Jingkang Zhang

Christian Nagler

Theater R1B

Dec. 19, 2019

Software Reuse and Open Source Software

The concept of reuse has been central to problem solving and innovation throughout human history. Without building on previous work, most of the brilliant accomplishments by human civilization wouldn't be possible. While the importance of knowledge reuse is self-evident in various natural science, social science, and engineering disciplines, even in highly self-contained fields such as mathematics, or creative fields such as art creation, inspirations from existing work or tacit know-hows have been crucial to the process of creation. As the inception of the Internet and onset of the information age have led to great upheavals in nearly all aspects in our daily life (including how knowledge itself is created, distributed, and reused), reuse, without exception, plays an indispensable role. Specifically, software reuse, the reuse pattern lately emerged with the practice of creating software and the foundation only on which the rapid growth of the information age was made possible, is worth investigating. As a special yet identifiable form of reuse, software reuse facilitates the rapid emergence of various personal devices, network infrastructures, and services that together ushered in the digital age. After introducing backgrounds of software reuse, identifying reuse practices on different levels, and presenting difficulties faced by software reuse, open source software will be introduced as a promising practice that celebrates existing reuse patterns and overcomes the introduced difficulties.

Computer Software, or software, is the computer code that instructs computers (hardware) what to do in a narrower sense. In the broader sense, however, software reuse is "the use of existing software or software knowledge to construct new software" (Frakes and Kang 2005). While the "use of existing software" can be easier to understand as we tend to imagine it as duplications of whatever tangible form (source code, for example) software is in and modifications thereafter, "use of software knowledge" requires further clarifications because of its abstractness and sometimes lack of explicit reuse pattern. In fact, aside from source-code fragments, reuse applies to all the intermediate work products generated during software development, including requirements documents, system specifications, design structures, and any information the developer needs to create software (Prieto-Diaz 1993). Generic algorithms in software engineering, for example, are not fundamentally different from mathematical theorems or models. They are generalized high-level abstractions that may be applied to specific problems through different implementations. Thus, practices of reusing algorithms in development of computer software is not dissimilar from using mathematical formulas in mechanical engineering. Code reuse, on the other hand, is unique and inherent to the relatively young software engineering discipline, as various software foundational to the whole industry, including the UNIX system and C language, are all created encouraging reusability (Frakes and Kang 2005).

The purpose of software reuse, in the broader sense as discussed above, is to improve software quality and productivity (Frakes and Kang 2005). Because of the importance in this purpose, software reuse, especially systematic and formal reuse, has been the focus of many computer scientists and management scientists. Much like other disciplines, creation of value in software engineering largely builds on previous work, both high-level abstractions (algorithms) and reusable components (*wheels* in software engineering terms). The

algorithmic knowledge reuse in software engineering works similarly to high-level knowledge reuse in any other industries or subjects — models in statistics, design guidelines in architecture, or even metrics in poem writing — that workers use the organized knowledge as an abstract framework and apply it to solve individual problems by filling in the skeleton and making adjustments. However, the special capacity of "direct reuse" as inherent in the way software is created and in the form they exist makes components highly reusable compared to any other engineering discipline. In other words, the cost of reusing code is disproportionally low compared to reusing artifacts in most other engineering disciplines. For example, to solve a recurring problem while designing a software, it is possible for a software engineer to copy the source code from an existing project and apply low effort to adapt it to the current project. However, it is not as easy to do the same, for example, in architecture. If a civil engineer is constructing a campus, could she "copy" the gate from another beautiful school that she has been proud of, and "paste" it onto her current construction site? The answer might be positive, in the sense that she can borrow the *design* of the gate and implement it again. However, our technology don't allow (yet) the automatic duplication of physical objects. In software engineering, on the other hand, the product is not physical, and computer allows for the low-cost duplication of available source code. The difference between borrowing a design and copying over the code is that the code fragments are ready-made product that can be "plugged" into current code base without the need of re-implementing a design, thus giving software engineering a natural edge over traditional industries in reusing components.

The reusability of software also partially accounts for the rapid onset of the information age. In mere decades, the Internet developed into a gigantic network of thousands of millions of personal computers and servers. On each of the devices, various software, including the

operating system they are running on, all require enormous effort by many computer scientists. Nonetheless, after the initial design, little work is required to deploy the software onto individual machines. The fast growth of technical companies also owes to the fact that the software they produce can be written once and distributed arbitrarily with minimal manufacturing costs. The internal logic of software production and distribution is analogous to how books are distributed after the invention of printing— once the replication cost is lowered, wide influences and fast growth are promised.

However, as the advantage of easy reuse promises a fast growth, "growing pains" also accompany the industry. As Prieto-Diaz points out, "the problem is not a lack of software reuse, but a lack of systematic reuse. Industry will achieve big payoffs only if this can be changed" (1993). Indeed, systematic and formal methods of software reuse will be key to further improving the quality and speed of the creation of new software even from its already advantageous starting point. More than twenty years after Prieto-Diaz made this statement, the outlook has definitely changed for better today, as we see more and more patterns developed striving for "systematic reuse", which unarguably have generated "large payoffs" and advanced the industry in ways unimaginable in 1993. Thus, before we investigate the reuse difficulties we face even today, we should first acquaint us with the numerous efforts made by computer scientists and software engineers over the years, to build our foundational knowledge about the inherit characteristics of software engineering that are central to understanding both the advantages and difficulties about software reuse.

First, the concept of reuse has been heavily represented in the evolvement of programming languages, the very form in which production labour is devoted. According to Frakes and Kang, there are two important ways in which programming languages and reuse are tightly coupled. First, "programming languages have evolved to allow developers to use

ever larger grained programming constructs, from ones and zeros to assembly statements, subroutines, modules, classes, frameworks, etc" (2005). While this gradient of abstractions in software engineering mentioned by Frakes and Kang is very technical, language-specific, and thus unnecessary to thoroughly define for the purpose of this article, I'd like to take one of the above mentioned levels of reuse, subroutine, and explain how it facilitates reuse. A subroutine is a "function" that the developer write once and can be "called" anywhere in the project. In this sense, even though the actual code written in the subroutine will be run multiple times during the execution of the program, thanks to language support for functions, the code only appears once in the code base — in other places where it should do its job, it is simply "called" with one statement. Thus, even though the cost of copying and pasting code is already low, programming languages' built-in support for code reuse can greatly reduce code size, improve the readability and maintainability of the code base, and reduce the debugging costs. Second, "programming languages have evolved to be closer to human language" (Frakes and Kang 2005). This fact also helps with code reuse, as the less effort is required from developers to understand the function of a certain fragment of code, the easier it is for them to incorporate existing code into their new work. Interestingly, the achieved similarity to human language is also thanks to the different levels of abstractions. As more abstractions, especially the concept of "object", are introduced to various programming languages, the data stored linearly in physical machines can be represented in more diverse, "3D", and intuitive ways. With "classes", nearly all concrete nouns in natural language can be represented in high level programming languages. In the code, for example, programmers can define object representations for a "customer", an "account", or a "transaction", store their unique information in their separate "fields" in an organized manner, and define the actions available for each object through "methods". With better abstractions, programmers feel like

they are directing a movie by instructing their actors to play their individual roles, instead of playing a movie all by theirselves by repeating every single action. Also due to this analogy, code written in modern high level languages is sometimes called “scripts” in computer science terms.

Second, software architecture has been recognized as an important consideration for reusing software since the late 1980’s (Frakes and Kang 2005) with the emergence of domain analysis and domain engineering. Early architectural decisions define the functions of code to be contributed to the system in later stages of software lifecycle and are therefore difficult to change late in the lifecycle. Domain analysis is the activity of “identifying objects and operations of a class of similar systems in a particular problem domain” (Neighbors 1980). Thus, architectural reuse of software is also closely related to domain engineering — a successful design of a generic template for a particular domain can be proven useful for other software dedicated to this domain. Thus, efforts have been made to formalize software architectures, because important decisions made in the process are highly applicable to new systems, especially those in the same domain. Additionally, generic templates will be the basis for creating components that are easy to reuse (Prieto-Diaz 1993). Therefore, good architectural choices can also facilitate the creation of reusable code on the lower level.

Third, procedures reuse, the most abstract and high-level reuse method, focuses on “formalizing and encapsulating software development procedures” (Prieto-Diaz 1993). Concerned less with the reuse of specific code fragments or even architectural decisions, procedures reuse means reusing expert skills and know-how. Development procedures, the process in which software is created, are concerned with development tools, development cycles, and collaboration patterns. More akin to managerial practices, this area has received significant attention from the expert-systems community (Prieto-Diaz 1993). According to

Prieto-Diaz, “Project managers tend to reuse skills informally when they reassign personnel, but no formal effort is made to capture and encapsulate knowledge” (1993). Procedures reuse falls into the category of applying “operational knowledge” as proposed by Karl Wiig in 1993. Defined by Wiig, operational knowledge “deals with all types of knowledge that may be used to make decisions, perform analyses, provide direction and guidance, or create new concepts and approaches” (1993). While procedures as a type of operational knowledge in software engineering may seem equivalent to its counterparts in nearly all other production fields that exceed a certain level of sophistication, the significant difference in how production is carried out and the form in which value is stored makes it necessary to investigate separately reusing expert procedural knowledge in software engineering. For example, the highly modularized nature and thus the high alterability of modern software allows for quick iterations of new versions of existing software, thus giving rise to efficient patterns of code development such as Agile Software Development which cherishes “responding to change over following a plan” and “customer collaboration over contract negotiation” (Beck et al. 2001), defying traditional business patterns. Domain-analysis researchers, similarly, have also been exploring how to reuse expert procedural knowledge as the practices to tackle problems specific to certain domains have enough similarities that it’s worth extracting the methods for future projects.

While the importance of formalizing software reuse methods has been widely acknowledged and efforts have been made to improve software reuse practices on various levels, software reuse has also faced many difficulties and limitations technically, legally, and morally. The technical challenge attributes to the complexity in the useful abstraction for large, complex, reusable software artifacts (Krueger 1992). According to Krueger, “in order to use these artifacts, software developers must either be familiar with the abstractions a

priori or must take time to study and understand the abstractions". Indeed, without solid understandings of the function of the component, it can be ineffective or even dangerous for software engineers to reuse the component. Aside from time commitment by individual developers required to understand the functionalities, the abstractness of the complex artifacts also poses a difficulty for classification, search, and exposition essential to constructing a reuse library. By definition, a reuse library "consists of a repository for storing reusable assets, a search interface that allows users to search for assets in the repository, a representation method for the assets, and facilities for change management and quality assessment" (Frakes and Kang 2005). While the costs of building and maintaining such a library can be high due to the software complexity, the pay-off can also be great. In light of this, some may argue that with proper incentives such as guaranteed better searchability and manageability, the costs involved in building such a library will likely be balanced out. While this is correct and is the reason why reuse libraries may exist in the first place, Markus' introduction of another drawback of more general knowledge repositories renders a grimmer prospect. According to Markus, "repositories created by one group for one purpose are unlikely to be successfully reused by other groups for different purposes without considerable rework or other kinds of intervention" (2001). Applying this proposition to our case of software library — with differences in purposes as reflected in the different classification methods, presentation interfaces, and management options, the reuse library built by one group during a certain period may not be as useful to another group or the same group in a different period, and may require considerable rework. This will also undermine the value and usefulness of a reuse library and thus lower the incentives of its creators.

Aside from technological and managerial limitations as discussed above, software reuse is also affected by legal and moral restrictions. Legally, the actual code implementing

software innovations has been a dangerous commodity (Lemley and O'Brien 1997). According to Lemley and O'Brien, "while new programming structures and techniques spread quickly as programmers moved from job to job, copyright law punished the transfer of actual code implementations of these structures and techniques. … Rather than risk liability for copyright infringement, programmers routinely sacrificed the benefits of successive refinements and bug fixes in existing code in favor of recoding the software themselves from scratch". Indeed, as a relatively new, yet highly profitable field, software engineering unsurprisingly faces strict legislations aiming to protect the actual products from being copied over with minimal effort. However, precisely due to the form in which new technologies usually exist — actual code in the form of a module, library, or framework — direct reusing of the test-proven and thus less bug-prone code is the proper way to reuse a new technology. Nonetheless, in conformance to copyright laws, developers refrained from copying the proprietary code, even if it pertains less to the actual product or business logic of each company but is more representative of the general "structures and techniques" that would, sooner or later, be standardized in the industry. As a result, Lemley and O'Brien observed that "external reuse is uncommon even in large corporations" due to the legal concern (1997). Thus, not only developers suffer from repetitive work, but the companies are forced to favor a more competitive instead of collaborative relationship with regard to software reuse, until, as we will see later, they embrace open source software.

In relation to the legal constraints, there exists questions of ethics in different phases of software reuse and distribution: the acquisition of software, the use (or abuse) of existing software programs and information system infrastructures, the production of software that may incorporate morally questionable features, or the provision of software tools that do not meet promised or agreed-upon requirements (Sojer et al. 2014). Aside from improper

acquisition and reuse of software which directly constitute copyright infringement as discussed earlier, other immoral acts can extend to all kinds of dealings with software. For example, “DeepNude”, a member of the “DeepFake” software family which utilizes artificial intelligence algorithms to create realistic fake videos or photos, functions to “undress” a woman by generating a naked picture of her, and has raised wide moral concerns in 2019. Shortly after its launch, the DeepNude on Twitter announced its termination for “the world is not yet ready for DeepNude”. Andrew Ng, a famous machine learning computer scientist and entrepreneur, commented on Twitter that DeepNude is “one of the most disgusting applications of AI”. As a good example of morally disputable reuse of machine learning algorithms, the project reveals how immoral features can be developed with help from existing software. With published machine learning software libraries ready for reuse, it has become significantly easier for developers to “train” machine learning models that employ similar algorithms as invented before, yet serve completely different purposes, achieved by feeding different “training sets” and adjusting parameters. As evident in the invention of DeepNude, we can see that the convenience to create new software thanks to various reuse methodologies also leads to the convenience to develop morally questionable software, proposing a challenge greater than the “how” aspects of reuse — “how not”.

In response to the above constraints and taking advantage of inherent characteristics and traditional practices in software reuse, the emergence of open source software and its derivatives such as GitHub have proposed a promising prospect of improved software accessibility and more fruitful code sharing communities. Extraordinarily successful, a group of open source software including GNU/Linux, Apache, Bind DNS server, OpenOffice, and Mailman has greatly benefited the general public. According to Haefliger, their success “has drawn attention from the public and both software-creating and software-using organizations

to this way of developing software." (2008) Going from the above discussion, we will investigate how open source software is able to celebrate the natural advantage of reuse and forage a constructive community, which, in turn, addresses many of the limitations we mentioned above.

First and foremost, various open-source licenses have gained increasing popularity in both the industry and academia as a forceful response to the legal restraints that have long limited industry-wide collaboration among different entities. For example, academic licenses including the BSD and MIT licenses were originally developed in higher education institutions, but became widely employed by many of the most popular industrial software. BSD (Berkeley Software Distribution) and its derivative licenses developed at University of California, Berkeley in its earlier exploration of developing computer operating systems actually laid foundation of and are still employed by a wide variety of UNIX-like operating systems including the largely successful macOS and iOS by Apple. The MIT license, the most popular open source license as of 2018 (Goldstein 2018), is employed by many of the most successful open source software which is developed, maintained, and used by a wide range of entities from internet tech giants such as Google, Facebook, Twitter, etc, to tech start-ups and individuals. Popular projects under MIT license include Node, Ruby on Rails, jQuery, and React.js. The striking popularity of the MIT license is thanks to its greater permissiveness than most other open source licenses including GPL, Apache, or BSD which are more restrictive (GPL, for example, is "viral", in the sense that modifications of GPL-licensed software must be released under the same license, whereas MIT license permits the use of MIT-licensed code even in proprietary software). The permissiveness of the MIT license is well reflected in its succinct license texts:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction,
> …
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
> …
> IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM
> (The MIT License)

As we can see, by "granting permission to deal in the Software without restriction", the MIT license gives significant liberty to the public to use, modify, and redistribute the software. As Haefliger et al. points out, "such a license inherently encourages a developer to reuse code" (2008). Aside from eliminating legal concerns regarding the use of the software, the MIT license also encourages the improvement, contribution, and redistribution of the software by waiving the authors of legal responsibilities in creation of the software. This premise forms the foundation of building an open source community in which developers may freely contribute to the codebase with minimal worries about being sued for contributing. As of 2019, open source projects under various permissive licenses still maintain a promising momentum, that we can expect to see more and more companies embrace open source licenses with an increased reciprocal awareness as even their proprietary software benefit from open source.

As open source licenses mitigate legal concerns regarding software reuse, more and more corporate and individual developers start to actively engage in open source development. Because of the collaborative and reciprocal nature of open source, an effective way of building an online community was sought after. In this trend, the online code hosting website GitHub, founded in 2008, gained popularity at an astonishing speed and became arguably the largest source code hosting site in the world (Thung et al. 2013). Acquired by

Microsoft at US$7.5 billion in 2018, GitHub proves its commercial success through accumulating a large active user base, which is of great value to the tech giant Microsoft. By providing documentation, issue tracking, pull request, small website hosting, and various other features, GitHub serves not only as a useful tool to facilitate software development by providing version control and organizing the development process, but also as a robust code sharing community whose power and value scale with the number of users and projects. With numerous projects highly publicized and searchable on GitHub, the website effectively serves as a "reusable asset library" as termed by Frakes and Kang. With efforts made to address the difficulties of building a code library as we earlier discussed, GitHub gained its success among users as a user-friendly platform. First, thanks to the technological advancement in indexing and searching over the years, the search feature in GitHub functions as well as any other commercial website in other fields. Enabling the users to quickly find either a project in any particular domain or specific code inside a project and display them in a highly ordered manner, GitHub fulfills its role as a good code repository as defined by Frakes and Kang. Second, to minimize the time and energy required from users to understand and reuse the open source software (usually in the form of a library), GitHub also facilitates easy creation of project Wikis, allows free hosting of project demos through "GitHub Pages", and adds "issues" feature as a small discussion forum to each code repository, thus to some extent addressing Krueger's concern of the defeated purpose of software reuse if the time taken to understand the software is significant. As a tool that not only excels in its basic role as a asset library but also largely affects the general process of software development, GitHub, though one of the largest promoters of open source, has also been favored by large tech companies who traditionally favored closed source or proprietary practices. Facebook, Google, and Microsoft, for example, each published multiple influential projects, such as React, Go, and

VS Code, under open source licenses on GitHub. Cross-organizational collaboration among big tech companies is thus increased, not through contracted agreements, but through encouraging developers, as individuals, to contribute to open source projects which eventually benefit everyone.

Having seen how open source licenses and the biggest open source platform, GitHub, address the difficulties faced by software reuse with great success, we may further analyze, from a theoretical point of view, how the idea and practices of open source conform to, celebrate, and refine software reuse patterns on different levels. On the most basic level, the development of software is usually achieved through modifying its source code and, as mentioned before, the concept of reuse has been buried in the mind of developers since the inception of computer software. With open source, as in its literal meaning, high accessibility to the source code and generous permissions in dealing with the code are guaranteed. Thus, developers are granted with the power to take advantage of the abstractions built into programming languages on all levels to facilitate reuse: from directly copying useful snippets of code (as statements or procedures/functions) to adapt them to the developers' own code base, to importing the projects as a utilization library/module (e.g., dedicated to performing math calculations, handling network requests, or implementing advanced data structures), to building application on top of open source frameworks (e.g., to implement internet servers). On a greater level, recurring architectural design patterns and choices in different fields crucial to certain domains of problems also prosper under the open source model. In fact, a lot of these architectures, including the widely used web front-end frameworks such as Angular and React and back-end frameworks such as Ruby on Rails and Node are all published under open source licenses and are only made possible through an active open source developer community. Lastly, the derived open source communities such as GitHub to

a certain extent encapsulate the procedural knowledge in software development. With Git as its core version control tool, GitHub enables developers to track every single version of the code and provides visualizations on the structure of the project. Each commit in the code base's history is ready to be compared with any other versions so that people can easily tell what has been changed and gain a better picture of the project's course of development. Besides, the "issues", "pull requests", and "fork" features on GitHub incorporate general software development procedures into the product, making it a tool convenient not only for storing and sharing the code, but also for the community to discuss and contribute to the projects. This practice largely encourages the share of expert knowledge and implicit know-hows regarding the subtle design motives, implementation choices, and production timelines that can also be "reused" by other developers.

Overall, software reuse, the high tech incarnation of the idea of reuse which has long rooted in human's everyday activities, has been central to software development since its inception. Though with inherent reuse advantages, software development nonetheless faces numerous challenges technically, managerially, legally, and morally. In a cooperative and reciprocal spirit, open source software and its derivative platforms such as GitHub gained significant popularity among individuals and companies, answering to some of the limitations of software reuse. While impediments still exist in the way of open source, e.g., the increasing moral concerns about applications of open source AI algorithms, as we gain better understandings of both software reuse and open source, we will be able to, like we did many times in the past decades, find approaches unimagined before to create and improve software.

References

Haefliger, S., Von Krogh, G., & Spaeth, S. (2008). Code reuse in open source software. Management science, 54(1), 180-193.

Markus, L. M. (2001). Toward a theory of knowledge reuse: Types of knowledge reuse situations and factors in reuse success. Journal of management information systems, 18(1), 57-93.

Wiig, K. M. (1993). Knowledge management foundations: thinking about thinking: how people and organizations create, represent, and use knowledge (Vol. 1). Arlington, TX: Schema press.

Lemley, M. A., & O'Brien, D. W. (1997). Encouraging software reuse. Stanford Law Review, 255-304.

Prieto-Díaz, R. (1993). Status report: Software reusability. IEEE software, 10(3), 61-66.

Sojer, M., Alexy, O., Kleinknecht, S., & Henkel, J. (2014). Understanding the drivers of unethical programming behavior: The inappropriate reuse of internet-accessible code. Journal of Management Information Systems, 31(3), 287-325.

Frakes, W. B., & Kang, K. (2005). Software reuse research: Status and future. IEEE transactions on Software Engineering, 31(7), 529-536.

Thung, F., Bissyande, T. F., Lo, D., & Jiang, L. (2013, March). Network structure of social coding in github. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 323-326). IEEE.

Neighbors, J. M. (1980). Software construction using components (Doctoral dissertation, University of California, Irvine).

Krueger, Charles W. "Software reuse." ACM Computing Surveys (CSUR) 24.2 (1992): 131-183.

Beck, Kent, et al. "Manifesto for agile software development." (2001): 2006.

Ayala Goldstein. (2018) Top 10 Open Source Licenses in 2018: Trends and Predictions. Retrieved from https://resources.whitesourcesoftware.com/blog-whitesource/top-open-source-licenses-trends-and-predictions

The MIT License. Retrieved from https://opensource.org/licenses/MIT