Programming the Tip of the Iceberg: Software Reuse in the 21st Century

Antero Taivalsaari
Nokia Bell Labs
Tampere, Finland
antero.taivalsaari@nokia-bell-labs.com

Niko Mäkitalo
University of Helsinki
Helsinki, Finland
niko.makitalo@helsinki.fi

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

***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 recently 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 a rather *ad hoc*, mix-and-match fashion. In this paper, we take a look at this emerging approach and argue that challenges associated with such development model are quite different from traditional software development.**

***Keywords*-Software reuse; software engineering; opportunistic design; opportunistic reuse; software platforms; software mashups; mashware; economics of scale.**

I. Introduction

General purpose, commercially available software component libraries have been proposed ever since the famous NATO 1968 conference in which the term software engineering was also introduced [1]. As a research topic, software reuse became especially popular in the 1980s [2], [3], [4], following the successful workshop on software reuse arranged by ITT Corporation in September 1983. In practice, commercial success of large-scale software reuse and component libraries did not begin until the 1990s, though.

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 [5], [6] has enabled an approach in which people routinely trawl for ready-made solutions for specific problems online; the discovered libraries and code snippets are included in applications with little consideration or knowledge about their technical quality. While such an approach can be very convenient for developers, such form of reuse is very *ad hoc* in its practices compared to the systematic textbook methodologies that were proposed for reuse already two or three decades ago [2], [7].

At the same time, open source software components have become available for nearly all imaginable areas of software development. For instance, one area that has been profoundly impacted by the emergence of open source software is backend development for cloud-based systems. Twenty years ago – during the Internet boom in the late 1990s – if one wanted to build a cloud-based Internet service, each company had to create their own custom solution. Typically, each company bought the biggest servers that they could afford (e.g., Sun's E10000 server or perhaps even a few of them for redundancy), and then installed web server software and the Java Enterprise Edition (J2EE) development stack on these machines. Nearly all other software had to be written from scratch. Today, open source components for backend development abound, and cloud system developers rarely have to write any of the major components themselves. Today's backend system development is mainly about picking the most applicable components and then configuring and orchestrating those components to work together. The need to acquire physical machines has largely disappeared, too, since the majority of backend components can be hosted in public clouds such as AWS, IBM Cloud, or Microsoft Azure.

In general, the emergence of the Software as a Service model and the widespread availability of open source components has led to an approach in which developers rarely write any significant new systems entirely from scratch. Rather, they construct systems by trawling the Internet for most applicable ready-made components online, and then combining and configuring those already existing solutions, decorating them with relatively minor application-specific modifications and additions. This approach is all about combining unrelated, often previously unknown hardware and software artifacts by joining them with "duct tape and glue code" [8]. Depending on one's viewpoint and desired connotation, such development is referred to as *opportunistic design* [8], *opportunistic reuse*, *ad hoc reuse*, *scavenging* [9], *mashware* [10], [11], or sometimes even "*frankensteining*" [8]. 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.

Although it is widely admitted that opportunistic designs are not automatically optimal and that such designs may require significant architectural adjustments to fulfill functional or non-functional requirements [13], developers have embraced this approach in droves. For instance, in client-side web development, *web mashups* have become very popular [14]. 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 830,000 reusable NPM modules available for nearly all imaginable tasks (http://www.modulecounts.com/). The corresponding numbers for the Java and Python ecosystems are over 280,000 modules (Java Maven repository) and 180,000 modules (PyPI Python Packet Index). 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.

As a lower-level example, the emergence of cheap, off-the-shelf hardware has resulted in a situation in which hardware components are often used in applications that have little to do with the original use cases for those hardware components. For instance, mobile phone chipsets are nowadays commonly used as a starting point for many other types of hardware projects simply because mobile chipsets and development boards are produced in much larger quantities and are thus significantly cheaper than dedicated custom components. Similarly, Raspberry Pi (https://www.raspberrypi.org/) or Arduino (https://www.arduino.cc/) boards are commonly used in various types of Internet of Things (IoT) development projects, even though such boards may be ”overly capable”, i.e., have too much computing power and storage capacity for the actual needs of simple sensing solutions.

The availability of reusable assets for just about any domain has changed the nature of system development profoundly. Contrary to textbook examples, developers rarely perform reuse in a planned and managed fashion, as assumed in product line development [15]. Instead, developers tend to reuse software in an opportunistic, mix-and-match fashion, trawling and scraping the Internet for most suitable or most easily available components at the time when they need them. The resulting systems resemble *icebergs*, with only the ”tip of the iceberg” written by developers themselves, whereas the bulk of the system comes from other sources and remains invisible and poorly understood under the water. Quite often, the developer has no idea of what code or how much code they are actually reusing, since the included components may dynamically pull in hundreds or even thousands of additional subcomponents. Such invisible, transitive reuse is especially common in the aforementioned Node.js/NPM ecosystem.

While it is acknowledged that software reuse may introduce significant productivity gains [7], the scale of opportunistic design and *ad hoc* reuse has received surprisingly little attention among researchers. Since this model is becoming common in various domains, the software engineering community must start seriously studying and considering the consequences and challenges associated with the approach.

In this paper, we examine the software engineering challenges arising from opportunistic design and reuse. This paper is an expanded variant of a short insights article that we recently published in IEEE Software [16]; while the insights article communicates our position and includes a brief motivating case study, this paper introduces the relevant academic background and connects the work to the broader scope of reuse in general. The primary goals of the paper are to discuss how profoundly this model changes application development, extend the call for action presented in [16], and provide directions for further research.

II. Implications for Software Engineering

The first author of the paper wrote his doctoral thesis on software reuse in the early 1990s [17]. Revisiting the literature from that era, there are some interesting statistics. In the 1980s, studies showed that a considerable amount of time in software development was spent on designing routines and structures that are almost identical with constructs used in other programs. In 1984, Jones reported that on average only 15 percent of code is truly unique, novel and specific to individual applications; the remaining 85 percent appears to be common, generic, and concerned with making the program to cooperate with the surrounding execution environment [18]. Other studies in the 1980s reported potential reuse rates between 10 and 60 percent [2], [3], [4].

Although the *potential* for software reuse was high in the 1980s and early 1990s, actual reuse rates remained very low. Those days developers actually preferred writing their own code, and took pride on 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 World Wide Web, trade magazines and journals were the primary source for advertisements and reliable reviews of third-party components. Furthermore, before the widespread adoption of open source software development, components were rarely available for free or with license terms favoring large-scale use.

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 a shame. On the contrary, many companies and individuals today are actually proud about the amount the third-party code in their products. For instance, to our surprise, we have recently ran into several new automobile advertisements and reviews in which well-known car manufacturers (e.g., Bentley and Volvo) proudly boast about the large amount of software in their cars, as if it was categorically a good thing [19]. 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].

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

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

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 becoming accustomed to designs that they do not fully understand, and yet using them even in domains that require high attention to security and safety.

We are concerned that the rapid growth of software systems created using opportunistic design will result in significant security problems. Such systems often have so much invisible code with so many dependencies that they are impossible to analyze by hand. 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 new techniques like Continuous Deployment [20] and DevOps [21] – is such that it is next to impossible to test all the combinations that may exist. API incompatible changes in any of the underlying components may suddenly render the entire system useless. While such behavior 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.

Paraphrasing Leslie Lamport's famous anecdote on distributed systems[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 know your software to depend on.*"

In some ways, these challenges have existed as long as computer hardware and operating systems have provided abstractions – an important associate of reuse [9] – that eliminate the need to fully understand how things work at a lower level. However, the unknown provenance aspect – dynamically combining components that really were not designed to work together and that were never properly integration tested – can amplify the problems to a whole new level.

III. Discussion

According to Krueger's classic survey paper on software reuse, the keys to successful software reuse are *abstraction* and *reduction of cognitive distance* [9]. A proper abstraction provides a clean separation between internal implementation details and an external public interface. Cognitive distance is the amount of intellectual effort that must be expended by software developers in order to take a software system from one stage of development to another. According to Krueger, a successful software reuse technique must fulfill the following four "truisms" [9]:

1) For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.
2) For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.
3) To select an artifact for reuse, you must know what it does.
4) To reuse a software artifact effectively, you must be able to "find it" faster than you could "build it".

The success of opportunistic design in the past years can be attributed largely to second and fourth points above. The World Wide Web and search engines have made it easy to search for potentially applicable software components and code snippets online. Furthermore, the availability of software in open source form has made it easy to experiment with potentially applicable components without any significant financial commitment ahead of time.

Yet, contrary to Krueger's third point above, it can be argued that the overall understanding of the reused software has decreased over the years. After all, the key premise in "classic" software reuse is that there are systematically designed systems with stable, well-documented interfaces. The sheer volume of open source software makes it difficult to analyze and compare technologies in detail, let alone fully understand the abstractions exposed by them, especially in light of the highly varying quality of documentation that is characteristic, e.g., of the aforementioned NPM ecosystem.

Contrary to the classic premise of stable, reusable code, in reality the majority of successful systems are under constant change. Following a popular informal law of computing, the more successful a system is, the more likely it is that it will have to be changed. Those systems that are (re)used the most, tend to be the ones that also experience the most rapid evolution. Such evolution does not always necessarily maintain backwards compatibility, as exemplified by the recent development of *Angular* and *React* JavaScript libraries – creating a version compatible setup can easily take a considerable amount of time. This is definitely the case with popular web libraries, across all the layers of the software stack, as new versions are published. For designs that rely on installed software this may not be such a serious problem, but even in such a context there can be subsystems that are updated on the fly, e.g., as security problems are discovered.

The basic problem in opportunistic design is that it does not follow any systematic, abstraction-driven approach. Instead, as characterized by Hartmann et al [8], 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.

Hartmann et al published their observations in 2008 – at the time when the area of web mashup development (see [14]) was experiencing rapid growth. In their paper, Hartmann et al mentioned that at the time there were over 3100 web mashups leveraging 775 distinct APIs [8]. Today, the scale of

[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.*" [22]

opportunistic reuse is dramatically larger, given the nearly exponential growth of the most popular component ecosystems. At the time of this writing, the `npmjs.org` website lists over 830,000 NPM modules. Some of the most popular NPM modules, such as `lodash`, have over 80 million download requests per month, reflecting very high levels of reuse. We are clearly past the point when any individual developer could master all the components in their domain, or objectively compare and choose the best components for their system, except simply by relying on recommendations and popularity ratings.

In view of the numbers presented 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.

IV. Call for action

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 research work carried out in the 1980s and 1990s.

Below is a summary of the recommended actions and topics that provide wide range of research opportunities ranging from analytical work to constructive development and risk management.

- Systematic survey and analysis of widely used open source repositories and components, and the quality of their interfaces and documentation.
- 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.
- Systematic studies of the evolution of the most popular open source repositories and components, with special attention to the stability of their interfaces.
- Study and definition of recommended reuse patterns and combinations of most popular open source components.
- Tools for visualizing all the "underwater" dependencies in a "tip of an iceberg" software system that relies extensively on SOUP components.
- Tools that help assess the stability and maturity of reused components, e.g., how likely they will change in terms of their interfaces, and assessing how trustworthy their contributors are.
- Improved tools for static and dynamic component dependency analysis, "tree shaking" (for eliminating duplicate components), crawling to the end of dependency chains to create transitive closure of all the needed modules, and so on.
- Tools and techniques (e.g., dynamic, regularly updated dependency charts) to monitor and understand changes in widely used component subsystems that are loaded on the fly from third party sources;
- 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.
- 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.

Indeed, successful opportunistic reuse is heavily dependent on *risk management*. The use of third-party components – especially if it occurs in a fashion in which first-level reused components end up transitively pulling in layers and layers of other components – raises the risks associated with a software system considerably. While concerns regarding the issues around trust have been raised already over 30 years ago [23], the fact that unrelated software components are nowadays so commonly fused together means that software written with even the best of intentions can introduce severe problems. This is especially true when those components are developed by unknown developers using unverified development methods, and if those components are allowed to update themselves dynamically. Risk management in the context of large-scale opportunistic reuse is still a surprisingly little studied topic.

More broadly, while many of the individual topics above have been studied earlier, we feel that a holistic approach for tackling the challenges introduced by opportunistic design is still missing. Effectively, the bullets listed above form a research agenda that we plan to follow in our work, e.g., in teaching our students and new recruits as well as in proposing relevant research topics to our graduate students.

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, stable interfaces and reusable components – provide a solid foundation to build on.

V. Conclusions

In the late 1990s, software reuse was declared dead [24]. The rumblings about the death of software reuse turned out to be premature, though. Over the past 5-10 years, large scale software reuse has finally become a reality. However, software reuse has happened in a very different way than originally envisioned by the software engineering community and its pioneers. While McIlroy's original 1968 vision of called for high-quality mass produced software components to be used in a large, industrial scale [25], today's software reuse scene is really all about hordes of software developers producing a cornucopia of overlapping open source components of varying quality – as exemplified by the extremely popular Node.js NPM ecosystem with its hundreds of thousands of modules.

At some point after the turn of the millennium, a tipping point was reached. Nowadays, it is nearly impossible to write any significant software systems without reusing third-party components extensively, with the developers themselves only writing the "tip of the iceberg" – while the bulk of the system comes from external sources and unknown developers. This is dramatically different from software development in the 1980s and 1990s when developers still prided on writing most of the software themselves.

In hindsight, it is really interesting note how quickly the world of software development evolved from "not invented here"[3] to a *smørgasbord* model with a cornucopia of ready-made components and modules available for nearly every imaginable domain and purpose. Looking back, the reasons for this transition should have been obvious, including the ease of publishing components on the Internet, the ease of searching, reading and posting reviews and trustworthiness ratings, as well as the general shift towards openness and open source software, complemented with the growing overall size of software systems. Instead of using systematic catalogues, component libraries or textbook methods for software reuse, developers perform web searches for potential components online, and then rely on popularity ratings and recommendations in choosing the most suitable candidates.

In this paper, we have studied opportunistic design and its implications, and presented a call for action for the research community. It is our belief that opportunistic reuse and the "tip of the iceberg" programming model raise the need for risk management to a whole new level than in traditional software reuse. We hope that this paper raises the awareness of opportunistic reuse, and encourages people to tackle the challenges associated with this important topic.

[3]Belief that in-house developments are inherently better, more secure, more controlled, quicker to develop, and incur lower overall cost (Wikipedia).

References

[1] P. Naur and B. Randell, *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee (Garmisch, Germany, Oct 7-11, 1968)*. NATO Scientific Affairs Division, Brussels, 1969.

[2] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reuseable Designs and Code," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 498–501, 1984.

[3] M. Lentz, H. A. Schmid, and P. F. Wolf, "Software Reuse Through Building Blocks," *IEEE Software*, vol. 4, no. 4, pp. 34–42, 1987.

[4] T. J. Biggerstaff and C. Richter, "Reusability Framework, Assessment and Directions," *IEEE Software*, vol. 4, no. 2, pp. 41–49, 1987.

[5] M. Turner, D. Budgen, and P. Brereton, "Turning Software into a Service," *Computer*, vol. 36, no. 10, pp. 38–44, 2003.

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

[7] Y. Kim and E. A. Stohr, "Software Reuse: Survey and Research Directions," *Journal of Management Information Systems*, vol. 14, no. 4, pp. 113–147, 1998.

[8] B. Hartmann, S. Doorley, and S. R. Klemmer, "Hacking, Mashing, Gluing: Understanding Opportunistic Design," *IEEE Pervasive Computing*, vol. 7, no. 3, pp. 46–54, 2008.

[9] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[10] T. Mikkonen and A. Taivalsaari, "The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering," in *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research*. ACM, 2010, pp. 245–250.

[11] A. Taivalsaari and T. Mikkonen, "Mashups and Modularity: Towards Secure and Reusable Web Applications," in *23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE, 2008, pp. 25–33.

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

[13] M. Shaw, "Architectural Issues in Software Reuse: It's not just the Functionality, it's the Packaging," in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI. ACM, 1995, pp. 3–6.

[14] S. Aghaee and C. Pautasso, "End-User Programming for Web Mashups," in *International Conference on Web Engineering*. Springer, 2011, pp. 347–351.

[15] P. Clements and L. Northrop, *Software Product Lines*. Addison-Wesley, 2002.

[16] T. Mikkonen and A. Taivalsaari, "Software Reuse in the Era of Opportunistic Design," *IEEE Software*, vol. 36, no. 3, pp. 105–111, 2019.

[17] A. Taivalsaari, *A Critical View of Inheritance and Reusability in Object-Oriented Programming (Doctoral Thesis)*. Jyväskylä Studies in Computer Science, Economics and Statistics 23, University of Jyväskylä, Finland, 1993.

[18] T. C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 488–494, 1984.

[19] D. Zax, "Many Cars Have a Hundred Million Lines of Code," *MIT Technology Review*, Dec. 2012.

[20] M. Fowler, "ContinuousDelivery," Available: http://martinfowler.com/bliki/ContinuousDelivery.html, Apr. 2013, accessed: 2017-10-21.

[21] P. Debois, "Devops: A Software Revolution in the Making," *Journal of Information Technology Management*, vol. 24, no. 8, pp. 3–39, 2011.

[22] L. Lamport, "Distribution," 1987. [Online]. Available: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt

[23] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[24] D. C. Schmidt, "Why Software Reuse has Failed and How to Make It Work for You," 1999, accessed: 2017-11-22. [Online]. Available: https://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html

[25] M. D. McIlroy, "Mass Produced Software Components," in *Naur and Randell (eds): Software Engineering: Report of Conference Sponsored by the NATO Science Committee, Garmisch, Germany, Oct 7-11, 1968*, pp. 79–85.