Mining Good Practices of Low-Code Software Development from Model-Driven Approaches

DANIEL PINHO, Faculty of Engineering, University of Porto and INESC TEC

ADEMAR AGUIAR, Faculty of Engineering, University of Porto and INESC TEC

VASCO AMARAL, NOVA LINCS, DI, FCT/UNL

Low-code software development enables the creation of applications without having to write many lines of code or even none at all. This lowers the level of technical skills required, providing a mechanism for non-technical people to develop software and reduces the effort on dependency management, integration and deployment. Research on low-code approaches is still in its infancy, and we aim to work towards expanding the available literature about good practices for the development of low-code platforms. Model-driven software development uses models and transformations as its main elements while providing higher levels of abstraction. It enables the automation software development processes, such as code generation or model interpretation. Given the similarities between low-code software development and model-driven software development, in this work, we focus on performing pattern mining for good low-code software development practices based on existing patterns from existing model-driven development literature. Taking the patterns that we found, we analysed and compared them to low-code contexts, having identified a potential starting set of patterns that showcase good practices for developing low-code software development platforms.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments; Model-driven software engineering.**

Additional Key Words and Phrases: low-code software development, model-driven software engineering, no-code software development, patterns

ACM Reference Format:

Pinho, D. et al. 2021. Mining Good Practices of Low-Code Software Development from Model-Driven Approaches. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 28 (October 2021), 9 pages.

1. INTRODUCTION

Modern society places great value on software systems. We use them in the most varied areas, ranging from healthcare and business applications to transportation, government, and education. We even make use of such systems in our homes, with IoT devices and entertainment-related applications such as video games and content streaming platforms.

This work was supported in part by the Fundação para a Ciência e Tecnologia (FCT) Grant 2021.08371.BD.
Corresponding author's address: Daniel Pinho, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal; email: daniel.pinho [at] fe.up.pt

In the problem domain, such systems may be particularly dependent on complex logic systems, business rules, or laws, and ensuring that developers implement what is needed while adhering to such rules may be difficult. Information transfers between stakeholders and developers can result in loss of information [Melnik and Maurer 2004], particularly if there are middlemen in this process.

One other thing to be mindful of is the current technological landscape, also known as solution domain; one often finds various technologies in an app's tech stack, such as libraries and frameworks. Such dependencies must be compatible with each other and, as the system's complexity grows, it may be hard to juggle different versions, potentially creating some technical debt [Hejderup et al. 2018].

Low-code software development (LCSD) tools have the potential to deal with these two concerns. Most of such tools allow developers to create applications and programs using limited amounts of writing source code[1], opting for a graphical user interface instead. They are typically web-based and may not even require the installation of a dedicated integrated development environment (IDE), just requiring a compatible web browser.

LCSD tools, with their visual approach, can lower the skill level needed to create the requirement-heavy programs mentioned earlier, allowing people without great technical skills to develop software. As such, they can help with the management of requirements and business rules by shifting some (or all) of the development work to the stakeholders and domain experts.

Meanwhile, LCSD tools try to focus development efforts on the program logic and not on the libraries and technologies used. Thus, they can reduce the need for dependency management and allow developers to focus on the features they want to implement, potentially reducing development time.

Despite its increasing prevalence in the software development industry, the term "low-code" is relatively new in academic contexts, being first used by researchers at Forrester Research in 2014 [Richardson et al. 2014; Sanchis et al. 2020]. While research on the topic in its current form has been increasing, there does not seem to be any work about it specifically related to patterns as conduits of good practices to recurring problems.

This work is the first step in filling that gap and providing guidance to the people behind LCSD tools by documenting good practices beneficial to developing low-code platforms. This work's focuses on the pattern mining phase, based on the close relation between LCSD processes and model-driven software development (MDSD) [Cabot 2020]. We expect there is some overlap between good practices in the contexts of LCSD and MDSD, leading to potential patterns down the line. The target audience of this work is industry professionals who design and create low-code platforms. We expect that these patterns result in high-quality software, increasing the usability faced by those who use them. In addition, however, scholars and academics who are interested in LCSD may take away something valuable from this.

The remainder of this document is structured as follows: Section 2 provides background regarding MDSD, Section 3 discusses the current landscape of LCSD both in academia and in the industry, Section 4 talks about the pattern mining approach to be taken, Section 5 provides early results on MDSD patterns that can be applied to LCSD contexts, and Section 6 provides conclusions to this work.

2. MODEL-DRIVEN SOFTWARE DEVELOPMENT

This section provides background on MDSD, detailing its main concepts and how software can be developed using models.

Model-Driven Software Development (MDSD) is an approach that uses models as its primary development artefacts. In the tasks that comprise this approach, models are generated and modified through the use of generation tools and model transformations, with the goal of developing software.

[1] There are also "no-code" tools, similar to LCSD tools, that do not include any user-written source code. For the sake of simplicity, we will be grouping no-code tools along with low-code ones in this work.

2.1 Models and transformations

The concepts of models and abstractions are closely related. We can see the former as abstractions of systems, whether real or language-based, that allow us to make predictions or inferences [Kühne 2006]. They can be defined as a simplified or partial representation of a certain concept, helping us understand it under a particular point of view [Stahl and Völter 2006].

In software development, models help us make sense of the complexity of software artefacts, deal with the increasing prevalence of software in our lives, and interact with any non-developer colleagues we may have to collaborate with in a given project that may not have strong technical coding skills [Brambilla et al. 2012].

Models can also model other ones; these are called metamodels, with their existence defining the modelling language and ensuring that the regular models are coherent. The act of metamodelling can be recursive, and one can have several levels of metamodels conforming to other meta-metamodels [Brambilla et al. 2012].

Transformations are defined at the metamodel level [Brambilla et al. 2012], manipulating automatically input models to produce output models. These manipulations conform to specifications and have specific intents [Lúcio et al. 2016], such as, for example, aiding in the generation of source code or helping understanding concepts between two perspectives. Along with models, transformations make MDSE operations possible [Brambilla et al. 2012].

Model-driven development (MDD) is one of several different model-driven approaches, along with model-driven architecture (MDA), model-driven engineering (MDE), and model-based engineering (MBE) [Stahl and Völter 2006]. MDD uses models as the primary development artefacts, thanks to generation tools and transformations. MDE also places importance on models but includes other engineering tasks. MBE is even less strict, not even necessarily having models as the primary development artefacts. Finally, MDA is a particular version of MDD, proposed by the Object Management Group (OMG) and relying on OMG standards [Brambilla et al. 2012]. One thing worth noting is that MDD can also be applied to other fields, such as model-driven product engineering (MDPE) [Brambilla et al. 2012]. This paper focuses on software engineering, and so it uses the acronym MDSE to discuss this modelling approach.

2.2 Software development automation

One of the most prominent use cases of MDSE is the automation of the software development process; using model-driven techniques permits the automation of much of the steps in this process, facilitating tasks ranging from dealing with requirements, testing, and reaching the final application binaries [Brambilla et al. 2012].

The automation process can be done by continuously iterating through models of the desired application. They become more refined, and the desired result starts to take shape, all thanks to transformations [Brambilla et al. 2012]. There is also the need for executable models (e.g., using executable UML), which ensure the semantics of the model are specified [Brambilla et al. 2012].

[Brambilla et al. 2012] identify two main avenues to pursue software development automation: source-code generation and model interpretation.

2.2.1 *Source code generation.*

As its name suggests, this strategy implies that the program will run from source code that was generated based on the models provided. The resulting code can then be refined for later compilation into the executable binaries, which leads to a running program.

One can try to generate the entirety of the source code but, if not possible (due to, for example, generator limitations or incomplete models), partial code generation can be done, which still has its uses [Brambilla et al. 2012]. A few ways to maximise the utility of partial code generation include generating parts of the system in their entirety instead of the whole system partially, defining protected areas to be edited by the developer, and having tools to synchronise changes between the source code and the model [Brambilla et al. 2012].

Source code generation has some advantages compared to model interpretation, including the protection of intellectual property (as the source code can be shared with a client while protecting the models), the portability of

source-code and avoidance of vendor lock-in, reuse of existing programming artefacts, ease of maintainability, and good performance [Brambilla et al. 2012].

Meanwhile, one should be mindful that source-code generation results in code that may not be initially recognisable by developers, as they did not write it and may not be structured in a way that would be intuitive [Brambilla et al. 2012].

2.2.2 *Model interpretation.* Just like with source-code generation, this strategy also has a self-explanatory name; the model will be fed to an interpreter, which parses it and executes the logic present within without generating any source code [Brambilla et al. 2012].

Using an interpreter makes it possible for developers to tweak the model on the fly, akin to live software development tools. Another advantage of interpretation is that there is no further need to debug any code, as it is not generated. In addition, [Brambilla et al. 2012] state that debugging models is more manageable, and their presence also brings a higher level of abstraction.

However, it is worth noting that one may be subject to vendor lock-in regarding the interpreter, and that runtime performance may not be equal to the one found in scenarios where source-code generation was used. Regardless, the performance interpretation brings is enough for most scenarios [Brambilla et al. 2012].

3. LOW-CODE SOFTWARE DEVELOPMENT

This section provides background information about LCSD, with a brief summary of its history, the current state of the low-code market, and its relationship with MDSD.

Low-code software development (LCSD) approaches make use of low-code tools, which provide their users with a way to develop software by writing a limited amount of lines of code. Some tools even advertise themselves as supporting a "no-code" approach, enabling users to execute development tasks without any traditional, code-based programming.

The definition of what "low-code" is is still not uniform nor precisely defined in the literature. Luo et al. researched internet discussion forums to understand the perspective practitioners have and found that there is currently no standard definition of low-code. However, they identified common concepts such as visual, drag-and-drop-based tools, and writing low amounts of code. Typical applications developed using low-code tools include mobile applications and web apps. On the other hand, [Cabot 2020] describes low-code in a position paper as a more restrictive view of MDD, given the restrictions placed by low-code vendors and the types of applications commonly built with such toolsets.

3.1 Origins of low-code

The concept of low-code tools has existed for a few decades, appearing as early as the 1980s [Martin 1982]. Having abstraction layers in place can help a developer not worry about the finer details of a specific part of a program [Berzins et al. 1986], such as, for example, using libraries to handle tasks regarding dates, timestamps, or mathematical operations.

Developers have always had a tendency to move from low-level languages to higher-level ones, as they tended to be more practical tools: an early example is the move from assembly code to the first languages such as Fortran, in the 1950s [Woo 2020]. As programming languages matured, newer ones appeared, with some being more suited to specific tasks, such as R with statistical computing [R Core Team 2000] and PHP with web development [Welling and Thomson 2003].

Early attempts that could be seen as precursors to modern LCSD tools include computer-aided software engineering (CASE) tools, which shared philosophies with computer-aided design tools. CASE tools peaked in the early 1990s before losing prominence with the decline in the use of mainframes [Iivari 1996].

The concept of end-user programming is closely related to LCSD, although it appeared earlier, boosted by the arrival of Web 2.0 [Fischer 2009]. One of the key features of Web 2.0 was a more widespread ability to interact and

make changes to websites, as is the case with social networks, comment sections, and content sharing websites. This increased interactability showed that web-based development tools could be possible [Fischer 2009].

3.2 The current low-code market

By this time, many LCSD platform vendors already existed, such as OutSystems [OutSystems 2021a] and Mendix [Mendix 2021c]. As the market grew, Forrester Research coined the term "low-code" in a report where they analysed the vendors' current positions [Richardson et al. 2014]. Currently, both Forrester [Rymer and Koplowitz 2019] and Gartner [Vincent et al. 2019] identify four low-code vendors as leaders: OutSystems, Mendix, Microsoft Power Apps, and Salesforce.

LCSD vendors showcase several types of applications users can develop using their tools, ranging from CRUD (create, read, update, delete) backend applications to mobile apps. They can be used in different fields such as education [Mendix 2021b] and retail [Bhangar 2020], but also in more sensitive areas such as government [OutSystems 2021c], finance [Mendix 2021a], and critical systems [OutSystems 2021b]. Gartner identifies the low-code market as an expanding one, predicting that LCSD will account for almost two-thirds of application development by 2024 [Vincent et al. 2019].

3.3 Relationship with model-driven software development

Considering what was mentioned in Section 2, we can see that the low-code paradigm may have a lot in common with MDSD, particularly with the latter's potential to automate software development. Additionally, the typical low-code app development workflow may also be reminiscent of modelling tasks.

The modelling scientific community has also expressed interest in low-code research, with the 2020 edition of the MODELS conference had their first workshop dedicated to low-code [Guerra et al. 2020]. As mentioned earlier, [Cabot 2020] states that LCSD and MDSD have a lot in common, with the former being a more restrictive version of the latter.

As such, given that these two concepts are, at their core, intimately related, we feel that this is a special case where one can take inspiration from MDSD to learn more about LCSD, and do better with LCSD tools. This relationship can be made useful even further, permitting us to potentially look at existing MDSD patterns to derive new patterns about LCSD.

3.4 Relationship between low-code tools and domain-specific languages

Domain-specific languages (DSLs), such as, for example, HTML, SQL, and GraphViz [Brambilla et al. 2012; Fowler and Parsons 2010], are a concept that, due to their nature, deserves to be analysed taking the existence of LCSD tools. As their name implies, DSLs are languages specifically made to be used in a given domain. They exist in contrast to general-purpose languages (GPLs), which are used in situations that require a more expanded scope [Brambilla et al. 2012; Fowler and Parsons 2010].

DSLs tend not to have complex syntaxes, focusing on the relevant elements and entities of the domain where it belongs. They balance the needs for ease of use and performance, being readable by humans and not requiring strong technical skills while also being executable by machines [Fowler and Parsons 2010; Khorram et al. 2020]. With this, their focus on good usability increases productivity by lowering the distance between thinking and doing [Barišić et al. 2018].

Considering this information, it is worth comparing and contrasting DSLs with LCSD tools. They both share a focus on being accessible to domain experts and people without strong technical skills. In addition, they are also closely related to models; low-code tools tend to include them in their tech stack while models can be defined using DSLs [Brambilla et al. 2012]. However, differences also exist. While a DSL is restricted to a specific domain, as implied by its name, low-code tools can be, similarly to GPLs, used to ed to develop software for various fields, as mentioned in Section 3.2.

As they share both similarities and differences, we must not think that these concepts are mutually exclusive. DSLs can be a part of a workflow that revolves around a low-code tool. Examples of this can be found in the OutSystems [Henriques et al. 2018] and Mendix [den Haan 2020] platforms, which provide DSLs to abstract certain logic processes.

4. MINING APPROACH

This work aims to find good practices that benefit the process of developing low-code platforms. As such, given the close relationship between LCSD and MDSD mentioned earlier in Section 3.3, we will start mining for patterns based on the literature.

Research about MDSD is quite extensive, and there have been authors in the past who documented many patterns for MDSD [Völter and Bettin 2004]. In addition, other types of good practices may be found in the literature even if not under the form of patterns, and they may as well be considered.

As such, we intend to analyse existing patterns to see if they are also applicable to LCSD, and to discover other new practices that hadn't been documented yet. This change in contexts is possible due to the high number of similarities between these two areas.

At a later stage, we aim to perform interviews with makers of LCSD tools, with the goals of validating the information mined so far and receiving information derived from their experience working on LCSD tools.

This paper addresses only the pattern mining phase of the pattern writing process. As described by [Iba and Isaku 2016], the pattern mining phase is the first of three, being followed by pattern writing, which includes writing tasks based on information acquired previously, and pattern symbolising, where the accompanying elements such as the name and visual elements are defined in a meaningful way.

As the first part of the process, pattern mining tasks revolve around gathering information for potential patterns. This information is typically based on empirical knowledge based on interviews and questionnaires [Iba and Isaku 2016]. However, the pattern mining performed in this work is based on the existing literature.

As there are already patterns detailing good practices about MDSD, we can expect that they are relevant in this field. The challenge is, thus, to find out if we can make use of some patterns in related fields, as is the case with LCSD. Being able to make this adaptation could also be interesting for other areas.

5. PATTERN CANDIDATES

Following the methodology laid out in Section 4, we set out to find in the literature examples of proven good practices for LCSD and, based on Section 3.3, we found that exploring the literature on MDSD was an appropriate place to start.

In this initial phase, we attempted to analyse existing MDSD patterns to see if they were applicable in low-code contexts. Searching through articles about such patterns led us to an article by [Völter and Bettin 2004], where they approached topics such as domain and platform development. Despite the paper's age, we considered it was a good starting point.

This section thus describes some of the patterns documented in [Völter and Bettin 2004] that we feel can be applied to the development of low-code platforms, giving our reasoning behind them and, if possible, also supplying examples from current LCSD vendors of their instantiation.

The first patterns from [Völter and Bettin 2004] we found relevant were ITERATIVE DUAL-TRACK DEVELOPMENT and EXTRACT THE INFRASTRUCTURE. In the first pattern, the authors note the need for parallel development of applications and the infrastructure, making it possible for the infrastructure developers to receive feedback based on the application development progress. This pattern's solution calls for the employment of the second pattern, which advises developers to extract transformations from an application developed regularly to kickstart the MDSD infrastructure development [Völter and Bettin 2004]. We feel that these patterns could apply to low-code as a conduit for the LCSD platform's requirements and models: as the teams behind the platform want to ensure users

Table I. Patlets of the pattern candidates mined from the literature.

Tentative pattern name	Patlet	Based on
TANDEM DEVELOPMENT	Develop the low-code tools alongside example applications that could be developed using such tools. From the example applications, low-code tool developers should derive the architecture of the tools they are developing.	ITERATIVE DUAL-TRACK DEVELOPMENT, EXTRACT THE INFRASTRUCTURE
FOUNDATIONAL META-MODEL	Implement a formally defined and unambiguous meta-model as a foundational element of the low-code tools to ensure a consistent behaviour among the applications developed using the tools.	FORMAL META-MODEL, IMPLEMENT THE META-MODEL
TWO-STAGE BUILD	To improve flexibility, divide the build process into two phases. The first one deals with dependency-related tasks, while the second one interprets the program's logic and performs generation and execution tasks.	TWO-STAGE BUILD
USER CODE BLACK BOX	Providing users with logic black boxes with defined entry arguments and return variables can provide greater flexibility.	SEPARATE GENERATED AND NON-GENERATED CODE
PLUG-IN VARIETY	Provide users with plug-ins for their applications to enable greater flexibility regarding the platforms they are built for. Possible examples include mobile OS camera APIs and AI-based operations.	RICH PLATFORM-SPECIFIC DOMAIN
NARROWLY-SCOPED TOOLS	Divide the different development domains into different yet connected views, such as, for example, a screen for the program's logic and another for the GUI.	TECHNICAL SUBDOMAINS

can use their product to develop applications, having real-world examples can give insights regarding the LCSD toolset's architecture and structure.

We have also found the patterns FORMAL META-MODEL and IMPLEMENT THE META-MODEL [Völter and Bettin 2004] to be relevant to low-code contexts. In the first pattern, the authors advocate for the presence of a formally defined and unambiguous metamodel to ensure the DSL and application models are correct regarding their domain. The second pattern describes the validation and further implementation of the metamodel. Such a metamodel can prove beneficial for low-code platforms, as the behaviours of the applications that users develop must remain consistent based on the tools they used.

As described by [Völter and Bettin 2004], the pattern TWO-STAGE BUILD approaches the problem of ensuring transformation processes are not impacted by differences in the dependencies and specifications of the different parts of a system. The authors suggest separating the model-driven generation process between a first phase that deals with configuration steps and a second one that executes the transformations [Völter and Bettin 2004]. Low-code platforms have support for different types of frameworks and technologies, and, as such, one can expect to see some behind-the-scenes separation between the generation of the user-defined program logic and dependency-related tasks.

The pattern SEPARATE GENERATED AND NON-GENERATED CODE addresses what may happen with partial code generation (see Section 2.2.1), where parts of the program are implemented manually. The problem stated by the pattern raises issues of consistency, versioning and overwriting the handwritten code, among others. The proposed solution by [Völter and Bettin 2004] is to have separate files for generated and handwritten code, urging developers to avoid making changes to generated code files. In LCSD environments, most platforms include specific elements where users can input their own handwritten code that is encapsulated in a logic black box and included in the low-code application.

When dealing with various target platforms, the required transformations may vary in complexity to ensure the logic remains. In the pattern RICH PLATFORM-SPECIFIC DOMAIN, [Völter and Bettin 2004] suggest dealing with this problem by bundling frameworks, libraries, and interpreters, among others, in a domain-specific application platform.

We can see the main ideas of this pattern in practice in the LCSD context, as low-code platforms provide users with plug-ins for different types of activities, such as mobile camera APIs and AI-based operations [OutSystems 2020].

The pattern TECHNICAL SUBDOMAINS addresses the complexity that can arise if a model describes a plurality of aspects ranging from, for example, business rules to GUI design. [Völter and Bettin 2004] advise dividing the system's structure into several subdomains, each one with its own metamodel and DSL. This separation of domains can also be seen in some low-code tools, where users do not model their programs' logic on the same screen as they design their GUI. Instead, they can have different tabs to go back and forth and create connectors between the different types of elements.

As mentioned earlier, this work focuses on the pattern mining phase of the pattern writing process. This set of existing MDSD patterns shows us that there is potential for finding valuable information for good practices for LCSD tool makers in the literature related to MDSD. As such, we compiled the information mined from these patterns into a set of proto-patterns, which are the main results of the pattern mining phase. These proto-patterns are accompanied by their own patlets and can be found in Table I.

6. CONCLUSION

Low-code software development makes it possible for users to develop applications by having to program a limited number of lines of code, reducing the need for developers to manage program dependencies and allowing domain experts without technical skills to have a more hands-on role in systems development. Model-driven software development, powered by models and transformations, can be used to automate the development processes, either by code generation or by model interpretation. In the literature, there seem to be signs that LCSD and MDSD share several similarities.

Scientific research on LCSD is still growing, and, given this aforementioned relationship, we set out to search for MDSD patterns that could apply to low-code contexts as well. In this initial phase, we explicitly looked for patterns, leading us to analyse a set of patterns by [Völter and Bettin 2004]. We found that several of them seemed to either be applicable or were already found in practice, which is a positive result.

As for future work, we aim to extend the pattern mining based on literature reviews by also taking into account non-pattern-related works and conducting interviews with low-code platform makers.

Limitations of this work include the currently narrow focus of the literature review, which we aim to address with the future work described above.

ACKNOWLEDGMENTS

We would like to thank Michael Weiss for his support during the shepherding process, as well as all the Writer's Workshop participants for their ideas and inputs.

REFERENCES

Ankica Barišić, Vasco Amaral, and Miguel Goulão. 2018. Usability Driven DSL Development with USE-ME. *Computer Languages, Systems & Structures* 51 (Jan. 2018), 118–157. DOI:http://dx.doi.org/10.1016/j.cl.2017.06.005

Valdis Berzins, Michael Gray, and David Naumann. 1986. Abstraction-based software development. *Commun. ACM* 29, 5 (May 1986), 402–415. DOI:http://dx.doi.org/10/bxszft

Sameer Bhangar. 2020. IKEA Sweden – Reimagining the customer experience with Microsoft Power Platform | Microsoft Power Apps. (May 2020). https://powerapps.microsoft.com/pt-pt/blog/ikea-sweden/

Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice*. Morgan & Claypool.

Jordi Cabot. 2020. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20)*. Association for Computing Machinery, 1–3. DOI:http://dx.doi.org/10/gh78b6

Johan den Haan. 2020. Low-Code Principle #1: Model-Driven Development. https://www.mendix.com/blog/low-code-principle-1-model-driven-development/. (Jan. 2020).

Gerhard Fischer. 2009. End-User Development and Meta-design: Foundations for Cultures of Participation. In *End-User Development (Lecture Notes in Computer Science)*, Volkmar Pipek, Mary Beth Rosson, Boris de Ruyter, and Volker Wulf (Eds.). Springer, 3–14. DOI:http://dx.doi.org/10/dh3gm9

Martin Fowler and Rebecca Parsons. 2010. *Domain-Specific Languages*. Pearson Education.

Esther Guerra, Association for Computing Machinery, and Special Interest Group on Software Engineering. 2020. *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. https://dl.acm.org/action/showBook?doi=10.1145/3417990

J. Hejderup, A. Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)* (2018). DOI:http://dx.doi.org/10/gkx3qs

Henrique Henriques, Hugo Lourenço, Vasco Amaral, and Miguel Goulão. 2018. Improving the Developer Experience with a Low-Code Process Modelling Language. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems - MODELS '18*. ACM Press, New York, New York, USA, 200–210. DOI:http://dx.doi.org/10.1145/3239372.3239387

Takashi Iba and Taichi Isaku. 2016. A Pattern Language for Creating Pattern Languages: 364 Patterns for Pattern Mining, Writing, and Symbolizing. In *ACM Reference Format*. 1–63.

Juhani Iivari. 1996. Why are CASE tools not used? *Commun. ACM* 39, 10 (Oct 1996), 94–103. DOI:http://dx.doi.org/10/c2mh8z

Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. 2020. Challenges & Opportunities in Low-Code Testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/3417990.3420204

Thomas Kühne. 2006. Matters of (Meta-) Modeling. *Software & Systems Modeling* 5, 4 (Dec 2006), 369–385. DOI:http://dx.doi.org/10/c9znng

Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model transformation intents and their properties. *Software & Systems Modeling* 15, 3 (Jul 2016), 647–684. DOI:http://dx.doi.org/10/f8wc6h

James Martin. 1982. *Application Development Without Programmers* (1st ed.). Prentice-Hall.

Grigori Melnik and Frank Maurer. 2004. Direct verbal communication as a catalyst of agile knowledge sharing. *Proceedings of the Agile Development Conference, ADC 2004* (2004), 21–31. DOI:http://dx.doi.org/10.1109/ADEVC.2004.12

Mendix. 2021a. Business Development Bank of Canada Lends Entrepreneurial Spirit to Loans Process. (2021). https://www.mendix.com/customer-stories/bdc/

Mendix. 2021b. Low-Code Empowers Developers to Transform NCSU's App Dev Culture and Exceed Business Expectations. (2021). https://www.mendix.com/customer-stories/ncsu/

Mendix. 2021c. Mendix: We Help Enterprises Achieve their Digital Goals with Low-code. (2021). https://www.mendix.com/company/

OutSystems. 2020. Extensibility and Integration. (Oct 2020). https://success.outsystems.com/Documentation/11/Extensibility_and_Integration

OutSystems. 2021a. About OutSystems. (2021). https://www.outsystems.com/company/

OutSystems. 2021b. Flight Dispatch Control Center at TAP | OutSystems Case Study. (2021). https://www.outsystems.com/case-studies/flight-dispatch-control-center/

OutSystems. 2021c. Oakland Delivers Transformative Digital Services for Residents to Save $1 Million. (2021). https://www.outsystems.com/case-studies/oakland-transforms-city-services-residents/

R Core Team. 2000. R language definition. *Vienna, Austria: R foundation for statistical computing* (2000).

Clay Richardson, John R Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. 2014. New development platforms emerge for customer-facing applications. *Forrester: Cambridge, MA, USA* (2014).

John R Rymer and Rob Koplowitz. 2019. *The Forrester Wave™: Low-code Development Platforms for AD&D Frofessionals, Q1 2019*. Forrester Research.

Raquel Sanchis, Óscar García-Perales, Francisco Fraile, and Raul Poler. 2020. Low-Code as Enabler of Digital Transformation in Manufacturing Industry. *Applied Sciences* 10, 11 (Jan 2020), 12. DOI:http://dx.doi.org/10/gh778q

Thomas Stahl and Markus Völter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley.

Paul Vincent, Kimihiko Iijima, Mark Driver, Jason Wong, and Yefim Natis. 2019. Magic quadrant for enterprise low-code application platforms. *Gartner report* (Aug 2019).

Markus Völter and Jorn Bettin. 2004. Patterns for Model-Driven Software-Development.. In *Proceedings of the 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004)*. 525–560.

Luke Welling and Laura Thomson. 2003. *PHP and MySQL Web development*. Sams Publishing.

Marcus Woo. 2020. The Rise of No/Low Code Software Development—No Experience Needed? *Engineering* 6, 9 (Sep 2020), 960–961. DOI:http://dx.doi.org/10/gh78bb

Received May 2021; revised September 2021; accepted September 2024

PLoP'21, October 4-7, Virtual Online. Copyright 2021 is held by the author(s). HILLSIDE 978-1-941652-17-6