Software Reuse in the Era of Opportunistic Design

Tommi Mikkonen
University of Helsinki, Helsinki, Finland
tommi.mikkonen@helsinki.fi

Antero Taivalsaari
Nokia Bell Labs, Tampere, Finland

Abstract. 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 – 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 paper, we take a look at this increasingly popular approach in light of our industry experiences. We argue that challenges associated with such development model are quite different from traditional software development and reuse.

Keywords: Software reuse; opportunistic design; opportunistic reuse; software platforms; software mashups; mashware; economics of scale; Programmable World, the Internet of Things.

1 Introduction

In the past twenty years, the way people develop software has been affected strongly by the World Wide Web. The emergence of the Software as a Service model [20, 2], 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 for ready-made solutions for all kinds of problems online; 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], software mashups, mashware [13], or sometimes even "frankensteining" [6]. The resulting approach bears the imprint of cargo-cult programming [12] – the ritual inclusion of code or program structures for reasons that the programmers do not fully understand without even minimally attempting to understand how those components work or how they might interfere with other parts of the system.

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 non- functional requirements [19], developers have embraced this approach in droves. For instance, in client-side web development, web mashups have become very popular [1]. In cloud backend development, the use of SOUP (Software of Unknown Provenance) components is nowadays even more prevalent, given the large amount 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/). Nowadays, there are over 700,000 reusable NPM modules available for nearly all imaginable tasks.

While opportunistic reuse can be very convenient for developers, such form of reuse is rather ad hoc in its practices compared to the systematic textbook methodologies that were proposed for software reuse two or three decades earlier [11, 8]. Instead of carefully crafted designs, the resulting systems resemble icebergs, with only the "tip of the iceberg" 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 valued highly 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 company- internal use, rapid progress is often valued far more highly than small memory footprint or interface consistency; however, there are many types of software systems – especially those intended for regulated domains (e.g., medical software) – in which the use of third-party components was traditionally discouraged or prohibited altogether because of these characteristics.

In general, in recent years it has become virtually impossible to develop any significant soft- ware systems without relying at least to some degree on available components ecosystems such as the NPM ecosystem mentioned above. Opportunistic reuse is common in spite of components potentially having unknown safety-related characteristics or having been developed by unknown developers using unknown methodologies. Component selection is often based simply on popularity ratings or recommendations from other developers.

In this paper, we derive from our experience software engineering challenges arising from opportunistic design and reuse. The primary goals of the paper are to discuss and raise the awareness of how profoundly this model changes application development, characterize the changes in the form of a brief real-world example, and provide a call for action and directions for further work.

2 A Motivating Example

As an example, let us use an industrial IoT development project that we initiated about three years ago. In that project, we needed to construct a scalable cloud backend 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 three-fold. 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 in 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. We will not dive into the actual technical domain, but such incoming streaming data use cases are common for instance in the VR/AR 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 is 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 (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 backend functional ity 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 flows from measurement devices from the left towards 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 a large number of other supporting components.

From the very beginning, we wanted to make it easy to add new functionality, components and APIs on top of the base system in order to make it possible redeploy the system 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.

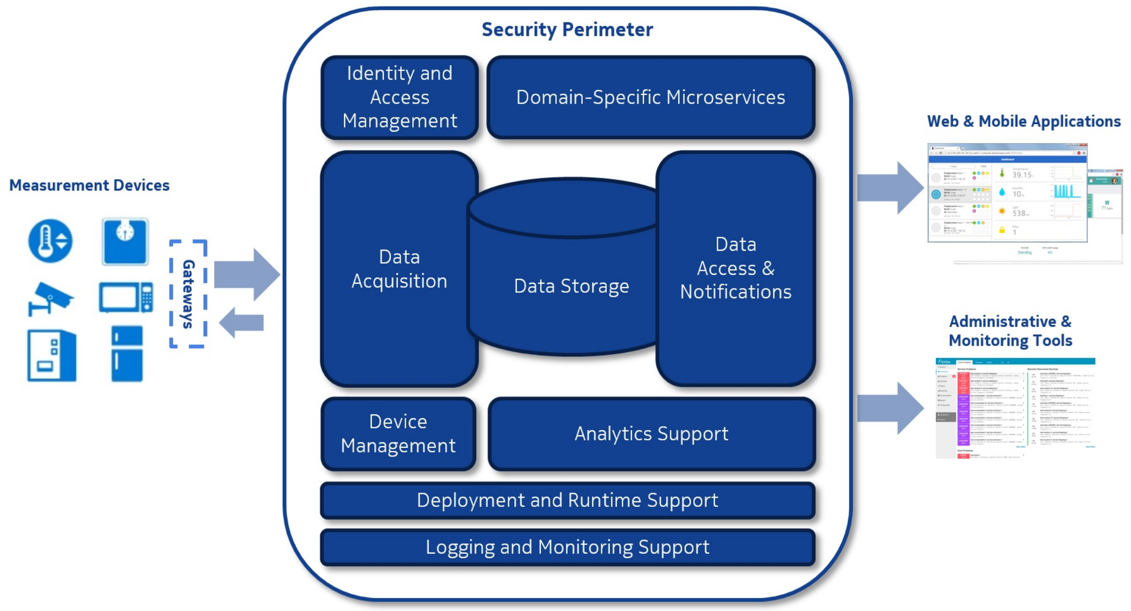
Measurement Devices
Gateways
Security Perimeter
Identity and Access Management
Domain-Specific Microservices
Data Acquisition
Data Storage
Data Access & Notifications
Device Management
Analytics Support
Deployment and Runtime Support
Logging and Monitoring Support
Web & Mobile Applications
Administrative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

Figure 1: High-Level Architecture of Our Case Study 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 writing it from scratch would have consumed all the development resources that we had avail- able 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 very beginning, we decided to rely extensively on available third-party open source software. Luckily, we had a significant amount of experience with IoT related development work and implementation components from our past projects. This helped us in the basic component selection considerably, since we knew from experience how and how well certain open source technologies worked. That said, there were dozens of other potentially applicable open source component candidates that were 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 backend development technologies, and hence a lot of 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) 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 over 30 Docker images, deployed onto 4-6 virtual machines. 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 the NPM modules pulled in by each microservice) varies from a few dozen to over a thousand per microservice. Cumulatively, the total number of different NPM modules (excluding duplicates) used by the system exceeds two thousand. While many of those NPM modules are very simple, such as *uuid*, 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.

3 Implications for Software Engineering

Although the potential for software reuse was high in the 1980s and early 1990s – e.g., Jones reported 1984 that on average only 15 percent of code is unique, novel and specific to individual applications; the remaining 85 percent appears to be common and generic [7] – actual reuse rates remained very low. Those days developers actually preferred writing their own code, and – based on the authors' personal observations – 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 avail- ability of open source software have led to a cultural shift in which software reuse is no longer a shame. For instance, in the aforementioned Node.js ecosystem, there are nowadays over 700,000 reusable NPM modules (see https://www.npmjs.com/). Today, many companies and individuals are actually proud of the amount of the third-party code in their products. To our surprise, we have recently ran into several new automobile advertisements and reviews in which well-known car manufacturers such as Bentley and Volvo proudly 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"[1]. 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 fashion software is being developed and the way it is consumed. As opposed to the situation in the 1980s and 1990s when the amount of reused software formed only a fraction of the entire software systems, 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 [14], where one particular set of dependencies is analyzed in detail, together with an analysis on 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 2000+ NPM modules in our case study system is a good example of this. Furthermore, the trend towards 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 techniques such as Continuous Deployment [4] and DevOps [3] – is such that it is becoming next to impossible to test all the combinations that may exist. API incompatible changes in any of the underlying components may suddenly change behavior in unexpected/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, this could be fatal in an embedded software system such as software controlling critical systems of an automobile, airplane or large machinery – not to mention security attacks caused by injecting malicious NPM modules named with typos similar to popular modules. In fact, there is a recent example in which hackers injected malicious code into a very widely used NPM module (with over two million downloads) with the aim of surreptitiously stealing funds stored in bitcoin wallets. The injection of malicious code remained unknown to users for about 1 1/2 months from early October until mid-November 2018 [5].

[1] http://edition.cnn.com/style/article/bentley- continental- gt/index.html

Paraphrasing Leslie Lamport's famous anecdote on distributed systems[2][2] it is possible to express the situation as follows: "Modern software development is characterized by failures that occur because there were changes in components that you didn't even know your software to depend on." While the existence of opportunistic design has been recognized for over 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 that is provided to developers.

4 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 end up creating 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.

Based on the above, there is really a paradigm shift in the making in the software industry. 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].

We feel that there is a need for a call for action for the software engineering research community at large. 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 really realized yet how significantly the effortless availability of vast numbers of open software components is affecting software development. Conversely, there are useful software reuse principles and practices from decades ago that today's developers are not generally familiar with. 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? Below we provide a short summary of proposed actions and topics that provide a wide range of 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 measured statistics of them in real-world applications.
- Study and definition of recommended reuse patterns and combinations of most popular open source components.
- Tools for visualizing the static and dynamic dependencies of all the "underwater" compo- nents in a tip of an iceberg software 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 that are 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

[2] [2]Leslie Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable". [10]

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 guidances 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 (Software of Unknown Provenance) requires detailed justification, and the use of automatically updating software compo- nents 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, stable interfaces and reusable components – provide a solid foundation to build on.

References

[1] Saeed Aghaee and Cesare Pautasso. End-User Programming for Web Mashups. In Interna- tional Conference on Web Engineering, pages 347–351. Springer, 2011.

[2] Ahmed Bouzid and David Rennyson. The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business. Xlibris, 2015.

[3] Patrick Debois. Devops: A Software Revolution in the Making. Journal of Information Technology Management, 24(8):3–39, 2011.

[4] Martin Fowler. Continuous Delivery. Available: http://martinfowler.com/bliki/ ContinuousDelivery.html, April 2013. Accessed: 2018-03-21.

[5] Dan Goodin. Widely used open source software contained bitcoin-stealing back- door. Available at https://arstechnica.com/information-technology/2018/11/hacker- backdoors-widely-used-open-source-software-to-steal-bitcoin/, November 2018. Accessed: 2018-11-27.

[6] Björn Hartmann, Scott Doorley, and Scott R Klemmer. Hacking, Mashing, Gluing: Under- standing Opportunistic Design. IEEE Pervasive Computing, 7(3):46–54, 2008.

[7] T. C. Jones. Reusability in Programming: A Survey of the State of the Art. IEEE Transactions on Software Engineering, SE-10(5):488–494, 1984.

[8] Yongbeom Kim and Edward A Stohr. Software Reuse: Survey and Research Directions. Journal of Management Information Systems, 14(4):113–147, 1998.

[9] Charles W Krueger. Software Reuse. ACM Computing Surveys, 24(2):131–183, 1992.

[10] Leslie Lamport. Distribution. Available: http://research.microsoft.com/en-us/um/ people/lamport/pubs/distributed-system.txt, May 28, 1987. Accessed: 2018-05-03.

[11] R. G. Lanergan and C. A. Grasso. Software Engineering with Reuseable Designs and Code. IEEE Transactions on Software Engineering, SE-10(5):498–501, 1984.

[12] Eric Lippert. Syntax, Semantics, Micronesian Cults and Novice Programmers. Available at https://blogs.msdn.microsoft.com/ericlippert/2004/03/01/syntax-semantics- micronesian-cults-and-novice-programmers/, 2004. Accessed: 2018-04-22.

[13] Tommi Mikkonen and Antero Taivalsaari. The Mashware Challenge: Bridging the Gap Be- tween Web Development and Software Engineering. In Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research, pages 245–250. ACM, 2010.

[14] Mateusz Morszczyzna. What's really wrong with node_modules and why this is your fault. Hackernoon, Dec 18, 2017. Available at https://hackernoon.com/whats-really-wrong- with-node-modules-and-why-this-is-your-fault-8ac9fa893823.

[15] Cornelius Ncube, Patricia Oberndorf, and Anatol W Kark. Opportunistic Software Systems Development: Making Systems From What's Available. IEEE Software, 25(6):38–41, 2008.

[16] Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.

[17] Bob O'Donnell. Your average car is a lot more code-driven than you think. USA Today, Jun 28, 2016. Available at https://eu.usatoday.com/story/tech/columnist/2016/06/28/your- average-car-lot-more-code-driven-than-you-think/86437052/.

[18] Douglas C. Schmidt. Why Software Reuse has Failed and How to Make It Work for You. Available at https://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html, 1999. Ac- cessed: 2018-03-22.

[19] Mary Shaw. Architectural Issues in Software Reuse: It's not Just the Functionality, it's the Packaging. In ACM SIGSOFT Software Engineering Notes, volume 20, pages 3–6. ACM, 1995.

[20] Mark Turner, David Budgen, and Pearl Brereton. Turning Software into a Service. Computer, 36(10):38–44, 2003.

[21] Chris Williams. How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. The Register, March 23, 2016. Available at https://www.theregister. co.uk/2016/03/23/npm_left_pad_chaos/.

[22] David Zax. Many Cars Have a Hundred Million Lines of Code. MIT Technology Review, December 2012.