Models as documents, documents as models

Models as documents, documents as models

Perdita Stevens[1][0000−0002−3975−7612]

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh, UK
Perdita.Stevens@ed.ac.uk
http://homepages.inf.ed.ac.uk/perdita

Abstract. In software engineering, documentation and models are both broad concepts which attract varying approaches and attitudes. Moreover, as the title indicates, they overlap: in some circumstances, an artefact thought of as a model can serve to document a software project, while in others, an artefact thought of as a document can be manipulated by model-driven engineering tools just like any other kind of model. In this short paper we briefly explore these issues and provide pointers to some of the relevant literature.

Keywords: documentation · programming · modelling

1 Introduction

What most software developers today think of as a "document" in software development, and what they think of as a "model", bear little resemblance to one another. The canonical document in today's practice is word-processed; it has a narrative structure, being intended to be read, at least initially, from beginning to end; and its intended consumers are humans, not machines. The canonical model is developed in a drawing tool or a specialised modelling tool, or perhaps is sketched on a whiteboard; it is graphical, making essential use of a two-dimensional canvas, and its elements are often capable of being apprehended in many orders; and depending on its form, its intended consumers may be tools, as well as humans.

Unfortunately, in writing that opening paragraph, I have already made steam start to curl out of the ears of some readers, whose ideas of "document", "model", or both, are different from the ideas I am intending to invoke. A reviewer asked for precise definitions of the terms: while on the surface the request is reasonable, I fear that it would be sterile to attempt to meet it. When we ask ourselves what documents and models are, and – crucially – what they are for, we find that there is a great overlap between the two concepts; indeed, both the positions "every document is a model" and "every model is a document" will turn out to be arguable.

This short paper, intended to accompany an expository talk, does not purport to present original research. Instead, we explore the similarities and differences between the connotations of the terms "document" and "model", and

draw attention to some of the work that has already been done making use of the connections between the two concepts. Further pointers and comments would be very welcome.

2 The purposes of software modelling

Modelling has been used within the process of software development, since the earliest days, as a means of controlling the information overload that otherwise prevents human software developers from effectively making decisions about how to build the software (see [17] for one interesting early example). Indeed some of the models that have been most commonly used with software pre-date the idea of computer software: flow charts, for example.

The key point about a model is that it presents all of, and ideally only, the information that is necessary for some particular purpose. Although this has arguably been the case since the earliest days of graphical modelling, it has become more important as the increasing scale of software development has made separation of concerns essential: key decisions must be taken safely, without requiring the decision-maker to understand everything about the software. (We will return to the implications of this for automation momentarily.) The purpose might be the support of any software engineering process, including requirements management, architecture, detailed design, verification, etc. For example, a flow chart (or activity graph) might show how one important process is to be carried out by the software, but abstract away from how the software is structured. On the other hand an architecture diagram might show the high level packages into which a large software system is split, without giving any information about the behaviour to be coded in each package.

When software developers talk about "models", we should usually understand a *graphical* representation of some aspect of a software system, perhaps placed within its environment. Latterly, especially with the formalisation and codification of techniques such as metamodelling for defining and manipulating models, it has become clear that it does not matter very much whether a model is graphical or textual: this is a matter of concrete syntax, and it is often useful to separate concrete syntax conceptually from abstract syntax and from semantics. This observation is a key part of what makes "model" and "document" overlap.

Models were originally used in software development for informal communication purposes between humans. The idea that they could be formal was relatively late to arrive: in 1976, Chen felt the need to emphasise that a graphical representation of an entity-relationship model could be isomorphic to a symbolic one [4]. This conception of graphical models as essentially informal is natural given many other human relationships with pictures: originally, and often still today, a model is seen as an informal aid to understanding of what the corresponding formal artefact, the code which is actually used to instruct the computer, should do. However, there are advantages to making a graphical notation precise, and indeed, once it is normal for a software development to involve separated representations of different concerns, and for there to be nobody who can understand

every detail simultaneously, it becomes almost essential to have tool support for the tasks of relating models to one another and to the code. Once the code is materially affected by *precisely* what is in the model – and it no longer matters what the human who drew the model hand-waved while drawing it – the model has become, in a certain sense, a formal artefact. Thus we reach modern conceptions of model-driven development, in which models are related by model transformations and code is (in part or in whole) generated from models, and related approaches such as language-driven engineering [15].

3 The purposes of software documentation

Documentation, broadly conceived, serves a number of processes within software development [1]. We will limit scope to documents that relate somewhat directly to the software – they describe or constrain its usage, structure, functionality, or development.

For us, then, a document in some sense *specifies* the software or part of its structure or function. Typically it records the result of some software engineering process, more than supporting the doing of that process – even if its purpose is then to support a downstream process. It may support:

- communication within the software project, e.g. clarifying the API of a component to both implementors and users of the component
- verification, validation and testing, e.g. when what software actually does is compared with what a document says it should do
- maintenance, e.g. by explaining to a future human developer what decisions have been taken by the original developers and why
- use of the software, e.g. by explaining to a human reader how to interact with the software to achieve some aim
- litigation, e.g. demonstrating that the software developers have not delivered what they were contractually obliged to deliver.

4 What counts as a model?

We have already mentioned that models may be formal or informal, graphical or textual. The reader may reasonably wonder what would *not* count as a model and indeed the author has often proclaimed "everything's a model!". For our purposes today, a model is an artefact which records all, and preferably only, the information necessary for some decision-making purpose relating to a software development.

As is by now apparent, in this paper "model" typically indicates a prescriptive model of the kind used in model-based or model-driven development, rather than a descriptive, mathematical model. The distinction between prescriptive and descriptive models can become blurred – after all, if a prescriptive model is correctly followed, so that it prescribes something about a resulting system, we expect that that same model now describes something about that system,

i.e., can now be regarded as a descriptive model. However, the primary purpose of models within model-driven development is prescriptive: descriptions don't drive things!

5 What counts as a document?

The prototypical "document" in software development is written using a word processor, e.g. Microsoft Word. The term "documentation" has broader connotations, and might include, for example, the in-program "Help" text a user can access. For purposes of this paper, key characteristics are:

- a document is intended to be read by humans, typically by a predicted class of humans with predictable background knowledge, e.g., developers, users;
- a document is relatively long-lived, typically having a lifetime that is commensurate with that of the software to which it relates; we would normally consider the minutes of a design meeting to be too ephemeral to count as documentation, for example.

Within these bounds, there are possibilities other than the prototypical Word document, differing especially in how long-lived human-readable text is structured and perhaps entwined with other software artefacts such as code. For example, comments within the code might well be considered to be documentation, especially if, as with JavaDoc, they are structured for extraction. By considering JML text embedded within programs, and indeed "self-documenting code", where code is considered to be so clearly written that it does not require comments or other form of documentation, we see the blurred edges of this concept: a key point is that not all humans are the same and so "human-readable" often has to be qualified with which humans are the intended readers.

6 Models as documents

Martin Fowler's seminal classification of the uses of UML as sketch, as blueprint, and as programming language [10] gives a suitable setting in which to discuss when models can be seen as documents. Within this setting, it is the second mode, models as blueprints, which is most convincing.

When models are used purely informally, as sketches, for example on a whiteboard, they will fail our second criterion for documents. Being produced to support a particular discussion, they will be ephemeral.

On the other hand, when a modelling language like UML is used as a programming language – by which Fowler meant, when the model becomes fully detailed *and* is used as input to tools – the model may fail our first criterion. The model is to be read by a tool, and there is a danger that in order to make it amenable to being read by a tool its readability by humans is reduced. We say "may" because the premise of model-driven engineering is that this is not the inevitable result of using models that can be processed by tools. It is important

to note, here, that Fowler was writing not about all modelling languages but specifically about UML, in a context where UML2.0 was emerging.

The middle use, models as blueprints, gives "a way to express software designs in such a way that the designs can be handed off to a separate group to write the code, much as blueprints are used in building bridges" [9]. The model documents the intended design, and the document is then read by the human programmer. The models thus produced might, indeed, be printed in the specification documents alongside explanatory text (this was a standard part of the ISO9000 process of the organisation the author worked for in the early 1990s). The problem is that one may end up with what John Daniels memorably called "the Great Corporate Data Modeling Fiasco" [6] – a model which contains so many details that it is readable, and maintainable, by nobody and nothing. The lesson here is that some models need to be more than documents: only rather simple software engineering artefacts can sensibly be long-lived, if they are only readable by humans and not by tools.

7 Documents as models

The idea that documents can usefully have structure which can then be exploited to use the document effectively is an old one. In the W3 standard, DOM stands for "document object model" although use of this standard is usually within the context of a program or model, rather than anything normally thought of as a document. Files following the DOM format are not easy for humans to read (or write); in that sense they are not really documents in the sense we have laid out. Readers may also think of the standard DTD, "document type definition", which preceded XML schema as a way to specify how an XML document should be structured. This, similarly, uses "document" in a broad sense, more usually applied to artefacts we would think of as models than to documentation.

Already a decade ago, Erissson, Wingkvist and Löwe were concerned about the variable quality of traditional technical documentation, which they found frequently to involve many clones (or near-clones) of text passages, leading to maintenance difficulties and error-prone-ness. Their paper [8] exploits metamodelling techniques to propose a software infrastructure for assessing and improving the quality of technical documentation.

Other work in this vein, which starts with a document, and produces from it a descriptive model for purposes of analysis or remediation, starts to stray out of our declared scope, but is nonetheless interesting. Before leaving behind this field we mention work on the modelling of contracts [3] and on privacy conditions of websites [16, 7]. However, to pursue this avenue would take us too far afield, into natural language processing.

Returning to central documentation of software, we should consider the OpenAPI standard (also known as Swagger). OpenAPI descriptions document application programmer's interfaces of components in standardised, plain text which, it is claimed, can be easily read and written by human developers using any text

editor. A key part of how OpenAPI is marketed is that an API description in this format can also be used as input to a tool.

The team contrast "code first" and "design first" approaches and advocate the latter: that is, the idea is that the OpenAPI document should be developed first, before the code that implements the API. There is a small academic literature amplifying this; David Sferruzza is prominent among researchers who have investigated using OpenAPI descriptions as sources from which to generate web services [13, 14].

A problem is that OpenAPI API descriptions often fail to follow the format properly (e.g. as analysed by Ralphson [12]) or include content that is inadvisable from a security point of view [5]. It might be argued that any editing that a human is allowed to do, they will do: because the files are designed to be readable, and writeable, by unassisted humans, such errors creeping in is unsurprising. The same could be said about ordinary programs, which are typically built in a text editor; but the compiler or interpreter normally catches such problems, and programmers are used to this. The lesson here is that blurring the line between human- and machine-readable text increases the risk of failing to achieve either.

An alternative approach is to make more central use of artificial intelligence to bridge the gap between a specification that is natural for a human to write, and an implementation; a recently famous example is GitHub Copilot[1], but the problems of naively adopting this technology have been well rehearsed [2, 11].

8 Conclusions

In this short paper we have suggested that there are overlapping intentions and purposes of documents and of models, which have more in common with one another than either does with programs.

What implications does this have for the future of programming? Most importantly, researchers and practitioners concerned with either models or documents should pay heed to advances that concern the other class of artefacts. There is, for example, scope for model-driven engineering to pay more attention specifically to documents, and to how documents interrelate with "other models".

Acknowledgements

I thank the anonymous reviewers for insightful comments, questions and pointers to relevant literature.

References

1. Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: the practitioners' perspective. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE*

[1] https://copilot.github.com/

'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pages 590–601. ACM, 2020.
2. Tim Anderson and Katyanna Quach. Github copilot auto-coder snags emerge, from seemingly spilled secrets to bad code, but some love it. https://www.theregister.com/2021/07/06/github_copilot_autocoder_caught_spilling/, July 2021.
3. John J. Camilleri and Gerardo Schneider. Modelling and analysis of normative documents. *J. Log. Algebraic Methods Program.*, 91:33–59, 2017.
4. Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
5. Cybersprint. Swagger API: Discovery of API data and security flaws. https://www.cybersprint.com/blog/swagger-api-discovery-of-api-data-and-security-flaws, December 2020.
6. John Daniels. Modeling with a sense of purpose. *IEEE Softw.*, 19(1):8–10, 2002.
7. Michiel de Jong, Jan-Christoph Borchardt, Hugo Roy, Ian McGowan, Jimm Stout, Suzanne Azmayesh, Christopher Talib, Vincent Tunru, Madeline O'Leary, and Evan Mullen. Terms of service; didn't read. https://tosdr.org/, 2011-date.
8. Morgan Ericsson, Anna Wingkvist, and Welf Löwe. The design and implementation of a software infrastructure for IQ assessment. *Int. J. Inf. Qual.*, 3(1):49–70, 2012.
9. Martin Fowler. UmlAsBlueprint. https://martinfowler.com/bliki/UmlAsBlueprint.html, May 2003.
10. Martin Fowler. UmlMode. https://martinfowler.com/bliki/UmlMode.html, May 2003.
11. Jeremy Howard. Is github copilot a blessing, or a curse? https://www.fast.ai/2021/07/19/copilot/, July 2021.
12. Mike Ralphson. What we learned from 200,000 OpenAPI files. https://blog.postman.com/what-we-learned-from-200000-openapi-files/, August 2021.
13. David Sferruzza. *Plateforme extensible de modélisation et de construction d'applications web correctes et évolutives, avec hypothèse de variabilité. (Towards an extensible framework for modelling and implementing correct and evolutive web applications, under variability hypothesis).* PhD thesis, University of Nantes, France, 2018.
14. David Sferruzza. Top-down model-driven engineering of web services from extended openapi models. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 940–943. ACM, 2018.
15. Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. Language-driven engineering: From general-purpose to purpose-specific languages. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 311–344. Springer, 2019.
16. Shomir Wilson, Florian Schaub, Aswarth Abhilash Dara, Frederick Liu, Sushain Cherivirala, Pedro Giovanni Leon, Mads Schaarup Andersen, Sebastian Zimmeck, Kanthashree Mysore Sathyendra, N. Cameron Russell, Thomas B. Norton, Eduard H. Hovy, Joel R. Reidenberg, and Norman M. Sadeh. The creation and analysis of a website privacy policy corpus. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

17. Frank W. Zurcher and Brian Randell. Iterative multi-level modelling. A methodology for computer system design. In A. J. H. Morrel, editor, *Information Processing, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5-10 August 1968, Volume 2 - Hardware, Applications*, pages 867–871, 1968.