Abstract:

My research dove into the topic of dependency management, specifically within the build tool Maven. Having an understanding of build tools is necessary for developers working on large projects. Fixing broken builds becomes a time consuming task at this level. Many studies have analyzed the relationship between best dependency management practices and the overall ability for projects that use these practices to be streamlined, automated, and reproduced. Using by research on Maven as a home base, I chose these five papers because each seemed to offer a window into the wider development world. The first research paper examines the relative reproducibility of builds between the Java tools Ant, Maven, and Gradle. The second paper delved into a related dependency management topic in Make-based build systems. The third article found an interesting way to collect data from Stack Overflow to see what queries occurred most frequently, with build failures being a significant portion of questions. The fourth article used an interesting technique of system call tracing to help locate the reason for failing builds. The fifth article explores security management in continuous integration/delivery environments, which step outside of the realm of build tools but are an area that I am interested in. All in all, these studies each represent a slightly different direction off of my original research that seem to be of particular interest in modern development environments. Thanks to this review process I have a much better idea of the world of build tools outside of Maven as well as other pertinent topics.

Title: A Quantitative Study of Java Software Buildabilty

Summary:

This paper sought to analyze a large number of open-source Java projects using the three most used build tools in the Java ecosystem: Maven, Gradle, and Ant. The project used a lightweight Docker virtual environment to automate the builds of 7,200 projects selected from a random sample of 10,000 projects on GitHub based on certain criteria; namely, that the project was open source, had at least one fork, and was written in Java. (At the time of the study, there were about 2,000,000 such java projects on GitHub.) They found that the most popular tool was Maven; ~53% of the sample was built using Maven, with Gradle occupying ~11%, Ant ~ 8%, and another ~27% of projects where there was no build tool detected. Of the projects with one of the three recognized tools, about 40% of the sampled builds failed. This means that in 4 out of 10 cases, the programmer would not be able to produce a target archive from the source code without manual intervention.

The paper went on to discuss the reasons for the failed builds within each build platform. They found that dependency resolution accounted for ~39% of failed builds within Maven. Reasons for this include invalid POM files, authorization errors, or missing files on servers due to reasons like discontinued dependency hosting or removed older versions. With Gradle and Ant, errors were most frequently compilation-related, which may be related to missing dependencies. The third most prevalent reason for build failure across the board is, surprisingly, document generation.

The researchers found that the older a project was and the more time that has elapsed since its last update increased the tendency that a build would fail. Also, there was a strong correlation between the size of a project and the likelihood of a build failure; projects with a successful build had a median of 70 source files, while this number was 101 for the failing ones. Maven builds had the highest rate of success at 65.8%, while Ant (the oldest of the respective tools) failed the most often at 44.4%. Gradle was in the middle at a little more than a 50% rate of success. The study is insightful not just in that it revealed that the largest number of build errors are dependency-related, but that the size of the project, time since last update, and choice of build tool are all statistically indicative of the likelihood of a successfully automated build. These are all points of consideration for the software developer, who would rather not spend time debugging build errors.

Weaknesses:

This study did not extend its sample to projects that included Android libraries. This may have skewed the results out of Gradle's favor, as many of Gradle's features were designed with Android development in mind. Unit tests were excluded from the Gradle and Maven builds, but there is no easy way to exclude them from Ant build scripts, so more dependencies failed in Ant because of this. The paper also acknowledged that while the analyzed the relative severity of build failures, they did not provide any guidance about how to avoid them.

Strengths:

This study's approach to analyzing the relative build success revealed statistical correlations that are suggestive of certain trends among Java build processes. Some of these, such that smaller projects are often more successful, seem intuitive. It seems that Maven is the most popular tool and the most likely to result in a successful build. Another strength of the study is that it is fully reproducible; the Docker environment, tools, and build script the researchers used are made available.

Future Direction:

It would be interesting to reproduce this study now, five years after its publication in 2016. I would be curious to see if the number of Gradle projects and successful Gradle builds would have increased, as Gradle has been around for longer now. I would also be interested in creating a similar project that had the capability to create a list of recommendations to follow when a build fails. I am now also thinking of other data that could be pulled from open-source GitHub projects using similar methods to this paper.

Comparison:

My research did not dive deep into analyzing the relative efficacy of Maven vs. the other two Java build tools, but it seems from this paper that Maven seems to have the advantage. This explains why I found in my research that Maven has about 60% of the market share of Java projects. It seems that Maven's conventions provide a stable groundwork for reproducible, automated builds, as long as the developers have maintained and update the dependencies accordingly. Of course, my research did not involve the massive quantity of projects analyzed by this one.

Sulír, Matúš, and Jaroslav Porubän. "A quantitative study of java software buildability." *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 2016.
https://arxiv.org/pdf/1712.01024

Title: An empirical study of unspecified dependencies in make-based build systems

Summary:

These researchers looked for the occurrence of unspecified dependencies in four large-open-source projects. An unspecified dependency is a dependency used in the build process but not explicitly defined in the build file. The interest in locating unspecified dependencies lies in that when changes are made to the source code of an unspecified dependency, code that relies on that dependency will not be rebuilt in incremental builds, and thus remains outdated. This can cause bugs in a build process. The researchers generated two dependency charts for each build process they analyzed: a conceptual one and a concrete one. The conceptual graph contains the targets and files explicitly defined in the build system, while the concrete graph contains the set of processes and files that are exercised by the build process. By comparing the discrepancies between the charts, they were able to find over 1.2 million unspecified dependencies in the four analyzed projects. They boil down their discussion into six primary root causes for the unspecified dependencies.

Weaknesses:

While their method was extremely successful at finding unspecified dependencies (after all, 1.2 million seems like an awfully large number) only 36 of these dependencies contributed to actual bugs. Many unspecified dependencies were resolved by customizations in the build process itself that the developers would manually add to a project. Most of these unspecified dependencies were condoned by the development teams. It turns out that in large part, developers were aware of the unspecified dependences and opted to leave them as such. The point of this is to avoid large, time-consuming incremental builds.

For example, the researchers addressed the fact that one of the root causes of an unspecified dependency is a missing generator dependency. When code generators are used to create code, targets may depend on the generated code but not on the generator itself, and therefore would be missing the generator dependency. However, these do not necessarily cause bugs in practice, because the generator code is often part of a compressed archive that was extracted during the build. So, it is unlikely that the generator code is ever modified and executed without recompilation.

Another limitation of the paper might be seen in the fact that the build systems in particular were all make-based. GNU'S make is one of the oldest build systems and the projects that were analyzed in question are longstanding projects. The implication of this is that developers in these projects are mature enough to manually configure the essential parts of the build and as such avoid running into problems related to unspecified dependencies.

Strengths:

Discussing strengths, I thought that the generation of the concrete vs. conceptual graph was a really revealing comparison in terms of understanding the complete picture of a build process. That they were able to produce and resolve two different schemas of a build process, with the conceptual being the overarching layer actualized by the steps described in the concrete graph, is really a brilliant way to bring the unspecified dependencies to light. I am sure this kind of data extraction could be useful to developers using other build tools, if for nothing more than to be able to see two accurate representations of the build for debugging purposes.

Their analysis of the root causes was what I thought was the most interesting part of the paper. Missing meta-files seem to be the root cause of greatest concern, and one that other build tools seek to address. A meta-file contains meta-data about a project or library, such as the data that is necessary to use a library on different platforms. Missing a meta-file can lead to a project using an outdated version of a library, and this type of unspecified dependency is known to cause bugs. 12 of the 36 significant bugs found by the researchers were caused by this type of missing meta-data. This strengthens the case to be made that it is important to search for unspecified dependencies of this kind in projects.

Future Direction:

It seems to me the paper could have been improved were there to be some way of filtering out missing dependencies that could be deemed as harmless or otherwise irrelevant. A future direction might be to focus the tools on finding unspecified dependencies within the particular contexts in which they are most likely to cause a build failure.

The qualitative analysis of a particular build is of great interest to me. It seems to me that there would be multiple ways of building a large project and finding the optimal build would be of great interest to developers. Tools like the ones this paper presented present a black-and-white view of unspecified dependencies; however, the way the tool analyzes projects is very well thought out. It seems that similar methodology could be used to create tools that do more than just scan for missing dependencies, but also could be intelligent enough to point out inefficient builds or suggest a more streamlined configuration.

Also, expanding the analysis beyond just make-based build tools would be interesting. Other build tools have features that are meant to avoid such missing dependencies, and a comparison of the utility of these features may be of interest to developers working on large projects. I would be curious to see a similar study done on Maven builds. I wonder what the number of unspecified dependencies in large projects that use Maven might be relative to these Make-based projects, and how many significant bugs would arise from these.

Comparison:

Besides the obvious contrast that Maven is oriented toward Java and the make-based tools considered in the paper are geared toward C and C++, Maven's convention-over-configuration approach and use of versioning semantics within the POM file are features of Maven that help avoid missing dependencies or incomplete meta-data. That was part of the point behind Maven's creation, to establish a convention in the build process. Before tools like Maven, developers would create their own custom build process for each project and the predictable result is that a lot of chaos and complexity ensued.

It seems that tools like Maven are designed with these problems in mind. I am interested in how tools like Maven simplify the build process of large projects and how they can be used to support practices such as test-driven development and continuous integration. It seems that in large projects, a non-trivial amount of time is spent managing and keeping dependencies up-to-date, along with ensuring that any modification to code does not break the build. Also, building a large project completely every time source code is changed would be too time-consuming, which creates the need to facilitate incremental builds. Maven's features are designed to help with these processes. Especially what comes to my mind is Maven's implementation of transitive dependencies and the ability of POM's to inherit from other POM's. It really comes down to the centralization of the important build information and metadata within the POM file that gives Maven the advantage to avoid the root causes of unspecified dependencies discussed within the paper.

In conclusion, my research involves dependency management, and these researchers crafted an intelligent tool to probe the efficacy of the specific aspect of dependency management known as unspecified dependencies. The paper illuminates the subtler aspects of building large projects that tools like Maven were designed to handle. In the discussion of dependency management, there is a tension between convention and flexibility. That is why tools like Gradle were developed—to handle tricky configurations that would break Maven's conventions. This all suggests that software developers would be keen to keep in mind the needs of their specific project when choosing a build tool and deciding on how to implement a build process.

Bezemer, Cor-Paul, et al. "An empirical study of unspecified dependencies in make-based build systems." *Empirical Software Engineering* 22.6 (2017): 3117-3148.

https://www.researchgate.net/profile/Ahmed_E_Hassan/publication/315918566_An_empirical_study_of_unspecified_dependencies_in_make-based_build_systems/links/5b7f20a292851c1e122e58f7/An-empirical-study-of-unspecified-dependencies-in-make-based-build-systems.pdf

Title: Analysis of Modern Release Engineering Topics

Summary:

These researchers developed a method to comparatively analyze a large sample of questions on Stack Overflow in terms of popularity and difficulty. The study focused specifically on release engineering topics such as continuous delivery and deployment, continuous integrations, source control management, testing, and cloud provisioning. The goal of the study was to find areas where more research, training, and investments may be warranted. The study also examined what types of questions were most prevalent, in terms of *how, what,* or *why*.

The methodology involved downloading a set of StackOverflow posts which contained specific tags related to the release engineering process. In total the study examined 260,023 posts, with 71% being (184,830) being questions and the rest 29% (75,193) being answers. The assessed the relative difficulty and popularity of the topics and found that in general, the more difficult a topic, the less popular it was. There were exceptions to this in the fields of big data topics, security, and mobile development, seen as both difficult and popular. They found that the most popular topics by category included continuous integration and deployment (22%) followed by build systems (20%). With in the CI/CD category, software testing was the most prevalent source of questions; within build systems, build failure represented the dominant trend of questions asked. They also found that “How” related questions occupied the majority of question types. This points out potentially fields of interest for researchers, practitioners, educators and students.

Weaknesses:

A possible limitation of the study has to do with how the tags were parsed to choose individual questions. Its possible that a different set of tags could have produced different results. However, the authors took steps to mitigate the possibility that choice of tags would have skewed the data they were looking for. Probabilistically, they choose a statistically relevant portion of the data, so it seems that all in all their findings are pretty solid.

Strengths:

The researchers did a good job of selecting and extracting the data from StackOverflow. Using their pool of data, they found a lot of evidence to support the fact that topics involving continuous integration and deployment stymy developers in a wide range of situations. Build systems and testing remain a source of unanswered questions. They were able to show a negative correlation between the difficulty of a topic and its popularity in most cases, pointing out that topics which were considered both difficult and popular represent the current fields of interest for developers.

Future Direction:

In addition to finding the relative proportions of the topics, it would have been interesting to have addressed the actual problems causing most of the questions. I would be curious to see what specific problems dominated questions in a particular category, and how many problems were seen as open-ended.

I would be interested in developing tools like this that parse StackOverflow for information. It seems like it would be useful to automate a scraper to collect information and possible notify the user when longstanding questions have recently been answered satisfactorily.

Comparison:

My research analyzed how Maven can speed up a development process, which is a specific topic within the larger domain of build systems. This study pointed out that build failures are a statistically relevant portion of questions for users of StackOverflow, suggesting that understanding and optimizing the build process is an important skill for developers. This might also suggest that more effort is needed to improve existing build systems as many still struggle to find working solutions.

Openja, Moses, Bram Adams, and Foutse Khomh. "Analysis of Modern Release Engineering Topics:–A Large-Scale Study using StackOverflow–." *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020.

https://www.researchgate.net/profile/Moses-Openja/publication/346582386_Analysis_of_Modern_Release_Engineering_Topics_-_A_Large-Scale_Study_using_StackOverflow_-/links/604cb816299bf13c4f0488dc/Analysis-of-Modern-Release-Engineering-Topics-A-Large-Scale-Study-using-StackOverflow.pdf

Title: Root Cause Localization for Unreproducible Builds via Causality Analysis over System Call Tracing

Summary:

These researchers created a unique tool to perform causality analysis of failed builds via system call tracing. Unlike other studies which seek to mitigate build failures by examining the build tool in question, these researchers looked at the system call traces of the processes spawned from the executed build commands. The advantages to system call tracing include the ability to more accurately capture the information behind the build process and the ability to be used in different build environments. The tool builds and compares dependency graphs that are used to find inconsistencies in build execution by noting discrepancies between reproduced builds. More specifically, the tool identified the *write* system calls that output different data between different builds and traced how these differences propagated to different artifacts. The researchers used the tool on 180 real-world packages from the Debian repository and found that it was 90% effective in identifying the root causes for unreproducible builds.

Weaknesses:

The study worked with a relatively small sample size. The researchers' justification for the small set of Debian packages was that the evolution of the Debian repository resulted in old packages which could not be built from source, which would expose the kind of error tracing that their custom application intended to analyze.

Strengths:

The researchers used differential analysis to limit the massive amount of system call information down to the relevant traces, a process that I have to admit I don't fully understand. However, I recognize the strength of this approach. There have been recent successes in using system call tracing in monitoring commands for various research fields such as intrusion detection, computational reproducibility, and system profiling, although it seems that few studies have applied the technique to build processes. An advantage of system call tracing is the potential to identify the specific build commands that are responsible for the unreproducibility issues. This can give a more in-depth picture of a build failure than mere dependency analysis.

Future Direction:

I would be really interested in making a similar tool to evaluate system call traces for purposes other than build reproducibility. It would be interested to see if this technique could be used to help developers solve problems related to continuous integration and deployment. It seems that being able to locate the system calls responsible for errors would save time in debugging a complex problem.

Comparison:

The reason I studied Maven and dependency management is because I am looking ahead to the point in my evolution as a programmer when I will need these tools to effectively develop larger and more intricate projects. Tools such as Maven have been around for awhile and are well-equipped to handle most situations; however, the utilities of Maven alone are not always enough to repair a build when it fails. The more complicated a project, the more likely the build will fail, and at this point having an effective tool such as the one described in this paper to locate the reason for a build failure seems like it would be useful. I wonder if in the future build tools like Maven will incorporate features of system call tracing to add features to their repertoire.

Ren, Zhilei, et al. "Root cause localization for unreproducible builds via causality analysis over system call tracing." *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

https://par.nsf.gov/servlets/purl/10190934

Title: Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines

Summary:

This article describes the problems associated with security testing in development scenarios involved with continuous integration and continuous delivery. Classical security management techniques are not equipped to keep up with the fast-paced software development lifecycle demanded by today's industry. While fast releases are considered in general to result increased product quality, tight schedules or a tough workload increases the chance that vulnerabilities will be introduced into a system. This paper examined the challenges of dynamically testing security and how to go about properly automate this task. They used Docker containers to create a virtual framework in order to perform tests to determine how long it took for running applications to respond to certain threats. They combined three techniques to achieve this goal: Web application security testing, which automates attacking a web application through its user interface, Security API scanning, which is meant to detect flaws of back-end services, and Behavior Driven security testing, which automate the execution of an attack from the hacker's perspective. All of these were used to probe for vulnerabilities in WebGoat, an application designed for security testing purposes.

Weaknesses:

It seems to me that this paper's analysis of the securities flaws within WebGoat may not be representative of security challenges faced more broadly within the industry. Unlike other studies that examined a large data set, it seems that these researchers draw their inferences from the one case study. Although their experiment was successful at detecting vulnerabilities, this result may not be generalizable.

Strengths:

The researchers described overcoming specific challenges inherent in testing security for CI/CD pipelines which many development teams face and propose solutions based on their findings, which include parallelizing different types of testing to reduce runtime and applying Test-Driven-Development Techniques for API design. They exposed some of the inherent complexities of this process and argued that this topic is increasingly becoming relevant in modern development teams.

Future Direction:

I would like to emulate the use of Docker containers to provide a means to automate these subtle aspects of security testing. It seems to me that as continuous integration and deployment come into greater use across the industry it is the responsibility of the developer to be aware of potential security pitfalls that could arise in this context. Future research could explore automated ways to patch detected vulnerabilities, or ways to automatically generate test should the application's behavior changes.

Comparison:

While continuous integration and deployment are not directly related to my research, build tools such as Maven are often used to help facilitate such practices. The security question is one that I have left almost entirely out of my research, but is obviously very important to real-world development. I enjoyed reading this study as it made me think ahead to the point where I might be on a development team and I would need to solve a problem assessing security risks involved in continuous deployment.

Rangnau, Thorsten, et al. "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines." *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2020.

https://www.researchgate.net/profile/Fatih-Turkmen-2/publication/346379276_Continuous_Security_Testing_A_Case_Study_on_Integrating_Dynamic_Security_Testing_Tools_in_CICD_Pipelines/links/6024f235a6fdcc37a81a92be/Continuous-Security-Testing-A-Case-Study-on-Integrating-Dynamic-Security-Testing-Tools-in-CI-CD-Pipelines.pdf