Documenting and Communicating Design Decisions

Steven Fraser, Innoxec, Santa Clara, CA, USA, sdfraser@acm.org
Dennis Mancl, MSWX Software Experts, Bridgewater, NJ, USA, dmancl@acm.org

ABSTRACT

Design is both a human and a technical endeavor. Documenting and communicating design decisions remains challenging. Lightweight graphical models, text, and design-focused unit tests help. However, whether trying to understand the software output of an AI-aided design tool or reusing software components developed by others – documentation, collaboration, and communication remain key to understanding and propagating design decisions.

Keywords—Software design, agile, collaboration, education

I. Introduction

As reported at the 2024 Designing Workshop in Lisbon [1], we previously explored how investing in learning activities such as internal workshops and design forums could promote a company-wide increase in team collaboration in the design process. In this paper, we focus on some of the communication activities that individual teams might adopt to improve their low-level design process.

The activity of software design has evolved over the past 75 years. Initially software was developed with an ad hoc design process focused on personal face-to-face communication complemented by text and graphical notations such as decision tables and flowcharts. Documents, notations, and models helped development teams design, develop, and deliver software products. In today's iterative and agile development environments, design collaboration in our opinion has become more focused on individual effort. Developers are more siloed in their coding efforts. Tool interactions have become more common than personal interaction.

While developers value creating high-level design to support collaboration and communication, low-level design collaboration is apparently less valued. To us, this seems consistent with less investment in education and in documenting and reviewing individual system components. Modern software education in our opinion tends to focus on understanding the "big picture" of a complex system combined with coding principles. Design models are used at a high level to describe and reason about tradeoffs in the behavior of large software modules and computer-human interface interactions – but they may not be seen as useful in guiding low-level design decisions.

II. Design process

Software design is a technical, collaborative, and social process that "encompass all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately

modifying complex systems – not just the activity following requirements specifications and before programming [2]." However, we believe that practitioner attitudes and software development culture are less focused on systematic design tasks. Today's design culture may be less focused on system-wide design discussions, perhaps due to the influence of Beck's [3] arguments against "Big Design Up Front" (BDUF) and the proliferation of microservices and mobile applications for smart device platforms.

Simple design diagrams can be helpful in design conversations. Although diagrams might be useful for communicating detailed design, we've observed that 21st century developers may prefer to start coding without creating detailed design pictures. Some developers feel that in a culture focused on code, design tasks such as diagramming and discussion are a wasted effort. Also, developers may avoid collaborative design because today's development practices emphasize individual effort.

Future developers benefit from information about the problem domain in their detailed design documentation. When domain-specific design explanations are omitted, future developers will be constrained (and/or confused).

Personal attitudes determine how design practices are executed, especially design reviews. We have evidence that developers are less interested in design "ceremonies" today. The importance of design and code reviews in software development has declined. These review practices were common in the development of telecom systems in the 1990s, but are less popular with the adoption of agile practices [4, 5, 6]. Reviews are still deemed relevant in the development of regulated systems where reliability and safety are important – for example aviation, medical, nuclear power, or defense systems.

III. Evolution from sequential to agile processes

Design became an identifiable element of development processes by the late 1960s as systems increased in size and complexity. Ad hoc design practices became insufficient to manage complexity. Companies adopted design processes to organize software requirements, algorithms, data flows, and interfaces. Software processes became monolithic sequential delivery processes, and design was a phase in the software development pipeline between requirements and development (coding). Design documents were an intrinsic part of multi-party negotiation between developers, product managers, senior executives, sales, marketing, customers – and sometimes end-users. Once the design was complete, design documents guided developers throughout the implementation (and maintenance) of the system.

Design processes evolved in two directions in the 1990s:

- more tools and more formalism (UML),
- more iteration and less formalism (Agile).

Design models and diagrams were favored by organizations that required a more formal approach to building a design. State machines, class diagrams, and interface specifications are powerful ways to express complex behavior. In some problem domains, models could support formal specifications and proofs of correctness. But they are not for everyone. In agile

development, design became a more continuous code-centric process throughout the software lifecycle.

It is our belief that software architecture documents and design models continue to have a role in defining and communicating a system's high-level abstractions. Diagrams can guide and help synchronize the design work of multiple teams. But in iterative development, most development tasks are "opportunistic" – tasks are chosen to satisfy an emergent high-value customer need. Tasks are small and quick: they may trigger only a minimum set of software extensions to the existing project code base. In this case, developers are unlikely to prioritize design documentation. Agile developers [Agilists] tend to think and document in code. With all design documentation embedded in the code, individuals maintaining or extending code will have a limited source of trustworthy design knowledge.

IV. Code-centric values

Both Programming Heroes and Agilists live by the creed: "we don't need documentation."

Steve McConnell explained the culture of the Programming Hero. "Programming heroes take on challenging assignments and write mountains of code…. They are essential to their projects. Success, it seems, rests squarely on their shoulders [7]." It seems that every large complex project should have some Programming Heroes.

McConnell also noted the downside of the Programming Heroes. "The reality is that for every programming hero who is capable of monumental coding achievements, there are other pathological programming disasters who just don't know how to work well with others. They hoard design information. They refuse to participate in technical reviews. They refuse to follow standards established by the team."

"The sum total of their actions is to prevent other team members from making potentially valuable contributions. A significant number of programming heroes don't turn out to be heroes at all; they turn out to be prima donna programming ball hogs."

Agile developers assert that the source code is the design. The notion that code was design "was widespread in the Smalltalk community in the 1980s…. Few Smalltalk developers bothered with the detailed design documentation seen in traditional software engineering projects [8]."

While developers may aspire to write code consistent with a self-documenting coding style, they often only keep functions and classes simple or choose descriptive names for variables and functions. A motivation for writing self-documenting code (as opposed to code containing inline-text comments) is that documentation remains in synch with the code because it "is" the code.

- Heroes think "I have said it all in my code. If other developers can't understand my design from reading the code, they should be replaced with smarter developers."
- Agilists think, "If design documentation – particularly of detailed internal scenarios – doesn't deliver any value to the customer, it's a waste of time to produce."

In both cases above, the lack of communication becomes a significant source of technical debt, which can slow development and future maintenance.

As code is maintained and/or extended, there is a risk that any changes will have subtle defects, especially if developers misinterpret the intent of the original code. The authors believe that investment in some design documentation is essential to sustain development velocity.

In 1985, David Parnas discussed the value of revising the documentation of a chaotic design, to make it appear that it was built using a "rational process." A perfect linear process is impossible, so Parnas referred to this clean-up work as "faking it." Organized documentation is very useful. Parnas was thinking of a rational top-down process being the engineering ideal: every part of the design would be traceable to the original problem statement and customer requirements [9].

> *We will never find a process that allows us to design software in a perfectly rational way. The good news is that we can fake it. We can present our system to others as if we had been rational designers and it pays to pretend do so during development and maintenance.*

While today's agile and iterative development is different from the context of Parnas's original insight, there still is value in faking it. If too much work is required to build a complete detailed design model for each iteration, developers should consider investing in ways to partially fake it. Robust design models, documentation, and tests can improve communication and support collaborative approaches to software design.

V. Thinking about more than code

When the only persistent design artifact is code, team members may struggle to engage in constructive dialog regarding design tradeoffs. It will also be more challenging for maintainers or future developers to decipher the design if they are unfamiliar with the code.

For more than 20 years, the "technical" side (advocating the creation of detailed code without documentation and BDUF) has been winning over the "soft" side of design, i.e., encouraging collaboration and communication through diagrams, models, and tests.

In our opinion, there are at least three paths to improving the design process. Processes and design artifacts can be tailored to support:

1. micro processes
2. design-focused unit tests
3. mini-architecture documentation

A. Micro Process

Grady Booch advocated for iterative and incremental software processes [10], and explained that every software project inherently has a *macro process* and a *micro process*. Booch contrasts the longer cycle of architecture work versus the rapid iteration of code details: "The macro process[es]… should comprise the successive refinements of the system's architecture," and "The micro process[es]… should comprise the overlapping waves of discovery, invention, and

implementation." Micro processes build up a plan for the interfaces and relationships between the classes/packages, and they define the details of packages or classes that implement the low-level abstractions. Booch sees each process as cyclical, and there are many micro-process cycles within a single turn of the macro process. Figure 1 shows one possible way to organize the two levels of iteration.

It is not necessary for every software development team to use the same micro process. In an agile world, teams try different approaches to build the artifacts they need – both for delivery to the customer and for the team's own future internal use. A micro process that relies solely on code (and code comments) for low-level design documentation is not ideal. The team might try a combination of simple diagrams, notes from group discussions, and interface specs.

Booch explains that discussion and documentation are critical to communication across a project. Booch asserts that the micro process must focus on more than just code for the current release. Team members should also explore risks and prepare the groundwork for future iterations. Team members should develop habits and rituals that support sharing information.

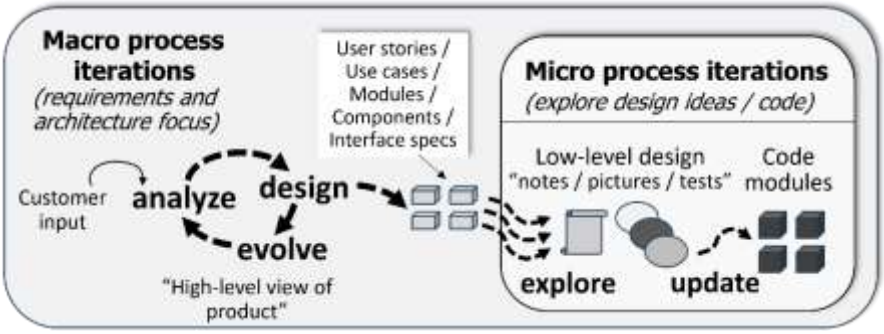

Fig. 1. Macro process and micro process iterations

Agile process frameworks such as Scrum, Extreme Programming, and Kanban don't prescribe specific detailed processes for exploratory work and design information sharing. But these tasks are necessary to support a successful long-lived project. One useful agile habit is to invest in some lightweight documentation.

How does one package rely on another? It is not always obvious from reading the code. Small development teams, even if they call themselves "agile", need to be aware of the value of a well-run micro process. Agile processes often include a "Definition of Done" for development tasks. Such a definition of done might require developers to include simple written explanations of how the responsibilities, interfaces, and relationships have been changed in the current iteration, either in documents or in code comments. Team-specific low-level documentation rules can define simple traceability practices, such as how developers should connect diagrams, notes, and interface specs to user stories.

B. Testing

Unit testing is an important part of agile development. Our experience is that developers are challenged to write effective tests. Some unit tests are built simply to check that a specific algorithm has been correctly coded.

Unit tests should also be viewed as an opportunity to add unambiguous information about detailed design decisions. We believe that instead of rewarding developers for writing many superficial code-level tests, more focus needs to be assigned to tests of the system's special design mechanisms.

Each test is an opportunity for designers to illustrate how great their code is, and to reduce the likelihood that their design isn't retired (or refactored) from the code base. Note that *design-focused unit tests* are different from "test-first development" and "acceptance test driven development". Design-focused tests are built as part of the low-level design and coding activities, and they may be linked to design decisions made during the coding process. Acceptance tests are based on software requirements, but they don't necessarily help in the validation or documentation of low-level design decisions.

C. Architecture

Parts of the low-level design can be represented by a "mini-architecture".

During the coding process in an agile iteration, much of the code can be routine and unremarkable. No elaborate documentation is required. Future developers tasked with extending or maintaining the system can understand algorithms and inputs/outputs by walking through the code either manually or with the assistance of a tool.

On the other hand, some code that is written within an agile iteration may implement architectural requirements. For example, some code implements security checks or performance enhancements. Some low-level designs will support operations for failure recovery or dealing with system overload. Logging and monitoring classes and objects may be added to a system to report information about anomalies to log files and to monitor system stability. Each of these design elements will become critical infrastructure that must be maintained and extended over many development iterations.

Teams may develop *architecture-focused design documents* because one or more "architecture stories" were raised to the top of the priority list during the iteration planning process. But in other cases, a user story could spur some questions about missing architecture features that customers didn't include explicitly in the requirements. Developers might build the needed functionality, but these architecture-related designs might not be obvious from reading the code.

Architecture stories are often undervalued in agile planning. The failure to implement adequate architectural infrastructure is responsible for the increased technical debt of agile projects [11]. Architectural infrastructure should be documented systematically if the system is to be maintained and extended efficiently.

Teams use agile development to make small code enhancements without considering the overall architecture. For example, adding new features to an existing design might cause the system to fail to deliver on some of its key reliability, performance, and security obligations. This is something that can happen in either agile or traditional development, but agile development's feature-at-a-time approach can make the design more fragile [12].

Lightweight architecture documentation is a valuable way to start. It's an idea that comes from George Fairbanks's work on Architecture Haiku [13]. His focus is to use architecture documentation to zoom in on the "risks" affecting a software product. Since low-level design failure can pose significant risks, those risks can be mitigated with targeted design documents.

A one-page document can list the key characteristics of a design (tradeoffs, quality attribute priorities, design rationale, constraints, and architectural style). One simple diagram is usually enough.

- The tradeoff section explains which code characteristics are key. For example, resilience might be more important than speed of execution.
- Quality attributes prioritize which system characteristics are critical or optional. For example, security may be more important than latency which is more important than throughput.
- Constraints explain what the system should not do. For example, message handlers should not write to local memory, to enable efficient distributed implementation [14].

"Design documentation format" is less critical than keeping the explanation simple. The advice from Fairbanks: "Produce an artifact that is easily consumed by readers. The end result is almost like a flyer advertising the best parts of the architecture [15]." Many designs follow well-known software design patterns – and developers will add code comments and small design descriptions that label each element of the pattern within the code. Fairbanks reports that design patterns are useful abstractions for developers. He notes that we use patterns that define and document key parts of the software architecture because they are a set of abstractions that helps us "reason about software."

A simpler text-based approach for writing architecture documentation may have team members recording specific architecture issues as tailored text documents: Architecture Decision Records (ADRs) [16]. An ADR is a short paragraph or two that records three facts about a single design decision: its context, the decision, and its consequences.

To support each of these three processes for adding more low-level design documentation (*micro process*, *design-focused unit tests*, and *architecture-focused design*), every developer needs to believe in the idea that "design" is more than just code.

VI. Maintaining the value of design documentation

Grady Booch offered a good list of design characteristics that need explanations that go beyond what can be expressed in even the most carefully structured software code [17].

Booch observed that "the code is the truth, but not the whole truth." *Architecture* (layered modules, security, and other system-wide design features) is one focus where code is insufficient to explain how the software functions. Design documentation can include *patterns*, *cross-cutting behavior*, and *design rationale*. Design documentation can record the tribal memory for the accumulated experiences of a small software development team.

As product release deadlines approach, it becomes increasingly difficult to find time to write or revise software design documentation. Developers increasingly focus on overcoming obstacles

that would delay release. Design documentation, inspections, reviews, and testing are de-prioritized. Decisions go undocumented and the code becomes the design. Last-minute code updates contribute to increased "technical debt" and increased design entropy.

In a good *micro process*, the design documentation should focus on rapidly evolving code modules. The format of the design documentation will depend on team design styles: with text sections or diagrams for data access rules, internal module states, critical scenarios, and transactions. *Design-focused unit tests* can explain states and scenarios. *Mini-architecture documents* can list the transactions and transformations, as well as the constraints for security and performance.

VII. Technologies for design understanding

We believe that two approaches have emerged over the past 30 years that contributed significantly to an increased level of design understanding when building or extending software systems: the reuse of existing designs (such as design patterns, libraries, and frameworks) and AI tools.

One way to reduce design and code complexity is to avoid it completely – by using software built by others. In his "No Silver Bullet" paper [18], Fred Brooks identifies software component reuse as one of the promising approaches for improving software development productivity. Reusing mature and well-tested software components (e.g., applications, frameworks, patterns, and open source) reduces the need for the creation of more design documentation for large chunks of the system's functionality.

While AI/ML based design tools may accelerate the design process, these design documentation challenges remain:

1. How will design decisions be documented in the face of product extensions or software maintenance?
2. How will "collaborative refactoring" be supported and will human developers be able to refactor AI-generated code?
3. What will the basis be for "trust" – particularly for users of software systems that support regulated systems and how will design decisions be documented in the face of product extensions or software maintenance?

VIII. Design investment

Software design seems to be failing because of two trends:

- **In a competitive marketplace, delivery speed can be more important than resilience and quality.** Developers are focused on delivery because the product planning and payback cycles are much shorter today than 30 years ago. Design must be done quickly to meet management and market deadlines, so detailed design work is often just a code-debug cycle.
- **Design processes seem less collaborative.** "A team of one" might code rapidly. But a more innovative path is to have a diverse team that includes dimensions of gender, geography, methodologies, domains, and corporate functions. Such a team will be better

positioned to offer design trade-offs, and the team may have more insight into how technology can delight and drive customer demand.

Exploration of the design space can lead to innovative and creative solutions. A diverse process can resist unconscious bias in design. In a SPLASH 2022 panel session on diversity in software design, Kelly Blincoe, a panelist from University of Auckland, shared her views: “When we don’t have diverse perspectives on software teams, we can accidentally create systems that exclude, frustrate, or endanger other people [19].”

When a developer modifies or extends undocumented code, there are few clues to the context and constraints. A new emphasis “beyond code” in the software micro process could increase the volume of simple low-level design models, design-focused automated tests, and mini-architecture documents. The goal would be to reduce the confusion and isolation of software developers.

We believe that there are three ways to augment low-level design documentation:

- require design artifacts as part of the Definition of Done,
- improve the scenario coverage and readability of unit tests, and
- make architectural properties more visible by maintaining lightweight architecture stories.

Agile and iterative development processes generally require less up-front design effort than sequential development. But an agile development organization should engage in effective collaboration and communication, and that generally requires that iterations deliver more than just code and unit tests.

In the age of AI, industry could benefit from research to innovate the educational paths of future software professionals. Developers should have foundational skills and knowledge breadth. To meet future development challenges, researchers can help identify and prioritize educational learning strategies for non-coding knowledge, e.g., the scientific method, ethics, communication, and program management. Another area for design research will likely include qualitative and quantitative studies to explore human-AI partnership design decisions that go beyond code and system structure. Effective software product delivery requires design understanding and delivery of more than just code. Markets and management are delighted with products that meet or exceed expectations. They are less interested in apparent bureaucratic processes which doesn’t demonstrate value or may delay delivery and increase costs [20].

References

[1] Steven Fraser and Dennis Mancl, “Investing in software design,” in Proc. 1st International Workshop on Designing Software (Designing ’24), pp. 34–39. https://doi.org/10.1145/3643660.3643944

[2] Peter Freeman and David Hart, “A science of design for software-intensive systems,” Comm. ACM, vol. 47, no. 8, Aug. 2004, pp. 19-21.

[3] Kent Beck, Extreme Programming Explained, Addison-Wesley, 1999.

[4] Martin Fowler, “Is design dead?” An essay posted on Martin Fowler’s website based on Fowler’s XP 2000 keynote talk, May 2004. [Online]. Available: https://www.martinfowler.com/articles/designDead.html

[5] S. D. Fraser and C. S. Saunders, “Enhanced reuse with group decision support systems,” in Proc. Advances in Software Reuse, March 1993, Lucca, Italy. IEEE, 168-175. https://doi.org/10.1109/ASR.1993.291706

[6] Steven D. Fraser and Dennis Mancl, “XP 25th anniversary workshop & panel report: Innovating software solutions – past, present, and future,” Proc. XP 2024 Conference. https://doi.org/10.1007/978-3-031-72781-8_30

[7] Steve McConnell, Professional Software Development, Addison-Wesley Longman Publishing, 2003.

[8] Pete McBreen, Questioning Extreme Programming, Addison-Wesley, 2003.

[9] David L. Parnas and Paul C. Clements, “A rational design process: how and why to fake it,” IEEE Transactions on Software Engineering, vol. SE-12, no. 2, pp. 251-257, Feb. 1986.

[10] Grady Booch, Object Solutions: Managing the Object-Oriented Project, Addison-Wesley Professional, 1995.

[11] Philippe Kruchten, “The (missing) value of software architecture,” Dec. 2013 blog article. [Online]. Available: https://philippe.kruchten.com/ 2013/12/11/the-missing-value-of-software-architecture/

[12] Dennis Mancl, “Architecture in an agile world,” Notes from several workshops on how architecture and agile work together, 2019. [Online]. Available: https://manclswx.com/talks/arch_agile_notes.html

[13] Michael Keeling, Design It!: From Programmer to Software Architect, Pragmatic Bookshelf, 2017.

[14] George Fairbanks, “Software architecture is a set of abstractions,” IEEE Software, vol. 40, no. 4, pp. 110-113, July-Aug. 2023.

[15] George Fairbanks, “Architecture haiku presentation,” Sept. 2010. [Online]. Available: https://www.georgefairbanks.com/assets/pdf/arch-haiku-2010-09-07.pdf

[16] Michael Keeling, “Love unrequited: the story of architecture, agile, and how architecture decision records brought them together,” IEEE Software, vol. 39, no. 4, pp. 90-93, July-Aug. 2022.

[17] Grady Booch, “Why don’t developers draw diagrams?” Keynote talk at SOFTVIS 2010, Oct. 25-26, 2010. [Online]. Available: https://softvis.wordpress.com/wp-content/uploads/2010/12/gradybooch-whydontdevelopersdrawdiagrams-softvis2010.pdf

[18] Frederick P. Brooks, “No silver bullet – essence and accident in software engineering,” UNC technical report (1986). [Online]. Available: http://www.cs.unc.edu/techreports/86-020.pdf, also IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

[19] Steven Fraser and Dennis Mancl, Dimensions of Diversity, Equity, and Inclusion, SIGSOFT Softw. Eng. Notes 48, 2 (April 2023), pp. 18–21. https://doi.org/10.1145/3587062.3587068

[20] Ryan Mac, Kate Conger, Jack Ewing, and Eric Lipton, “Slash first, fix later: how Elon Musk cuts costs,” New York Times, Nov. 19, 2024. Available: https://www.nytimes.com/2024/11/16/technology/elon-musk-cost-cuts.html

---

This report was presented at the Designing 2025 Workshop at ICSE 2025, April 27-28, 2025.
https://doi.ieeecomputersociety.org/10.1109/Designing66910.2025.00006