

Experiments for Linking the Complexity of the Business UML Class Diagram to the Quality of the Associated Code

Gaetanino Paolone[1], Martina Marinelli[1], Romolo Paesani[1], and Paolino Di Felice[2(✉)]

[1] Gruppo SI S.c.a.r.l, 64100 Teramo, Italy
{g.paolone,m.marinelli,r.paesani}@softwareindustriale.it
[2] Department of Industrial and Information Engineering and Economics, University of L'Aquila, 67100 L'Aquila, Italy
paolino.difelice@univaq.it

Abstract. A relevant goal of software engineering is to assure the quality of object oriented software from the conceptual modeling phase. UML Class diagrams constitute a key artifact at that stage. Among existing models for iterative and incremental development of software systems, Model Driven Architecture (MDA) has reached a leadership position. MDA enables model-driven software development which treats models as primary development artifacts. The present empirical study answers the following three research questions: (RQ1) are there available in the literature class complexity metrics that can be adopted at the business level? (RQ2) Is it possible to adopt those metrics (if any) to predict the quality of the code returned by `xGenerator`? The latter is a Java technology platform for the creation of MVC Web applications, which implements the model-driven approach. (RQ3) Is it possible to identify a threshold for the adopted metrics (if any) that might suggest when a business Class diagram should be refactored?

Keywords: UML Class diagram · Complexity metrics · Attribute complexity · Method complexity · Association complexity · Class complexity · Code quality

1 Introduction

At a coarse level of granularity, the lifecycle of the development of software systems includes the phases of: *business analysis*, *system design*, and *implementation*. Maciaszek [1] defines business analysis as "the activity of *determining* and *specifying* customer requirements". The output of the requirements determination phase is a requirement document, input of the requirements specification phase. The Unified Modeling Language (UML) is the de facto standard for implementing the Object-Oriented (OO) paradigm that today is largely

This research was funded by Software Industriale.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12951, pp. 95–110, 2021.
https://doi.org/10.1007/978-3-030-86970-0_8

used in the development of complex software systems. UML describes notations for a number of different models that may be produced during analysis and design. UML *class diagrams* and *use-case diagrams* are the two most relevant specification techniques. These diagrams cover, respectively, the *structure* and the *behavior* of the system to be developed, as well as their *interactions*.

Among existing models for iterative and incremental development of software systems, Model Driven Architecture (MDA) has reached a leadership position. "MDA provides guidelines for structuring software specifications that are expressed as models. MDA separates business and application logic from the underlying platform technology."[1] MDA enables model-driven software development which treats models as primary development artifacts.

Figure 1 shows the three phases of the standard development of software systems as well as the alternative path in case of adoption of MDA. The same figure shows how MDA concepts link to the main phases of analysis, design, and implementation. In the right path, at the PSM level takes place a (manual, semi-manual, or automatic) transformation that generates the code.

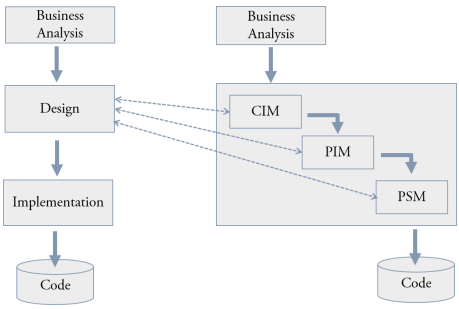

Fig. 1. Overview of two alternative software development lifecycles.

In [2] it is described an MDA-oriented software development process and a tool (`xGenerator`) which accomplishes the transformations between the levels. The latter is a proprietary Java technology platform for the creation of enterprise MVC Web applications. At a high level of abstraction, such a tool acts as a black box that receives as input an UML model and returns the Java code of the Web application.

[3] reports about the experiments carried out in order to investigate the quality of the code outputted by `xGenerator`. Through the `Springlint` tool,[2] it was possible to investigate the quality of the `Controller` and `Model` classes of the `ATMProject` by measuring a set of code metrics. The assessment of the quality of the code was based on the Software Architecture Tailored Thresholds proposed by Aniche and his colleagues in [4]. In such a paper, the authors take into account

[1] Sentences taken from: https://www.omg.org/mda/.
[2] http://www.github.com/mauricioaniche/springlint (accessed on June 20, 2020).

five different code metrics measuring coupling, cohesion, and complexity of OO code. Those metrics are able to distinguish the architectural role of the different layers of the MVC pattern. This feature is fundamental since `Controller` classes are different from `Model` classes. Accordingly, the threshold of the five metrics changes with regard to the class's architectural role. The numerical values of the five code metrics in [4] measured for the `ATMProject` were always below the threshold. Generalizing the findings of the experiments, Paolone et al. [3] stated that the classes returned by `xGenerator` are characterised by high cohesion, low coupling, and low complexity.

The present empirical study is motivated by answering the following three research questions:

(RQ1) are there available in the literature class complexity metrics that can be adopted at the business level? The business class diagram documents the internal structure of the business, at the same time it models the structure of the application to be developed; ultimately, it lays the foundation for the implementation work to be done. Therefore, monitoring the class diagram quality is a precondition to produce quality OO software.

(RQ2) Is it possible to adopt those metrics (if any) to predict the quality of the code returned by `xGenerator`? Anticipating the evaluation phase at the business level reduces the cost of the whole validation process. This is true in general and in MDE as well, where the models should be validated before being transformed, preventing the spread of defects introduced into the design.

(RQ3) Is it possible to identify a threshold for the adopted metrics (if any) that might suggest when a business Class diagram should be refactored? The knowledge of such a value is fundamental because it tells us when the refactoring process of the UML Class diagram should be started. UML model refactoring is a highly active area of research (see, for instance, [5]) and tools supporting the automation of the refactoring of UML Class diagrams are already available (e.g., [6,7]).

This study provides an answer to the three research questions listed above. To answer RQ1, we conducted a careful investigation of the state of the art. The decision was to adopt the complexity metrics proposed by Masmali and Badreddin [8,9] to measure the quality of the UML Class diagram of the `ATMProject`.

Concerning RQ2, the answer is affirmative but only for a portion of the whole generated code, namely the `Bean` classes which belong to the Model layer of the MVC pattern. The explanation is the following. First, as it is explained in [2], the `Bean` classes are in a 1-to-1 correspondence with the business classes in the UML Class diagram. Second, since `xGenerator` produces code in a deterministic way, once the quality of the Class diagram is proved, then it is correct to infer that the quality of the code that will be generated will be satisfactory as well. According to our long term experience in the development of enterprise Web applications for the bank sector, the `Bean` classes represent from 25 to 30% of the total number of lines of code of the whole application.

Concerning RQ3, the study allowed to determine empirically the upper bound of the value of the adopted complexity metrics (for the business classes), beyond

which we cannot exclude that might be appropriate refactoring the UML Class diagram.

The present study is an integral part of our previous research on automatic generation of code [2,3].

The paper is structured as follows. Section 2 introduces the background underlying the present work. A real-life case study is the focus of Sect. 3, while Sect. 4 presents and discusses the results of the study. Section 5 mentions a survey paper about complexity metrics for UML Class diagrams. Section 6 concludes this work.

2 Background

This section presents a slight variation of the proposal by Masmali and Badreddin [8], followed by the recall of the code metrics in Aniche et al. [4]. Many alternative metrics applicable to the Class diagram are available today. Some of them will be part of our future studies devoted to evaluate the quality of the code returned by `xGenerator`. The reasons behind the adoption of the code metrics in Aniche et al. [4] are explained in [2].

2.1 UML Class Diagram Complexity Metrics

Masmali and Badreddin [8,9] proposed to assign to a UML class a score equal to the sum of the complexity rate of its components, namely the attributes, methods and associations. The class complexity metrics has been validated from a practical point of view by applying the metrics evaluation framework proposed by Kaner and Bond [10] and from a theoretical point of view by referring to the Weyuker's properties [11]. In the following we recall the fundamentals of the class complexity metrics by Masmali and Badreddin since it plays a relevant role in our study.

Equation 1 computes the class complexity ($Class_{comp}$) as the sum of the complexity of all the attributes ($\sum_{k=1}^{n} Attr_{comp}$) and methods ($\sum_{k=1}^{n} Meth_{comp}$) in the class and the associations ($\sum_{k=1}^{n} Asso_{comp}$) among the classes.

$$Class_{comp} = \sum_{i=1}^{n_1} Attr_{comp} + \sum_{j=1}^{n_2} Meth_{comp} + \sum_{k=1}^{n_3} Asso_{comp} \tag{1}$$

Attribute Complexity. The complexity of each attribute ($Attr_{comp}$) is computed according to Eq. 2, where $Visibility_{score}$ and $Type_{score}$ are, in sequence, the complexity score of attribute visibility and attribute type. Table 1 and Table 2 show, respectively, the complexity score of the attribute visibility and attribute type. The values in the latter are not identical to those adopted in [8].

$$Attr_{comp} = Visibility_{score} + Type_{score} \tag{2}$$

Table 1. Complexity score of the attribute visibility.

Visibility	*Score*
Private	1
Protected, Package	2
Public	3

Table 2. Complexity score of the attribute type. The names are as those of Java since `xGenerator` returns Java code.

Type	*Score*
byte, short, int, float, long, double, char, boolean	1
String	2
Array, Date, Map	3
Object, List	4

Method Complexity. The complexity of each method ($Meth_{comp}$) is computed as shown in Eq. 3, where $Visibility_{score}$ is the complexity score of the method visibility, $ReturnType_{score}$ is the complexity score of the method return type, and the third term denotes the complexity score of its parameters. The complexity score of the method visibility is identical to the complexity score of the attribute visibility (see Table 1), while the complexity score of the method's return type and parameters are identical to that of Table 2.

We count constructor methods as ordinary methods. McQuillan and Power [12] have pointed out that there is no uniformity on this point among the available proposals about metrics for OO software. This uncertainty becomes an obstacle when comparing different metrics.

$$Meth_{comp} = Visibility_{score} + ReturnType_{score} + \sum_{l=1}^{n_4}(Parameter_{score}) \quad (3)$$

Association Complexity. UML classes may be interrelated to each other through: Association, Directed Association, Reflexive Association, Aggregation, Composition, Inheritance, and Realization. In [8], authors took into account Association and Directed Association. In our study we focus on Association and Aggregation, the only two relationships that are used in the real-life case study of the next section. These two relationships are equivalent from the point of view of the complexity. We use the generic term of association complexity to refer to both. The association complexity ($Asso_{comp}$) of a class is computed as shown in Eq. 4, where $Link_{score}$ denotes the complexity score of the association link in the class diagram. The idea behind the equation is simple: the higher is the number of links of a class the higher is its complexity.

$$Asso_{comp} = \sum_{p=1}^{n_5} Link_{score} \tag{4}$$

Table 3 collects the complexity score of the associations. Our values are not identical to those in [8].

Table 3. Complexity score of the associations.

Multiplicity	*Score*
1..1	1
0..1 or 0..* or 1..*	2
..	3

2.2 Code Metrics

Code metric analysis is the common method for assessing the quality of a software system. In the metrics-based methods the false-positives cannot be eliminated until the *context* is taken into account, and this because one set of thresholds do not hold good necessarily in another context. Until few years ago, software tools and published proposals did not take the system architecture into account. Consequently, *all* classes of OO applications were treated as they were equal to each other and, hence, assessed in the same way, regardless of their specific *architectural role*. This approach is not satisfactory if applied to a MVC Web application, where Controller classes are quite different from Model classes simply because they play very different roles: the former are responsible for coordinating the flow between the View and the Model layers, while the latter implements business concepts.

Adding context to code metrics is a recent research topic. In Aniche et al. [4], the authors adapt the threshold of metrics to the "class's architectural role" with regard to the (Spring) MVC pattern; where the *architectural role* is defined as the particular role that a class plays in a given system architecture. Specifically, Aniche et al. focus on the server-side code, namely on the Controller and Model layers. [4] relies on the Chidamber and Kemerer metrics suite [13], as it covers different aspects of OO programming, such as *coupling*, *cohesion*, and *complexity*.

Table 4 shows the class level metrics taken from [13], while Table 5 shows the thresholds found by Aniche et al. [4] for the classes of the five architectural role belonging to the Controller and Model layers. Each metric varies from 0 or 1 to infinite. The triple $[v_1, v_2, v_3]$ denotes, in order, the *low*, *high* and *very high* thresholds corresponding to *moderate/high/very high risk*. Classes in which metric values range in the 70%–80% percentiles have "*moderate risk*", while from 80%–90% the risk is "*high*", and "*very high*" between 90%–100%.

Table 4. The reference class level metrics in [4].

Metric	Acronym	Meaning
Coupling between Object Classes	CBO	The number of classes a class depends upon. It counts classes used from both external libraries as well as classes from the project
Lack of COhesion in Methods	LCOM	The lack of cohesion in a class counts the number of intersections between methods and attributes. The higher the number, the less cohesive is the class
Number Of Methods	NOM	Number of methods in a class
Response For a Class	RFC	It is the count of all method invocations that happen in a class
Weighted Methods per Class	WMC	Sum of McCabe's cyclomatic complexity for each method in the class

Table 5. The thresholds for the reference class level metrics of Table 4.

Architectural role	CBO	LCOM	NOM	RFC	WMC
Controller	[26, 29, 34]	[33, 95, 435]	[16, 22, 37]	[62, 78, 110]	[57, 83, 130]
Repository	[15, 21, 31]	[36, 106, 351]	[19, 26, 41]	[50, 76, 123]	[60, 104, 212]
Service	[27, 34, 47]	[133, 271, 622]	[23, 32, 45]	[88, 123, 190]	[97, 146, 229]
Entity	[16, 20, 25]	[440, 727, 1844]	[33, 42, 64]	[8, 12, 25]	[49, 61, 88]
Component	[20, 25, 36]	[50, 123, 433]	[15, 22, 35]	[56, 81, 132]	[52, 79, 125]

The MVC pattern implemented by `xGenerator` is not identical to Spring MVC. The consequence is that the structure, and hence the content, of the three layers of the code are not identical in the two solutions. In [3] it has been shown that the `Bean` classes (part of the MVC Model layer implemented by `xGenerator`) match the `Entity` classes (part of the Spring MVC Model layer). Within this study the `Bean` classes are the only classes we refer to because they are in a 1-to-1 correspondence with the business classes in the UML Class diagram.

3 The Case Study

The case study is the `ATMProject` presented in [3] and recalled below, briefly. It concerns the development of a Web application. The example was carried out by following the steps of Software Development Process proposed in [2] and by making use of `xGenerator`.

The basic *business vocabulary* (i.e., the general concepts) of our example is composed of the following nouns and noun phrases: *Customer, ATM, Bank Account, PIN, Bancomat code, Check Balance, Amount, Deposit Money, Transfer Money, Transaction, Currency, Maintenance,* and *Repair*.

The *business rules* to be implemented were the following: (a) a customer can hold many bank accounts, while an account is owned by a single customer. (b) An account may be linked to many cards. (c) A customer must be able to make a cash withdrawal from any account linked to the card entered into the ATM;

moreover a customer must be able to make a deposit to any account linked to the card entered into the ATM. (d) Many transactions can be made on an account. (e) A transaction refers to one currency and takes place at a physical ATM. (f) Many maintenances and repairs may happen on ATMs.

Figure 2 shows the diagram of the Business Objects of the ATM Subsystem at the CIM level. To be able to apply the method proposed in [8], it was necessary to enrich such a diagram with the data type of attributes, the list of methods as well as the input/output parameters and return values of each method. Notice that in the MDA paradigm, the information about data types, input/output parameters and the return values are part of the PIM and PSM activities.

4 Results and Discussion

4.1 Measurements at the Code Level

Figure 3 shows the names of the classes that compose the `Bean` package of the Model layer of the `ATMProject`. As already said, the attention is restricted to the `Bean` classes because they correspond to the classes of the UML Class diagram. Table 6 collects the numerical values of the metrics of Table 4 returned by `Springlint` for the `Bean` classes. The empty boxes in these tables denote the zero value. The values of metrics NOM and WMC are identical because the McCabe's cyclomatic complexity equals 1 for each of the methods belonging to the `Bean` classes. The output of `Springlint` is an HTML file. We named it `springlint-result ATMProject.html` and made available at URL www.softwareindustriale.it/ATMProject.html ATMProject. The results are excellent. In fact, the values of the five metrics for the `Bean` classes are all below the smallest threshold of Table 5 (row "Entity"). Those values confirm that the classes returned by `xGenerator` are characterised by high cohesion and low coupling.

Table 6. The values of the metrics for the `Bean` classes.

	CBO	LCOM	NOM	RFC	WMC
`ATM`	8	56	12	1	12
`BankAccount`	12	120	17		17
`Currency`	7	11	6	1	6
`Customer`	9	91	15		15
`Maintenance`	12	45	11		11
`MaintenanceCategory`	7	7	6		6
`Repair`	11	28	9		9
`Transaction`	12	120	17		17

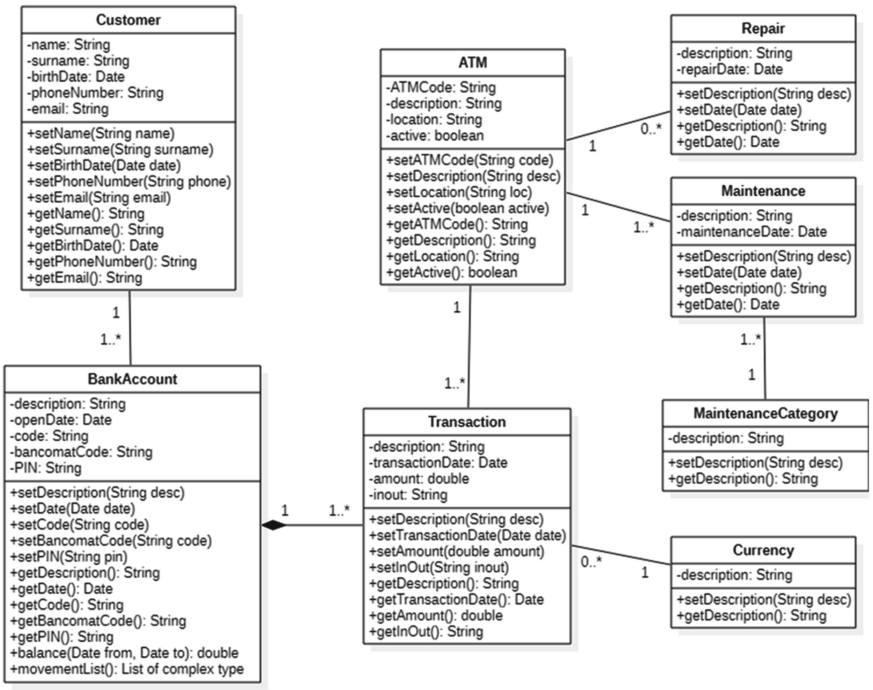

Fig. 2. The enhanced diagram of the Business Objects of the ATM Subsystem.

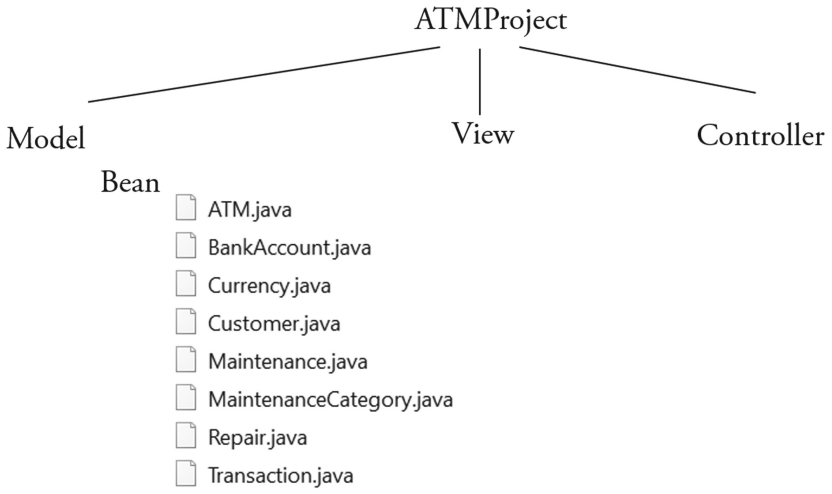

Fig. 3. The classes of the `ATMProject`

Table 7. The values of the complexity metrics for the business classes in the UML Class diagram of the `ATMProject`.

	$Attr_{comp}$	$Meth_{comp}$	$Link_{comp}$	$Class_{comp}$
ATM	11	38	6	55
BankAccount	16	69	3	88
Currency	3	10	2	15
Customer	16	52	2	70
Maintenance	7	22	2	31
MaintenanceCategory	3	10	2	15
Repair	7	22	1	30
Transaction	12	40	3	55

4.2 Evaluation at the CIM Level

Table 7 collects the numerical values of the complexity metrics of the business classes in the UML Class diagram of Fig. 2.

4.3 Discussion

As it is said in Sect. 4.1, the values of the five metrics for the `Bean` classes are below the smallest threshold of Table 5 (row "Entity"). This means that *all* these classes of the `ATMProject` are of good quality. This awareness implies that we can propagate backward the goodness of the code and state that the quality of the business classes in the UML Class diagram (from which the `Bean` classes come from) is good too. Exiting the `ATMProject` case study, now we are able to make the following claim that is true in general. Given a Class diagram, if none of its classes has a complexity value above the highest value in column $Class_{comp}$ of Table 7 (namely, 88), then we can state that the `Bean` classes that `xGenerator` can associate to the business classes will be of quality. This statement comes from the knowledge that `xGenerator` maps each class of the Class diagram at the CIM level uniquely (i.e., through a deterministic process) in a `Bean` class. This conclusion is not true in general, for example it is not true necessarily if programmers with different programming skills are charged for mapping the business classes into the code. Concrete evidence in support of the last claim is provided below. In summary, we are proposing the adoption of 88 as the empirical threshold for the class complexity computed according to Eq. 1.

In [15], Masmali and Badreddin proved that there is a high correlation between the values of the complexity metrics for the classes in the UML Class diagram and code smells in the associated code. For the empirical study, they investigated nine versions of the Java code of MobileMedia[3]: an application that manipulates photos, music, and video on mobile devices. Overall, the code

[3] https://github.com/julioserafim/MobileMedia.

(about 28,000 LOC) is composed of 252 classes and 997 methods. The third row of Table 8 (taken from [15]) shows the number of smelly classes detected by `JDeodorant` [16]. `JDeodorant` is an open source Eclipse plugin for Java, actively developed and well maintained[4]. The tool supports five code smells, namely: feature envy, type/state checking, long method, god class and duplicated code. `JDeodorant` has been widely investigated in the literature. For example, in [17] Paiva et al. conclude as follows a comparative study involving four smell tools: "`JDeodorant` identified most of the correct entities." An indirect evidence of the merits of such a tool is represented by the fact that it is used in several important studies (see, for instance, [18,19]). The examination of Table 8 shows that the percentage of smelly classes passes from 20% to 100% in correspondence with the values of the class complexity metrics that pass from the range 1–50 to 150–200. This data is a direct evidence that in absence of a deterministic transformation process of the business classes into `Bean` classes, the probability that the final code suffers of smells is very high.

Table 8. Correlation between the values of the complexity metrics for the classes in the UML Class diagram and code smells in the associated code of MobileMedia in absence of a deterministic mapping process between the two levels (source: [15]).

Class complexity	1–50	51–100	101–150	151–200
Number of classes	180	43	21	8
Number of smells	36 (20%)	15 (35%)	13 (62%)	8 (100%)

Table 9. The values of the NOM metrics applied to the business classes in the UML Class diagram of the `ATMProject`.

	NOM
ATM	8
BankAccount	12
Currency	2
Customer	10
Maintenance	4
MaintenanceCategory	2
Repair	4
Transaction	8

Overall, from the above considerations it follows that the three research questions in Sect. 1 have been answered.

[4] https://github.com/tsantalis/JDeodorant.

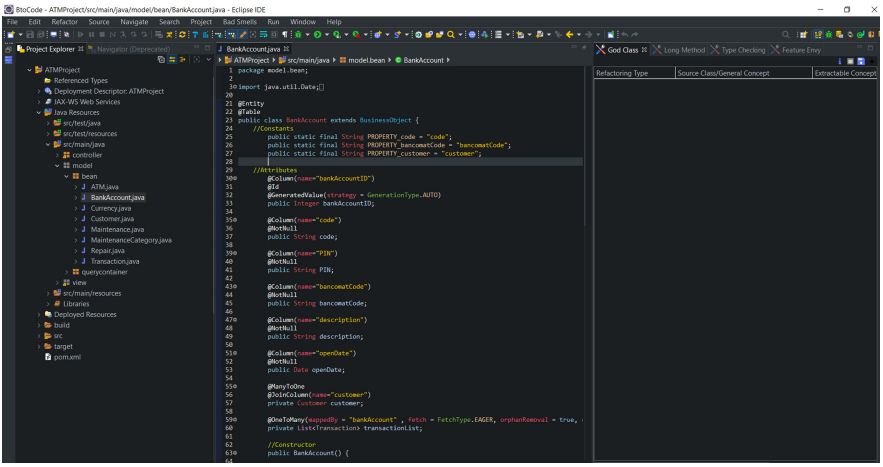

Fig. 4. The output of `JDeodorant` for the `BankAccount` class.

We conclude the discussion by adding three more comments, as indirect further evidences of the quality of the business classes of the `ATMProject` returned by `xGenerator`:

- of the five metrics for measuring the code quality (i.e., CBO, LCOM, NOM, RFC, and WMC), NOM is applicable also to the business classes of the Class diagram. Table 9 reports the NOM value for the eight business classes of the `ATMProject`. The values confirm the expectation, namely that, at the business class level, they are always lower than the values of the corresponding metrics applied to the code (see Table 6) and, moreover, their value is below the threshold of the code metrics (i.e., 33 of Table 5, row "Entity"). It follows that, if the value of NOM for the business classes is sufficiently below the 33 value, then this is a further evidence for predicting that the quality of the `Bean` classes returned by `xGenerator` will be good.
- We analysed the code of the `Bean` classes of the `ATMProject` by means of `JDeodorant`. No smells were detected by the tool (Fig. 4).
- In [3] the authors carried out experiments to measure the value of the six MVC smells part of the catalog in [14] for the `Bean` classes of the `ATMProject`. The values were all below the thresholds of the six metrics, which confirm that those classes are not smelly.

Threats to Validity

- *Internal validity* concerns factors that may impact the measurements' outcome. The only one factor that could negatively affect the internal validity of the results is about errors in the computation of the class complexity in absence of a supporting tool. In our case this eventuality did not occur

given the small number of classes in the UML Class diagram and because we repeated the computation many times.
- *External validity* refers to the relevance of the results and their generalizability. We conducted the experiments on a Java software project returned by a proprietary tool (`xGenerator`); consequently, we cannot claim generalizability of our approach.
- *Conclusion validity* refers to threats that can impact the reliability of our conclusions. At the moment, we are not able to exclude that projects bigger than `ATMProject` might give rise to a bigger value of the upper bound of the complexity metrics of the business classes (namely, 88 in Table 7). To overcome this issue, more experiments are needed. The future experiments have to concern Class diagrams with almost all the relationships among classes. This constraint, in turn, implies that the complexity model we referred to in the present study (taken from [8]) has to be extended or, alternatively, that different methods, among those already appeared (see next section), are taken into consideration.

5 Metrics for UML Class Diagrams

Most studies aiming at predicting the quality of OO software systems (e.g., class fault proneness, code readability, maintainability, implementation time and cost of the software system under development, ...) use metrics extracted from the source code. A much smaller number of proposals use design metrics collected from design artifacts. In this paper we were interested in the adoption of "early" metrics for the prediction of the quality of the code outputted by the proprietary tool `xGenerator` by measuring the quality of the UML Class diagram. We adopted a slight variation of the complexity metrics described in [8].

Due to the reached page limits, this section first focuses on a published survey paper about metrics for UML Class diagrams; then it mentions two specific proposals which adopt two completely different approaches. We plan to adopt both in future studies.

[20] is the first survey about metrics for UML Class diagrams. Genero et al. compared twelve proposals along five dimensions: (a) the definition of the metrics, (b) the goals pursued by the metrics, (c) the theoretical validation of the metrics (if any), (d) the empirical validation of the metrics (if any), and (e) the availability of automatic tools supporting the metric calculation. The findings of the survey can be summarized as follows:

- the research on measures for UML Class diagrams at the business level is scarce and is not yet consolidated (both theoretical and empirical validation is needed);
- most metrics are related to classes while little emphasis has been put on the Class diagrams as a whole;
- limited efforts have been put on measures about the relationships among classes;

– last but not least, the identification of robust thresholds for the proposed metrics is still an open issue.

Marchesi [21] proposed one of the earliest sets of metrics for computing the complexity of UML Class diagrams. The suite is applicable to a single class, a package or an overall software system. The metrics for a single class concern: weighted number of responsibilities; weighted number of dependencies; depth of inheritance tree; number of immediate subclasses; and number of distinct classes depending on the class.

In [22] a Class diagram is modeled as a graph. The nodes of the graph denote classes while the relations are modeled as edges between the nodes. A weight is associate to each edge according to the type of relations it corresponds to in the class diagram. Ten different kinds of relations in UML Class diagrams are considered. The *entropy distance* is used to associate a complexity measure to the class diagram.

6 Conclusions

This study put in correspondence the complexity metrics of the business classes of the Class diagram of a real-life project (`ATMProject`) with the complexity metrics of the associated code. A tool (`xGenerator`) generated the code mapping it, in a deterministic way, from the business classes. The determinism of such a mapping implied that from the quality of the code we inferred the quality of the classes. In this way it was possible to set an upper bound for the value of the class complexity metrics. Our proposal is to keep this value as the threshold for the complexity metrics for the business classes of future projects. Staying below such a threshold, we can say that any code outputted by `xGenerator` will be of good quality.

It has been pointed out that this claim is true only for a portion of the whole generated code, namely the `Bean` classes which belong to the Model layer of the MVC pattern. According to our long term experience in the development of enterprise Web applications for the bank sector, the `Bean` classes represent from 25 to 30%percent of the total number of lines of code of the whole application.

The joint usage of the class complexity measurement method discussed in the paper and `xGenerator` makes it unnecessary to carry out the measurements of the quality of the generated code by means of any of the available suites (e.g., [13]).

It is worth emphasizing that the measurement of the complexity of the business classes should be mandatory if the software development process is manual. In fact, in such a scenario discovering that there are business classes with a too high complexity value imposes refactoring actions on the Class diagram before proceeding in the development, because very likely the code that will be produced will result smelly.

The next experiments we are going to perform will link the complexity of the code outputted by `xGenerator` to the complexity of the business classes

computed by means of the suite proposed by Marchesi [21] and the graph-based approach in [22].

References

1. Maciaszek, L.A.: Requirements Analysis and System Design, 3rd edn. Addison Wesley, Harlow (2007)
2. Paolone, G., Marinelli, M., Paesani, R., Di Felice, P.: Automatic code generation of MVC web applications. Computers **9**(3), 56 (2020). https://doi.org/10.3390/computers9030056
3. Paolone, G., Paesani, R., Marinelli, M., Di Felice, P.: Empirical assessment of the quality of MVC web applications returned by xGenerator. Computers **10**(2), 20 (2021). https://doi.org/10.3390/computers10020020
4. Aniche, M., Treude, C., Zaidman, A., van Deursen, A., Gerosa, M.A.: SATT: tailoring code metric thresholds for different software architectures. In: IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Raleigh, NC, pp. 41–50 (2016). https://doi.org/10.1109/SCAM.2016.19
5. Misbhauddin, M., Alshayeb, M.: UML model refactoring: a systematic literature review. Empir. Softw. Eng. **20**, 206–251 (2015). https://doi.org/10.1007/s10664-013-9283-7
6. Nikulchev, E., Deryugina, O.: Model and criteria for the automated refactoring of the UML class diagrams. Int. J. Adv. Comput. Sci. Appl. (IJACSA) **7**(12), 76–79 (2016)
7. Deryugina, O., Nikulchev, E., Ryadchikov, I., Sechenev, S., Shmalko, E.: Analysis of the AnyWalker software architecture using the UML refactoring tool. Procedia Comput. Sci. **150**, 743–750 (2019). 13th International Symposium "Intelligent Systems" (INTELS'18)
8. Masmali, O., Badreddin, O.: Code complexity metrics derived from software design: a framework and theoretical evaluation. In: Proceedings of the Future Technologies Conference, 5–6 November, Vancouver, Canada (2020)
9. Masmali, O., Badreddin, O.: Comprehensive model-driven complexity metrics for software systems. In: The 20th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C 2020) Macau, China, 11–14 Dic (2020). https://doi.org/10.1109/QRS-C51114.2020.00115
10. Kaner, C., Bond, W.P.: Software engineering metrics: what do they measure and how do we know?. In: Proceedings of the 10th International Software Metrics Symposium, 11–17 September, Chicago, IL, USA, pp. 1–12 (2004)
11. Weyuker, E.: Evaluating software complexity measures. IEEE Trans. Softw. Eng. **14**, 1357–1365 (1988)
12. McQuillan, J.A., Power, J.F.: On the application of software metrics to UML models. In: Kühne, T. (ed.) MODELS 2006. LNCS, vol. 4364, pp. 217–226. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69489-2_27
13. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994). https://doi.org/10.1109/32.295895
14. Aniche, A., Bavota, G., Treude, C., Gerosa, M.A., van Deursen, A.: Code smells for model-view-controller architectures. Empir. Softw. Eng. **23**, 2121–2157 (2018). https://doi.org/10.1007/s10664-017-9540-2

15. Masmali, O., Badreddin, O., Khandoker, R.: Metrics to measure code complexity based on software design: practical evaluation. In: Arai, K. (ed.) FICC 2021. AISC, vol. 1364, pp. 142–157. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-73103-8_9
16. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: JDeodorant: identification and removal of type-checking bad smells. In: 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, pp. 329–331 (2008). https://doi.org/10.1109/CSMR.2008.4493342
17. Paiva, T., Damasceno, A., Figueiredo, E., Sant Anna, C.: On the evaluation of code smells and detection tools. J. Softw. Eng. Res. Dev. **5**, 7 (2017). https://doi.org/10.1186/s40411-017-0041-1
18. Nyamawe, A.S., Liu, H., Niu, Z., Wang, W., Niu, N.: Recommending refactoring solutions based on traceability and code metrics. IEEE Access **6**, 49460–49475 (2018). https://doi.org/10.1109/ACCESS.2018.2868990
19. Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A.: The scent of a smell: an extensive comparison between textual and structural smells. IEEE Trans. Softw. Eng. **44**(10), 977–1000 (2018). https://doi.org/10.1109/TSE.2017.2752171
20. Genero, M., Piattini, M., Calero, C.: A survey of metrics for UML class diagrams. J. Object Technol. **4**(9), 59–92 (2005). http://www.jot.fm/issues/issue_2005_11/article1
21. Marchesi, M.: OOA metrics for the unified modeling language. In: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy (March 8–11, 1998), pp. 67–73 (1998). https://doi.org/10.1109/CSMR.1998.665739
22. Xu, B., Kang, D., Lu, J.: A structural complexity measure for UML class diagrams. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004, Part I. LNCS, vol. 3036, pp. 421–424. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24685-5_56