Editor in Chief: **Diomidis Spinellis**
Athens University of Economics and Business, dds@computer.org

Architecture from a Developer's Perspective

Diomidis Spinellis

I CAN STILL remember when, back in 2003, a fellow FreeBSD developer chastised me for an architectural misstep. I had proposed adding a reference to a related C library function in the documentation of a Unix system call. "I believe this is bad practice (a layering violation)," he wrote to me. He was right; I hadn't thought carefully about that small addition. The truth is that as a developer you practice architecture daily, but only rarely do you have time to reflect on your corresponding decisions, actions, and their consequences. Software architecture affects the quality of what you build and how you build it.

The Importance of Software Architecture ...

The most obvious way that architecture affects quality is maintainability. Code that lacks clear boundaries and interfaces is difficult to analyze. It's also brittle and, therefore, difficult to change. A small addition or fix in one place can cause a cascade of additional required work, or worse, bugs. Also, you can't easily test and debug such software because it will lack obvious interfaces where you can apply test probes or add logging functionality.

> Architecture is difficult to learn and practice.

Then comes performance. On a small scale, code jumbled together can be famously efficient: think of tightly coded routines, graphics kernels, and some game engines. However, once the scale increases, the only hope to cope with rising demand comes from parallelism architectures. These let you split your work horizontally (along tasks) or vertically (across multiple clients). They also guide you on how to shard or partition your data. Through such architectures, you increase both your current service capacity and future scalability. Similar approaches can increase your service's reliability. First, you can manage fault tolerance by distributing the work among nodes that can step in to cover each other in the event of a failure. Second, the same nodes can then help the more complex task of recovery. Don't even think about orchestrating recoverability into your service without an architecture to guide the delicate required dance.

Another quality aspect that software architecture aids is portability—a must in an age of rapid innovation and shifting technology alliances. Through clear layering,

IEEE Software Mission Statement

To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

0740-7459/15/$31.00 © 2015 IEEE

SER&IP 15 BEST PAPER AWARD

The 2nd Annual Software Engineering Research & Industrial Practice 2015 (SER&IP 15) workshop, held in conjunction with the International Conference on Software Engineering 2015 (ICSE 15), focused on the sometimes problematic interface between the academic and practitioner communities.

Given *IEEE Software*'s mandate to present the practical and impactful work that can help bridge this gap, the magazine was pleased to sponsor the Best Paper Award recognizing some of the outstanding work presented at the workshop. *IEEE Software* editor in chief emeritus Forrest Shull served on the selection committee and helped select the best paper on the basis of the criteria of readability, rigor, and relevance.

The selection committee chose "Principles and a Process for Successful Industry Cooperation— the Case of TUM and Munich Re," by Maximilian Junker, Manfred Broy, Benedikt Hauptmann, Wolfgang Boehm, Henning Femmer, Sebastian Eder, Elmar Juergens, Rainer Janßen, and Rudolf Vaas, for the Best Paper Award on the basis of the soundness of the lessons learned and the balanced treatment of both positive and negative aspects of tech transfer. *IEEE Software* editor in chief Diomidis Spinnellis presented the award on behalf of the magazine. (For more on this paper, see Practitioner's Digest on p. 27.)

Please join us in congratulating the authors for their excellent work and thank all of the authors for taking the time to share their experiences in such a forum.

your software can quickly adapt to new hardware platforms and software interfaces. Proper encapsulation can also make your software easier to install and coexist with other offerings.

Software architecture also affects your main development processes—the way you can split the teams that develop the software, how you can run it across countries and time zones, and how you can maintain it without disruptions. It also helps your ancillary processes. A suitable architecture goes hand-in-hand with effective configuration management tasks, such as versioning, branching, merging, and continuous integration. Software architecture can provide clear boundaries to manage quality efficiently. For instance, it can allow you to tailor quality and processes characteristics for diverse software modules. Regarding testing processes, modern testing frameworks are typically embodiments for corresponding architectural styles.

Finally, software architecture is the key enabler for reusability—processes that span many of your organization's products and services. It can help you create modules that can be reused within your organization, and it can drive software product lines.

… And What to Do

Given software architecture's importance, what should you be doing as a developer? This is a tough question, because architecture is difficult to learn and practice (it's been

described as an old man's art), and its mistakes can be hugely expensive.

My advice is to focus relentlessly on the primary concerns. Smaller ones are important, but the big ones determine success or failure. Look at your software's most common, large, and critical functions; study your software's future evolution path, looking for things that are difficult to change; and determine the key quality attributes. These elements will point toward the important things that your architecture needs to address. Once you have them, invest significant effort in developing a matching architecture. Consider other successful examples, ask around, look for already available modules, prototype, and experiment. Be ready to toss out a solution if something better emerges. Remember, this is what can make or break your software.

I recently withdrew a paper I had submitted and started working almost from scratch on a two-year effort when a much more experienced colleague suggested a drastic improvement in a design. Not all software deserves such sacrifices, but you should be ready to make them when you see the potential.

Then, avoid the temptation to overengineer. The worst architectural sins have been committed by developers keen to demonstrate their (often half-baked) knowledge of design patterns and other elements often misused as architectural crutches. Frameworks, design patterns, and enterprise-scale platforms are all useful, but applying them to the wrong area creates more problems than it solves. The cognitive load of a needlessly complex software architecture is higher than that of a slightly simplistic one. Therefore, avoid designing structures when there isn't a clear demand for them, and choose the simplest solution that can do the work.

This brings me to another principle: be ready to refactor when the need emerges. Your lean and mean software architecture will be pressured as the system evolves, accumulating technical debt. In contrast to an overengineered system, the pressure will quickly become apparent and the pressure points will reveal where refactoring is truly required. Consider yourself lucky at that point: in contrast to green-field development, you have a very clear requirement of where to invest your architectural and refactoring effort. Do it without stinginess or looking back. An interesting example is the evolution of the pipes and filters architecture under Unix. When pipes were introduced to Unix, Bell Labs researchers worked tirelessly to convert all their existing programs into filters that could be connected through them. The rest is history.

> Focus relentlessly on the primary concerns.

Keep in mind that architecture is about more than software code. Consider how your whole system (in the widest possible sense) will be decomposed into processes or services; how data are stored, communicated, and processed; and how all parts fit together to deliver the required functionality, reliability, capacity, scalability, maintainability, and portability. Your decisions here may affect which parts you can purchase, reuse, or outsource. Earlier this year, a team I worked with faced the problem of maintaining a large set of data that would slowly evolve over time, changing through both daily automated processes and human interactions. All changes should be auditable, and it should be possible to rerun the processing starting at an arbitrary point in time.

Initially, we considered as an obvious choice a complex relational database schema encompassing time-stamped records, user authorizations, processing chain identifiers, and an event log. We also considered using file-system directories to implement part of this functionality. Both approaches involved considerable amounts of application code. It then dawned on us that by using a revision control system such as Git to version the data files, we could get most of the required features "for free." As an added bonus, team members could also employ user-friendly Git interfaces to manipulate the data. This simple decision, which took us about a day of deliberation and discussions to agree on, saved us weeks of development effort and debugging.

Finally, when you develop your architecture, you should adhere to sound software design principles:

- abstraction of processes, data, and control structures;
- low coupling and high cohesion;
- separation of concerns, decomposition, modularity, encapsulation, and information hiding;
- separation of interfaces from implementation and of policy from mechanisms; and
- completeness, economy, and simplicity.

Adhering to all these tenets might sound like a tall order. But nobody ever said that architecture is cheap; it's a sound investment for your current needs and future evolution. As Brian Foote and Joseph Yoder once said, "If you think good architecture is expensive, try bad architecture."

See www.computer.org/software-multimedia for multimedia content related to this article.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.