Open Sourcing and Release Engineering @ Eclipse.org

The Making of an Eclipse Project

Was steckt eigentlich hinter einem Eclipse-Projekt? Welche Entscheidungen sind zu treffen, welche Bedingungen zu erfüllen, wie läuft das alles? Das Eclipse-Jubula-Team berichtet in loser Folge über seine Erfahrung beim Open Sourcing von Jubula [1]. Dabei geht es nicht nur um Technik, sondern auch um Strategien, Abläufe und schwierige Entscheidungen.

von Markus Tiede

Der letzte Teil dieser Kolumne zum Thema „Jubulas Erfahrungen auf dem Weg in die Open-Source-Welt“ soll konkrete Schritte aufzeigen, die nötig gewesen sind, um ein Projekt unter den Fittichen der Eclipse Foundation zu veröffentlichen. Der Ausdruck „veröffentlichen“ ist dabei in mehrfacher Hinsicht zu verstehen: angefangen vom simplen Bereitstellen des Quellcodes für die Allgemeinheit, über das Aufsetzen eines Continuous Builds, bis hin zur Erzeugung von Eclipse-konformen Artefakten und der Teilname am so genannten „Release Train“. In den drei bereits erschienenen Folgen dieser Reihe haben wir schon Themen wie den „ersten Einstieg“, den rechtlich notwendigen „Intellectual Property (IP)“-Prozess der Eclipse Foundation und den Einsatz von EclipseLink als O/R Mapper beleuchtet. Dieser Teil fokussiert daher weniger auf das rechtliche „Drumherum“, vielmehr soll er konkrete Arbeitsschritte und Verantwortlichkeiten aufzeigen, die einen Entwickler erwarten, wenn er sich auf das Abenteuer „Eclipse-Projekt“ einlässt.

Open your Source

Was sich wie eine Selbstverständlichkeit anhört, nämlich dass ein Open-Source-Projekt öffentlich zugängliche Quellen besitzt, war für uns anfänglich mit einem nicht zu unterschätzenden Aufwand verbunden. Denn nur zu leicht lässt man sich dazu verleiten, den eigentlichen Schritt, seinen Quellcode zu veröffentlichen, auf den einmaligen „Commit“ desselbigen zu reduzieren. Tatsächlich war aber einiges an Vorarbeit zu leisten. An dieser Stelle möchten wir aber schon klarstellen, dass dies längst nicht auf jedes neue Projekt zutreffen muss. Jubula stellt insofern eine Sondersituation dar, als dass es nicht wirklich als neu einzustufen ist. Wir haben uns gegen Ende 2010 dazu entschieden, weite Teile des jahrelang schon bestehenden kommerziellen Produkts GUIdancers unter dem Projekt Jubula zu veröffentlichen. Somit gab es bereits eine massiv über die Jahre gewachsene Codebasis (> 350k LoC), die für eine Veröffentlichung zuerst vorbereitet werden musste.

Neben dem notwendigen Papierkram, um alle rechtlichen Vorgaben zu erfüllen (Details dazu gab es bereits in Ausgabe 4 und 5.2011 des Eclipse Magazins), stand am Anfang dieser Vorbereitung eine hochsensible, wenn auch teilweise stumpfsinnige Aufgabe. Dies war die korrekte namentliche Abgrenzung der Teile, die der Open-Source-Welt zugeführt werden sollten – auf gut Deutsch eine Umbenennung. Dieser Schritt war von unterschiedlicher Motivation getrieben:

Eclipse Release Train

Unter dem Eclipse Release Train versteht man die Zusammenarbeit aller teilnehmenden Projekte auf den einen Tag hin, den vierten Mittwoch im Juni eines jeden Jahres, an dem die neue Eclipse-„Hauptversion“ erscheint. Da für diese Zusammenarbeit häufig eine treibende externe Kraft von Nöten ist, die dafür sorgt, dass alle Projekte in der Spur bleiben, liegt die Analogie zum Betrieb eines Zuges sehr nah.
„Toi, toi, toi“ – 2011 ist das achte Jahr in Folge, in dem die Eclipse Foundation mit tagesgenauer Pünktlichkeit die Freigabe des neuen Release verkünden konnte.

- Zuallererst sollte es eine klare Abgrenzung zum bereits bestehenden Produkt darstellen, um Verwechslungen vorzubeugen und eine eindeutige Zuordnung zu gewährleisten.
- Ein weiteres Ziel war die Schaffung einer einheitlichen Terminologie für interne und potenzielle externe Committer und Contributer. Da das Projekt bereits jahrelang gewachsen war, und sich dabei sukzessiv bestimmte, zum Teil auch unterschiedliche und mehrdeutige, firmeninterne Terminologien entwickelt hatten, war es zwingend erforderlich diese zu vereinheitlichen, um eine gemeinsame sprachliche Basis zu schaffen.
- Nicht zuletzt hatte diese Umbenennung auch einen rechtlichen Hintergrund: Jedes Eclipse zugehörige Projekt muss in den vielfältigen Qualifiern, die es gerade im RCP-Umfeld gibt, das Präfix „org.eclipse.*“ tragen. Angefangen von Java-Paketnamen, über OSGi-Bundle-Namen, Extension-(Point-)IDs bis hin zu Feature- und Produktbezeichnungen.

Heutige IDEs bieten vielfältige Möglichkeiten für ein solches Refactoring und unterstützen den Entwickler erheblich bei der Durchführung solcher mitunter sehr weit reichender Änderungen. Gesagt, getan – etwa eine Mannwoche später war die Umbenennung der etwa 70 Bundles und deren Inhalt vollbracht. Echte Stolpersteine gab es nur in Bereichen, in denen die Änderungen erst zur Laufzeit sichtbar wurden, wie beispielsweise bei Verwendung des Java Reflection API zum dynamischen Laden und Instanziieren von Klassen.

Alle Dateien zusammen wurden dann als ein einziges ZIP als Anhang an einen IPZilla Bug gehängt und in Form eines CQ (Contribution Questionnaire) an den Eclipse IP Check übergeben. Nach kurzer, eingehender, automatisierter Prüfung befanden wir uns in der so genannten Inkubationsphase. Jetzt war es uns bereits erlaubt, den Quellcode und was sonst noch so dazu gehört der Allgemeinheit in einem öffentlichen Versionsverwaltungssystem zugänglich zu machen.

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
                    org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>org.eclipse.jubula.releng.client</artifactId>
    <groupId>org.eclipse.jubula</groupId>
    <version>1.1.0-SNAPSHOT</version>
    <relativePath>../org.eclipse.jubula.releng.client</relativePath>
  </parent>
  <groupId>org.eclipse.jubula</groupId>
  <artifactId>org.eclipse.jubula.client.ui</artifactId>
  <version>1.1.0-SNAPSHOT</version>
  <packaging>eclipse-plugin</packaging>
</project>
```

Git it right

Die Frage, was es zu veröffentlichen gab, war von diesem Zeitpunkt an geklärt, die Frage nach dem Wie hingegen noch offen. Firmenintern etabliert war und ist seit Langem das Versionsverwaltungssystem Subversion. Einschlägig bekannt und durch diverses Tooling im Eclipse-Umfeld unterstützt, hat es seine Alltagstauglichkeit bereits dutzende Male unter Beweis gestellt.

Der neue Quasi-Standard in Hinblick auf Versionsverwaltung und Quellcodemanagement im Eclipse-Umfeld trägt aber den Namen EGit bzw. JGit [3]. Also hieß es für uns umsatteln und eintauchen in die Welt von Git. Wir wollen an dieser Stelle keine Einführung in das Arbeiten mit Git geben, sehr wohl aber ein kurzes Fazit unseres Einsatzes innerhalb der letzten zwölf Monate ziehen. Git arbeitet sehr zuverlässig und vor allem schnell: Bedenkt man, dass jedes lokale Repository eine vollständige Kopie des Ursprungs-Repositorys ist. Zu jeder Zeit und überall hat man sein komplettes Repository dabei, inklusive der vollen Historie. Der technische Umstieg vom alten SVN auf das neue Git Repository verlief problemlos, nicht zuletzt deshalb, weil wir darauf verzichtet haben, die gesamte SVN-Historie zu portieren. Der fachliche Umstieg hingegen ist ein kontinuierlicher Prozess und selbst nach der verhältnismäßig langen Zeit noch nicht abgeschlossen (Selbsteinschätzung des Autors). Bis man die Arbeit mit Git wirklich beherrscht und vollständig von seiner Mächtigkeit profitiert, muss man sich Zeit lassen. Erste Schritte wie das „Committen“ und „Pushen“ gelingen aber schon nach wenigen Minuten.

Strict Constraints

An diesem Punkt angelangt mussten wir schnell feststellen, dass das alleinige Bereitstellen von öffentlichem Quellcode aber längst nicht alles ist, was ein Eclipse-Projekt für den Release Train tauglich macht. Eine Reihe von Anforderungen [4] des Release Trains zielen konkret auf einen stabilen und erwachsenen Build-Prozess und dessen Artefakte ab (beispielsweise JARs und ZIPs). Das Bauen der Artefakte muss dokumentiert, automatisiert, wiederholbar und auch von Dritten durchführbar sein. Und auch die entstehenden Artefakte müssen eine ganze Reihe von Eigenschaften erfüllen:

- Alle Bundles müssen als *JAR* vorliegen und das OSGi-Bundle-Format mit einem *MANIFEST.MF* besitzen.
- Sie müssen darüber hinaus ihre minimale Laufzeitumgebung definieren (Bundle Required Execution Environment, BREE) und Gebrauch von vierstelligen Versionsnummern machen.
- Gebündelt werden sollen alle Artefakte (Features und Bundles) in p2 Repositories.
- Jegliche Artefakte, die im Namen der Eclipse Foundation veröffentlich werden, müssen mit dem entsprechenden Eclipse-Zertifikat signiert sein.

Zwar hatten wir zu diesem Zeitpunkt einen bereits seit Langem etablierten Build-Prozess, doch wurde uns schnell klar, dass dieser – galant ausgedrückt – eine hohe Affinität zur lokalen Firmeninfrastruktur besaß. Diese konnten wir unmöglich der Allgemeinheit zugänglich machen. Darüber hinaus können Artefakte nur in der internen Build-Infrastruktur der Eclipse Foundation signiert werden. Die Herausgabe dieser Zertifikate ist nicht möglich. Von daher mussten wir uns schnell von dem Gedanken verabschieden, die für den Release Train benötigten Artefakte lokal vor Ort zu bauen. Auch hier hieß es wiederum Neuland betreten: die Auslagerung der kompletten Build-Infrastruktur auf die Rechner der Eclipse Foundation.

> Das Auskundschaften neuer Territorien hat anfangs häufig in eine Sackgasse geführt.

Remote and local Builds

Dieses Auskundschaften neuer Territorien hat anfangs häufig in eine nicht tragfähige Sackgasse geführt. Aus diesem Grund möchten wir an dieser Stelle die Evaluationsdetails verschiedener Build-Mechanismen überspringen und gleich unseren final verwendeten Lösungsansatz aufzeigen. Jubula wird mit einer Kombination aus Maven3 und Tycho (und einigen wenigen Ant- und Shell-Skripten) in der Hudson-CI-Infrastruktur [5] jeden Tag aufs Neue gebaut. Diese Variante hat sich als sehr flexibel herausgestellt. Das ausgereifte automatische Dependency-Management von Maven in Verbindung mit der Mächtigkeit von Tycho war für uns der entscheidende Faktor. Mit Tycho ist es sehr leicht möglich, diverse Arten von Eclipse-spezifischen Artefakten zu erzeugen. Denn einerseits wollen wir den Build-Prozess zwar primär in der Eclipse-Infrastruktur betreiben, andererseits es uns jedoch nicht vorenthalten, auch alle benötigten Artefakte weiterhin lokal in unserer Infrastruktur bauen zu können. Und nicht selten haben wir von diesem Fallback im letzten Jahr, beispielsweise bei auftretenden Problemen im entfernten Rechenzentrum, schon Gebrauch machen müssen. Diese Kombination lässt es jetzt ohne großen Mehraufwand zu, Folgendes in zwei verschiedenen Infrastrukturen bauen zu können:

- 70 Bundles
- Drei Features

- Verschiedene p2 Repositories
- Unterschiedliche Konfigurationen wie Version und Umfang

In Listing 1 wollen wir zeigen, wie eine solche Bundle-Build-Konfiguration in Maven auszusehen hat. Dank des engen Zusammenspiels von Maven und Tycho fällt diese nämlich erstaunlich übersichtlich aus.

Diese Konfiguration reicht bereits aus, eines unserer größten Bundles zu bauen. Der spannende Teil lässt sich sogar auf *<packaging>eclipse-plugin</packaging>* reduzieren. Diese Option in Verbindung mit einem *mvn clean verify* auf der Kommandozeile genügt, um Maven/Tycho Folgendes mitzuteilen:

> Es war ein Abenteuer, diesen Weg in die Open-Source-Welt zu beschreiten.

- Zuerst alle vorhandenen alten Build-Artefakte aufräumen.
- Dann die Abhängigkeiten zu weiteren Bundles transitiv auflösen.
- 3rd-Party-Bibliotheken, die sich beispielsweise auf dem Runtime Classpath befinden, neben den Bundle-Abhängigkeiten mit auf den Compile Classpath nehmen.
- Compiler-Einstellungen aus den bereits im Bundle definierten OSGi/Equinox-Metadaten wie dem *MANIFEST.MF* und *build.properties* ziehen.
- Alle Klassen aus dem *src*-Verzeichnis kompilieren.
- Alle benötigten Teilartefakte des Bundles in Form eines JARs zusammenfassen.
- Gegebenenfalls existierende Unit Tests ausführen.

Alle weiteren Details zum Build-Prozess können direkt im öffentlichen Repository [6] eingesehen werden – in über 90 % der Fälle beläuft es sich aber auf die zuvor beschriebene POM-Konfiguration eines Projekts.

Auch hier noch eine kleine Anmerkung zum entstandenen Aufwand: Nach abgeschlossener Evaluation des zu beschreitenden Weges hat die Umstellung und Migration ungefähr drei Mannwochen in Anspruch genommen. Denn trotz der hohen Redundanzvermeidung und übersichtlichen Konfiguration, die Maven und Tycho erlauben, steckt der Teufel dann manchmal doch im Detail. Als Stichworte seien hier beispielweise statisches Bytecode-Weaving für das JPA Lazy Loading, die Generierung von Lexern und Parsern sowie die automatische Erzeugung und Konvertierung von Onlinehilfeinhalten genannt.

Do no Harm but do it early

Trotz siebenmonatiger Vorlaufzeit haben wir im Nachhinein betrachtet „auf den letzten Drücker“ am Indigo Release Train teilgenommen. Denn neben der Vielzahl an rechtlichen und technischen Voraussetzungen gab es noch eine Reihe von Deadlines, die es einzuhalten galt. Bereits elf Monate vor den jährlichen Eclipse-Releases beginnen die ersten Integrationszyklen in Form von Aggregations- und Packaging-Läufen. Inzwischen nehmen mehr als 60 Projekte am Release Train teil und damit diese in der finalen Version alle entsprechend gut miteinander harmonieren und „friedlich“ nebeneinander laufen, ist frühes Testen angesagt. Diese anfänglich monatlichen Milestones und später wöchentlichen „Release Candidates“ besitzen dabei fest definierte Zeiten. Zwar ist die Teilnahme anfänglich nicht verpflichtend, aber trotzdem sehr empfehlenswert. Denn findet man ein Problem beim Zusammenspiel der einzelnen Komponenten, dauert es mitunter einige Zeit, bis das Problem behoben werden kann.

And now?

Abschließend lässt sich sagen, dass es in der Tat ein Abenteuer war, diesen Weg in die Open-Source-Welt zu beschreiten. Nicht immer war man sich sicher, wohin der nächste Schritt zu setzen war. Wir hoffen aber, dass diese Kolumne dazu beitragen konnte, ein wenig Licht ins Dunkle zu bringen. Trotz aller etwaiger Startschwierigkeiten hat sich der Schritt für uns in vielerlei Hinsicht bereits jetzt gelohnt. Denn die große Menge an Feedback und die hohe Anzahl an interessierten neuen Anwendern sprechen für sich. Zeit zum Ausruhen bleibt ebenfalls nicht – wer mitgerechnet hat – wir sind bereits beim vierten Milestone für das 2012 erscheinende Juno-Release angelangt. Und auch dort wollen wir wieder mit einer Reihe toller Neuigkeiten eine Jubula-Version veröffentlichen.

Markus Tiede ist Eclipse Committer im UI-Testautomatisierungsprojekt Jubula und Package Maintainer für „Eclipse for Testers“. Als Entwickler und Tester bei der BREDEX GmbH ist er seit 2008 auch an Jubulas kommerziellem „großem Bruder“ GUIdancer beteiligt.

Links & Literatur

[1] Jubula Project Page: http://www.eclipse.org/jubula/

[2] Eclipse Public License v1.0: http://www.eclipse.org/legal/epl-v10.html

[3] Git: http://git-scm.com, JGit: http://eclipse.org/jgit, EGit: http://eclipse.org/egit/

[4] Eclipse Simultaneous Release Requirements: http://wiki.eclipse.org/SimRel/Simultaneous_Release_Requirements

[5] Hudson-Projekt: http://hudson.eclipse.org

[6] Maven POM zum Jubula Build: http://git.eclipse.org/c/jubula/org.eclipse.jubula.core.git/tree/org.eclipse.jubula.releng/pom.xml