Modernisierung des Entwicklungsprozesses – ein Projektbericht

Markus Heinisch
Trivadis GmbH
München

Schlüsselworte
Continuous Integration , Hudson/Jenkins, Java, Nexus, Apache Subversion, Eclipse RCP

Einleitung
Neue und höhere Anforderungen an eine Entwicklungsabteilung eines Finanzinstituts erfordern ein neues Vorgehen in der Softwareentwicklung. Wie können mehr Projekte, höhere Komplexität der Geschäftsfälle und ein schnelleres „time to market“ realisiert werden? Die Antwort lautet: modular statt big block, Fast Feedback statt Ungewissheit, Standards statt proprietär. Das Ergebnis ist ein Continuous-Integration-Prozess, der agile Softwaremethoden unterstützt und für jeden Entwickler überfällige Verbesserungen bedeutet. Der Projektbericht greift die Themen Release-Management, Konfigurationsmanagement und den Einsatz von Tools wie Jenkins, Maven und Nexus auf. Wer über den Umstieg auf einen agilen Entwicklungsprozess nachdenkt, lernt in diesem Erfahrungsbericht vieles über Erfolgsfaktoren und Vorgehen.

Ausgangslage
Über die Jahre haben sich die Rahmenbedingungen für die Inhouse-Entwicklung von Desktop- und Web-Clients für den Vertrieb von Finanzprodukten geändert. Die Anzahl an Projekten, die in der Abteilung entwickelt werden, ist gestiegen. Gleichzeitig hat die fachliche Komplexität durch mehr Geschäftsfälle zugenommen. Zusätzlich ist die Erwartungshaltung der Kunden bezüglich eines „time to market“ deutlich höher als vor 5-10 Jahren. Für alle Projektbeteiligten ergibt sich das nicht unerhebliche Risiko, dass neue Versionen eines Produktes nicht zum gewünschten Termin bereitgestellt werden können.

Auf Basis dieser Ausgangslage hat das Entwicklungsteam das Build- und Deployment-System als Problemfeld identifiziert. Es verursacht sehr hohe Wartungskosten und ist nicht stabil und flexibel genug für die geänderten Rahmenbedingungen. Insbesondere hat das System große Auswirkung auf die Produktivität der einzelnen Entwickler.

Entsprechend wurde ein Projekt gestartet, das die identifizierten Probleme durch den Aufbau eines modernen Continuous Integration-Prozesses löst. Zum Start des Projekts wurden die folgenden Ziele vorgegeben:

- Höherer Automatisierungsgrad im Build- und Deployment-Prozess
- Mehr Zeit für die Entwickler zur Erfüllung der fachlichen und zeitlichen Kundenanforderungen
- Standardisierung der Build-Umgebung und Prozesse
- Veraltete und ungeeignete Lösungen sollen durch etablierte „state of the art“-Technologien abgelöst werden

Bisheriger Build- und Deployment-Prozess
Der bisherige Build- und Deployment-Prozess wurde als einer der wesentlichen Risikofaktoren im Entwicklungsvorgehen identifiziert. Der Grund für diese Bewertung liegt im komplexen Zusammenspiel der verwendeten Tools und Technologien im gelebten Entwicklungsprozess.

Zum Build der Software-Produkte wird das Tool „Automated Build Studio“ von Smartbear verwendet. Das Tool unterstützt u.a. Scripting für den Ablauf von Builds, was intensiv genutzt worden ist. Eine solche Buildprozedur zum Bau eines Produkts kann aus 50 und mehr manuell konfigurierten Build-Schritten bestehen. Der Entwicklungsprozess sieht so aus, dass ein kleines Team von Entwicklern für den Build verantwortlich ist. Jeden Montagmorgen um 11:00 wird ein Build manuell auf einen zentralen Buildserver durchgeführt. Das Ergebnis ist ein sogenannter Release, der eine offizielle Releasenummer bekommt und dem Testteam zur Verfügung gestellt wird. Ansonsten wird ein Produkt nicht komplett aus den Sourcen zusammengebaut. Dieser Build-Vorgang dauert relativ lange und ist fehleranfällig. Oftmals brechen die Builds auf Grund von Fehlern ab und das Buildteam muss die Fehler analysieren und beheben. Die Entwickler können auf ihrem Entwicklungsrechner ebenfalls einen kompletten Bau eines Produktes durchführen, dazu müssen sie allerdings Eclipse verwenden. Dieser Build auf den lokalen Rechner ist nicht identisch mit dem Build auf dem zentralen Server.

Zur Sourcecode-Verwaltung wird IBM Rational ClearCase eingesetzt. Das Tool verwaltet neben den Sourcecode der Produkte auch die gebauten binären Artefakte zur Datensicherung und wird entsprechend in den Build-Prozeduren ungewöhnlich häufig involviert. Für den Einsatz von ClearCase muss jeder Entwickler eine Konfiguration seiner lokalen Arbeitskopie (View genannt) besitzen. Über diese Konfiguration wird gesteuert, welche Dateien in welcher Revision verfügbar sind. Es hat sich gezeigt, dass diese Konfiguration in der Praxis komplex und sehr anfällig ist und zu einer fehlerhaften oder unvollständige lokale Arbeitskopie führt. Neue Mitarbeiter bringen in nur in seltene Fällen ClearCase-Erfahrung mit. Somit ist es notwendig, dass sie den Umgang mit ClearCase erlernen müssen.

Insgesamt kann folgendes festgestellt werden,

- Build eines Produktes nur einmal in der Woche durchgeführt wird
- Entwickler entsprechend spät Feedback zu ihrem Coding bekommen
- der Prozess manuell durchgeführt wird und zusätzlich nur von einem kleinen Team beherrscht wird

Release-Strategie
Unter dem Stichwort Release Strategie wird im Rahmen des Projektes der Umgang mit einem Versionsverwaltungssystem (SCM), insbesondere der Bereich Branching und Merging, in Bezug auf das Erstellen eines Produkt-Releases verstanden.

Grundsätzlich existiert unabhängig von dem gewählten Versionsverwaltungssystem eine Reihe von Branching&Merging-Verfahren, wie zum Beispiel:

- Develop on Mainline
- Branch for Release
- Branch by Feature
- Branch by Team

Im vorliegenden Projektbericht werden nur die ersten beiden Strategien vorgestellt, da sie auch im Projekt zum Einsatz gekommen sind.

„Develop on Mainline“ bedeutet, dass die Entwickler ihren Sourcecodeänderungen nahezu ausschließlich auf der Mainline (trunk) durchführen. Alle Änderungen sind für alle anderen Entwickler direkt nach einem Checkin der Sourcen verfügbar. Das Verfahren vermeidet das Merging von Sourcecode zu definierten Projektständen, da die Mainline immer den aktuellen Sourcecode enthält. Damit ist sichergestellt, dass Continuous Integration stattfinden kann und somit ein Build auf der Mainline die Integration der Sourcen aller beteiligter Entwickler wiedergibt. In der Praxis zeigt sich, dass nicht jeder Build wird erfolgreich sein. Das Vermeiden von Branches kann bedeuten, dass die Entwicklung etwas höheren Aufwand beim Entwickeln von neuen Features haben kann. Branches machen allerdings Sinn unter der Voraussetzung, dass Änderungen im Branch nicht mehr in die Mainline gezogen (merge) werden, wie beispielsweise bei einem Release.

„Branch for Release“ bedeutet, dass man kurz vor einem Release eine Branch für den Zweck eines Release durchführt. Dieses Vorgehen komplettiert die „Develop on Mainline“ Strategie. Die grundlegende Idee ist, dass ein Team auf der Mainline neue Features entwickelt während ein zweites Team auf dem Release-Branch Bugfixes durchführen will - ohne dadurch neue Features in das Release zu bringen.

Wenn ein Branch erzeugt ist, dann wird im Wesentlichen der Code in dem Branch nur noch getestet und das Release validiert, während in der Mainline an Features weiterentwickelt wird. Dieses Vorgehen minimiert den sogenannten „Code Freeze“, während der Zeit kein Code im SCM geändert werden darf und fördert die unabhängige Entwicklung von Features mit dem Erstellen eines Release. Das Entwicklungsvorgehen ist wie folgt: Zunächst werden die Features auf der Mainline entwickelt. Ein Branch wird erst dann gezogen, wenn der Feature-Code „releaseable“ ist, d.h., alle für den Release notwendigen Features sind entwickelt und wurden von den Entwicklern getestet. Von diesem Codestand wird ein Release-Branch gezogen. In diesem Branch werden nur noch Bugfixes durchgeführt und sofort in die Mainline übernommen (merging).

Dieses Verfahren eignet sich weniger für große Teams, weil es auf Grund der Teamgröße schwierig wird festzulegen, wann ein Set von Features fertig ist und ein Release-Branch gezogen werden sollte.

Für die Aufgaben in der Abteilung ist das Verfahren „Develop on Mainline“ und „Branch for Release“ vorgesehen. Die Gründe für diese Entscheidung liegen in den folgenden fünf Vorteilen:

- Optimale Unterstützung des Continuous Integration-Prozesses
- Vergleichsweise einfache Handhabung in der Praxis
- Es sollen maximal zwei Releases der Software in der Entwicklung sein
- Nahe am bisherigen Branching&Merging-Verfahren
- Höhere Planungssicherheit zu offiziellen Release-Terminen

Versionsverwaltungssystem (SCM)
Als neues SCM-Tool wird Subversion verwendet. Im Subversion (SVN) wird der Sourcecode für mehrere zum Teil von einander abhängiger Produkte verwaltet. Bevor die Sourcen dort abgelegt werden muss die Verteilung und die Struktur der Sourcen im SVN festgelegt werden.

Bei mehreren Produkten stellt sich die Frage, ob ein Repository für alle Produkte (Variante A) oder ob pro Produkt ein Repository (Variante B) angelegt werden soll?

Variante A

```
http://svn.company.com:3080/abc/Produkt_A/...
                            /Produkt_B/...
                            /Produkt_C/...
```

Variante B

```
http://svn.company.com:3080/abc_Produkt_A/...
http://svn.company.com:3080/abc_Produkt_B/...
http://svn.company.com:3080/abc_Produkt_C/...
```

Abb. 1: ein Repository für alle Produkte (Variante A) oder ob pro Produkt ein Repository (Variante B)?

Die Unterschiede zwischen beiden Varianten sind aus technischer Sicht nicht so gravierend, beispielsweise werden Berechtigungen per Repository gepflegt, die Revisionsnummer wird per Repository verwaltet und eine Merging von Sourcen wird nur innerhalb eines Repositories unterstützt. Letztlich ist die Entscheidung für Variante A gefallen, da sie den geringsten administrativen Aufwand (außerhalb der Entwicklungsabteilung) bedeutet.

Nachdem die erste Frage geklärt ist muss nun festgelegt werden, wie die Struktur der Sourcen innerhalb eines Repositories ist.

Variante (1)

```
…/abc/
    /ProjektA/
              /trunk
              /branches
              /tags
    /ProjektB/
              /trunk
              /branches
              /tags
    /ProjektC/
              /trunk
              /branches
              /tags
```

Variante (2)

```
…/abc/
    /trunk/
          /ProjektA
          /ProjektB
          /ProjektC
    /branches/
             /ProjektA
             /ProjektB
             /ProjektC
    /tags/
         /ProjektA
         /ProjektB
         /ProjektC
```

Abb.2: Welche Struktur der Sourcen innerhalb eines Repositories?

Für Subversion gibt es bei beiden Varianten keine technischen Unterschiede. Die Variante 1 trennt die Produkte sehr anschaulich während die Variante 2 die SCM Eigenschaften wie Tags und Branches betont. Aus Sicht eines Entwicklers kann bei Variante 1 der Check-out aller Sourcen eine Produktes über einen Link durchgeführt werden während bei Variante 2 der Check-out aller Sourcen nach „Typ“ *trunk*, *tag* oder *branch* explizit ausgewählt werden muss. Wenn man sich eine mögliche Weiterentwicklung der Produkte ansieht und Produkte aus Komponenten bestehen, die individuell fortgeführt werden, dann sehen die Varianten wie folgt aus:

Variante 1

```
…/abc/
    /ProjektA/trunk/cmp1/
                        /trunk
                        /branches
                        /tags
                   /cmp2/
                        /trunk
                        /branches
                        /tags
            /branches
            /tags
    /ProjektB/…
```

Variante 2

```
…/abc/
    /trunk/
          /ProjektA/cmp1
                   /cmp2
          …
    /branches/
             /ProjektA/cmp1
                      /cmp2
             …
    /tags/
         /cmp1
         /cmp2
         …
```

Abb.3: Was passiert, wenn zusätzliche Komponenten eingeführt werden?

Die Entscheidung ist für Variant 2 gefallen, da die Organisation der Sourcen verständlicher ist und der Entwickler den *trunk* eines Produktes kopieren kann ohne das er die *tags* und *branches* der darunter liegenden Komponenten wie *cmp1* oder *cmp2* automatisch mitkopiert.

Build-Prozess

Der bisherige Build- und Deployment-Prozess jedes Produktes wurde mittels Automated Build Studio auf einem zentralen Server zum Teil manuell durchgeführt. Die entwickelten Produkte basieren technisch auf Eclipse RCP und einer Reihe eigener Komponenten (Plugins). Die funktionale Basis der Produkte ist eine Eclipse RCP basierte Kernanwendung, die allgemeine Geschäftslogik implementiert. Diese Kernanwendung wird von einem Partnerunternehmen zur Verfügung gestellt und im Entwicklerteam um spezifische Geschäftslogik erweitert.

Im Build-Prozess wurde immer ein komplettes Produkt aus allen notwendigen Sourcen gebaut. Dies gilt auch für Komponenten, deren Sourcen sich nicht geändert hatten. Der neue Build-Prozess ist etwas anderes konstruiert, die wesentlichen Unterschiede sind:

- Statt in einem Build-Prozess wird das Produkt mittels mehreren aufeinander aufbauender Builds gebaut
- Umfangreiche Produkte werden modularisiert
- Jeden Tag (bzw. nach jedem Checkin ins SVN) wird gebaut
- Nicht jeder Build führt zu einem neuen „Release“

Im heutigen Build-Prozess wird das Open Source Tool Jenkins (http://jenkins-ci.org/) als Build-Service. Der Build wird mit Hilfe von Apache Maven durchgeführt. Dieser Build mittels Maven kann auf dem Rechner des Entwicklers identisch durchgeführt werden. Ein Build liefert ein Artefakt, das in nachfolgenden Build verwendet werden kann. Zum versionierten Verwalten dieser Artefakte wird das Open Source Tool Nexus (http://www.sonatype.org/nexus/) verwendet. Somit werden Abhängigkeiten, die eine Komponente während des Builds benötigt, über Nexus unter Angabe der Version aufgelöst. Durch das Zusammenspiel von Jenkins, Nexus und SVN ergibt sich folgender Build-Prozess:

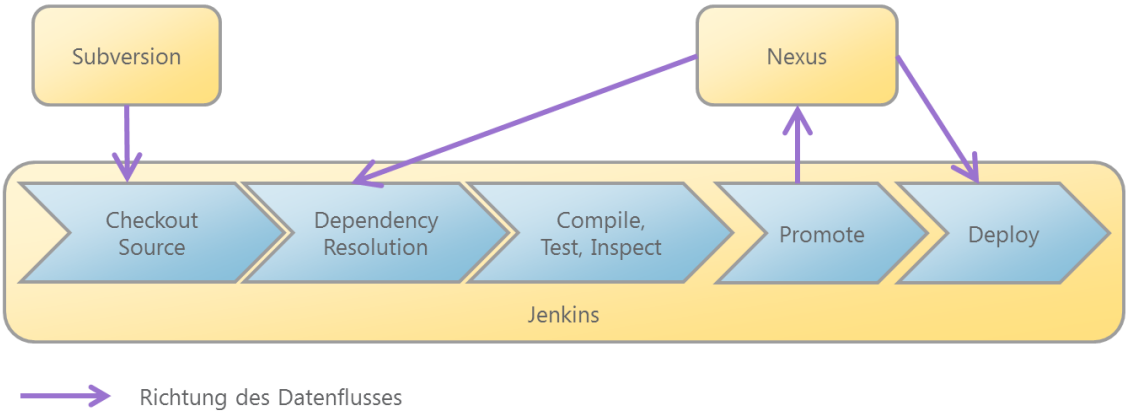

Abb.4: Neuer Build- und Deployment-Prozess

Die ersten drei Prozessschritte „Checkout Source“, “Dependency Resolution“, “Compile, Test, Inspect“, werden automatisiert ausgeführt, sobald ein Entwickler seine Änderungen in Subversion eingespielt hat. Der vierte Schritt „Promote“ wird manuell aufgerufen, wenn das erzeugt Artefakt eine

definierte funktionale Vollständigkeit und einen bestimmten Qualitätsstand erreicht hat. Damit kann das Artefakt in anderen Builds zur Dependency Resolution wiederverwendet werden und muss zu diesem Zweck nicht nochmal erzeugt werden. Der Schritt Deploy wird ebenfalls manuell aufgerufen. Dieser Schritt existiert nur für Produkte und nicht für einzelne Komponenten. Ein Produkt wird in diesem Schritt aus den einzelnen Komponenten zusammengesetzt und in Form eines installierbaren Executables dem Testteam auf einen Dateiserver zur Verfügung gestellt.

Fazit

Das Entwicklungsteam hat die Umstellung auf den neuen Continuous Integration-basierten Entwicklungsprozess gut angenommen. Die sichtbarsten Änderungen für die Entwickler liegen in der Verwendung von Subversion statt ClearCase und in dem schnellen Feedback des CI-Prozesses auf Sourcecode-Änderungen. Das Buildteam muss sich heute auf Grund des höheren Automatisierungsgrades des Prozesses nicht mehr um den Bau der Produkte kümmern, sondern kann sich auf die technische Releaseplanung konzentrieren.
Die geforderten Projektziele wurden erreicht. Insbesondere hat das Entwicklungsteam einen agilen Prozess implementiert, mit dem eine höhere Qualität und größere Zuverlässigkeit erreicht werden kann.

Kontaktadresse:

Markus Heinisch
Trivadis GmbH
Lehrer-Wirth-Str. 4
D-81829 München

Telefon:	+49 (0) 89 99275930
Fax:	+49 (0) 89 99275959
E-Mail	markus.heinisch@trivadis.com
Internet:	www.trivadis.com