Diversity in UML Modeling Explained: Observations, Classifications and Theorizations

Michel R. V. Chaudron[1(✉)], Ana Fernandes-Saez[2], Regina Hebig[1], Truong Ho-Quang[1], and Rodi Jolak[1]

[1] Chalmers | Gothenburg University, Gothenburg, Sweden
michel.chaudron@cs.gu.se, {regina.hebig,rodi.jolak}@cse.gu.se, truongh@chalmers.se
[2] University Castilla La-Mancha, Ciudad Real, Spain
AnaMaria.Fernandez@uclm.es

Abstract. Modeling is a common part of modern day software engineering practice. Little evidence exists about how models are used in software development and how they help in producing better software. In this talk we introduce a classification-matrix and a theoretical framework that helps explain the large variety of models and modeling styles found in industrial practice. As part of this explanation, we will explore empirical findings on the uses of UML modeling in practice. We intersperse this paper with some insights about modeling in software development that may be common to some, but certainly not generally accepted throughout the software engineering community.

1 Introduction

There exists a large variety of modeling languages in the field of software engineering. These range from languages for modeling user interfaces, business processes, data-exchange formats, and software designs. In this paper we focus on the use of UML in the modeling of the design of software systems. The UML language has emerged in the mid-1990's after a phase in which many software design notations existed. Often each of these design notations was proposed in conjunction with a software design method. The naissance of UML was no different: it came together with an object-oriented design method. However, nowadays, UML is considered mostly a notation. Ever since its introduction, the use of UML in software development has been subject to (almost religious) debate. In this paper we aim to contribute to clarifying the field of modeling by explaining different type of approaches to modeling.

The structure of this paper is as follows: First, we describe models as they can be found in current software development. In order to understand the differences found across such models, we present two classifications based on different distinguishing characteristics. Next, we discuss different purposes of models in software development, and explain that the different ways of modeling can be understood by recognizing different goals and contexts of different projects. Then, we reflect

A M. Tjoa et al. (Eds.): SOFSEM 2018, LNCS 10706, pp. 47–66, 2018.
https://doi.org/10.1007/978-3-319-73117-9_4

on some insights and findings from empirical studies into modeling. Finally we discuss selected future directions.

We intersperse this paper with some propositions that highlight insights about modeling in software development that may be familiar to some, but are certainly not commonly accepted throughout the software engineering community.

2 Classifications of Software Models and Their Uses

Nowadays, we have come to realize that software modeling (using UML) is done in a large variety of ways. Indeed various terms are used to suggest different ways of using models in software development: model-driven sw development, model-based sw development, model-based engineering, model-centric development. Unfortunately, there is no common agreement on the meaning or characteristics of these terms. This has o.a. led to the running (and publishing) of survey studies that lump together every respondent that says that they do 'model-* development'. Yet, in order to properly perform and interpret scientific studies on modeling in software development, we need a way to precisely define the object of study. Based on the empirical studies from the last decades, we next propose multiple classifications for characterizing UML modeling and their uses in software development.

2.1 A Classification of Models by Abstraction Level

In this section we illustrate how modeling can be classified by looking at the abstraction level of the system that they aim to capture. We recognize the following levels of abstraction:

- A-type: Architecture modeling
- D-type: Design modeling
- I-type: Implementation modeling.

We give a brief characterization of each of these approaches:

Architecture modeling targets a high level of abstraction of the system. Following [1], architecture targets the overall structure and behaviour of a system as defined by the components, their relations and their interactions. Also, as part of the architecting activity, a model is used to assess whether the design meets the extra-functional[1] requirements of the system. An architecture is typically defined in terms of the main system components and layers. Rarely do architecture designs include actual mention of classes, methods or attributes. At the abstraction level that they target, architectures aim to be complete in the sense that all important components are included in the model. For completeness sake, one could distinguish two different levels of architecture: (i) software/system-architecture, and (ii) enterprise architecture - which can be seen as

[1] Also known as non-functional.

systems-of-systems abstraction. UML can be used for the enterprise architecture level, but we will not make this distinction in this paper.

In Design modeling, there is a medium abstraction of the implementation of the system. The design level model of a system is typically represented in terms of classes (or components or packages) and the relations between them. For some classes, details such as methods and attributes are defined. Models at the design-level of abstraction typically focus on important parts of the system. Importance is relative to the producers and consumers of the model, but generally is driven by importance and risk (see [18]).

For implementation modeling, there is a close correspondence between the system model and the implementation: In principle, every class in the model can be mapped onto one of more classes (or other artifacts) in the implementation. Hence, because the model mirrors the implementation, the model must be complete.

Some projects use modeling at all three levels of abstraction. Modeling at more than one level of abstraction introduces the challenge of keeping the models at different levels consistent with each other.

An insightful diagram (Fig. 1 about the spectrum of approaches to the use of modeling in software development was presented by Brown [2]. His spectrum is organized around different types of key uses of models in a project. The top row in the original paper only stated 'Model' in all boxes. The use of the same term 'Model' across all boxes is actually a bit misleading. Based on our studies of the use of models, we have come to understand that the models for the types of use suggested in the diagram are quite different. Hence, we have added letters 'A' (Architecture), 'D' (Design), and 'I' (Implementation) to indicate that the models in the types of use suggest are typically of various levels of abstraction.

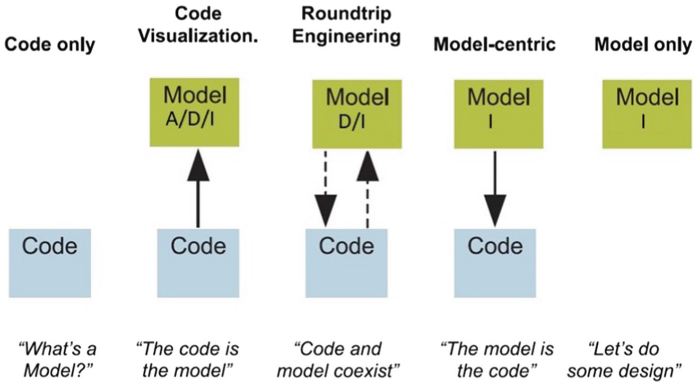

Fig. 1. Spectrum of modeling approaches by Brown [2] - Annotated

2.2 A Classification of Models by Stage of Development

In this section, we explain a complementary classification of models in software development from the perspective of the stage of development for which they

are used. The scientific field of Design recognizes several stages of a design in the process of developing a product [4]. In the context of software development, we state these stages and their use of models as follows. We illustrate these stages in Fig. 2[2]:

- Ideation/Conceptualization: The main objective of this step is to create a concept of the system to be created. This is one of the most creative and synthetic steps in the design: it requires the exploration, formation and combination of ideas.
- Externalization: The main objective of this step is to construct an external/explicit (as opposed to internal (to the mind of the designer)/tacit) representation of the system to be built. This representation serves as a vehicle for achieving shared understanding in a team/organization, and as persistent reference for a complicated abstraction that cannot be maintained in the memory of the engineers.
- Production/Implementation: In this stage, the system is actually being constructed. The model of the system is used to produce specifications of the parts that need to be constructed, as well as recipes on how to assemble the parts. In software engineering, models can indeed be used to generate (parts of) the implementation.

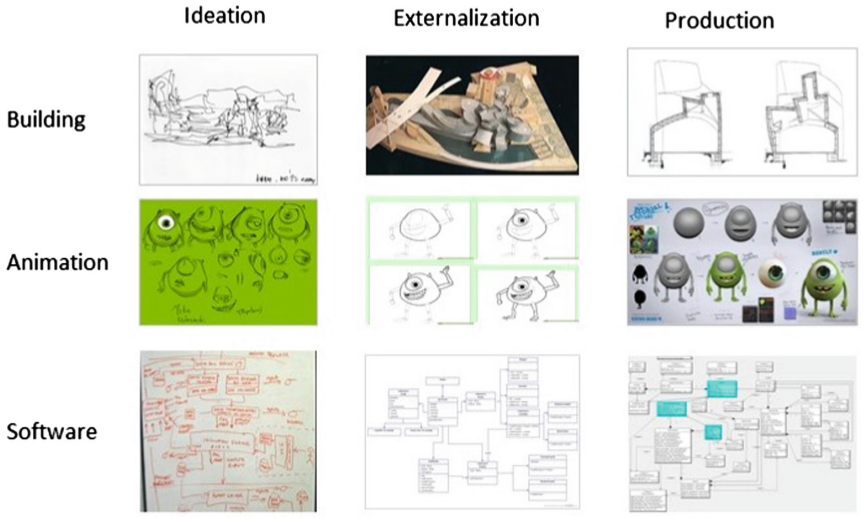

Fig. 2. Models in different stages of development

Ideation or conceptualization in software development is commonly done by sketching on a whiteboard or on a piece of paper. At this stage, the syntax of

[2] Images of Bilbao Guggenheim Museum (c) by Gehry, and Mike from Monsters Inc. (c) by Pixar.

the actual representation is not considered critical. Presumably, this is because the people involved in the ideation share the same room, hence can clarify issues by talking to each other. Ideation sessions tend to range on a timescale of tens of minutes to a few hours. This ideation effort is independent of the size of the system.

Externalization can be done in different ways. The quick and dirty way is to take a picture of the drawing on the whiteboard using a smartphone and then store the image in the project repository. The next step up in rigour is to create a design using a generic drawing tool, such as Powerpoint, or Visio. The advantage of generic Office tools is that the resulting diagrams can be easily integrated into overall 'Software Architecture Design' (SAD) documents that typically are a mix of text and diagrams. The most rigorous representations are made using a UML-CASE tool. Such representation supports basic forms of version management, but are considered a bit more complicated to integrate with word-processors for creating SAD documents. Using a CASE tool to create a UML model for a modest system can take a few hours, while creating a detailed UML model for a complex system can be a matter of days.

For using models in the production of software, the models need to be complete in the sense that they cover all of the implementation functionality and also in strict conformance to the syntax of the modeling language so that a compiler/code generator can produce implementation code. Creating such models requires dedicated CASE tools and (as they represent the main implementation activity) can take a large part of the effort of the overall project (say 30–40%).

A key difference between on the one hand the ideation and externalization stage and on the other hand the production stage, is that in the ideation and externalization stage, the main consumer/audience of the models are people, whereas for the production stage computers are an essential consumer of the models - see Fig. 3. Aiming for a computer as consumer requires that models are specified following a rigorous syntax and semantics. The fact that humans are the audience of models can be used to tailor the approach to modeling

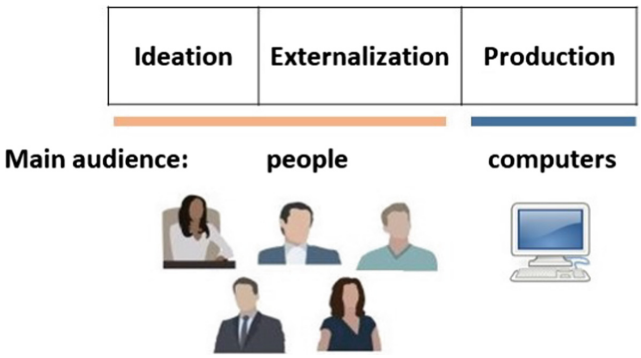

Fig. 3. Main audience of models in different stages of design

to the audience's needs: One best practice observed in industry is to test a 'design model/document': before committing a document as 'stable', the document should be reviewed/tested by the consuming party/parties.

Proposition 1. *When created wisely, design models (and by generalization: documentation) are consulted much more often than that they are created/modified. For the 'consumption' of models to work well, producers and consumers of models should agree (from early on in a project) on representation (detail, conventions for naming and layout), organization (layering), and conventions for navigation in- and searching for models.*

2.3 Syntactic Characterization of Software Design Models

The previous sections have introduced two key dimensions for classifying models. In this section, we will introduce some characteristics by which models differ from each other. We see these characteristics as mostly syntactical, and also more as a resultant of the dimensions 'abstraction' and 'development stage' than as additional angles by which to classify models. Table 1 shows an overview of these characteristics. Detail of a models can be seen by the amount of aspects of elements that are represented in the model. For example a class can be represented only by a rectangle with a class name (which would be low detail). Alternatively, a class can additionally be represented by attributes and methods. The latter can have public/private attributes, signature with typing. Using all these aspects represents a class in a high level of detail. Nugroho et al. introduced a metric for level of detail for UML models in [19]. Using this metric, this paper shows that a higher level of detail in sequence diagrams correlates with a lower defect density in the implementation of the corresponding classes.

Rigour refers to the degree to which a representation conforms to a formal syntax. A low conformance to formal syntax is common in the ideation stage. However, also in industrial SAD documents we frequently find that the design diagrams are enhanced by 'free format' shapes and icons which are not part of the UML syntax. We call a low adherence to a formal syntax 'sketchy'.

Table 1. Syntactic dimensions of software design models

Dimension	Description of range
Detail	A model can be represented in low detail or very high detail
Rigour	A model can precisely follow the syntax of the language or largely ignore the syntax (e.g. sketchy) (even mixed levels of rigour are common)
Completeness	A model can focus on representing key parts only or can be a complete mirror-image of the implementation
Consistency	A model can be consistent or contain many inconsistencies

Completeness refers to the degree to which all parts of the system are represented by the model. From the work of Osman [20] we know that UML models (made as part of forward design) contain only between 50% and 10% of the classes of the corresponding implementation (See Fig. 4. Moreover, we know from [17] that designers focus on parts of the system that is complex and critical, hence follow a risk-driven approach to choosing which information to include and leave out of a design.

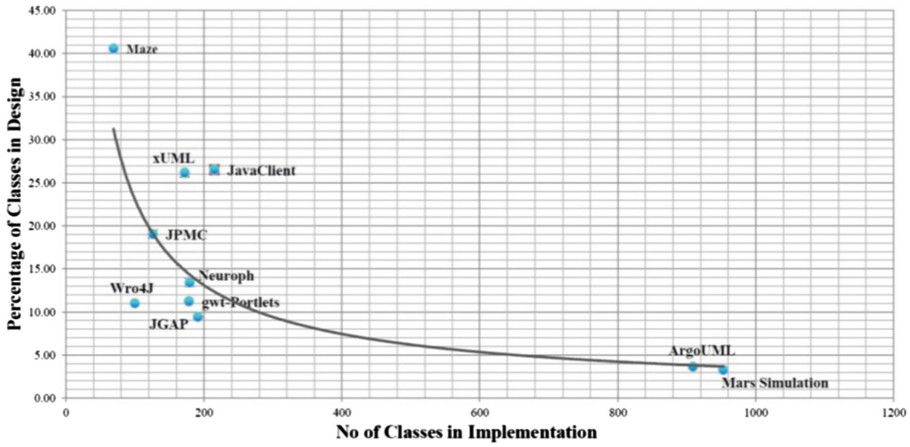

Fig. 4. Ratio of #Classes in UML design vs #Classes in implementation (from [20])

Consistency refers to the degree of intrinsic consistency in the model. The issue of consistency arises mostly from the fact that UML supports multiple types of diagrams that are logically linked to each other through reference to the same classes (and states). The problem of inconsistency has been identified in the early 2000's. A recent mapping study aimed to establish a definitive collection of consistency rules [22]. As part of an overall research program that aimed at assessing quality of UML models, Lange et al. [13] describe an empirical study in which they show that the amount of inconsistencies that exists in UML models of a few industrial case studies is very high. Partially this is due to the fact that incompleteness of a model can often also be interpreted as an inconsistency, and we know from our aforementioned empirical study on completeness of UML models [17] that designers leave out many parts of the implementation. In a follow-up experiment [14] we found that inconsistencies in UML models increase the divergence of interpretations of the models and thus increase the risk of various mistakes.

2.4 A Classification of the Uses of Software Design Models

At first the huge diversity of types of models found across industry puzzled us. Was there not one right way or best way to do modeling in software projects? In

the previous section we have already explained that there are different project settings that drive modeling practices. In addition to those, we explain in this section that design models are used in support of many different activities. Figure 5 shows an overview of different activities that have been reported in various industrial case studies to use UML models (see e.g. [6] as a starting point).

	Generic		Management							Analysis & Design			Implementation		V & V		...
	Overview	Understanding	Planning	Progress Monitoring	Cost Estimation	Risk Management	Compliance / Certification	Coordination - Standardization	Communication – Knowledge Sharing	Ideation	Analysis - Domain	Analysis – XFP	Prototyping	Code-generation	Traceability	Testing	
Architecture Modeling	X	X	X	X	X	X	X	X	X	X	X	X			R->D		
Design Modeling	X	X	X	X		X	X	X	X	X	X				R/A->I		
Implementation Modeling				X			X		X				X	X	D->I	X	

Fig. 5. Uses of design models in software development

We have classified the uses into several global categories: Generic: 'create overview' and 'understanding': these apply to all types of models. There is a surprisingly large number of project management type of activities that are supported by design models:

- *Planning:* a design model allows to split the work in parts and delegate these to different teams/developers.
- *Progress monitoring:* a design model can be uses to track progress by providing an overview of the progress of individual components, or - at a higher abstraction level - by showing which components have been completed.
- *Cost estimation:* similarly to planning, the fact that a design model provides a breakdown of the system into components, allows the estimation of costs for parts which can then be used to estimate cost (and schedule) for the entire system.
- *Risk management:* a design model makes explicit, and helps discover, which components are needed in a system, this in turn triggers discussion about possible risks that may arise in the construction and composition of components into the overall system.
- *Compliance:* One typical use of design models is to use them to verify that the implementation indeed conforms to the design. When no design model exist, there is a higher risk of 'drift' in the implementation. Additionally, models can be used to verify that particular policies are integrated in the system

(such policies exist in the banking-domain); alternatively, some domains (e.g. medical, automotive) ask that certain models are constructed and used for analysis of critical properties of the system.
- *Coordination/standardization:* For teams that work across multiple locations, it is important that a common standard on how to handle the design and implementation is available. Design models play such a role.
- *Knowledge sharing:* modeling a system is a way of capturing knowledge about a system. Through its representation this knowledge can be shared in a development team.
- *Ideation:* Ideation is the formation, exploration and combinations of ideas. In the case of software, these apply to the design (and analysis) of a system. Having an explicit model serves as an aid in inventing ideas and exploring new directions.
- *Analysis (XFP):* a design model can be used for various types of analysis of the system: ranging from more qualitative 'what if' scenarios (e.g. about maintainability) to quantitative analysis of extra-functional properties such as performance, reliability, safety and others.
- *Prototyping:* design models may be (partially) executable and can hence by used to demonstrate and try out how the system will work.
- *Code-generation:* models of the system are essential for code-generation. The main objective of this, is to increase the overall development speed of the project.
- *Traceability:* design models provide an intermediate abstraction esp. between requirements and the implementation. As such design models can act as a pivot point and aid in establishing traceability between requirements and the implementation.
- *Testing:* models can be the basis for specifying and prioritizing tests.

Figure 5 shows that that are many uses of design models and that these uses serve different stakeholders in software engineering projects. Indeed, some of these uses are secondary or by-catch of other more important uses of design models. So, the use of design models should not be seen as exclusive to one purpose. Moreover, the main purposes of a model change during the execution of a project. We will elaborate this theme in Sect. 4. We summarize the findings on the multiple uses of models through the following propositions:

Proposition 2. *Models of software designs serve a multitude of purposes in software development projects.*

Proposition 3. *In software development projects, the purposes of models of software designs change focus over time.*

Proposition 4. *The value of models in achieving the goals changes over time.*

Proposition 4 applies to various goals, but we will explain it using one example that is illustrated by Fig. 6. Figure 6 depicts the utility of documentation (as a generalization of models) as a function of the experience of developers. For developers that are new to a system, the documentation is of much value/utility

because it helps them understand the system which they need in order to do their work effectively. However, as developers work for longer time on the same system, they build up in their working memory an understanding of the system. Hence, the value of the documentation becomes less to them (while the documentation itself has not changed - only the context has changed!).

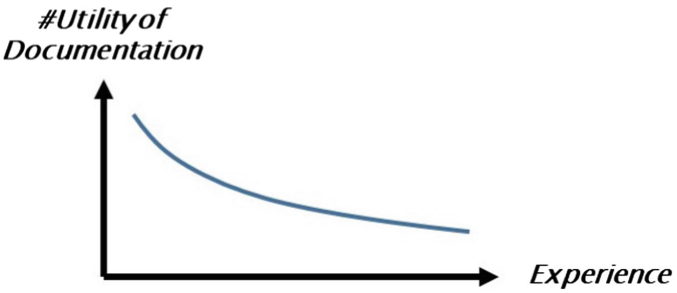

Fig. 6. Utility of documentation over time

Thus the purpose of models is a moving target. Clearly, this complicates finding empirical evidence for effectiveness of modeling because this has to be assessed relative to the purpose.

Interestingly, when going over the list of uses of models, there are only few uses for which models are indispensable. For most other uses, alternative approaches can be used. Clearly this is unlike the inevitability of producing implementation-code of systems. Indeed, in general models are a means to an end - the end being: the efficient development of (quality) software. This 'weakens' the commitment to modeling. And if alternatives work better than modeling, then projects are indeed better served with such alternatives. Possibly a good metaphor for the use of models is that they function as lubricant: they make many task run more smoothly.

Proposition 5. *There are many uses of models that do not directly follow from the main goals for using modeling.*

Various surveys have explored the main goals of using modeling approaches in software development projects. The commonly mentioned goals are: reduce development time/increase productivity/agility/velocity, improve quality (of code and of design), improve efficiency/reduce cost. When looking at the uses of models in Fig. 5, then there is not a very direct contribution of the uses of models to the aforementioned goals. Again through its diversity of uses, modeling has contributions to many goals in many ways. Empirical evidence regarding the 'effect-size' of modeling is still elusive. We point to two attempt at collecting evidence on the effectiveness of modeling: In [3] Chaudron et al. propose a theory that offers a causal explanation of the impact of UML modeling on quality and productivity. Some steps in this chain are supported by evidence from research

papers, for other causal steps no empirical evidence is mentioned. For a more general perspective, Garousi et al. provide studies into the factors that affect the use and usefulness of documentation [7]. Their study culminates in the formulation of costs and benefits of technical documentation and a theory (meta-model) for the quality of software documentation [24]. However, their study does not look at the process of the use of documentation over a development project. In the next section we propose a theory that explains the different types of modeling found.

3 A Theory for Explaining the Plethora of Approaches to Modeling

In this section, we propose a theoretical framework that captures the insights from the previous sections that modeling practices are linked to project goals. Our theoretical framework is shown in Fig. 7. In this diagram 'SE' stands for Software Engineering. The interpretation of the framework is as follows: Projects happen in a context, have stakeholders and can be in a particular stage of development. Context may include may facets (See e.g. [5]). For example, one can think of: risk-propensity of the organization, available time/money, organizational culture (e.g. [11]), but also size and geographic distribution. In practice many more factors of the context may play significant roles. Stakeholders have goals, such as increase development speed, a particular quality-level of the final product and so on. The goals may change across the stages of execution of a project. These goals of the stakeholders drive the development process used and the practices used in the overall approach to SE. A process denotes the collection of (formalized) steps of tasks that the project follows to engineer software. The processes and practices in turn drive the choice and use of tools. The next aspects of the diagram we explain are the nested rounded rectangles: The outermost rounded rectangle denotes the overall approach to software engineering including all its processes, practices and tools. Part of the overall approach to SE are the approach to documentation (AtD) and the approach to implementation (AtI). AtD and AtI refer to a combination of processes, practices and tools for documentation and implementation respectively. Together these AtD and AdI drive the approach to modeling (AtM). The approach to modeling itself again consists of a modeling process, a set of modeling practices and a collection of modeling tools. To summarize, this theoretical framework enables explaining which modeling approach is followed in a project by tracing it to the goals of the stakeholders and the project context.

For explanation purposes we have used 'drives' arrows between process, practice and tools in one direction. In reality, there may as well be arrows in the opposite direction where tools make a practice (im)possible or constrain possible processes. The same applies for the arrows between concepts specific to modeling and concepts general for SE: a modeling approach may enable or constrain the general SE approach.

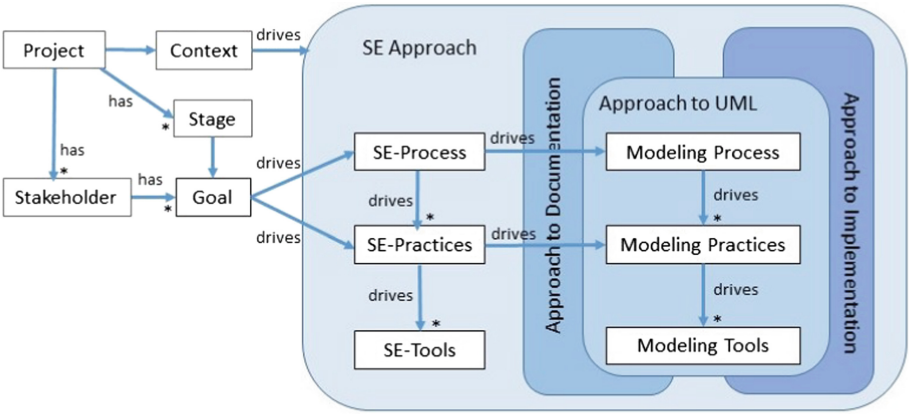

Fig. 7. Theoretical framework for modeling practices

4 Modeling Pathways

Earlier we observed that models of software designs serve multiple purposes to different stakeholders of software development projects. In this section we will zoom into how the types of models used in a project change as a project progresses. We explain this by means of Fig. 8. This diagram combines the main dimensions for classifying models that were introduced in Sect. 2. In the horizontal direction this diagram shows stages of modeling: moving from ideation into production. The vertical direction shows the abstraction levels: architecture, design and implementation. A project's modeling practices can occupy zero or more cells in this matrix. In this matrix, we have drawn a pathway that illustrates the evolution of the focal role of design models over time. Together with a change of their focal role, design models also change their abstraction level, their rigour as well as other dimensions mentioned in Table 1.

The first phase of the pathway is denoted 'I' and deals with ideation. Typically ideation addresses the architecture and/or design levels of abstraction. It is not common for models to be used for ideation of the implementation of a whole system, but ideation is not uncommon for designing parts of an implementation - such as e.g. use of patterns. A typical next use of design models (denoted 'II') is for externalization (communication, standardization, persistence). This can happen for the architecture and for the design. Also externalization is not common for the whole implementation - because the source code is a good source of information for the implementation. The third stage (denoted 'III') is production. Here the model is used to guide or produce the implementation. For this stage, the model must contain all the details necessary for a developer or compiler to generate the implementation. Its level of abstraction is therefore generally medium to low ('implementation level'). Please note that such pathways focus on the process of creating and refining a design. As development of the system progresses and also in the maintenance of the system, all of the

design models created along a pathway will have to be updated when significant changes are made.

Indeed, a key problem for most modeling-pathways is that of updating more abstract design models to reflect (the significant parts of) increments of models at a lower level of abstraction. In theory, this 'continuous synchronization' of models at different levels of abstraction seems technically possible when both levels are described in a rigorous/formal manner and there exist clear traceability between two representations. Osman has demonstrated a prototype for synchronizing models across different levels of abstraction [21], yet much more work is needed in this direction.

A company may have a design-flow that supports the creation of a next system model on the basis of a set of systematic transformations of a model from a preceding stage. In this case there is high traceability between successive models. Alternatively, subsequent design models may be created largely independently from previous models. This happens for example when the models are used mostly for supporting the own understanding of the designers (at that stage). In the latter case there is poor traceability between successive models.

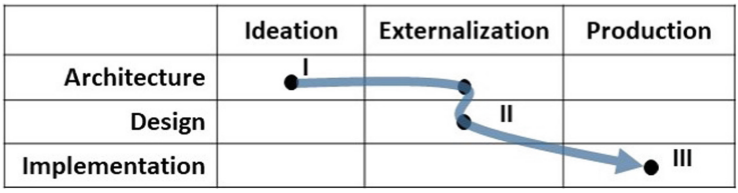

Fig. 8. Pathway of models in software development

In the future, we aim to show how these pathways can be used to illustrate the different approaches to modeling found across different projects.

5 Observations on Modeling in Open Source Projects

We set out to look into open source projects for empirical evidence on the use of modeling.

To answer these question, we mined GitHub for open source projects that use UML. We identified more than 20,000 projects that use UML [9]. Then, we ran a survey to collect information from more than 400 open source developers who work on these projects [10]. In this section, we report a brief selection of our findings.

Stage of Development: In our set of open source projects models are introduced at all phases on the life-cycles. Yet we found a concentration of first appearance of models around the start of projects [9]. Furthermore, for 26% of the projects

we found that models were updated as projects progressed over time. The questionnaire confirmed that models are used for all three stages of development: ideation, externalization, and production. For ideation we found a large number of photos of sketches of UML diagrams. As documentation (i.e. externalization), models most commonly targeted the design-level of abstraction. Also, reverse engineered diagrams are also frequently used to serve as documentation. However, code generation based on implementation-level models, was only reported for few of the OSS projects [10]. Possibly because this requires an advanced level of training on methods and tools, and advanced coordination amongst the contributors.

Use of Models: Our survey asked after the use of models through a multiple choice questions to which multiple responses were possible. The responses are shown in Fig. 9[3]. By far the most common uses are for documentation (an externalization use) and for ideation and production of designs at the architecture- and design-level.

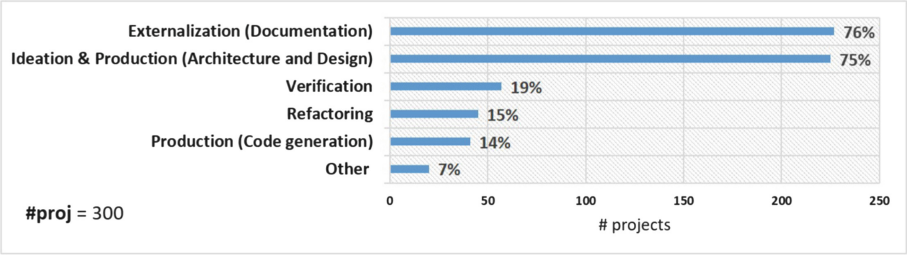

Fig. 9. Uses of models in open source software development

The responses to the survey confirm the non-negligible use of models for less obvious uses: 19% of the respondents uses models for verification tasks and 15% of the respondents uses models as part of doing refactoring. Another observation from our survey is that novices often use existing UML models as one of the most important sources for gaining an understanding of a system. Interestingly, the developers who create these models often seem unaware that other project members use their models in this way.

Models for coordinating distribution of work (planning): To investigate one other use of UML models, we explored whether models are used to coordinate work. The first interesting observation was that the design model was implemented by only a single person (no coordination) in only 33% of our set of open source projects. In 41% of the projects 3 or more persons participated in the implementation of a design. This suggests that models indeed are frequently used

[3] The categories have been renamed to be consistent with the naming used in this paper.

to coordinate implementation tasks. When asked for the involvement of these developers in the modeling process, it turned out that 88% of the persons implementing a modeled design also participated in the creating of that model. Thus, it seems that many open source projects adopt a *team modeling* approach, where developers create models together. This may be a particular trait of open source projects as the open source community values 'equality' and 'transparency'.

For more findings on these studies we refer to [9,10].

6 Future Directions

In this section we discuss selected ideas on future directions for improving the effective use of modeling in software development.

6.1 Aligning the Tools with the Tasks and the Process

Tooling continues to be mentioned as a problematic area in model-based development. In this section we use our classification matrix together with design pathways to understand why tooling continues to be problematic.

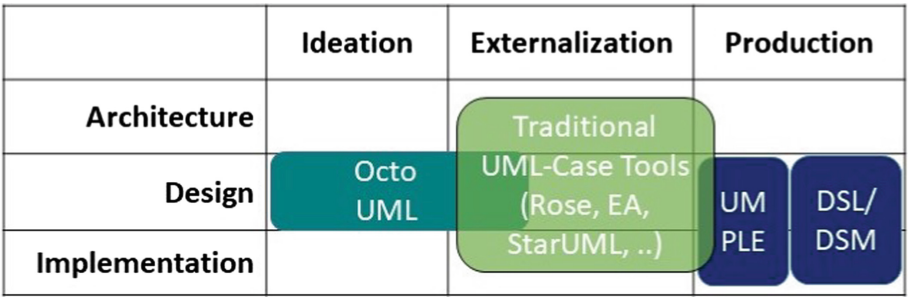

Fig. 10. UML tools across design stages

There are several major challenges related to modeling-tooling: One challenge is the large diversity in modeling styles. It is very difficult for any one tool to be a good match in supporting so many different uses. Integrating all possible uses of modeling in a tool would lead to a 'universal Swiss army knife': it becomes too complicated to use. Hence, this diversity of uses also explains the continued existence of a large variety of UML-based software modeling and design tools.

Figure 10 uses our classification matrix to explain that different tools focus on supporting different stages of development: Traditional UML CASE tools focus on creating UML models that strictly follow the UML syntax. One can consider them 'UML editors'. They can cover parts of the implementation- and production-stage (through code generation), but generally ignore informal notations and sketching that is typical for the ideation stage. Tools that aim to

support MDA with code-generation force developers to model at the implementation level of abstraction. One example of such a tool is UMPLE [8]. Moreover, while there are certainly benefits of these types of tools, their use is mostly limited to the production stage. Other tools aim to bridge the gap between ideation and externalization by offering both informal sketchy modeling and transformations of these into rigorous/geometric UML shapes. One example of such a tool is for example OctoUML [12]. In summary, in practice models are developed along pathways that cross different stages of development and change in abstraction level. None of the modeling tools that is currently around efficiently supports a complete pathway.

Fig. 11. UML tools across design stages

Figure 11 uses our classification matrix to illustrate where various supporting features of modeling tools fit in the development process. It shows that code generation transforms design level representations into code implementation level representation. Reverse engineering tries to reconstruct a design representation from an implementation. Conformance checking (such as [23], and [16]) try to verify (and quantify the degree of) the correspondence between the implementation and design. To this end, conformance checking techniques need to find ways to represent the abstractions that are made between design and implementation.

Future Direction 1. *Future software design tooling should support the mixing of text, sketches, formal diagrams, and source code in a flexible manner.*

Motivation: (i) different developers have a need for different combinations of text, sketches, diagrams and code. (ii) some artifacts evolve from one form (sketch) into another form (formal diagram). If this is a common use of models, then tools should support this.

Future Direction 2. *We need to move away from documentation as a static source of information about a system. Instead, we should move to dynamic 'information/knowledge' management about a design: multiple sources of data about a system should be combined dynamically and smart selections and abstractions of these data should be presented in an interactive way is both user-centric and task-centric.*

The aforementioned issues are related to two key aspects of modeling tools: (i) usability, and (ii) efficient chaining of tools in tool-chains.

6.2 A Promising Future: Domain Specific Architecture- and Modeling

There are interesting model-based approaches in practice that counter the aforementioned usability and tool-chaining issues: One notable example are so-called 'low-code' platforms as offered by e.g. Mendix and OutSystems. Their approaches capitalize on the fact that the most common architecture is a 3-layered architecture that consists of a data-layer, a business logic layer and a user-interaction layer. The 'low-code' approach offers 3 separate modeling languages (each of which can be considered a domain-specific modeling language): one modeling language for specifying data-models, one modeling language for defining business processes, and one for specifying user-interaction (user interface, possibly with user-processes). This approach is illustrated in Fig. 12.

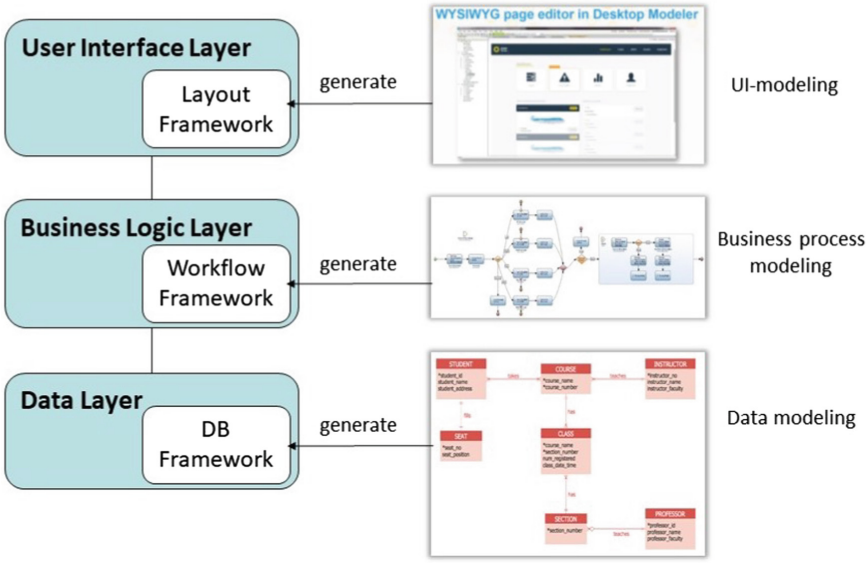

Fig. 12. Low-code approach to modeling for 3-layered architectures

These approaches are called 'low-code' because an entire running system can be generated out of a triplet of three types of models - hence no (textual) code is involved. Through specializing for particular types of software architecture, and separating the concerns in different modeling languages, the modeling of an application becomes fairly simple. Indeed these companies have shown factors of 3×–10× of speed-up in application development. One other key aspect of these

new approaches is that the development pipeline is highly automated and even includes automated deployment (e.g. in the cloud) and automated production of app's for mobile platforms. In our view, this is one example of how modeling can be used in a very effective way by specializing the modeling language and linking it to a common architecture and architectural style in a particular domain.

One complementary study evaluated the impact of migrating of an existing (legacy) 3-layered architecture that was programmed 'manually' from scratch into a format that was generated out of a (3-layered) domain specific model (DSM) [15]. The results show that after introducing the DSM-approach the defect density lowered, defects were found earlier, but resolving defects took longer. Other observed benefits are that the number of developers and the number of person-hours needed for maintaining the system decreased, and the portability to new platforms increased.

When seen together, the use of the combination of a domain specific architecture and domain specific modeling languages promise to offer higher levels of abstraction while still being able to generate the implementation.

6.3 Practical Guidelines for Tailoring Modeling Approaches

Our theoretical framework Fig. 7 argues that project goals and project context are the drivers for the approach to modeling that is used in project. Currently the theory is explanatory in character: we can use it to explain the differences that we observe. Possibly the same framework could be made actionable if we could use it to create guidelines on *how to* chose modeling processes, practices and tools to best fit particular projects goals and context. This requires collecting best practices in modeling approaches and systematic ways for documenting contexts as part of case studies [5].

7 Summary and Conclusions

The term 'modeling' is a general term and it is used in a large variety of meanings. Possibly some people even want 'modeling' to mean certain things. However, as a result many scientific studies in software engineering fail to provide sufficiently precise characterizations of the modeling-practices that are used in the projects that they study. This lack of precise characterization leads to confusion and contradictions about the findings of modeling in software development. In this paper, we introduce several classification to more precisely describe the types of modeling encountered in software development. The main dimensions of these classification are: (i) the different levels of abstraction, and (ii) different stages of development. We introduced a classification matrix that combines these dimensions. This matrix can be used to characterize models as well as illustrate pathways that characterize how the focus of models evolve as projects progress.

Additionally, we proposed a theoretical framework that explains how the use of different modeling practices can be explained by looking at how they are driven by different context and different project goals. Further, we showed that

UML models are used for many more activities than only the guiding of the implementation.

All these extra uses of models impose additional requirements on the tools and processes used for the creation and maintenance of UML models throughout software projects. UML tools have to cater for interoperability with other processes and tools used in software development in order to support software engineers in moving tasks through differ stages of development. Only by recognizing and embracing the diversity of uses of models throughout software projects can we improve the tools needed to support their effective use. Currently, usability of modeling tools is one of the main challenges in modeling. In order for modeling to be more effective and achieve higher adoption, tools should become a better fit with the tasks of developers and better support various uses of models throughout the entire software development process.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley Professional, Boston (2012)
2. Brown, A.W.: Model driven architecture: principles and practice. Softw. Syst. Model. **3**(4), 314–327 (2004)
3. Chaudron, M.R.V., Heijstek, W., Nugroho, A.: How effective is UML modeling? Softw. Syst. Model. **11**(4), 571–580 (2012)
4. Cross, N.: Design Thinking: Understanding How Designers Think and Work. Berg, Oxford (2011)
5. Dybå, T.: Contextualizing empirical evidence. IEEE Softw. **30**(1), 81–83 (2013)
6. Fernández-Sáez, A.M., Chaudron, M.R.V., Genero, M.: Exploring costs and benefits of using UML on maintenance: preliminary findings of a case study in a large it department. In: EESSMOD@ MoDELS, pp. 33–42 (2013)
7. Garousi, G., et al.: Usage and usefulness of technical software documentation: an industrial case study. Inf. Softw. Technol. **57**, 664–682 (2015)
8. Garzón, M.A., Aljamaan, H., Lethbridge, T.C.: Umple: a framework for model driven development of object-oriented systems. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 494–498. IEEE (2015)
9. Hebig, R., Quang, T.H., Chaudron, M.R.V., Robles, G., Fernandez, M.A.: The quest for open source projects that use UML: mining GitHub. In: Proceedings of the ACM/IEEE 19th International MODELS Conference, pp. 173–183. ACM (2016)
10. Ho-Quang, T., Hebig, R., Robles, G., Chaudron, M.R.V., Fernandez, M.A.: Practices and perceptions of UML use in open source projects. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp. 203–212. IEEE Press (2017)
11. Hofstede, G., Hofstede, G.J., Minkov, M.: Cultures and Organizations - Software of the Mind: Intercultural Cooperation and its Importance for Survival, 3rd edn. McGraw-Hill, New York (2010)
12. Jolak, R., Vesin, B., Chaudron, M.R.V.: OctoUML: an environment for exploratory and collaborative software design. In: ICSE 2017, vol. 17 (2017)

13. Lange, C., Chaudron, M.R.V., Muskens, J., Somers, L.J., Dortmans, H.M.: An empirical investigation in quantifying inconsistency and incompleteness of UML designs. In: Workshop Consistency Problems in UML-Based Software Development II, pp. 26–34 (2003)
14. Lange, C.F.J., Chaudron, M.R.V.: Effects of defects in UML models: an experimental investigation. In: Proceedings of the 28th International Conference on Software Engineering, pp. 401–411. ACM (2006)
15. Mellegård, N., Ferwerda, A., Lind, K., Heldal, R., Chaudron, M.R.V.: Impact of introducing domain-specific modelling in software maintenance: an industrial case study. IEEE Trans. Softw. Eng. **42**(3), 245–260 (2016)
16. Muskens, J., Bril, R.J., Chaudron, M.R.V.: Generalizing consistency checking between software views. In: Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), 6–10 November 2005, USA, pp. 169–180. IEEE Computer Society (2005)
17. Nugroho, A., Chaudron, M.R.V.: A survey of the practice of design-code correspondence amongst professional software engineers. In: ESEM 2007, September 2007, Spain, pp. 467–469. ACM/IEEE Computer Society (2007)
18. Nugroho, A., Chaudron, M.R.V.: A survey into the rigor of UML use and its perceived impact on quality and productivity. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, 9–10 October 2008, Germany, pp. 90–99. ACM (2008)
19. Nugroho, A., Flaton, B., Chaudron, M.R.V.: Empirical analysis of the relation between level of detail in UML models and defect density. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 600–614. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_42
20. Osman, M.H., Chaudron, M.R.V.: UML usage in open source software development: a field study. In: Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling Co-located MODELS 2013, USA, vol. 1078, pp. 23–32. CEUR-WS.org (2013)
21. Osman, M.H., Chaudron, M.R.V., van der Putten, P.: Interactive scalable abstraction of reverse engineered UML class diagrams. In: APSEC 2014, South Korea, December 2014, pp. 159–166. IEEE (2014)
22. Torre, D., Labiche, Y., Genero, M.: UML consistency rules: a systematic mapping study. In: EASE 2014, UK, 13–14 May 2014. ACM (2014)
23. van Opzeeland, D.J.A., Lange, C.F.J., Chaudron, M.R.V.: Quantitative techniques for the assessment of correspondence between UML designs and implementations. In: 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (2005)
24. Zhi, J., et al.: Cost, benefits and quality of software development documentation: a systematic mapping. J. Syst. Softw. **99**, 175–198 (2015)