Migrating Large Object-oriented Applications into Component-based ones

Instantiation and Inheritance Transformation

Zakarea Alshara Abdelhak-Djamel Seriai Chouki Tibermacine
Hinde Lilia Bouziane Christophe Dony Anas Shatnawi

LIRMM, CNRS and University of Montpellier, France
{alshara, seriai, tibermacin, bouziane, dony, shatnawi}@lirmm.fr

Abstract

Large object-oriented applications have complex and numerous dependencies, and usually do not have explicit architectures. Therefore they are hard to maintain, and parts of them are difficult to reuse. Component-based development paradigm emerged for improving these aspects and for supporting effective maintainability and reuse. It provides better understandability through a high-level architecture view of the application. Thereby migrating object-oriented applications to component-based ones will contribute to improve these characteristics. In this paper, we propose an approach to automatically transform object-oriented applications to component-based ones. More particularly, the input of the approach is the result provided by software architecture recovery: a component-based architecture description. Then, our approach transforms the object-oriented source code in order to produce deployable components. We focus in this paper on the transformation of source code related to instantiation and inheritance dependencies between classes that are in different components. We experimented the proposed solution in the transformation of a collection of Java applications into the OSGi framework. The experimental results are discussed in this paper.

Keywords Component, Object, Code Transformation, Refactoring, Inheritance, Encapsulation, Class Instantiation, Java, OSGi

1. Introduction

Most existing large legacy applications are object-oriented (OO) [19]. These applications have complex and numerous internal dependencies. Usually, they do not have explicit architectures, in which coarse-grained high-level entities and their dependencies are described. This makes these applications hard to maintain (understand and change), and hard to reuse.

Conversely, component-based (CB) applications have coarse-grained high-level architecture views [7]. They have loosely coupled and highly cohesive entities that have explicit dependencies. Therefore, these applications are easy to understand through their architecture views, and it is easy to reuse their coarse-grained entities [5, 9]. Thus migrating these OO applications to CB ones contributes to improve their maintainability, in addition to their reusability by feeding existing component repositories [12].

The migration process is composed of two main steps [1]: the first step is CB architecture recovery where components and their dependencies are identified. The second step is code transformation where OO code is transformed into equivalent CB one.

The step of CB architecture recovery was largely treated in the literature [2–4, 7, 9]. Most of these works aim to identify components as clusters of classes. They use clustering algorithms, among other techniques, aiming at maximizing intra-component cohesion and minimize inter-component coupling to identify the architectural elements (components and connectors). CB architecture recovery aims to identify components and connectors but not to create them. It does not transform these clusters of classes into a concrete component model. Moreover, the dependencies between clusters remain OO ones.

Code transformation (The second step) aims at creating programming level components by transforming OO code. It aims at creating components based on given clusters of classes. Component encapsulation (i.e., Interaction through explicit provided and required interfaces) is a main characteristic of component and therefore a major difference between the component and object concepts [16]. Thus, object to component transformation consists on transforming OO dependencies between classes belonging to different clusters to interactions through interfaces to avoid component encapsulation violation. It needs the transformation of OO mechanisms used at the implementation level (i.e. instantiation, inheritance, exception handling, etc.) into ones related to CB (interface-based connections) [12].

This paper proposes a method that automatically transforms an OO application code to a CB one. We assume that an existing CB architecture recovery method provides us architecture descriptions as an input to our method. Based on the taxonomy of component models proposed in [11], we chose, as the target of our transformation, an object-based component model (i.e. components implemented based on OO source code). These component models are implemented as an extension of mainstream OO programming languages (e.g. OSGI is an extension of Java [13], CCM is an extension of C++ [11]). This choice allows us to reuse the OO source code to be migrated. In this work, we experimented the proposed solution on the transformation of Java applications into the OSGi framework.

The remainder of this paper is organized as follows. Section 2 presents the migration process and its related issues. Section 3 explains the proposed solution to transform class instantiation depen-

dencies. Section 4 describes our solution to transform OO inheritance relationships. Section 5 presents implementation and experimental results. Before concluding, we present the related work in Section 6.

2. Problem Statement

To better illustrate the problem and solutions related to the OO-to-component migration, we introduce an example of a simple Java application. This application simulates the behavior of an information screen (e.g. a software system which displays on a bus's screen information about stations, time, etc.).

In Figure 1, *ContentProvider* class implements methods which send text messages (instances of *Message*), and time information obtained trough *Clock* instances based on the data returned by *TimeZone* instances. The *DisplayManager* is responsible for viewing the provided information through a *Screen*.

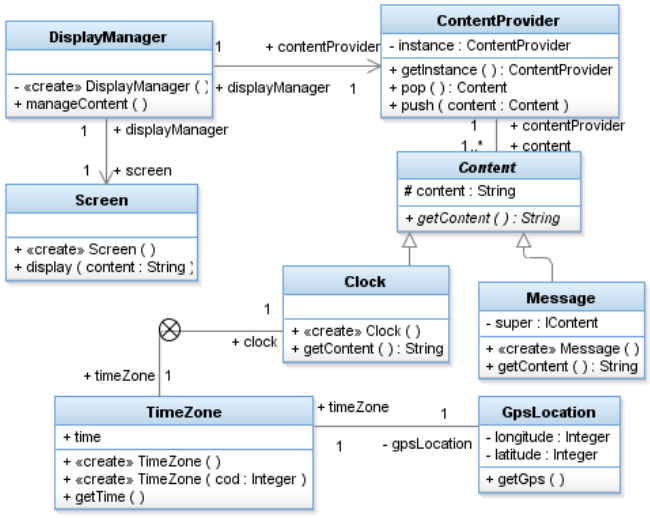

Figure 1. Information screen class diagram.

2.1 Component-based Architecture Recovery

Architecture recovery was largely studied in the literature [3]. In these works a software component is recovered as a cluster (set) of classes that collaborate with each other to provide the component functionalities [6]. In most of these works, a cluster is identified based on the definition of a fitness function and a clustering algorithm. For example, in our previous works [4, 9], we have proposed an approach which aims to recover component-based architectures based on a fitness function and a search-based algorithm. The fitness function is based on quality measurements on the component-based architecture (i.e. maintainability, reliability, autonomy, specificity and composability). The search-based algorithm aims to maximize this fitness function.

Figure 2 shows the result of architecture recovery step applied on our example. The recovery step identifies five clusters, each cluster contains one or several classes. For example *Component1* is responsible for displaying information on the screen through the collaboration of its classes, *Screen* and *DisplayManager*. A cluster is composed of two types of classes: internal classes and boundary classes. Internal classes are classes that do not have dependencies (e.g. a method invocation or an inheritance relationship) with other classes placed into other clusters (e.g. *GpsLocation* and *Screen*). And the boundary classes are classes that have dependencies with classes placed into other clusters (e.g. *TimeZone* and *Clock*). We consider a component-based architecture as a set of components connected via interfaces, where interfaces are identified from boundary classes.

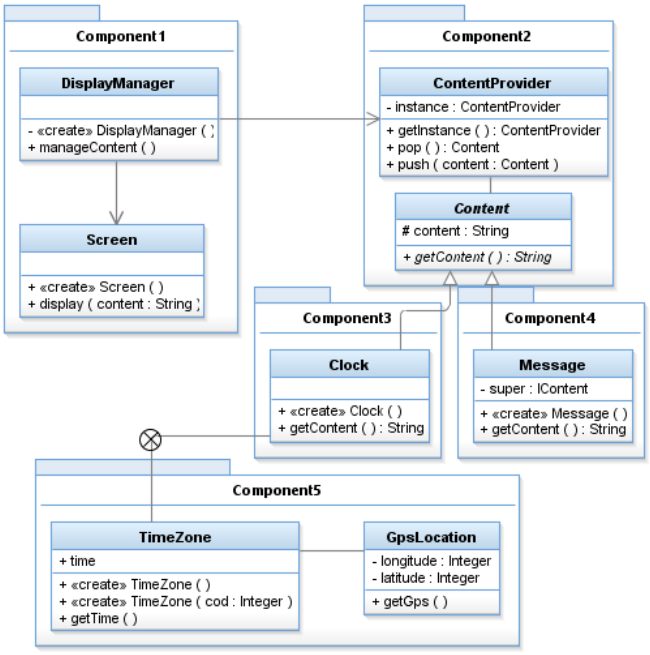

Figure 2. Architecture recovery for the information screen.

2.2 Code Transformation

In this paper we use clusters of classes obtained based on recovery approaches as an input of the source code transformation step. To transform clusters of classes to components we need to solve two main problems:

1) Explicit component encapsulation violation: the component must hide its internal structure and behavior [16]. It should provide its services without exposing the classes that implement it. Two source code expressions fall under this category. First, "class instantiation", where a class creates an instance of another class placed on a different component. For example, *Clock* class creates an object of *TimeZone* class, while these classes belong to two different clusters. Second, "method invocation", where a method defined in a given class of a cluster invokes a method defined in a class placed in another cluster.

2) Implicit component encapsulation violation: It is related to implecit dependency between components caused by OO mechanisms, such as inheritance, exception handling and event handling. For instance, for the inheritance mechanism, a class and its subclasses cannot be necessarily placed in the same cluster. This is the case in Figure 2 for *Clock* and *Message* subclasses of *Content* Class. In this case, the inheritance relationship between these classes crosses component boundaries, facing an implicit dependency between the underlying components. Since component models do not all support inheritance (e.g. ComponentJ, COM, etc.) [15], source code related to inheritance need to be transformed.

In this paper we focus on solutions related to source-code transformations of explicit component encapsulation violation (instanti-

ation and method invocation) and OO inheritance which is a case of implicit component explicit violation.

3. Instance Handling Transformation

Considering the result of the recovery step, a class belonging to a component (cluster) can be instantiated in a method of class belonging to another component by using directly this class constructors. This causes a violation of the principle of component encapsulation. Our approach proposes two steps to transform direct instantiation dependencies: (i) Uncoupling classes belonging to different components (clusters) by creating object interfaces. (ii) Defining specific component interfaces playing the role of object factories.

3.1 Creating Object Interfaces: Uncoupling Boundary Classes

We transform direct references (method calls) between classes of different components to interface-based calls. Thus when a class *A* uses class *B* where *A* and *B* are parts of two different components, we create a couple of the same provided and required interfaces (*IB*). The provided one will be defined in the component of the class *B* and the required one in the component of the class *A*. Theses interfaces define the same methods of all public methods of class *B*. In addition they define other methods to access public attributes of this class (i.e. setter and getter methods). Each direct use of class *B* in the class *A* will be refactored as a use of the required interface (*IB*) added to the component of *A*.

To illustrate this, consider our illustrative example, where *Clock* creates an instance of *TimeZone*. This is depicted in Listing 1. We create *ITimeZone* interface for class *TimeZone*. *ITimeZone* specifies the signatures of all public methods in *TimeZone*. Moreover, it declares setter and getter methods for its public attribute (*time*). Listing 2 shows the result of our transformation in both *Clock* and *TimeZone* classes.

Listing 1. Instantiation dependency in Java code.

```
public class Clock extends Content{
  public Clock() {
    TimeZone timeZone = new TimeZone();
    String time = timeZone.getTime();}
    ...
}

public class TimeZone {
  public String time;

  public TimeZone() {...}
  public String getTime(){...}
}
```

Listing 2. Creating object interfaces.

```
public class Clock extends Content{
  public Clock() {
    ITimeZone timeZone = new TimeZone();
    String time = timeZone.getTime();
    }
    ...
}

public class TimeZone implements ITimeZone{ ... }

public interface ITimeZone {
  public String setTime();
  public String getTime();
}
```

3.2 Using Component Interfaces through the Factory Pattern

The second step of this transformation is based on the Factory design pattern. Thus the expression in the source code related to the instantiation of a class *B* by a class *A* where these classes are parts of two different components is transformed to a use of a component interface playing the role of an object factory. This interface is defined as provided by the component of class *B*. It contains methods that return objects instantiated from classes of the component of class *B*. Each method of this interface corresponds to an existing class constructor. The methods of this interface are implemented in a factory class which is added to the classes of the component of the class *B*.

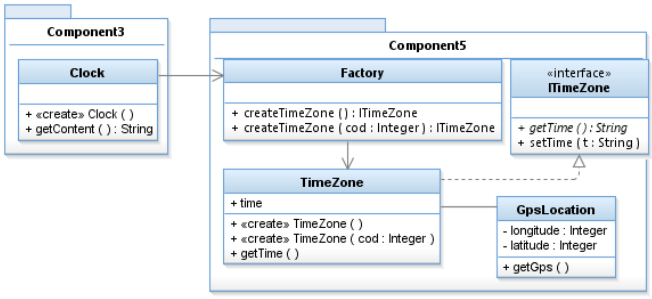

Figure 3. Transforming class instantiation based on the factory pattern.

In Figure 3, we create a provided factory interface whose methods are implemented in the factory class. It has a method *createTimeZone()* that returns a new *ITimeZone()* instance. The *Clock* class invokes this method instead of creating the instance. It does not expose the class that implements the interface, but exposes only the interface. Thereby, client code does not know the internal structure of *Component5*. Listing 3 shows how *Clock* class gets a new instance of type *ITimeZone* using the *Factory* class.

Listing 3. Transforming class instantiation based on the factory pattern in OSGi code.

```
public class Clock extends Content{
  public Clock() {
    ITimeZone timeZone = Factory.createTimeZone();
    }
  ...
}

public class Factory{
  public static ITimeZone createTimeZone() {
    ITimeZone timeZone = new TimeZone();
    return timeZone;
    }
  public static ITimeZone createTimeZone(int cod) {...}
  ... // other provided factory methods
}
```

4. Inheritance Transformation

Inheritance links between classes belonging to different components need to be transformed. Our solution to transform these inheritance dependencies is based on the delegation pattern [18]. In the case of object-oriented code, delegation pattern related to two objects *A* and *B* corresponds to an explicit transfer (forward) for all method invocations received by *A* (called delegator) to *B* (called delegatee) trough methods of *B*. All internal method invocations in methods of *B* related to this transfer must be transfered to delegator. This avoids the problem of the lost of the initial receiver [8, 10].

4.1 Replacing Inheritance by Delegation

Our solution to transform the inheritance link consists in implementing the delegation pattern, but at component level (see Figure 4). Thus, inheritance link between two classes *A* and *B* (*A* subclass of *B*) which belong to two different components is transformed as follows. From the one hand, all methods invoked on the class *A* are transferred (delegated) to the component of *B* (considered as the delegatee) through a required interface. The required interface is implemented by the component of *A* (considered as the delegator) which is connected to a provided interface defined by the delegatee component. The provided interface defines all methods of the superclass *B*. From the other hand, all internal method invocations in the superclass *B* must be transferred to the delegator component through a required interface. This interface is implemented by the delegatee component and connected to a provided interface defined by the delegator component. This interface defines all methods of the subclass *A*.

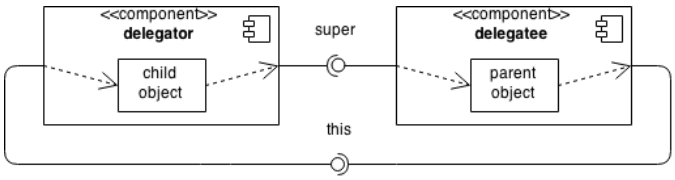

Figure 4. Implementation of the delegation pattern at level of component.

At class level, the transfer of method invocations between delegator and delegatee components and vice versa is realized by creating an instance of superclass *B* in the subclass *A* and by invoking the concerned method in this instance. This instance is created each time the class *A* is instantiated. Attributes of the instance of the class *B* are initialized using values given in the constructor of class *A*. The transfer of a method invocation from the delegatee to the delegator is realized in the same way by invoking this method on the instance of *A*. The reference of the instance of *A* is communicated to the instance of *B* as an additional parameter in each method invocation transferred from delegator to delegatee.

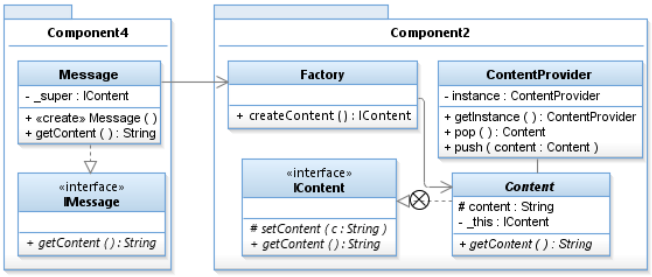

Figure 5. Replacing inheritance by delegation.

Figure 5 describes the transformation of inheritance between *Message* subclass and *Content* superclass. A new interface *IContent* is created for superclass *Content* into delegatee component (Component2). Then, a variable is added to assign the initial receiver of type *IMessage* (*this*). Finally, the factory design pattern is applied to provide *IContent* object interface. On the other side, we create a new interface *IMessage* for subclass *Message*. Then a new instance of the superclass object interface *IContent* is composed to delegate incoming method invocations.

Listing 4 describes the result of transforming inheritance to delegation. As we mentioned before, the transformation concludes into three steps: (i) create new interfaces *IMessage* and *IContent*; (ii) *Message* is composed of an instance of type *IContent* as *super* interface; (iii) *IContent* is composed of an object interface of type *Message* as *this* interface.

Listing 4. Replacing inheritance by delegation.

```
public class Message implements IMessage{
  IContent _super = new Content(this);

  public void getContent(){
    _super.getContent();
  }
  ...
}

public class Content implements IContent{
  private IContent _this;

  public Content(IContent initReceiver) {
    _this = initReceiver;
  }
   public void getContent(){...}
  ...
}
```

Our solution transforms inheritance dependency but produces another dependency which is instantiation, where a subclass creates an instance of its superclass. So we apply here the solution proposed in the previous section (cf. Section 3).

4.2 Handling Subtyping

This section proposes a solution for the problem of breaking the supertype chain. In particular, a variable of superclass type can be assigned a reference to an instance of subclass type (polymorphic assignment), but the necessary assignment compatibility (subtyping) is removed by replacing inheritance with delegation. Another case occurs when a casting to superclass or a type test (*instanceof* in Java) exists in the program. For example, a variable (*content*) in class *ContentProvider* is typed with *Content*. It can be assigned an instance of *Message* or *Clock*. However, after transformation, this variable can not be assigned *Message* nor *Clock* instances.

Our solution suggests to use interface inheritance, which is the most common way to form subtypes between components [16]. We introduce subtyping by adding inheritance between component interfaces providing methods of the subclass and its the component interface providing methods of the superclass. In the example of Figure 5, *IMessage* interface must inherit *IContent* interface. In the same way, *IClock* interface inherits from *IContent* interface. Therefore, a type of *IContent* can be assigned an instance of both types *IMessage* and *IClock*. Moreover, fields defined in *IContent* are now available in both *IMessage* and *IClock* by setter and getter methods (e.g. *setContent(c : String)* in class *Message*).

4.3 Dealing with Abstract Superclasses

As we explained before, a delegator is composed of an instance of a delegatee to delegate method invocations. However, what if the superclass is abstract? An abstract class cannot be instantiated, so no delegatee can be created.

Our solution is based on the proxy pattern [18]. We use a third class as a proxy that breaks the inheritance between the subclass and its superclass when the latter is abstract. Thus the subclass inherits from this proxy, the proxy class inherits the abstract superclass. The proxy class defines the same methods with the same signatures of the abstract superclass. These methods are considered

as proxy methods. Each of these methods delegates the received message to the abstract class when this class provides a concrete implementation of this method. In the case of an abstract method on the superclass, the corresponding method on proxy reforwards the message to subclass.

Actually, in our example (see Figure 1) *Content* is an abstract class and has an abstract method *getContent()*. *Factory* interface can not return an instance of this class. So we need to apply proxy pattern before applying delegation pattern. In Figure 6, we create a *Proxy* class that inherits from *Content* abstract class and implements *IContent* interface. Then *Factory* class provides an object interface of type *IContent* to *Message* which is placed in *Component4*. This enabled us to decouple the inheritance dependency between the abstract superclass *Content* and its subclass *Message* that is placed in a different component.

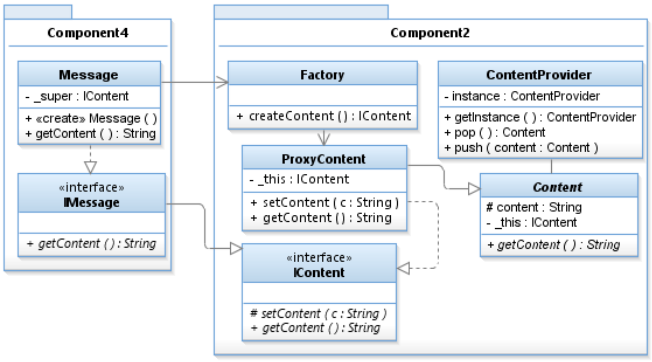

Figure 6. Handling abstract superclass based on proxy classes.

Listing 5 shows the result of transforming inheritance that have abstract superclass to proxy pattern. *ProxyContent* class was created to break the inheritance between *Message* and *Content*. The new class implemented all abstract methods inherited from it's superclass (*Content*). The calling of these methods are backward delegated to it's caller using our initial receiver variable *_this*. The non-abstract methods is called usually under the inheritance relationship between *ProxyContent* and *Content* and the composition between *Message* and *ProxyContent*. Consequently, the inheritance relationship that has un abstract superclass is transformed by using proxy pattern.

Listing 5. Handling abstract superclass based on proxy classes.

```
public class ProxyContent extends Content implements IContent{

  IContent _this = new Content(this);

  public ProxyContent(IContent initReceiver) {
    _this = initReceiver;
  }

  public void getContent(){
    _super.getContent();
  }

  ...
}

public class Message implements IMessage{
  private IContent _super;

  public Message() {...}
  public void getContent(){...}
}
```

5. Experimental Evaluation

This section reports on some experiments we conducted to evaluate our approach.

5.1 Experiment Design and Planning

Research Questions

RQ1: *Does the transformation result avoid component encapsulation violation?*
Our approach transforms the OO code to avoid component encapsulation violation by making the dependencies between components explicit. The transformation aims at creating and using component interfaces to achieve component encapsulation. Thus, the aim of this research question is to measure the contribution of our approach to transform OO dependencies to CB noes.

RQ2: *To which extent does the automatic transformation reduce the developer's effort?*
The aim of this research question is to measure the saved efforts of developers when using our automatic transformation approach instead of using manually one.

Evaluation Methods The research question **RQ1**
To answer **RQ1**, we need to evaluate how much the OO dependencies are transformed to interface-based ones. This can be measured by the ration of number of interface-based dependencies to the total number of dependencies between components after transformation. The *Abstractness* metric proposed by Martin [17] for evaluating OO software fulfill this goal. This metric represents the ratio of abstract types (interfaces and abstract classes) in a package to the total number of types in that package. The range for this metric is 0.0 indicating a completely concrete package to 1.0 indicating a completely abstract package. In the context of CB software, This metric has been adapted by [25] to measure the quality of a component's interfaces, where the classes that represent the component's provided interfaces are grouped in a package to compute *Abstractness*. Therefor, we used this metric in the same way as [25] to answer **RQ1**. Based on this metric, a well designed component is supposed to provide only interfaces. Therefore, a component with high *Abstractness* means a high component encapsulation (i.e., it avoids the component encapsulation violation).

To answer **RQ2**, we compared the estimated efforts expressed by time spent by developers through manual transformation to the time made by our automatic transformation. We compute the time for each type of transformation, instance handling and both inheritance with and without an abstract superclass transformations.

Data Collection We have conducted our transformation approach on 9 Java projects in order to validate our approach. The projects have been selected from *Qualitas* Corpus [26]. In order to guide project selection in such a way that the coverage of a sample is maximized, we have followed the following selection criteria:

i **project size**: We have selected projects with different sizes.

ii **Domain**: We have selected projects from different domains to avoid the influence on experimental results of characteristics associated to a specific domain.

iii **Development team**: We have selected projects that have been developed by different teams to avoid the characteristics related to programming team habits to influence on experimental results.

Table 1 provides some descriptive measures about these projects. It provides each project name and it's version. We can observe the differences of these projects through it's sizes and domains. We can infer the deferences of development teams by the deferences of

owned company, where each project developed by different companies.

Table 1. Data collection.

Application	Version	Domain	No of classes	Code Size (KLOC)
Tomcat	7.0.71	middleware	1359	196
Ant	1.9.4	parsers/generators/make	1233	135
Checkstyle	6.5.0	IDE	897	63
Freecol	0.11.3	games	669	113
JFreeChart	1.0.19	tool	629	98
HyperSQL	2.3.2	database	539	168
Colt	1.2.0	SDK	288	35
Log4j	1.2.17	testing	220	21
Galleon	0.0.0-b7	3D/graphics/media	137	26

Protocol For architecture recovery, we used our method called *ROMANTIC* [4, 9] which enables to identify a component-based architecture from an existing Java application[1]. We applied *ROMANTIC* on our selected Java projects (Table 1). *ROMANTIC* clusters each project as a set of disjoint clusters (components).

For code transformation, we developed a tool (an Eclipse plugin) that automatically transforms the result of architecture recovery (clusters of java code) to OSGi components (bundles). Our tool parses the source code using the Abstract Syntactic Tree (AST) generated by Eclipse's JDT. After that, it makes transformations on this AST for the instantiation, method invocation and inheritance dependencies between components.

We conducted two experiments to answer our two research questions respectively. In the first experiment we compare the *Abstractness* between the recovered components before transformation (i.e., OSGi components with direct OO dependencies) and the same components after transformation (i.e., OSGi components with dependencies though provided and required interfaces)[2]. We voluntarily limited the number of types to those which are provided and required by components that depend each from the other. In the second experiment, we compare developers' efforts (time) between manual transformation with automatic transformation. It is obvious that the automatic transformation provides better results (small values for the transformation time), but what we would like to show here is the estimated average time to perform transformations manually on a whole Java project. The time to do it automatically is measured to estimate the multiplying factor between the two transformation processes.

In this evaluation we firstly computed *Abstractness* for components (clusters) that resulted from architecture recovery step. To compute *Abstractness* for a component ***C***, we start by searching for classes of ***C*** that are used by classes of other components (provided types). Then we compute the ratio of the number of interfaces and abstract classes that belong to provided types to the total number of provided types (see Equation 1). After that, we used our transformation tool to transform Java clusters into OSGi components (bundles) with provided and required interfaces. Then we recompute *Abstractness* as described in equation 1 for OSGi components. finally, we compare the *Abstractness* values to answer our research question **RQ1**.

[1] It is worth recalling that the experiment can be conducted using another recovery method. Only the output (in which we "trust"), which takes the form of a set of class clusters, is important for the remaining steps.

[2] OSGi model allows creating components with either direct OO dependencies between classes composing these components or through interface-based connections [13]

$$Abstractness(C) = Na/Np \quad (1)$$

where Na is the number of interfaces and abstract classes that belong to provided types of component C,
and Np is the number of provided types of component C.

In the second experiment, we performed the transformation manually. To this end, we selected from our data collection three projects that have different sizes. We chose *Log4j* as a small project, *JFreeChart* as a medium project and *Tomcat* as a large project. We selected just three representative projects from the nine composing our data collection (cf. Table 2). We do this selection to adapt the manual experimentation to the available resources (persons and time). We invited 15 developers to transform Java source code. To make sure that we obtain a relatively fair valuation, we split this group of persons into three groups, five persons for each. Table 2 provides descriptive information about these persons. Before starting the experimentation, we checked that these persons have well understood the steps presented in our approach to applied for transforming OO to CB code. In each group, we provided each person the source code. In addition, we gave them the information about three components with different sizes (small, medium and large) in each project (9 components in total). Then we asked them to randomly select three classes from each project and from different components. The selected classes must be transformed by satisfying our condition: selected instantiation and/or inheritance dependencies that must be transformed must be related to classes belonging to other component(s). We measured the time for three types of transformation: instantiation, inheritance and inheritance with an abstract superclass.

Table 2. Information about persons involved in the experiment.

Persons	# persons	Group	Experience in Java
Ph.D Students	5	1	3-6 years
Developers	5	2	4-6 years
M.S. Students	5	3	2-4 years

5.2 Results

5.2.1 Architecture Recovery Results

Table 3 provides some descriptive statistics about architecture recovery results (on the whole data collection, and not on the three projects selected for manual transformation). It displays the number of components recovered from each project (16-129). Moreover, it shows the nature of the components; average number of classes per component (8.5-20.7) and how strongly components are related te each other using *Afferent Couplings* (10.33-24.85). *Afferent Couplings* (also known as Outgoing Dependencies) is a metric that measures the number of types outside a component that depend on types inside the component. According to the obtained results, we observed that the number of components is almost directly proportional to the project size except in case of *Tomcat* and *Freecol* projects.

5.2.2 Code Transformation Results

Table 4 provides descriptive statistics about transformation types for our data collection. It describes the number of transformations that must be performed according to our approach for each project. The results show that instantiation is the most transformation type with an average of 69.9% from all transformation types in all projects. Then transforming inheritance that have abstract superclass with an average of 16.3% (except for HyperSQL, Clot and Galleon, where transforming inheritance is slightly bigger than

Table 3. Architecture recovery results.

Application	# components	Avg. number of classes per component	AVG. Efferent coupling per component
Tomcat	125	10.8	13.87
Ant	129	9.5	10.33
Checkstyle	63	14.5	13.98
Freecol	36	18.5	22.47
JFreeChart	40	15.7	15.63
HyperSQL	26	20.7	24.85
Colt	23	12.5	10.70
Log4j	23	9.5	10.74
Galleon	16	8.5	15.56

transforming abstract superclass). Finally, transforming inheritance with an average of 13.8%.

Table 4. Statistics of transformation types.

Application	# instantiation transformations	# inheritance transformations	# inheritance with abstract class transformations
Tomcat	350	49	79
Ant	364	50	54
Checkstyle	249	37	41
Freecol	164	28	34
JFreeChart	116	22	38
HyperSQL	99	20	19
Colt	70	17	13
Log4j	62	16	28
Galleon	56	18	14

According to the obtained results in Table 4, we observed that the number of transformations is (in most cases) directly proportional to the number of components. As can be seen in Figure 7, the relationships between the transformation types and the number of components for our data collection. However, a small exception of that relationship occurred in case of *Tomcat* and *Freecol* projects.

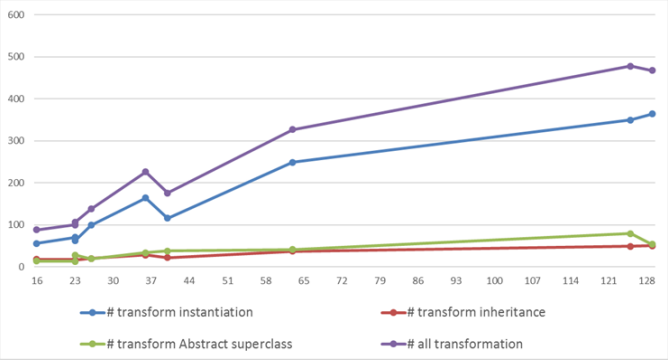

Figure 7. Relation between number of transformations with number of components.

Abstractness Results Table 5 shows the difference of *Abstractness* values between the components before and after transformation. Moreover, it gives the multiplying factor between the two Abstractness measures. The improvement factor ranges from 2.93 for *Tomcat*, which basically has a good design from the abstractness point of view, to 7.33 for *HyperSQL*. The improvement of the level of abstractness depends thus on the analysed software system. On average in the considered data collection, our approach improved *Abstractness* by 4.33 times (answer to *RQ1*).

We can observe that the *Abstractness* for all applications is significantly improved. This improvement lies behind our transformation approach, where we transform OO direct dependencies into component interface dependencies. Another observation is that the values of *Abstractness* does not reach the optimal value (1.0). That because we have not yet transformed all OO direct dependencies between components to interface-based ones (e.g. exception and event handling, etc.). For example, we improve the *Abstractness* of *Tomcat* at 82%. The remaining unimproved value, which is 18%, is caused by the others dependencies that must be transformed but were not presented in this paper.

Table 5. Improvement of abstractness after transformation.

Application	Abstractness before transformation	Abstractness after transformation	Improvement factor
Tomcat	0.28	0.82	2.93
Ant	0.18	0.83	4.61
Checkstyle	0.12	0.84	7.00
Freecol	0.17	0.81	4.76
JFreeChart	0.22	0.81	3.68
HyperSQL	0.12	0.88	7.33
Colt	0.26	0.83	3.19
Log4j	0.19	0.87	4.58
Galleon	0.21	0.88	4.19
AVG.	0.194	0.841	4.33

Manual vs. Automatic Transformation Results The results of the second experiment are presented in Table 6. It shows the results of manual transformation for the three selected projects. The first two columns present the selected projects (*Tomcat*, *JFreeChart* and *Log4j*) and the transformation types. The number of needed transformations that must be achieved according our approach are presented in the third column. Then the fourth and the fifth columns show the number of the transformations that were performed manually. The values of the fourth column shows all these transformation while the values of the fifth column present only the number of unique transformations (i.e. transformations on dependencies done only buy one developer). For instance, the number of manually transformed instantiation in *Tomcat* is 35. But the number of different manual transformation is 20. That mean 15 classes from 35 were repeatedly transformed by different persons. For example, class *WebappLoader* that belong to *Tomcat* was transformed three times. We note that the ratio of the number of realized manual different transformations to the number of all transformations automatically achieved is about 22%. This ratio constitutes a good base to compare results of manual and automatic transformations.

The sixth column shows the number of wrong manual transformations. A wrong transformation corresponds to a case where this transformation is not properly done. For example, it is the case when a person transformed OO inheritance between two classes which belong to the same component. As we have noted before, the persons manually transformed source code have highly understood our approach before we did the experiments. Therefor, the wrong transformations are not the result of misunderstanding of our approach. The ratio of the number of wrong transformations to the number of total number manual transformations is about 18%. This means that approximatively one fifth manual transformations was wrong.

The transformation time is presented in the rest of the table (last four columns). The first three ones represent statistics about the manual transformation time in seconds for each selected project presented following the type of transformation. We can observe that the mean time for each type of transformation realized in different selected projects (*AVG. time column*) is approximatively the same. For example, the mean time taken to transform inheritance is ranged from 1053 to 1106, where the difference is just 53 seconds which is a small value compared with the transformation time (i.e. 1053 or 1106). The *Min/Max time* shows the minimum and the maximum time for the corresponding types of manual transforma-

tions which indicates to the variation of the transformations time. Moreover, a standard deviation is provided in the column *STD time* to better illustrate the amount of variation or dispersion of the manual transformation time. The little standard deviation values compared to the mean reflect a small amount of variation of the transformations time values.

The mean of the estimated time in hours to manually realize a type of transformation for each selected project is presented in the last column (*AVG. estimated time*). We compute these values by multiplying the number of the needed transformations for each project by the mean values of manually transformation time. The conclusion related to this column is that the manual transformation is not an easy task. For example, to completely transform *Tomcat*, *JFreeChart* and *Log4j* manually, we need 79.48, 31.82 and 18.86 hours respectively. For example, in the case of *Tomcat*, this corresponds to more than two weeks of work compared to French work lows. In addition, we did not compute the cost caused by wrong transformations that have an error percentage about 18%.

On the contrary, our tool transforms *Tomcat* for example in a few minutes (about 6 minutes) without any wrong transformation. The ratio between the manually and the automatically transformation times for *Tomcat* is 795. Thus, we can answer **RQ2** that our automatic approach effectively reduces the developer's efforts especially on large projects.

5.2.3 Threat to Validity

Internal threats: one internal threat needs to be considered when interpreting our experimentation results. This is related to the used architecture recovery approach in our experiment. For example, we observed that the number of the needed transformation depends on the number of the recovered components. The number of components depends on the used architecture recovery approach. Consequently, the improvement ratio of *Abstractness* and the saved transformation efforts obtained by our approach can be affected following the architecture recovery approach that are used (*ROMANTIC* approach [9]). For example, *Tomcat* have 350 instantiation dependencies that must be transformed (see Table 6). As the architecture recovery approach is responsible for identifying components (i.e., find clusters of classes), the number of dependencies between these components differs depending on the used recovery approach. Thus, the 350 instantiation dependencies that must be transformed in *Tomcat* may be less or more depending on this architecture recovery used approach.

External threats: External validity refers to generalizability of the results. In this study, we have two threats to external validity to generalize our results. The first one is related to our data collection and the second one is related to the types of persons whom applied our approach manually.

Data Collection We performed our experiments on nine different-sizes, several-domains, well-known and open-source Java projects. Moreover, the projects are selected from different development teams. Because the variety of our data collection, we can say that our results can be generalized to involve most Java projects.

Types of Persons We experimented our approach manually on three groups, five person in each group. The groups are PhD students, Java developers and master students (MS). By reference to Table 2, all these groups have an experience in Java development ranging from 2 to 6 years. Additionally, the experiments were applied on the three groups under the same conditions. In this threat we need to validate that the time consumed by manually transformation does not affected by the persons types if they know Java development. In addition, we need to validate that the errors cased by manually transformation does not affected by the persons types also. Figure 8 shows the time consumed by each group for each transformation type. We can see that the transformation times are closed to each other for the three groups. For transforming instantiation, the manual transformation time ranged between 315 to 397 seconds. For transforming inheritance, the manual transformation time ranged between 1073 to 1106. For transforming inheritance that has abstract superclass, the manual transformation time ranged between 1231 to 1258. Consequently, the differences of time consumed by the three groups is negligible. Thus we can emphasize that the time consumed to manual transformation is independent from person type.

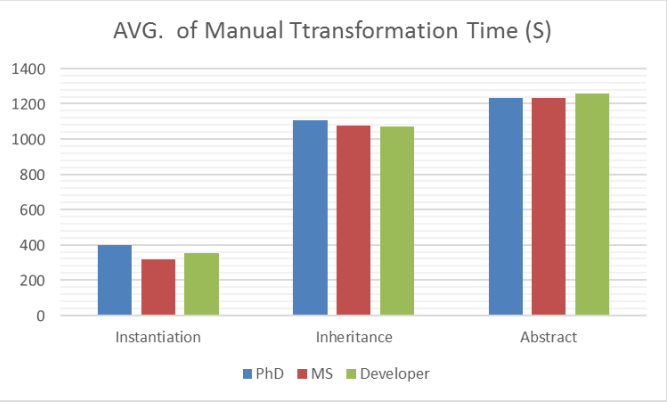

Figure 8. The mean of manual transformation time for each group.

Table 9 shows the error percentage caused by manual transformation for each group. We can observe that the results is almost the same. The percentages of error transformation were 15%, 16% and 17% for PhD students, developers and MS respectively. Thus, we can emphasize that the error caused by manual transformation is independent from person type.

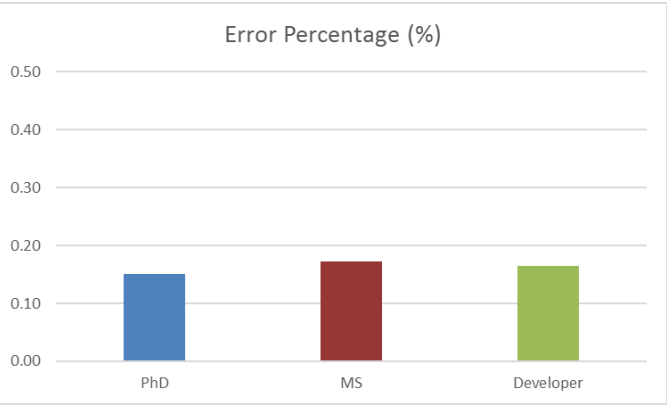

Figure 9. The error percentage of manual transformation for each group.

6. Related Work

Migrating OO applications to CB ones has two types of related works. The first relates to CB architecture recovery, and the second relates to code transformation. Many works have been proposed

Table 6. Estimated time for manual transformation.

Application	Transformation type	# needed transformation	# manual transformation	# different manual transformation	# wrong transformation	AVG. time (s)	Min/Max time (s)	STD time (s)	AVG. estimated time (h)
Tomcat	Instansiation	350	35	20	2	367	230/1008	126	35.68
	Inheritance	49	3	3	6	1106	928/1380	241	15.05
	Abstract superclass	79	16	9	2	1310	1019/1803	195	28.75
JFreeChart	Instansiation	116	37	16	0	395	192/901	169	12.73
	Inheritance	22	16	15	2	1053	862/1301	148	6.44
	Abstract superclass	38	11	3	2	1198	1012/1405	135	12.65
Log4j	Instansiation	62	34	17	0	377	248/869	158	6.49
	Inheritance	16	9	6	11	1054	892/1401	188	4.68
	Abstract superclass	28	6	6	5	989	982/1106	64	7.69

for recovering CB architectures from OO legacy code. A survey on these works is presented in [7] and [3]. However, only few works have proposed a transformation from OO code to CB one.

The approach proposed by [1] applies in transforming Java applications to OSGi. The approach uses OO concepts to implement components. They use the Facade design pattern to implement component required interfaces and the Adapter design pattern to implement component provided interfaces. But in contrast to this work, in our approach we deal with inheritance. In addition, in the transformation of instantiations, we take into consideration argument passing (by creating customized Factory methods), while in their work they do not deal with this aspect. Besides, in their approach, for a single connector between components, they create an adapter class, a facade class and a provided interface, with many duplicated code. In our approach, only a factory class and its provided interface are created.

Another method for transforming Java applications into the JavaBeans framework is proposed in [19]. They developed an automatic refactoring method that can identify components from OO programs using a graph which models class relations. After that, their method modifies the surrounding parts of the extracted components in the original programs. The modification solves the dependencies appeared into class relation graphs, using the Facade interface and other code refactoring. This approach did not treat all dependencies in an OO application. They solve a subset of these dependencies (e.g. instantiation and method invocation), while in our work we deal, in addition, with other dependencies that exist in OO applications, such as inheritance.

In the context of OO software engineering, many works have been proposed for refactoring instantiation and inheritance. For refactoring instantiation, Friedrich et al. [20] developed a new refactoring approach for the extraction of interfaces in order to decouple classes. The extracted interface infers from the type of variable declarations (interface-as-type) and automatically inserts it into the code. The refactoring approach is applied on Java that is available as an Eclipse plugin. Tip et al. [21] proposed an approach that extracts interfaces for replacing the access to a class via a newly created interface. The approach uses type constraints to verify the preconditions as well as to determine the allowable modifications on source code. The approach is implemented in the standard distribution of Eclipse. These two approaches aim to decouple classes buy using interface-as-type instead of class-as-type. However, they did not have preconditions or scenarios (on our approach is the direct dependencies between components) to apply the refactoring on all project automatically. The developer decides which types of variable declarations should be refactored.

For refactoring inheritance, Hauck [22] proposed an approach to replace inheritance by a special kind of aggregation. The approach introduces two kinds of fields, super and self fields. The super filed is placed in the subclass and points to an instance of the superclass. The self field is placed in the superclass and points to the instance of the subclass. Inherited methods are called via delegation and reverse delegation using super and self fields, respectively. However, the approach did not analyses all inheritance cases. For example it did not provide a solution for abstract super class. Genssler et al. [23] presented a refactoring approach that transforms inheritance to delegation (or, as they call it, composition). The approach introduces certain design patterns (namely Bridge, Strategy, and State) to replace inheritance with composition. INTELLIJ IDEA [24] introduces a refactoring tool that replaces inheritance with delegation for Java. The tool performs a program analysis and refactors inheritance to delegation using inner classes. However, the refactoring introduces delegation by adding an inner class into a subclass. The inner class delegates inherited methods to delegatee. It does not introduce reverse delegation therefore it prevents overridden methods in a subclass from being called.

7. Conclusion

In this paper, we proposed an approach to automatically transform object-oriented applications to component-based ones. We targeted the transformation of applications which are built using object-oriented languages into applications built with an object-based component model. We focus on the transformation of source code in order to produce decoupled components that are compliant with the architecture recovered in a previous step. We proposed a solution for dealing with instantiation, method invocation and inheritance dependencies.

The experimentation results shows that our approach improves *Abstractness* and as consequence reduce the violation of component encapsulation. Moreover, it effectively reduces the developer's transformation efforts especially on large projects.

As a future work, we plan to deal with others object-oriented mechanism's transformations, like exception and event handling transformation. The ultimate goal is to produce a code which will be completely based on connections of component interfaces.

References

[1] S. Allier, S. Sadou, H. Sahraoui, and R. Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 214–223, June 2011.

[2] Simon Allier, HouariA. Sahraoui, Salah Sadou, and Stphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 216–231. Springer Berlin Heidelberg, 2010.

[3] Dominik Birkmeier and Sven Overhage. On component identification approaches classification, state of the art, and comparison. In GraceA. Lewis, Iman Poernomo, and Christine Hofmeister, editors, *Component-Based Software Engineering*, volume 5582 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2009.

[4] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 285–288, Feb 2008.

[5] Eleni Constantinou, Athanasios Naskos, George Kakarontzas, and Ioannis Stamelos. Extracting reusable components: A semi-automated approach for complex structures. *Information Processing Letters*, 115(3):414 – 417, 2015.

[6] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, Sept 2011.

[7] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, July 2009.

[8] W. Weck et al. Do we need inheritance?, 1996.

[9] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 181–190, Aug 2012.

[10] Hannes Kegel and Friedrich Steimann. Systematically refactoring inheritance to delegation in java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM.

[11] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 88–95, Aug 2005.

[12] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, Oct 2007.

[13] Osgi Service Platform. The osgi alliance, release 6, 2015.

[14] A.-D. Seriai and S. Chardigny. A genetic approach for software architecture recovery from object-oriented code. In *proc. of SEKE*, 2011.

[15] Petr Spacek, Christophe Dony, Chouki Tibermacine, and Luc Fabresse. An inheritance system for structural & behavioral reuse in component-based software programming. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 60–69, New York, NY, USA, 2012. ACM.

[16] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[17] Martin, Robert Cecil. *Agile software development: principles, patterns, and practices*. Upper Saddle River, NJ: Pearson Education. ISBN 9780135974445. 1st edition, 2002.

[18] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[19] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56(12):99 – 116, 2005. New Software Composition Concepts.

[20] Steimann, Friedrich and Mayer, Philip and Meisner, Andreas. Decoupling Classes with Inferred Interfaces. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1404–1408, New York, NY, USA, 2006. ACM.

[21] Tip, Frank and Kiezun, Adam and Bäumer, Dirk. Refactoring for Generalization Using Type Constraints. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 13–26, New York, NY, USA, 2003. ACM.

[22] Hauck, Franz J. Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance. *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*, SIGPLAN Not., pages 231–239, New York, NY, USA, 1993. ACM.

[23] T Genssler, B Schulz. Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance. *Transforming inheritance into composition A reengineering pattern*, proc. of 4th EuroPLoP, 1999.

[24] IntelliJ IDE. http://www.jetbrains.com.

[25] Hamza, S. and Sadou, S. and Fleurquin, R. Measuring Qualities for OSGi Component-Based Applications. *Quality Software (QSIC), 2013 13th International Conference on*, pages 25-34, 2013.

[26] Tempero, E. and Anslow, C. and Dietrich, J. and Han, T. and Jing Li and Lumpe, M. and Melton, H. and Noble, J. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. *Software Engineering Conference (APSEC)*, 2010 17th Asia Pacifice, pages 336-345 2010.