# Sustainable Software Design

Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
martin@cs.mcgill.ca

## ABSTRACT

Although design plays a central role in software development, the information produced in this activity is often left to progressively *evaporate* as the result of software evolution, loss of artifacts, or the fading of related knowledge held by the development team. This paper introduces the concept of *sustainability* for software design, and calls for its integration into the existing catalog of design quality attributes. Applied to software design, *sustainability* conveys the idea that a particular set of design decisions and their rationale can be succinctly reflected in the host technology and/or described in documentation in a way that is checkable for conformance with the code and generally resistant to evaporation. The paper discusses the relation between sustainability and existing research areas in software engineering, and highlights future research challenges related to sustainable software design.

## CCS Concepts

•**Software and its engineering → Designing software;**

## Keywords

Software Design; Software Evolution

## 1. DESIGN EVAPORATES

The critical importance of *design* in software engineering has been recognized for over four decades [14]. The output of this multi-faceted activity [23, 38] is "a description of the structure of the software to be implemented..." [33, p. 38] But what is this description, where does it live, and how does it relate to the system?

When required, justified, and/or mandated, design will result in an explicit design artifact, such as an SDD (Software Design Description [16]). In other contexts, the design will be realized into the final system, and auxiliary information will remain in the form of traces in email discussions, entries in notebooks, pictures of white boards, and human memories.

*FSE 2016 Visions and Reflections Paper – November 2016*
© 2016 ACM. ISBN .
DOI: 10.1145/2950290.2983983

In both cases, the practical discontinuity between the creation of a design and its realization introduces a progressive *evaporation* of the design knowledge for a system among its developers. When explicit design documents are produced, loss of information comes from design *drift* and *erosion* (as a general case of the concept of architectural drift and erosion [36, 39]). When no explicit design document exists, design information becomes lost or untraceable as the fragments of digital traces are discarded or become increasingly out of date and difficult to retrieve [2, 9, 40]. The pernicious consequences of design evaporation include time wasted in avoidable program understanding effort, and *ignorant surgery* [27].

The loss of design knowledge is a well-known problem that has received copious attention from researchers, as is evidenced by the decades-long research agenda on design recovery [5, 10, 21], including its incarnations in the form of design model reverse engineering [18] and design pattern detection [25, 37]. Although design recovery techniques can offer relief to software developers trying to make sense of a large system, they cannot rediscover information, such as design rationale, that is not reflected in the source code. They also raise the question of what to do with the recovered design. Will it, too, evaporate?

## 2. SUSTAINABLE SOFTWARE DESIGN

This vision paper is a call for developing the concept of *sustainability* for software design, and integrating it into the existing catalog of design quality attributes [3, 24]. Applied to software design, *sustainability* conveys the idea that a particular set of design decisions and their rationale can be succinctly reflected in the host technology and/or described in documentation in a way that is checkable for conformance with the code and generally resistant to evaporation. The concept of sustainable software design can be positioned within the *technical* dimension of a general perspective on software and sustainability [4].

As a design quality attribute, sustainability will naturally be in a trade-off position with other quality attributes, such as extensibility. For example, a software system can be made more extensible through the use of additional parameterization, but when this parameterization introduces new concepts and terminology, requires documentation, and obscures the original intent of a modular structure, the additional extensibility will come at the cost of sustainability.

I provide an illustration of different points on the sustainability spectrum by drawing two examples from the design of JetUML, an open-source UML editor written in Java [19].

The application is not very large but the principles discussed here apply to design abstractions that would be expected to scale in proportion to the size of the overall system.

The design of JetUML involves two class hierarchies that implement, respectively, a `Node` interface and an `Edge` interface. In the `Node` hierarchy, subclasses contribute additional state and functionality through standard mechanisms (overriding) and the use of a well-known design pattern (Template Method [15]). In the `Edge` hierarchy, in order to leverage some of the features of the JavaBeans framework, the behavior extensions provided by some edge subclasses must take into account an unusual naming convention, whose violation will result in major faults. Also, as opposed to the `Node` hierarchy, in the `Edge` hierarchy the presence of intermediate subclasses has repercussions that are visible to the user. Neither of these aspects can be properly captured by the programming language, and must therefore be documented externally [20]. In this case, one could argue that, without the external documentation, the design of the `Edge` hierarchy would be less sustainable than that of the `Node` hierarchy. The documentation helps to bridge the gap, but introduces the need to be validated for conformance, and in some cases updated, when the code changes.

Following this short discussion, one may wonder if sustainability may not be another name for *simplicity*. Could it be that in this example, the `Node` hierarchy is simply "simpler", and the `Edge` hierarchy "more complex"? The problem with the notion of complexity in software is that it lacks a credible definition. In contrast, sustainability puts the emphasis on more concrete sub-attributes that include self-descriptiveness and checkability. Although the `Node` hierarchy comprises 18 types and about 250 members, if I were to walk away from the project for a year and come back to it, I would be able to grasp its design without documentation or external help because the design elements and rules are embedded in language constructs, they are related by well-known patterns, and they map to recognizable concepts in the solution domain. In contrast, without explanatory artifacts, it would be very difficult to rediscover some of the design ideas underlying the `Edge` hierarchy, and that despite the fact that this hierarchy is actually smaller in terms of total number of classes and members (about 150).

## 3. PREVENTING DESIGN EVAPORATION

Different areas of software engineering research and practice explore problems related to design evaporation and contribute solutions that help prevent it. How are they related to the concept of sustainable design?

*Modularity.* *Modularity* and its tandem concept of *dependency* are prevalent themes in software design [31]. Modularity, in its simplest expression, concerns the decomposition of a system into parts, which immediately begs the question of how. Parnas' foundational proposal is to hide information with the goal of designing for change [26]. This dogma has however been challenged, in particular in favor of broadening the goal of design to that of adding value to a system [34]. Modularity can contribute to the sustainability of a design when it makes the overall structures and intent of the design apparent and easy to respect. However, that is yet another goal whose value must be assessed in relation to all other competing goals for a design. The extensive work on multi-dimensional separation of concerns [30, 35] and empirical evidence that design rules are routinely violated [8, 41] illustrate the natural tensions between conflicting design goals and the effect of development pressure. In this context the question is not whether sustainability aligns with modularity but, rather, how modularity can support sustainability, and what we should do when it cannot. Future research will help us determine what makes a design sustainable. Although modularity is bound to play a role, it will likely not be an exclusive role.

*Documentation.* The relation between documentation and sustainable design is paradoxical. On the one hand, good documentation helps sustain a design over time [28]. On the other, the mere prospect of having to create documentation can be seen as a challenge to sustainability [42], and reliance on documentation past a given threshold can be an indication of an unsustainable design. In the JetUML example presented above, the documentation supports design in the same way a crutch supports a frail leg: as an unfortunate necessity with an added cost. Is the idea of sustainable design, then, to have fully self-documenting systems in a way that parallels the idea of literate programming [22]? Parnas and Clements compare this ideal with that of the philosopher's stone (i.e., an impossibility) [28]. Although self-documenting design may be partially possible in certain contexts (such as that of an Application Programming Interface [7]), the relation between documentation and sustainable design is one of essence and accident of software engineering: Sustainable design is supported by essential documentation, whereas unsustainable design is supported by excess documentation addressing accidental limitations of the design. But how can we tell the difference?

*Programming Languages.* Design decisions are naturally self-documenting when the structures and rules they define align with features of the programming language. In the JetUML example, the design requires `ClassNode` and `InterfaceNode` instances to be interchangeable, and this design decision is directly reflected in the fact that both classes share a common supertype. Expressing design decisions in terms of language constructs is a promising way to achieve sustainability. The problem is that the set of language features is bounded, whereas the set of design decisions is not. Moreover, general-purpose programming languages support the realization of implementation-level concerns, and consequently support the expression of ideas at a low level of abstraction. Programming languages can be extended to better support important concepts tied to design rules and constraints, such as immutability [6, 12]. At the other end of the abstraction spectrum, language constructs can be created to support high-level reasoning about system structure, such as components and connectors [1]. The challenge with programming language constructs that express design ideas is to move beyond simple descriptiveness ("this is a component") towards checkability for conformance ("this component is not connected to this other one"). Despite steady advances in program analysis, conformance checking of design models is by no means a solved problem [17]. In this context, the concept of design sustainability may provide additional insights into the value of linguistic or library elements that support the expression of general design rules. As a middle ground between basic module definitions and comprehensive architectural models, the annotations used by many infrastructure frameworks capture useful, if partial, design information. For example, object-relational mapping frameworks include operators that describe data models. In a similar way, dependency injection frameworks require developers to use annotations such as `@Inject` and `@Component`, thereby providing limited but checkable information about component boundaries and integration points.

*Design Patterns.* Design patterns [15] are another notable means by which important design decisions can be reflected in a system and carried forward. The nomenclature introduced by design pattern catalogs effectively creates a map between parts of a system identified by recognized names (e.g., "Observer") and both a system of design rules and their corresponding rationale. The integration patterns mandated by application frameworks also qualify as design patterns in the context of sustainability, because they achieve the similar effect of capturing design decisions by relating application-specific extensions to the overall system in structured ways. Although an important building block of sustainability, patterns are not a panacea. First, patterns are, by definition, solutions to *common* design problems, which means the more idiosyncratic parts of a software system's design may not map to any design pattern. Second, patterns are not yet fully checkable for conformance. Although tool support can be created to detect [25, 37] or instantiate [13] patterns in a system, in the general case the conformance checking of design patterns is only a special case of the difficult conformance checking problem for design models in general. Finally, most design patterns address a rich design space with many points of variability. To solve a concrete design problem with the help of a pattern, a software developer will need to make some design decisions to complete the pattern. How these decisions can be captured will determine the sustainability of the design pattern's instantiation.

*Model-Driven Engineering.* One of the stated goals of model-driven engineering (MDE) is to address the "semantic gap between the design intent [...] and the expression of this intent..." [32, p.26]. To further this aim, MDE technologies involve the use of domain-specific modeling languages that describe systems at a level of abstraction that is better suited for capturing design intent than the implementation constructs of general-purpose programming languages. At the same time, models are themselves the product of design activities, so the question of sustainability really becomes transferred to the model artifacts: How can they capture the important decisions that led to their inception?

*Software Process and Development Culture.* Although this paper focused on issues of design representation, it is worth mentioning that the software development process and the development culture surrounding the use of programming languages are also bound to influence design sustainability. A software process determines how much design information is produced and the form it takes, two factors that are inevitably tied to design sustainability. In the case of programming languages, different user communities have different cultural practices, such as naming conventions, use of meta-programming, and typical module size, that are likely to impact sustainability.

## 4. RESEARCH CHALLENGES

Many challenges lay ahead on the path to realize a vision of sustainable software design. To be useful, the concept will need to be defined precisely enough to support the unambiguous qualification of degrees of sustainability in different technical and organizational contexts. This determination is bound to be influenced by divergent opinions and experiences about the longevity of software design, and the challenge will be to find a definition that transcends individual projects. Given the multi-faceted nature of design sustainability, supporting this quality attribute will require experimentation with a wide range of ideas including new programming language constructs, emergent documentation [29], and advances in traceability [11]. Fostering sustainable software design will also require understanding how the creation and use of design information needs to be coordinated across development activities. The ultimate challenge will be to find reliable ways to evaluate the cost and benefits of design sustainability, so as to build a body of evidence that can be used to demonstrate the up-to-now intangible benefits of sustainable design, and hopefully empower organizations to incentivize software designers to invest in the future.

## 5. ACKNOWLEDGMENTS

I am grateful to B. Dagenais, G. Mussbacher, C. Treude, the members of the McGill SE Seminar, and the reviewers for insightful comments. This work is funded by NSERC.

## 6. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proc. 24th ACM/IEEE International Conference on Software Engineering*, pages 187–197, 2002.

[2] S. Baltes and S. Diehl. Sketches and diagrams in practice. In *Proc. 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 530–541, 2014.

[3] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[4] C. Becker, S. Betz, R. Chitchyan, L. Duboc, S. M. Easterbrook, B. Penzenstadler, N. Seyff, and C. C. Venters. Requirements: The key to sustainability. *IEEE Software*, 33(1):56–65, 2016.

[5] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[6] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *Proc. 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 35–49, 2004.

[7] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507, 2006.

[8] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proc. FSE/SDP Workshop on the Future of Software Engineering Research*, pages 47–52, 2010.

[9] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*, pages 557–566, 2007.

[10] E. J. Chikofsky, J. H. Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[11] J. Cleland-Huang, O. C. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman. Software traceability: trends and future directions. In *Proc. Future of Software Engineering*, pages 55–69, 2014.

[12] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring language support for immutability. In *Proc. 38th ACM/IEEE International Conference on Software Engineering*, 2016.

[13] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *Proc. 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 75–88, 2006.

[14] P. Freeman. The central role of design in software engineering. In A. I. Wasserman and P. Freeman, editors, *Software Engineering Education*, pages 116–199. Springer, 1976.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[16] IEEE Computer Society. *IEEE Standard for Information Technology—Systems Design—Software Design Descriptions*, 2009. IEEE Std 1016-2009.

[17] D. Jackson and M. Rinard. Software analysis: A roadmap. In *Proc. 22nd ACM/IEEE International Conference on Software Engineering*, pages 133–145, 2000.

[18] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering*, 27(2):156–169, 2001.

[19] JetUML: A Free simple UML diagramming tool. http://cs.mcgill.ca/~martin/jetuml/, 2016.

[20] JetUML design documentation–Edge hierarchy. https://github.com/prmr/JetUML/blob/v0.12/doc/functional/EdgeHierarchy.md, 2016.

[21] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Proc. 21st ACM/IEEE International Conference on Software Engineering*, pages 226–235, 1999.

[22] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[23] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering*, 41(2):135–156, 2015.

[24] R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance*, pages 701–704, 2005.

[25] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. 24th ACM/IEEE International Conference on Software Engineering*, pages 338–348, 2002.

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

[27] D. L. Parnas. Software aging. In *Proc. 16th ACM/IEEE International Conference on Software Engineering*, pages 279–287, 1994.

[28] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986.

[29] M. P. Robillard and N. Medvidović. Disseminating architectural knowledge on open-source projects. In *Proc. 38th ACM/IEEE International Conference on Software Engineering*, pages 392–403, 2016.

[30] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.

[31] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 167–176, 2005.

[32] D. C. Schmidt. Guest editor's introduction: Model-Driven engineering. *Computer*, 39(2):25–31, 2006.

[33] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

[34] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–108, 2001.

[35] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proc. 21st International Conference on Software Engineering*, pages 107–119, 1999.

[36] R. N. Taylor, N. Medvidović, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2010.

[37] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[38] A. van der Hoek and M. Petre, editors. *Software Designers in Action: A Human-Centric Look at Design Work*. CRC Press, 2014.

[39] J. van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[40] J. Walny, J. Haber, M. Dörk, J. Sillito, and S. Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8, 2011.

[41] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd ACM/IEEE International Conference on Software Engineering*, pages 411–420, 2011.

[42] U. Zdun, R. Capilla, H. Tran, and O. Zimmermann. Sustainable architectural design decisions. *IEEE Software*, 60(6):46 – 53, 2013.