SEMAT Position Statement

Jean Bézivin, INRIA & Ecole des Mines de Nantes, France

From Programs to Mograms[1]

Position Paper for the Software Engineering Methods and Theory Initiative

Changes in Software Engineering

The apparently smooth and constant evolution in software production and maintenance practices, over nearly half a century, may be hiding some forthcoming radical changes, more qualitative than quantitative. There are some indications that we are currently crossing some new frontiers in technology and practices and it may be helpful to make these situations explicit. The SEMAT initiative comes out very timely to provide an open and rich discussion forum where these issues of change management may be identified and their possible consequences on the future of software engineering discussed. Three important past transitions may be remembered and these may help understanding the possible impact of more recent and even future changes.

Past Ruptures

We have already seen several ruptures in the young history of software engineering. The first one was discussed at the Garmisch NATO meeting in October 1968. The emergence of complex systems obliged to realize that the period of the individual (and isolated) programmer was over, and that the target was "large programming systems of more than 30,000 instructions, produced by more than 25 programmers, in more than 6 months' development time, with more than one level of management". The move from individual programming to collective software production was then initiated. Recent interest in agile programming shows that the debate cannot yet be considered as completely closed, but one of the indirect consequences of this NATO meeting was probably the recognition of the importance of team issues, process issues and software architecture issues. The work done later on “programming in the large vs. programming in the small” (DeRemer, Kron) may also be linked to this influence and one may notice the ubiquity of language issues in all these discussions (mainly specification, architecture and process languages).

A second important rupture could be observed in the early 80’s with the paradigm change from procedural to object-oriented programming. The analysis of this move to object-oriented technology is complex and has many possible interpretations mentioned by several authors (Dahl, Nygaard, Kay, Meyer, etc.) like:

- The inadequacy of the procedural paradigm to easily describe real world situations,
- The inadequacy of the procedural paradigm to allow large scale software reusability,
- The inadequacy of the procedural paradigm to build stable software architectures.

---

[1] Mogram is a contraction of “model” and “program” suggested by A. Kleppe. A mogram is not always computer-excutable, but it is nevertheless precise and conforms to a given metamodel.

Here again the importance of language issues (Smalltalk, Eiffel, Objective-C, C++, etc.) appears central in raising the abstraction level.

A third rupture was triggered by the OMG MDA initiative in November 2000. The initial motivation was the need for portability of information systems in regard to the rapid evolution of technological platforms (CORBA, DotNet, XML, etc.). Some ideas suggested by the OMG in this MDA vision were again about raising the abstraction level with the help of a family of languages named "modeling languages" by opposition to "direct programming languages", and the corresponding impact on:

- The use of these abstract modeling languages to describe business and application systems independently of the underlying technical platforms,
- The use of transformational techniques to map these business systems on various present or future platforms,
- The separation and composition of facets (e.g. platform-dependent and -independent facets) by various domain-specific modeling languages.
- The handling of most interoperability issues at the language level (abstract syntax or metamodel) and not at the middleware level.

Present and Future Ruptures

These three examples have contributed to shape the software engineering landscape that we know today. However new important ruptures may be arriving, for example:

- The emergence of the end-user programmer as a key actor and the redefinition of the role of the professional programmer (an Excel user may be seen as an end-user programmer when producing spreadsheet content for example),
- The emergence of multiple Domain Specific Languages (DSLs) as an alternative to general purpose languages for handling specific tasks in specific contexts,
- The rapid arrival on the market of huge amounts of software applications in so-called application stores (Apple, Google, etc.) not only limited to smartphone users,
- The increasing diversity of technological solutions and the need to make them interoperate at a reasonable cost in a continuum integrating legacy assets[2] and more advanced solutions (e.g., cloud computing),
- The need to constantly adapt and even synchronize the information systems to more and more rapidly evolving business organizations.

The question is now to suggest an ambitious new vision for software engineering in the 21st century, encompassing theory and practice and able to cope with the aforementioned ruptures and additional ones that will certainly show up in the future.

We know that the number of applications that will have to be produced in the next 30 years for individual, professional or social needs will be in exponential grow. We know that the total number of professional programmers in the world going to implement them is likely to be stable or in a very slow linear increase. We learnt from the past that whatever progresses we may achieve in software tools, the productivity of software professionals to develop applications will not improve by a major factor. The equation seems thus impossible to solve if we stay in the same context. We need to be imaginative to

[2] Making the future interoperate with the past will be an important software issue for this century, considering not only the COBOL, but also the Java legacy that is being accumulated. Many applications will not been built from scratch, but by evolution of a previous legacy.

find realistic solutions to this apparently impossible equation. The key to solving this problems probably lies in the collaboration between professional and end-user programmers, but do we need to invent a specific software engineering set of practices for end-users?

So there will be a huge number of rapidly evolving interacting applications, often written by different people in a number of different languages, and usually not developed with classical software development lifecycle. Each of these applications has a specific data model, state model and event model. The result is high fragmentation and the need to make heterogeneous parts interoperable has never been so acute. We need paradigm-agnostic integration methods that will allow coping with language coordination and large-scale application interoperability.

Plus ça change, plus c'est la même chose (Software Engineering is Language Engineering)

In software engineering also, the more things change, the more they remain the same. The invariant that may be noticed in practices and technology ruptures is that they are very often related to language issues. The only new thing is that the definition of language has been extended. Textual languages stay very important but many kinds of visual languages are appearing and some languages offer an abstract syntax with various concrete syntaxes, sometimes textual, tabular or visual. These languages are defined not only by grammars, but also by metamodels, schemas, ontologies, etc. The application scope of these languages will be considerably extended, mainly to specific tasks or domains. Most of these languages will allow capturing facets that will have to be combined with other facets to be meaningful. Programmers (professional or end-users) will use these languages to write programs, or terminal models, or domain code or "mograms". Precision of these mograms is of paramount importance even if very often they are not executable. Computer executability is not necessary to achieve precision. Languages will allow describing products as well as processes, code and data, requirements, specifications and algorithms, test, architecture, etc. Applications will be built with a number of different languages, usually more than ten or twenty for each. More important, the evolution and maintenance of these applications will have to be carried on with all these languages.

The practical and theoretical core of software engineering is language engineering. But we need to revisit the new scope of definition, implementation and usage of languages in a more general way than the old general purpose executable grammar-based textual programming languages.

Model Driven Engineering (MDE) is one of the proposals that are currently on the table as a proposed core contribution. MDE tries to imagine a new set of practices to produce and manage precise mograms, conforming to open and evolving metamodels, within a network of precisely related artifacts. Milner's vision in his "Towers of Models" grand challenge sets up a very exciting and ambitious target for a new software engineering kernel that could be built from MDE. In this perspective, more than inventing new languages, the difficulty will be in establishing precise and operational semantic relations between all these DSLs that will constitute the core of the new software engineering practices for the 21st century. Language interoperability will anyway be an important part of the corresponding landscape.