2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

Current Challenges in Practical Object-Oriented Software Design

Maurício Aniche
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

Joseph W. Yoder
The Refactory, Inc.
Urbana, USA
joe@refactory.com

Fabio Kon
University of São Paulo
São Paulo, Brazil
kon@ime.usp.br

***Abstract*—According to the extensive 50-year-old body of knowledge in object-oriented programming and design, good software designs are, among other characteristics, lowly coupled, highly cohesive, extensible, comprehensible, and not fragile. However, with the increased complexity and heterogeneity of contemporary software, this might not be enough.**

This paper discusses the practical challenges of object-oriented design in modern software development. We focus on three main challenges: (1) how technologies, frameworks, and architectures pressure developers to make design decisions that they would not take in an ideal scenario, (2) the complexity of current real-world problems require developers to devise not only a single, but several models for the same problem that live and interact together, and (3) how existing quality assessment techniques for object-oriented design should go beyond high-level metrics.

Finally, we propose an agenda for future research that should be tackled by both scientists and practitioners soon. This paper is a call for arms for more reality-oriented research on the object-oriented software design field.

***Index Terms*—software design, class design, object-oriented design, domain modeling, software engineering, software architecture, object-oriented programming.**

I. Introduction

The term "object" in programming dates back from the early 1960s, appearing in several MIT projects at that time, such as Sketchpad [34]. The first programming language to introduce the concept as a core element was Simula, which introduced concepts such as classes, objects, inheritance, polymorphism, and dynamic binding in the late 1960s [11].

Since then, we have seen the birth of several languages who followed OOP ideas in one way or another, such as Smalltalk, C++, Java, C#, JavaScript, Ruby, and Scala. 50 years later, OOP languages became prevalent in the software development field, as 7 out of the 10 most popular programming languages can be considered an object-oriented language (TIOBE index, September 2018, https://www.tiobe.com/tiobe-index).

One might see an object as something that has identity, state, and behavior [29]. In other words, objects can be distinguished from one another based on its unique identity. Also every object holds a set of mutable variables that, together, represent its current state. Finally objects communicate and trigger different behaviors, one to another by sending messages or functions.

The OOP movement made it clear for software engineers that modeling is an integral part of software development. From the beginning, modeling meant that the actions and interactions of the objects that the program created (implemented in an object-oriented language) represent the actions and interactions of the corresponding real-world physical (or virtual) objects [4]. The goal is to model the real-world scenario through objects and interactions to represent the domain as best as possible. Clearly, more complex real-world scenarios are more challenging to be modeled into a set of classes.

Parnas introduced the concept of information hiding in modular programming in his 1972 seminal paper "On the Criteria to Be Used in Decomposing Systems into Modules" [31]; a concept related to what later was referred to as high cohesion and loose coupling. In his "The Mythical Man-Month: Essays on Software Engineering" from 1975, Brooks brought the attention to the importance of conceptual integrity in software design with the idea that there should be one mind (or a small, cohesive team) that designs the architecture of the system in a consistent and well-thought way [7].

Meyer [27], in his canonical "Object-Oriented Software Construction" book, discussed several techniques to make reusable, extensible, and maintainable object-oriented design. The Gang of Four [18] proposed a large set of design patterns that help developers when looking for elegant solutions to recurrent (creational, structural, and behavioral) and reusable object-oriented design systems. Rumbaugh, Booch, and Jacobson [32] proposed the Unified Modeling Language (UML), which the goal was to provide a general-purpose language to aid software developers and business stakeholders throughout the modeling process. Martin collected the five most crucial object-oriented design principles, according to his experience, in his renowned paper "Design Principles and Design Patterns" [24], which later were known as the SOLID principles.

Freeman and Pryce [17] presented their views on how test code and, more specifically, mock objects, can help developers in understanding and better defining their class contracts, and how they should collaborate. Evans [13] suggested that developers should work together with domain experts putting all their emphasis on modeling the system core domain and the associated domain problems, and proposed several patterns to help developers on such activity which he called Domain Driven Design (DDD).

Due to the joint effort and complementary work of industry

978-1-7281-1758-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-NIER.2019.00037

and research, our community believes it has a clear vision about what an object-oriented software design that is maintainable, evolvable, and comprehensible should look like.

Or so we think. Although our body of knowledge on software design is already extensive and significant, modern software brings modern challenges. In the following sections, we describe three timely challenges we see in modern software design. We summarize the challenges below:

1) *The strong influence of libraries and frameworks on the design decisions.* Developers never start software from scratch. Instead, they reuse several libraries, frameworks, and out-of-the-box architectures not only to speed up their development, but also to support their different functional and non-functional requirements. These libraries, frameworks, and architectures often force developers to change the way they implement their models in object-oriented languages.
2) *The plethora of stakeholders and their different representations of the same problem.* Software becomes more and more complex as our businesses become more and more complex. The same software has multiple stakeholders whom all have their perspectives on the business flow. Software supports multiple final users, which also have their specific requirements. In practice, this means that a single model representation of the real-world problem is not enough anymore.
3) *The need for contextual quality measurements.* Measuring the quality of a class design is fundamental in supporting developers to decide where to improve. Currently, developers often do not trust on existing automated design analysis tools as they generate too many false positives. We argue that a plausible cause for this problem is that our approaches currently do not take the context of the system into account.

II. The impact of software architecture and technology stacks on software design

Context: Today, no software development team starts developing software from scratch. Different libraries and frameworks are chosen for different parts of the system. A software development team might choose to follow the MVC pattern for the back-end server, use Hibernate as a framework for database persistence, and AngularJS for the front-end development. Additionally, programming languages provide many libraries to be used during development. However, the underlying software architecture and technology stacks a software development team chooses as the basis for their implementation *plays a fundamental role* in the way developers design and implement their object-oriented models. In other words, these technology decisions will influence developers to follow a series of design decisions. In fact, in this paper, we argue that they often exert such a strong influence that developers currently cannot model without keeping them in mind.

A clear example of how technology impacts design decision relies on the history of Java Enterprise Edition (JavaEE) itself. Together with the EJB technology (Enterprise Java Beans), Sun proposed the Core Java EE catalogue [1], a set of patterns that developers would have to follow to make better use of its technology. Thus, to better use EJB, developers often had to mix their object-oriented models with patterns like the "Transfer Object" pattern (an object that groups many objects) or the "Business Object" pattern (to separate business logic from data), even though they may not have wanted to.

We see the influence of the technology stack in most modern web and mobile software systems. In mobile development, the Android architecture imposes several restrictions to developers, *e.g.*, to implement *Activities* for each of the user interfaces (screens) of the app, or to create *Listeners* for background tasks. This means that developers should find a way to plug their models into the constraints that are imposed by the architecture. Modern web systems are often designed in a stateless manner. This means that developers, whenever implementing their models in web systems should consider the fact that objects will have a very short life cycle, and that retrieving and storing them will be a common task.

Even deciding how to persist data might pressure developers in taking certain design decisions that otherwise would not be needed. As a concrete example, the simple usage of an Object-Relation Mapping (ORM) framework (a framework that takes care of database access by mapping objects to tables such as Hibernate) forces developers to re-think about modeling bi-directional relationships in entities (e.g., an *Invoice* class has a list of *Items*, and each *Item* class has a pointer to the *Invoice* they belong).

Finally, software systems are now highly heterogeneous and possibly distributed over several microservices. This means that the domain could be implemented by means of several different programming languages and technologies, deployed in different places, and changes in part of the domain that is implemented in language (and/or) service A needs to be propagated to the part of the domain that is implemented in language/service B. Although using the microservices architectural style with continuous delivery and DevOps [20] can help with coupling and other related issues, tracing and evolving the model that is spread across different technologies and services can be challenging.

Current solutions: To address the challenges above, practitioners have been extensively proposing developers to separate as much as possible the implementation of the domain itself from requirements of the application (e.g., databases, Android APIs). Several patterns and approaches follow this idea, such as Ports and Adapters (Hexagonal Architecture) architectural pattern [10], layered architectures and bounded contexts [13], and interface discovery [17].

Vision for the future: We need software design implementation theories that acknowledge the fact that modern software is often heterogeneous, distributed, and composed by a large set of different frameworks and libraries. Each common architecture needs to be carefully studied, and their positive and negative impact on design explored. Our extensive background on modularization and separation of concerns should serve as a basis.

III. Multiple models in large complex domains

Context: Complex real-world domains often involve complex business flow operations related to several stakeholders. Different stakeholders might see the same business process differently from another, and both are probably correct. As an example, an *Invoice* might represent something simple for the Sales department, but may play a significant role for the Payment department. An *Address* might be a simple detail for the Payment department, but might play a major role in the Delivery department.

Time also plays a vital role in complex models. Often, we observe businesses being "event-driven", meaning that the next state of the system is based on events that happen asynchronously. A large payment system should wait for a credit card company to confirm the payment before proceeding to the next steps. The shipping of a product only starts after the inventory system processes it properly and allows the system to continue with the delivery.

These different "views" on the same problem forces developers to develop several different representations of the model. In practice, this means that developers should not only model the main entities and their actions through different perspectives, but they should also model business events and how these events change the current state of the model.

Current solutions: Evans' Domain-Driven Design approach [13] proposes a set of strategic design patterns that help developers in dividing large models into different "Bounded Contexts", and to build a "Context Map" that explicitly shows the relationships between the different contexts. We also observe the rise of Event-Driven Architectures [16], [5], where developers explicitly deal with domain events.

Researchers are also aware that modeling real-world domains is a fundamental activity in requirements engineering [30] and that modeling large complex systems are, indeed, an open challenge [8]. Empiricists have been studying how developers perform requirements engineering in practice (e.g., [14]), the advantages and disadvantages of using modeling languages such as UML (e.g., [15], [33]), and how different modeling tools (*e.g.*, [12]) and techniques (*e.g.*, [25]) perform.

More recently, developers have been proposing microservices as a possible alternative to reduce the complexity and the coupling between their systems and specific technologies. In essence, we see the idea of modularization [31] being discussed by developers from different perspectives.

Vision for the future: We lack a clear understanding of how complex real-world business processes are and should be modeled. Moreover, we lack an understanding of how multiple models interact, evolve, and are maintained together, not only from an abstract level, but also from the implementation point of view.

How (and how much) can developers reuse the implementation from one model to the other? How much can one model (or should) be coupled with another model without causing any harm? How to interpret the notion of cohesion when the same aspect is now represented over multiple classes?[1] Theories explaining such questions are a fundamental step towards building guidelines and approaches that can help developers in coping with such complexity. We see it as a call for collaboration between software engineers and requirements engineering researchers.

IV. Contextually measuring the quality of object-oriented software design

Context: In a large model, it is fundamentally important to be able to detect pieces that are (not) well designed or implemented. However, an important (and hard) question is: what constitutes a "good model"? So far, our community has been relying on proxies, such as coupling, cohesion, and complexity [9], *i.e.*, classes that are highly coupled or not cohesive are normally considered poorly designed OO classes. Code smells are also a common way to point to bad implementations, *e.g.*, a God Class is a poorly designed class. The simplicity of Code Smells make them both easily understood by developers and automatically detectable by tools.

While multiple studies have shown the negative impact of such code smells in software systems (*e.g.*, [6], [28]) and developers have been using quality measurement tools (*e.g.*, Sonar), our current metrics fail to capture the context (architectural-wise or domain-wise) of that software system. There are no single truths in software design. A class might have a high coupling, and still be considered a well-designed class. As a concrete example, developers already expect their Controller classes in an MVC system to be more coupled than the rest; after all, Controllers are the bridge between the user interface and the model.

Understanding in which context a measurement makes sense and, more importantly, in which context a measurement does not make sense is fundamental in our quest for high-quality object-oriented software design. In this paper, we conjecture that an important cause for the large number of false positives that static analysis and code quality measurement tools currently generate [21] is their lack of context.

Current solutions: Taking into consideration the domain and the architecture of the system under study has been gaining attention from the community in the last years. Although linters are widely used [35], [36], and quality monitoring strategies such as Continuous Inspection have been proposed [26], researchers have shown that the domain of the application matters when it comes to the presence of code smells [22], that code metric distributions are statistically different among the different architectural roles of classes in a system [3], [23], and that specific architectures may have their own specific smells [2], [19].

Besides, recent research on Technical Debt has shown that awareness of technical debt influences team behavior. De-

[1]Not to mention that object-orientation might not be the best tool to model certain types of systems and processes. For example, functions might be a better fit in some cases. The recent rise of functional programming and serverless architectures demonstrates the power of such paradigms.

veloping better ways of identifying, monitoring, categorizing, measuring, prioritizing, and paying off the technical debt can significantly improve software development practices [37].

Vision for the future: We need empirically derived theories on what are the characteristics of complex models that should be considered of high quality and in which context, as well as numeric ways to measure such characteristics. Machine learning and data science may play an essential role in the field, given the fact that many design characteristics might be determined only by complex combinations of different quality attributes. We conjecture that such theories and approaches will support, once and for all, the development of quality assessment tools that produce less false positives.

V. Conclusions

Software becomes more and more complex as the real world gets more and more complex. To cope with it, software engineers require better modeling techniques. In this paper, we highlighted several challenges faced by contemporary developers when modeling object-oriented systems.

We hope this paper serves as a reminder to the software engineering community about the importance of good object-oriented software design in the daily life of a software engineer. Although a vast amount of knowledge on the topic has already been produced, there is still a long road ahead. It is fundamental that the real-life experience of practitioners influences the work of researchers in the field. The software engineering research community must still bring context into the picture, providing more comprehensive theories and better tools for developers to model and assess their models.

References

[1] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., 2003.

[2] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, pages 1–37, 9 2017.

[3] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa. SATT: Tailoring code metric thresholds for different software architectures. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, 2016.

[4] A. P. Black. Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, 231:3–20, 2013.

[5] A. Brandolini. Event storming. http://eventstorming.com/, 2018.

[6] L. C. Briand, J. Daly, V. Porter, and J. Wüst. Predicting fault-prone classes with design measures in object-oriented systems. In *Software Reliability Engineering. Proceedings. The Ninth Intl. Symposium on*. IEEE, 1998.

[7] F. P. Brooks, Jr. *The Mythical Man-month*. Addison-Wesley, Boston, MA, USA, 1975.

[8] B. H. Cheng and J. M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6), 1994.

[10] A. Cockburn. Ports and adapters architecture. http://alistair.cockburn.us/Hexagonal+architecture, 2009.

[11] O.-J. Dahl. The birth of object orientation: the simula languages. In *From Object-Orientation to Formal Methods*, pages 15–25. Springer, 2004.

[12] J. M. C. de Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno. Requirements engineering tools. *IEEE software*, 28(4):86–91, 2011.

[13] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[14] D. M. Fernández and S. Wagner. Naming the pain in requirements engineering: A design for a global family of surveys and first results from germany. *Information and Software Technology*, 57:616–643, 2015.

[15] A. M. Fernández-Sáez, M. Genero, D. Caivano, and M. R. Chaudron. Does the level of detail of uml diagrams affect the maintainability of source code?: a family of experiments. *Empirical Software Engineering*, 21(1):212–259, 2016.

[16] M. Fowler. Martin fowler bliki. https://martinfowler.com/bliki, 2003. Entries: AnemicDomainModel, DomainEvent.

[17] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[18] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[19] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, May 2015.

[20] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[22] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM, 2014.

[23] B. d. S. Marcos Dosea, Claudio Sant'Anna. How do design decisions influence the distribution of software metrics? In *Proceedings of the 2018 International Conference on Program Comprehension*, 2018.

[24] R. C. Martin. Design Principles and Design Patterns. Technical report, Object Mentor, 2000.

[25] J. Mendling, H. A. Reijers, and J. Recker. Activity labeling in process modeling: Empirical insights and recommendations. *Information Systems*, 35(4):467–482, 2010.

[26] P. Merson, A. Aguiar, E. Guerra, and J. Yoder. Continuous inspection: a pattern for keeping your code healthy and aligned to the architecture. In *3rd Asian Conf. on Pattern Languages of Programs*, 2014.

[27] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[28] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[29] J. Noble. The myths of object-orientation. In *European Conference on Object-Oriented Programming*, pages 619–629. Springer, 2009.

[30] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[31] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[32] J. Rumbaugh, G. Booch, and I. Jacobson. *The unified modeling language reference manual*. Addison Wesley, 1999.

[33] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, G. Tortora, M. Risi, and G. Dodero. Do software models based on the uml aid in source-code comprehensibility? aggregating evidence from 12 controlled experiments. *Empirical Software Engineering*, pages 1–39, 2018.

[34] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 329–346. ACM, 1963.

[35] K. F. Tómasdóttir, M. Aniche, and A. van Deursen. Why and how javascript developers use linters. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, 2017.

[36] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, 2018.

[37] G. S. Tonin. *Technical Debt Management in the Context of Agile Methods in Software Development*. PhD thesis, University of São Paulo, 2018.