Agile bottom-up development of domain-specific IDEs for model-driven development*

Steffen Vaupel, Daniel Strüber, Felix Rieger, Gabriele Taentzer

Philipps-Universität Marburg, Germany
{svaupel,strueber,riegerf,taentzer}@informatik.uni-marburg.de

Abstract. Diminishing time-to-market and rapidly evolving technology stacks stretch traditional software development methods to their limits. In this paper, we propose a novel process for bottom-up development of domain-specific IDEs based on agile principles. It aims to enable a fine-grained co-evolution of domain-specific modeling languages (DSMLs) and their model editors and code generators. We illustrate our approach by iteratively developing an IDE for model-driven development of mobile applications. As a key success factor for continuous DSML development, we determine the automated deduction of migration scripts for all dependent artifacts of a DSML evolution step.

1 Introduction

Vastly increasing numbers of applications and users make the development of mobile applications one of the most important fields in software engineering. In this field, short time-to-market, differing platforms and rapidly emerging technologies stretch traditional software development methodologies to their limits. A viable research direction to tackle these challenges involves the combination of model-driven development (MDD) [1] and agile software development [2]: Aligning the domain-specific, platform-independent abstractions provided by MDD with agile principles such as *quick response to change* and *early delivery* promises a high potential to handle the requirements of rapidly evolving software domains.

Experience has shown that model-driven and agile practices complement each other well during the development of individual applications [3,4]. In these scenarios, a static modeling language was assumed; the evolution of this language was not considered. Yet in the reality of rapidly evolving software domains, the evolution of domain-specific modeling languages (DSMLs) has become an unavoidable fact. A major challenge is the co-evolution of the enabling technologies for DSMLs – in particular, their IDEs. Domain-specific IDEs include model editors and code generators. The development of these components is facilitated by a wealth of meta-tools: GMF [5], Sirius [6] and Xtext [7] for editor, Xtend [7] and EGL [8] for generator development.

* This work was partially funded by LOEWE HA project no. 355/12-45 (State Offensive for the Development of Scientific and Economic Excellence) on "Platform Independent Mobile Augmented Reality" (PIMAR).

In the state-of-the-art process of using these meta-tools, the developer analyzes one or several reference applications and extracts knowledge to specify the IDE components. This approach, referred to as *bottom-up development* [9], assumes that full reference applications are provided upfront, which is reasonable if the involved technologies and user requirements are stipulated at the beginning of the project. In rapidly evolving software domains, however, this assumption does not hold anymore: Due to changing user demands and underlying technologies, a DSML is exposed to evolution during its whole lifespan. The following research question arises: **How can domain-specific IDEs be developed systematically in the presence of modeling language evolution?**

In this paper, we aim to address this question. Our main contribution is an *agile bottom-up process* for the development of domain-specific IDEs, focussing on the co-evolution of a DSML, its editors and code generators. The key idea is to organize language evolution into fine-grained evolution steps: In each step, prototype models are employed to generate one or several application prototypes. The developer manually modifies the prototypes as required for the evolution step. Then, the IDE developer identifies aspects concerning the DSML, editors, and code generators. These aspects are used as input for their synchronous evolution. Afterward, the application prototype is no longer required. The process is not designed for, but can be aligned with a specific agile methodology, such as Scrum. As our second contribution, we integrate this process in the overarching vision of a *three-tier process model*, involving the development of meta-tools, tools and applications. Our third contribution is an *experience report* concerning the development of a DSML and domain-specific IDE for mobile applications.

The remainder of this paper is structured as follows: In Sect. 2, we present the process, outlining its main activities. Sect. 4 introduces the three-tier process model. Sect. 3 reports on experiences we made applying the process in a research project on the development of mobile applications. In Sect. 5, we discuss related work. In Sect. 6, we conclude and elaborate our plans for future work.

2 Agile bottom-up development of domain-specific IDEs

Domain-specific IDEs are an enabling technology for the model-driven development of specific applications. At a minimum, a domain-specific IDE comprises one or several model editors for the underlying DSML and code generators for one or several target platforms. Additional components may include version management, testing and debugging tools.

In this section, we give an overview of the proposed agile bottom-up IDE development process: First, to define an initial DSML and IDE, a *domain analysis* is carried out, involving the extraction of domain concepts and generator templates from existing reference applications. Second, in the course of *continuous language and IDE development*, developers perform evolution steps, including the generation and modification of prototypes and successive evolution of the DSML and IDE. Third, evolution steps may require a follow-up *migration* step to reconcile inconsistencies introduced in existing prototype app models during

the evolution step. These app models are model-based descriptions of prototypes. For each of these activities, we outline the involved manual and automated tasks and the tools supporting these tasks.

2.1 Domain analysis

Different rationales can motivate a change to model-driven development: First, in large software projects, a lot of boilerplate code may exist due to use cases showing certain similarities. Second, a number of separate applications might show similarities in structure and behavior. Third, it may be required to deploy one individual application to several target platforms. In each of these scenarios, the abstraction level of development can be lifted by using DSMLs with code generation facilities. The initial step to establish such a DSML based on one or several reference applications is called *domain analysis*.

Domain analysis involves three steps: *Quality assurance*, *domain concept identification* and *template extraction*. *Quality assurance* ensures that the existing applications exhibit high quality, rendering them suitable as reference applications for code generation. This task involves the identification of anti-patterns and refactoring towards design patterns. During *domain concept identification*, concepts recurring throughout the reference applications are identified; they are reflected as model elements in the DSML. The aim of *template extraction* is to specify generator templates: A generator template represents a unit of code with gaps. The gaps are filled during application development by the generator, using application-specific information given by instances of the DSML.

Quality assurance can be partly automatized using static analysis tools supporting the detection of anti-patterns and code smells [10]. A promising technology to detect recurring concepts is automated clone detection [11]. To our knowledge, no specific tools exist to manage the extraction of templates based on reference applications, leaving it a fully manual step.

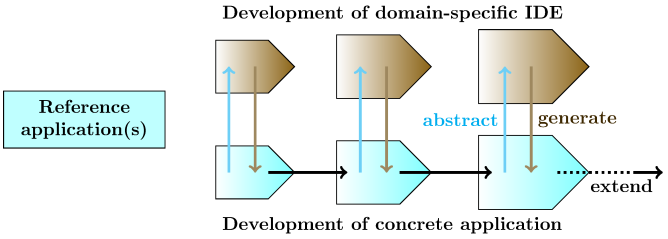


Fig. 1. Agile MDD process in action: Fine-grained evolution steps

2.2 Continuous language and IDE development

We propose to develop IDE components, notably the model editors and a code generators, in fine-grained iterations (cf. Figure 1): First, developers decide on the next feature that should be supported by the DSML and its IDE. Then, the IDE developer generates one or more prototypes from app models and manually

extends these prototypes by the code required to implement this feature. The extension is then analyzed and results in a synchronous evolution step of the DSML and its IDE.

In this approach, the IDE developer is required to inspect the generated prototypes and extend them to incorporate new features. Therefore, it is essential that generated applications are working software systems and that the generated code is of good quality, i.e. well structured and easy to understand. As an aid to support the comprehension of the generated code, we provide a mapping between DSML elements and the individual code generator templates involved in implementing these elements. In our experience, such a mapping has proven itself valuable.

Various meta-tools allow specifying editors, transformations and further tools. GMF, Xtext and Sirius support high-level specification of graphical and textual model editors. ATL [12], Henshin [13], ViaTra [14] and many more support the specification of model translations, simulations and optimizations. There are further meta-tools for IDE components such as EMF Refactor [15] for model quality assurance tools, and EMFCompare [16] and SiLift [17] to support version management features. Since continuous language evolution results in continuous IDE evolution, co-evolution processes are important to be considered and to be supported by tools. Therefore, meta-tools are needed for migrating all dependent artifacts such as instance models, model transformations, especially code generators, model editor specifications, model quality assurance and version management tools. Future research is needed to automate these migrations.

2.3 Migration of app models

Since app models are directly dependent on the evolution of their DSML, they have to be kept consistent with the DSML. One possibility is to only make changes that do not necessitate adapting the software systems on lower layers. However, this might lead to compromise solutions in language design. The alternative is to migrate them accordingly which allows to freely develop DSMLs.

Co-evolution tools such as Edapt [18] and Flock [19] are available, but still show specific limitations: For instance, Edapt supports the evolution of meta-models using pre-defined operations and the automatic deduction of a suitable migration script for all instance models. However, integrating these pre-defined operations requires a significant adoption of existing modeling workflows and tools. As a consequence, migration processes are currently performed by hand, which can be tedious and error-prone. In the future, we aim to provide tool support for the automated co-evolution of app models. We intend to base these tools on results concering the co-evolution of language meta-models and instance models [20,21].

3 Experience Report

In this section we provide an experience report concerning the development of a DSML and a domain-specific IDE for mobile applications. First, we pro-

vide a domain description, including the resulting modeling approach. Second, to demonstrate the application of meta-tools within the proposed process, we describe one iteration of language and IDE development. Third, we report on continuous language and generator extension. To determine the usefulness of the application of meta-tools, we investigated how these tools can shorten iterations by allowing the automatic (re)generation of code for certain IDE artifacts. We tracked the size of an editor and a code generator and the number of covered use cases during the development of the IDE. Finally, we discuss the open problems.

3.1 Domain description

Along with our industrial partner *advenco* [22], a medium-sized software consulting company, we discovered the domain by analyzing two of their products. The first product is a multimedia guide for tourists, guiding them through places of interest such as museums, exhibitions or towns. A second product allows defining business processes using mobile and further devices. As a result, we define several characteristics of the application domain that should be supported by the IDE:

Core functionality:
- provide language elements for data, behavior, and GUI modeling
- generate apps that operate in multiple contexts (e.g., online or offline) and support user-provided content (e.g. current information about tourist events)

Enhanced functionality:
- provide access to device sensors (e.g. optical barcode and RFID tags)
- provide functionalities for augmented reality applications
- provide functionalities for e-Learning applications (e.g., vocabulary training, self-assesment, safety training, etc.)

Supporting new technology:
- new platforms and versions
- new kinds of devices (e.g., wearables, tablets, embedded Android, etc.)

On the basis of our domain analysis, we support different modeling aspects and generate native apps that can be flexibly configured by users. Figure 2 illustrates the resulting modeling approach. It comprises two code generators, one for Android and one for iOS, which generate runnable applications (100% of the application code is generated from the models). Detailed information regarding the abstract and concrete syntax of data, behavior, and GUI models is provided in [23].

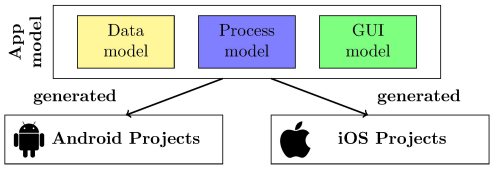
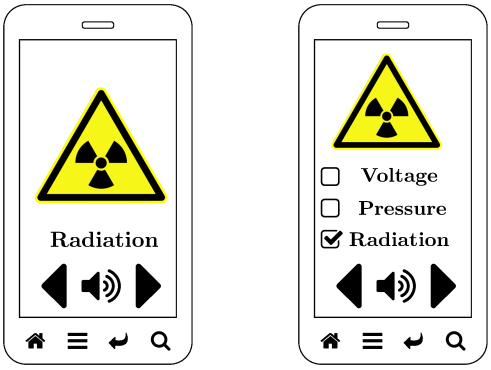


Fig. 2. MDD approach for mobile applications

3.2 Example: Language design and development iteration

To illustrate an evolution step, we implement an e-Learning application for safety instructions. The e-Learning application, illustrated in Figure 3, comprises two use cases: The first use case, called *learning mode*, concerns learning using different media types (e.g., videos, pictures and sound recordings). The second use case, called *testing mode*, allows to practice learned content using assignment tasks.



(a) Learning mode (b) Testing mode

Fig. 3. e-Learning application for a safety instruction

The GUI meta-model includes style settings and generic page types serving different purposes. For example, there is a *ViewPage* for displaying objects and an *EditPage* for modifying them. To offer the e-Learning functionality, we introduced an *ELearningPage* into the GUI meta-model. As shown in Figure 3, the purpose of the *ELearningPage* is to present learning content (*learning mode*) or provide a self-test format (*testing mode*). The *ELearningPage* hides the technical details (e.g., playing the sound file, loading the media files, etc.) from the modeler. Adding *ELearningPage* was the only DSML extension.

After having extended the modeling language, the visual editor was regenerated. Then, the code generators had to be adapted to the new language elements. In order to process the new *ELearningPage* element, a new template (*ELearningPageGenerator*) was added to the generator. This template initially generated an empty Android activity or iOS view. We then extended the empty mock class with the required code. After testing, we abstracted the inserted code to code templates. The iteration ended when the generated application fulfilled the same requirements as the extended prototype. The prototype is no longer required.

3.3 Continuous language and generator extensions

The entire process of IDE development usually contains several iterations of the above-mentioned kind. We have changed the DSML 26 times within a period of approx. 18 months to cover the 19 use cases of the two reference applications.

Most of the changes (19 times) were pure extensions of the DSML (from 25 up to 46 elements). We have developed five case studies on mobile apps of different kinds simultaneously through the course of the development of the IDE.

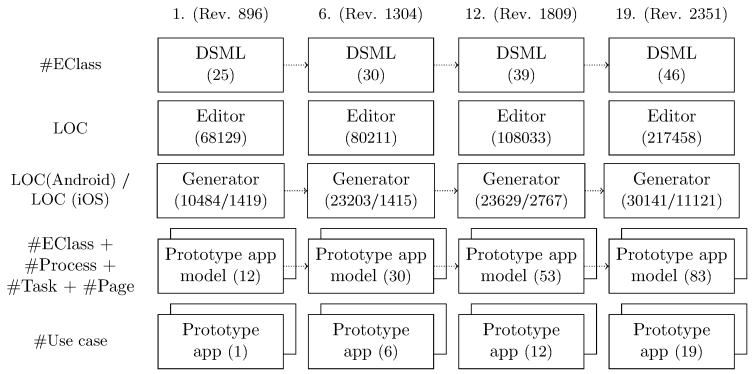


Fig. 4. Continuous language, IDE and prototype extensions

Figure 4 shows four samples of the 19 iterations; we can see the incremental growth of both the IDE and the prototype (generated from the app model). Each iteration realizes an additional use case. During the first six iterations (Rev. 1304), the core functionalities were implemented, followed by the enhanced functionalities.

After each DSML modification, the visual model editor, a key component of the domain-specific IDE, was (re)generated. Figure 4 shows the growth in size of the visual model editor and the Android code generator: In terms of LoC, the model editor was at least three times larger than the code generator during all iterations. Being able to generate this large percentage of the codebase helped to shorten the development cycles: Changes to the DSML were immediately available in the editor. The effect of these changes to the visual editor allowed early detection of language design flaws and appropriate refactorings.

Besides, both platform-specific code generators were extended manually. The iterative approach reduced the complexity of generator construction and extension since developers could focus on recent changes made to the DSML. There were exceptions in the form of cross-cutting modeling elements (e.g., application-wide style settings), which affected many generator templates.

Another advantage of our approach was the co-evolution of the IDE and the app model. IDE users could create app models at an early stage of IDE development and provide test models for the code generators. Figure 5 shows the Eclipse-based domain-specific IDE (visual editor) resulting from continuous language and generator extensions.

3.4 Threats to validity and open problems

Considering *external validity*, it is not ensured that our experiences generalize to arbitrary syntactic and semantic changes in DSMLs. To mitigate this threat,

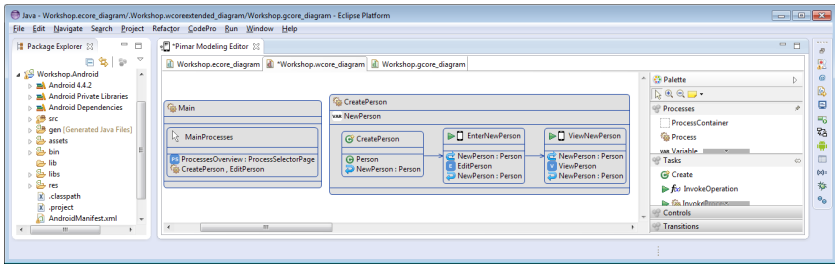


Fig. 5. Domain-specific IDE for model-driven development of mobile applications.

we consider a wide range of use-cases and code generation for multiple target platforms. More experiments are needed to allow general conclusions about the utility of the approach. A threat to *internal validity* is the lack of a case study using traditional processes. Still, we argue that these processes do not provide a course of action to support the co-evolution of DSMLs and IDEs, which renders them unsuitable for our scenario.

The automated deduction of migration scripts for all dependent artifacts is an unsolved problem: For example, changes to the DSML affect the respective templates and existing app models. Currently, all modeling artifacts are migrated manually after each iteration step, a time-consuming and error-prone task.

4 Three-tier agile process model

From a global perspective, taking into account all processes and tools involved in model-driven development leads to three tiers of software development: The development of concrete applications, the development of domain-specific IDEs for model-driven development, and the development of meta-tools to specify IDE components. Although the main contribution of this paper is an agile development process for domain-specific IDEs, we argue that concrete applications and meta-tools shall be developed based on agile principles as well. To quickly respond to new user demands and technologies, all involved software systems should be developed continuously. Their development should incorporate short feedback cycles based on running software.

This set of requirements leads to the stipulation of a three-tier agile development process model, outlined in Figure 6. In the domain of mobile applications, for example, a concrete application is a mobile app that is developed using an IDE for model-driven development of mobile apps. Meta-tools such as editor generators or model-to-text transformation approaches can be used to specify model editors and code generators of these IDEs. The interplay of three different kinds of software projects leads to challenges: Changes in one software project can affect the other ones. These challenges are aggravated by different life cycles and change frequencies. While applications are quickly developed by model-driven development, IDE development is much slower, and meta-tools are usually developed completely independently of concrete IDEs.

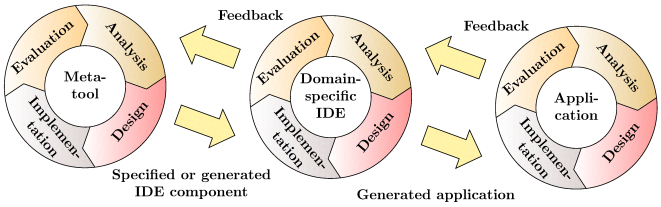


Fig. 6. Three-tier agile software development process model

5 Related work

Several papers describing agile model-driven development, such as [3,4], focus on developing application software by using existing IDEs for MDD, whereas our scope extends to agile development of the domain-specific IDEs themselves.

Völter [24] reflects on best practices for model-driven engineering and DSMLs. Our approach incorporates many of these practices and joins them to the agile development process of domain-specific IDEs. We also focus on minimizing manually written code and co-evolving our language and concepts.

Bagheri et al. [9] propose extracting partial models from stable parts of the system, leaving the remaining parts to be developed manually. They support combining generated and manually written code through their technique called *partial synthesis*. In our process, at the end of an iteration, all manually written code is eventually integrated in the generator, app model, or DSML. Thus, the resulting prototype only contains code produced by the generator.

Fehrenbach et al. [25] consider software evolution by introducing an embedded DSL into a legacy code base. Their approach is orthogonal to ours which assumes that the DSL itself is subject to evolution.

6 Conclusion

We propose an agile bottom-up development process for domain-specific IDEs supporting model-driven development. Starting from reference applications, the domain is analyzed. Prototype applications are continuously generated and extended, leading to continuously evolving domain-specific modeling languages and supporting IDEs. The greater vision is a *process model* for the integrated agile development of applications, domain-specific IDEs and meta-tools. Continuous evolution of all involved software artifacts is key to realize this vision.

As future work, we aim to integrate isolated IDEs for different domains. In an upcoming case study, we will examine the integrated development of data-centric and gaming applications sharing a set of communication points in the IDE and application layers. To specify the communication points, we extended the EMF meta-tool infrastructure with an interface concept [26], allowing us to generate intents between activities and/or services of the involved apps.

References

1. Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2006.
2. Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
3. Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE Software*, 28(2):84–91, 2011.
4. Vinay Kulkarni, Souvik Barat, and Uday Ramteerthkar. Early experience with agile methodology in a model-driven approach. In *14th Int. Conf. on Model Driven Engineering Languages and Systems*, pages 578–590, 2011.
5. Graphical Modeling Framework. `http://www.eclipse.org/gmf`.
6. Eclipse Sirius. `http://www.eclipse.org/sirius`.
7. Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd., 2013.
8. EGL Development Tools (EDT). `http://www.eclipse.org/edt`.
9. Hamid Bagheri and Kevin J. Sullivan. Bottom-up model-driven development. In *35th Int. Conf. on Software Engineering*, pages 1221–1224. IEEE/ACM, 2013.
10. Jernej Novak, Andrej Krajnc, and Rok Zontar. Taxonomy of static code analysis tools. In *MIPRO, 2010 Proc. of the 33rd Int. Conv.*, pages 418–422. IEEE, 2010.
11. Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
12. Eclipse ATL. `http://www.eclipse.org/atl`.
13. Eclipse Henshin. `http://www.eclipse.org/henshin`.
14. Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.
15. EMF Refactor. `http://www.eclipse.org/emf-refactor`.
16. Eclipse EMF Compare. `http://www.eclipse.org/emf/compare`.
17. Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-Preserving Edit Scripts in Model Versioning. In *28th IEEE/ACM Int. Conference on Automated Software Engineering*, pages 191–201. IEEE, 2013.
18. Eclipse Edapt. `http://www.eclipse.org/edapt`.
19. Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In *3rd Int. Conf. on Theory and Practice of Model Transformations*, pages 184–198. Springer, 2010.
20. Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards Synchronizing Models with Evolving Metamodels. In *Ws. on Model-Driv. Softw. Evolution*, 2007.
21. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th Int. Conf. on Enterprise Distributed Object Computing*, pages 222–231. IEEE, 2008.
22. Advenco Consulting GmbH. `http://www.advenco.de`.
23. Steffen Vaupel, Gabriele Taentzer, Jan Peer Harries, Raphael Stroh, René Gerlach, and Michael Guckert. Model-driven development of mobile applications allowing role-driven variants. In *17th Int. Conf. on Model-Driven Engineering Languages and Systems*, pages 1–17, 2014.
24. Markus Völter. Md* best practices. *J. of Object Technology*, 8(6):79–102, 2009.
25. Stefan Fehrenbach, Sebastian Erdweg, and Klaus Ostermann. Software evolution to domain-specific languages. In *Software Language Engineering*, pages 96–116. Springer, 2013.
26. Daniel Strüber, Gabriele Taentzer, Stefan Jurack, and Tim Schäfer. Towards a distributed modeling process based on composite models. *Fundamental Approaches to Software Engineering*, pages 6–20, 2013.