The Sisyphus Continuous Integration System

Tijs van der Storm
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
storm@cwi.nl

February 7, 2007

Abstract

Integration hell is a prime example of software evolution gone out of control. The Sisyphus continuous integration system is designed to prevent this situation in the context of component-based software configuration management. We show how incremental and backtracking techniques are applied to strike a balance between maximal feedback and being up-to-the-minute, and how these techniques enable automation of release and delivery.

1. Introduction

Continuous integration is a practice to keep software evolution in control [1]. As soon as changes on a code line become available, they are integrated to the mainline. The complete product is built, tested and the results are published so that the global status of the software product is known at all times. Continuous integration has therefore been called the "heartbeat" of software development. If it stops beating, you cannot ship.

Sisyphus is a continuous integration system targeted at component-based software configuration management settings in order to reduce the risk of integration and to automate release and delivery [3]. Currently, it is implemented as a prototype to build the components of the ASF+SDF Meta-Environment [2] in a continuous fashion. The current status of its builds can be viewed live at:

`http://sisyphus.sen.cwi.nl:8080`.

2. Innovations

The Sisyphus continuous integration system applies a number of innovations to maximize the value of continuous integration by exploiting the fact that sources are divided over separate source components. They are summarized as follows :

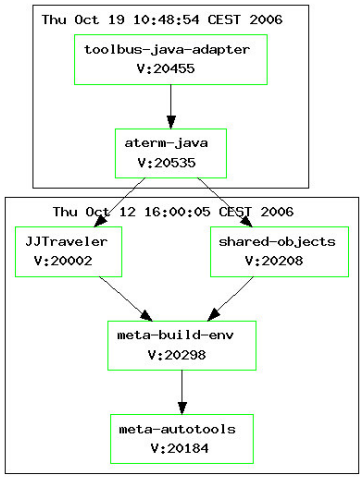

Figure 1. Build sharing among build iterations

- *Components are first class.* Components are represented by independently versioned sets of sources that may reside in different repositories. Each component may declare build-time dependencies on other components. Figure 1 shows six source components and their dependencies. Nodes represent builds of a certain revision (the number in the label) of a component. The edges indicate a *use* relation.

- *Performing a build only when needed.* A source component will only be built if there are affecting changes, which are changes to the sources of the component itself or changes to one or more of its dependencies. In the figure, build iterations are visualized by clusters of nodes labeled by a timestamp. The example shows that the `aterm-java` build of October 19th uses builds from a week earlier; apparently no changes on these components did occur since then so build artifacts can be reused.

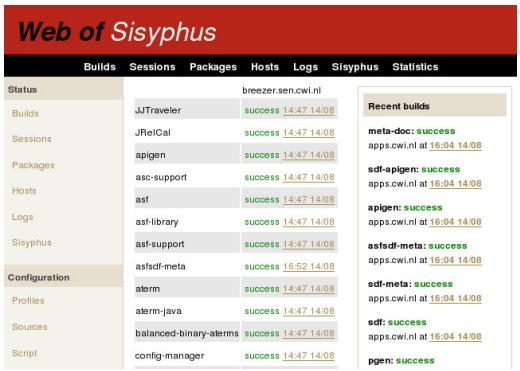

Figure 2. Build, release and delivery portal

- *Any build is better than no build.* If the latest build of a component has failed, but is still required in another build, an earlier successful build is used instead. This way of backtracking upon failure trades some up-to-dateness for increased feedback. Without backtracking, it would not make any sense to build a component if any of the builds of its dependencies have failed.

Due to the incremental nature of integration, no time is wasted building components that have not changed in between integrations and feedback is generated more quickly. Similarly, the backtracking technique ensures that there always will be feedback, even if dependency builds have failed. More feedback, more often, can help reduce the risks imposed by integration.

3. Release and Delivery

The Sisyphus system differs from many other continuous integration systems in that it maintains a knowledge-base that keeps track of which version of which component has been built in what project against which dependencies with what result. Build results – success or failure – of individual components are related to the exact revision number of the corresponding sources. This revision number, however, does not capture which versions of dependencies were used in a certain build. This information is stored separately in the knowledge-base so that every build identifies a certain composition.

Every component that has been built successfully represents a (theoretical) release candidate. Formally releasing a product means preparing the software product so that it can be delivered for installation to users. Often an informative version number is assigned so that users can interpret the changes with respect to previous versions. Using the software knowledge-base populated by Sisyphus, release boils down to selecting the desired build and providing an informative version number. The system will ensure that releases are internally consistent, that dependency links are properly versioned and that no build is released more than once.

Since builds identify compositions, it is known which sources have been used in which builds. This facilitates the derivation of source-based distributions that are commonly used in open source projects. Moreover, since build sharing and backtracking require that all build artifacts are stored persistently, any binary distribution can be accurately and efficiently reproduced, and incremental updates can be computed for efficient delivery [4].

The Sisyphus front-end consists of a build, release and delivery portal. See Figure 2 for a screen-shot. This web application displays the current build status of every component. For every build, detailed information is available about build actions, used dependencies and revisions. In addition there are links to distributions (source and binary) and visualizations of the bills of materials (see Figure 1 for an example).

4. Relevance for Software Evolution

Integration hell is a direct consequence of software evolution. It has been said that the effort of integration is exponentially proportional to the amount of time you postpone it. The longer you wait, the more chance there is that changes conflict, either at build-time or at runtime. There is an increased risk of introducing integration bugs and they may be very hard to track down. Being able to integrate as soon as possible is therefore a *sine qua non* to keep software evolution in control.

Furthermore, users see the result of evolution in the form of updates, and at the same time fuel software evolution by the feedback they provide (e.g. bug reports). To shorten the feedback loop between development and user, it is essential to be able to deliver new versions quickly, and to maintain traceability between sources and installed versions of a product. The Sisyphus system accomplishes these two requirements in a setting of component-based development.

References

[1] M. Fowler and M. Foemmel. Continuous integration. Online: `http://martinfowler.com/articles/continuousIntegration.html`.

[2] M. van den Brand et. al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[3] T. van der Storm. Continuous release and upgrade of component-based software. In J. Whitehead and A. P. Dahlqvist, editors, *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, 2005.

[4] T. van der Storm. Lightweight incremental application upgrade. Technical Report SEN-R0604, Centrum voor Wiskunde en Informatica (CWI), 2006.