Catching Up with State of the Art Continuous Integration Pipelines in Palladio — An Experience Report

Stephan Seifermann
stephan.seifermann@kit.edu
Karlsruhe Institute of Technology

Sebastian Krach
krach@fzi.de
FZI Research Center for Information Technology

Abstract

Palladio is a fairly large research project providing various software artifacts. The large amount of maintained projects makes Continuous Integration (CI) vital. However, CI is more useful if the source of a detected problem becomes clear. The earlier CI infrastructure did often not allow tracing back problems and even made adding new projects challenging. In 2018, we decided to completely rebuild the whole CI infrastructure and the organization of source code to catch up with the state of the art. Two years later, we can now report on our experience in migrating such large projects as well as on the benefits of spending the effort in this migration.

1 Introduction

Palladio [2] is still a growing research project despite its age. Software architects use the Palladio Bench to predict the performance and other quality properties of their envisioned architectures using a decent set of various Eclipse extensions. These extensions trace back to many smaller, individual projects maintained by researchers at different universities. Continuous Integration (CI) is one building block to monitor all these projects for problems and report them.

The CI infrastructure used till 2018 struggled with providing useful feedback to developers. It could successfully identify compilation issues and missing dependencies but reported them in a incomprehensible way. Setting up the build environment locally to reproduce the issue was not feasible either. Additionally, creating the build management descriptions required a deep understanding of the build technology, which hindered the integration of new projects into CI. The deprecation of the used build management technology Buckminster was the last straw before we decided to change our CI process.

We collected the issues of the existing CI infrastructure and decided to spend considerable effort in catching up with the state of the art: We migrated all shipped source code from SVN to Github to increase visibility and ease code contributions. We switched to the build management tool Maven Tycho, which eases reproducing issues locally and provides clearer error messages. The build descriptions became part of the repository itself and we reduced them to a bare minimum to ease setting up builds for new projects.

As of now, we used the new infrastructure for about two years. CI works as expected and we successfully managed to do two Palladio releases. In this paper, we report on this new infrastructure and our experiences in migration and daily usage.

2 Requirements for Infrastructure

We collected requirements for the new CI infrastructure to define an appropriate solution. The requirements mostly stem from shortcomings of the previous CI infrastructure.

R1) Comprehensible Error Messages: The CI should provide all details required to identify a problem in reported error messages.
R2) Locally Reproducible Builds: Developers should be able to reproduce build results and predict them by executing the build instructions locally.
R3) Support External Contributions: The Palladio source code was located in a semi-public SVN repository that required a local account on the SVN server. The new infrastructure should allow external users to suggest code contributions.
R4) Validation of External Contributions: Before spending effort in reviewing external contributions, the CI should ensure that the contribution compiles and passes defined tests.
R5) Autonomous Build Definitions: The integration of builds into CI should no longer require administrative but only regular developer permissions.
R6) Resilience to Eclipse Downtimes: Builds should not be affected by downtimes of the Eclipse update sites, which happened often in the past.

We do not mention obvious requirements such as no use of deprecated technologies or the ability to build all artifacts required for Palladio.

3 Overview on CI Infrastructure

In this section, we report on how our new CI infrastructure meets the requirements of the previous sections. An overview on our infrastructure is given in Figure 1. We explain the components and their relation in the following.

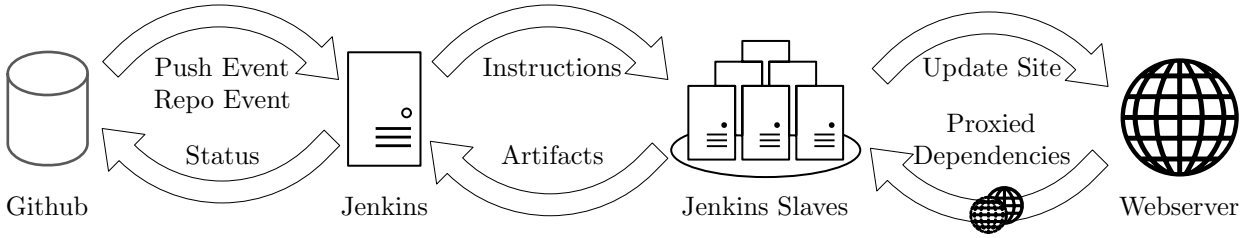

Figure 1: Overview on CI components and exchanged information.

To improve error messages (R1) and allow local builds (R2), we switch from the build management tool Buckminster[1] to Maven Tycho[2]. Developers now only have to have a recent Maven version to build a project. To ease build definitions, we introduce a parent POM hosted at Maven Central that contains all necessary build descriptions. It is only necessary to specify dependencies of the project in a target platform. Remaining tasks like building Xtend code, source features and update sites are already configured. By using POM-less builds[3], there is almost no overhead for specifying the build compared to pure development inside of Eclipse.

To support external code contributions (R3), we migrated all Palladio projects from our institutional SVN repository to Github. The challenge was to not only migrate the code but also branches, tags and the history. Especially, old projects like the metamodel have a huge history with many authors. We created a python script to automate as many tasks as possible. The source code is now visible on Github and external users can submit code contributions by pull requests.

To verify external contributions (R4) as well as internal contributions, we use a Jenkins extension that detects and registers hooks in Github repositories of our Github organization. We chose Jenkins over other CI solutions because we are already competent with it. The hooks notify the build server hosted at our institute about changes and triggers a build. The actual build is dispatched to our build slaves hosted at our institute and in bwCloud[4] to speed up parallel builds. The execution of the Maven-based build takes place in a Docker container to isolate build jobs on the same slave. The idea of using Docker containers for build tasks has already been presented earlier by Düllmann [3] as well as by Reed et al. [4]. If the build succeeds, Jenkins publishes the build result to Github. We configured all master branches to not accept a push. Instead, all contributions require pull requests and therefore are checked before merging into master.

To ease defining build jobs (R5), we use predefined build pipelines that developers only have to parameterize. The pipeline covers execution of Maven build jobs, collection of quality metrics and deployment of the produced update site to our web server. There are sensible defaults, so the only information required to integrate a new build job is providing the location of the artifacts to deploy. Developers create a *Jenkinsfile* containing this information in the repository and the build server automatically creates the build job. This also considers pull requests and branches but only the master branch is deployed by default.

To improve resilience to downtimes of Eclipse update sites (R6), we proxy requests and redirect them to the more reliable mirror of the University of Erlangen–Nuremberg. We inject the proxy configuration via Maven settings on the build server. In contrast to hosting a local mirror of the Eclipse update sites, this approach saves storage space but still reduces issues because of unavailable update sites. Additionally, the URLs in the target platforms remain the same and local builds still work as expected.

4 Reusable Building Blocks

When realizing the previously described infrastructure, we had to work around limitations of existing tools that were hindering in our context. Therefore, we created several tools that can be reused in other contexts than Palladio.

Target Platform Mechanisms of Tycho were not as modular as required, i.e. composing a target platform of multiple target platform definitions was not possible. Additionally, fixed version numbers are required, which are difficult to define if we use rapidly changing nightly builds. Therefore, we developed a Maven extension[5] that merges multiple target platforms into one, updates version numbers to the latest available and provides filters to switch between nightly and release dependencies. The benefit is that developers only have to define specific dependencies of their projects rather than dependencies to Eclipse. Instead, the extension can merge one of our predefined target platforms for various Eclipse versions[6]. We favor the use of dedicated target platforms to only specifying repositories with automated artifact resolution because we plan to realize reproducible release builds with fixed artifact versions.

[1] `https://eclip.se/h5`
[2] `https://eclip.se/h4`
[3] `https://eclip.se/h3`
[4] `https://www.bw-cloud.org`
[5] `https://git.io/JUmRa`
[6] `https://git.io/JUmRj`

Ecore Code Generation is essential to our projects because many define Ecore meta-models and rely on generated source code for handling model instances. The original EMF workflow supports customizations on a method-level granularity by replacing and annotating parts of the generated source code. For large meta-models, maintaining these customizations quickly becomes cumbersome. The *Generation Gap* pattern [1] describes a solution by separating custom from generated code through subclassing. Initial support for this pattern is provided through the Ecore code generation component of the Eclipse Modeling Workflow Engine (MWE2)[7]. We provide comprehensive support for working with the generation gap as part of a MWE library[8]. We ensure transparent behaviour of depending code by renaming the generated parent class while reserving the meta-model class name for the custom subclass. Any reference, including generated factories, thereby correctly references the customized class. We further integrated the workflow execution in a *convention-over-configuration* manner as part of the shared parent POM. If the developer provides suitable modeling workflow descriptions at a predefined location, the respective workflows are executed at the appropriate phase of the maven build. Consequently, we engage developers to only commit their code customizations, allowing our CI to ensure the consistency between meta-model and the generated code artifacts.

Build Pipelines predefined by us are suitable to most of our projects but there are some which have different requirements, e. g. use Gradle instead of Maven. In order to minimize repetitive Jenkins pipeline definitions, we opted for a modular approach based on the Jenkins Modular Pipeline Library (MPL)[9]. MPL allows us to define the pipeline as sequence of abstract stages and provide customizations for each stage as separate module. The modules provide support for dynamic composition, thereby giving us means of fine granular extensibility. For instance, the composition allows us to reuse the same module for setting up the build environment in isolated docker containers, and exchange the module calling the build tool inside the container. We implemented an extension[10] to the Groovy-based Jenkinsfile DSL to provide an intuitive way of selecting and configuring the required extensions to the default pipeline.

5 Experience Report

We have been using the presented CI infrastructure for about two years. The overall experience is good. Especially, reproducing build errors locally or even being able to test the outcome of the build process before a commit is beneficial and improves confidence of experienced and novice developers. Novice developers also start defining builds for their projects, which has rarely happened before. The review process of contributions via pull requests and the visualization of build results of these pull requests was an enabler for involving more and new people in the development. Experienced developers benefit from easier reviews while novice developers get feedback to their contributions.

However, there is still a need for dedicated CI experts that are responsible for monitoring and maintaining the infrastructure. The effort of these experts usually boils down to installing updates and providing new target platforms and reusable Maven descriptions in case of changes of the Java or Eclipse version. Giving new developers a short introduction is another important task because there is still a lack of satisfying documentation of these tools. We certainly have to improve in this field.

In summary, we believe that our effort was well spent. However, the effort was not neglectable. The migration from SVN to Github and from Buckminster to Maven took a student researcher about half a year. Our previous strategy of migrating projects once they change did not work at all. Such migrations require dedicated persons being in charge of the process.

6 Conclusion

In this paper, we presented the CI infrastructure of Palladio how it improved the overall development process. The use of state of the art technology in the CI pipeline enables fast feedback cycles, code reviews and comprehensible build results. Adding a project to CI is now possible without deep understanding of the build process and without administrative privileges.

Practitioners can benefit from the reusable building blocks of our CI pipeline. Other research projects can consider our experiences when reasoning about spending effort in their CI infrastructure.

As part of future work, we plan to improve the documentation of all tools and the overall process. We also would like to define a concept for creating reproducible release builds based on target platforms.

References

[1] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[2] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, 2016. 408 pp.

[3] T. F. Düllmann. *Don't just watch the containers pass by: How we (plan to) use Docker to streamline the Kieker development process and infrastructure*. Presentation at SSP'17. 2017.

[4] N. Reed et al. *Automating the Build Pipeline for Docker Containers*. Presentation at SSP'17. 2017.

[7] `https://eclip.se/h6`
[8] `https://git.io/JU3bX`
[9] `https://git.io/JU3At`
[10] `https://git.io/JU3xo`