A Naked Objects based Framework for Developing Android Business Applications

Fabiano Freitas and Paulo Henrique M. Maia
Academic Master in Computer Science, State University of Ceara, Fortaleza, Brazil

Keywords: Naked Objects, Framework, Android Application.

Abstract: Naked Objects is an architectural pattern in which the business objects are handled directly in the user interface. In this pattern, developers are responsible only for the creation of business classes and do not need to concern with the implementation of the other layers. Although used in several frameworks for making the development of web and desktop systems faster, there is still a lack of tools that add the benefits of that pattern to the creation of Android applications. This paper introduces JustBusiness, a Naked Objects based framework that aims at supporting the creation of Android applications through automatic generation of user interfaces and persistence code from mapping the business classes. Two case studies that describe how the framework was used and its evaluation are also provided.

1 INTRODUCTION

The attention that the Android platform has gained over the recent years has resulted in an increasing demand for applications, a situation justified by the growth of the Android community. According to data from Net Marketshare [1] for March 2015, Android is the most widely used platform on the planet, with a percentage of 47.51%, reaching more than a billion mobile devices among smartphones and tablets.

To answer the market demands, developers and companies are increasingly using frameworks and libraries that can ease the development of applications, thus helping to boost productivity. Among these, we can cite Ormlite (ICE, 2015c), a framework to simplify the access and communication between the Android application and the SQLite database, and Robolectric (ICE, 2010c), a framework to facilitate testing Android applications.

Although useful, those frameworks do not solve or mitigate one of the main application development bottlenecks, which is the creation of user interfaces (UIs) and CRUD (create, read, update and delete) code for business objects, since they do not provide automatic generation mechanisms of those artifacts. That task is already commonly performed to web (Milosavljević et al., 2003) and desktop systems (da Cruz and Faria, 2010). According to Pawson (Pawson, 2004), the development of user interfaces is, most of the times, the task responsible for a significant proportion of all the effort involved in developing an interactive business system due to not only the complexity of coding, but also to the time spent with the presentation details.

As a result, the developer must manually build each of those interfaces and CRUD code, which is a tiring, time consuming and error prone task. Moreover, if the user wants to migrate an existing application from another platform, he/she may have difficult on reusing the business classes code, which implies in a rework to create them on the Android platform.

To fill that gap, this paper presents JustBusiness[2], a framework for developing Android business applications that provides automatic generation of user interfaces and CRUD code from information obtained from the object-user interface mapping of the business classes. For that, the framework implements the architectural pattern Naked Objects (Pawson and Matthews, 2001) (Pawson, 2004), in which the main application parts (the domain objects) are displayed in the interface and the user can manipulate them directly by using those objects' methods. With JustBusiness, the developer is only responsible for implementing the business classes, while the framework performs the heavy task of generation and configuration of all necessary classes and files for running an Android

[1]https://www.netmarketshare.com

[2]Download available at https://jbframework.wordpress.com

application.

The rest of this paper is divided as follow: section 2 explains the main concepts of Naked Objects, while the related work are discussed in Section 3. Section 4 presents the JustBusiness framework, describing its architecture and key features. In Section 5, two case studies describing the implementation of the framework and its evaluation are shown. Finally, Section 6 presents the conclusions and future work.

2 NAKED OBJECTS

The Naked Objects pattern is an object-oriented approach where the domain objects are exposed in the user interface, and the user has the power to manipulate them directly by performing invocations of methods implemented by those objects (Pawson, 2004). In that approach, the code of all classes must respect the object behavioral completeness, one of the most important principles of the object-oriented paradigm and that states that objects must fully implement what they represent (Pawson and Matthews, 2002) (Pawson, 2004) (Raja and Lakshmanan, 2010). The application of that pattern removes from the programmer the need for implementing user interface or security and persistence mechanisms, making him/her responsible only for creating the application domain objects, which must implement all behavior they propose to represent completely.

According to Raja and Lakshmanan (Raja and Lakshmanan, 2010), the Naked Objects approach has three principles that characterize the pattern: (i) the whole business logic should be encapsulated by business objects; (ii) the user interface should reflect the business objects; and (iii) the user interface generation should be automated from the domain objects. Also according to the authors, those principles boost the software development cycle, ease the requirements analysis, bring greater agility and produce more efficient user interfaces.

Naked Objects is an alternative to 4-layers (presentation, control, domain and persistence) architectural pattern. It uses a ratio 1:1 between elements of different layers, where for each domain object there exists only one match in the other layers, while in the 4-layers pattern there may be more complex mappings between the layers (Brandao et al., 2012). The comparison between the two architectural patterns is illustrated in Figure 1.

In (Pawson and Wade, 2003), Pawson and Wade conducted a study about using Naked Objects in an agile software development process. Among the found benefits, the authors point out that the approach foster the concept of exploration phase, in which users and/or customers, along with the development team, perform the UI prototyping simultaneously to the business objects modeling activity.

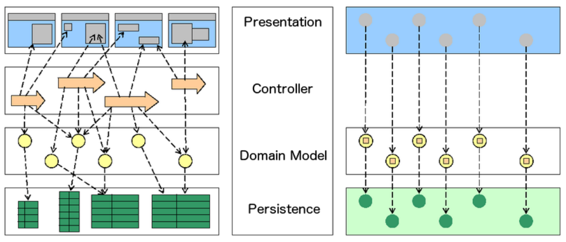


Figure 1: 4-layers (left) and Naked Objects (right) architectural patterns (Pawson, 2004).

3 RELATED WORK

There are several frameworks and tools that promote the development of Naked Objects-based systems. We can highlight the following ones.

1. Naked Objects Framework (ICE, 2015a): one of the pioneer tools. It was developed in Java and used the concept of reflection, which was considered one of the main factors that influenced the choice of the programming language, according to its creator. It is focused on assisting the web and desktop application development.
2. Naked Objects MVC (ICE, 2015b): built upon the Microsoft ASP.NET platform, it aims at creating web applications, providing full-generation of user interfaces using ASP.NET MVC.
3. Apache Isis (ICE, 2010a): an open source framework implemented and focused on Java platform for rapid development of web applications. The Java reflection feature is used along with annotations, with which the programmer makes configuration specifications in the domain objects that will be treated by the framework. With this, Isis is responsible for managing the user interface, security and data persistence resources.
4. JMatter (ICE, 2008): a Java open source project that implements the Naked Objects pattern using, among other resources, Swing and Hibernate. It aims at helping the development of Java web applications by implementing their necessary infrastructure, including all CRUD structures, persistence and search mechanisms. As well as other similar frameworks, JMatter supports the UI automatic generation at runtime.

5. Entities (ICE, 2013a): framework implemented in Java to help developing web applications. One of its main benefits is the possibility of generation of customizable interfaces to the application through an object-user interface mapping. The generated interfaces can be customized through the annotations in the domain classes.

6. Isis Android Viewer (ICE, 2013b): this framework is not a tool to assist the development of Naked Objects-based applications, but rather an Android project to communicate to a web application developed with the Apache Isis framework. An Isis Android Viewer application creates representations of both user interface and persistence based on the domain objects present in the web application.

7. Naked Object for Android (ICE, 2010b): According to the tool's website, this is the first framework implemented to accelerate the Android application development using the Naked Objects principles. The tool, unlike the previous ones, does not support automatic generation of user interfaces nor persistence mechanisms. Currently it is not working, and there has not been recent updates nor maintenance activities.

An extension of the Naked Objects framework using annotations to allow the manipulation of higher-level abstractions, such as specialization of object relationships, is proposed by Broinizi *et al* in (Broinizi et al., 2008). According to the authors, using that approach to validate requirements brings as benefits the reduction of conceptual specification problems, like a weak identification of requirements, decreases the distance between domain and project experts, and allows the simultaneous exploration of conceptual data design and system requirements.

Keranen and Abrahamsson present in (Keranen and Abrahamsson, 2005) a study that compared two mobile development projects of the same mobile application for Java ME platform, where the first one used the traditional development, while the second one was developed using the Naked Objects Framework (NOF). As a result, there was a reduction of 79% in application code and 91% in interface code for the application with NOF. Despite those benefits, the authors concluded that NOF is still not mature enough to develop mobile applications.

Model-driven approaches for the creation and management of user interfaces, in which the software engineer develops the project of the system conceptual models from object oriented meta-models, are described in (Milosavljević et al., 2003) and (da Cruz and Faria, 2010). By applying pre-defined mapping rules, it performs the refinement and transformation of conceptual models, generating user interface and CRUD code from meta-models automatically.

All aforementioned approaches and tools do not support development for current mobile platforms, but rather only for web or desktop ones. Although the approaches proposed in (Keranen and Abrahamsson, 2005) and (Nilsson, 2009) use Naked Objects, they were designed for obsolete mobile platforms. Regarding the Android platform, the only found work was the Naked Objects for Android, but it has been discontinued. The framework proposed in this paper aims at bringing the benefits of Naked Objects to Android developers.

4 JustBusiness

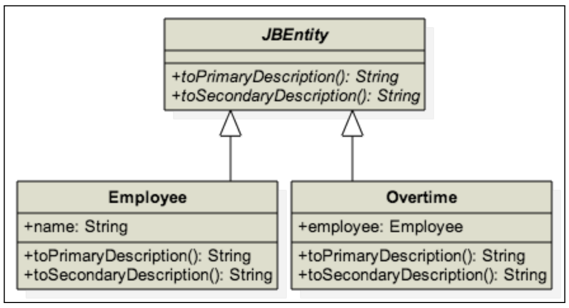
JustBusiness is a framework based on the Naked Objects architectural pattern that provides support for the development of object-oriented business applications in the Android platform and that simplifies the migration of applications from other platforms, such as web ans desktop, to that mobile one. Like in other existing Naked Objects-based frameworks, such as the Naked Objects Framework, Apache Isis, JMatter and Entities, JustBusiness exposes the domain objects in the user interface, allowing users to manipulate them directly via invocations of methods implemented by those objects. Furthermore, it also removes from the programmer the responsibility of building user interfaces and persistence mechanisms, since it supports the automatic generation of those artifacts.

Besides being used for development and migration of business applications for the Android platform, JustBusiness can be used in the generation of the initial skeleton of the application. By providing automatic code generation, the developer can give up from using JustBusiness at any time in the development application process without the loss of source code that has already been produced.

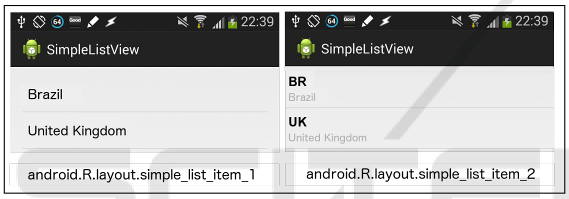
More details about the JustBusiness framework are shown in the following sections.

4.1 Architecture

The framework has in its class structure, the abstract superclass *JBEntity*, which must be specialized by all business classes of the application project. The *JBEntity* class has no attributes and contains only two methods: *toPrimaryDescription* and *toSecondaryDescription*, which must be implemented by its subclasses, as shown in Figure 2.


JBEntity
+toPrimaryDescription(): String
+toSecondaryDescription(): String
Employee
+name: String
+toPrimaryDescription(): String
+toSecondaryDescription(): String
Overtime
+employee: Employee
+toPrimaryDescription(): String
+toSecondaryDescription(): String
Figure 2: Extending the superclass JBEntity.

Those methods provide information about the object in the user interface, focusing on list screens (ListActivities). By default, the list of cells that are used by ListActivities has one or two fields to display information, as shown in Figure 3. Instead of using a single *toString* method, the structure with two methods supports more details in the interface, as well as the use of the two types of basic standard Android cells.


22:39
SimpleListView
Brazil
United Kingdom
BR
Brazil
UK
United Kingdom
android.R.layout.simple_list_item_1
android.R.layout.simple_list_item_2
Figure 3: Listing cell standard Android platform.

The business classes should be implemented in a simplified way using a standard constructor, private attributes, and its respective *get* and *set* methods. The structure of the business classes is similar to a POJO (Plain Old Java Object) class, but differs from it because they extend the *JBEntity* superclass and use predefined notes in its construction.

JustBusiness uses a processor that analyzes, at compile time, each annotation used in the configuration of the business classes. Information obtained from the processing are stored in a data dictionary that contains all the information of classes, enumerations, attributes, methods and mapped parameters. When a "*clean and rebuild*" action is performed in the project, the saved information in the data dictionary are processed by code generators, which are responsible for creating and configuring all the needed files to deploy the application and run the project.

For each business class in the project, at least 25 files are created, as illustrated by Figure 4. Those files consist of control and data access classes and resource files, which are divided into layout and menu. For each of those files, there is a code generator that encodes them based on information obtained from the annotations. In addition, JustBusiness also creates a

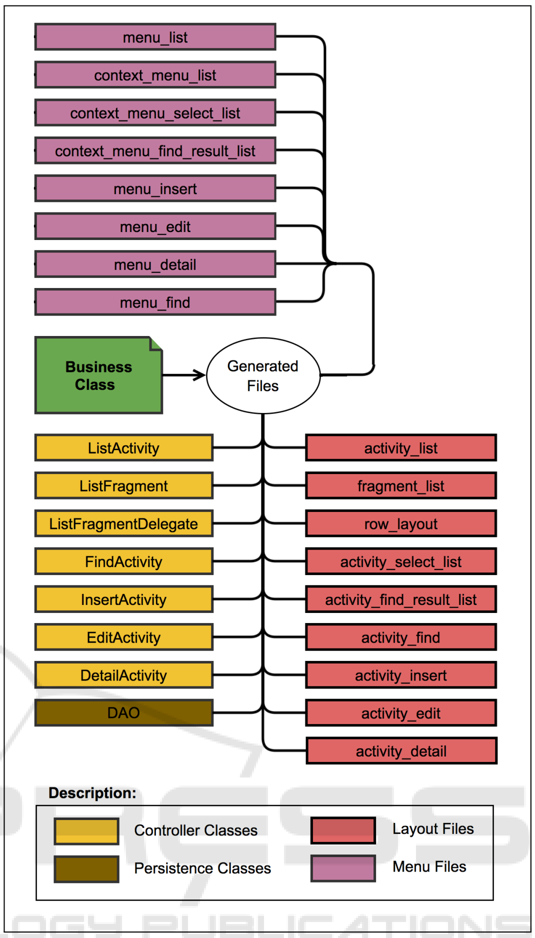


Figure 4: Structure of generated files to the business classes.

set of classes and resource files, based on information from annotations and business classes, for project organization and execution.

From the mapped information, three control classes are created to support the business application initial screen, which brings the list of entities contained in the application. The *dimension* and *styles* resource files are modified, bringing general information for dimensions and layout styles, respectively, while the *strings* file is modified to store all text content that will be used in the application. Furthermore, the *persistence* file, which contains persistence settings to access the database, is created. Finally, the *AndroidManifest* file is also altered.

For each new file creation or existing project file update, there is a code generator associated. The structure of project files generated or updated is shown in Figure 5.

4.2 Main Features

JustBusiness provides a set of annotations to enable

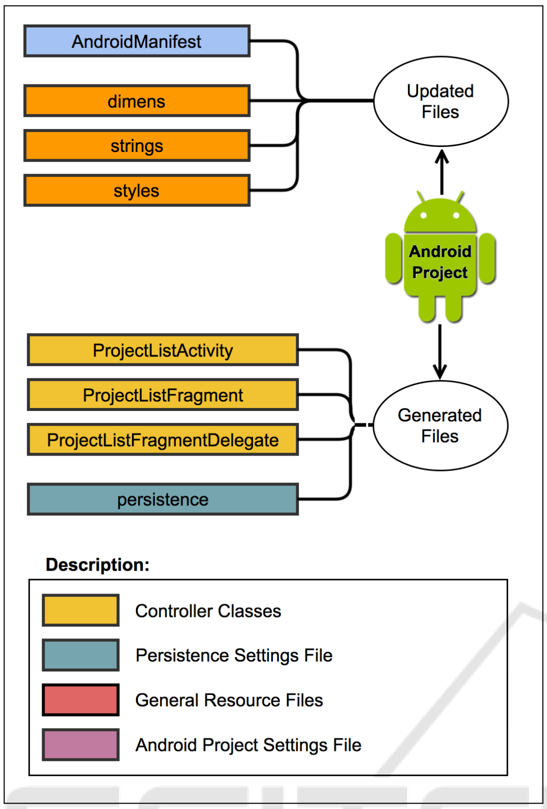


Figure 5: Structure of generated or changed project files.

the mapping information and to configure the business classes. Through those annotations, the programmer can detail information ranging from presentation settings in the UI to data persistence parameters.

In addition to the annotations feature, the framework uses a code generation mechanism that consists of a set of classes that, using information obtained from the mapping, generate automatically classes, resources and settings required for the production of an Android application.

4.2.1 Annotations

In order to configure and map information in the business classes, two sets of annotations have been defined: one for user interface and the other one for persistence. The former aims at providing details of the visual settings, while the latter details information for the object-relational mapping. The information and values passed through those annotations provide a greater level of detail about the mapped elements.

Annotations for User Interface. A single class, with its attributes and methods, does not have all necessary information for the complete generation of a graphical interface. To fill this gap, JustBusiness provides a set of annotations that allows the programmer to inform the missing information needed to perform that task.

Using the annotations, it is possible to define settings such as the business classes that will be recognized by the framework, the attributes that will be displayed on the screen and in which order, and the methods that will be available in the interface. JustBusiness uses the annotations *@Entity*, *@Attribute*, *@Action*, *@Parameter* and *@Enumeration* to identify and configure the classes, attributes, methods, parameters from methods and enumerations, respectively. Those annotations are detailed in Table 1.

Annotations for Persistence. The Android mobile platform, by default, uses the SQLite database, a very simple and limited database. One of its main limitations refers to the few supported data types (only INTEGER, TEXT, NONE, REAL and NUMERIC). Another important factor that should be considered is that the Android platform does not support the Java Persistence API (JPA) and does not recognize the *javax.persistence* package, which consists of a set of annotations and other classes aimed at mapping persistence information.

Given those limitations, JustBusiness has added some persistence features to provide object-relational mapping between business classes and SQLite database tables. For this, a set of annotations, inspired on the *javax.persistence* annotations, has been defined. Furthermore, the framework provides a mechanism for creating tables and SQLite database access from the information mapped with those persistence annotations. The set of annotations used for data persistence is listed in Table 2.

4.2.2 Automatic Code Generation

The framework supports the automatic generation of all necessary infrastructure for an Android application, including classes and user interface resources, and the SQLite database and data access classes. The programmer is responsible only for implementing the business classes and configuring them to use the framework. The code generation occurs at compile time, since, at that moment, interface classes and resource files are created, and some project configuration files, such as the *AndroidManifest*, are modified to incorporate the changes made by the framework.

Table 1: Annotations for User Interface.

Annotation	Description
@Entity	Describes and configures the classes that will be recognized by the JustBusiness. Parameters: **label** - Name that will be displayed for the entity **collectionLabel** - Plural name for the entity **icon** - Name of the image to be used as icon by the entity. The image must be in the folder (res/drawable)
@Attribute	Describes and configures the class attributes. Parameters: **name** - Name that will be displayed for the attribute **order** - Order in which the attribute is shown in the user interface **views** - Screens in which the attribute is displayed. It can be used more than one value. The possible values are: KindView.ALL: All screens KindView.INSERT: Entry screen KindView.EDIT: Update screen KindView.DETAIL: Detail screen KindView.FIND: Search screen
@Action	Describes and configures the class methods. Parameters: **name** - Name that will be displayed for the method **order** - Order in which the method is shown in menu
@Parameter	Describes and configures parameters of class methods. Parâmetros: **name** - Name that will be displayed for the parameter **order** - Order of the parameter in the form
@Enumeration	Describes enumerations.

User Interface Code Generation. Through the information obtained from the mapping of classes, attributes, relationships, and methods using the aforementioned notes, the framework generates the whole user interface mechanism.

Table 2: Annotations for Persistence.

Annotation	Description
@Table	Identifies a class and sets it as a table in the database.
@Id	Identifies an attribute as a key in the table.
@Column	Identifies and configures an attribute as a column in the table.
@JoinColumn	Identifies and configures a column as an attribute to perform a join operation with another table.
@Transient	Identifies an attribute that is not mapped in the table.
@OneToMany	Identifies and configures an attribute as a 1:N relationship.
@OneToOne	Identifies and configures an attribute as a 1:1 relationship.
@ManyToOne	Identifies and configures an attribute as a N:1 relationship.
@ManyToMany	Identifies and configures an attribute as a N:M relationship.
@JoinTable	Identifies and configures an attribute as a column for the operation join with another table.
@Enumerated	Identifies an attribute as an enumeration.
@Temporal	Sets an attribute that stores temporal information.

For each business class, the framework constructs the interfaces for inserting, editing, detailing and searching information as forms, where the components associated with each class attribute are arranged on the screen in a single vertical column according to the sequence determined by the developer in the business class. Those components can be enabled or disabled on the interface by setting the *@Attribute* annotation in the business class.

According to the Naked Objects pattern, changes are made exclusively in the business model, so the developer does not need to modify the interfaces directly. Therefore, in a project using JustBusiness, changes occur only in the business classes and, to incorporate that modifications in the project, the programmer only needs to recompile it, thus increasing the application's maintenance and evolution level.

Persistence Code Generation. By mapping the classes and their attributes and relationships using the proposed annotations, the framework generates the entire SQLite database automatically, including tables and keys. In addition, for each mapped class in the project, the framework automatically generates a data access class using the Data Access Object (DAO) pattern.

Project Files Configuration. In addition to generating the control classes (Activities) and interface layout files, JustBusiness is also responsible for modifying the project configuration files, such as the *AndroidManifest*, which should be modified at compile time so that the project can identify all control classes that have been added, as well as receive information such as nomenclature and application icon. Besides the *AndroidManifest*, the resource files *strings*, *dimens* and *styles* are also modified.

Internationalization. The textual information mapped by the programmer using the annotations in business classes are compiled and stored in the *strings* resource file, which contains the words used within the application context. By using this approach, the application can be easily adapted to support a new language or dialect further.

4.3 Setting Up a Project with JustBusiness

To use the JustBusiness framework in an Android project, it is necessary to follow the steps depicted in Figure 6. As JustBusiness was designed, initially, to be used with Eclipse, we consider the use of that IDE to perform the following steps.

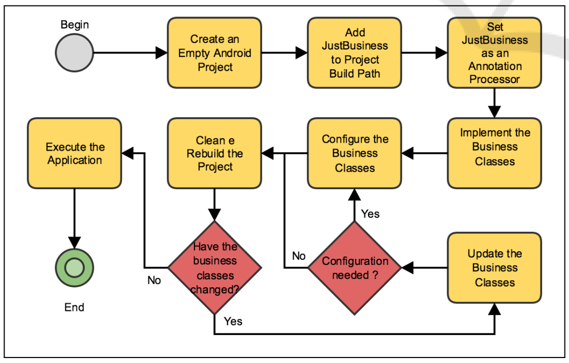

Begin
Create an Empty Android Project
Add JustBusiness to Project Build Path
Set JustBusiness as an Annotation Processor
Implement the Business Classes
Configure the Business Classes
Clean e Rebuild the Project
Execute the Application
End
Have the business classes changed?
No
Yes
Configuration needed ?
Yes
No
Update the Business Classes

Figure 6: Flowchart for the use of JustBusiness.

Firstly, the developer must create an empty Android project and then configure it to recognize the framework. The project configuration consists of two activities: adding the framework to the project libraries and configuring JustBusiness as an annotation processor in the project properties.

The next step consists of implementing the business classes, which must inherit the *JBEntity* superclass and override its abstract methods. Subsequently, the business classes must be consistently annotated with annotations provided by the framework in order to build the interface and persistence mechanisms.

Finally, the project should be cleaned and rebuilt, since the code generation occurs at compile time. After that, the project is ready to be executed. If a new change in a business class is carried out, the developer must verify whether it is necessary to reconfigure the classes using the annotations and, if so, repeat the project clean and rebuild step.

5 CASE STUDY

To demonstrate the use of JustBusiness, a case study that consisted of the development of an application for request and approval of service overtime, as used in (Brandao et al., 2012), was carried out. The scenario starts with the employee requesting a service overtime, stating the justification and the initial and final date. The requests are firstly reviewed by the supervisor, who can either authorize or reject them. The authorized requests will be reviewed by the Human Resources (HR) sector to calculate and approve the payment of the hours. If HR has some questions, the request may return to the supervisor. Figure 7 is the resulting class diagram of the proposed case study.

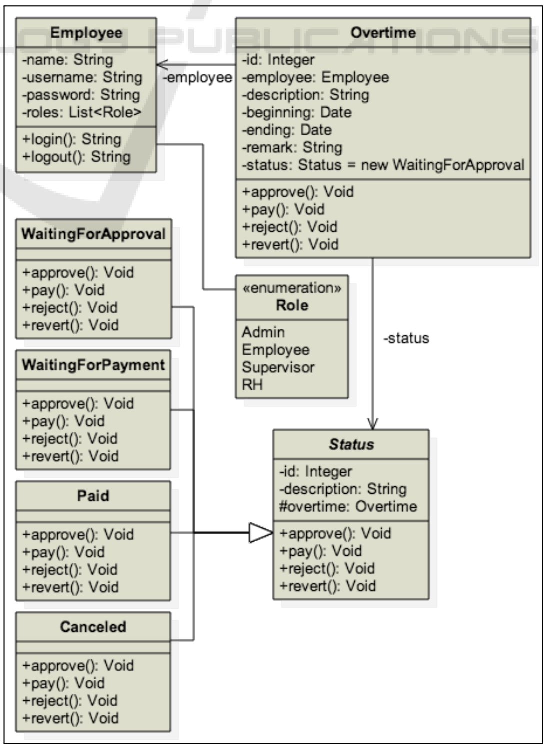


Figure 7: Class diagram. Source: (Brandao et al., 2012).

5.1 Applying JustBusiness to the Described Scenario

To use JustBusiness, the developer needs to implement only the business classes with their attributes, methods and relationships. He/she should firstly identify which business model elements are classes and which ones are enumerations, since there is a different treatment for each one. Classes must specialize the superclass *JBEntity*, have a default constructor with no arguments, *get* and *set* methods for each attribute, and must be mapped with the annotations *@Entity* and *@Table*. Enumerations should only be mapped with the annotations *@Enumeration* and *@Table*. Listing 1 shows part of the *Overtime* class source code. For the sake of simplicity, some details like *gets* and *sets*, as well as of some attributes and methods settings have been omitted.

In line 1 the *@Entity* annotation is used to indicate that *Overtime* class will be recognized by JustBusiness as a business class. Using it, the developer should inform the class *label*, that corresponds to the class name, and the *collectionLabel*, which is equivalent to the class plural name. In line 2, the *@Table* annotation is used to create the *overtime* table, whose name was informed in the *name* parameter, and to create the DAO classes with the SQL queries.

After, in line 4, the *@Id* annotation is used to indicate that the attribute *id* is a key in the *overtime* table. The *@Column* annotation determines, in line 5, that the id attribute correspond to the *id_overtime* column in the table. In line 6, the *@Attribute* annotation specifies that an element should be recognized as an attribute with label *Identifier*, has *order* 1, i.e., it will be first element on screen, and will appear just in the detail view in the user interface.

Then, in line 9, the *@ManyToOne* annotation specifies that an attribute represents a relationship N:1 with the *Employee* class, mapped through the *targetEntity* parameter. In the next line, the *@JoinColumn* annotation informs that an attribute will perform a join operation through the *id_employee* with the table mapped by the *Employee* class. In line 11, the *@Attribute* annotation specifies that an element should be recognized as an attribute with label *Employee*, has *order* 2, i.e., it will be second element on screen, and will appear in all views in the user interface.

Finally, the maps relating to methods *approve* and *pay* can be seen in the lines 24 and 29, respectively. The *@Action* annotation is used to identify a method that will be available for the user. The *name* parameter value is the title of the button that calls the method, while the *order* parameter specifies the position in the menu where the method appears.

```
 1 @Entity(label="Overtime",collectionLabel="Overtimes")
 2 @Table(name="overtime")
 3 public class Overtime extends JBEntity {
 4     @Id
 5     @Column(name="id_overtime",nullable=false,unique=
         true)
 6     @Attribute(name="Identifier",order=1,views={KindView
         .DETAIL})
 7     Integer id;
 8
 9     @ManyToOne(targetEntity="business.Employee")
10     @JoinColumn(name="id_employee",nullable=false,unique
         =false)
11     @Attribute(name="Employee",order=2,views={KindView.
         ALL})
12     Employee employee;
13
14     //Annotations omitted
15     Date beginning;
16     Date ending;
17     String description;
18     String remark;
19     Status status = Status.WAITING_FOR_APPROVAL;
20
21     //Getters and Setters omitted
22     //toPrimaryDescription and toSecundaryDescription
         methods omitted
23
24     @Action(name="Approve",order=1)
25     public void approve() {
26         status.approve(this);
27     }
28
29     @Action(name="Pay",order=2)
30     public void pay() {
31         status.pay(this);
32     }
33
34     //Other methods omitted
35 }
```

Listing 1: Source Code of Overtime Class.

Listing 2 shows the simplified source code of the *Employee* class. Like the *Overtime* class, for simplicity, some details like *gets* and *sets*, as well as some attributes and methods settings have been omitted. To avoid repetition, only the annotations that have not been used in the the *Overtime* class will be explained.

In line 15, the *@ManyToMany* annotation specifies that an attribute represents a relationship N:N with the *Role* enumeration, mapped through the parameter *targetEntity*. In the next line, the *@JoinTable* annotation informs that the *roles* attribute will perform a join operation with the table mapped by the *Role* enumeration (shown in Listing 4) using the intermediate table *employee_role*.

```
 1 @Entity(label="Employee",collectionLabel="Employees")
 2 @Table(name="employee")
 3 public class Employee extends JBEntity {
 4     @Id
 5     @Column(name="id_employee",nullable=false,unique=
         true)
 6     @Attribute(name="Identifier",order=1,views={KindView
         .DETAIL})
 7     Integer id;
 8
 9     //Annotationsomitted
10     String name;
11     String username;
12     String password;
13     List<Overtime> overtimes;
14
15     @ManyToMany(targetEntity="business.Role")
16     @JoinTable(name="employee_role",
17         joinColumns={@JoinColumn(name="id_employee",
18         referencedColumnName="id_employee")},
19         inverseJoinColumns={@JoinColumn(name="id",
20         referencedColumnName="id_role")})
21      @Attribute(name="Roles",order=5,views={KindView.ALL
          })
22      List<Role> roles;
23
24      //Getters and Setters omitted
25      //toPrimaryDescription and toSecundaryDescription
          methods omitted
26
27      @Action(name="Login",order=1)
28      public String login() {
29          return null;
30      }
31
32      @Action(name="Logout",order=2)
33      public String logout() {
34          return null;
35      }
36  }
```

Listing 2: Source Code of Employee Class.

As previously mentioned, the enumerations are identified and configured differently of the classes since they consist of limited entities with a structure already set. Listings 3 and 4 show the definition of the enumerations *Status* and *Role*, respectively. In both source codes, the *@Enumeration* annotation, in line 1, is used to identify the enumerations that will be recognized by JustBusiness, while the *@Table* annotation is used in line 2 to create a table in the database whose name will be the value of the *name* parameter.

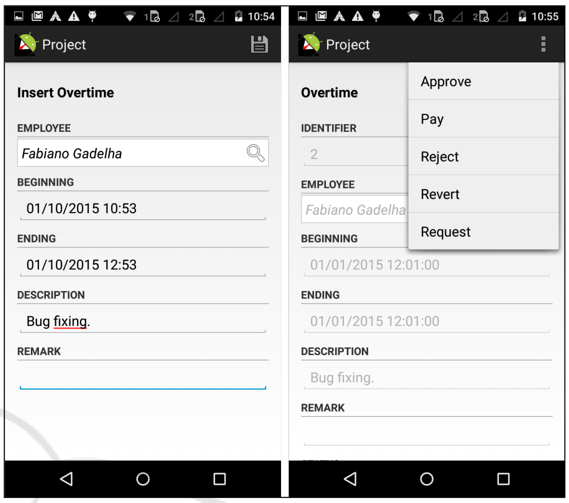
```
1   @Enumeration
2   @Table(name="status")
3   public enum Status {
4       WAITING_FOR_APPROVAL{
5           public void approve(Overtime overtime) {
6               overtime.setStatus(WAITING_FOR_PAYMENT);
7           }
8           public void pay(Overtime overtime) {
9               //Donothing
10          }
11          public void reject(Overtime overtime) {
12              overtime.setStatus(CANCELED);
13          }
14          public void revert(Overtime overtime) {
15              //Donothing
16          }
17      },
18      WAITING_FOR_PAYMENT {},
19      PAID{},
20      CANCELED{};
21
22      public abstract void approve(Overtime overtime);
23      public abstract void pay(Overtime overtime);
24      public abstract void reject(Overtime overtime);
25      public abstract void revert(Overtime overtime);
26
27      @Override
28      public String toString(){
29          return this.name();
30      }
31  }
```

Listing 3: Source Code of Status Enumeration.

```
1   @Enumeration
2   @Table(name="role")
3   public enum Role {
4       ADMIN,
5       EMPLOYEE,
6       SUPERVISOR,
7       RH
8   }
```

Listing 4: Source Code of Role Enumeration.

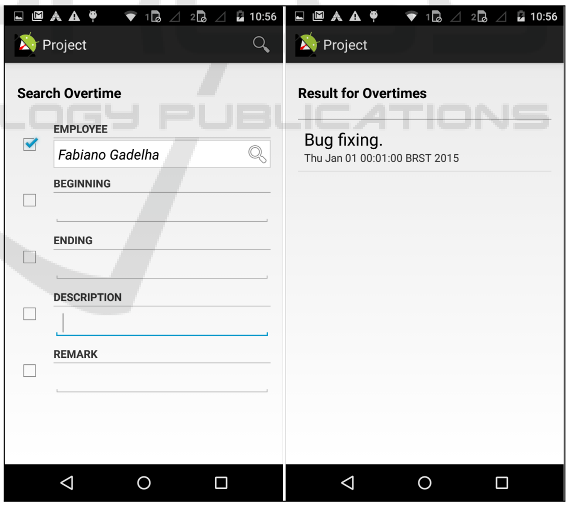
Figure 8 displays the user interfaces for inserting and detailing the automatically generated information for the *Overtime* class from the information mapped in the class code, as shown in Listing 1. Figure 9 shows the user interfaces for the search operation and search result listing by type *Overtime* objects. Figure 10 displays the screen that contains the list of all objects of type *Overtime*. From that screen, the user can access the individual objects and perform operations on them.



Insert Overtime
EMPLOYEE: Fabiano Gadelha
BEGINNING: 01/10/2015 10:53
ENDING: 01/10/2015 12:53
DESCRIPTION: Bug fixing.
REMARK:

Overtime
IDENTIFIER: 2
EMPLOYEE: Fabiano Gadelha
BEGINNING: 01/01/2015 12:01:00
ENDING: 01/01/2015 12:01:00
DESCRIPTION: Bug fixing.
REMARK:

Figure 8: Entry and detail screens of object from type Overtime.



Search Overtime
☑ EMPLOYEE: Fabiano Gadelha
☐ BEGINNING:
☐ ENDING:
☐ DESCRIPTION:
☐ REMARK:

Result for Overtimes
Bug fixing.
Thu Jan 01 00:01:00 BRST 2015

Figure 9: Search and result list screens of objects from type Overtime.

5.2 Evaluation

To assess the benefits of using JustBusiness framework, two comparative experiments have been conducted. For the first analysis, we developed two projects of the same Android application based on the scenario described in the previous detailed case study: the first one using the traditional development model, i.e., without

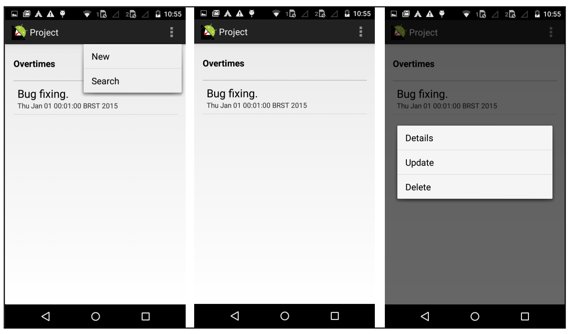


Figure 10: Listing screen of objects from type Overtime

JustBusiness framework, while the second one using the framework. This assessment included a single programmer, who developed the two projects: initially the one without the framework and, subsequently, the other one using JustBusiness.

At the end of the implementation of the two projects, it was found that the project developed with JustBusiness achieved a reduction of approximately 96% of development time, 91% of written lines of code and 94% of files created by the developer. Comparative data for the two developed projects are detailed in Table 3.

Table 3: Comparative analysis of JustBusiness and the Traditional Development.

Type of Development	JustBusiness	Traditional
Time	27 minutes	720 minutes
Lines written by Developer	376	4315
Files created by Developer	4	66
Total files in the Project	66	66

To try to get around one of the threats to the validity of that experiment, in which only one developer was used, the second experiment consisted of another case study, this time involving 4 developers, all with one or two years of experience in Android development. Their task was to implement, with and without the framework, a simple project involving three business classes, shown in the diagram of Figure 11.

In this case study, each developer has implemented both projects individually, i.e., there was no collaboration among them in any project. As in the first experiment, each developer first implemented the project without using JustBusiness and, after, using it.

At the end of the second case study, it was observed that, on average, the project developed with JustBusiness decreased approximately 93% of development time, 93% of written lines of code and 96% of files created by the developer, corroborating the results of the

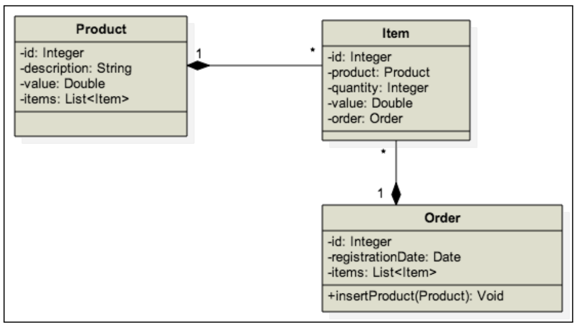


Figure 11: Class Diagram of the Second Case Study.

previous experiment that indicated gains in productivity when the framework was used. Comparative data for the two developed projects are detailed in Table 4, where the values represent the arithmetic averages of the individual values for each developer.

Table 4: Comparative analysis of JustBusiness and the Traditional Development of the Second Case Study.

Type of Development	JustBusiness	Traditional
Time	50 minutes	743 minutes
Lines written by Developer	256	3935
Files created by Developer	3	79
Total files in the Project	79	79

Finally, we identify as other threats to the validity of the experiments the developers' knowledge level in the Android platform and the difficulty level of the developed projects. Although those factors possibly would imply changes in the values of the items evaluated for each developer, particularly the development time and written lines of code, we believe that the resulting values for the projects that used the framework will be even lower than the ones obtained by the projects without the framework due to the difference in the number of files generated for both cases. We intend to conduct other case studies to prove that hypothesis.

6 CONCLUSION AND FUTURE WORK

This work presented JustBusiness, a framework for developing Android business applications using the Naked Objects architectural pattern. The main benefit of the framework is the automatic generation of user interfaces and CRUD code, thus accelerating the Android application development. Two case studies have been carried out and demonstrated that the use of the framework promotes a gain in productivity, since it reduces the development time and the number of lines of code and files generated by developers, when compared to solutions that have not used JustBusiness.

Despite its advantages, the framework has some limitations, such as the support to only local data persistence in SQLite database and the lack of customization in the interfaces that were generated automatically.

As future work, we intend to add model-driven developement techniques to the code generation task and provide support for other data types, like images and videos, and other data persistence mechanisms, such as XML and JSON. Additionally, it is intended to introduce validation mechanisms for forms components. Finally, we plan to conduct a study to improve the usability of the generated interfaces.

REFERENCES

(2008). Jmatter. http://jmatter.org/. [Online; accessed 9-July-2015].

(2010a). Apache isis. http://isis.apache.org. [Online; accessed 9-September-2015].

(2010b). Naked object for android. http://sourceforge.net/projects/noforandorid/. [Online; accessed 10-October-2015].

(2010c). Robolectric. http://robolectric.org/. [Online; accessed 13-October-2015].

(2013a). Entities. http://entitiesframework.blogspot.com.br/. [Online; accessed 7-October-2015].

(2013b). Isis android viewer. https://github.com/DImuthuUpe/ISISAndroidViewer/. [Online; accessed 7-August-2015].

(2015a). Naked objects framework. http://www.nakedobjects.org/. [Online; accessed 9-September-2015].

(2015b). Naked objects mvc. http://nakedobjects.net/. [Online; accessed 9-September-2015].

(2015c). Ormlite. http://ormlite.com. [Online; accessed 14-October-2015].

Brandao, M., Cortes, M., and Goncalves, E. (2012). Entities: A framework based on naked objects for development of transient web transientes. In *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, pages 1–10.

Broinizi, M. E. B., Ferreira, J. a. E., and Goldman, A. (2008). Using annotations in the naked objects framework to explore data requirements. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 630–637, New York, NY, USA. ACM.

da Cruz, A. M. R. and Faria, J. P. (2010). A metamodel-based approach for automatic user interface generation. In Petriu, D. C., Rouquette, N., and Haugen, O., editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin Heidelberg.

Keranen, H. and Abrahamsson, P. (2005). Naked objects versus traditional mobile platform development: a comparative case study. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 274–281.

Milosavljević, B., Vidaković, M., Komazec, S., and Milosavljević, G. (2003). User interface code generation for ejb-based data models using intermediate form representations. In *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*, PPPJ '03, pages 125–132, New York, NY, USA. Computer Science Press, Inc.

Nilsson, E. G. (2009). Design patterns for user interface for mobile applications. *Adv. Eng. Softw.*, 40(12):1318–1328.

Pawson, R. (2004). *Naked Objects*. PhD thesis, University of Dublin, Trinity College.

Pawson, R. and Matthews, R. (2001). Naked objects: A technique for designing more expressive systems. *SIGPLAN Not.*, 36(12):61–67.

Pawson, R. and Matthews, R. (2002). Naked objects. In *Companion of the 17th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 36–37, New York, NY, USA. ACM.

Pawson, R. and Wade, V. (2003). Agile development using naked objects. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, XP'03, pages 97–103, Berlin, Heidelberg. Springer-Verlag.

Raja, A. and Lakshmanan, D. (2010). Article: Naked objects framework. *International Journal of Computer Applications*, 1(20):37–41. Published By Foundation of Computer Science.