The Death of Object-Oriented Programming

Oscar Nierstrasz(✉)

Software Composition Group, University of Bern, Bern, Switzerland
oscar@inf.unibe.ch
http://scg.unibe.ch/

Abstract. Modern software systems are increasingly long-lived. In order to gracefully evolve these systems as they address new requirements, developers need to navigate effectively between domain concepts and the code that addresses those domains. One of the original promises of object-orientation was that the same object-oriented models would be used throughout requirements analysis, design and implementation. Software systems today however are commonly constructed from a heterogeneous "language soup" of mainstream code and dedicated DSLs addressing a variety of application and technical domains. Has object-oriented programming outlived its purpose?

In this essay we argue that we need to rethink the original goals of object-orientation and their relevance for modern software development. We propose as a driving maxim, *"Programming is Modeling,"* and explore what this implies for programming languages, tools and environments. In particular, we argue that: (1) source code should serve not only to specify an implementation of a software system, but should encode a queryable and manipulable model of the application and technical domains concerned; (2) IDEs should exploit these domain models to enable inexpensive browsing, querying and analysis by developers; and (3) barriers between the code base, the running application, and the software ecosystem at large need to be broken down, and their connections exploited and monitored to support developers in comprehension and evolution tasks.

1 Introduction

Is object-oriented programming dying?

The code of real software systems is structured around a number of interacting and overlapping technical and application domains. As we shall see, this fact is not well supported by mainstream languages and development environments. Although object-oriented software development made early promises to close the gaps between analysis, design and implementation by offering a unifying object-oriented modeling paradigm for these activities, we still struggle to navigate between these worlds. Do the emergence of domain-specific languages (DSLs) and model-driven development (MDD) prove that object-orientation has failed?

In this essay we explore some of the symptoms of this apparent failure, and argue that we need to be bolder in interpreting the vision of object-orientation.

P. Stevens and A. Wąsowski (Eds.): FASE 2016, LNCS 9633, pp. 3–10, 2016.
DOI: 10.1007/978-3-662-49665-7_1

We propose the slogan "Programming is Modeling" and identify a number of challenges this leads us to.

Let us briefly summarize the key symptoms:

There Exists a Large Gap Between Models and Code. In an ideal world, requirements and domain models are clearly visible in the implementation of a software system. In reality, most mainstream programming languages seem to be ill-equipped to represent domain concepts in a concise way, leading to a proliferation of DSLs. Internal DSLs, for example, "fluent interfaces" that exploit the syntax of a host language, are often less fluent and readable than they should be. External DSLs (*i.e.* with their own dedicated syntax) can lead to a "soup" of heterogeneous code that is hard to navigate, understand, and analyse.

MDD represents another important trend, in which high-level models are typically transformed to implementations, but such "model compilers" tend to pay off only in well-understood domains where changes in requirements can be well-expressed by corresponding changes to models.

Mainstream IDEs are Glorified Text Editors. Although software developers spend much of their time reading and analyzing code, mainstream IDEs mostly treat source code as text. In general, the IDE is not aware of application or technical domain concepts, and does not help the developer to formulate domain-specific queries or custom analyses, such as: *Where is this feature implemented? Will this change impact the system architecture? Who is an expert on this part of the code?* Similarly classical development tools belonging to the IDE are unaware of the application domain. A classical example is the interactive debugger, which offers a uniform interface to debugging based on the run-time stack, without any knowledge of the underlying application domain. Although popular IDEs offer plugin architectures that allow third-party developers to integrate new tools into the IDE, the barrier to building such tools remains relatively high, and the application domain models of the underlying code base remain relatively inaccessible.

Programming Languages and Tools Live in a Closed World. Mainstream programming languages assume the world is closed and frozen. Static type systems, for example, assume that the type of an entity is fixed and will never change or evolve. When a type changes, the entire world must change with it. In reality, complex software systems have to cope with evolving and possibly inconsistent entities. Another symptom is the strict divide between "compile time" and "run time" in mainstream programming. For example, it is not possible to navigate seamlessly from a feature of a running system to the code that implements it. Finally, we see that developers often resort to web search engines and dedicated Q&A fora to answer questions that the IDE cannot. We need to acknowledge that code lives within a much larger ecosystem than the current code base.

In this essay we argue that we should revisit the object-oriented paradigm to address these issues by adopting the maxim that "Programming is Modeling." We further propose a number of research challenges along the following lines:

1. *Bring models closer to code* by expressing queryable and manipulable domain models directly in source code;
2. *Exploit domain models in the IDE* to enable custom analyses by developers;
3. *Link the code to its ecosystem* and monitor them both to steer their evolution.

Caveat: we apologize in advance for referencing only little of the vast amount of relevant related work.[1]

2 Bring Models Closer to Code

When we develop and evolve code, we need to comprehend the relationships between requirements that refer to domain models, and the underlying code that realizes those requirements. Ideally we want to see domain concepts directly in the code. We therefore argue that *a program should not just serve to specify an implementation of a set of requirements, but it should encode domain models suitable for querying and analysis.*

This, we believe, was one of the early promises of object-oriented programming as expressed in the 1980s. Nowadays, however, complex software systems are implemented as a soup of mainstream and domain-specific languages. DSLs can be used to address either technical or application domains. Typically several DSLs are needed to address a complex application. Despite the availability of many dedicated DSLs, important aspects of a software system may not be explicitly modeled at all. Notoriously, architectural constraints are implemented with the help of frameworks and architectural styles, but rarely represented explicitly or checked as the system evolves.

Introducing ever more DSLs is not a solution. Having many external DSLs complicates program comprehension and makes it difficult for tools to reason about the relationships between them.[2]

Internal (or embedded) DSLs are hard to achieve because (1) the syntax of many mainstream object-oriented languages does not support well the design of truly fluent interfaces (with some notable exceptions, such as Smalltalk, Ruby, Scala, ...), and (2) design methods emphasize the development of "fluent interfaces," so they can be hard to achieve *post hoc*.

We think that many of these problems have their roots in a *fundamental misunderstanding of the object-oriented paradigm.* While the imperative programming paradigm can be summarized as *programs = algorithms + data structures*, object-oriented programming is often explained (following Alan Kay [8][p 78]) as *programs = objects + messages*. While this is not incorrect, it is a mechanistic interpretation that misses the key point.

In our view, the object-oriented paradigm is better expressed as: *"design your own paradigm"* (*i.e. programming is modeling*). A well-designed object-oriented

[1] A representative selection of related work can be found in the research plan of our SNSF project, "Agile Software Analysis": http://scg.unibe.ch/research/snf16.

[2] Coping with this complexity is one of the goals of the GEMOC initiative [6]. See http://gemoc.org.

system consists of objects representing exactly the domain abstractions that are needed for your application and suitable operations over them (if you like, a many-sorted algebra). Code can be separated into the objects (or "components") representing domain concepts, and *scripts* that configure them [1].

We therefore posit as a challenge to *revive object-oriented programming by viewing OO languages as modeling languages, not just implementation languages.* Rather than viewing DSLs and MDD as the competition, we should encourage the use of OO languages as modeling tools, and even as language workbenches for developing embedded DSLs.[3]

3 Exploit Domain Models in the IDE

Although developers are known to spend much of their development time reading and analyzing code, mainstream IDEs do not do a good job of supporting program comprehension. IDEs are basically glorified text editors.

Developers need custom analyses to answer the questions that arise during typical development tasks [7,16]. Building a dedicated analysis tool is expensive, even using a plugin architecture such as that of Eclipse. Dedicated analysis platforms like Moose [12] and Rascal [9] reduce the cost of custom queries, but they rely on the existence of a queryable model of the target software.

As we have seen in the previous section, even though we would like to see programs as models, they are not in a form useful for querying and analysis, so we need to do extra work to extract these models and work with them.

We see two important challenges. The first is *"Agile Model Extraction"*, *i.e.* the ability to efficiently extract models from source code. This is not just a problem of parsing heterogeneous code and linking concepts encoded in different languages (*e.g.* Java, SQL, XML), but also of recognizing concepts coming from numerous and intertwined domain models. We are experimenting with approximate parsing technology, inexpensive heuristics, and other techniques [10,13] to quickly and cheaply extract models from heterogeneous source code.

The second challenge is *"Context-Aware Tooling"*, *i.e.* the ability to cheaply construct dedicated, custom analyses and tools that close the gap between IDEs and application software. The key idea is, once we have access to the underlying domain model of code (whether it is offered by the underlying infrastructure or obtained by Agile Model Extraction), to make it easy to exploit that model in tools used by developers to produce code, browse and query it, analyze it and debug it. On the one hand, generic core functionality is needed for querying and navigating models. On the other hand, tools and environments need to be aware of the context of the domain model of the code under study so they can adapt themselves accordingly.

An example is the "moldable debugger" which, instead of presenting only a generic stack-based interface to the run-time environment, is aware of relevant domain concepts, such as notifications in an event-driven system, or grammar

[3] See, for example, Helvetia, a workbench for integrating DSLs into the IDE and toolchain of the host language [15].

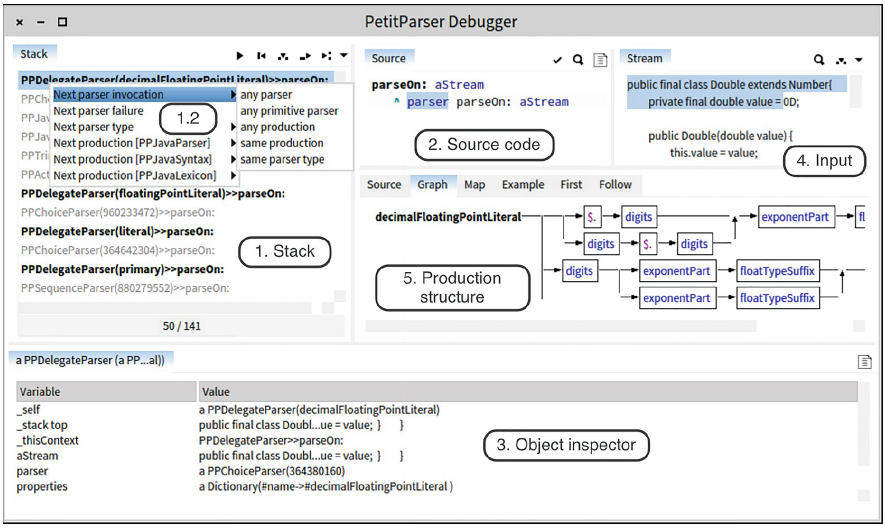

Fig. 1. A domain-specific debugger for PetitParser. The debugging view displays relevant information for debugging parsers ((4) Input, (5) Production structure). Each widget loads relevant debugging operations (1, 2, 4).

rules in a parser [4]. In Fig. 1 we see a screenshot of a domain-specific debugger for *PetitParser*, a parser combinator framework for Pharo Smalltalk [14]. Each widget of the debugger is context-sensitive and loads the appropriate debugging operations for the current context. The debugger is aware of a grammar's production rules and is capable, for example, of stepping to the next production or the next parser failure, rather than simply to the next expression, statement or method. Custom visualizations are also loaded to display the production structure in a suitable way. Custom debuggers can be defined in a straightforward way by leveraging the explicit representation of the underlying application domain.

The same principles have been applied to the "moldable inspector," a context-aware tool for querying and exploring an object space [5]. Domain-specific views are automatically loaded depending on the entities being inspected. As with the moldable debugger, custom views are commonly expressed with just a few lines of code.

In the long run we envision a development environment in which we are not forced to extract models from code, but in which the code is actually a model that we can interact with, query and analyze.

4 Link the Code to Its Ecosystem

Conventional software systems are trapped behind a number of artificial barriers. The most obvious is the barrier between the source code and the running application. This is manifested in the usual program/compile/run cycle. This

makes it difficult to navigate between application features and source code. The debugger is classically the only place where the developer can navigate between the two worlds. It does not have to be that way, as seen in the Morphic framework of Self, in which one may navigate freely between user interface widgets and the source code related to them [11]. (This is just one dramatic manifestation of "live programming", but perhaps one of the most important ones for program comprehension.)

A second barrier is that between a current version of a system and other related versions. In order to extract useful information about the evolution of the system, one must resort to "mining software repositories", but this possibility is not readily available to average developers who do not have spare capacity to carry out such studies. Furthermore, different versions cannot normally co-exist within a single running system, complicating integration and migration. (There has been much interesting research but not much is available for mainstream development.)

A third barrier exists between the system under development and the larger ecosystem of related software. Countless research efforts in the past decade have shown that, by mining the ecosystem, much useful knowledge can be gleaned about common coding practices, bugs and bug fixes, and so on. Unfortunately this information is not readily accessible to developers, so they often turn instead to question and answer fora.

We see two main challenges, namely *"Ecosystem Mining"* and *"Evolutionary Monitoring."* By mining software ecosystems and offering platforms to analyze them [2], we hope to automatically discover intelligence relevant to a given project. Examples are opportunities for code reuse, automatically-generated and evolving documentation, and usage information than can influence maintainers of libraries and frameworks.

Evolutionary monitoring refers to steering the evolution of a software system by monitoring stakeholder needs. An example of this is *architectural monitoring* [3] which formalizes architectural constraints and monitors conformance as the application evolves. Other examples include tracking the needs of stakeholders (*i.e.* both developers and users) to determine chronic pain points and opportunities for improvements; tracking technical debt to assess priorities for reengineering and replacement; and monitoring technical trends, especially with respect to relevent technical debt.

In the long run, we envision a development environment that integrates not just the current code base and the running application, enabling easy navigation between them, but also knowledge mined from the evolution of the software under development as well as from the software ecosystem at large. The development environment should support active monitoring of the target system as well as the ecosystem to identify and assess opportunities for code improvements.

5 Conclusion

Object-oriented programming has fulfilled many of its promises. Software systems today are longer-lived and more amenable to change and extension than

ever. Nevertheless we observe that object orientation is slowly dying, with the introduction of ever more complex and heterogeneous systems.

We propose to rejuvenate object-oriented programming and let ourselves be guided by the maxim that "programming is modeling." We need programming languages, tools and environments that enable models to be directly expressed in code in such a way that they can be queried, manipulated and analyzed.

Acknowledgments. We thank Mircea Lungu for his comments on an early draft of this essay. We also gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018), and its predecessor, "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

1. Achermann, F., Nierstrasz, O.: Applications = components + scripts—a tour of piccola. In: Aksit, M. (ed.) Software Architectures and Component Technology, pp. 261–292. Kluwer, Alphen aan den Rijn (2001)
2. Caracciolo, A., Chiş, A., Spasojević, B., Lungu, M.: Pangea: a workbench for statically analyzing multi-language software corpora. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 71–76. IEEE, September 2014
3. Caracciolo, A., Lungu, M., Nierstrasz, O.: A unified approach to architecture conformance checking. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 41–50. ACM Pres, May 2015
4. Chiş, A., Denker, M., Gîrba, T., Nierstrasz, O.: Practical domain-specific debuggers using the moldable debugger framework. Comput. Lang. Syst. Struct. **44**(Part A), 89–113 (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014)
5. Chiş, A., Gîrba, T., Nierstrasz, O., Syrel, A.: The moldable inspector. In: Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, New York (2015) (Onward! 2015, page to appear)
6. Combemale, B., Deantoni, J., Baudry, B., France, R.B., Jézéquel, J.-M., Gray, J.: Globalizing modeling languages. Computer **47**(6), 68–71 (2014)
7. Fritz, T., Murphy, G.C.: Using information fragments to answer the questions developers ask. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, ICSE 2010, pp. 175–184. ACM, New York (2010)
8. Kay, A.C.: The early history of Smalltalk. In: ACM SIGPLAN Notices, vol. 28, pp. 69–95. ACM Press, March 1993
9. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, pp. 168–177 (2009)
10. Kurš, J., Lungu, M., Nierstrasz, O.: Bounded seas. Comput. Lang. Syst. Struct. **44**(Part A), 114–140 (2015). Special issue on the 6th and 7th International Conference on SoftwareLanguage Engineering (SLE 2013 and SLE 2014)

11. Maloney, J.H., Smith, R.B.: Directness and liveness in the morphic user interface construction environment. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, UIST 1995, pp. 21–28. ACM, New York (1995)
12. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), pp. 1–10. ACM Press, New York, September 2005 (invited paper)
13. Nierstrasz, O., Kurš, J.: Parsing for agile modeling. Sci. Comput. Program. **97**(Part 1), 150–156 (2015)
14. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical dynamic grammars for dynamic languages. In: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, pp. 1–4, June 2010
15. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 380–404. Springer, Heidelberg (2010)
16. Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. **34**, 434–451 (2008)