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

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

Software Reuse in the Era of Opportunistic Design

Tommi Mikkonen and Antero Taivalsaari

From the Editors
Software reuse has succeeded beyond expectations. Our industry has recently added software of unknown provenance (SOUP) to commercial off-the-shelf software (COTS) as the options available to architectural decision makers. Tommi Mikkonen and Antero Taivalsaari reflect on the long-term implications, good and bad, of rapidly assembling working solutions out of easy-to-reuse software packages and services by sharing stories from one of their latest projects that led to an action plan for the community. —*Cesare Pautasso and Olaf Zimmermann*

OPPORTUNISTIC DESIGN, AN approach in which people develop new software systems by routinely reusing and combining components that were not designed to be used together, has become very popular. This emergent pattern places focus on large-scale reuse and developer convenience with the developers "trawling" for most suitable open source components and modules online. The availability of open source assets for almost all imaginable domains has led to software systems in which the visible application code, as written by the application developers themselves, forms only the "tip of the iceberg," compared to the reused bulk that remains mostly unknown to the developers. The actual reuse takes place in an ad hoc, mix-and-match fashion. In this article, we take a look at this increasingly popular approach in light of our industry experiences. We argue that challenges associated with such a development model are quite different from traditional software development and reuse.

A Silent Revolution in Software Development

In the past 20 years, the World Wide Web has strongly affected the way people develop software. The emergence of the software-as-a-service model,[2,20] Internet-based developer forums (e.g., Stack Overflow, https://stackoverflow.com), and open source software repositories (e.g., GitHub, https://github.com) have enabled an approach in which people routinely trawl online for ready-made solutions for all kinds of problems; the discovered libraries and code snippets are included in applications with little consideration or knowledge about their technical quality or details. This approach is all about combining unrelated, often previously unknown software and hardware artifacts by joining them with "duct tape and glue code."[6] Depending on one's viewpoint and desired connotation, such development is referred to as *opportunistic design*,[6] *opportunistic reuse*, *ad hoc reuse*, *scavenging*,[9]

Digital Object Identifier 10.1109/MS.2018.2884883
Date of publication: 16 April 2019

software mashups, *mashware*,[13] or sometimes even *frankensteining*.[6] The resulting approach bears the imprint of cargo-cult programming,[12] which is the ritual inclusion of code or program structures for reasons the programmers do not fully understand. Such programmers make no attempt to understand how those components work or how they might interfere with other parts of the system.

Developers in droves have embraced this approach, although it is widely admitted that opportunistic designs are not automatically compatible and that such designs may require significant architectural adjustments to fulfill functional or nonfunctional requirements.[19] For instance, in client-side web development, web mashups have become very popular.[1] In cloud back-end development, the use of what are called *software of unknown provenance* (*SOUP*) *components* is nowadays even more prevalent, given the large number of available open-source components and the apparent complexity in building corresponding functionality from scratch. In the latter domain, the popularity of opportunistic design has exploded because of the success of Node.js (https://nodejs.org/) and its Node Package Manager (NPM) ecosystem (https://www.npmjs.com/). Today, more than 700,000 reusable NPM modules are available for nearly all imaginable tasks.

While opportunistic reuse can be very convenient for developers, such reuse is rather ad hoc in practice compared to the systematic textbook methodologies proposed for software reuse two or three decades ago.[8,11] The resulting systems are not carefully crafted but instead resemble icebergs in that only the "tip" is written by developers themselves, while the bulk of the system comes from other sources and remains invisible and often poorly understood by the application programmer. Consequently, many of the characteristics that have traditionally been highly valued in software design and implementation—such as performance, small memory footprint, consistent interfaces, ease of maintenance, and fault tolerance—become emergent and highly dependent on the (mostly invisible) qualities of the external components. Granted, the importance of such characteristics may vary. For instance, when writing testing tools for internal use at a company, rapid progress is often valued far more highly than small memory footprint or interface consistency. However, for many types of software systems—especially those intended for regulated domains (e.g., medical software)—the use of third-party components has traditionally been discouraged or prohibited altogether because of these characteristics.

In general, it has become virtually impossible in recent years to develop any significant software systems without relying at least to some degree on available component ecosystems, such as the NPM ecosystem previously mentioned. Opportunistic reuse is common despite the risks that components developed by unknown developers, using unknown methodologies, may contain unknown and possibly harmful safety-related characteristics. Component selection is often based simply on popularity ratings or recommendations from other developers.

From our experience, we understand the software engineering challenges arising from opportunistic design and reuse. This article aims to raise awareness of how profoundly this model changes application development. We will offer a brief real-world example and provide a call for action and directions for further work.

A Motivating Example

As an example, let us use an industrial Internet of Things (IoT) development project we initiated about three years ago. In that project, we needed to construct a scalable cloud back end for an IoT system that would collect large amounts of measurement data arriving from very data-intensive measurement devices. The goals of the system were threefold. In the beginning, the system would act as a technology demonstrator to showcase the benefits of a live-streaming end-to-end data platform. Soon thereafter, the system became the foundation for a number of commercial software products. In parallel, the system was also used as a research and exploration platform for IoT-related device-development activities.

Requirements

Unlike typical IoT systems that usually collect point measurements only [i.e., relatively small amounts of data (such as heart-rate measurements, GPS coordinates, or altitude data) that are uploaded periodically], our system needed to support incoming streaming data (i.e., data that would be streamed in continuously at high data rates). Such cases are common, for instance, in the virtual-reality/augmented-reality media domain as well as in certain types of medical systems (e.g., in collecting electrocardiogram measurements) or industrial systems (e.g., manufacturing control processes).

In addition to streaming data, our system needed to provide support for real-time data analytics, i.e., be able to analyze the data in near real time as the data are streamed in; process the data; and generate responses, visualizations, and actions with minimal latency. Furthermore, an extensive set of query mechanisms had to be provided for reading previously collected data

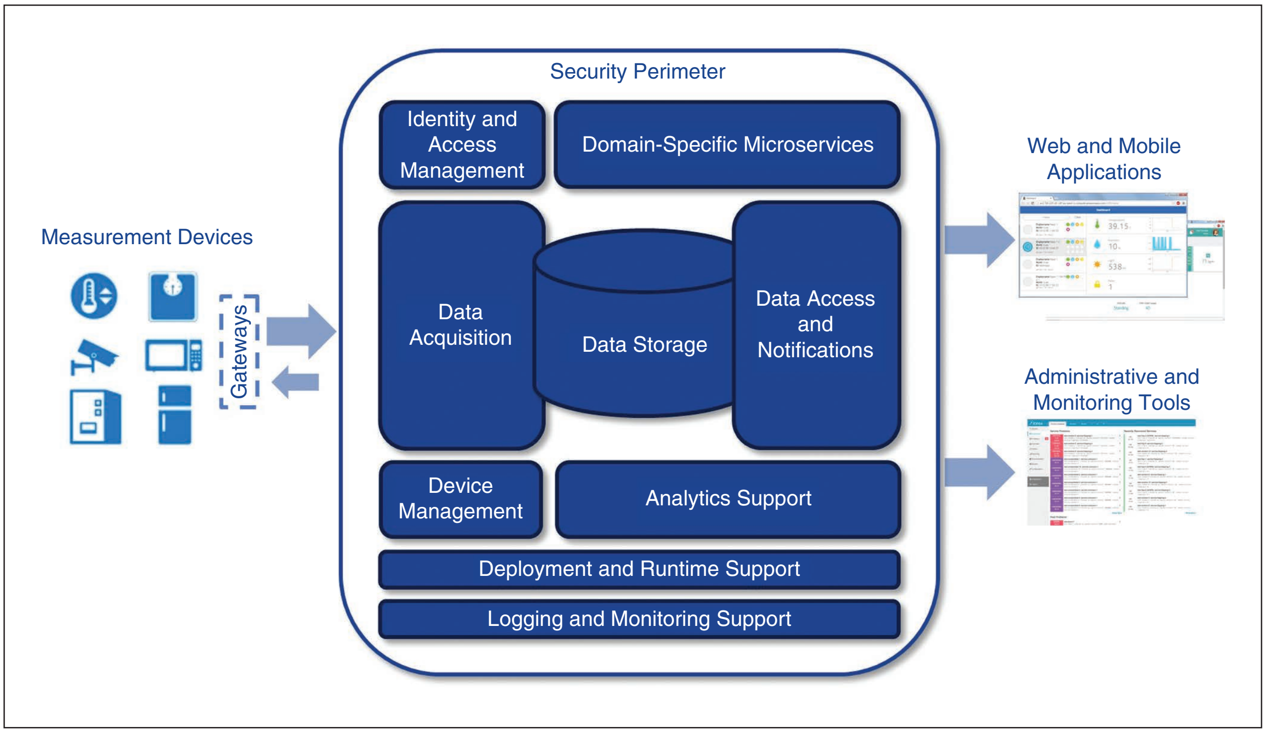

Security Perimeter
Identity and Access Management
Domain-Specific Microservices
Data Acquisition
Data Storage
Data Access and Notifications
Device Management
Analytics Support
Deployment and Runtime Support
Logging and Monitoring Support
Measurement Devices
Gateways
Web and Mobile Applications
Administrative and Monitoring Tools

FIGURE 1. A diagram showing the high-level architecture of our case-study system.

(time series) with various query parameters, e.g., from a certain sensor and within a given time range. A notification mechanism capable of generating notifications when data values met certain predefined criteria was required as well.

In addition, our system needed to provide a lot of "bread-and-butter" cloud back-end functionality, such as user-identity management (user accounts and access permissions), device management, logging and monitoring capabilities, and some administrative tools for managing the overall system. We also wanted to have a flexible, scalable cloud-deployment model that was not physically tied to any particular machines, data centers, or vendors. The deployment model had to include the ability to easily deploy multiple instances of the entire cloud environment onto different types of cloud environments, including OpenStack (https://www.openstack.org/).

Architecture

Figure 1 provides a high-level overview of the architecture of the case study. Functionally, the system can be seen as an IoT data pipeline in which the data flow from measurement devices from the left toward the web and mobile applications on the right. In between, the cloud provides the necessary data acquisition, analytics, storage, access, and notification mechanisms as well as many other supporting components.

From the beginning, we wanted to make it easy to add new functionality, components, and application programming interfaces (APIs) on top of the base system so that the system could be redeployed in different industry verticals. A microservice-based architecture[16] was selected as a generic solution for plugging in additional components without interfering with the rest of the system.

Component Selection and the Development Approach

Given the small size of our original development team, we knew that we would not be able to write the entire system from scratch. For instance, the streaming data-acquisition and real-time analytics functionality alone was so complex that building it from the ground up would have consumed all of the development resources we had for the entire project. Instead, we wanted to make sure that our development team had enough time to focus on developing the differentiating, domain-specific microservices. Thus, from the beginning, we decided to rely extensively on available third-party open source software. Luckily, we had significant experience with

IoT-related development work and implementation components from our past projects. This experience was useful in selecting basic components, since we knew how and how well certain open-source technologies worked. That said, there were dozens of other potential open source component candidates entirely unfamiliar to us.

Node.js was chosen as the implementation technology for the pluggable microservices. In recent years, Node.js has become one of the most popular back-end development technologies, and hence many readily available packages exist for different IoT- and deployment-related functions. Docker (https://www.docker.com/) and Docker Swarm were selected for virtualizing the deployment and runtime architecture. The actual physical deployment architecture [e.g., the exact number of virtual machines (VMs)] can vary based on the needs of each deployment. It is also possible to run the entire cloud environment on a single machine (even just a laptop if it has enough memory) for testing purposes. While this may sound like a curiosity, it can actually be very convenient and useful for testing new features without having to deploy components onto a farm of external computers or VMs.

Measurements

Because of the large number of components/services and the packaging of the system components into Docker images, the exact total size of the system is not easy to measure. A typical deployment of our system consists of more than 30 Docker images, deployed onto 4–6 VMs. About half of the services are written on top of Node.js. In our Node.js-based microservice implementations, the number of NPM modules (transitive closure of all of the NPM modules pulled in by each microservice) varies from a few dozen to more than 1,000 per microservice. Cumulatively, the total number of different NPM modules (excluding duplicates) used by the system exceeds 2,000. While many of those NPM modules, such as uuid, are very simple, there are also significantly more complex ones, such as core-js, shelljs, or redux. Overall, we estimate that only about 5% of the source code of the system was written by our developers, while the vast majority comes from third-party open source components.

Implications for Software Engineering

Although the potential for software reuse was high in the 1980s and early 1990s (e.g., Jones reported in 1984 that, on average, only 15% of code was unique, novel, and specific to individual applications; the remaining 85% appeared to be common and generic[7]), actual reuse rates remained very low. Those days developers preferred writing their own code and took pride in doing as much as possible from scratch. In fact, they were effectively expected or forced to do so, since third-party components were not widely available or easy to find before the advent of the web. Furthermore, before the widespread adoption of open source software development, components were rarely available for free or with license terms favoring commercial reuse.

Today, the situation is dramatically different. The World Wide Web and the widespread availability of open source software have led to a cultural shift in which software reuse is no longer considered shameful. For instance, in the aforementioned Node.js ecosystem, there are nowadays more than 700,000 reusable NPM modules (see https://www.npmjs.com/). Today, many companies and individuals are proud of the amount of the third-party code in their products. To our surprise, we recently found several automobile advertisements and reviews in which well-known car manufacturers, such as Bentley and Volvo, boast about the large amount of software in their cars, as if it was categorically a good thing.[22] For instance, the 2018 version of the Bentley Continental GT is said to contain "93 processors, feeding more than a 100 million lines of code through eight kilometers of wiring" (http://edition.cnn.com/style/article/bentley-continental-gt/index.html). Arguably, this is largely due to the traditional car design approach in which many features have their own dedicated control systems, leading to duplicate functions.[17]

In general, the opportunity to reuse software from various origins is reshaping both the way software is being developed and the way it is consumed. Compared to the 1980s and 1990s, when the amount of reused software formed only a fraction of the entire software system, the situation is now decidedly the opposite. While opportunistic designs promise short development times and rapid deployment, developers are relying more and more on code and APIs that they do not understand well or at all and yet are using them even in domains that require high attention to security and safety. A good example is the analysis provided in Morszczyzna,[14] where one particular set of dependencies is analyzed in detail, together with an analysis of associated problems.

We are concerned that the rapid growth of software systems created using opportunistic design will result in significant security problems.

Systems built with opportunistic reuse often have so much invisible code with so many dependencies that they are impossible to analyze by hand; the 2,000 plus NPM modules in our case-study system is a good example of this. Furthermore, the trend toward software systems in which components are updated dynamically on the fly (even over the air) results in dynamic dependencies that cannot be analyzed statically. The pace at which we get new versions and updates, enabled by such techniques Continuous Deployment[4] and DevOps,[3] is such that it is becoming next to impossible to test all of the combinations that may exist. API-incompatible changes in any of the underlying components may suddenly change behavior in unexpected or undesired ways or, in the worst case, render the entire system useless. Furthermore, removing a single package from the repository can result in a failure in numerous, seemingly unrelated projects.[21]

While such changes may be just a nuisance in a simple desktop application, they could be fatal in an embedded software system, such as in software controlling critical systems of an automobile, airplane, or large machinery. Such changes may also result in security attacks caused by the inadvertent injection of malicious NPM modules with names similar to those of popular modules. In fact, there is a recent example in which hackers injected malicious code into a very widely used NPM module (with more than 2 million downloads) with the aim of surreptitiously stealing money from bitcoin wallets. The injection of malicious code remained unknown to users from early October until mid-November 2018.[5]

Leslie Lamport famously described distributed systems as "one in which the failure of a computer you didn't even know existed can render your own computer unusable."[10] We have our own similar view of modern software development: It is characterized by failures that occur because there were changes in components that you didn't even know your software depended on. While opportunistic design has been recognized for more than a decade (for instance, *IEEE Software* published a special issue focusing on this theme in November/December 2008[15]), not much has happened in terms of concrete tools and other support for developers.

Call to Action

The basic challenge in opportunistic design is that it does not follow any systematic, abstraction-driven approach. Instead, as characterized by Hartmann et al.,[6] developers create significant systems by hacking, mashing, and gluing together disparate, continually evolving components that were not designed to go together. Developers publishing such components often have no formal training in creating high-quality software components, and the developers performing opportunistic, ad hoc reuse might not have any professional skills for selecting and combining such components.

As a result of these trends, the software industry is undergoing a paradigm shift. Unlike in the past, when software reuse was just an anomaly, reuse is now becoming the norm for any significant software-development projects. Yet software reuse is occurring in a very different way than originally envisioned a few decades ago. It is also quite surprising how little attention these dramatic changes and the current massive scale of reuse have received in the software engineering research community. In fact, software reuse was even declared dead in the late 1990s.[18]

The software engineering research community needs a call to action. Software reuse is finally occurring in a very large scale, but the level of awareness of opportunistic reuse and the tip-of-the-iceberg development approach in the software engineering research community has remained surprisingly low. We argue that academic researchers have not realized how significantly the effortless availability of vast numbers of open-software components is affecting software development. Meanwhile, today's developers are not generally familiar with useful software reuse principles and practices from decades ago. In a way, software reuse is "a lost art" that is now being reinvented by practitioners with little attention to extensive research and development efforts in the 1980s and 1990s.

What should be done about this? We provide here a summary of possible actions and topics that offer research opportunities ranging from analytical work to constructive development and risk management:

- systematic analysis of the compatibility of the most popular open source components for key domains and recommendations of best available components for each area, based on objective reviews and measurements in real-world applications
- study and definition of recommended reuse patterns and combinations of the most popular open source components
- tools for visualizing the static and dynamic dependencies of all the "underwater" components in a tip-of-an-iceberg software

ABOUT THE AUTHORS

TOMMI MIKKONEN is a professor of software engineering at the University of Helsinki. His research interests include web programming, software architectures, and IoT systems. Contact him at tommi.mikkonen@helsinki.fi.

ANTERO TAIVALSAARI is a Bell Labs fellow at Nokia Bell Labs. Contact him at antero.taivalsaari@nokia.com.

system that relies extensively on SOUP components [preferably, the tools should enable the monitoring of component evolution (e.g., dynamic, regularly updated dependency charts) in widely used component subsystems loaded on the fly from third-party sources; visualizing the dependencies by version history in essence results in the ability to replay the evolution of analysis results]

- tools and techniques that enable the development and testing of "iceberg" software systems within safe boundaries [such sandboxing technologies are especially important in complex systems in which software runs on multiple servers or VMs; for instance, with Docker Compose (https://docs.docker.com/compose/), it is possible to package an entire cloud onto a single machine for testing purposes ahead of deploying the system onto an actual farm of servers or VMs; this also supports the creation of validated snapshots that can be isolated from the evolution of the component subsystem]
- tools and techniques that expose programming errors as early as possible, minimizing risks, and allowing recovery with minimal damage to the end users (such techniques are important in permissive, error-tolerant web-based systems that by default do not report their errors until absolutely necessary)
- risk-management guidance and techniques that help assess the risks associated with tip-of-the-iceberg systems that depend fundamentally on rapidly evolving third-party components.

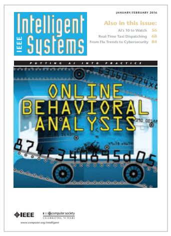
The eventual solution to programming the tip of the iceberg will be developer education to understand the contexts in which opportunistic design and tip-of-the-iceberg development are acceptable, and where more risk-aware approaches are needed. For instance, in highly regulated areas, such as medical software development, the use of SOUP components requires detailed justification, and the use of automatically updating software components is outright prohibited. To this end, practices and software reuse principles developed in the 1980s and 1990s, especially in the area of creating modular, well-documented, and stable interfaces and reusable components, provide a solid foundation to build on.

References

1. S. Aghaee and C. Pautasso, "End-user programming for web mashups," in *Proc. Int. Conf. Web Engineering*, 2011, pp. 347–351.
2. A. Bouzid and D. Rennyson, *The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business*. Bloomington, IN: Xlibris, 2015.
3. P. Debois, "Devops: A software revolution in the making," *J. Inform. Technol. Manage.*, vol. 24, no. 8, pp. 3–39, 2011.
4. M. Fowler, "ContinuousDelivery," MartinFowler.com, Apr. 2013. [Online]. Available: http://martinfowler.com/bliki/ContinuousDelivery.html
5. D. Goodin, "Widely used open source software contained bitcoin-stealing backdoor," Ars Technica, Nov. 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/
6. B. Hartmann, S. Doorley, and S. R. Klemmer, "Hacking, mashing, gluing: Understanding opportunistic design," *Pervasive Comput.*, vol. 7, no. 3, pp. 46–54, 2008.
7. T. C. Jones, "Reusability in programming: A survey of the state

of the art," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 488–494, 1984.

8. Y. Kim and E. A. Stohr, "Software reuse: Survey and research directions," *J. Manage. Inf. Syst.*, vol. 14, no. 4, pp. 113–147, 1998.
9. C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, 1992.
10. L. Lamport, "Distribution," Microsoft, May 28, 1987. [Online]. Available: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt
11. R. G. Lanergan and C. A. Grasso, "Software engineering with reusable designs and code," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 498–501, 1984.
12. E. Lippert, "Syntax, semantics, Micronesian cults and novice programmers," Microsoft, Mar. 1, 2004. [Online]. Available: https://blogs.msdn.microsoft.com/ericlippert/2004/03/01/syntax-semantics-micronesian-cults-and-novice-programmers/
13. T. Mikkonen and A. Taivalsaari, "The mashware challenge: Bridging the gap between web development and software engineering," in *Proc. FSE/SDP Workshop Future of Software Engineering Research*, 2010, pp. 245–250.
14. M. Morszczyzna, "What's really wrong with node_modules and why this is your fault," Hacker Noon, Dec. 18, 2017. [Online]. Available: https://hackernoon.com/whats-really-wrong-with-node-modules-and-why-this-is-your-fault-8ac9fa893823
15. C. Ncube, P. Oberndorf, and A. W. Kark, "Opportunistic software systems development: Making systems from what's available," *IEEE Softw.*, vol. 25, no. 6, pp. 38–41, 2008.
16. S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastapol, CA: O'Reilly Media, 2015.
17. B. O'Donnell, "Your average car is a lot more code-driven than you think," *USA Today*, June 28, 2016. [Online]. Available: https://eu.usatoday.com/story/tech/columnist/2016/06/28/your-average-car-lot-more-code-driven-than-you-think/86437052/
18. D. C. Schmidt, "Why software reuse has failed and how to make it work for you," 1999. [Online]. Available: https://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html
19. M. Shaw, "Architectural issues in software reuse: It's not just the functionality, it's the packaging," *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 3–6, 1995.
20. M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," *Computer*, vol. 36, no. 10, pp. 38–44, 2003.
21. C. Williams, "How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript," The Register, Mar. 23, 2016. [Online]. Available: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/
22. D. Zax, "Many cars have a hundred million lines of code," *MIT Technol. Rev.*, Dec. 3, 2012. [Online]. Available: https://www.technologyreview.com/s/508231/many-cars-have-a-hundred-million-lines-of-code/



Digital Object Identifier 10.1109/MS.2019.2906533