Fourth Generation Languages are Technical Debt

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Extended Abstract

Fourth Generation Languages, or 4GLs, were highly praised in the 1960s through 1980s [1], [2] for being non-procedural high level specification languages that allow software developers to write concise yet readable code that is easier to design, develop, evolve and maintain. As time goes by, some of those languages evolved into what we call now Domain-Specific Languages, or DSLs, and manage to satisfy their customers by providing domain-specific notations and abstractions, advanced tool support, sufficient configurability, boosts in productivity and maintainability, etc. However, many of them are being retired and pushed out of the market, some faster than others when the vendors of their compilers announce the day when they stop maintenance and support. In slower cases the language compiler changes its owner several times and usually ends up being owned by a big corporation in its trophy collection next to other compilers with minimal support and maximal fees. Such languages are usually seen as legacy [3], [4], and owners of codebases and portfolios largely relying on such 4GLs, actively undertake steps towards their retirement, investing millions in multi-year plans for software modernisation [5], [6].

4GLs do not have to be seen as technical debt, but we claim that they should. Having a codebase written in a legacy unmaintained 4GL is a result of a series of (arguably small) mistakes in digital portfolio management and software development strategies. Moreover, relying on such a 4GL is not deadly for a company, since usually the code works as intended and can be kept operational for many years to come. However, it is rather costly—for many reasons, the simplest one being the inability to find experts in that language and as a consequence, the necessity to invest in hiring people with generic skills and a desire to learn, and educating them up to the required level of specialisation. Other reasons typically include paying unjustifiably high fees for both system software (such as a compiler, and IDE and/or a DBMS) and hardware (e.g., the mainframe). These costs are constantly and unstoppably growing, until something drastic is done: either the codebase is discarded, or the assets are migrated, or the owner declares bankruptcy, etc.

When we view 4GLs as technical debt, there is one obvious technical solution to the problems related to them: refactoring. Since the conceptual gap between a 4GL and mainstream GPLs is too big [4], [6], it cannot be done directly. However, many 4GLs are built as compilers to a higher-level software language with a compiler available (such as COBOL or PL/I). This generated code in COBOL or PL/I is feature-equivalent to the code in 4GL because it is essentially the code that is being deployed and tested. All its undesirable properties from bad indentation to unstructured `GO TO`s, can then be refactored. Once the COBOL or PL/I code has reached tolerable levels of quality, it can be migrated to another platform, a more suitable IDE, integrated with other systems and languages, etc.—the options available for legacy GPLs are much wider than the options for legacy 4GLs.

In our demonstration we would like to focus on one tool in particular, used to migrate over 200 MLOC of production code owned by various companies [7]. The migration starts with a codebase written in a 4GL called PACBASE, developed in France in 1972 [8]. The PACBASE code is compiled to COBOL code, which suffers from all kinds of problems and is for all means and purposes impossible to maintain. However, we developed hundreds of refactorings aiming at improving the technical quality of COBOL code with a specific focus on COBOL generated from PACBASE. Namely, we:

- clean up `GO TO` and `PERFORM THROUGH` constructs
- promote loops to `VARYING` clauses
- reverse engineer `COMPUTE` statements
- remove dead code and unused variables
- enforce consistent use of `END-IF` and its kind
- pretty-print the remaining code

This tool has been used in many projects and served as a foundation for many successful migration endeavours for various customers of Raincode Labs over the years.

References

[1] L. Schlueter, *User-Designed Computing: The Next Generation*. Lexington Books, 1988.
[2] J. Martin, *Applications Development Without Programmers*. Prentice-Hall, 1981.
[3] M. Feathers, *Working Effectively with Legacy Code*. Prentice-Hall, 2004.
[4] V. Zaytsev, "Open Challenges in Incremental Coverage of Legacy Software Languages," in *PX/17.2*, 2017, pp. 1–6.
[5] R. Khadka, A. Saeidi, S. Jansen, J. Hage, and G. P. Haas, "Migrating a Large Scale Legacy Application to SOA: Challenges and Lessons Learned," in *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, 2013, pp. 425–432.
[6] A. A. Terekhov and C. Verhoef, "The Realities of Language Conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, Nov./Dec. 2000.
[7] Raincode Labs, "PACBASE Migration: More than 200 Million Lines Migrated," https://www.raincodelabs.com/pacbase/, 2018.
[8] A. Alper, "Users Say Pacbase Worth Effort," *Computerworld*, vol. 21, no. 33, pp. 21–23, Aug. 1987.