Modelling By the People, For the People

Steven Kelly (http://orcid.org/0000-0003-0931-157X)

MetaCase, Jyväskylä, Finland
stevek@metacase.com

Abstract. Domain-specific modelling has moved the focus of model content from the implementation domain to the problem domain. However, much research still sees most modellers as programmers – a tacit assumption that is exemplified by the use of IDEs as modelling environments. Model-Driven Engineering research should instead be reaching out to make itself presentable to subject matter experts – as language creators, users, and research subjects. Every leap in developer numbers has been triggered by a new format, and we need another such leap now. Domain-specific modelling is ideally placed to step up and enable the creation of applications by people of all backgrounds.

Keywords. domain-specific modeling, productivity, programmer demographics

1 Introduction

Domain-specific modelling has moved the focus of model content from the implementation domain to the problem domain [1]. However, much Model-Driven Engineering (MDE) research still sees most modellers as programmers – a tacit assumption that is exemplified by the use of IDEs as modelling environments, diagrams bearing a striking resemblance to UML, and structures straitjacketed into the hierarchical tree of XML. The false assumption seems to be "because I must build the language workbench in a programming IDE, experts must create their languages in that IDE, and modellers must model in that IDE". MDE research is stuck in a programmer mindset and in IDEs with roots in the last millennium, when it should be reaching out to make itself presentable to subject matter experts, both as language creators and users.

This paper looks at the development of our industry over its whole history, considering the broad factors that are at play in its evolution. The problem of the focus on programmers is shown to be the lack of an infinitely expandable supply of programmers, and the inevitable effects on average programmer productivity caused by the natural 'best first' allocation of programmers from the high end of the ability scale.

2 Developments in Developer Demographics

Every leap in developer numbers has been triggered by a new format, either in languages (machine code to assembly language to third-generation languages (3GLs)) or

devices (mainframes to PCs to mobile). ‘Language leaps’ came from research making software development easier, so that more people are able to successfully create applications. ‘Device leaps’ are different: rather than increasing the supply and decreasing the cost, they increased the demand. Language leaps have increased developer numbers much more than devices leaps: an order of magnitude with the move to assembly or to 3GLs, as opposed to a ‘mere’ doubling alongside the PC revolution in the early 1980s, when the number of computers grew by a factor of over 25.

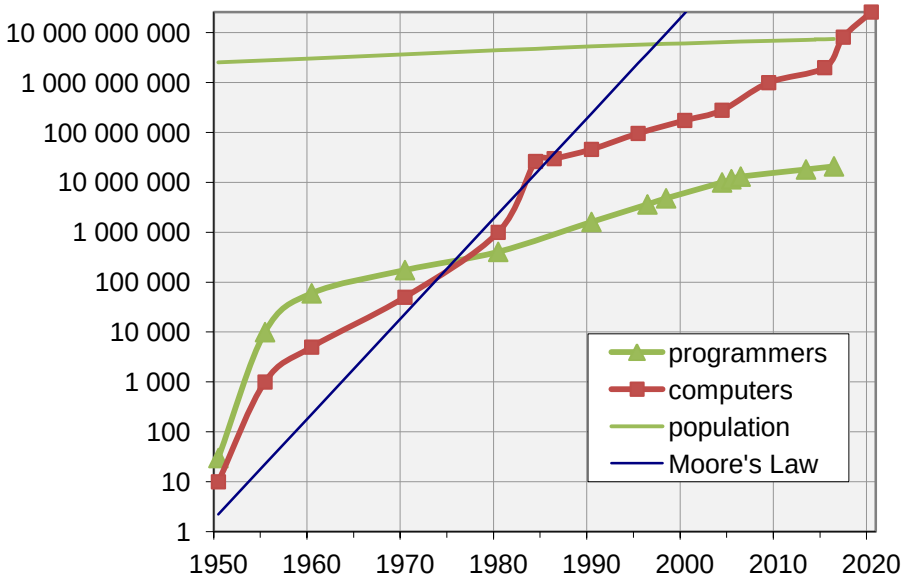

Fig. 1. Growth in numbers of programmers and computers globally

Interestingly, the increase in the number of developers in the language leaps seems to mirror the increase in productivity of a given average developer, e.g. roughly five times faster with 3GLs compared to assembly language. Since the early 3GLs, there has not been an appreciable leap in productivity, so it seems fair to assume that we have made only incremental progress in making software development accessible to people for whom it was previously too hard. Learning and working in JavaScript is not much easier than learning and working in COBOL. Contrast that with how much easier either is than working directly in machine code.

At the start of the 1980s, there were 313,000 programmers in the USA; a decade later, one million. (The US percentage of world programmers shrunk only modestly, from 77% to 62%, so this figure is useful globally too.) Let us assume a normal distribution for programming skill in the population: any other plausible distribution gives essentially the same results. For the sake of familiarity, and in the lack of better data, let us take the same mean of 100 and standard deviation of 15 as in IQ tests. Similarly, for simplicity let us take the US population to be constant at 240 million (only a few percent different at either end of that period). As programming was by no means a new profession in 1980, let us further assume that the programmers then

were the top people in terms of innate programming skill, and that the rest of the million added during that decade were the next best, rather than distributed more widely. Many assumptions, but we are not seeking an exact answer, just a general picture: What does the innate skill of programmers look like over time?

The programmers in 1970 ranged from percentile 99.933 to 100, so the median programmer was at percentile 99.966. By 1980, new programmers were being added at percentile 99.870. In 1990, the number of programmers in the US passed one million, so percentile 99.73. By 2000, the number of programmers in the US passed 3 million, so new programmers then were being added at percentile 98.75. On the IQ scale[1], these percentiles would correspond to scores of 151, 145, 142 and 134. For a programmer working since 1970, there has been a significant drop in the expectation of a teammate's innate ability back then compared to a new programmer in 2000.

In fact, innate programming skill is made up of more than IQ. IQ tests, being a series of small problems, do little to measure ability to cope with complexity. That skill, increasingly important for today's programmers, is thus a somewhat independent variable, plausibly also on a normal distribution. Another attribute useful for programming is being able to put up with low-level detail (whether of problems, languages or tools). Some smart people like that, others hate it; again, it seems an independent variable. A fourth skill is being able to communicate with and work with people – a skill which seems almost to correlate inversely with the previous one, and which the stereotype of a programmer is generally seen as lacking. With some positive correlation and some negative correlation on these skills, we can perhaps take them as four independent normal distributions. What does innate programming skill look like, if it is the product of four normally distributed independent variables? Each additional normal variable sharpens the peak of the distribution, further increasing the drop in scores for each percentile change at the top tail of the distribution.

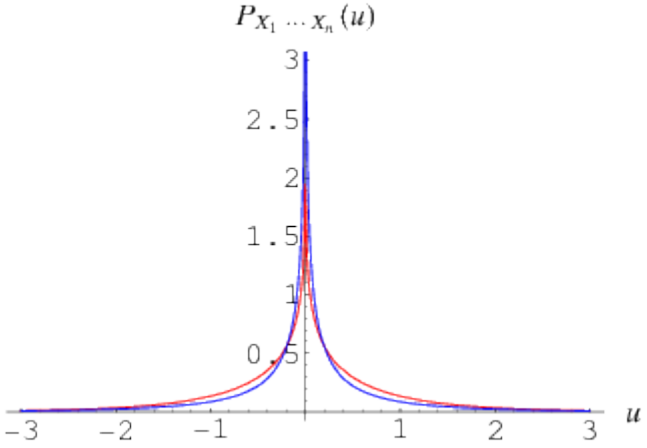

Fig. 2. Product of two (red) and three (blue) normally distributed variables [2]

[1] https://www.iqcomparisonsite.com/iqtable.aspx

For any given programmer, there is significant variability in development velocity, depending on the difficulty of the task. In particular, near the limit of the programmer's ability, velocity drops sharply, becoming zero at the limit. As the innate ability of new programmers has dropped, the proportion of tasks which those programmers can achieve progress with has dropped. When considered as part of a team, an individual's productivity contribution becomes negative around the limit of his ability.

In 2004, the number of programmers in the US peaked at 4 million, shrinking by 10% over the next decade before starting to grow again slowly, but still remaining below 4 million. Outsourcing had an effect, as did the broader economy, but both of those have now largely been overturned. As the level of programming skill available outside the current ranks of programmers decreased, with no easing of the complexity of tasks or improvements in the languages used, there came a point where a corollary to Brook's Law [3] was reached: Adding new programmers to an industry would have a negative effect on production.

This cessation in growth of the number of programmers in the US has lasted over a decade. At the same time the US population has grown over 10%, and the demand for applications has grown significantly with the rise of the web, the smart phone and other devices. Part of the demand has been met by programmers from overseas. However, these have largely been meeting the needs of a US market rather than their own home markets, and as their home markets grow, and wages even out across the world, this temporary relief will inevitably dry up. When we hit our corollary to Brook's Law globally, what can step in to save us?

3 Libraries, Frameworks and Languages

Components, libraries and frameworks are sometimes touted as offering a decisive advantage. A pre-existing component or library may save a company's time, but if there is enough need to make working without a library inefficient, and no such library yet exists, a company will simply assign a developer to write the library (explicitly or implicitly). Writing the application code that uses it will of course be largely the same, regardless of whether it was created in-house or available from elsewhere: one may be better-targeted, whereas the other may be more mature. A library thus offers a productivity gain to a company, but does not generally change the task of the programmers who use it, or their productivity on that task.

A framework can be described as 'a library with attitude': it does more than offer functions, letting you 'fill in the blanks' of its ready-built behaviour, and bounding and guiding you on the content of those blanks. With respect to the problem domain task and the architecture, a framework thus makes things easier for the software developer, increasing productivity. But for the majority of frameworks, the content entered 'in the blanks' is still largely 3GL code. If you are not up to the task of writing 3GL code, you cannot successfully create an application even with the framework. A framework alone will not make application creation available to a larger audience.

A domain-specific language, on the other hand, helps both in increasing productivity and in reducing the demands on the developer. The 'blanks' are filled in with do-

main concepts, not code. By abstracting away from the implementation technology, a DSL also often makes it possible to generate the same application for different formats: web server and client, desktop app, and mobile app.

A graphical or projectional DSL in particular generally constrains you to only be able to create legal models, as opposed to the illegal source code you could write while trying to use a framework. Among programmers there are clearly a disproportionate number who prefer text over graphics, compared to the population at large; conversely, a graphical DSL is thus often a better choice when reaching out to current non-programmers.

4 Related Research

Over such a broad period of history, the amount of related research is huge. We will limit ourselves to either end of the period, to see what – if anything – has changed, and whether there are new directions on the rise.

In 1954, when most development was still in machine code and moving to assembly language, MIT hosted a session called "Future Development of Coding – Universal Code" [4] on the use of flow diagrams in coding. 16% of participants had "a higher level individual set up a flow diagram and persons at a lower level do the coding of each block". Grace Hopper "remarked that flow diagrams offer a potent means of communication since they are not tied to the computer." C. W. Adams "suggested that if the specifications are sufficiently rigid, the machine itself could handle the coding": automatically generating full code from flow diagrams. G. E Reynold's group built diagrams with "magnetized blocks that are easy to erase and to assemble."

Model-Driven Engineering has achieved its best successes when based on Domain-Specific Modelling, rather than UML or text-based "models" [5]. A number of tools exist, with the most successful being distinct from programming IDE or abstracting away from it. Representational formats have coalesced around graphical diagrams, which have changed little over the last 60 years. A growing trend changes those 'box and line' diagrams, at least for beginners, to blocks that connect by interlocking [6]. Modelling is also starting to become feasible on mobile platforms [7].

5 Conclusion

Good domain-specific modelling languages in non-IDE tooling have consistently shown productivity improvements of a factor of 5–10 [5]. Given the earlier link between productivity increase and increase in the number of people who can successfully develop software, there would seem to be a good possibility that a DSM approach could make application creation possible for an order of magnitude more people. Our own experience with MetaEdit+ [8] tends to confirm this: we have clients where none of the modellers are programmers, clients where all are, and clients where there is a mix. The future could well reach the ideal of a balanced mix of good programmers and smart subject matter experts, all collaborating directly in the same set

of models. MDE is ideally placed to step up and make possible the creation of applications of all formats, by people of all backgrounds.

Since the 1990s, much modelling research has focused on creating tools and languages, but with declining returns in terms of novelty, incremental benefit over current industrial use, and adoption. For tools, researchers need to throw off their "not invented here" attitudes, and reach out to investigate real industrial use of non-IDE based commercial domain-specific modelling tools, both language workbenches and fixed tools like Simulink and LabVIEW. For languages, rarely does a researcher have the domain knowledge and experience to create a good DSM language for a company. Rather, they should concentrate on being a catalyst, enabling a company to create its own language, studying that process, and incrementally improving it as they learn over several cases. In this area, more research is definitely needed and will be fruitful.

Bibliography

1. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008)
2. Wolfram Research, I.: Normal Product Distribution. Available at: http://mathworld.wolfram.com/NormalProductDistribution.html
3. Brooks, F.: The Mythical Man Month: Essays on Software Engineering 2nd edn. Addison-Wesley, Reading (1995)
4. Adams, C., Hopper, G.: Future Development of Coding – Universal Code. In Adams, C., Gill, S., Combelic, D., eds. : Digital Computers – Advanced Coding Techniques, vol. Summer Session 1954, pp.80-81 (1954)
5. Tolvanen, J.-P., Kelly, S.: Model-Driven Development Challenges and Solutions: Experiences with Domain-Specific Modelling in Industry. In : Proceedings of MODELSWARD 2016, 4th International Conference on Model-Driven Engineering and Software Development, pp.711-719 (Feb 2016)
6. Bau, D., Gray, J., Kelleher, C., Sheldon, J., Turbak, F.: Learnable Programming: Blocks and Beyond. Communications of the ACM, June 2017, pp. 72-80 (May 2017)
7. Vaquero-Melchor, D., Garmendia, A., Guerra, E., de Lara, J.: Towards Enabling Mobile Domain-specific Modelling. In : ICSOFT, pp.117-122 (2016)
8. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In : Proceedings of the 8th International Conference on Advances Information System Engineering, pp.1-21 (1996)
9. Weintrop, D., Wilensky, U.: To Block or Not to Block, That is the Question: Students´Perceptions of Blocks-based Programming. In : Proceedings of the 14th International Conference on Interaction Design and Children, New York, NY, USA, pp.199-208 (2015)