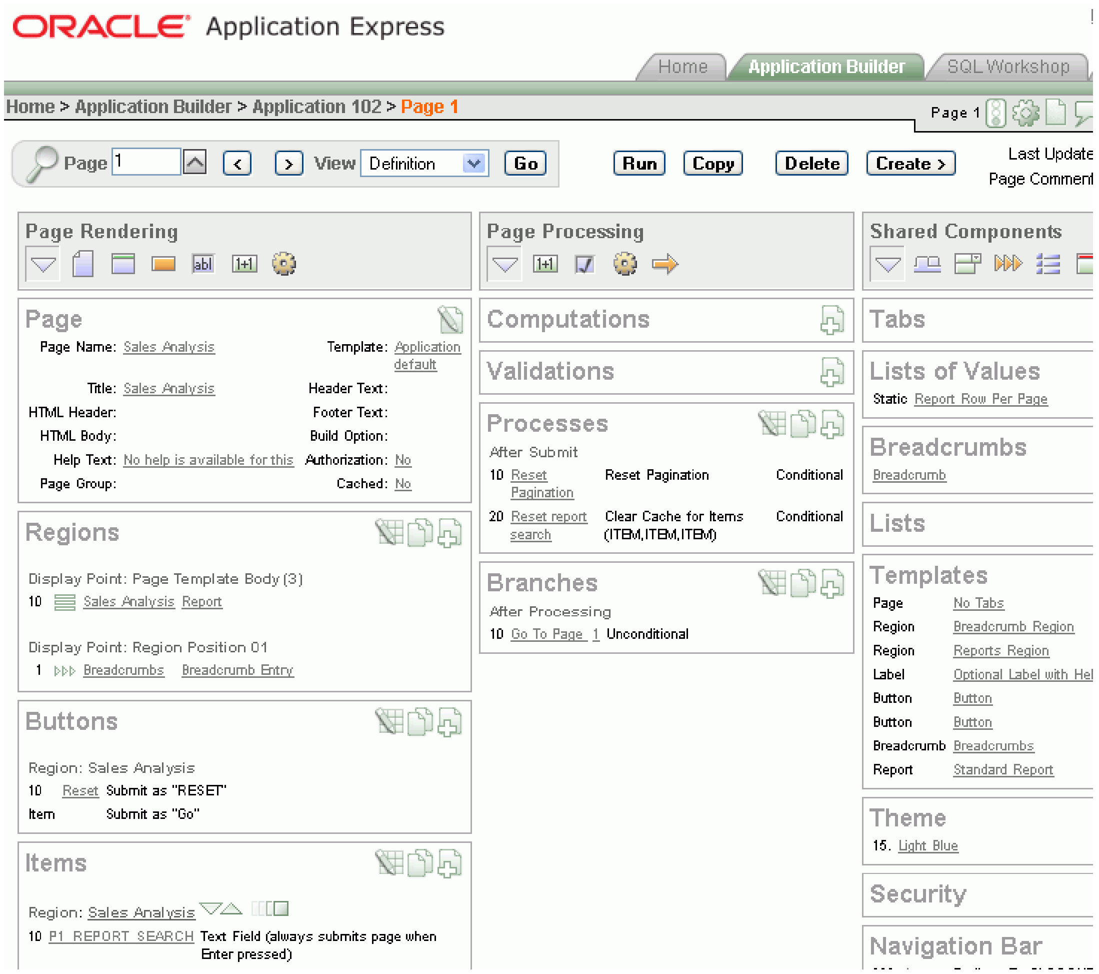


Figure 1. An overview of the UWA-based MDD process. Source: [8].



Page

Page Name: Sales Analysis | Template: Application default
Title: Sales Analysis | Header Text:
HTML Header: | Footer Text:
HTML Body: | Build Option:
Help Text: No help is available for this | Authorization: No
Page Group: | Cached: No

Regions

Display Point: Page Template Body (3)
10 Sales Analysis Report

Display Point: Region Position 01
1 Breadcrumbs Breadcrumb Entry

Buttons

Region: Sales Analysis
10 Reset Submit as "RESET"
Item Submit as "Go"

Items

Region: Sales Analysis
10 P1_REPORT_SEARCH Text Field (always submits page when Enter pressed)

Computations

Validations

Processes

After Submit
10 Reset Pagination Reset Pagination Conditional
20 Reset report search Clear Cache for Items (ITEM,ITEM,ITEM) Conditional

Branches

After Processing
10 Go To Page 1 Unconditional

Tabs

Lists of Values

Static Report Row Per Page

Breadcrumbs

Breadcrumb

Lists

Templates

Page	No Tabs
Region	Breadcrumb Region
Region	Reports Region
Label	Optional Label with Hel
Button	Button
Button	Button
Breadcrumb	Breadcrumbs
Report	Standard Report

Theme

15. Light Blue

Security

Navigation Bar

Figure 2. Oracle Application Express. Source: [9]

Another approach to create a web application might employ the Model Driven Architecture (MDA) paradigm. The idea is to define a model (or models) and, through some transformations, receive a working application. There is a lot of such systems, see for instance [10], [11], [8]. However, they are not widely utilized. One of the reason could be the amount and type of work, which has to be done to get a working website. For instance, the proposal presented in [8], which is quite common to all MDA solutions, needs the following models and information to be precisely defined (Figure 1):

- UWA requirements,
- Information model,
- Navigation model,
- Transaction & operation model,
- Publishing model,
- Customization model,
- Logical models (UML diagrams): class, sequence.

Of course, the above information is not only required by MDA tools. Furthermore, they have to be provided by all websites' developers. It seems that the way of defining them makes the difference in popularity.

The last described solution is not exactly a framework for programmers. Oracle Application Express (Figure 2) [12] is more like a tool for a rapid web application development for the Oracle database. It is available, under different names, since 2000. The application requires a dedicated server and provides easy-to-use programming environment accessible via a web browser.

Most of its functionalities are available through dedicated graphical user interfaces, various wizards and helpers. But, still there are possibilities for using a programming language, namely PL/SQL. SQL, despite of thirty-year existence and big popularity, is the subject of heavy criticism. The SQL's flaws like: inconsistencies, incompatibilities between vendors and shortcomings of the relational model, decrease the value of solutions. Furthermore, application generators have some inherent shortcomings, which make their products less flexible (in terms of usability, functionality, GUI) than applications developed by programmers. We believe that using a more powerful programming and query language, together with an object-oriented model, presents a much better approach.

B. The Most Popular IDEs

Below we present the most popular IDEs for particular programming languages/platforms:

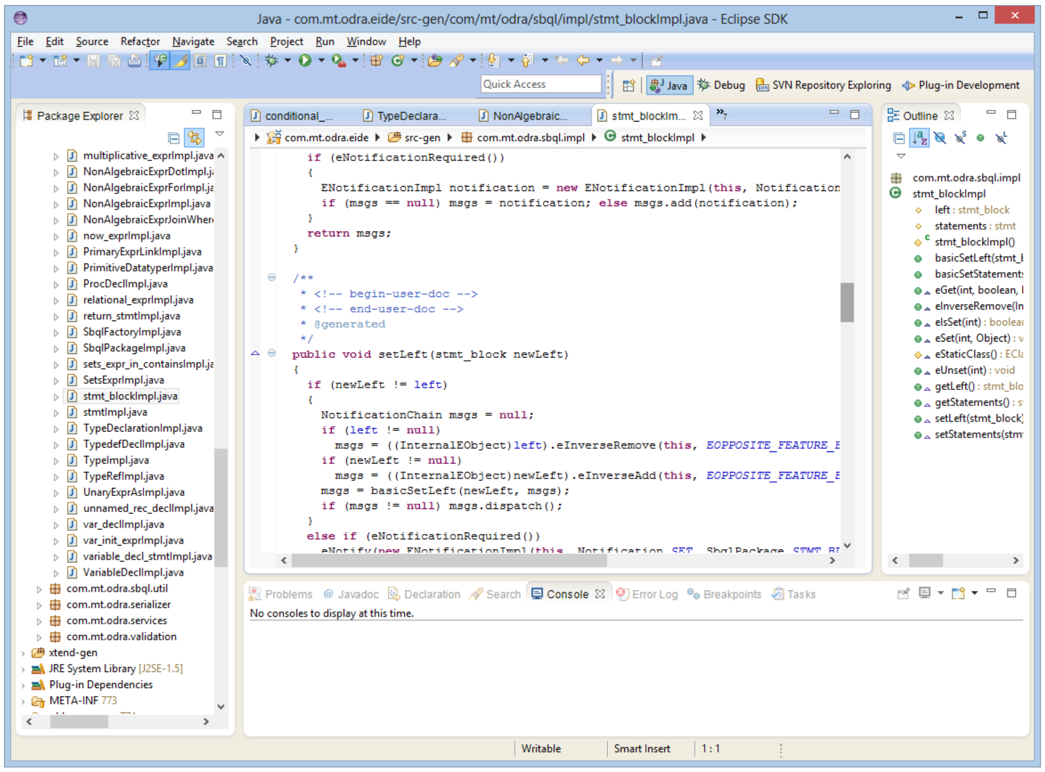
- Java: Eclipse [13], NetBeans [14], IntelliJIDEA [15];
- MS C#: MS Visual Studio [16], MonoDevelop [17], SharpDevelop [18];
- PHP: Aptana Studio [19], Eclipse PDT [20], PhpStorm [21], KDevelop [22];
- Smalltalk: Pharo [23], VisualWorks [24];
- Ruby: RubyMine [25], NetBeans [26], Aptana RadRails [27].

The IDEs differ in many ways. They can be distributed completely for free (e.g., Eclipse, NetBeans) or have various (usually limited) editions, e.g., Visual Studio: Professional Edition costs a few hundreds of USD whereas Express Edition is free.

Some of them are strictly dedicated to particular language/technology e.g., RubyMine. Others support various programming languages – sometimes using special plugins (e.g., Eclipse: Java, C++, and PHP).

Concerning functionality, it seems that the basic possibilities are quite similar in all IDEs and include:

- Syntax coloring (see Figure 3). Text of a program (source code) is presented using different colors, accents (e.g., bold) and decorators (e.g., an underline);
- Autocomplete. This is one of the most important features of a decent IDE and shows a list of possibilities for a particular context (e.g., after the dot the programmer sees attributes of a class);
- Error reporting. Errors and warnings are presented in a special window and sometimes inside the editor as well;
- Quick fix. When an error or a warning is shown to a user, he/she can choose one of proposed fixes to the problem (e.g., adding an import statement in case of an unknown class);
- Semantic navigation among artefacts. This functionality allows jumping from an occurrence of an element to its definition, e.g., from an object to its class;
- References. This makes possible finding all utilizations of particular artefacts (e.g., all method calls);
- Project Management. Takes care of all project's files including sources, assets, folders, etc. Usually it is possible to connect it to a versioning management system (e.g., Subversion, Git, etc.);
- Plugins. They provide a way for adding new functionalities (e.g., support for a new framework and/or programming language) to existing core. Sometimes (e.g., Eclipse) an entire IDE is based on plugins.
- Refactoring. This technology is responsible for making changes to a source code without modifying its behavior. There are dozens of different refactoring, e.g., renaming, extracting an interface or a method, adding getters/setters, etc.;



```
    if (eNotificationRequired())
    {
      ENotificationImpl notification = new ENotificationImpl(this, Notification
      if (msgs == null) msgs = notification; else msgs.add(notification);
    }
    return msgs;
  }

  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  public void setLeft(stmt_block newLeft)
  {
    if (newLeft != left)
    {
      NotificationChain msgs = null;
      if (left != null)
        msgs = ((InternalEObject)left).eInverseRemove(this, EOPPOSITE_FEATURE_E
      if (newLeft != null)
        msgs = ((InternalEObject)newLeft).eInverseAdd(this, EOPPOSITE_FEATURE_E
      msgs = basicSetLeft(newLeft, msgs);
      if (msgs != null) msgs.dispatch();
    }
    else if (eNotificationRequired())
```

```
No consoles to display at this time.
```

Figure 3. The Eclipse IDE. Source: own elaboration.

- Analyzing the code. In some cases, IDE is able to make suggestions regarding a source code, e.g., renaming an artefact according to a particular convention.
- Scaffolding. This is a mechanism, which helps programmers and/or designers in creating a starting point for further development. It could generate various artefacts, but in most cases is utilized for a GUI creation. It is especially popular in web frameworks where the mechanism is able to generate necessary files (HTML, CSS, etc.) required for CRUD (Create, Retrieve, Update, Delete) operations. Sometimes it is hard to distinguish if this tool is integrated with a framework or it is a separate component embedded in IDE.

In many cases, modern IDEs are equipped with GUI editors (embedded or installed through additional plugins). Their functionalities vary starting from simple generators to sophisticated bi-directional masterpieces.

Modern software projects are usually implemented using a wide range of technologies, platforms and frameworks. Thus support for them is essential. This is an area where commercial products show their strengths (e.g., IntelliJIDEA).

The choice of a particular IDE is not always based solely on its functionalities. In some projects also non-functional requirements could be important, e.g., cross-platform support. This is satisfied by most of the IDEs because many of them is developed using cross-platform technologies like Java (Eclipse, NetBeans, Aptana Studio).

It is also worth noting "smart" editors (Notepad++, Vim, Sublime Text, and TextMate), which could be very close to fully-fledged IDEs, especially after a correct configuration. They usually offer less sophisticated functionalities like syntax coloring or project's files management.

III. The ODRA Database

As previously mentioned, our proposal for creating websites is based on utilization of an object-oriented database together with a powerful query and programming language. DBMS could be used as a data storage and could be utilized to implement business logic. For the purpose of the first requirement we need a database query language. However, because of the second necessity, we might need something more flexible and powerful: a fully-fledged programming language with imperative constructs. Both criteria are met by our prototype DBMS called ODRA.

ODRA is a prototype object-oriented database management system [28], [29], [30], [31] based on SBA (Stack-Based Architecture) [16]. The ODRA project started to develop new paradigms of database application development. This goal is going to be reached mainly by increasing the level of abstraction, at which the programmer works. ODRA introduces a new universal declarative query and programming language SBQL [28], together with distributed, database-oriented and object-oriented execution environment. Such an approach provides functionality common to the variety of popular technologies (such as relational/object databases, several types of middleware, general purpose programming languages and their execution environments) in a single universal, easy to learn, interoperable and effective to use application programming environment.

ODRA consists of three closely integrated components:

- Object Database Management System (ODMS),
- Compiler and interpreter for object-oriented query programming language SBQL,
- Middleware with distributed communication facilities based on the distributed databases technologies.

The system is additionally equipped with a set of tools for integrating heterogeneous legacy data sources. The continuously extended toolset includes importers (filters) and/or wrappers to XML, RDF, relational data, web services, etc.

ODRA has all chances to achieve high availability and high scalability because it is a main memory database system with memory mapping files and makes no limitations concerning the number of servers working in parallel. In ODRA many advanced optimization methods that improve the overall performance without compromising universality and genericity of programming interfaces have been implemented.

The next subsections contain a short discussion of the ODRA main features including its query and programming language SBQL.

A. ODRA Object-Oriented Data Model

The ODRA data model is similar to the UML object model. Because in general UML is designed for modeling rather than for programming several changes have been made to the UML object model that do not undermine seamless transition from a UML class diagram to an ODRA database schema. The ODRA object model covers also the relational model as a particular case; this feature is essential for making wrappers to external sources stored in relational databases. Below, we present a short description of the main data model elements:

- Objects. The basic concept of the ODRA database model is object. It is an encapsulated data structure storing some consistent bulk of information that can be manipulated as a whole. A database designer and programmers can create database and programming objects according to their own needs and concepts. Objects can be organized as hierarchical data structures, with attributes, sub-attributes, etc.; the number of object hierarchy levels is unlimited. Any component of an object is considered an object too.
- Collections. Objects within a collection have the same name; the name is the only indicator that they belong to the same collection. Usually, objects from a collection have the same type, but this requirement is relaxed for some kinds of heterogeneous collections. Collections can be nested within objects with no limits (e.g., in this way it is possible to represent repeating attributes).
- Links. Objects can be connected by pointer links. Pointer links represent the notion that is known from UML as association. Pointer links support only binary associations; associations with higher arity and/or

with association classes are to be represented as objects and some set of binary associations. This is a minor limitation in comparison to UML class diagrams, introduced to simplify the programming interface. Pointer links can be organized into bidirectional pointers enabling navigation in both directions.

- Modules. In ODRA, the basic unit of database organization is a module. As in popular object-oriented languages, a module is a separate system component. An ODRA module groups a set of database objects and compiled programs and can be a base for reuse and separation of programmers' workspaces. From the technical point of view and of the assumed object relativism principle, modules can be perceived as special purpose complex objects that store data and metadata.
- Types, classes and schemata. A class is a programming abstraction that stores invariant properties of objects, in particular, its type, some behavior (methods, operations) and (optionally) an object name. A class has some number of member objects. During processing of a member object the programmer can use all properties stored within its class. The model introduces atomic types (integer, real, string, date, boolean) that are known from other programming languages. Further atomic types are considered. The programmer can also define his/her own complex types. Collection types are specified by cardinality numbers, for instance, [0..*], [1..*], [0..1], etc.
- Inheritance and polymorphism. As in the UML object model, classes inherit properties of their superclasses. Multiple inheritance is allowed, but name conflicts are not automatically resolved. The methods from a class hierarchy can be overridden. An abstract method can be instantiated differently in different specialized classes (due to late binding); this feature is known as polymorphism.
- Persistence and object-oriented principles. The model follows the orthogonal persistence principle, i.e., a member of any class can be persistent or volatile. Shared server objects are considered persistent, however, non-shared objects of a particular applications can be persistent too. The model follows the classical compositionality, substitutability and open-close principles assumed by majority of object-oriented programming languages.

Distinction between proper data and metadata (ontology) is not the property of the ODRA database model. The distinction can be important on the business model level, but from the point of view of ODRA both kinds of resources are treated uniformly.

B. Query and Programming Language SBQL

SBQL (Stack-Based Query Language) is a powerful query and programming language addressing the object model described above. SBQL is precise with respect to the specification of semantics. SBQL has also been carefully designed from the pragmatic (practical) point of view. The pragmatic quality of SBQL is achieved by orthogonality of introduced data/object constructors, orthogonality of all the language constructs, object relativism, orthogonal persistence, typing safety, introducing all the classical and some new programming abstractions (procedures, functions, modules, types, classes, methods, views, etc.) and following commonly accepted programming languages' and software engineering principles.

SBQL queries can be embedded within statements that can change the database or program state. We follow the state-of-the-art known from majority of programming languages. Typical imperative constructs are creating a new object, deleting an object, assigning new value to an object (updating) and inserting an object into another object. We also introduce typical control and loop statements such as if…then…else…, while loops, for and for each iterators, and others. Some peculiarities are implied by queries that may return collections; thus, there are possibilities to generalize imperative constructs according to this new feature.

SBQL in ODRA project introduces also procedures, functions and methods. All procedural abstractions of SBQL can be invoked from any procedural abstractions with no limitations and can be recursive. SBQL programming abstractions deal with parameters being any queries; thus, corresponding parameter passing methods are generalized to take collections into account.

SBQL is a strongly typed language. Each database and program entity has to be associated with a type. However, types do not constraint semi-structured nature of the data. In particular, types allow for optional elements (similar to null values known from relational systems, but with different semantics – e.g., see Listing 1) and collections with arbitrary cardinality constraints. Strong typing of SBQL is a prerequisite for developing powerful query optimization methods based on query rewriting and on indices.

Listing 1. Sample SBQL query (Get employees who have salary and earn the same as Brown). Source: [30]

```
(Emp with salary) where salary =
                ((Emp with salary where name =
                               ”Brown”).salary);
```

C. Virtual Updatable Views

Another interesting and quite unique ODRA property are updatable views. Classical SQL views do the mapping from stored data into virtual data. However, some applications may require updating of virtual data; hence, there is a need for a reverse mapping: updates of virtual data are to be mapped into updates of stored data. This leads to the well-known view updating problem: updates of virtual data can be accomplished by updating of stored data on many ways, but the system cannot decide, which of them is to be chosen. In typical solutions these updates are made by side effects of view invocations. Due to the view updating problem, many kinds of view updates are limited or forbidden.

In the ODRA project (basing on previous research) another point of view has been introduced. In general, the method is based on overloading generic updating operations (create, delete, update, insert, etc.) acting on virtual objects by invocation of procedures that are written by the view definer (Listing 2). The procedures are an inherent part of the view definition. They have full algorithmic power, thus there are no limitations concerning the mapping of view updates into updates of stored data. SBQL updatable views allow one to achieve full transparency of virtual objects: they cannot be distinguished from stored objects by any programming option. This feature is very important for distributed and heterogeneous databases.

Listing 2. An SBQL updatable view definition. Source: [30]

```
view RichEmpDef {
 virtual RichEmp : record
                   {name:string;
                    salary:integer;
                    worksIn: ref Dept;}[0..*];
    seed: record {e: ref Emp;}[0..*] {
       return (Emp where salary > threshold) as e;
    }

    on_retrieve { return e.(name as name,
                            salary as salary,
                            ref (Dept where name =
                               deptName) as worksIn);
    }

    on_update {
                e.name := value.name;
                e.deptName := value.worksIn.name;
                if(e.salary < value.salary) {
                  e.salary := value.salary;
                }
    }

    on_new newEmp {
        if(newEmp.salary > threshold)
         create permanent Emp(
               newEmp.name as name,
               newEmp.salary as salary,
               newEmp.worksIn.name as deptName);
    }

    threshold: integer;
}
```

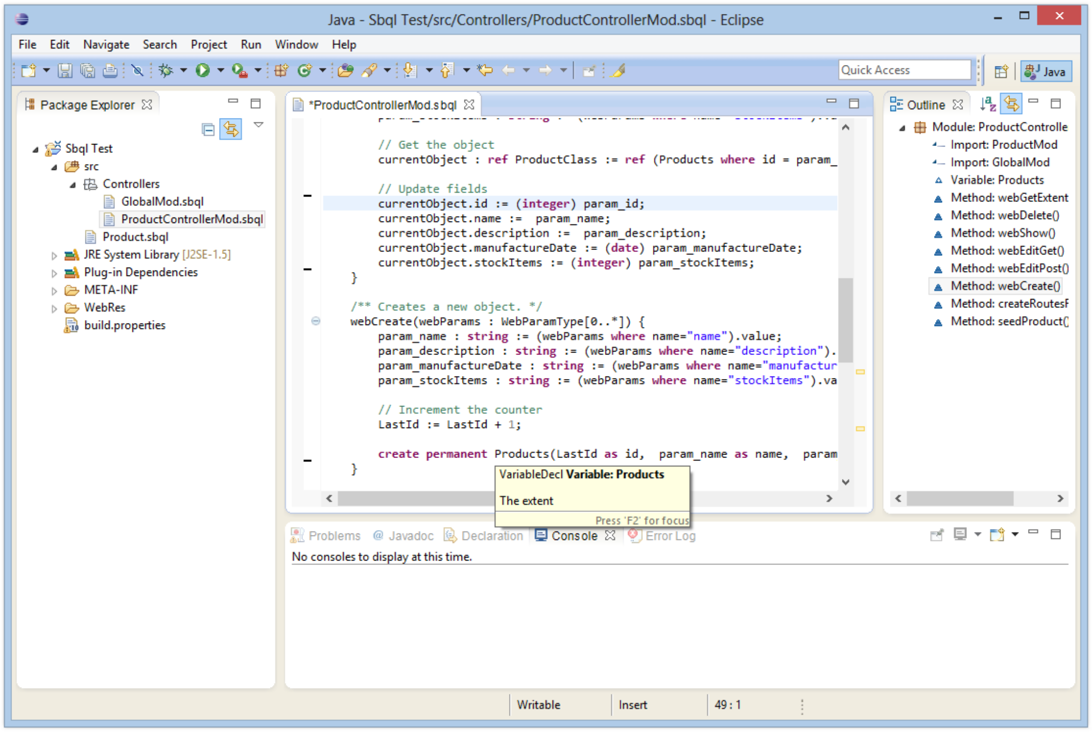
IV. Our Proposal

We believe that every developer should focus on the main task, e.g., creating a new application, a component or just a single function. In order to do this, his/her production environment should be as helpful as possible. This goal is fulfilled by modern IDEs. Of course, sometimes there are programmers who like to write their source code in a simple editor, but they are a minority. Most efficient IT professional use dedicated tools for their jobs. Thus, we have also decided to support programmers working with our WebODRA2 framework (and the ODRA DBMS in general) by introducing the ODRA eIDE2. The software together with tutorials is freely available; see [32].

A. ODRA eIDE2

There are different approaches to developing an IDE. In particular, there are the following possibilities:

- Start from scratch and manually create all necessary components;



```
        // Get the object
        currentObject : ref ProductClass := ref (Products where id = param_

        // Update fields
        currentObject.id := (integer) param_id;
        currentObject.name :=  param_name;
        currentObject.description :=  param_description;
        currentObject.manufactureDate := (date) param_manufactureDate;
        currentObject.stockItems := (integer) param_stockItems;
    }

    /** Creates a new object. */
    webCreate(webParams : WebParamType[0..*]) {
        param_name : string := (webParams where name="name").value;
        param_description : string := (webParams where name="description").
        param_manufactureDate : string := (webParams where name="manufactur
        param_stockItems : string := (webParams where name="stockItems").va

        // Increment the counter
        LastId := LastId + 1;

        create permanent Products(LastId as id,  param_name as name,  param
    }
```

No consoles to display at this time.

Figure 4. A window of ODRA eIDE2. Source: own elaboration.

- Extend an existing text editor. This approach was chosen by us a few years ago when we created the first ODRA IDE [33] based on the jEdit [34]. jEdit is a decent programmer's editor but does not provide the most advanced features of modern IDEs (see Section II.B). Thus, the IDE had syntax coloring, project management, error information, but lacked auto completion, refactoring, semantic navigation, etc.;
- Tailor an existing IDE to our needs. Depending on the IDE's architecture the process may require modification of the core code (which is not always publicly available) or create custom plugins.

Basically, modern IDEs are very complicated and sophisticated software. For instance, according to [35] the entire Juno edition of Eclipse (4.2.x) consists of 72 projects and contains 55 million lines of code. Obviously this is huge project and creating something similar requires a lot of resources.

After analyzing our expectations, which involved all previously described functionalities, we came to the conclusions that only the third approach is feasible for us. Therefore, we conducted some research to select the right IDE to extend.

Our choice was Eclipse together with Xtext framework [36]. The framework simplifies the process of creating our own DSL language together with a dedicated editor. From our point of view the latter feature was especially interesting.

1) *The SBQL Grammar*

In order to implement the functionalities like autocomplete or refactoring, the editor has to "understand" the source code. All particular artefacts of the supported language (e.g., class definitions or method parameters) have to be precisely defined and recognized in the source code (in the text). Usually, it could be achieved using a dedicated grammar describing the language. This is also the case of the framework, but there were some problems. The Xtext uses ANTLR parser, which utilizes LL(*) algorithm. Despite many advantages, the algorithm does not permit left recursion in grammars. On contrary, our existing ODRA compiler was based on Cup Parser Generator, which employs LALR(1) algorithm. Thus, we had to rewrite it taking into account the fundamental differences. The current version, implemented in the eIDE2, covers about 95% of the SBQL. Due to its size (about 400 lines of code and 80 production rules) we are not able to include it in the paper.

The grammar is just a starting point for more advanced functionalities. They have to be implemented by adding dedicated classes and/or methods. The more detailed description could be found in the next sections.

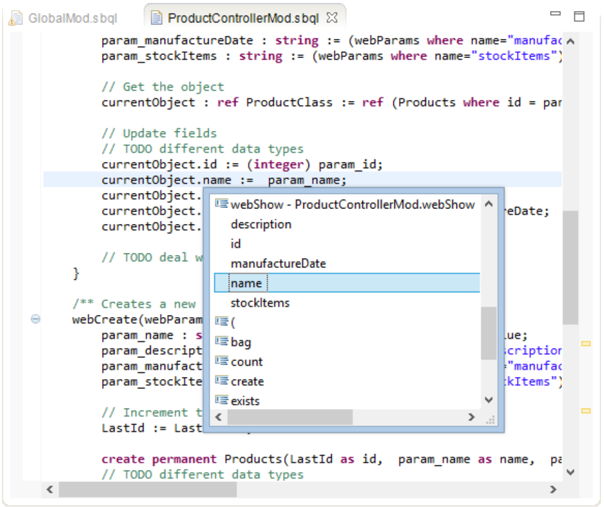
2) *Basic Editor Functionalities*

After defining the grammar, many features are available without any additional modifications (Figure 4). That includes things like: syntax coloring, text editing, basic navigation, tooltips (javadoc-like).

3) *Content Assist*

The content assist (autocomplete) is one of the most useful features provided by IDE (Figure 5). The Xtext framework contains a default implementation for this functionality. It could be modified by overriding some methods and/or adding new classes. We had to apply such modifications in some cases:

- The `import` statement. In the current version of the SBQL, the import statement adds entire module rather than a class (like in Java);
- Variable's declaration;
- The `where` expression should link to the content of the "inner" expression;
- The dot expression should be distinguished from the above mentioned `where`;



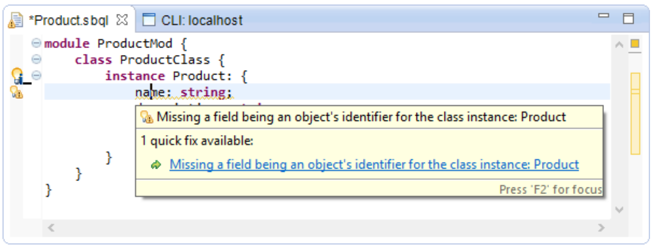
```
        param_manufactureDate : string := (webParams where name="manufac
        param_stockItems : string := (webParams where name="stockItems")

        // Get the object
        currentObject : ref ProductClass := ref (Products where id = par

        // Update fields
        // TODO different data types
        currentObject.id := (integer) param_id;
        currentObject.name :=  param_name;
        currentObject.
        currentObject.                                    eDate;
        currentObject.

        // TODO deal w
    }

    /** Creates a new
    webCreate(webParam
        param_name : s                                    ue;
        param_descript                                    cription
        param_manufact                                    "manufac
        param_stockIte                                    kItems")

        // Increment t
        LastId := Last

        create permanent Products(LastId as id,  param_name as name,  pa
        // TODO different data types
```

```
webShow - ProductControllerMod.webShow
description
id
manufactureDate
name
stockItems
(
bag
count
create
exists
```

Figure 5. Autocomplete feature in eIDE2. Source: own elaboration.

4) *Quick fixes*

When IDE detects a problem related to edited code, it is marked as a warning. Some of the warnings could be automatically fixed. Currently, we have implemented a few quick fixes, e.g., for missing fields (Figure 6).



```
module ProductMod {
    class ProductClass {
        instance Product: {
            name: string;

        }
    }
}
```

```
Missing a field being an object's identifier for the class instance: Product
1 quick fix available:
Missing a field being an object's identifier for the class instance: Product
Press 'F2' for focus
```

Figure 6. A quick fix in eIDE2. Source: own elaboration.

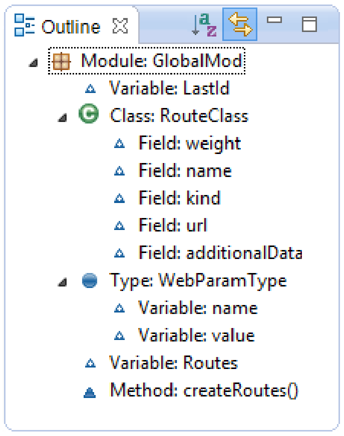
5) *Outline Window*

The outline window (Figure 7) shows a content's summary of the entire file. We modified the default implementation by adding custom graphical artefacts and altered some behaviors, e.g., presenting fields in the `Record` type.

6) *Refactoring*

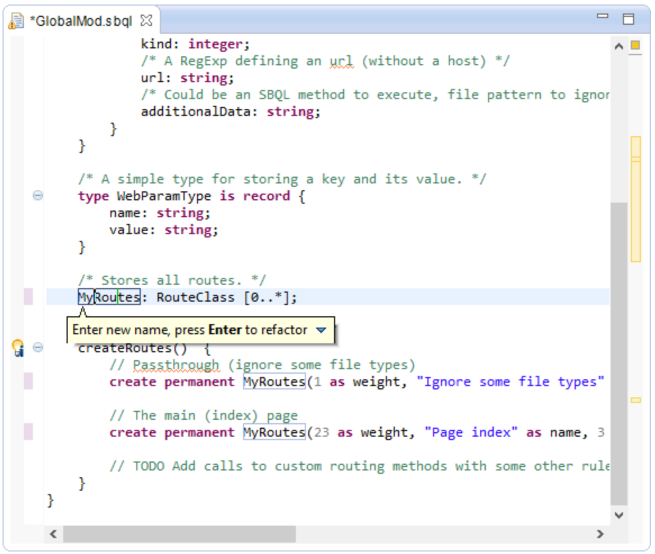
The refactoring is probably the second most widely used feature (after the content assist). Currently, we have only the default implementation, which allows for renaming (Figure 8;

please note the active borders around the renamed element: `MyRoutes`).



```
Module: GlobalMod
    Variable: LastId
    Class: RouteClass
        Field: weight
        Field: name
        Field: kind
        Field: url
        Field: additionalData
    Type: WebParamType
        Variable: name
        Variable: value
    Variable: Routes
    Method: createRoutes()
```

Figure 7. The outline window showing a semantic summary of the edited source file. Source: own elaboration.



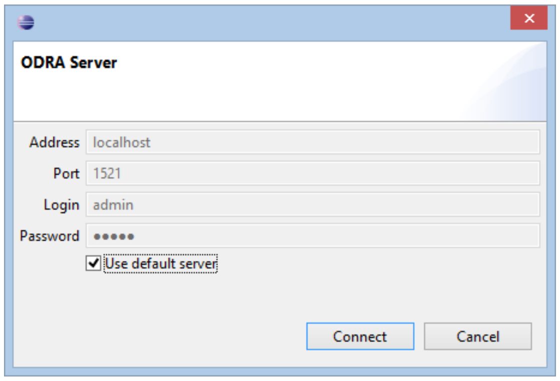
```
                kind: integer;
                /* A RegExp defining an url (without a host) */
                url: string;
                /* Could be an SBQL method to execute, file pattern to ignor
                additionalData: string;
            }
        }

        /* A simple type for storing a key and its value. */
        type WebParamType is record {
            name: string;
            value: string;
        }

        /* Stores all routes. */
        MyRoutes: RouteClass [0..*];
        Enter new name, press Enter to refactor
        createRoutes()  {
            // Passthrough (ignore some file types)
            create permanent MyRoutes(1 as weight, "Ignore some file types"

            // The main (index) page
            create permanent MyRoutes(23 as weight, "Page index" as name, 3

            // TODO Add calls to custom routing methods with some other rule
        }
    }
```

Figure 8. Refactoring in eIDE2. Source: own elaboration.

7) ODRA DBMS Integration

The ultimate goal of creating eIDE2 was facilitation of the ODRA development process both for its web framework and pure DBMS. Thus, we had to take care of easy integration. To do this, we introduced the following functionalities:

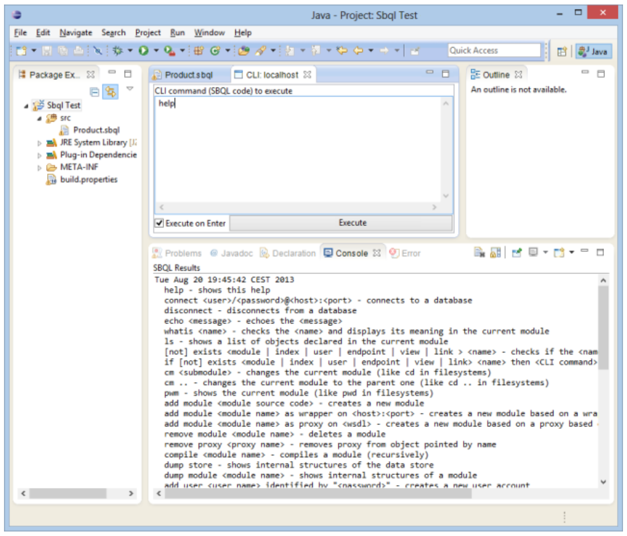
- Connection dialog (Figure 9). Every project managed by IDE could have its own, connected ODRA instance. It is possible to use just default values or provide custom ones.
- A programmer can also send text commands using a dedicated dialog window via CLI (Figure 10). This way of working with the DBMS allows for perform many advanced actions, currently not available directly from GUI.


ODRA Server

Address	localhost
Port	1521
Login	admin
Password	●●●●●

[✓] Use default server

Figure 9. ODRA DBMS connection dialog (managed by eIDE2). Source: own elaboration.



CLI command (SBQL code) to execute

```
help
```

SBQL Results

```
Tue Aug 20 19:45:42 CEST 2013
  help - shows this help
  connect <user>/<password>@<host>:<port> - connects to a database
  disconnect - disconnects from a database
  echo <message> - echoes the <message>
  whatis <name> - checks the <name> and displays its meaning in the current module
  ls - shows a list of objects declared in the current module
  [not] exists <module | index | user | endpoint | view | link > <name> - checks if the <nam
  if [not] exists <module | index | user | endpoint | view | link> <name> then <CLI command>
  cm <submodule> - changes the current module (like cd in filesystems)
  cm .. - changes the current module to the parent one (like cd .. in filesystems)
  pwm - shows the current module (like pwd in filesystems)
  add module <module source code> - creates a new module
  add module <module name> as wrapper on <host>:<port> - creates a new module based on a wra
  add module <module name> as proxy on <wsdl> - creates a new module based on a proxy based
  remove module <module name> - deletes a module
  remove proxy <proxy name> - removes proxy from object pointed by name
  compile <module name> - compiles a module (recursively)
  dump store - shows internal structures of the data store
  dump module <module name> - shows internal structures of a module
  add user <user name> identified by "<password>" - creates a new user account
```

Figure 10. An access to ODRA DBMS CLI (Command Line Interface). Source: own elaboration.

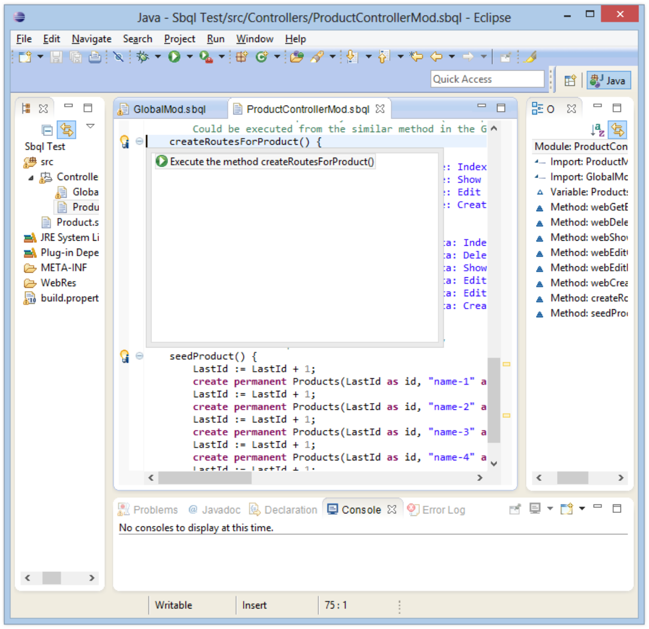
- Executing a SBQL method. When the IDE detects a parameterless method a small icon is placed near the code. A user is able to execute the method directly from IDE on the connected ODRA DBMS (Figure 11).

8) The Scaffolding

In case of eIDE2, the scaffolding supports a programmer in creating a web GUI for WebODRA2. The generated GUI together with SBQL controllers add CRUD operations (Create, Retrieve, Update, Delete) using a client-side library called Knockout.js [37]. The functionality is easy-to-use yet powerful. For more information see Section V.

9) Other goodies

It is also worth noting that thanks to the Eclipse ecosystem it is possible to utilize many other 3-rd party plugins. They can provide additional possibilities including web editors, versioning systems (Subversion, Git), spellcheckers, UML tools, GUI editors.



```
    Could be executed from the similar method in the G
createRoutesForProduct() {
```

Execute the method createRoutesForProduct()

```
seedProduct() {
    LastId := LastId + 1;
    create permanent Products(LastId as id, "name-1" a
    LastId := LastId + 1;
    create permanent Products(LastId as id, "name-2" a
    LastId := LastId + 1;
    create permanent Products(LastId as id, "name-3" a
    LastId := LastId + 1;
    create permanent Products(LastId as id, "name-4" a
```

No consoles to display at this time.

Java - Sbql Test/src/Controllers/ProductControllerMod.sbql - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
GlobalMod.sbql
ProductControllerMod.sbql
Problems
Javadoc
Declaration
Console
Error Log
Writable
Insert
75 : 1

Figure 11. Executing an SBQL method in eIDE2. Source: own elaboration.

B. *WebODRA2*

Basically, every web application, no matter how it is developed, requires the following set of logical components:

- A graphical user interface,
- A routing system,
- A business logic,
- Data to work with.

The above components could be implemented using various approaches. In some cases, a programmer has to manually define them whereas other solutions use generators to create some of them automatically. Additionally, real world websites also require some static files: html templates, css, jpeg, etc.

We have decided to use pure programmatic approach, which means that all necessary definitions are provided by a programmer. It may look like a lot of work, but thanks to the high level of abstraction, the amount of information is significantly reduced.

Another feature, which simplifies development is the MVC (Model – View - Controller) architecture utilized in many previously mentioned frameworks. Comparing to the other frameworks, our approach uses the same object-oriented model both for a business logic (Controller) and data (Model). This method not only removes the impedance mismatch but also allows for using a powerful query and programming language for developing a business logic (behavior of the application). Furthermore, it is known that query languages operate on higher level of abstraction, effectively reducing the amount of code that needs to be written to achieve the same goals. For instance, a few tenths lines of Java code could be equivalent to a literally few lines of SBQL (or SQL). Not to mention performance and various optimizations, which are much more advanced in query languages.

Another very important area of a web framework is a graphical user interface. There are different methods to deal with the topic, some of them follow the MVC pattern. One of the most popular is using a server-side templating engine. A template contains an HTML code mixed with special tags, usually provided by the framework. In most cases, the tags allow to embed parts of a programming language (e.g., Java), mainly to insert some data (e.g., a list of products or customers). However, some programmers use them to implement additional functionality, which duplicates the controller’s responsibility. Of course, it is an incorrect application of the tags affecting maintainability of the code. At the end, tags are processed by an engine, a final HTML page is generated and sent to a web browser.

Comparing to the first release (see [1]), the second edition (WebODRA2) contains some bug fixes and modifications related to configuration and a project’s structure. They were mainly caused by the support in the eIDE2.

Figure 12 contains a simplified logical architecture of our prototype framework for developing web application called WebODRA2. The framework consist of two principal parts:

- A web server. It is responsible for responding to incoming requests from a web browser. The implementation of the server is based on open source tool called Jetty [38];
- ODRA Database Management System. This is a standard instance of the ODRA server introduced in Section 3.

The following subsections describe each of the components (from Figure 12) in details.

1) *Routing Module*

The center of WebODRA2 consists of a routing module, which is responsible for a correct processing of incoming web requests. The module is driven by rules defined by a programmer. Each definition, written in SBQL (as an object with specific properties), contains the following information:

- Url. A regular expression, which will be applied to the incoming request’s url. If there is a match, then the rule will be executed;
- Weight. It affects an order of the processing;
- Name. Human-readable name of the rule. It is especially useful during logging;
- Additional Data. The utilization of the additional data depends on a rule kind;
- Rule Kind. It affects the following processing:
    - Passthrough. The web framework ignores those rules and they are processed by the Jetty server. They serve static files like: pictures, css, Java script, etc.;
    - Data route. They contain an SBQL method’s name to execute. The method will get all HTML form parameters entered by a user, which makes it possible to process them by an SBQL code. The result of the method is transformed (see further) and returned to the browser;

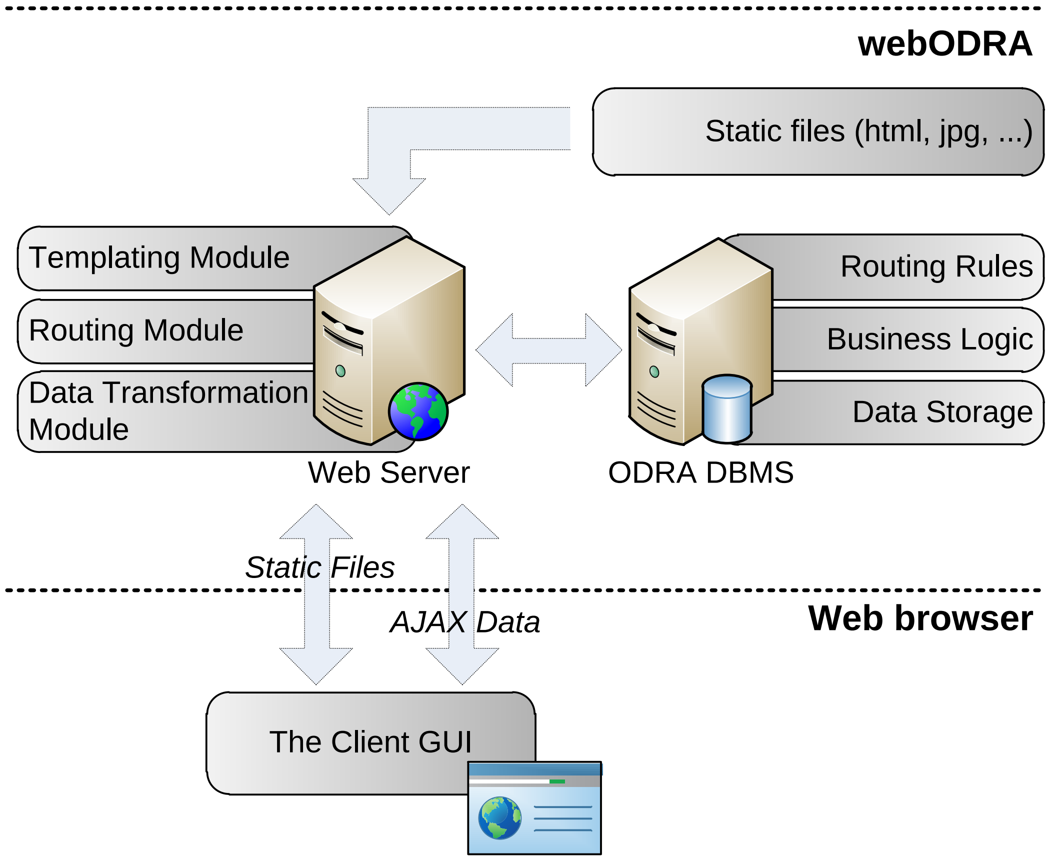


Figure 12. Logical architecture of WebODRA2. Source: own elaboration.

- Page route. An HTML page post-processed by our simple templating engine (see further).

2) *The Client GUI*

As previously mentioned, typical server-side web templating engines may lead to overuse tags by implementing some business functionality. To prevent this we have decided to use a client-side GUI framework. The idea is based on embedding in a web page some (meta) information, which will be used to present business data. We have chosen a framework called Knockout [37] utilizing new HTML5 *data-*attributes. They allow to create custom attributes and store any information. The process of showing a web page contains two steps. First, an HTML page is downloaded from a server, containing the markers. Then, the library sends an AJAX request to asynchronously retrieve necessary data, which are “injected” into the page.

The user data submission is performed on a similar rules. An asynchronous request is send to the server, triggering a Data Rule processing the provided data.

Standard website navigation is performed using regular hyperlinks (“outside” the framework).

We do not provide any dedicated GUI controls as a part of the framework. However, some of them are generated during the scaffolding process (see Section V). A programmer is also free to use any available solutions as long as they could be tuned to work with JSON format (e.g., some JavaScript libraries/frameworks).

3) *Templating Module*

The templating module is responsible for a coherent look and fill of the entire website. It operates on a single master page, which has a dynamic area fulfilled with some functional pages, i.e., a document repository, a forum, news, etc. For instance, the master page can contain a header, a navigation panel and a footer.

The process is triggered by the Page Route rule. When a particular page is requested by a browser, the master page is applied, or to be more precise, the requested page is embedded in the master page and then returned to the browser.

4) *Data Transformation Module*

When a Data Route rule executes a given SBQL method, the result could be any SBQL data type, e.g., a collection, a single object or a text. It needs to be processed to the format recognized by the Client GUI. The Data Transformation Module recursively converts the result into JSON [7] string, sends it back to the web browser where it is further processed.

5) *Routing Rules, Business Logic and Data Storage*

The above components are described in other Sections:

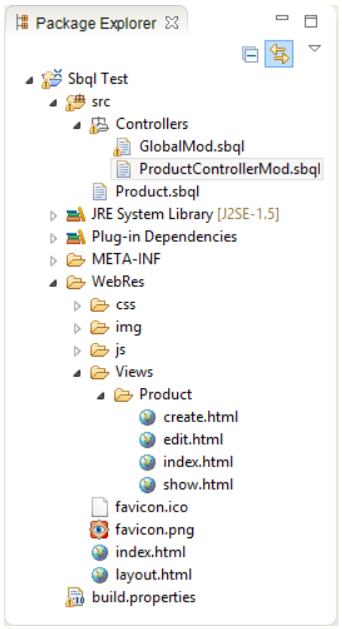
- The Routing Rules in Section IV.B.1;
- The Business Logic stores appropriate SBQL code (see Section III.B) referenced from *Data routes*;

- The Data Storage uses ODRA DBMS (see Section III).

V. SCAFFOLDING FOR THE WEBODRA2

As previously mentioned, scaffolding is a major programmer's facilitation in creating real-world web applications. It generates (one-way) necessary artefacts (e.g., source code, media and HTML files) being a starting point for further development.

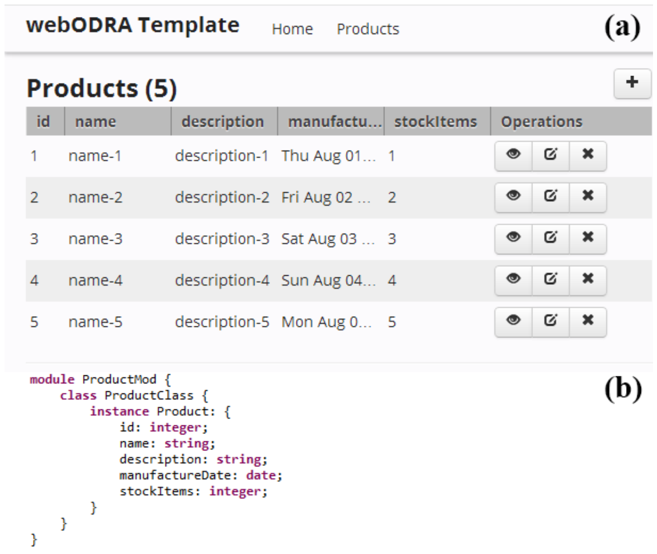
In case of eIDE2, for each SBQL class a dedicated button is placed near its source definition. Then, it is possible to start the process and all necessary files and folders will be generated (Figure 13):



```
Package Explorer
◢ Sbql Test
  ◢ src
    ◢ Controllers
        GlobalMod.sbql
        ProductControllerMod.sbql
      Product.sbql
  ▷ JRE System Library [J2SE-1.5]
  ▷ Plug-in Dependencies
  ▷ META-INF
  ◢ WebRes
    ▷ css
    ▷ img
    ▷ js
    ◢ Views
      ◢ Product
          create.html
          edit.html
          index.html
          show.html
      favicon.ico
      favicon.png
      index.html
      layout.html
    build.properties
```

Figure 13. Files scaffolded by eIDE2. Source: own elaboration.

- *GlobalMod.sbql*. This file stores some global data, common for the entire project:
  - Two definitions: `RouteClass` (each instance stores a single routing rule) and `WebParamType` (utilized for passing parameters from the web);
  - `Routes` acting as an extent for all the rules;
  - A `createRoutes()` method, which puts global routing rules into the DB;
- *ProductControllerMod.sbql*. This is a controller for the business class `Product`. It is responsible for:
  - Storing all instances of `Product`;
  - Processing web requests (all methods: `webXXX`. The methods are referenced by *Data routing* rules (inside the `createRoutesForProduct()` method));
  - Creating routes for CRUD operations for the Product class (the `createRoutesForProduct()` method);
  - Generating sample data (the `seedProduct()` method);
- *WebRes* folder, which is a root folder for the web server utilized by the framework. Aside of self-explaining typical folders (*css*, *img*, *js*) there are some others worth a short discussion:
  - The *Views* folder stores subdirectories for each business class. The subdirectories contain dedicated *html* files for processing particular CRUD operations. The files are referenced by *Page routing* rules (inside the `createRoutesForProduct()` method);
  - *index.html* file is a starting point for a web navigation;
  - *layout.html* file is a master template file providing a coherent look and feel for the entire site;



id	name	description	manufactu...	stockItems	Operations
1	name-1	description-1	Thu Aug 01...	1	
2	name-2	description-2	Fri Aug 02 ...	2	
3	name-3	description-3	Sat Aug 03 ...	3	
4	name-4	description-4	Sun Aug 04...	4	
5	name-5	description-5	Mon Aug 0...	5	

```
module ProductMod {
    class ProductClass {
        instance Product: {
            id: integer;
            name: string;
            description: string;
            manufactureDate: date;
            stockItems: integer;
        }
    }
}
```

Figure 14. One of the views (index) generated by the scaffolding (a) for the sample Product class (b). Source: own elaboration.

To sum up, a programmer using just one click can create a simple prototype of a working web application (Figure 14). The generated text files contains the code (both SBQL and HTML), which could be easily edited using the IDE or any external tools.

Currently, WebODRA2 is shipped with two examples: the forum (described in [1]) and the scaffolded one described above.

VI. CONCLUSION AND FUTURE WORK

We have presented our approach to creating web applications using a single, coherent model utilized both for data and business logic. Thanks to the powerful query and programming language SBQL, a programmer stays on the same high level of abstraction, saving time and making less errors.

Our approach is supported by two tools working together: a web framework called WebODRA2 and a dedicated IDE (eIDE2). Furthermore, we have added a scaffolding mechanism generating a web GUI with CRUD operations for any business class. The software is freely available online together with some tutorials [32].

The contribution of this paper is based on quite new method for creating websites. To our best effort, we were not able to find a similar solution, directly employing the power of a modern database to develop web portals.

We believe that this kind of solutions could be a valuable alternative to existing tools for creating data intensive web applications. Thus, we would like to continue our research in that field, improving our framework and the eIDE2 to make them production-ready.

References

[1] M. Trzaska, "WebODRA - A Web Framework for the Object-Oriented DBMS ODRA," in *The First International Conference on Building and Exploring Web Based Environments (WEB 2013)*, Seville, Spain, IARIA XPS Press, 2013, pp. 1-6.

[2] M. Trzaska, "Data Migration and Validation Using the Smart Persistence Layer 2.0," in *The 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)*, Las Vegas, USA, Acta Press, November 12 – 14, 2012, pp. 186-193.

[3] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt and R. Hirschfeld, An Introduction to Seaside, Hasso-Plattner-Institut, 2008.

[4] "ActiveRecord," [Online]. Available: http://api.rubyonrails.org/classes/ActiveRecord/Base.html [Accessed 05.09.2013].

[5] "CouchApp," [Online]. Available: http://couchapp.org/page/index. [Accessed 21.08.2012].

[6] C. Anderson, J. Lehnardt and N. Slater, CouchDB: The Definitive Guide, O'Reilly Media, 2010.

[7] "JSON: JavaScript Object Notation," [Online]. Available: http://www.json.org/. [Accessed 01.09.2013].

[8] D. Distante, P. Pedone, G. Rossi and G. Canfora, "Model-Driven Development of Web Ap-plications with UWA, MVC and JavaServer Faces," in *Web Engineering Lecture Notes in Computer Science*, Springer, 2007, pp. 457-472.

[9] Oracle, "Oracle Application Developer's Guide," [Online]. Available: http://docs.oracle.com/. [Accessed 05.09.2013].

[10] D. Arraes Nunes and D. Schwabe, "Rapid Prototyping of Web Applications combining Do-main Specific Languages and Model Driven Design," in *Proceedings of the 6th International Conference on Web Engineering (ICWE'06)*, Palo Alto, California, USA, July 11-14, 2006.

[11] S. Ceri, P. Fraternali and M. Matera, "Conceptual Modeling of Data-Intensive Web Applications," *IEEE Internet Computing vol. 6, no. 4,* July/August 2002.

[12] J. Williamson, Oracle Application Express: Fast Track to Modern Web Applications, McGraw-Hill Osborne Media, 2012.

[13] "Eclipse," [Online]. Available: http://www.eclipse.org/. [Accessed 6.9.2013].

[14] "NetBeans IDE," [Online]. Available: https://netbeans.org/. [Accessed 06.09.2013].

[15] "IntelliJ IDEA," [Online]. Available: http://www.jetbrains.com/idea/. [Accessed 04.09.2013].

[16] "Visual Studio," [Online]. Available: http://www.microsoft.com/visualstudio/plk. [Accessed 03.09.2013].

[17] "MonoDevelop," [Online]. Available: http://monodevelop.com/. [Accessed 05.09.2013].

[18] "SharpDevelop," [Online]. Available: http://www.icsharpcode.net/opensource/sd/. [Accessed 01.09.2013].

[19] "Aptana Studio," [Online]. Available: http://www.aptana.com/. [Accessed 02.09.2013].

[20] "Eclipse PHP Development Tools," [Online]. Available: http://projects.eclipse.org/projects/tools.pdt. [Accessed 04.09.2013].

[21] "PhpStorm," [Online]. Available: http://www.jetbrains.com/phpstorm/. [Accessed 02.09.2013].

[22] "KDevelop," [Online]. Available: http://www.kdevelop.org/. [Accessed 02.09.2013].

[23] "Pharo," [Online]. Available: http://www.pharo-project.org/home. [Accessed 02.09.2013].

[24] "VisualWorks," [Online]. Available: http://www.cincomsmalltalk.com/main/products/visualworks/. [Accessed 04.09.2013].

[25] "RubyMine," [Online]. Available: http://www.jetbrains.com/ruby/. [Accessed 03.09.2013].

[26] "Ruby and Rails - plugin for NB," [Online]. Available: http://plugins.netbeans.org/plugin/38549. [Accessed 04.09.2013].

[27] "Aptana RadRails," [Online]. Available: http://www.aptana.com/products/radrails. [Accessed 04.09.2013].

[28] K. Subieta, "Stack-based Query Language," in *Encyclopedia of Database Systems*, Springer US, 2009, pp. 2771-2772.

[29] R. Adamus, P. Habela, K. Kaczmarski, M. Lentner, T. Pieciukiewicz, K. Stencel, K. Subieta, M. Trzaska and J. Wislicki, "Overview of the Project ODRA," in *Proceedings of First International Conference on Object Databases (ICOODB) 2008*, Berlin, Germany, 2008.

[30] K. Subieta, "Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL)," [Online]. Available: http://www.sbql.pl/. [Accessed 01.09.2013].

[31] "ODRA (Object Database for Rapid Application development): Description and programmer manual," [Online]. Available: http://www.sbql.pl/various/ODRA/ODRA_manual.html. [Accessed 02.09.2013].

[32] "ODRA eIDE2 Home," [Online]. Available: http://www.eide2.pjwstk.edu.pl/. [Accessed 07.09.2013].

[33] "ODRA IDE (jEdit)," [Online]. Available: http://www.mtrzaska.com/odra. [Accessed 04.09.2013].

[34] "jEdit - Porgammer's Text Editor," [Online]. Available: http://www.jedit.org/.

[35] "Eclipse Juno Release Train Has Arrived," 27.06.2012. [Online]. Available: http://www.eclipse.org/org/press-release/20120627_junorelease.php. [Accessed 06.09.2013].

[36] "Xtext - Language Development Made Easy," [Online]. Available: http://www.eclipse.org/Xtext/. [Accessed 01.09.2013].

[37] "Knockout Framework," [Online]. Available: http://knockoutjs.com/. [Accessed 16.08.2013].

[38] "Jetty - Web Server," [Online]. Available: http://jetty.codehaus.org/jetty/. [Accessed 18.08.2012].

[39] "The WebODRA Framework," [Online]. Available: http://www.mtrzaska.com/webodra. [Accessed 26.08.2013].