Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmerman**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

Software Process versus Design Quality

Tug of War?

Girish Suryanarayana, Tushar Sharma, and Ganesh Samarthyam

"Something smells rotten in the state of our design." This realization might come despite all good intentions behind choosing and following the "right" process. Don't underestimate a process's inadvertent effects on the resulting software design's quality, as evidenced in two insightful stories from Girish Suryanarayana, Tushar Sharma, and Ganesh Samarthyam that are based on their recently published book on design smells. Enjoy! —*Cesare Pautasso and Olaf Zimmermann, department editors*

SOFTWARE LIFE-CYCLE PROCESSES provide a structured, disciplined way to guide the development of complex real-world software.[1,2] These processes can be primary (acquisition, supply, development, operation, and maintenance), supporting (for example, documentation, configuration management, quality assurance, reviews, audits, or problem resolution), or organizational (management, infrastructure, improvement, and training).[3] Our interests lie in software design, so we pose this question: Do software life-cycle processes (henceforth simply called software processes) benefit software design?

The answer is a clear yes! For instance, a software process that recommends periodic architecture and design reviews to ensure the design quality, and supports traceability between the requirements and design elements to ensure the design's completeness, helps ensure high-quality design.[4,5]

However, our experience with design smells in real-world projects and interviews with software engineers from various organizations[6] have revealed a paradox. Sometimes, a design exhibits smells because a software process (or combination of processes) has inadvertently become a significant hindrance to high design quality, thus negating the benefits the process was meant to deliver. In some cases, a process has actually undermined design quality. In these cases, an approach that aims to address the design smells and improve the design quality can't merely rely on tactical refactoring of the design artifacts. It also must refactor the process, remove it, or introduce another process.

0740-7459/15/$31.00 © 2015 IEEE

THREE SOFTWARE DESIGN SMELLS

Here we look in more detail at the three design smells mentioned in the main article. For more on them and other design smells, see *Refactoring for Software Design Smells*.[1]

DUPLICATE ABSTRACTION

This design smell has two forms. *Identical name* is when two or more abstractions have identical names. *Identical implementation* is when two or more abstractions have semantically identical member definitions, but the design hasn't captured and used those implementations' common elements.

INSUFFICIENT MODULARIZATION

This smell arises when an abstraction hasn't been completely decomposed and a further decomposition could reduce its size, implementation complexity, or both. This smell has two forms. *Bloated interface* is when an abstraction has many members in its public interface. *Bloated implementation* is when an abstraction has many methods in its implementation or has one or more methods with excessive implementation complexity.

MULTIPATH HIERARCHY

This smell arises when a subtype inherits both directly and indirectly from a supertype, causing unnecessary inheritance paths in the hierarchy. This complicates the hierarchy and increases developers' cognitive load, thus reducing the hierarchy's understandability. Furthermore, developers might overlook existing implementations on the redundant paths and try to provide their own implementation for the realized interface. In this process, they could provide a considerably different implementation (or no implementation). Such mistakes can lead to run-time problems. So, this smell can impact reliability.

Reference

1. G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, Morgan Kaufmann, 2014.

A key insight from these cases is that software processes and design quality are inextricably intertwined. To highlight this interplay, we examine the following two real-world cases.

A Suboptimal Change Approval Process

In a globally distributed software development project, a central team (Team A) owned the code base. Team A included domain experts who had originally designed the software. The development was offshored to another team (Team B).

When an external consultant analyzed the code base after the release of the software's first version, he found smells in the design and code. For instance, several classes suffered from the identical-implementation form of the Duplicate Abstraction smell (see the sidebar). Surprisingly, Team B was aware of the extensive code duplication but had made no effort to refactor the design.

The consultant discovered that the lack of refactoring was due to the process the project followed. To prevent unwarranted modifications that could negatively impact the product's functionality, the project relied on a stringent process to control source code changes. Team A had to review any code change made by Team B before the change could be approved. This review focused on functional correctness (from a domain perspective), not the code's structural quality. This change approval process was long and arduous and required multiple emails and telephone interactions between the teams.

To reduce the time to ratify changes that slowed the development rate, Team B wanted to avoid this process as much as possible. Because refactoring (including refactoring to eliminate code clones) would involve only structural changes to the code without impacting the functionality, it seemed logical to avoid refactoring to avoid the change approval process. So, Team B wasn't keen to refactor the source code. In this case, the environment viscosity[7] created by the process led to the software's poor design quality. The consultant shared this finding in his final report and suggested refactoring the change approval process.

The project management could have refactored the change approval process in several ways. For example, they could have incorporated review of code quality into it. Instead, they refactored it to have Team A ratify only new class additions and not every small change that Team B made. The project management be-

lieved this would eliminate the viscosity (by reducing the number of times the change approval process was initiated), which in turn would improve the design quality.

Upon the release of the software's second version, the consultant again analyzed the design quality. Surprisingly, he found many more instances of the Insufficient Modularization smell (see the sidebar), compared to the first release. For example, one class in the source code had approximately 40,000 LOC, and the weighted methods per class (the sum of the cyclomatic complexities[8] of a class's methods) exceeded 2,000.

The consultant found that part of the root cause of the many instances of the Insufficient Modularization smell was still the change approval process. Toward the software's second release, the project management at the off-shore location became concerned about the many bugs (in the order of hundreds) found during testing, which posed a risk for timely delivery. To ensure that the software was released on time, the project management assigned each developer in Team B a target of fixing four bugs every week.

Fearing that they would be considered underperforming if they couldn't fix four bugs per week, the members of Team B explored ways to avoid the change approval process for new classes that bug fixing might require. One easy way to avoid introducing classes was to insert new code in existing classes. Because of schedule pressures, many developers adopted this workaround, which resulted in many Insufficient Modularization instances.

Because this workaround was convenient and the large classes produced no immediate runtime effects, it evolved into a bad habit during development for the third release. It also explained why Team B didn't refactor these large classes immediately after the release. Refactoring them would have required introducing smaller classes, which would have required going through the change approval process.

The project management could have addressed this problem in various ways. For instance, it could have refactored existing processes (for example, refactor the change approval process to remove bottlenecks and improve the turnaround time for change review). Or, it could have removed them (for example, remove the bug-target-setting process so that developers don't bypass change approval). Alternatively, or in combination with the ways we just mentioned, the project management could have introduced a process (for example, introduce a local change approval process to speed up approval).

In this case, for the product's third release, the project management removed the bug targets. They also adopted a *design quality gate* process that required each developer to run a set of design analyzers on the portion of code he or she had modified, before checking-in the code. This helped address the problem significantly, and the number of smells drastically decreased during the third release.

An Ineffective Design Communication Process

Industrial software systems often create complex domain objects to fulfill complicated business requirements. The initialization of such objects typically involves a sequence of steps, including preinitialization and postinitialization. These two steps are crucial, and software developers must remember to write an implementation for them. To ensure this, a common practice is to create an interface that encapsulates them and require developers to realize this interface in the classes corresponding to the domain objects.

> Software design quality is a function of the effectiveness of the followed process in a given context.

In this context, we share an anecdote in which one of us helped develop an application for creating visually attractive user interfaces, using the concept of gadgets. Figure 1a shows a fragment of the application design wherein a `TextGadget` class extends its parent class `GadgetBase` and realizes an `ISupportInitialize` interface. This interface contains two methods, `preInitialize()` and `postInitialize()`, that must be defined by `TextGadget`.

Over time, the need arose to support multiple gadgets such as `GraphicGadget` and `NumericGadget`. The team realized that the implementation for `preInitialize()` and `postInitialize()` remained similar across gadgets. So, the team decided that instead of each gadget separately realizing `ISupportInitialize`, `GadgetBase` could itself

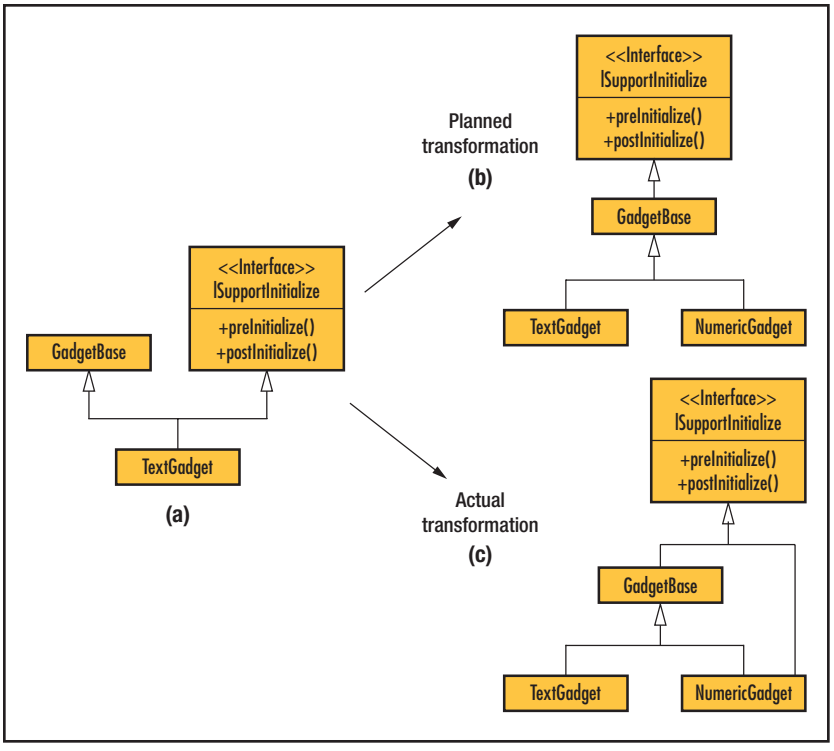

FIGURE 1. Planned versus actual transformation in the **TextGadget** class hierarchy. (a) A fragment of the original application design wherein **TextGadget** extends its parent class **GadgetBase** and realizes an **ISupportInitialize** interface. (b) Instead of each gadget separately deriving from **ISupportInitialize**, **GadgetBase** realizes **ISupportInitialize** and provides the default implementation in the planned refactoring. (c) However, in the realized design, **NumericGadget** extends **GadgetBase** and unnecessarily realizes **ISupportInitialize**. This design fragment suffers from the Multipath Hierarchy design smell.

realize **ISupportInitialize** and provide the default implementation of **preInitialize()** and **postInitialize()** (see Figure 1b).

However, the developer entrusted with implementing **NumericGadget** was on leave when the rest of the team discussed this new design. By the time he returned, the old design had been refactored and the new design was in place. Unfortunately, no one told him about this design decision.

This situation occurred because the project followed an agile methodology and subscribed to the principle that individuals and interactions are more important than detailed documentation.[9] Specifically, the team discussed and communicated design decisions during stand-up meetings. It explicitly documented only the architectural design decisions[10] (such as introducing a new layer or changing the middleware being used) in the architecture specification. In this case, the team considered that having **GadgetBase** directly implement **ISupportInitialize** wasn't an architectural decision, so the team didn't explicitly document this design change.

Because the developer was unaware of the noncritical design decisions, he implemented **NumericGadget** using the old paradigm. That is, **NumericGadget** extended **GadgetBase** and realized **ISupportInitialize** (see Figure 1c). The resulting design thus suffered from the Multipath Hierarchy design smell (see the sidebar).

In short, this design smell arose because the process used to communicate design decisions and changes to all the team members wasn't effective. A more potent process aligned with the agile methodology would have employed multiple communication modes to convey design decisions. For example, it would have additionally used emails or lightweight knowledge management systems such as wikis to document all design decisions so that they were always available to the entire team.[9]

These two cases highlight the process–quality paradox: software processes are designed to bring discipline to software development and intend to help achieve and maintain high-quality software design. However, some software processes (because of how they're implemented or the project conditions) turn out to be cumbersome or porous, leading to situations that can decrease design quality. Software design quality is a function of the effectiveness of the followed process in a given context. So, such situations require us to introduce, tune, or refactor existing processes to achieve and maintain high design quality. In conclusion, these cases lead to the following insights.

First, all the relevant stakeholders need to recognize the interplay between software processes and design quality.

Second, development teams must periodically evaluate design quality (for instance, by looking for design smells). If the quality is poor, teams must determine whether any software process is the cause.

Finally, if the cause of poor design quality is process-related, teams can address it by refactoring or removing an existing process or introducing a new one, as we mentioned before. Refactoring a process might include identifying and removing the obstacles that directly or indirectly hamper good quality.

References

1. N.S. Potter and M.E. Sakry, *Making Process Improvement Work: A Concise Action Guide for Software Managers and Practitioners*, Addison-Wesley, 2002.
2. D. Damian et al., "An Industrial Case Study of Immediate Benefits of Requirements Engineering Process Improvement at the Australian Center for Unisys Software," *Empirical Software Eng.*, vol. 9, nos. 1–2, 2004, pp. 45–75.
3. *IEEE/EIA Std. 12207-2008—ISO/IEC/IEEE Standard for Systems and Software Engineering—Software Life Cycle Processes*, IEEE, 2008.
4. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, vol. 38, nos. 2–3, 1999, pp. 258–287.
5. G. Samarthyam et al., "MIDAS: A Design Quality Assessment Method for Industrial Software," *Proc. 2013 Int'l Conf. Software Eng.* (ICSE 13), 2013, pp. 911–920.
6. G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, Morgan Kaufmann, 2014.
7. R.C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Addison-Wesley, 2003.
8. T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, 1976, pp. 308–320.
9. M. Paasivaara, S. Durasiewicz, and C. Lassenius, "Using Scrum in Distributed Agile Development: A Multiple Case Study," *Proc. 4th IEEE Int'l Conf. Global Software Eng.* (ICGSE 09), 2009, pp. 195–204.
10. O. Zimmermann, "Architectural Decisions as Reusable Design Assets," *IEEE Software*, vol. 28, no. 1, 2011, pp. 64–69.

GIRISH SURYANARAYANA is a senior research scientist at the Corporate Research and Technologies Center, Siemens Technology and Services Private Ltd., India. He's a member of the *IEEE Software* advisory board. Contact him at girish.suryanarayana@siemens.com.

TUSHAR SHARMA is a technical expert at the Corporate Research and Technologies Center, Siemens Technology and Services Private Ltd., India. Contact him at tusharsharma@ieee.org.

GANESH SAMARTHYAM is an independent consultant and a corporate trainer. Contact him at ganesh.samarthyam@gmail.com.