Optimization by design patterns and static analysis of web applications for a sharp adaptation of e-business start-ups in the city of Lubumbashi in DR Congo (Nesher)

Optimisation par les patrons de conception et l'analyse statique des applications web pour une adaptation pointue des start-ups de commerce électronique dans la ville de Lubumbashi en RD Congo (Nesher)

FYAMA Blaise
Teacher researcher - Dean of Faculty
Faculty of Computer Sciences
Université Protestante de Lubumbashi(UPL)
Department of Information systems engineering
Democratic Republic of Congo
bfyama@gmail.com

KADIATA Freddy
PhD student
Faculty of Computer Sciences
Université Protestante de Lubumbashi(UPL)
Department of Information systems engineering
Democratic Republic of Congo
freddyilunga94@gmail.com

Date de soumission : 14/06/2021

Date d'acceptation : 19/08/2021

Pour citer cet article :

FYAMA. B & KADIATA. F (2021) "Optimization by design patterns and static analysis of web applications for a sharp adaptation of e-business start-ups in the city of Lubumbashi in DR Congo (Nesher)", Revue Internationale du Chercheur "Volume 2: Numéro 3" pp: 10349 - 1372

Abstract

Due to some limitations of imperative development approaches, in recent years, many companies have introduced object-oriented technology into their software developments. The component paradigm, which advocates the assembly of autonomous and reusable software bricks, is indeed an interesting proposition for reducing development and maintenance costs while increasing the quality of applications.

Whatever development paradigm we find ourselves in, it is essential for architects and developers alike to assess what they produce in order to produce software capable of meeting user needs, which is why we propose in this article article an approach based on the static analysis of the software and the use of some software engineering model such as the design pattern of which we note the notable impact of the latter on the regulation of certain metric parameters to acceptable thresholds and thus optimize the internal and external quality of the software.

Keywords: Software optimization; quality metrics; Chidamber and Kemerer; Model-driven engineering; Design Patterns

Résumé

En raison de certaines limites des approches de développement impératif, ces dernières années, de nombreuses entreprises ont introduit la technologie orientée objet dans leurs développements logiciels. Le paradigme des composants, qui préconise l'assemblage de briques logicielles autonomes et réutilisables, est en effet une proposition intéressante pour réduire les coûts de développement et de maintenance tout en augmentant la qualité des applications.

Quel que soit le paradigme de développement dans lequel nous nous trouvons, il est essentiel pour les architectes comme pour les développeurs d'évaluer ce qu'ils produisent afin de réaliser des logiciels capables de répondre aux besoins des utilisateurs, c'est pourquoi nous proposons dans cet article une approche basée sur l'analyse statique du logiciel et l'utilisation d'un certain modèle de génie logiciel tel que le design pattern dont nous notons l'impact notable de ce dernier sur la régulation de certains paramètres métriques à des seuils acceptables et ainsi optimiser la qualité interne et externe du logiciel.

Mots-clés : Optimisation du logiciel ; métriques de qualité ; Chidamber et Kemerer ; Ingénierie dirigée par les modèles ; Design Patterns.

Introduction

In recent years, many companies have introduced object oriented technology in their software development. The component paradigm, which advocates the assembly of autonomous and reusable software bricks, is indeed an interesting proposition for reducing development and maintenance costs while increasing the quality of applications.

In the object paradigm, as in all the others, architects and developers must be able to assess the quality of what they produce as early as possible, especially throughout the design and coding process. quality is understood as software capable of perfectly meeting customer expectations, all without execution flaw. Thus, software quality is determined as a set of rules and principles to be followed during the development of an application in order to design software that meets these expectations.

Metrics on the code are essential tools, to do this they make it possible, to a certain extent, to predict the “external” quality of a software or an architecture being coded. Various proposals for metrics have been made in the literature.

Unfortunately, none of the proposed metrics has been the subject of a serious study as to their completeness, their cohesion and especially as to their ability to predict the external quality of the developed artefacts.

Worse still, the lack of support for these metrics by the code analysis tools on the market makes their industrial use impossible. As it stands, the quantitative and “a priori” prediction of the quality of their developments is impossible. There is therefore a significant risk of an increase in costs following the late discovery of defects.

In the context of this article, we offer a pragmatic answer to this problem. Starting from the observation that a large part of industrial frameworks are based on object-oriented technology, we have studied the possibility of using some of the "classic" code metrics, not specific to the object-oriented world, to evaluate oriented applications. object. Indeed, these metrics have the advantage of being well defined, known, equipped and above all of having been the subject of numerous empirical validations analyzing the power of prediction for imperative codes or objects.

Among the existing metrics, we have identified a subset of them which, by being interpreted and applying at certain levels of granularity, can potentially give indications on the respect by the developers and the architects of the main principles. software engineering, in particular on coupling and cohesion.

These two principles are indeed at the very origin of the component paradigm in the object-oriented world.

This series of metrics, identified by hand, was then applied to a home-made application in order to ensure, through a study and analysis by software metrology, that it effectively conveyed relevant information to the object-oriented world.

The big problem for developers on the a priori prediction of the quality of their development is a highly risky bet with the consequence of an increase in costs following the late discovery of defects. En state quality control in the object, imperative, component paradigm is difficult to perform.

Faced with this lack of consensus, hindsight and tools for dedicated metrics, what can we do? The objective of this article was to propose a pragmatic answer to this problem to ensure the quality of the software from the conception until its evolution. We first noted that a large part of the current industrial frameworks for application development are based on object-oriented technology. The paragon of this approach is the component model, designed on a Java layer. The question we asked ourselves was then the following: is it possible and relevant to rely on certain existing code metrics, well-known and even more widely equipped to study the quality of components and architectures of object-oriented software? is it possible and relevant to use the model-based approach (design pattern) to produce quality software and solve the problem of software aging? Our research responds positively to this question and even demonstrates that, indeed, certain metrics, not specific to the object, component or imperative world, can prove to be valuable both at the level of the component (for the developer) and at the level of the 'application (for the architect).

Thus, the values obtained from these metrics become valid indicators on the choice or not of a development style, design patterns have remained by far a better choice, we will try to demonstrate how, in the rest of this document.

The metrics by being interpreted and by applying to certain levels of granularity give indications on the respect by the developers and the architects of the main principles of software engineering, in particular on the coupling and the cohesion. These two principles are at the very origin of the component paradigm in the object-oriented. This is why they are used as main optimization criteria by the work on the restructuring of object-oriented applications into component-oriented applications.

We followed the software engineering methodology which includes several transversal methods:

1. The software performance method
2. The reliability engineering method
3. The safety engineering method

All these methods were considered in a quantitative approach to show through a static analysis the internal external performances of the software.

We first had to try to identify a "minimal core" of a few metrics that could potentially serve as "complete systems" of internal measurement. A suite of metrics able to represent all the facets of an object-oriented application: internal view, interface and compositional. For this, it was necessary to extract dozens of existing metrics those capable, for a certain level of granularity, of translating "something" on these 3 facets and on the object-oriented design principles followed by the developer or the architect. This hand-identified series of metrics was then applied to our application (Nesher) in order to ensure through a study of the distributions that it actually conveyed relevant information to the object world. This study,

Finally, the use of a test tool was used, and it is an essential examination because it allowed to a large extent to bring out certain parameters of our application which undoubtedly would not be done easily by hand.

All these steps were carried out to conclude on the "practical" interest of using common and well-equipped metrics to measure the quality of applications in the object-oriented world as early as possible and to propose good design practices to regulate values. obtained from these metrics.

We are not the first or even the only one to speak of software metrology of applications oriented object by metrics to examine the design flaws, other authors long before us have also spoken about it, among these works we have put the hand on the thesis of Stéphanie GAUDAN presented to the National Institute of Applied Sciences of Toulouse for obtaining the title of DOCTOR in 2007.

In her thesis entitled, `` Management of the risks of design faults linked to object-oriented technologies for their use in critical avionics applications ", she identifies the risks of design faults and their risks, then she successively used Bayesian networks and new metrics to estimate the risk level of design faults in object-oriented applications, it ends with an experimentation of these metrics on an avionics system (GAUDAN, 2007).

As far as we are concerned, we do not manage the risks of design faults but rather an analysis by software metrology on a home-made object-oriented application, by the application of

Chidamber and Kemerer metrics and offer good design practices. making it possible to obtain satisfactory values of these metrics and thus significantly increase the quality, both internal and external, of the software.

In the two points we first present the optimization of Nesher by the use of design patterns then we end by giving the results of the static analysis of our software.

1. Result of the static analysis carried out on Nesher

As we said at the very beginning, we have designed a homemade software that we call Nesher, designed in Java SE language and Java FX.

This part will examine the possibility of using certain metrics certain from the procedural and object world in the object oriented paradigm. This possibility is considered in this part according to two criteria of a rather <<syntactic>> nature. At this level, it will not only be a question of determining whether such or such a metric has a relevant <<sense>> in terms of quality or not, but of verifying whether it is calculable and potentially carrying discriminating information and structurally relevant.

The process of identifying candidate metrics relied on double filtering. The purpose of the first filter is to say whether for a given metric a calculation is possible in the object world by fixing, when possible and / or necessary, a potentially relevant level of granularity. Indeed, some metrics offer a level of genericity allowing their applications at different levels of granularity (classes, packages, components). The point is to choose a level of granularity that is potentially useful and revealing in the object world.

1.1. Identification of standard metrics that can be used in the OO

The metrics used must be able to help two types of actors, namely developers (sometimes independently of the application context) and architects (who assemble the different modules and components to design the application). In a first subsection, we will analyze this need and highlight the need to approach measurement from 3 points of view: a) the internal structure of a package or a class, b) its "surface" external (its interface), c) the relationships between the internal components of an application (class or package). The following subsections will then list in a table the metrics of the procedural world and exploitable object that we had retained for each of these three points of view. (Sylvain Chardigny, 2008).

1.2. Choice of granularity of measures and points of view

In the object-oriented paradigm an application is generally structured as follows:

- An application is divided into components that interact through required and provided interfaces;

- A component can be structurally broken down into one or more packages;
- A package may in turn be divided into one or more classes;
- A class is conventionally composed of one or more fields (attributes and / or methods).

Researchers in the field of restructuring rely on metrics of granularity greater than or equal to that of the class. A class is for them a structural unit of the "black box" type. Only the dependencies maintained by the classes to determine the components to build count. (Hamza, 2014). In our case, we have not adopted this approach. Of the dozens of existing metrics, we have selected metrics having not only the class as the minimum level of granularity, but those which also take into account the internal class structure (methods and properties).
Thus, the metrics associated with internal phenomena to the class such as the cohesion of the class (i.e. the coupling between the methods), the number of methods, the number of properties, the cyclomatic complexity, etc. are taken into consideration in our study.

TABLE NO.1: STATIC ANALYSIS OF NESHER GES-PAYS WITH THE METRICS TOOL

No.	Metric	Meaning	Average value	Interpretation
1	SLOC	Number of lines of code	12818	According to COCOMO Nesher is an intermediate level project (W, 1981) This classification could make it possible to estimate the number of developers, the time and their efforts required for such a project.
2	Cyclomatic complexity	Measure the number of linearly possible paths in a function	1,487	This value is the average of the whole project, 2 methods out of 1211 exceed the admissible value of 10, with one 11 and another 12, or around 0.16%. What is acceptable (McCabe, 1976) (MENGAL, 2013).
3	Afferent coupling	number of references to the measured class	8.319	The admissible threshold is 10, the average value obtained proves a good reuse code in Nesher (MENGAL, 2013).
4	Efferent coupling	measure of the number of types that the class <<knows>>	2,574	This low value proves that Nesher respects the principle of single responsibility, only one package out of 47 exceeds the value of 20, which is a

				threshold proposed by the RefactorIT (Larive, 2015)
5	Lack of Cohesion Of Methods (LCOM)	Level of cohesion in a class	0.391	Also directly linked to the principle of single responsibility, Nesher's LCOM demonstrates a strong capacity for code reuse, ease of testability and above all a strong capacity for maintainability. As argue (MENGAL, 2013) (Goodman, 2013) (Smacchia, 2013) (Kemerer, 1994).
6	Specialization Index	Degree of specialization of a class	0.004	This value of 0.004 simply means that our classes do not redefine the methods of the classes they inherit too much, so the depth of inheritance is low and negligible. (Larive, 2015).
7	Instability	Instability of a module	0.452	Varying between 0 and 1, the values approaching 0 attest a strong extensibility of the code, Nesher is found in an appreciable value (Larive, 2015) (MENGAL, 2013).
8	Abstractnesse	Abstraction level of a module compared to other classes present in the code	0.131	Between 0 and 1, the closer it to 0 the better, it shows that most of our classes are concrete (Larive, 2015) (MENGAL, 2013).
9	Distance form main sequence	The balance of the modulus between abstraction and instability	- 0, 417	Our sequence hand axis touches the two points described on the reference model and thus reflects a certain respect for design rules (Martin, 2000) (Smacchia, 2013).
10	depth of inheritence tree (DIT)	distance between a class and the root in its inheritance tree	2.312	In the inheritance tree, our classes do not descend beyond the second level, which indicates a worrying level of inheritance but acceptable according to(Kemerer, 1994)
11	Coupling Between Object Classes (CBO)	Coupling or dependency level in object	3.17	A good value of this metric tells us that it is easy to find a fault and to fix it easily. Nesher this mean CBO value remains below the threshold of 10 (C., 2002).

12	Weighted Methods per Class (WMC)	Sum of cyclomatic complexities of a class	12,083	If each method has a complexity of 1 in the code, WMC only counts the number of methods. The relevance of this metric is empirically justified in the article(Dugerdil, May 2005) by stating that the number of methods and their complexity is a measure of the effort associated with the creation of the class and its maintenance.

SOURCE: AUTHORS

This analysis allowed us to do the static analysis of our software, using tools such as Metrics, STAN, PMD and to see how our software behaved when we apply a metric from the OO world such as quality.

Overall, the result is satisfactory, although for some metrics there were classes and methods that violated the benchmarks. This is how we asked ourselves, what are the good development techniques that architects and developers should adopt in order to produce quality software, extensible and open to possible modifications?

The use of design patterns has therefore been a major contribution in the development of Nesher Ges-Paie to improve its internal and external quality.

Upstream of this static analysis, some of these good development techniques such as design patterns were used, it is they themselves who are at the origin of these good metric values obtained.

This is what we cover in the next section, in order to give a clear idea of how they work and use in different contexts.

2. Optimization of Nesher by designs pattern

2.1. Optimization of the source code by the use of designs patterns

The static analysis carried out on our source code gave us satisfactory values, this is not a coincidence, although in some place our source code violated some OO programming rules, we estimated their impact negligible (... viewpoints considered).

As Lehman clearly states: "A program used in a real world environment must necessarily change otherwise it will gradually become less and less useful in that environment" (Netinbag, 2017). It was therefore essential for us to give a certain tolerance to modifications to our source code while maintaining an acceptable external quality and meeting the quality standard of

software used in industry. The use of design pattern has therefore made it possible to respond to this problem.

To illustrate all this, let's present the different design patterns used in the design of Nesher and their contributions to the internal and external quality of the software, we will show how we have pushed the limits of inheritance and show the inherent polymorphism to inheritance.

2.1.1. Design pattern: history and definition

The origin of Designs patterns dates back to the early 1970s with the work of architect Christopher Alexander. He notices that the design phase in software architecture reveals recurring problems. He then seeks to resolve all of these problems linked to interdependent constraints. For this he established a language of 253 patterns, which cover all aspects of construction (such as how to design a frame) (design-patterns, 2017).

A Design pattern is defined as a solution to a recurring problem in the design of object-oriented applications. A design pattern then describes the proven solution to solve this software architecture problem. As a recurring problem we find, for example, the design of an application where it will be easy to add functionalities to a class without modifying it, without degrading the quality of its code (increasing the complexity in the methods, violating the constraints of single responsibility …), Or in the concept “a re factorization”.

2.1.2. Organization of Design patterns

Design patterns are classified into three categories:

- **Creation**: they allow you to instantiate and configure classes and objects.
- **Structure**: they allow you to organize the classes of an application
- **Behaviour**: they allow objects to be organized so that they collaborate with each other.

2.1.3. Presentation of the Design patterns used

In the lines we present some Design patterns used in the software by first making a description of the problem to which it answers (problematic), then by giving the description of the solution by UML diagrams (solution) and finally by showing the advantages and limits of this solution (consequences) on the code.

2.1.3.1. Pattern Strategy

The “Strategy” pattern mainly seeks to separate an object from its behaviors / algorithms by encapsulating them in separate classes.

This design pattern allows multiple encapsulated and interchangeable algorithms to be defined dynamically (on the fly).

- **Problematic**:

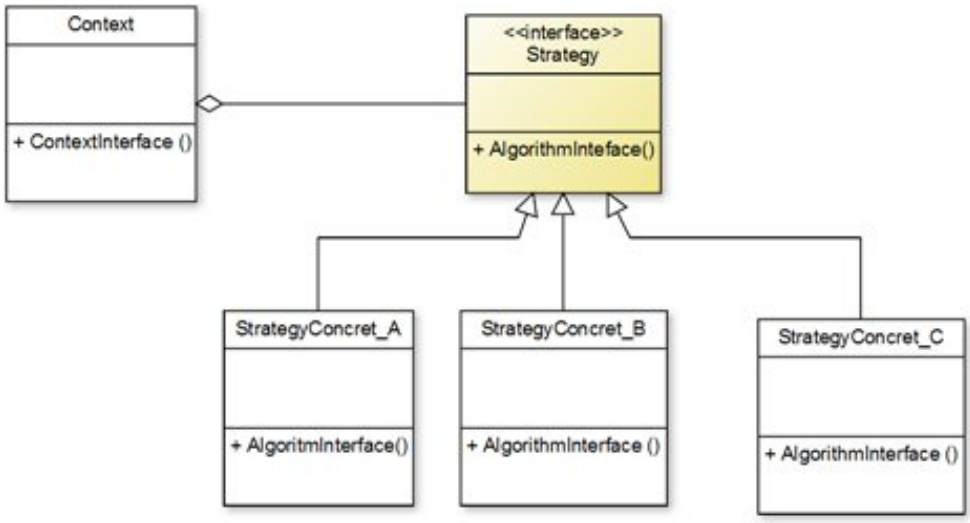
The "Strategy" pattern is used when:

- A problem has several algorithms to solve it.
  - Especially if the algorithms have different efficiencies in different situations.
- An object can have several different behaviors.
  - Also applicable when different classes are only different in behavior.
- Data that the user should not know ends up in an algorithm.

- **Solution:**

The solution proposed by this pattern consists in isolating the part of the code which varies the most and to encapsulate it!

FIGURE 1: UML CLASS DIAGRAM OF THE PATTERN STRATEGY



SOURCE: (WHISMERIL, 2019)

If the algorithms / behaviors are in a class of their own, it is much easier to:

- Finding yourself in the main code
- Remove, add and modify an algorithm / behavior
- Decrease the use of conditional tests
- Eliminate redundancy and cut and paste it
- Increase code reusability and flexibility
- Strong extensibility

- **Consequences**:

In the case of Nesher, this pattern was very useful to us in the sense that it allowed us to set up the MVC pattern that we will present a little below.

Second, the Strategy pattern allowed us to have:

- Satisfactory cyclomatic complexity values for our algorithms because the algorithms are now selected on the fly without going through several control structures such as (if, switch,).
- A good index of software specialization, in the sense that this pattern acts on the value of NORM (number of redefined methods) by significantly reducing it, for Nesher alone 2 methods out of 1211 in the software are redefined, because this metric is also proportional to NORM. Using the Strategy interface, the methods encapsulated in the objects will be used in different contexts without having to be redefined.
- A good factor of software instability because the dependencies between classes also decrease because of the low level of coupling (CBO).
- A good cohesion factor (LCOM) between classes because, as announced, this metric is directly linked to the principle of unique responsibility of a class and also linked to the principle of inversion of dependency.

2.1.3.2. Pattern Observer

The Observer pattern defines a one-to-many object relationship, so that if an object changes state, all those that depend on it are informed and updated automatically.

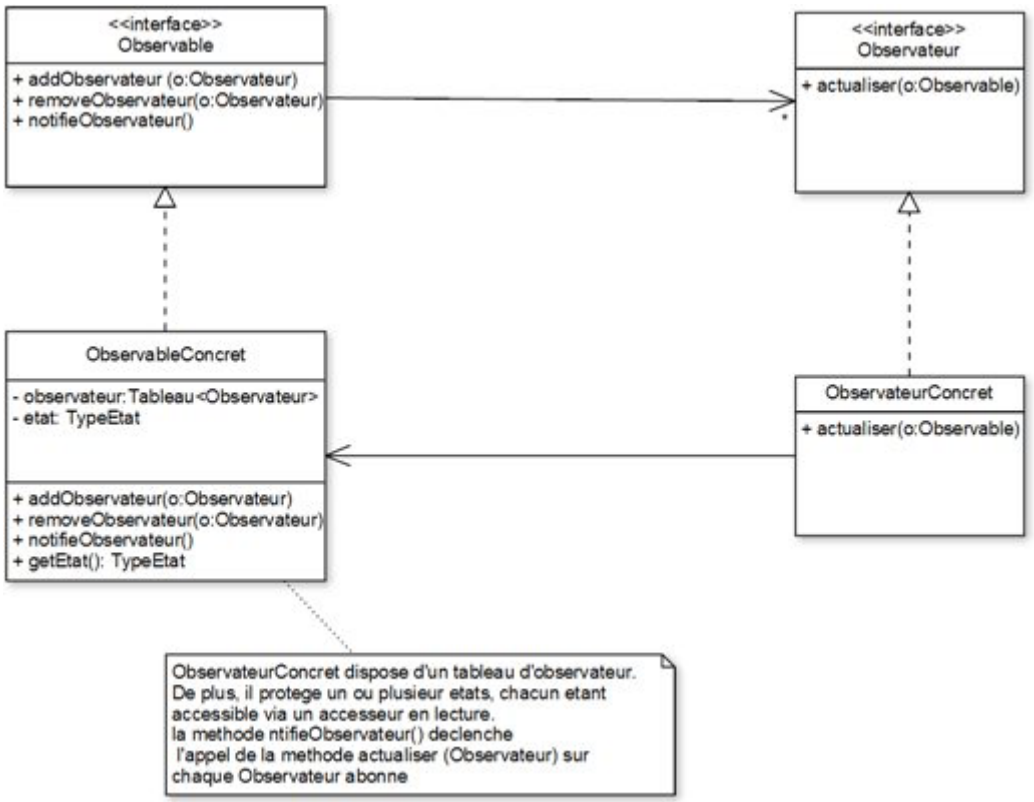
- **Problematic**:

There are classes with attributes whose values change regularly. In addition, a certain number of classes must be kept informed of the evolution of these values. It is not kept informed of changes in these values. Many of the developers like us were faced with this particular problem when developing a business class and the corresponding display classes.

- **Solution:**

The solution proposed by this pattern consists in letting the business class inform its display classes of its changes in values so that they also change state.

FIGURE 2: UML CLASS DIAGRAM OF THE OBSERVER PATTERN



SOURCE:(G, DESIGN-PATTERNS, 2012)

The UML diagram of the Observer pattern defines two interfaces and two classes. The Observer interface will be implemented by all classes that wish to have the role of observer(Whismeril, 2019). This is the case of the ObservateurConcret class which implements the update method (Observable). This method will be called automatically when the state of the observed class changes.

There is also an Observable interface which should be implemented by classes wishing to have observers. The ObservableConcret class implements this interface, which allows it to keep its observers informed. This has as an attribute a state (or several) and an array of observers. The table of observers corresponds to the list of observers wishing to follow the evolution of its values. Indeed, it is not enough for a class to implement the Observer interface to be listening, it must subscribe to an observer via the method addObserver (Observer).

Indeed, the ObservableConcret class has four methods: addObserver (Observer), removeObserver (Observer), notifyObserver () and getState (). The first two allow, respectively, to add observers to the listening of the class and to remove them. Indeed, the Observer pattern makes it possible to dynamically link (make a link during the execution of the program as opposed to statically linking at compilation) observables to observers. The

notifyObserver () method is called when the state undergoes a change in value. This warns all observers of this update (design-patterns, 2017).

It is certainly possible to find in some works a use of this pattern without the two interfaces, but, we point out that our use of these two interfaces makes it possible to weakly couple the observable to its observers. Indeed, a design principle is to link interfaces rather than classes in order to be able to evolve the model easily. Although the use of these two interfaces is not compulsory, it is strongly recommended to avoid the coupling problem.

Not derogating from this rule of coupling objects through interfaces or abstract classes, Nesher applies it in its pure and complete sense.

Some code has been removed from the classes to provide some clarity in reading the pattern setup.

- **Observer interface:**

```
public interface Observe {
        public void update ();
        public void updateRunLater ();
          }
```

- **Observable abstract class:**

```
public abstract class Subject {
        private Observe observers;
        private Traineeship traineeship ;
        private Service <Observer> service ;

        public void attach (Observe o ) {
                        this.observers = o;
        }
        public void attach (Internship traineeship ) {
                 this.traineeship = traineeship;
 }
        public void notifyObservers () {
                ...
        }
        Protected final void notifyThreadRun () {
                this.load ();
        }
        private void load ()
        {... }
}
```

- **Consequences**:
  - The observer pattern makes it possible to weakly and dynamically link an observable to its observers. This solution is weakly coupled which gives it the

possibility of easily evolving with the model. The LCOM of the code is thus regulated.

- It is widely used and is one of the essential patterns for the implementation of the MVC model currently in vogue.
- The last consequence, which of course is no less obvious, is the one advocated by our Green It, which signals a sharp reduction in energy consumption of around 73%, hence a strong reduction in the emission of, significant decrease in CPU load and increased CPU response time increaseCO_2(grenncodelab, 2017).

2.1.3.3. Pattern Singleton

The Singleton ensures that a class has only one instance and provides a global access point to that instance.

- **Problematic**:

Some applications have classes that must be instantiated only once. This is for example the case for a class which implements a driver for a peripheral, a logging system or even a database connection class. Indeed, twice instantiating a class serving as driver for a printer or connection to a database would cause unnecessary overload of the system and inconsistent behavior.

In the case of Nesher, our problem was to create a single connection to the database to avoid overloading the database server, secure and trace accesses and also to avoid excessive consumption of resources. This is how we asked ourselves the question:

How to create a class, used several times within the same application, which can only be instantiated once?

One solution for us was to instantiate the class as soon as the application was launched in a global variable (accessible everywhere in the program). However, this solution infringes the principle of encapsulation and has many drawbacks. Because there is no guarantee that a developer does not instantiate the class a second time instead of using the defined application global variable. In addition, we have to instantiate this global variable as soon as the application is launched and not on demand (which can have a significant impact on the performance of the application). Finally, when we arrive at several hundred global variables, the development becomes unmanageable, especially since Nesher was developed as a team working simultaneously.

- **Solution**:

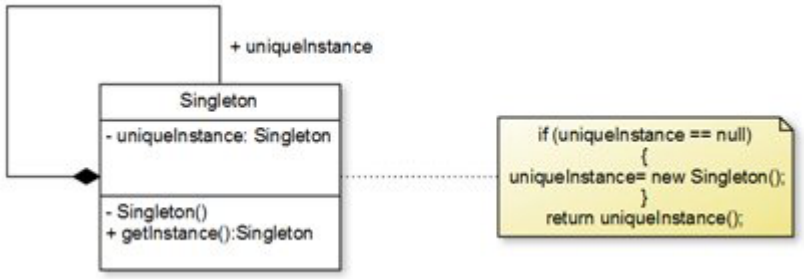
As the use of a super global variable is not possible, we thought of using the Singleton design pattern.

It is to prevent developers from using the constructor of the class to instantiate it. To do this, it suffices to implicitly declare all the constructors of the class in private.

Once this step is completed, it is possible to instantiate this class only from itself, and create a static method which will return an object corresponding to the type of the class. With the advantage that, unlike the constructor, we can control the value that this method will return. The fact that it is static we can call it without having an instance of this class.

We create a static attribute which will make it possible to store the unique instance of the class. Then, in the pseudo constructor (static method) we will test if this attribute is null then we create an instance, if not what the attribute already has an instance of the class. In all cases, the value of the attribute having the unique instance of the class is returned.

FIGURE 3: UML CLASS DIAGRAM OF THE SINGLETON PATTERN



SOURCE: (G, DESIGN-PATTERNS, 2011)

Source code snippet of the use of this pattern in managing a single database connection with Nesher.

```
public class MySQL extends Connection_db
{
        private static DBConfigXMLDAO DbConfigXMLDAO = new DBConfigXMLDAO
(XMLDAOFactory.getfileConfig ());
        private MySQL () { Great(); }
        public static Connection getConnection ()
        {
                for(DBConfig dbConfig : DbConfigXMLDAO.find ()) {
                        host = dbConfig.getDbhote ();
                        dbuser = dbConfig.getDbuser ();
                        dbbase = dbConfig.getDbbase ();
                        dbpass = dbConfig.getDbpass ();
                        load();
                }
                yew( connection == null ) {
```

```
                new MySQL ();
        }
return connection ;
}}
```

For the sake of simplicity, we only present one method of the class, the same one is responsible for acting as a pseudo constructor.

The problem with our way of doing things is that, with multithreading applications, in the sequence of instructions, a first process can execute the function and find that the connection attribute is null. A second process running simultaneously also finds that this attribute is zero. The two processes will therefore each create an instance, we then end up with two attributes of the instance of connection to the DB.

- **Consequences**:
  - To save resources, we only created one connection object and thus only have a single instance of it throughout the program.
  - This pattern is based on a private constructor associated with a method returning the instance created in the class itself.
  - In order to overcome the problem of multithreading (ability of the program to launch several processes simultaneously) and end up with several connections to the database, we just need to use the keyword synchronized in the declaration of our method for retrieving the instance, but this synchronization is only useful once. Instead, you could instantiate the object when the JVM loads the class, before calling it.

2.1.3.4. Pattern DAO (Data Access Object)

This pattern makes it possible to make the link between the data access layer and the business layer of an application (our classes). It allows better control of the changes likely to be made to the data storage system; therefore, by extension, to prepare a migration from one system to another (BDD to XML files, for example…). This is done by separating access to data (BDD) and business objects (POJO: Plain Old Java Object).

- **Problematic**:

  We have serialized data in a database and we want to manipulate it with Java objects.

However, the company is in the process of restructuring and we don't know if our data is going to:

  - Stay where they are;

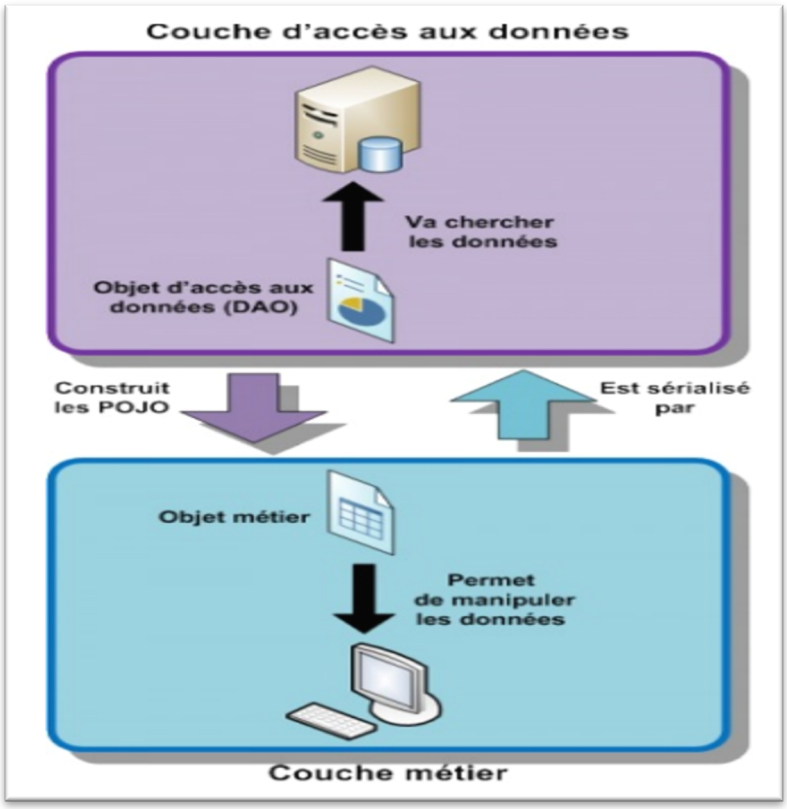
- Migrate to another database;
- Be stored in XML files;
- ...

How can we ensure that we do not have to modify all the uses of our objects? How to achieve a system which could adapt to future modifications of data carriers? How do we keep the objects we are going to use as they are?

- **Solution:**

The solution is to make sure that one type of object is responsible for retrieving the data in the database and that another type of object (often POJOs) is used to handle this data. We were inspired by the diagram presented by openclassroms to represent this solution(openclassrooms, sd).

FIGURE 4: OPERATION OF THE DAO PATTERN



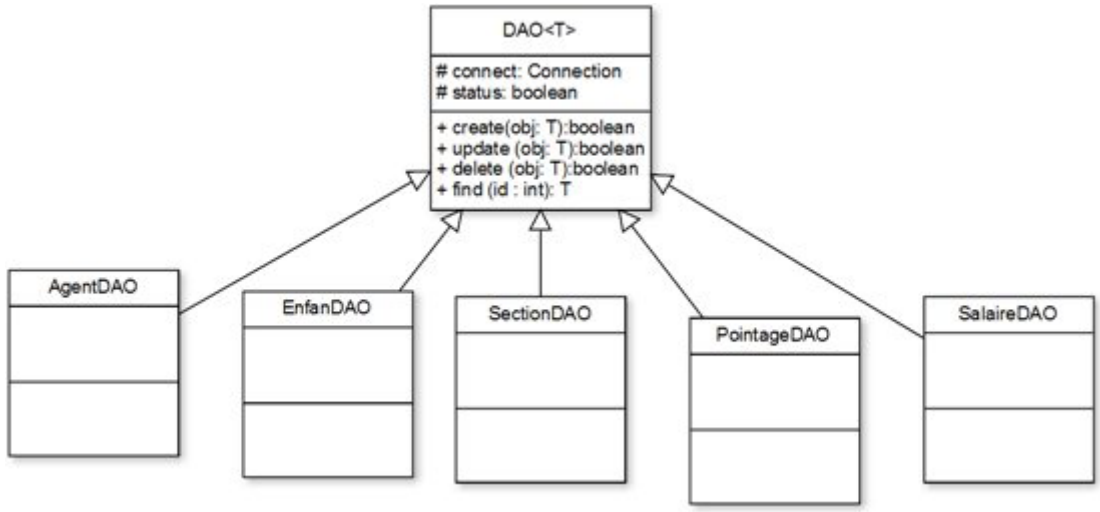
Source openclassroms

These objects which will fetch data from the database must be able to perform searches, insertions, updates and deletions. Therefore, we make the best use of polymorphism ... by

creating an abstract class (or an interface) implementing all the aforementioned methods, also called (CRUD).

In the case of Nesher, we wanted to do a sampling and present the implementation of the DAO pattern in the software, the following figure shows his UML class diagram:

FIGURE 5: DAO CLASSES



Source: AUTHOR

The DAO class is an abstract class using the notion of genericity, to make our DAO objects ask the type of objects to serialize.

Source code of the DAO abstract class collected from eclipse:

```
public abstract class DAO <T>
{
	protected Connection connect ;
	protected boolean status = false ;
	private String message ;

	public String getMessage () { return message; }

	protected void setMessage (String message)
	{
		yew(message ! = null &&!message.isEmpty ())this.message = message ;
	}

	public DAO (Connection connect)
	{
		this.connect = connect ;
	}

	public abstract boolean create (T obj);

	public abstract boolean update (T obj);
```

```
        public abstract boolean delete (T obj);

        public abstract T find (Object obj);

        public abstract ObservableList <T> find ();
    }
```

Source code of the AgentDAO class, we only present the method of creating a new agent to give an overview of the internal process such as show on the class diagram:

```
public class DAO Agent extends DAO <Agent>
{
        public AgentDAO (Connection connect) { Great(connect); }

        public boolean create (Agent agent)
        {
                boolean status = false;

                yew (agent.isNew () &&agent.isValid ())
                {
                        try
                        {
                        PreparedStatementprepare=this.connect.prepareStatement (
                        "INSERT INTO Agent SET" +
                        "matricule =?, numINSS =?, etatCivil?, levelStudy =?, address
=?, email =?, telephone =?, idCategorie =?, name =?, postname =?, firstname =?,
gender =?, birthdate = ?, " "nationality =?, dateE = NOW (), dateM = NOW (), function
= ?, base salary =?, photo =?);
                                prepare.setString (1,agent.getMatricule ());
                                prepare.setString (2,agent.getNumeroINSS ());
                                prepare.setString (3,agent.getCivilState ());
                                prepare.setString (4,agent.getStudyLevel ());
                                prepare.setString (5,agent.getAddress ());
                                prepare.setString (6,agent.getEmail ());
                                prepare.setString (7,agent.getTelephone ());
                                prepare.setInt (8,agent.getCategorieAgent ());
                                prepare.setString (9,agent.getName ());
                                prepare.setString (10,agent.getPostname ());
                                prepare.setString (11,agent.getName ());
                                prepare.setString (12,agent.getGenre ());
                                prepare.setString (13,agent.getDateOfBirth ());
                                prepare.setString (14,agent.getNationality ());
                                prepare.setString (15,agent.getFunction ());
                                prepare.setString (16,agent.getBaseSalary ());
                                prepare.setString (17,agent.getPhoto ());
                                int state = prepare.executeUpdate ();
                                yew(state == 1) status = true ;
                        }
                        catch (SQLException e) {
                                e.getMessage ();
                        }
                }
                return status ;
        }
    }
```

For performance and security reasons we used prepared queries. With the PreparedStatement object.

- **Consequences**:
  - Our base objects are used via Java objects

Nesher interacts with the database by encapsulating access to it and is thus prepared for a possible migration of the storage system in the event of a change, without having to impact the business layer of the software.

2.2. Contribution of patterns to the internal and external quality of the software

Design patterns are basically good design practices, proven solutions to recurring problems, a shared vocabulary for architects and developers to discuss issues, and an effective way for developers to discuss and share experience.

They make it possible to produce quality software with weak coupling between objects (components) constituting it, a strong cohesion between objects and a high sensitivity to modifications of the source code.

The internal quality of a software designed with an intensive use of design patterns, allows to have a source code conforming to the basic principles in OO (encapsulation, collaboration), definitions of lightweight classes, easy to understand, to maintain, to reuse, behavior distributed among classes that have the necessary information, a robust and maintainable system.

Conclusion

Our study has shown that it is, in the field of internal metrics of the object world, relevant to use some of these metrics in their context. The advantage of the study and the static analysis carried out is that: a) these metrics have been known for a long time and are well defined in the literature, b) many tools support them and therefore automate their calculation. This result therefore offers an immediate perspective to all developers and architects, in order to remain proactive and anticipate the result of their development through the use of good development practices. This result constitutes the fundamental contribution of this article. This contribution takes the form of 3 contributions.

We have identified 12 metrics of the object world. This set of metrics makes sure to sweep the 3 points of view on the internal components of an object-oriented software (class, package, etc.) as mentioned at the beginning of the second section of this article: internal view (cohesion, complexity, etc.), view of its interface (Afferent, Efferent, instability, abstraction, etc.), view of

its interactions in a given application context (Coupling). These three points of view are indeed useful to the two main stakeholders in the process of developing object-oriented applications: the developer of software components and the architect of an application that assembles these components. These metrics also operate at different levels of granularity to cover structural phenomena in all their variety: class level, package level, component level. Internal metrics reflect the internal complexity of an application by quantifying its size and internal dependencies (with class and package granularity). Metrics from the interface point of view capture the visible characteristics of a software component, that is to say the flow of information exchanged with its environment and its level of abstraction. Application point of view metrics quantify the coupling relationships between components of an application in terms of their number and "thickness". This is the first contribution of this article. Metrics from the interface point of view capture the visible characteristics of a software component, that is to say the flow of information exchanged with its environment and its level of abstraction. Application point of view metrics quantify the coupling relationships between components of an application in terms of their number and "thickness". This is the first contribution of this article. Metrics from the interface point of view capture the visible characteristics of a software component, ie the flow of information exchanged with its environment and its level of abstraction. Application point of view metrics quantify the coupling relationships between components of an application in terms of their number and "thickness". This is the first contribution of this article.

Through the static analysis carried out, we were interested in the information conveyed by these 12 metrics on our application (Nesher Ges-Paie). These metrics have shown their usefulness in internal use by making it possible to build an image of good development practices through the use of design patterns in the optimization of Nesher.

The optimization of Nesher by the use of design patterns allowed to give it a strong extensibility, to allow it to evolve easily in different fields of use and all this while reducing the cost and the maintenance effort.

This optimization reached its peak by the level of instability of its source code and that of its level of abstraction considered at different level of granularity (package), we noticed that the Distance from Main Sequence line of different packages does not exceed the critical value of 0.7, and its asymptotism to the referential line connecting point 1 of the abscissa (Instability) to point 1 of the ordinate (abstraction) inviting us that these two lines will never intersect at infinity . This is in no way a coincidence. It is undoubtedly an obvious consequence of the use of these patterns that allowed us to reach the second contribution of this article.

Without a rigorous discipline of the development team, the risk would be great that the software developed would be an empty shell with an irregular structure and weakly coherent. This analysis of good practices was a guarantee of satisfaction for our development teams and users of Nesher Ges-Paie.

By using at least 7 design patterns, we were able to put in place these good practices which have optimized Nesher Ges-Paie as a whole, we are focusing here on the Strategy pattern for example. By this pattern we focused on the three plundered of OO which are: inheritance, polymorphism and encapsulation.

Through the Strategy pattern we have been able to push back the limits of Inheritance and the polymorphism which is inherent to it by the use of composition to inheritance on the one hand and a strong manipulation of the object encapsulation on the other hand. , the realization of this pattern by interfaces and / or abstract class, disadvantages the dynamic polymorphism and binds the objects with a weak coupling and a strong internal cohesion between object.

This object encapsulation has made it possible to drastically reduce conditional tests, increase the extensibility of the source code to modifications, eliminate code redundancy, reduce the effort of finding oneself in the code in the event of failure and increase code reusability and flexibility.

In the end, our study allows us to affirm that: "the use of certain internal metrics coming from the procedural and object paradigm constitutes a relevant and operational tool for developers and architects in order to define as early as possible the internal quality of their products and anticipated the result externally ”.

This article can have several extensions and perspectives, the first perspective concerns orienting this approach on internal metrics towards the theme of refactoring.

BIBLIOGRAPHY

C., L. (2002). Applying UML and Patterns 2 Edition. Prentice Hall.

design-patterns. (2017). Retrieved from http://design-patterns.fr

Dugerdil, P. (May 2005). Complexity measurement and code maintenance. Univ. Of Applied Sciences.

G, M. (2011, 10-18). Retrieved from design-patterns: https://design-patterns.fr/singleton

G, M. (2012, 09 04). Retrieved from design-patterns: https://design-patterns.fr/observateur

GAUDAN, S. (2007). Doctoral thesis in computer science. Risk management of design faults related to object-oriented technologies for their use in critical avionics applications. Toulouse, France: National Institute of Applied Sciences of Toulouse.

Goodman, D. (2013). Software Design Principles (SOLID). Retrieved from davidgoodman.co.uk: http://davidgoodman.co.uk/tag/solid-principles/

grenncodelab. (2017). Retrieved from https://www.grenncodelab.fr/content/l'implementation-du-design-pattern-observer-peut-elle-reponde-àl'exigences

Hamza, S. (2014, December 14). Doctoral thesis. A pragmatic approach to measure the quality of applications based on software components. Brittany, France: UNIVERSITY OF SOUTHERN BRITTANY.

Kemerer, CS (1994). A metrics suite for object oriented design. In A metrics suite for object oriented design (pp. 20 (6): 476–493). Software Engineering IEEE Transactions on.

Larive, A. (2015). Source code quality measurement - Algorithms and tools. Retrieved from http://www-igm.univ-mlv.fr/~dr/XPOSE2008/Mesure%20de%20la%20qualite%20du%20code%20source%20-%20Algorithmes%20et%20outils/indice-de-specialisation. html

Martin, RC (2000). Design Principles and Design Patterns. Object Mentor.

McCabe, TJ (1976). A complexity measure. In TJ McCabe. Software Engineering, IEEE Transactions on.

MENGAL, P. (2013). METRICS AND CRITERIA FOR ASSESSING THE QUALITY OF THE SOURCE CODE OF SOFTWARE. Brussels: unpublished.

Netinbag. (2017, May 04). netinbag. Retrieved from netinbag.com: https://www.netinbag.com/fr/internet/what-is-software-evolution.html

openclassrooms. (nd). Retrieved from https://www.openclassrooms.com

Smacchia, P. (2013). nDepend Metrics. Retrieved from ndepend.com: http://www.ndepend.com/Metrics.aspx#RelationalCohesion

Sylvain Chardigny, AS (2008). Quality-driven extraction of a component-based architecture from an object-oriented system.

W, BB (1981). Software engineering economics.

Whismeril. (2019, May 06). Retrieved from source codes: https://codes-sources.commentcamarche.net/faq/823-design-pattern-strategy