Measuring Productivity from Model-Based Development
A Tale of Two Companies

Juha-Pekka Tolvanen, MetaCase

Model-based development, using models as the primary source when creating systems and software, is claimed to improve productivity. We describe practical evaluations that two companies have conducted internally. In both cases the key success factor has been able to produce the code and other artefacts largely automatically from the models and use of internally developed domain-specific modeling languages giving full control to their users.

Raising the level of abstraction through the languages used by developers has improved the productivity and quality of development work. Today this trend is perhaps most readily seen in interest in low-code and no-code environments [2]. At their core, these environments use specialized modeling languages that reduce, or totally remove, the use of traditional programming [4][10]. Technically this is not a new idea: a higher-level specification created by humans is translated to a lower level of abstraction. We have seen the same in control engineering [11][13], with ITU's SDL [5] and naturally with compilers.

Rather than expecting tool vendors to provide these higher-level languages, companies have also moved towards creating specialized languages for themselves. These Domain-Specific Modeling (DSM) languages move away from plain text to be visual modeling languages and narrow down the application domains into just the type of products the companies develop [7]. Thanks to advances in tooling, the effort to define specialized languages has decreased drastically: a review of industry cases shows that with good tooling it can be less than 2 person-weeks [15].

We report two cases where companies have estimated the impact of model-based development to productivity. For measuring productivity, one of the most common measures, both in academic studies [1][6] and in reports from practice [14], has been the development time. It works over different languages and is a variable that acts as a proxy for the cost too [8]. Development time is also a measure companies are using as a basis for budgeting, project planning and is related to salaries. This is also the measurement the companies are often most interested – as in our two case companies.

Case 1: Enterprise Applications in the Cloud

A company had created their own modeling language for developing web-based applications for the cloud. The potential impact to productivity was measured in the same way as the company's project business: How much effort was needed to implement an application that met customer requirements [3]. An engineer developed a typical application in two ways: with traditional programming and with their DSM solution. Data was collected by recording the time used, categorized according to the three main parts of enterprise applications: database, user interface (UI) and logic.

One engineer conducted both development cases and had less experience on modeling than on programming. While this situation did not treat the approaches equally, it resembles the situation today in most companies. The programming approach was chosen to be evaluated before the modeling, to avoid the risk that the generated implementation would guide manual implementation.
With the same requirements and review phase for acceptance, the comparison focused on the differences between the approaches. Programming steps were:

1. Schema definition with SQL Server Management Studio using SQL.
2. Business logic implementation in C# with Visual Studio.
3. UI implementation in HTML and JavaScript.

With modeling approach steps were:

1. Modeling the application (data, user data, user rights, roles, alerts etc.) with MetaEdit+ and generating the code.
2. Adjusting the UI with SQL.

SQL, C#, HTML and JavaScript been widely documented are not detailed here. For modeling the company used its own DSM language. Details of this language are not public, but Image 1 shows a model of the developed Customer Project Management (CPM) application using their modeling language. Whole application was generated from this model and run in Azure.

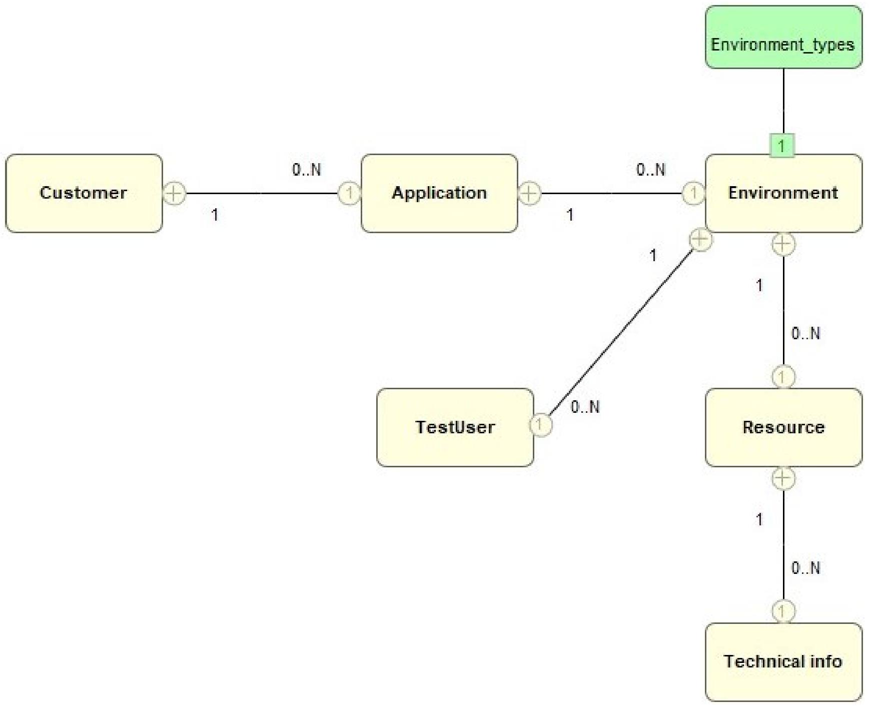

Image 1: Model of the CPM application [3]

Table 1 summarize the effort in hours per steps. Creating the application with the company's own DSM solution was significantly faster, taking 6.5 hours compared to manual programming taking 36 hours. The productivity difference was over 554%. The main effort in programming went on implementing the UI (16.5 h) and business logic (12.5 h) rather than defining the schema. When using the company's own modeling language, the situation was very different: modeling took 1.25 hours and the code generation from the models took just some seconds. After running the initial application, 2.25 hours went to adjust the generated UI with SQL.

Step	**Programming (h)**	**Modeling (h)**
Modeling (and generation)		1.25
Database schema	5.50	
Business logic	12.50	
User interface	16.50	2.25
Other (project setup etc.)	1.50	3.00
Total	**36.00**	**6.50**
	Productivity ratio:	**554%**

Table 1: Comparing hours of work

Interestingly, installation and project settings took less time when using programming tools than the company's own modeling solution. One reason could be that programming tools were ready products with installations and wizards whereas some of the in-house developed tools those were not at the same level.

Case 2: Industrial Automation Systems

While the first case focused on software only, the second case deals with developing automation system addressing also electrical parts, installation and materials. The company provides fish farm automation systems together with a partner taking care of the on-site electrical and hardware installation. In brief, each fish farm system is unique per customer depending on the type and number of ponds and related functionality, like feeding, light control, alarms and monitoring (e.g. water pH value, temperature, muddiness etc.).
The company had developed a DSM solution with the aim of reducing development and maintenance efforts [9]. Rather than comparing traditional PLC programming (IEC 61131) and building each system from the scratch, the comparison was done against the applied "clone&own" approach: An existing software is copied and adapted based on the requirements of the new system.
The comparison excluded the requirements analysis part been the same on both approaches and focused on:

1. Developing the software and guides for installation (e.g. PLC I/O and cabling)
2. On-site installation (electrical and hardware)

The modeling language was created and applied in MetaEdit+ tool [12]: Image 2 shows an example model of one fish farm system. The blue ellipses describe ponds having various connections with oxygen input, feeder, electricity, signals etc. Full details of this language are not available in public. Resulting model is formal so that software and guides for installation can be generated.

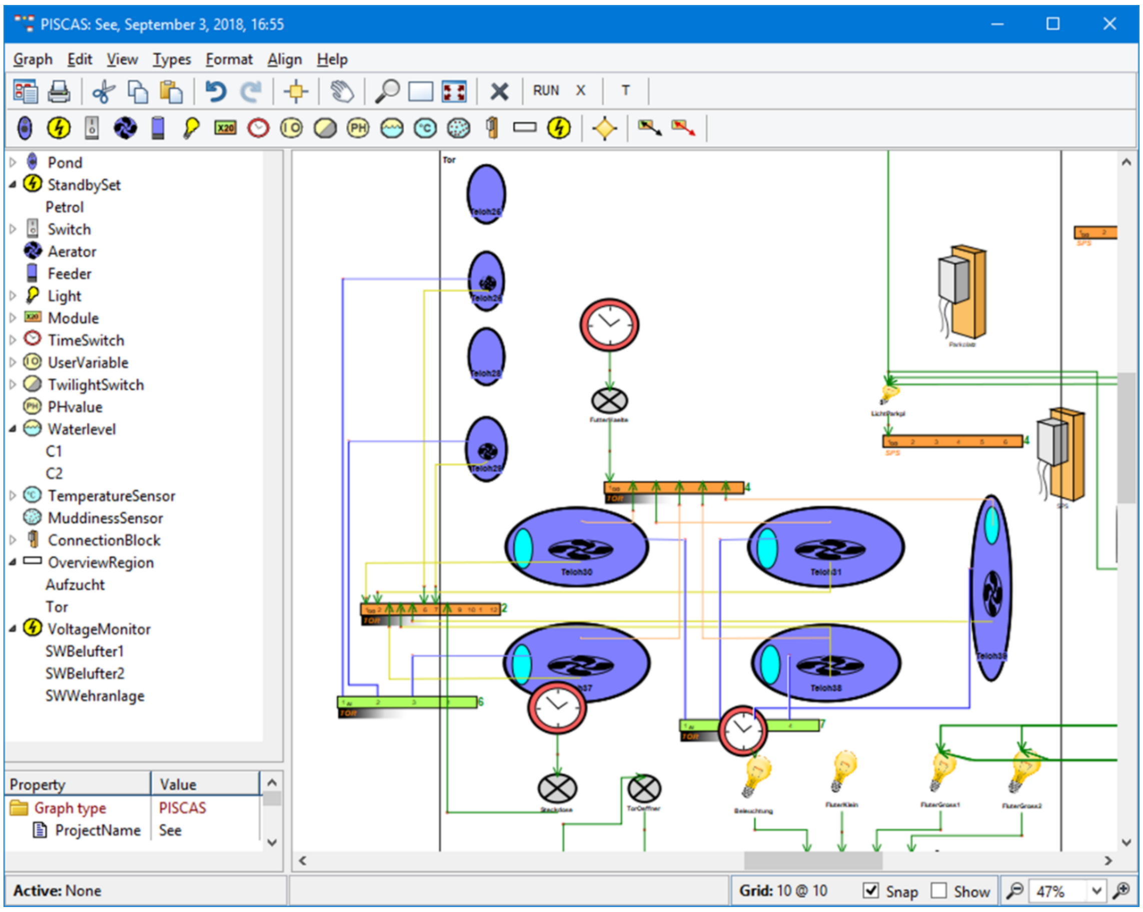

Image 2: Model of a fish farm system

Table 2 summarizes the evaluation results. Developing the system by modifying the software code took 8 times longer than model-based approach.

Parts	**Code clone&own (h)**	**Modeling (h)**
Software	16	2
Electrical installation changes	5	
On-site installation	80	60
Total	**101**	**62**
	Productivity ratio:	**163%**

Table 2: Comparing hours of work

When including also on-site installation the productivity ratio was 163% in favor to model-based approach. These non-software related differences occurred because there were errors in the initial electrical specifications that were found after installation and additional changes needed to be made.
Insufficient communication can easily lead to misunderstandings, resulting in system integration errors. These costs are often "hidden" ones but measuring them as in the case shows significant extra effort.

Conducting Evaluations in Practice

Companies conduct empirical evaluation rarely, because of the high costs in terms of time or resources: Building the same system twice with different development approaches, using several teams, evaluating dozens of developers or analyzing large number of development tasks. Many good scientific research methods are simply too expensive and time-consuming for practical use in a commercial setting.

The two companies decided to follow a rather easy to conduct and practical evaluation: Build a typical system twice to get relevant data. Neither of the studies provide generalizable or statistically significant results but aimed to provide enough data to see the difference to make a decision**.**

In both cases a key reason for clear productivity improvements was producing the code largely automatically from the models. Also, in both cases the same model was used to produce various kind of artefacts rather than creating separate models for different generation needs (e.g., generate database, operations, wiring plan, UI etc.). This was possible because the companies had access to the generators and modeling language definitions. This control over languages and generators in model-based development is identified as a success factor also in other studies [16].

Concluding Remarks

The comparison conducted by two companies report over 500% increase in software development productivity when moving to model-based development. When including electrical and on-site installation the productivity ratio was 163%.

Both cases also show a change in the development work: rather than dividing the work into different languages, the use of domain-specific languages allows to integrate work into one model. Changes are made to this model and all checks were performed on this same model too: there was no need to manually maintain different kind of code files, installation guides or documentation.

List of References

[1] Dzidek et al., A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance, IEEE Transactions on Software Engineering, No 3, 2008

[2] Fact.MR, Low Code Development Market Outlook (2022-2032), 2022

[3] Fadjukoff, L., Domain-Specific Modeling in application development, Uni of Tampere, 2021, https://urn.fi/URN:NBN:fi:tuni-202111018071

[4] Farshidi et al., Model-driven development platform selection: four industry case studies. Softw Syst Model 20, 2021

[5] ITU, Specification and Description Language, https://www.itu.int/rec/T-REC-Z.100

[6] Kieburtz et al., A software engineering experiment in software component generation. In ICSE ’96: Procs of the 18th Int. Conf. on Software engineering, IEEE, 1996

[7] Kelly, S., Tolvanen, J.-P., Domain-Specific Modeling: Enabling Full Code Generation. Wiley, 2008

[8] Kelly, S., 2013. Empirical Comparison of Language Workbenches. Procs of the ACM Workshop on Domain-Specific Modeling, 2013
[9] Leitner et al., Effective development of automation systems through domain-specific modeling in a small enterprise context, Softw Syst Model, 2014
[10] Di Ruscio et al., Low-code development and model-driven engineering: Two sides of the same coin?, Softw Syst Model, 2022
[11] MathWorks, Simulink, https://www.mathworks.com/products/simulink.html
[12] MetaCase, MetaEdit+, https://metacase.com/
[13] National Instruments, LabVIEW NXG 5.1 Manual: Programming in G, 2020
[14] Tolvanen, J-P. and Kelly, S. Model-Driven Development Challenges and Solutions, Procs of the 4th Int. Conf on Model-Driven Engineering and Software Development, 2016
[15] Tolvanen, J., Kelly, S., Effort Used to Create Domain-Specific Modeling Languages. Procs of the Conference on Model Driven Engineering Languages and Systems, ACM, 2018
[16] Whittle et al., The State of Practice in Model-Driven Engineering, IEEE Software, no.3, 2014

Author

Dr. Juha-Pekka Tolvanen works at MetaCase. He has been involved in domain-specific approaches and tools since 1991. Juha-Pekka has acted as a consultant world-wide for modeling language and code generation development. He has co-authored a book (Domain-Specific Modeling, Wiley) and over 100 articles in software development magazines, journals and conferences.

Contact

Internet: www.metacase.com
E-mail: jpt@metacase.com