Unter der Haube, Teil 2: Ein Blick auf den Entwicklungsprozess des Eclipse-Plattform-Projekts

The Eclipse Way

VON MARTIN LIPPERT

Im ersten Teil dieser Reihe haben wir einen Blick in das Innerste von Eclipse gewagt und uns die Runtime, den Kern von Eclipse, genauer angesehen [8]. In diesem zweiten Teil widmen wir uns einem weniger technischen Thema. Wir gehen der Frage auf den Grund, wie die Entwicklung des Eclipse SDK funktioniert und warum der dahinter stehende Prozess so stabil ist, dass das Projekt über Jahre hinweg jede Deadline einhält und stets qualitativ hochwertige Software ausliefert. Eine Aussicht, die auch für viele andere Projekte verlockend scheint.

Es ist immer wieder aufs Neue beeindruckend, mit welcher Präzision das Eclipse-Plattform-Team Softwarepakete ausliefert. Das Team hält nicht nur die jährlichen Release-Termine exakt ein, auch die sechswöchigen Milestone Builds liefert es fristgerecht ab. Das akribische Einhalten von Deadlines ist allerdings nicht der einzige beeindruckende Punkt: Die Qualität der ausgelieferten Software ist bemerkenswert hoch. Dies spiegelt sich unter anderem darin wider, dass viele Softwareentwickler die Milestone Builds des Eclipse-SDK für ihre tägliche produktive Arbeit einsetzen.

Eine solche Präzision und Zuverlässigkeit ist bei Softwareprojekten leider nicht die Regel. Viele Projekte kämpfen damit, planbar und vorhersehbar Fristen einzuhalten und qualitativ hochwertige Features auszuliefern – und unterliegen häufig genug in diesem Kampf. Aber vielleicht können sich Projekte nicht nur die Eclipse-IDE und -Technologie zunutze machen, sondern sich darüber hinaus eine Scheibe des Entwicklungsprozesses abschneiden? Grund genug, sich den Entwicklungsprozess des Eclipse-Plattform-Projekts einmal genauer anzuschauen und zum Abschluss zu reflektieren, wie sich Teile des Prozesses in eigenen Projekten übernehmen lassen.

Unter der Haube – die Kolumne

- Eclipse Runtime
- **Entwicklungsprozess des Eclipse-Plattform-Projekts**
- Build to last und API-Evolution
- More to follow

Agile Softwareentwicklung

Wie andere moderne Entwicklungsprozesse basiert auch der Entwicklungsprozess des Plattform-Teams, auch „The Eclipse Way“ (nicht zu verwechseln mit dem „Eclipse Development Process“ [1]) genannt, auf den Ideen und Grundsätzen der agilen Softwareentwicklung [2]. Er adaptiert zwar keinen der bekannten und in Büchern beschriebenen agilen Prozesse wie Extreme Programming oder Scrum, bedient sich jedoch einer Reihe von Techniken, die teilweise aus anderen agilen Prozessen bekannt sind.

Der Eclipse-Prozess, so wie er heute vom Team gelebt wird, ist über die vergangenen Jahre hinweg durch ständige Verbesserungen und Anpassungen entstanden. Das ständige Reflektieren über den Prozess ist fester Bestandteil des Prozesses selbst. Zusätzlich enthält der Prozess, ähnlich zumExtreme Programming, eine Reihe von Techniken, die gut zueinander passen und sich gegenseitig positiv beeinflussen. Abbildung 1 zeigt einen Überblick über die Techniken des Eclipse-Entwicklungsprozesses.

Allen Techniken des Eclipse-Prozesses liegt der Wunsch zugrunde, möglichst frühzeitig *Feedback* zu bekommen. Der Wunsch nach Feedback erstreckt sich über alle Bereiche des Entwicklungsprozesses: Das Entwicklungsteam möchte möglichst schnell Rückmeldung darüber bekommen, ob sich ein neuer Fehler in die Software eingeschlichen hat, ob die implementierten Features den Wünschen der Nutzer entsprechen, welche Features dringend benötigt werden, welche Erfahrungen Nutzer mit dem System machen, ob sich die definierten APIs der Plug-ins in der Praxis bewähren oder ob die eingeplanten Features noch bis zum Release implementiert werden können – um nur einige Argumente zu nennen. Im Grunde genommen wird die komplette Entwicklung vom Feedback gesteuert. Und je schneller Feedback geliefert wird, desto schneller kann das Team darauf reagieren.

Aber wer liefert nun möglichst zügig Feedback? In den kleinen Entwicklungs-

zyklen bekommt der Entwickler Rückmeldungen von den Unit-Tests des Systems. Im nächsten Schritt informieren Kollegen, die beispielsweise als Klienten des API auftreten, darüber, welches durch ein Plug-in definiert wird. Der wichtigste Feedback-Lieferant ist aber sicherlich die inzwischen beachtlich große Eclipse-Community.

Project Rhythm

Der grundlegende Herzschlag des Projektes besteht aus den verschiedenen Zyklen, in denen das Team an der Software arbeitet. Der längste Zyklus ist der *Release*-Zyklus. Er umfasst zirka ein Jahr und endet (das gilt jedenfalls für die letzten Releases und das geplante 3.2-Release) klassischerweise im Juni.

Um zu verhindern, dass innerhalb dieses doch recht langen Zeitraums die Qualität und Stabilität des Gesamtsystems zunächst einmal kontinuierlich abnehmen (was bei so langen Entwicklungszeiträumen sehr häufig passiert) und am Ende des Release-Zyklus alle Entwickler hektisch damit beschäftigt sind, das Gesamtsystem wieder zum Laufen zu bringen, werden bei der Eclipse-Entwicklung Release-Zyklen in *Milestone Builds* von je sechs Wochen unterteilt. Ein Milestone Build ist ein Mini-Release-Zyklus mit dem Ziel, eine für die Community nutzbare Version der Software zu erstellen, die einen deutlichen Mehrwert gegenüber dem vorangegangenen Release oder Milestone Build bietet. Gleichzeitig reduzieren die Eclipse-Entwickler dadurch den Stress im Team, der normalerweise zum Ende eines Release auftritt. Durch das sofortige Feedback der Community können Probleme sofort identifiziert werden, nicht erst zum Ende des Release-Zyklus. Jeder Milestone Build besteht aus vier Schritten:

1. Planung des Milestone (was soll implementiert werden?),
2. die eigentliche Entwicklungsarbeit,
3. der Test des kompletten Systems und schlussendlich
4. reflektieren die Team-Mitglieder am Ende eines Milestone Build mittels einer Retrospektive über den vergangenen Milestone und identifizieren, was innerhalb des Prozesses gut funktioniert hat und was ab dem nächsten Milestone verbessert werden soll.

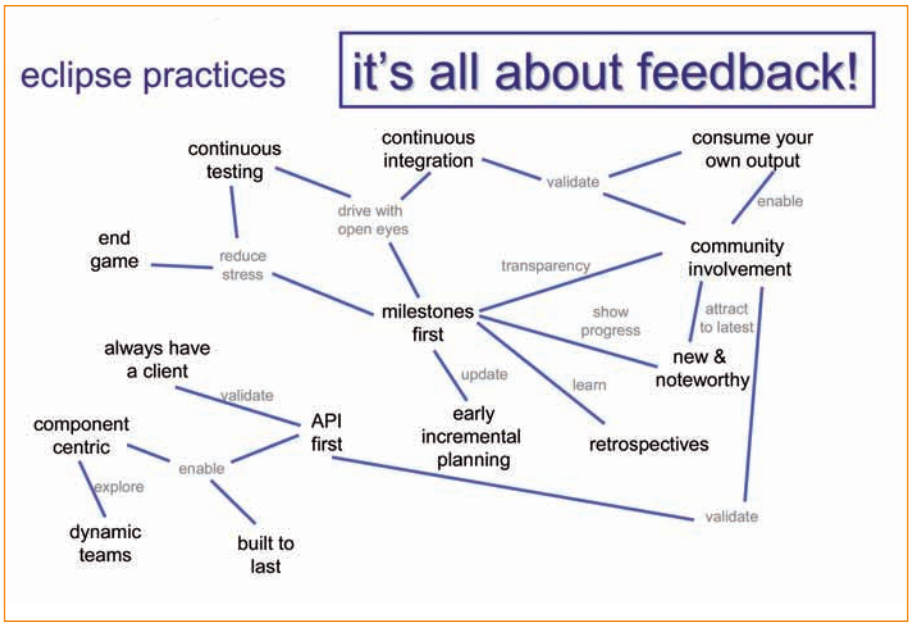

Abb. 1: Die Techniken des Eclipse-Entwicklungsprozesses im Überblick (EclipseCon 2005: The Eclipse Way; © 2005 International Business Machine; EPL 1.0)

Sechs Wochen Zeit zwischen den einzelnen Milestones ist nicht viel, um mit einem Team von mehr als 60 Entwicklern ein konsistentes System auszuliefern, in dem die von unterschiedlichen Sub-Teams entwickelten Komponenten nahtlos integriert sind. Das Eclipse-Team verfolgt deshalb die Strategie der *kontinuierlichen Integration*. Dabei handelt es sich um eine Entwicklungstechnik, die bereits vom Extreme Programming bekannt ist. Sie wird innerhalb der Eclipse-Entwicklung durch den komplett automatisierten Build-Prozess unterstützt. Während jeder Nacht wird ein *Nightly Build* erzeugt, der Integrationsprobleme zwischen Komponenten des Gesamtsystems aufzeigt. In der nächsten Stufe wird einmal wöchentlich ein *Integration Build* erstellt. Zu diesem Integration Build werden alle Integrationsprobleme zwischen Komponenten behoben. Zusätzlich muss die komplette Unit-Test-Suite des Systems fehlerfrei durchlaufen. Schlagen einzelne Tests eines Integration Build fehl, werden diese Fehler behoben und ein neuer Build angestoßen. Dies passiert so lange, bis er fehlerfrei durchläuft.

Durch die unterschiedlichen Builds und die kurzen Entwicklungszyklen befindet sich das Projekt ständig im Beta-Status. Der eigene Anspruch des Teams an die kontinuierlichen Builds des Systems ist, dass jedes einzelne Build ein Release Candidate ist und komplett funktionieren soll. Das bedeutet für die Entwickler auch, ihre eigene Arbeit in kleinen Schritten in den gemeinsamen Code-Bestand zu integrieren und sehr sorgfältig mit Änderungen umzugehen. Das Gesamtsystem darf durch Änderungen nicht in Gefahr gebracht werden. Das bedeutet aus Sicht des Risiko-Managements auch, dass umfangreichere Änderungen an der Codebasis in einem separaten Bereich (bspw. einem Branch oder einem speziellen Incubator-Projekt) entwickelt und dort erprobt werden, bevor sie in den gemeinsamen Code-Bestand zurückgespielt werden. Für die Entwicklung der neuen OSGi-basierten Runtime innerhalb des 3.0-Release-Zyklus ist etwa das damals noch als Technology-Projekt laufende Equinox-Projekt genutzt worden. Ein anderes Beispiel ist die Entwicklung des Java 5-fähigen Compilers gewesen, die zunächst in einem einzelnen Branch stattfand.

Natürlich stellt sich die Frage, wie sichergestellt werden kann, dass durch Änderungen keine neuen Defekte in das Gesamtsystem eingespielt werden. Als Si-

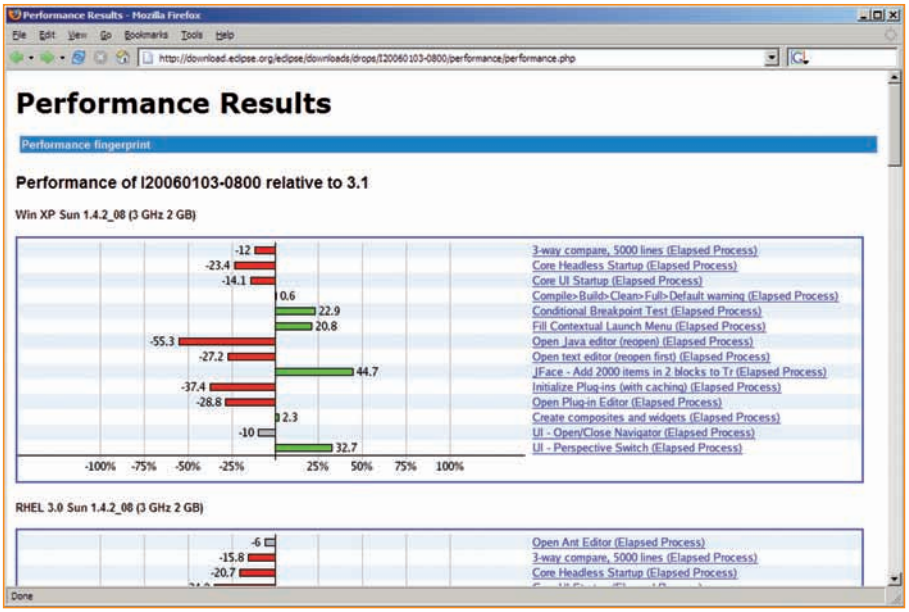

Abb. 2: Ergebnisse der automatisierten Performance-Tests

cherheitsnetz dient im Eclipse-Projekt eine unfangreiche Unit-Test-Suite. Sie wird bei jedem Build automatisiert ausgeführt und protokolliert (die Ergebnisse lassen sich für jeden nachvollziehbar im Web betrachten). Zusätzlich zu Unit-Tests verwendet das Team die eigenen Integration Builds als IDE und folgt damit der Grundidee „Consume Your Own Output" (auch bekannt als „Eat Your Own Dogfood").

Neben dem Feedback aus den eigenen Reihen legt der Eclipse-Entwicklungsprozess viel Wert auf Feedback aus der Community. Um diese Einbindung der Community zu fördern, sieht der Prozess eine Reihe von unterstützenden Maßnahmen vor:

- *Transparenz:* Die Entwicklung innerhalb des Eclipse-Projekts ist sehr transparent und kann von jedem eingesehen werden. Dazu zählen sowohl Milestone-Pläne als auch der komplette Satz an Bugs in Bugzilla und das für lesende Zugriffe offene CVS. Die Entwickler kommunizieren untereinander unter anderem über öffentlich zugängliche Mailing-Listen. Über diese Listen werden regelmäßig Zusammenfassungen der Meetings verteilt und somit öffentlich gemacht.
- *Wertschätzung:* Die Mitarbeit der Community wird geschätzt und entsprechend honoriert. Jeder hat die Chance, sich zu beteiligen und gehört zu werden.
- *Information:* Um das Interesse der Community beispielsweise an Milestone Builds zu wecken, wird jeweils eine „New and Noteworthy"-Dokumentation erstellt. Sie enthält eine kurze Übersicht über Neuerungen im entsprechenden Milestone Build.

Testing

Umfangreiche und vor allem automatisierte Tests spielen im Eclipse-Entwicklungsprozess eine große Rolle. Auch sie dienen wieder dem frühzeitigen Feedback über die Qualität des Systems und dem raschen Hinweisen auf Probleme.

Auf der einen Seite besitzt das Eclipse-SDK-Projekt eine Unit-Test-Suite mit mehr als 20.000 JUnit-Tests. Diese JUnit-Tests werden bei jedem Build automatisiert durchgeführt und liefern Feedback darüber, ob das System korrekt funktioniert. Neben diesen Tests hat das Eclipse-Team eine Reihe von Performance-Tests implementiert, die bei jedem Integration Build ebenfalls automatisch durchgeführt werden. Die Ergebnisse der Performance-Tests geben Auskunft darüber, wie sich die Performance des Build relativ zu der des letzten Release verhält (Abb. 2). Hat sich beispielsweise die Performance des Systems in einem bestimmten Bereich stark verschlechtert, können die Entwickler aufgrund der automatischen Performance-Tests schnell feststellen, seit welchem Integration Build die Performance nachgelassen hat, und Rückschlüsse darauf ziehen, welche Änderungen die Performanceverschlechterung ausgelöst haben. Die Ergebnisse der Performance-Tests sind für jedermann einsehbar. Sie werden, ebenso wie die Ergebnisse der JUnit-Tests, mit dem Build im Web veröffentlicht.

Finishing

Nähert sich das Entwicklungs-Team einem Release-Termin, geht die Entwicklung in das so genannte *End Game* über. Während des ein- bis zweimonatigen End Game konzentriert sich das gesamte Team darauf, die Software intensiv zu testen, die letzten Bugs in der Software zu fixen und so die höchstmögliche Qualität des Release sicherzustellen.

Milestone Builds werden in dieser Zeit durch Release Candidate Builds ersetzt, die in deutlich kürzeren Abständen produziert werden – häufig ein Release Candidate pro Woche. Zwischen zwei Release Candidates findet je eine Test- und eine Fix-Phase statt. Dabei wird es mit jeder Fix-Phase schwieriger, Bugfixes in das System zu integrieren. Je später ein Fix in das System eingespielt wird, desto mehr Team-Mitglieder müssen den Fix absegnen. Damit minimiert das Team das Risiko, die Qualität des Systems durch eine späte Code-Änderung zu gefährden. Die genaue Planung des End Game ist ebenfalls offen im Web einzusehen.

Dem End Game folgt die *Decompression*. Sie hilft dem Team, sich zu erholen, über die vergangene Zeit zu reflektieren und aus den Erfahrungen zu lernen – und das nächste Release zu planen. Zeitlich teilt sich ein Release-Zyklus in etwa neun Monate für Milestone Builds, ein bis zwei Monate für das End Game und einen Monat für die Decompression.

Planning

Die Planung von Features innerhalb eines Release orientiert sich an *Themes*, welche übergeordnete Ziele für ein Release definieren. Für das Release 3.2 sind diese Themes beispielsweise: Scaling Up, Enterprise Ready, Design for Extensibility:

Bitte lesen Sie weiter auf S. 29

Be a Better Platform, Simple to Use, Rich Client Platform, Appealing to the Broader Community. Orientiert an Themes planen die einzelnen Teams *Component Plans*. Component Plans verzeichnen einzelne Features, jeweils mit einer kurzen Erläuterung und Verweisen auf die entsprechenden Bugs in Bugzilla und aufgeteilt in drei Kategorien: *Committed*, *Proposed* und *Deferred*. Ebenso wie Themes für ein Release sind auch die einzelnen geplanten Features für jedes Subprojekt im Web veröffentlicht [3].

Die initiale Planung zu Beginn eines Release ist keinesfalls für das komplette Jahr festgeschrieben. Stattdessen wird der Plan regelmäßig (je Quartal) an den aktuellen Fortschritt des Projekts und das Feedback aus der Community angepasst. Der finale Plan wird deshalb erst am Ende eines Release festgeschrieben.

Natürlich passiert es selbst bei einem so vorbildlichen Entwicklungsprozess, dass sich das Team verschätzt und weniger Features innerhalb eines Release implementieren kann, als ursprünglich geplant. Trotzdem ist stets die oberste Priorität, den Release-Termin zu halten. Deshalb nimmt man gegebenenfalls wieder Features aus der Planung heraus, wenn sich herausstellt, dass sich nicht alle Features bis zum Termin fertig stellen lassen. Ziel ist, lieber weniger Features komplett zu implementieren, als mehr Features nur unvollständig aufzunehmen.

Eine wichtige Maßnahme bei der Planung ist, einerseits frühzeitig risikoreiche Features oder Änderungen an der Code-Basis zu identifizieren und andererseits geplante Änderungen zu erkennen, die sehr viele Abhängigkeiten im System besitzen. Solche risikoreichen Aufgaben werden im Entwicklungsprozess möglichst frühzeitig innerhalb eines Release-Zyklus eingeplant. Dadurch verringert man das Risiko, Features oder Änderungen nur unvollständig zu implementieren und das Team gerät nicht kurz vor dem Release unter Zeitdruck.

Build to last

Die Eclipse-Plattform dient inzwischen sehr vielen kommerziellen wie auch freien Projekten als Basis für die eigene Arbeit. Diese immer größer werdende Community hat den immensen Vorteil für die Entwicklung der Eclipse-Plattform, dass sehr viel Feedback und Unterstützung aus der Community zurück in das Eclipse-Projekt fließt. Obwohl diese Community an immer neuen Features und verbesserten Strukturen der Plattform interessiert ist, möchte sie selbstverständlich nicht mit jedem Release den eigenen Code komplett umschreiben, um in den Genuss der neuen Plattform-Version zu kommen. Dieser Zielkonflikt ist für das Eclipse-Plattform-Projekt mit äußerst großen Anstrengungen verbunden, die sich auch im Prozess niederschlagen.

Ohne allzu tief in die technischen Details der Binär-Kombatibilität und des evolutionären API-Designs zu gehen (der Leser sei auf den nächsten Teil dieser Serie verwiesen), bedeutet dieser Plattform-Charakter für den Prozess, dass deutlich intensiver an der API-Gestaltung der einzelnen Komponenten gearbeitet werden muss. Mehr Informationen zu API-Design und -Evolution findet man unter [4] [5] [6].

In eigenen Projekten

Viele der hier dargestellten Eigenschaften des Eclipse-Entwicklungsprozesses orientierten sich an bewährten Techniken agiler Methoden und lassen sich nahtlos für eigene Projekte, auch gängige Inhouse-Entwicklungsprojekte, übernehmen. Insbesondere die Grundidee, möglichst schnell Feedback über die produzierte Software zu erhalten, kann in eigenen Projekten dazu dienen, den Prozess neu auszurichten oder schrittweise zu verbessern.

Schwieriger wird es natürlich, Techniken wie „Consume Your Own Output“ direkt zu übertragen, da in gängigen Entwicklungsprojekten meist keine Software für Softwareentwickler erstellt wird, sondern für Anwender der entsprechenden fachlichen Domäne. Solche Techniken müssen entsprechend ersetzt oder angepasst werden. Ähnlich schnelles Feedback können häufig Mitarbeiter der Fachseite geben, die möglichst im Projekt direkt mitarbeiten sollten. Sie agieren damit als Community. Wichtig ist, dass diese Community genauso vertrauensvoll in das Projekt eingebunden wird, wie dies beim Vorbild Eclipse der Fall ist: Sie muss ernst genommen werden. Besonderes Gewicht liegt auf der Transparenz des Entwicklungsprozesses für alle Beteiligten. Jeder muss zu jedem Zeitpunkt in jeden Bereich des Prozesses Einblick haben und Feedback liefern können. Es hilft z.B. nicht, Probleme vor dem Kunden (bzw. der Community oder den Kollegen aus dem Fachbereich) zu verschweigen. Nur eine vertrauensvolle Zusammenarbeit macht das den Prozess steuernde Feedback ehrlich, direkt und schnell – und hilft dem Team, qualitativ hochwertige Software termingerecht abzuliefern.

Viele der Eigenschaften des Eclipse-Entwicklungsprozesses orientieren sich an Techniken agiler Methoden und lassen sich für eigene Projekte übernehmen.

Die Darstellung des Prozesses basiert maßgeblich auf der Keynote „The Eclipse Way: Processes that Adapt“ von John Wiegand und Erich Gamma auf der EclipseCon 2005 [7]. Mein Dank gilt außerdem Bernd Kolb für sein Feedback zu einer Draft-Version des Artikels.

Martin Lippert ist Senior-IT-Berater bei it-agile. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie und ist Committer im Eclipse Equinox Incubator-Projekt. Er ist zu erreichen unter: martin.lippert@it-agile.de.

Links & Literatur

[1] www.eclipse.org/projects/dev_process/

[2] www.agilemanifesto.org

[3] www.eclipse.org/eclipse/development/eclipse_project_plan_3_2.html

[4] www.eclipse.org/articles/Article-API%20use/eclipse-api-usagerules.html

[5] www.eclipse.org/eclipse/development/java-api-evolution.html

[6] www.eclipsecon.org/2005/presentations/EclipseCon2005_12.2APIFirst.pdf

[7] eclipsecon.org/2005/presentations/econ2005-eclipse-way.pdf

[8] Martin Lippert: Was Eclipse im Innersten zusammenhält. Unter der Haube, Teil 1: Ein Blick auf die Eclipse Runtime, in *Eclipse Magazin* Vol. 5

[9] „Zentral ist die Transparenz in der Entwicklung“. Interview mit Dirk Bäumer und Erich Gamma, in *Eclipse Magazin* Vol. 4