Low-Code Is Often High-Code, So We Must Design Low-Code Platforms to Enable Proper Software Engineering

Timothy C. Lethbridge(✉)

University of Ottawa, Ottawa, Canada
timothy.lethbridge@uottawa.ca

Abstract. The concept of low-code (and no-code) platforms has been around for decades, even before the term was used. The idea is that applications on these platforms can be built by people with less technical expertise than a professional programmer, yet can leverage powerful technology such as, for example, for databases, financial analysis, web development and machine learning. However, in practice, software written on such platforms often accumulates large volumes of complex code, which can be worse to maintain than in traditional languages because the low-code platforms tend not to properly support good engineering practices such as version control, separation of concerns, automated testing and literate programming. In this paper we discuss experiences with several low-code platforms and provide suggestions for directions forward towards an era where the benefits of low-code can be obtained without accumulation of technical debt. Our recommendations focus on ensuring low-code platforms enable scaling, understandability, documentability, testability, vendor-independence, and the overall user experience for developers those end-users who do some development.

Keywords: Low-code platforms · Modeling · Technical debt · End-user programming · Umple · Spreadsheets

1 Introduction

Low-code and no-code platforms are diverse in nature, but have some common features intended to allow people to build complex software mostly by configuring powerful underlying engines. Some applications built using low-code platforms have proved game-changing, or even life-saving, for example applications built rapidly to handle the Covid-19 pandemic [1].

Unfortunately, in practice it is common for people to write far more code and far more complex code than the low-code platform designers likely expected. The code can become exceptionally hard to understand and maintain, yet frequently needs to be modified, since low-code platforms are particularly prone to change for commercial reasons. What starts out as a good idea, building a powerful application with little code, turns into a mountain of technical debt. An example of this that many people can relate to is Excel, where individual formulas can sometimes span many lines, yet cannot even

T. Margaria and B. Steffen (Eds.): ISoLA 2021, LNCS 13036, pp. 202–212, 2021.
https://doi.org/10.1007/978-3-030-89159-6_14

be indented. A spreadsheet may be packed with many such formulas, and some of those may refer to macros or code in third-party plugins. None of this can easily be browsed, tested, documented, or reused.

It would seem reasonable to presume that low-code approaches are a key to the future of programming. But in this paper, we make the case that the designers of such platforms must always assume users will in fact write large amounts of code in their platforms, so they need to design the platforms to be high-code-ready – in other words ready for proper software engineering to be applied.

In the next section we will discuss low-code platforms and some of the challenges they present in practice. Then we illustrate some of the issues through case studies based on our experience. Finally, we will propose a set of principles that low-code platform developers should embrace.

2 No-Code and Low-Code as a Long-Standing and Growing Trend

No-code platforms provide a spectrum of core functionality for various classes of applications that can be tailored, typically through graphical or form-based user interfaces, to provide business-specific end-user experiences. Through the graphical interfaces, users select, arrange, configure and connect elements from built-in libraries of elements as well as from third-party plugins.

Examples of no code platforms include Shopify [2], which is at the core of a wide array of e-commerce web sites. WordPress [3], similarly, is at the core of a vast array of web sites. Spreadsheets, such as Microsoft Excel and similar products also fall into this category. Even software developers working with traditional programming languages use no-code platforms: Jenkins for example is one of several open-source platforms that can be configured for a wide range of automation tasks: It is most typically used for continuous integration (building and testing versions of software), but can do much more. A competing commercial no-code system is Circle-CI.

Low-code platforms share similarities with no-code platforms but with low-code there is expectation that a certain amount of code will be written, where code in this context implies small custom conditional expressions or algorithms. The idea is that such code operates on highly abstract and powerful features present in the system core, as well as on an ecosystem of plugins. Tools such as Appian, Oracle Application Express and Wavemaker are widely used to allow businesses to create information-systems on top of databases. The Eclipse platform used by developers, with its plugin capability can be seen as a low-code platform, as can some other IDEs.

Today many low-code and no-code platforms are cloud-native, meaning that the software runs as web applications. However, we include programmable downloadable apps in the low-code umbrella.

For simplicity, in the following we will refer to both no-code and low-code platforms as low-code even though there are some differences with regard to capabilities, scale and applicability. The difference between low-code and no-code is often merely in how it is used, and may be a matter of perception: For example, Excel clearly transitions from no-code to low-code when macros are used, but one might also say that the use of conditions in formulas means that Excel should not be considered no-code.

The low-code concept is far from new. Apple's original Hypercard was designed as a low code platform, and many interesting applications were built in it (e.g. [4]). The World Wide Web itself started as a low-code way to create static hypertext information systems, also pre-dating the invention of the low-code terminology. It is instructive to note how the Web has clearly become a high-code platform both on the front end and back end.

To further illustrate the long history, we can note that spreadsheets have been with us for over 40 years. The 1980's also saw the marketing of so-called fourth-generation languages (4GLs) [5, 6, 7]. These included languages such as Natural and Sperry's Mapper, which sold themselves as having many of the same benefits we attribute today to low-code platforms. Mapper, for example envisioned enabling business executives to generate their own analytic reports from mainframe databases.

The rise of model-driven engineering (MDE) and UML in the 1990's allows us to consider the low-code concept from another perspective: If one can describe an application's data in a class model and its behavior in state machine diagrams one can theoretically avoid a lot of textual coding. Reducing coding and enabling systems to be described in part by non-programmers has thus been part of the vision of many proponents of MDE.

Even many scripting languages can be seen as being motivated by the low-code vision. From the 1970's to today, every competent Unix/Linux programmer has written short scripts to automate operating system tasks and make their workflow easier; Apple Macintosh end-users similarly also have been able to use AppleScript and Automator.

Fast-forwarding to today, we see the confluence of low-code with ascendent technologies such as machine learning. Whereas it used to take a lot of coding to build machine learning into an application, tools like Google Auto-ML [8] provide a low-code experience. Low-code for quantum computing seems an inevitable next step – perhaps an essential step given the seeming complexity of describing quantum algorithms directly.

The benefits of low-code are clear: It allows for rapid deployment of powerful computerized functionality, tailorable by well-educated end-users, and certainly without the need for a developer to have deep knowledge of the underlying platform or of computer science. For basic use they merely require an understanding of a 'model' embodied by the platform (e.g. the tables in a database, or the layout of a spreadsheet), and some sense of how to extend and configure the system using mostly-declarative constructs.

However, in *actual practice from a software engineering perspective*, low-code applications deployed in industry turn out to be not really that different from traditional ones programmed in traditional 'high-code' languages like C++ or Java. The following subsections outline some of the reality.

2.1 Large Volumes and Complexity Make Low-Code a False Promise

The code volume found in deployed applications on low-code platforms is all-too often not 'low'. We have encountered app-control 'scripts' of thousands of lines, spreadsheets with many thousands of complex formulas, and programs in supposedly low-code business applications containing hundreds of thousands of lines. Clearly, code can accumulate to volumes that are just as high as in traditional languages. Such applications might be written entirely in the platform's built-in native domain-specific language (DSL), or else rely on programmed plugins or wrappers written in some other language.

Low-code applications can hence accumulate high complexity, bugs and other forms technical debt just as badly or even worse than traditional applications. However, the languages created for low-code platforms commonly lack features that would assist in their understanding and maintenance, such as the ability to organize them into files and modules, or even to add detailed comments attached to individual code elements.

2.2 Low Code Applications Often Lack Features Needed for Maintainability and Understandability

Low code applications are also often lacking documentation, both in terms of what developers write, and what documentation can feasibly be embedded with the code even if an attempt is made.

In traditional applications it was once thought that documentation should be found in external design documents, but over the last couple of decades that has been replaced by an ethos of literate programming (e.g. well-written names for data and functions, with carefully laid-out code), coupled with extensive code comments, in-repository files aligned with the code (e.g. Readme.md files) and auto-generated documentation enabled in part using code annotations (e.g. Javadoc). Opportunities to do this sort of documentation tend to be lacking in low-code platforms as some of the case studies below will testify.

Low-code applications are also often challenged with regard to separation of concerns and reusability. Although plugins are a dominant feature, the ability to make one's own code, written in a low-code language, modular or to re-use such code (without making a plugin) in multiple applications is often absent.

2.3 Turnover, Deprecation and Vendor-Dependence Further Challenge Low-Code

Whereas the code in many traditional platforms can be modified for years, with older code still runnable, there is a tendency (although not universal) for low-code applications to be vulnerable to rapid obsolescence.

The platforms on which low-code applications are built tend to be rapidly developed with new major versions requiring the low-code applications to be modified (i.e. maintained) to keep running. This is in part because most such platforms are commercial, and companies want to produce 'improved' or 'all-new' offerings. Historically, many low-code commercial platforms have just ceased to be developed. Many 4GLs of the 1980's are examples.

Extensive modification or replacement of low-code applications is hence needed at levels more frequent than would be the case for traditionally-programmed applications.

This would not be so much of a problem if the code were really 'low' in volume, and there was documentation and little technical debt. But the reality is that these assumptions tend to be false, resulting in premature death of large applications. Even open-source low-code platforms are subject to this problem, especially when the plugins on which low-code often depends are not maintained by other open-source developers. We have noted that this is the case with tools such as Jenkins.

3 Short Case Studies

In the following subsections we briefly give some case studies highlighting challenges with low-code platforms. Each of these summarizes the author's own experiences working in research, the public sector, the private sector, and the volunteer sector.

3.1 Excel and Other Spreadsheets

Spreadsheets like Excel are clearly low-code platforms and often no-code. At a base level, they are usable by almost any educated person; they have a layered-2D (table with sheets) model, an easily understandable instant-calculation semantics and a rich library of functions with a variety of plugins also available. Modern spreadsheets have migrated to the cloud, like other low-code platforms.

But spreadsheets can grow fantastically in complexity [9, 10]. Excess complexity can arise through five different types of scaling: the number of formulas, complex arrangement and interconnections of those formulas, massively complex formulas (many lines of text with no way to lay it out and comment it), macros (in Visual Basic for the case of Excel) and plugins.

Spreadsheets tend to be very fragile (proneness to bugs if modified incorrectly), and difficult to understand, with subtle differences in formulas not easy to notice, even if the spreadsheet makes some attempt to warn users. There is difficulty separating concerns: separate sheets or files can be used for this, but traditional languages are much more flexible in this regard, allowing almost-limitless flexibility in arranging files. It is almost impossible in spreadsheets to do proper detailed documentation (e.g. of complex formulae, patterns of cells, and so on); this is only possible in macros which have a traditional programming-language structure. Reuse of formulas in different spreadsheets is also not generally practical, so they tend to be cloned with the consequent problem of bug propagation.

Division of work among multiple developers is extremely challenging in a spreadsheet. Developers of code in languages like C++, Java and Python now are used to using configuration management and version control tools like Git, collaborating on code and using pull requests with code reviews and automated testing to reach agreement on what should become the next version. Although cloud-based spreadsheets do indeed allow multiple people to edit the code at once, and have 'change tracking' capability, this is far from the power available in traditional code.

Talented software developers can still create poor-quality spreadsheets, partly because Excel has core limitations regarding what is possible as described above, and partly because spreadsheets (as all software) tend to grow organically and inexorably, surprising even their own developers.

As an example of over-exuberance with low-code, the author witnessed a situation where a company put out a CFP for development of an application that might be expected to take over a person-year to develop. A talented Excel expert instead proposed that the requested requirements could all be satisfied in a few days of Excel spreadsheet development. The expert offered a company a completed product, developed in Excel from scratch within two days without even participating in the competitive bidding process. The application did more than the customer expected, so was welcomed almost

as a miracle. But it was 'too clever'; it was only readily maintainable by its single developer. It had to be replaced within two years by a more traditionally-developed system.

3.2 WordPress

Vast numbers of websites rely on WordPress. Some are professionally maintained but very many are in the hands of people with zero training in software development, perhaps people maintaining a site for a local sports club.

Yet as new ideas for specialized information presentation, or uses of plugins are added, and as additional volunteers work on the site (often serially, as new people are elected each year), such sites often descend into the worst kind of software mess. The author has witnessed computer professionals stepping in to help such end users, and, finding the 'low code' incomprehensible, they have 'hacked' at it, making it worse.

Regular updates to WordPress or its numerous plugins, forces further hacking: The ugliness of the technical debt becomes ever more visible. The solution is often to 'start again'. But starting again on a different low-code platform just perpetuates the situation, since the new version will descend into the same sort of mess.

3.3 Jenkins

Jenkins, as a self-hosted automation and CI technology can be used with relatively little or no code, although technical expertise is needed for installation and maintenance. The real lesson, however, is the challenge of the plugins. These plugins mostly add additional configuration fields to various GUI panels in the classic no-code fashion. Many plugins have been created, yet many have ceased being maintained by their open-source developers. Many are marked as 'for adoption' yet are not adopted. They thus represent technical debt for people who incorporated them in their automation workflows.

3.4 Modeling Languages with Code Generation

As a final case study, we would like to mention the notion of modeling technologies often related to UML that allow code generation and integration of code using 'action languages'. An example is Papyrus. The reality is that although such technologies can save a lot of coding, our experience is that they can only reduce code volume by about half.

The action-language code then has to be managed in some way, and tends to be subject to the same weaknesses as we noted for Excel and WordPress. In some tools the action language code is embedded in XML files that also are used to convey the model (class model, state model, etc.), so can only be edited in the modeling tool. When this action code becomes extensive it suffers from low understandability, low testability, difficulty with collaborative development and so on.

The author's team have been working to overcome these limitations through the development of Umple [11, 12, 13]. Umple is a compiler, developed in itself, that translates a textual representation of models to both final systems and diagrams; it also allows

editing of the diagrams in order to update the textual representation. Umple can be used in many environments including in the UmpleOnline website [14], where users can also experiment with a library of examples.

Our experience is that to build large systems with model-driven development, one has to deploy all the best software engineering practices that have been developed over the last half century for use with traditional code. Umple models are thus organized in the same sorts of files as used for traditional programming languages; they can be operated on using collaboration, version-control, and documentation-generation tools. Umple allows seamless blending of model representations with code in multiple traditional programming languages, and the instantly-generated documentation can consist of UML diagrams and Javadoc-style pages. Most Umple users do indeed use it in a low-code manner, but a few very large applications have been created without the limitations imposed by other low-code platforms.

4 Directions Forward

It seems likely that low-code platforms will continue to proliferate and will be the dominant way many types of software are developed in coming decades. But as discussed above, the amount and complexity of code in such systems will often not in fact be 'low', and may be exceedingly high.

The following are a few principles that, if adopted, we believe would help software to achieve the 'best of both worlds.' In other words, to allow applications to be developed with modest amounts of code on top of powerful platforms, while at the same time enabling the scaling of such code as needed, by enabling good software engineering practices.

4.1 Enable Low Code, but Plan for Lots of Code

Firstly, as a community, we should drop the pretense that low-code applications will remain low-code. There is nothing wrong with enabling powerful capability from very little code, but we need to understand that people *will* write high-code applications on low-code platforms.

4.2 Enable Documentability in Low-Code Platforms

Low-code platforms need to be designed to be documentable and written in a literate fashion. It should be possible to see live diagrams of the code as it is edited, or to edit the diagrams themselves to modify the underlying code. This is something we have achieved with Umple.

For example, in spreadsheet, it should possible to see formulas rendered as easy-to-understand textual entities with proper indentation and syntax highlighting. It should also be possible to write comments within the formulas.

Although spreadsheets have 'auditing' tools to do things like showing which cells depend on which others, much greater effort needs to go in to helping make complex code in spreadsheets and other low-code platforms understandable.

4.3 Improve Separation-of-Concerns, Re-use, and Collaboration Capabilities

Traditional languages have a variety of ways of separating concerns, such as using functions, multiple files organized in a hierarchy, mixins, traits and aspects. One can reuse one's own code in multiple contexts. In Umple we have arranged for all these features to be available in the action code added to models.

It would be nice if there were more effective ways of reusing formulas in spreadsheets. Macros go part way, but force the developer to delve into a totally different programing paradigm.

Separation of concerns and reuse go hand-in-hand with collaboration: As the volume of 'low code' gets high, multiple developers need to divide up development, and follow all the best practices of agile development.

4.4 Enable Automated Testing in Low-Code Platforms

One of the great revolutions in software over the last 20 years has been the now ubiquitous practice of automated testing (both unit testing and system testing). This prevents regression as all tests must pass when changes are made. Test-driven development takes that one step further, requiring tests to be delivered with each change. Pull-request testing allows testing of proposed changes against the current released version to ensure there is compatibility.

Automated testing has some distance to develop in the low-code context, yet it is desperately needed. There has been a small amount of research in this direction regarding Excel [15], but it is not part of the core platform yet needs to be. Testing in the context of model-driven development is also in its infancy [16].

4.5 Foster Multi-vendor Open Standards for Low-Code Languages

Although there are many traditional programming languages, and some form the basis for low-code platforms, there is still a lack of open-standard ways of creating code for related classes of low-code platforms. For example, Visual Basic macros and plugins written for Excel won't work in other spreadsheets; plugins for Jenkins won't work in Circle CI and code written for one of the database low-code platforms is not portable to others. This needs to change.

The solution we have chosen for Umple is to allow one or more of several traditional programming languages (Java, C++, Php, Ruby) to be used for the action code (i.e. the low-code). There will be some API calls this code needs to make to Umple-generated code, but using Umple's separation of concerns mechanism, these can be isolated, thus rendering most of the action code fully portable.

4.6 Emphasize Developer Experience at All Scales

There is a lot of emphasis on user experience (UX) in software engineering today. Developer experience [17] is a key subtopic. Low-code tools would benefit from strong focus on this, particularly when the code becomes large and complex. Most low-code tools seem to only pay attention to the experience of developers that create small amounts of code.

5 Some Other Perspectives

Low-code platforms with a very low barrier to entry are also widely called end-user development (EUD) platforms. There is considerable discussion in the literature of various aspects of these platforms that relate to the points we make in this paper.

Sahay et al. [18] provide a taxonomy of both terminology and features of such platforms. Central to their analysis are interoperability, extensibility, learning curve and scalability.

Paternò [19] points out that EUD platforms not only need to avoid intimidating beginners, but also need a to be able to scale to allow experts to use the tools more expansively, as we have indicated is indeed the reality. He also emphasizes the need for tools to support collaborative and social development.

Repenning and Ioannidou [20] emphasize ensuring that tools allow people to have a sense of flow (get neither bored nor anxious), and make syntactic errors hard or impossible. Similar to what we are suggesting, they highlight that such tools should support incremental development, testing and multiple views. They also suggest creating scaffolding examples that users can adapt, and building community-supported tools.

6 Conclusions

We should stop worrying about the end of the need for programmers as we know them. History shows that no matter whether a platform asserts it is low-code or even no-code, businesses will find requirements that expand the scale and sophistication of programs developed using the platform. Hence there will be a steady need for skilled developers.

History shows that code written in low-code platforms often becomes complex, and is hard to understand, document and reuse. This results in increasing technical debt and the need for replacement of systems, exacerbated by rapid obsolescence of the underlying low-code platforms.

As a result of this, we need to ensure low-code platforms have just as deep a capability to support modern software engineering practices as traditional languages. In particular they need to enable literate coding, self-documentation, separation of concerns, collaboration, and automated testing. Vendors of similar applications should find ways to work together to allow exchange of code among such applications. Finally, the user interfaces of all low-code and modeling tools should be subjected to focused work to improve their developer experience.

What is next in programming? New high-code languages with sophisticated textual syntax will continue to arrive on the scene, as has been happening for decades. However, in our view these will be applied more and more in tight synchrony with low-code technology such as editing of model diagrams, and blending domain specific languages (DSLs) with the high-code languages. This will enable greater abstraction and return on programmer investment. However, to ensure that the return on investment occurs, companies will need to recognize the need to apply key software engineering techniques such as test-driven development.

References

1. Woo, M.: The rise of no/low code software development-no experience needed? Eng. (Beijing China) **6**(9), 960–961 (2020). https://doi.org/10.1016/j.eng.2020.07.007
2. Dushnitsky, G., Stroube, B.K.: Low-code entrepreneurship: Shopify and the alternative path to growth. J. Bus. Ventur. Insights **16**, e00251 (2021). https://doi.org/10.1016/j.jbvi.2021.e00251
3. Stern, H., Damstra, D., Williams, B.: Professional WordPress: Design and Development. Wiley, Indianapolis (2010)
4. Estep, K.W., Hasle, A., Omli, L., MacIntyre, F.: Linneaus: interactive taxonomy using the Macintosh computer and HyperCard. Bioscience **39**(9), 635–639 (1989)
5. Aaram J.: Fourth generation languages. In: Rolstadäs, A. (eds.) Computer-Aided Production Management. IFIP State-of-the-Art Reports. Springer, Heidelberg (1988). https://doi.org/10.1007/978-3-642-73318-5_14
6. Nagy, C., Vidács, L., Ferenc, R., Gyimóthy, T., Kocsis, F., Kovács, I.: Complexity measures in 4GL environment. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011. LNCS, vol. 6786, pp. 293–309. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21934-4_25
7. Coulmann, L.: General requirements for a program visualization tool to be used in engineering of 4GL-programs. In: IEEE Symposium on Visual Languages, pp. 37–41 (1993). https://doi.org/10.1109/VL.1993.269576
8. Xin, D., Wu, E.Y., Lee, D.J.L., Salehi, N., Parameswaran, A.: Whither AutoML? understanding the role of automation in machine learning workflows. In: 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–16, May 2021
9. Boai, G., Heath, A.: When simple becomes complicated: why excel should lose its place at the top table. Global Reg. Health Technol. Assess. (2017). https://doi.org/10.5301/grhta.5000247
10. Badame, S., Dig, D.: Refactoring meets spreadsheet formulas. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 399–409 (2012). https://doi.org/10.1109/ICSM.2012.6405299
11. Lethbridge, T.C., Forward, A., Badreddin, O., et al.: Umple: model-driven development for open source and education. Sci. Comput. Program. (2021). https://doi.org/10.1016/j.scico.2021.102665
12. University of Ottawa: Umple website. https://www.umple.org. Accessed Aug 2021
13. University of Ottawa: Latest Umple Release. http://releases.umple.org. https://doi.org/10.5281/zenodo.4677562
14. University of Ottawa: UmpleOnline. https://try.umple.org. Accessed Aug 2021
15. Khorram, F., Mottu, J.M., Sunyé, G.: Challenges & opportunities in low-code testing. MODELS 2020, pp. 70:1–70:10 (2020). https://doi.org/10.1145/3417990.3420204
16. Almagthawi, S.: Model-driven testing in Umple, Ph.D. thesis, University of Ottawa 2020. https://doi.org/10.20381/ruor-24577
17. Fagerholm, F., Münch, J.: Developer experience: concept and definition. In: 2012 International Conference on Software and System Process (ICSSP), pp. 73–77 (2012). https://doi.org/10.1109/ICSSP.2012.6225984
18. Sahay, A., Indamutsa, A., Di Ruscio, D., Pierantonio, A.: Supporting the understanding and comparison of low-code development platforms. In: 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 171–178 (2020). https://doi.org/10.1109/SEAA51224.2020.00036

19. Paternò, F.: End user development: survey of an emerging field for empowering people. Int. Sch. Res. Not. Softw. Eng. (2013). https://doi.org/10.1155/2013/532659
20. Repenning, A., Ioannidou, A.: What makes end-user development tick? 13 design guidelines. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) End User Development. Human-Computer Interaction Series, vol. 9, pp. 51–85. Springer, Dordrecht (2006). https://doi.org/10.1007/1-4020-5386-X_4