

Evaluating the Quality of Class Diagrams Created by a Generative AI: Findings, Guidelines and Automation Options

Christian Kop^a

University of Klagenfurt, Universitaetsstrasse 65 – 67, Klagenfurt, Austria

Keywords: Class Diagram, ChatGPT, PlantUML, Domain Modelling, Quality Checking.

Abstract: Working with a Generative AI such as ChatGPT to create conceptual models and particularly Class Diagrams became very popular recently in the modelling community. Therefore, the objectives of this paper are the following: It analyses the previous scientific work to summarize the findings about the quality of AI-generated Class Diagrams. Own tests were carried out too. Based on these findings, the paper provides guidelines for manual quality evaluation. It also discusses automation options for evaluating the quality.

1 INTRODUCTION

Working with a Generative AI became very popular in the last two years. The driving factor was openAI, which successfully made ChatGPT (ChatGPT, 2024) available to the public.

Conceptual modelers also thought about the benefits of such Generative AI tools for modelling. They were interested to find scenarios in which such a support can be used.

Therefore, the first objective of this paper is to investigate what have already been found out in previous work with regard to the quality of automatically created conceptual models. This literature study was supplemented by own tests in order to get a better insight regarding the quality of the created models.

Since possible quality issues in a model make it necessary to establish guidelines for the evaluation of a model, such guidelines are the second objective of this article.

The third objective deals with the question, to which extent, the quality of the model can be automatically checked to minimize the manual effort.

In order to better address the above objectives, the article specialises to several specific aspects: ChatGPT was solely used to create Class Diagrams in PlantUML (PlantUML, 2024) for domain modelling. Prompt engineering was only carried out to the extent that it can support the generation of such a Class

Diagram. Thus, the findings of this papers are only applicable for these specialised aspects.


To address the objectives, the rest of the paper is structured as follows. In Section 2 the related work is listed. Section 3 firstly explains the specific aspects of how ChatGPT was tested and then it describes the findings regarding the quality of the AI-generated Class Diagrams. In Section 4 guidelines for manual evaluation are given on the basis of the findings. Afterwards this section focuses on the possibilities as well as limitations of automatic checking. The paper is summarized in Section 5.

2 RELATED WORK

Papers already exist, investigating the use of Generative AI for several aspects of the software development life cycle. Since this paper is dedicated on the support of Generative AI and Large Language Models (LLM) to generate Class Diagrams for domain modelling, the literature survey is confined to analyse only literature related to this scope.

(Combemale, 2023) gave a brief overview of the potentials of Generative AIs and the possibilities it could have for the future of conceptual modelling.

Some authors focused on the performance of Generative AI and particularly on ChatGPT to create Class Diagrams or Entity Relationship Diagrams (ERD) from textual descriptions.

^a <https://orcid.org/0000-0001-5800-458X>

A detailed analysis of working with ChatGPT was done in (Cámara, 2023). The authors listed 18 findings after testing ChatGPT to create UML Class Diagrams. Some of these findings focused on prompt engineering and how the possible size of the resulting model can influence the quality of such a generated model. In other findings, they explained what kind of modelling language concepts the AI cannot create properly. The authors also listed some types of errors, which they recognized, when evaluating the generated result.

In (Fill, 2023), the authors used textual descriptions to test how ChatGPT creates an ERD, a Business Process Diagram, a UML Class Diagram and the HERAKLIT model. On a general level, they discussed the system performance to generate these specific types of models. They also explained what kind of prompts they used to instruct ChatGPT how to do this task.

In (Chen, 2023), the authors examined, to which extend the ChatGPT versions GPT-3.5 and GPT-4.0 can support the automated generation of Class Diagrams. Therefore, the authors compared the results with reference models, which were created by experts. The authors found four categories of how AI generated model elements matches with model elements in the reference models: a semantically exact match, a semantically exact match but not of the same type, a partial match and no match. Based on the examinations, the authors concluded, that an automated generation do not always produce a perfect result. However, they found out, that the creation of classes performed better than the creation of attributes. The creation of attributes performed better than the creation of relationships. They also achieved better results with GPT-4.

Another paper (Wang, 2024) examined how well ChatGPT performed to create three types of UML diagrams, namely Use Case Diagrams, Class Diagrams and Sequence Diagrams. They also came to the conclusion that the Generative AI performed differently for different types of modelling elements. The system performed better in recognizing e.g., classes and use cases from respective textual descriptions. However, the system struggled to recognize relationships.

A set of contributions not only focused on one Generative AI (e.g., ChatGPT) but compared its effectiveness with that of other approaches, intended for creating conceptual models.

In (Omar, 2023), the authors examined how good ChatGPT can model entity types, relationship types and multiplicities in ERD with respect to other modelling tools that use natural language processing

or ontologies. The authors concluded that among all the tools ChatGPT performed best but also stated, that human experts should evaluate the generated result

A rule-based approach, a machine learning approach feature engineered by one of the authors and ChatGPT were compared in (Bragilovski, 2024). It turned out that the two machine learning approaches outperformed the rule-based approach, but none of them was able to outperform a human expert.

In (Bozyigit, 2023), the authors compared their own tool with ChatGPT. They came to the conclusion that on average the own tool performed better. However, the performance in generating relationships was low in both tools.

Another type of contributions raised the question about the role a Generative AI can play in the educational context of conceptual modelling and software engineering. For instance, in (Cámara, 2024) and (Xue, 2024), the authors focused on the question, how students perform in using ChatGPT. In both studies, it turned out that Students who used ChatGPT performed better. In (Xue, 2024) it was stated that the difference was not significant. In another study (Saito, 2023) ChatGPT was used to return the difference between a modelling solution from students with a given perfect solution. In (Morales, 2023), the authors tested ChatGPT in the task to explain, if a generalisation relationship between two notions is appropriate. After a couple of tests, the authors concluded that ChatGPT cannot be used as an advisory tool for this specific purpose. Finally, in (Wang, 2023), the authors also discussed the issue of plagiarism and how good it can be detected either by course instructors or specialized AI tools.

The impact of Generative AIs in the software life cycle were addressed in the following contributions.

Netz et. al, (Netz, 2024) presented an approach, where ChatGPT transformed natural language text into a domain specific modelling language for building web applications. Although the results were promising for the authors, they also stated that the detected issues regarding semantic correctness should be evaluated by a human expert.

In (Kanuka, 2023), ChatGPT was used to create both a design model (UML Class Diagram) as well as python code from a textual description. In addition, it had to establish a traceability between these two specifications. Regarding traceability, it took several attempts interacting with the AI until it fulfilled their expectations.

The use of LLMs for the development of model and code variants in software product lines was introduced in (Acher, 2023).

The use of ChatGPT in different early phases of

software development, is presented in (Rouabhia, 2023). The AI created, requirements from an input document. From these requirements, use cases were generated. The use cases were in turn the starting point for the creation of a Class Diagram.

In (Härer, 2023), the author developed a modelling application, that interacts with Generative AI APIs. The application then takes the generated model specification and draws it in the required type of diagram (e.g., Class Diagram).

Lastly two authors (Conrardy, 2024) (Buchmann, 2024), explored if drawings and sketches of Class Diagrams can be transformed to a textual specification of this Class Diagram.

To summarize, authors of previously published related papers presented general findings like e.g., the percentage of correctness for different types of model elements. More detailed findings about some errors were given in (Cámara, 2023). Overall, the conclusions in these papers are, Generative AI and particularly ChatGPT does a good job, however, the quality differs with respect of the type of modelling element.

This literature analysis raised the question: Do even more and particularly more concrete quality issues exist that were not already mentioned? Furthermore, no explicit guidelines were found in literature that could guide modelers to evaluate the quality of the AI-generated Class Diagram. Finally, no discussion about automation options for this evaluation task was found.

This paper is therefore dedicated to these open questions.

3 FINDINGS

This section deals with the testing of ChatGPT for generating Class Diagrams. It was tested both with existing examples of input texts given in the literature as well as new, own text examples.

3.1 General Considerations Regarding the Tests

ChatGPT was used, because it was frequently used in the literature analysed. During the interaction with the AI, an explanation was given about the nature of the expected output. Namely, it should be a domain model and it should be a UML Class Diagram in PlantUML notation. Therefore, the own textual descriptions started with the following explanation: “Create a domain model with a UML class diagram for the text below and output it in PlantUML: “. Afterwards the text describing the content for the

class diagram followed. In the text examples found in literature, the introductory and explanatory part of these texts were adopted, if needed, such that the purpose of the models and the desired Class Diagram notation were expressed. No effort was spent to repeat the interaction with the AI within one chat session by giving additional text inputs in different, more detailed variations in order to improve the received output from the previous interaction. Here, it is assumed that a modeller should always know what is right and what is wrong in the AI-generated conceptual model, before proceeding with any next step. Thus, checking quality of the resulting Class Diagram output is important anyway.

PlantUML was taken, since it was frequently used in literature too. Furthermore, it is a text-based model specification language that ChatGPT already knows. Hence, it was assumed that creating a model should be like creating code for ChatGPT. Although, PlantUML has advantages, it has a drawback in the context of domain modelling. PlantUML was designed for modelling UML diagrams at each phase of software development. In domain modelling, a lot of these concepts and notations are not necessary. However, apart from a few exceptions, many of those concepts relevant for the implementation phase were not created by the AI. Therefore, only a subset of PlantUML had to be considered.

Once the models were created, the quality of these models then were manually evaluated.

3.2 Test Results

The following was found about the important concepts in the resulting model.

Classes: ChatGPT created classes from the textual description but due to missing information in the textual description, sometimes classes did not have attributes. Below there is such an example:

```
class AcademicStaff {
}
```

In all the tests, classes were involved in relationships (i.e., association, aggregation, composition or generalization).

Enumeration Types: ChatGPT created enumeration types properly, where it could recognize it in the input text.

Attributes: ChatGPT created attributes that represent relationships to another class. For example, in a created class `sensor` the following attribute appeared referencing to another class `Manufacturer`:

```
-manufacturer: Manufacturer
```

Here, the AI did not create an association, the usual modelling element in domain modelling. This kind of defect was already mentioned in (Cámara, 2023). In the following, it will be referred to such an attribute as a “referencing attribute” to distinguish it from an attribute that describes the property of a class without referencing to another class. The AI also sometimes added attributes to classes that were not part of the input text, since it used its acquired internal knowledge.

Finally, if in the input text, names of attributes are mentioned for different classes, then these attributes also appear in these different classes in the resulting model. This is generally okay, but it could also mean that restructuring is necessary. Take for instance the following excerpt from a created Class Diagram, which a modeler can either take as it is or s/he can think of alternative modelling decisions.

```
class Owner {
    +String firstName
    +String surname
    +String address
    +String telephone
    +String email
}

class Customer {
    +String firstName
    +String surname
    +String email
    +String telephone
}
```

Types of Attributes: ChatGPT does a good job in finding types of attributes even if these types are not mentioned in the input text.

Associations: The Generative AI created associations that consists of multiplicities. These multiplicities however were not always correct. Sometimes, also association names were missing. In one case, the association name actually was a role specification of a participating class. Sometimes, the AI created associations with a navigation direction. Such associations with navigation direction were also observed in (Cámara, 2023). In addition, and due to a lack of sufficient information in the input text, they did not have multiplicities and association names either. Here are several examples of associations from different created models:

```
WeatherStation --> Location
University "1" -- "0..*" Faculty
Resident "0..*" -- "many" Item
Customer "1" -- "0..*" Account : holder
Institute "1" -- "0..*" Employee :
```

Sometimes, it also happened that the AI created associations that were not given in the input text.

In one case, it also created two associations from two different sentences, although these sentences only described one association from the perspectives of each involved class.

Aggregations and Compositions: In (Cámara, 2023) it was complained, that the composite notation was placed illegally on both ends of a composition. However, this could not be reproduced during the tests for this paper. During the tests, it was recognized that the AI did not generate the multiplicities for special relationships. It even happened that a composition was created although it is not needed and it had an association name. Below, there is a composition and an aggregation example from different created models.

```
Property *-- Newspaper : AdvertisedIn
EBike o-- Battery
```

Generalizations: The AI often created correct generalisations but sometimes, it found alternative constructs for a generalization. It used association names e.g., `shippingAddress` and `billingAddress` to express the role of the class `Address` participating in an association. But no generalization between `shippingAddress` and `billingAddress` to `Address` was created.

In one test case it also wrongly generated an association with a navigation direction and generated the association name `inherits` although a generalization would be correct in this case. In another test case, the generalization was expressed with the keyword `extend`.

In a case, it also happened that the AI wrongly modelled a class as a super class although it was the subclass in this generalization e.g.,

```
Lecturer <|-- ScientificEmployee
ScientificEmployee <|-- Employee
```

Association Classes: They are useful, if for example, attributes do not belong to only one of the participating classes. The attribute “mark” is a common example in the domain of a student information system, where students can manage, to which courses they will enrol. A mark belongs neither to a course nor to a student only. In (Cámara, 2023) the authors found out, that an association class is not generated properly. When carrying out the own tests, it turned out that instead a compensation was created in the test case of the student information system scenario. The additional class `Enrolment` was created. This class contained the specific attribute `mark` but the two classes `Student` and `Course` were

not connected to each other. Instead, each of them referred to `Enrolment` via an association. Hence, the created PlantUML output did not have the correct concept and notation for specifying an association class (e.g., `(Student, Course) .. Enrolment`). Thus, `Enrolment` was only treated as a normal class in the output.

Methods: In general, methods should not appear in a domain model focusing on relationships between notions. However, if ChatGPT already generates methods in some cases, then they can also be seen as a chance to get more information about the resulting model. Methods of a class have in common, that they operate either on attributes of this class or on an instance of another class, to which this class has an association, aggregation relationship or composition relationship respectively. An example for a method specification that was created in a test case is:

```
+ getBranches(): List<Branch>
```

Therefore, it is worth to take a look at the signature of the method (i.e., the name of the method and the parameter of the method as well as the return parameter of the method if available). If the signature of the method indicates that the object of another class is involved, then it can be checked if the class of the method is directly connected with this other class via an association, aggregation- or composition relationship. Otherwise, it can be checked if attributes appear in this class that are used in the method. The name of a method is a hint sometimes. During the tests with the input texts, it was sometimes recognized that the AI created methods, but from its signature it could not be concluded, which attributes are involved if the method does not operate on instances of classes. This however would be necessary, if the modeler would like to be sure that the model is complete in this respect.

4 MANUAL EVALUATION AND AUTOMATION OPTIONS

Based on the results of the tests, this section provides guidelines for manual evaluation as well as a discussion of automation options.

4.1 Guidelines for Manual Evaluation

Although AI often delivers good modelling results, problems can arise that are not present in manually generated models. Therefore, a modeler has to carefully check the resulting model.

Classes: For each class firstly, the modeler should check if it is necessary in the model according to his/her domain experience. Afterwards, it has to be checked if it has attributes. It should be also tested if a class is at least involved in one relationship (association, aggregation, composition or generalization) with another class.

Enumeration Types: Although, enumeration types were created well, it is also good to check them. For each enumeration type the modeler should check if it is necessary. Afterwards, s/he should check if it has sufficient and correct enumeration values.

Attributes and Types: For each attribute per class firstly it has to be checked if it is not a referencing attribute to another class that shall be better modelled via an association, aggregation or composition. If it is not a referencing attribute, it has to be checked if it is relevant or not. If other classes also have attributes with the same name, the structural usefulness has to be checked. If a multiplicity is specified for an attribute, the modeler should also check if it is correct. Afterwards, the modeler should check if the type of the attribute is correct.

Associations: For each association firstly, it has to be checked, if it is necessary in the model and it is not redundant with respect to another association, aggregation or composition connecting the same pair of participating classes. Afterwards, the modeler should check if the multiplicities are correct, if it does have an adequate association name and if a created navigation direction is really necessary.

Aggregations and compositions: For each aggregation and composition respectively, firstly, it has to be checked if it is necessary in the model and it is not redundant with respect to another association, aggregation or composition connecting the same pair of participating classes. Afterwards, it has to be checked if it does have the aggregate/composite notation only on one end and it does have correct multiplicities. In case of a composition, it has to be checked, if the multiplicity at the composite end is "1". It should be also evaluated if navigation directions are necessary.

Generalizations: For each generalization, firstly, the modeler should check if it is necessary in the model and is not redundant with respect to another generalization connecting the same pair of participating classes. Afterwards, it should be checked if the role of the super class and sub class is correctly assigned to the participating classes.

Association Classes: The modeler should evaluate the model regarding association classes. S/he should check, if association classes are missing or classes are wrongly generated as a compensation for an

association class. In addition, it has to be checked, if an attribute is assigned to only one participating class although it should be part of an association class.

Methods: For each method the modeler should consider, that it can provide further information. A method should therefore be checked to determine if it uses the attributes of its class or instances of another class. For a method that uses the attributes, it has to be clarified, which attributes are used and if they already appear in the class. If instances of another class are used in a method, then it has to be checked, if the class of the method also has a direct relationship (i.e., via an association, aggregation relationship or a composition relationship) to that other class.

Missing Model Elements: Finally, once all the elements listed in the created model are evaluated, the modeler should check if any further necessary information (e.g., additional classes, enumeration types, attributes associations, aggregations, compositions, generalizations, association classes) is still missing.

4.2 Automation Options

This section discusses, to which extend a modeler can get tool support to find issues. Both, issues regarding semantic quality as well as syntactic quality (Krogstie, 2002) are discussed. According to Krogstie (Krogstie, 2002), particularly semantic quality is inter-subjective, i.e., agreed upon the social actors. It also has to be noted, that many issues are subjective with respect to the intention of a modeler. What could be a defect for one modeler could be correct for another one. The main task of a tool offering automatic support, is therefore only to draw the attention of the modeller to a specific section of the model, where an issue might exist. It is the decision of the modeler if this is a defect that has to be corrected. As a proof of concepts, a prototype was developed. This prototype covers the automation options explained here.

Classes: Testing if a class might be underspecified can give the modeler a hint that there is missing information. Once each class with its attributes is collected in an internal data structure, it can be computed, if the class has no attributes, or the number of its attributes is under a manually defined threshold. Setting an adequate threshold of course needs the experience of a modeler.

A class is also underspecified if it is not involved in any relationship, i.e., if the difference between the set of all classes and the set of classes participating in a relationship is not empty, then the elements in the non-empty set are classes with no relationships.

Enumeration Type: An issue can appear in an enumeration type, if it does not have sufficient enumeration values. In general, an enumeration type should have at least two values, as a general threshold. But if it has more than two enumeration values, it remains to the modeler's experience if s/he agrees with this number.

Attributes and Types: An attribute can be automatically tested, if they belong to the category of referencing attributes. The simplest case is an attribute name in singular form, which is identical to a class name other than the class name, in which the attribute appears. For transforming the plural form of an attribute's name to its singular form, though, rules or a digital dictionary support have to be used. It can also be tested if the name without a certain suffix (e.g., "ID") matches with the name of a class. Lastly, if the name of the type in the attribute's specification is identical to another class, then this also supports the detection of referencing attributes.

It can also be automatically detected, if an attribute's name is listed in multiple classes. If this is detected, this could indicate that restructuring could be necessary. As already mentioned in Section 3.2 however, it depends on the context and thus the design decision of the modeler. Here, an automatic support only gives the information that it exists.

No automatic support can be given to check, if the type of an attribute is appropriately chosen. This has to be done manually by the modeler.

Associations: The check for incompleteness itself does not need any contextual information and can be done automatically. An association name and multiplicities are either available or not. Navigation directions also can be detected easily by just looking in association specifications, if they exist.

However, context plays a dominant role in deciding whether a multiplicity is right or wrong (e.g., is "*", is "1..*" or even "1" the correct solution). Therefore, it cannot be automatically tested, if the multiplicities are correct. The same holds for association names. Whether an association should have an association name or whether existing names are correct remains the decision of the modeler. Wrong association names might be detected, if a tool also manages a list of association names that are wrong in many contexts but this gives no clue if a name is wrong or correct in a given context.

Aggregations and Compositions: A wrong notation as mentioned in (Cámara, 2023) – i.e., aggregate / composite symbol on both ends - can be automatically filtered out as a syntactical error.

Missing multiplicities can also be detected automatically. Apart from one exception, it is once

again impossible to automatically distinguish between wrong and correct multiplicities in aggregations and compositions. The only exception is the multiplicity at the composite end of a composition. This multiplicity must be always "1" end hence such a violation can be automatically detected.

It is the decision of the designer, if a name should be given for an aggregation or composition. Usually, composition and aggregations already have a semantic, hence a name is an add-on and it should not be the wrong name. Therefore, the tool can inform the modeler about named aggregations and compositions. Showing such names to the modeler gives him/her the opportunity to decide if a name for the aggregation or composition is adequate.

Generalizations: It is difficult to automatically detect the relevance of a generalisation and correctness of a generalization specification. But in two situations, a tool can at least tell the modeler to take a look.

Regarding relevance of a generalization, let's suppose a class G has several subclasses e.g., S1, S2, S3 and none of these subclasses have attributes as well as relationships to other classes. Showing such a case to the modeler gives him/her the chance to think if restructuring is necessary or not. Both is possible, it might be correct but it could also be an indicator for missing information.

Regarding correctness of a generalization specification, suppose a situation, where the name of one class is also a substring of the other involved class name (e.g., `ScientificEmployee`, `Employee`). In this example the class `Employee` is the superclass in the generalization. If, however, the generalisation relationship does not reflect this, but the generalization arrow is pointing from `Employee` to `ScientificEmployee` instead, then this generalisation specification is wrong. This can be easily detected by comparing the two strings.

Since issues in generalization relationships can only be recognized under certain conditions, the modelling expert is particularly in demand here.

Redundant Relationships (associations, aggregations, compositions, generalizations): An automatic support can at least inform the modeler that a pair of classes appears in more than one relationship. Once again, it depends on the context if the same pair of classes should really be connected to each other in more than one association. The modeler with his/her knowledge should decide if a redundancy exist in the model. However, it is more unlikely that the same pair of classes is part of more than one composition, aggregation or a combination of

composition, aggregation and association. It is also unlikely that the same pair of classes is part of more than one generalization relationship.

Association Classes: The evaluation if an association class exists, also strongly depends on the context and cannot be done without the experience of the modeler. If the AI cannot create such a class properly, then automatic support is only possible, if the Generative AI uses a specific strategy to compensate an association class. Particularly, if a class is created, whose name is a nominalization of a verb or an adjective. Since some nominalizations have suffixes like e.g., "ing", "ment" etc., all classes could be inspected if their names contain one of those suffixes. These classes are then shown to modeller. Admittedly, this is only a minimal support, and does not guarantee successful detection of association classes. Neither it can be expected that the AI creates association classes in this way nor it can be expected, that a class with a specific suffix is always an association class.

Methods: As already explained, the signature of methods can be used to find out, if attributes or relationships to classes are missing. The idea here is to analyse the signature in order to find words or tokens in the method's name or in a non-empty parameter list that matches with existing attributes of the method's class or other class names. The modeller has to be informed if this analysis fails. S/he then has to decide if information is missing. This automatic support has limitations, since there is no commitment at all, that the name of a method always has to match with an attribute's name or the method has parameters.

5 CONCLUSION

This paper presented guidelines to evaluate the quality of AI-generated Class Diagrams. The paper also discussed if a modeler can be supported in these evaluation tasks. It was argued that automation is only possible in co-operation with the modeller. A tool can just encourage the modeler to have a look at a created model. It is then up to the modeler do make the right decision.

Currently, only the performance of ChatGPT for generating a Class Diagram was tested. Future work could focus on other Generative AI tools and on other modelling languages. The presented analysis of the output are qualitative observations (i.e., what kind of issues appear). A more extensive study could focus on quantitative factors like the frequencies of certain issues in the created models.

REFERENCES

- Acher, M., Martinez, J. (2023). Generative AI for Reengineering Variants into Software Product Lines: An Experience Report, In *SPLC '23: Proceedings of the 27th ACM International Systems and Software Product Line Conference*, pp. 57 – 66, <https://doi.org/10.1145/3579028.3609016>.
- Bozyigit, F., Bardakci, T., Khalilipour, A., Challenger, M., Ramackers, G., Babur, Ö, Chaudron, M., R., V. (2023) Generating domain models from natural language text using NLP: a benchmark dataset and experimental comparison of tools, *Software and Systems Modeling*, <https://doi.org/10.1007/s10270-024-01176-y>.
- Buchmann, T., Fraas, J. (2024) AI-Based Recognition of Sketched Class Diagrams. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pp. 227-234
- Bragilovski, M., van Can, A. T., Dalpiaz, F., Sturm, A. (2024) Deriving Domain Models from User Stories: Human vs. Machines. In *IEEE 32nd International Requirements Engineering Conference (RE)*, IEEE Xplore, pp. 1 - 12, DOI: 10.1109/RE59067.2024.00014-
- Cámara, J., Troya, J., Burgueño, Vallecillo, A., (2023). On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. In *Software and Systems Modeling* Vol. 22, pp. 781 – 793, <https://doi.org/10.1007/s10270-023-01105-5>.
- Cámara, J., Troya, J., Montes-Torres, J., Jaime, F. J. (2024). Generative AI in the Software Modeling Classroom: An Experience Report with ChatGPT and UML. In *IEEE Software*, vol 41 (6) pp. 1 – 10.
- ChatGPT (2024). <https://openai.com/chatgpt/> (last access: 17th Dec. 2024)
- Chen, K., Yang, Y., Chen, B., López, J., H., A. (2023). Automated Domain Modeling with Large Language Models: A Comparative Study. In *26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE Xplore, pp. 162 – 172.
- Combemale, B., Gray, J., Rumpe, B. (2023). ChatGPT in software modeling. In *Software and Systems Modeling*. <https://doi.org/10.1007/s10270-023-01106-4>.
- Conrardy, A., Cabot, J. (2024). From Image to UML: First Results of Image-based UML Diagram Generation Using LLMs. In *arxiv.org*, arXiv:2404.11376v1.
- Fill, H.-G., Fettke, P., Köpke, J. (2023), Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT. In *Enterprise Modelling and Information Systems Architectures*, Vol. 18, No. 3, pp. 1 – 15, DOI:10.1847/emisa.18,
- Härer, F. (2023). Conceptual model interpreter for Large Language Models. In *arxiv.org*, arXiv:2311.07605.
- Kanuka, H., Koreki, G., Soga, Ryo, Nishikawa, K. (2023). Exploring the ChatGPT Approach for Bidirectional Traceability Problem between Design Models and Code. In *arxiv.org*, arXiv:2309.14992v2.
- Krogstie, J., (2002). A Semiotic Approach to Quality in Requirements Specifications. In *Organizational Semiotics. IFIP, vol 94*. Springer, pp 231–249
- Morales, S., Planas, E., Clariso, R., Gogolla, M., (2023). Generative AI in Model-Driven Software Engineering Education: Friend or Foe? In *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, IEEE Xplore, pp. 110 – 113.
- Netz, L., Michael, J., Rumpe, B. (2024). From Natural Language to Web Applications: Using Large Language Models for Model-Driven Software Engineering. In *Modellierung 2024, Lecture Notes in Informatics (LNI)*, pp. 179 – 195.
- Omar M.A., (2023). Measurement of ChatGPT Performance in Mapping Natural Language Specification into an Entity Relationship Diagram. In *Proceedings of the 2023 IEEE 11th International Conference on Systems and Control*, IEEE Xplore, pp 530 – 535.
- PlantUML, (2024). <https://plantuml.com/en/>, (last access: 17th Dec. 2024)
- Rouabhia, D., Hadjadj, I. (2023). AI as a Co-Engineer: A Case Study of ChatGPT in Software Lifecycle. In *Research Square*, pp 1 – 86, <https://doi.org/10.21203/rs.3.rs-3809973/v1>.
- Saito, D., Minagawa, T., Hisazumi, K. (2023). Automated Review Tool for Educational Models Utilizing Generative AI. In *Asia Pacific Conference on Robot IoT System Development and Platform (APRIS2023)*, pp. 50 – 51.
- Wang, B., Wang, C., Liang, P., Li, B., Zeng, C. (2024) How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts. In *arxiv.org*, arXiv:2404.17739v1
- Wang, K., Akins, S., Mohammed, A., Lawrence, R. (2023). Student Mastery or AI Deception? Analyzing ChatGPT's Assessment Proficiency and Evaluating Detection Strategies. In *arXiv.org*, arXiv:2311.16292v1.
- Xue, Y., Chen, H., Bai, G. R., Tairas, R., Huang, Y. (2024)- Does ChatGPT Help With Introductory Programming? An Experiment of Students Using ChatGPT in CS1. In *46th International Conference on Software Engineering: Software Engineering Education and Training (ICSESEET)*, pp. 331 – 341, <https://doi.org/10.1145/3639474.3640076>.