

From Object to Class Models: More Steps towards Flexible Modeling (Short Paper)

Martin Gogolla¹, Bran Selic², Andreas Kästner¹, Larousse Degrandow¹ and Cyrille Namegni¹

¹University of Bremen, Computer Science, 28334 Bremen, Germany

²Malina Software Corp., Ottawa K2J 2J3, Canada

Abstract

This contribution discusses a flexible modeling approach that proposes to develop class diagrams starting from object diagrams. On the one hand, the contribution explains in a general way the benefits of flexible visual modeling by allowing a lively development process through relaxing the formal requirements for artefacts in the work process. On the other hand, the contribution shows a concrete example and explains an implementation in a tool, in particular how to cover whole-part relationships, generalization and an improved handling for association multiplicities. The aim is to give developers the option to let their ideas flow in a free way with few creativity restrictions by a tool. Flexibility may be gained by transitioning in the work process between specific, instance-based visual models and generic, type-based visual models where both kinds of models allow for incompleteness or inconsistency.

Keywords

Object model, Class model, Flexible modeling, Incomplete model, Inconsistent model

1. Motivation

There is no doubt that precision plays a fundamental role in all good engineering. It is particularly significant in software engineering, which is founded to a great extent on applied mathematical logic.

In essence, precision implies the elimination of ambiguity, which is typically a source of uncertainty and can ultimately lead to invalid or inappropriate design decisions. In engineering, precision is typically achieved by the application of some type of formal mathematical methods. By means of formal mathematical constraints and validity rules, it is possible to define elements of a design in a way that eliminates subjectivity or the possibility of misinterpretation.


However, this level of precision does not come easily. Often it is only possible if we have sufficient understanding of the topic. And therein “lies the rub”; reaching an understanding of some complex aspect takes time. One of the most effective means for reaching understanding is direct experience with the subject matter. This typically involves trial and error, so that we can appreciate not only what works and why, but equally important, what does not work and why. For this reason, prototyping is essential to most complex engineering projects.

2nd Int. Workshop on Foundations and Practice of Visual Modeling, July 4–8, 2022, Nantes, France, Co-located with STAF 2022

✉ gogolla@uni-bremen.de (M. Gogolla); selic@acm.org (B. Selic); andreask@uni-bremen.de (A. Kästner); degrandow@uni-bremen.de (L. Degrandow); jikename@uni-bremen.de (C. Namegni)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

In this process of forming an understanding through trial and error, introducing formal constraints too early in this process can create a kind of “bureaucratic” hurdle that can stand in the way of not only understanding but also creativity. Innovative ideas often start off in vague and imprecise form, and need to be refined gradually before an informed decision can be made whether to adopt them or discard them. Consequently, formal methods should only be applied when sufficient information (i.e., understanding) has been attained.

Based on the above, at the core of the work described here is the idea of a flexible design approach, which allows formality and, hence, precision, to be introduced gradually and selectively, as design and understanding progress. Thus, we may start off with an early informal model of a proposed design. This initial model may suffer from incompleteness and even inconsistency, but it may still be useful in helping designers gain an understanding of its properties. If this ambiguous model appears promising, we may then decide to apply formal approaches selectively, to help us gain confidence. This “mild” level of formality might reveal fundamental flaws in the design, at which point it may be amended or even discarded due to serious flaws. Over time and as the design is refined, the degree of formal checking can be gradually increased, ultimately reaching the fullest extent possible.

One of the advantages of such an approach is early detection of design flaws in what may have seemed as a promising approach. This is because the overhead involved in “full” formal validation is avoided. This overhead involves specifying the full set of details required to avoid incompleteness and inconsistency errors even of an early putative model is eliminated. On the other hand, by allowing selective application of formal checking, it allows designers to detect key flaws in those areas where they may be most uncertain. The end result is likely to be a faster path to the ultimate solution.

The paper is structured as follows. Section 2 discusses the technical context of our contribution. In Section 3 we present our view and positions on flexible visual modeling. In Section 4 we put forward the technical content of a recent tool extension. The paper closes with a short summary, some conclusions and future work.

2. Context

A key ingredient to a high level of productivity and product quality as promised by Model-Based Engineering (MBE), e.g., with UML [1, 2, 3], is computer-supported automation. But practical experience with current MBE tools indicates that we are still far from this ideal. Typically, tools are difficult to learn and use and are complex. Frustrating situations where the tools are forcing users into workarounds and constrained operating modes are frequent. This is contrary to free expression of ideas. For achieving effective tool support, the transition between an informal, provisional mode and a formal, precise mode is crucial. This contribution is a further step into that way of working with tools.

To enable a development process that includes an ability that is similar to informal diagrams sketched “on a napkin”, we are working on a flexible modeling approach [4, 5], which focuses on objects [6]. Starting with incomplete or even inconsistent UML object diagrams, we have developed an automated transformation of these into class diagrams, as a plugin for the USE tool [7, 8]. Both the object diagrams as well as the class diagrams use a flexible syntax which

comes close to an informal drawing tool while still following an internal meta-model. In the work process, developers have the option to incrementally create the object diagram while receiving feedback from the resulting class diagram. This paper extends on the technical side the previous work [4, 5] by handling whole-part relationships and inheritance as well as an improved handling for association multiplicities. The current extension of the USE tool has been applied for smaller teaching projects, in particular by students developing course projects; a systematic study is planned for future work. The paper also presents an extension on the conceptual side by summarizing the benefits of our approach and its potential to the development process.

Related approaches for flexible development of systems and transformations have been proposed: Related works on example based modeling include [9, 10, 11, 12]; flexible transformations, partly on an example focused basis, are [13, 14, 15]; uncertainty and partiality in modeling has been studied in [16, 17, 18, 19]. General dimensions of flexible modeling together with concrete application options are presented in [20]. The work in [21] discusses flexible typing. [22] concentrates on flexibility in domain-specific modeling. The Typing Requirements Models (TRM) in [23] permit a high degree of variability and flexibility for typing model transformations and lead to improved reuse options.

3. Positions on and Benefits of Flexible Modeling

Visualization techniques and methodologies: Our approach utilizes mainstream modeling visualization techniques with slight modifications and proposes a particular methodology for their application. Basically, we start from conventional UML class and object diagrams, and we extend them to what we call “imperfect” class and object diagrams. That means our class and object diagram do not follow strictly the conventional UML metamodel, but an extended one that allows incomplete and inconsistent diagrams. This opens in the work process to developers the option to let their ideas flow in a free way without having to obey conventional metamodel restrictions (e.g., “Attributes must have a type.”) and typical tool requirements (e.g., “Only class diagrams valid w.r.t. the conventional UML metamodel can be stored”). The principle of modifying and relaxing a language metamodel to allow for incompleteness and inconsistency can be applied to other UML sub-languages as well. For example, allowed UML operation call sequences that are abstracted to UML protocol state machines could be relaxed to “imperfect” UML operation call sequences (where, e.g., not all all calls of a fixed operation have the same number of parameters) and “imperfect” UML protocol state machines (where, e.g., not all operation calls have a corresponding operation in the class diagram).

Visualizing errors in models: In our approach we have implemented a particular way of handling incompleteness, inconsistency and incorrectness (we call them the three “incos”), as displayed in Fig. 1. Currently, the focus is on class and object diagrams, but the principles can be extended to other UML diagrams, or more generally to other kinds of models. *Incompleteness* is indicated by elements with a plus mark or by dashed elements in our object diagrams. In our class diagrams a question mark indicates an incomplete specification. *Inconsistency* is put forward by elements marked with an exclamation in our class diagrams. *Incorrectness* is presented with dashed elements in our class diagrams. The technical realization and the justification for the different kinds of representation will be discussed and become clear in the

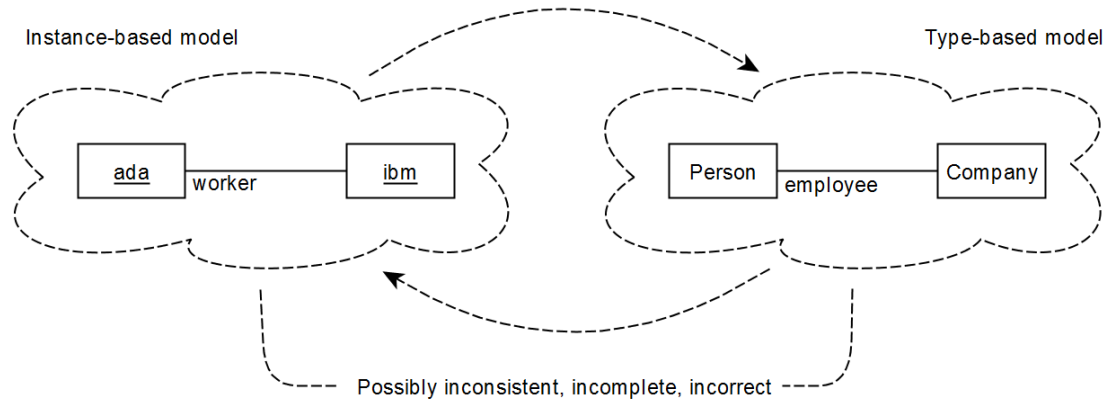


Figure 1: Idealized work process with imperfect instance- and type-based models.

following section. We emphasize that these additional language features (for the three incos) in our diagrams extend the conventional UML notation.

To our knowledge, the three “incos” have not been formally defined, and it is impossible to do so. Nevertheless, we want to state an informal explanation on how we view these three, overlapping notions. *Incompleteness* refers to the observation that an important aspect is yet missing in the model or description. *Inconsistency* expresses that there are at least two details in the model that contradict each other. *Incorrectness* comes in our view in two shades, namely syntactic and semantic incorrectness: Syntactic incorrectness means that the model does not meet its metamodel, and semantic incorrectness means that the model does not completely meet the real-world excerpt that it is intended to describe.

Collaborative development with human-in-the-loop: Our approach relies on model improvement through iteration: we start with an imperfect object diagram and from this we derive a first imperfect class diagram; by adding more possibly improved object diagrams or object diagram for further scenarios and through repeating the (object,class) transitions, ultimately a settled class diagram describing correctly all developed scenarios is achieved. As mentioned already, the (object,class) transitions could be generalized to instance-based artefacts alternated by type-based artefacts, e.g., by transitioning between example command sequences and protocol state machines.

Imperfect artefacts: In any case, our aim is to give developers the option to let their ideas flow in a free way with few creativity restrictions by a tool and the implicit steps in the work process. Flexibility may be gained by transitioning in the work process between specific, instance-based visual models and generic, type-based visual models where both kinds of imperfect models allow for the three incos: incompleteness, inconsistency, and incorrectness.

4. Whole-Part Relationships, Generalization, Multiplicities

This section discusses the newly designed and implemented tool functionality by means of an example. Figure 2 shows the formation of the (incomplete) output class diagram in the right side

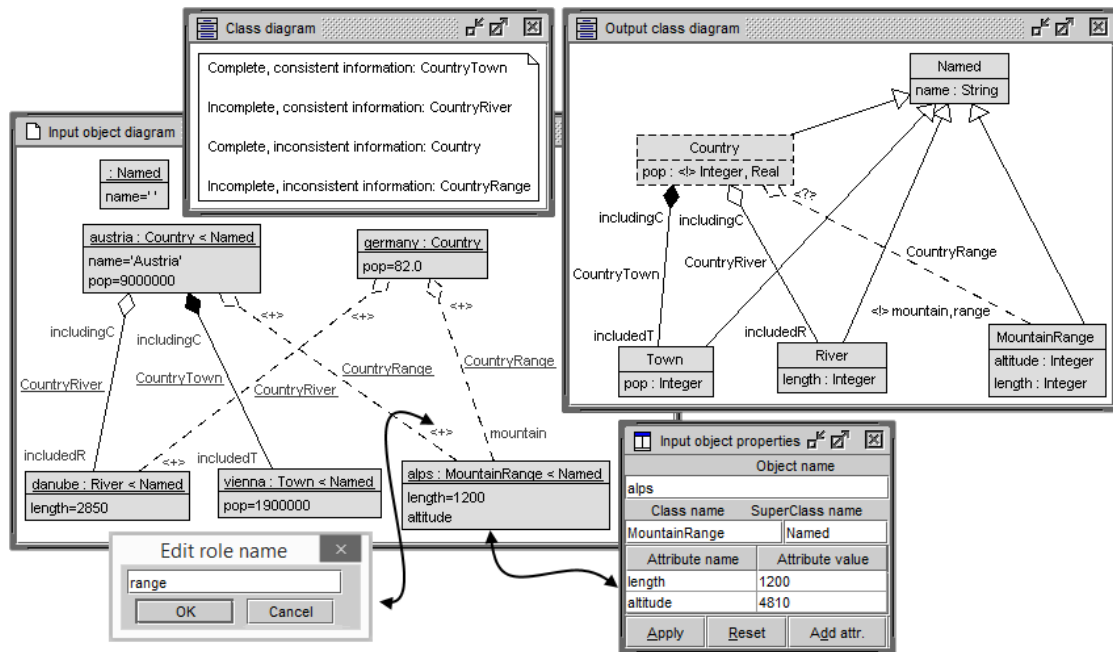


Figure 2: Utilizing Features for Whole-Part Relationships and Generalization

on the basis of the (incomplete) input object diagram in the left side. The input object diagram specifies incomplete objects (e.g., the attribute ‘name’ is present in the object ‘austria’ but missing in the object ‘germany’) and incomplete aggregation and composition links (e.g., role names are present for the left ‘CountryRiver’ link but are partly missing for ‘CountryRange’). In the input and in the output, graphical elements drawn using *continuous, solid* lines and contours represent fully specified entities, whereas those drawn using *dashed* lines and contours stand for entities with somewhat incorrect specification, i.e., incomplete or inconsistent information. The intention of our approach is that such incomplete, even inconsistent object diagrams may be used when new ideas and concepts are introduced into the models (e.g., typically in the early phases of the software development process). Developers should have the freedom to let their ideas flow in a *natural* way, even if their diagrams do not (yet) meet all formal requirements of the underlying modeling language or tool.

There are four cases w.r.t. available information in the object diagram for mapping objects and links to classes and associations (more specifically to aggregations and compositions).

Complete, consistent information: The composition ‘CountryTown’ can be completely derived (obtaining composition name and role names), given the complete and consistent object diagram information.

Incomplete, consistent information: The aggregation ‘CountryRiver’ can also be completely derived, although some links in the object diagram are only partially specified. The incomplete object diagram information can be matched against the more complete class diagram information.

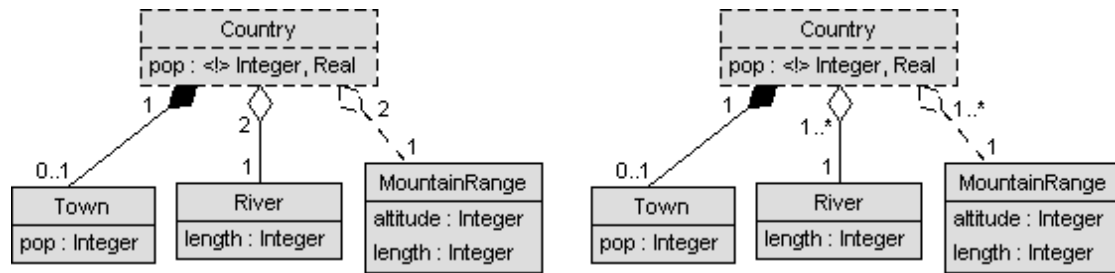


Figure 3: Exact and Standard Multiplicities

Complete, inconsistent information: The object diagram information for class ‘Country’ is complete, but inconsistently specified (contradicting attribute datatypes). This contradiction is highlighted in the resulting class diagram; i.e., the dashed rectangle of the class ‘Country’ flags it as a ‘to-be-improved’ element.

Incomplete, inconsistent information: The aggregation ‘CountryRange’ is incomplete, as role names on the ‘Country’ side are missing in the object diagram. The object diagram information is inconsistent because of mutually contradictory role names (‘mountain’ vs. ‘range’) on the other aggregation side. The aggregation ‘CountryRange’ is also flagged as ‘to-be-improved’, using the dashed aggregation link.

We have decided to use the same visual elements (i.e., for the ‘incos’ incompleteness, inconsistency, incorrectness) in the object and class diagram (or to say it more generally for the instance-based description and the type-based description). In the specific situation with deriving a class diagram from object diagrams, basically only the object diagram can be manipulated by the developer and the class diagram is automatically derived from the object diagram. In an even more flexible (but also more complicated) approach, both descriptions could be edited. To sum up in simple words, the plus (in the object diagram) and question mark (in the class diagram) stands for incompleteness, the exclamation mark for inconsistency, and the dashed elements for incorrectness resp. for items that still need more work.

The specification of *inheritance* in the object model is indicated by allowing the name of a superclass (as a type) in the name field of the object’s rectangle. The determination of attributes of the superclass is done by prototypical objects. For example, in Figure 2, there is one such unnamed prototypical object whose type is the superclass ‘Named’. This object has an attribute ‘name’, whose value is specified as an empty String value. Superclasses could also be identified (maybe in an even smoother way) by a refactoring process after a number of objects have been constructed. This is currently not possible in the tool.

The handling of *multiplicities* (as in Fig.3) has been improved compared to earlier versions of the tool. In the output class model there is now a new option to either use multiplicities as exactly stated in the object model (e.g. ‘0..1’ or ‘2’) or to use only multiplicities from a fixed collection of frequently applied standard multiplicities (‘0..1’, ‘1’, ‘0..*’, ‘1..*’).

5. Conclusion and Future Work

The contribution has shown how principles of flexible modeling can be applied and how an existing approach for flexible visual modeling can be extended. A central goal was the early detection of design flaws avoiding the overhead of “full” formal validation. Incompleteness and inconsistency are temporarily accepted through selective application of formal checking giving designers the option to detect key flaws in those areas where they may be most uncertain.

Much more work on other modeling aspects (e.g., transitioning from prototypical behavior models in the form of example command sequences to complete protocol state machines) remains to be done. The principle of transitioning between instance-based and type-based imperfect descriptions and models can probably be carried over to more modeling areas. In a collaborative modeling context, different objects and links from different developers could be presented and handled differently. Last but definitively very important, user studies must give more feedback about the applicability of the approach.

References

- [1] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [2] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed., Addison-Wesley, 2005.
- [3] B. Rumpe, *Modellierung mit UML*, Springer, 2004.
- [4] A. Kästner, M. Gogolla, B. Selic, From (Imperfect) Object Diagrams to (Imperfect) Class Diagrams, in: O. Haugen, R. Paige (Eds.), *Proc. 21th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS’2018)*, ACM/IEEE, 2018, pp. 13–22.
- [5] A. Kästner, M. Gogolla, B. Selic, Towards Flexible Object and Class Modeling Tools: An Experience Report, in: D. di Ruscio, J. de Lara, A. Pierantonio (Eds.), *Proc. 4th Flexible MDE Workshop (FlexMDE 2018)*, CEUR Proceedings 2245, 2018, pp. 233–242.
- [6] B. Selic, Career Award Talk, YouTube <https://youtu.be/9qPbGksB3d4?t=20m32s>, 2016.
- [7] M. Gogolla, F. Büttner, M. Richters, USE: A UML-Based Specification Environment for Validating UML and OCL, *Science of Computer Programming* 69 (2007) 27–34.
- [8] M. Gogolla, F. Hilken, K.-H. Doan, Achieving Model Quality through Model Validation, Verification and Exploration, *Journal on Computer Languages, Systems and Structures*, Elsevier, NL 54 (2018) 474–511.
- [9] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, J. de Lara, Example-driven meta-model development, *Software & Systems Modeling* 14 (2015) 1323–1347.
- [10] S. Maoz, J. O. Ringert, B. Rumpe, Modal Object Diagrams, in: M. Mezini (Ed.), *ECOOP 2011 – Object-Oriented Programming*, Springer Berlin Heidelberg, 2011, pp. 281–305.
- [11] D. Zayan, A. Sarkar, M. Antkiewicz, R. S. P. Maciel, K. Czarnecki, Example-driven modeling: on effects of using examples on structural model comprehension, what makes them useful, and how to create them, *Software & Systems Modeling* (2018).
- [12] D. Wüest, N. Seyff, M. Glinz, Flexisketch: a lightweight sketching and metamodeling

- approach for end-users, *Softw. Syst. Model.* 18 (2019) 1513–1541. URL: <https://doi.org/10.1007/s10270-017-0623-8>. doi:10.1007/s10270-017-0623-8.
- [13] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model Transformation By-Example: A Survey of the First Wave, in: A. Düsterhöft, M. Klettke, K.-D. Schewe (Eds.), *Conceptual Modelling and Its Theoretical Foundations*, Springer Berlin Heidelberg, 2012, pp. 197–215.
 - [14] T. Mens, P. Van Gorp, A Taxonomy of Model Transformation, *Electronic Notes in Theoretical Computer Science* 152 (2006) 125–142.
 - [15] W. Smid, A. Rensink, Class Diagram Restructuring with GROOVE, in: P. Van Gorp, L. Rose, C. Krause (Eds.), *Proceedings Sixth Transformation Tool Contest*, *Electronic Proceedings in Theoretical Computer Science*, 2013, pp. 83–87.
 - [16] R. Salay, M. Chechik, M. Famelis, J. Gorzny, A Methodology for Verifying Refinements of Partial Models, *Journal of Object Technology* 14 (2015). URL: <https://doi.org/10.5381/jot.2015.14.3.a3>. doi:10.5381/jot.2015.14.3.a3.
 - [17] O. Semeráth, D. Varró, Graph Constraint Evaluation over Partial Models by Constraint Rewriting, in: E. Guerra, M. van den Brand (Eds.), *Theory and Practice of Model Transformation*, Springer International Publishing, Cham, 2017, pp. 138–154.
 - [18] R. Salay, M. Famelis, M. Chechik, Language independent refinement using partial modeling, in: J. de Lara, A. Zisman (Eds.), *Fundamental Approaches to Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 224–239.
 - [19] M. Famelis, S. Santosa, Mav-vis: A notation for model uncertainty, in: *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2013, pp. 7–12.
 - [20] E. Guerra, J. de Lara, On the quest for flexible modelling, in: A. Wasowski, R. F. Paige, Ø. Haugen (Eds.), *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, Copenhagen, Denmark, October 14-19, 2018, ACM, 2018, pp. 23–33.
 - [21] A. Zolotas, N. Matragkas, S. Devlin, D. S. Kolovos, R. F. Paige, Type inference in flexible model-driven engineering using classification algorithms, *Softw. Syst. Model.* 18 (2019) 345–366.
 - [22] F. R. Golra, A. Beugnard, F. Dagnat, S. Guérin, C. Guychard, Using free modeling as an agile method for developing domain specific modeling languages, in: B. Baudry, B. Combemale (Eds.), *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, Saint-Malo, France, October 2-7, 2016, ACM, 2016, pp. 24–34.
 - [23] J. de Lara, J. D. Rocco, D. D. Ruscio, E. Guerra, L. Iovino, A. Pierantonio, J. S. Cuadrado, Reusing model transformations through typing requirements models, in: M. Huisman, J. Rubin (Eds.), *FASE 2017, Part of ETAPS*, volume 10202 of *LNCS*, Springer, 2017, pp. 264–282.