KfW-Bildarchiv/Fotograf: Thomas Klewar

# Praxisbericht

## Modernisierung der Entwicklungsinfrastruktur bei der KfW (Kreditanstalt für Wiederaufbau)

von Jürgen Biebl und Wolfgang Werner

**In der letzten Ausgabe der NEWS wurde im Artikel „Blick hinter die Kulissen – wie man typische Stolpersteine bei der Softwareentwicklung umgeht“ eine Entwicklungsinfrastruktur vorgestellt, die die Implementierung individueller Softwarelösungen in hoher Qualität durch Automation und kontinuierliche Integration, Installation und Inspektion unterstützt. Eine solche Infrastruktur auf der sprichwörtlichen „grünen Wiese“ aufzubauen, ist weitgehend unproblematisch. In der Realität sehen wir uns jedoch oftmals einer seit Jahren gewachsenen, hochgradig heterogenen Technologielandschaft gegenüber, deren Transformation in eine moderne Entwicklungsinfrastruktur als schwer greifbare Mammutaufgabe erscheint.**

**Dieser Praxisbericht beschreibt das Vorgehen bei der Modernisierung der Versionskontroll-, Build- und Deployment-Infrastruktur am konkreten Beispiel der KfW, leitet allgemeingültige Vorgehensweisen und Patterns ab und zeigt mögliche Hürden auf.**

### Die Ausgangssituation

Der Handlungsbedarf für die Modernisierung der Entwicklungsinfrastruktur der KfW hatte verschiedene Gründe. Ein kontinuierlich wachsendes Volumen bei Softwareentwicklungsprojekten war und ist aufgrund geplanter Vorhaben vorauszusehen. Die Standardisierung der über viele Jahre gewachsenen, heterogenen Strukturen und Prozesse wird deshalb zunehmend wichtiger. Den

angestrebten Softwareentwicklungsprozess und die Anforderungen moderner, technologieübergreifender Großprojekte kann die aktuelle Infrastruktur nur noch ungenügend abbilden.

## Das Zielbild

Bei der Modellierung des Gesamtsystems waren die notwendigen Komponenten schnell identifiziert: Das bestehende Versionskontrollsystem musste aktualisiert werden, weiterhin sollte eine zentrale Build-/Continuous-Integration-Plattform etabliert werden. Um die historisierte Ablage von Softwareartefakten und die Abbildung von Abhängigkeiten zwischen diesen zu erfassen, wird darüber hinaus ein Artifact Repository benötigt.

**Das Projekt auf einen Blick**

**Eckdaten zum Entwicklungsprojektbestand:**

- Ca. 200 Entwickler nutzen die betroffenen Systeme täglich intensiv
- Über 20.000 aktive Entwicklungsprojekte werden bearbeitet
- Ca. 65 Millonen Lines of Code
- Monatlich ca. 120.000 Builds und 3.200 Deployments
- Unterstützte Technologien: Java, PL/I, JCL, C++, Summit, C, C#, Centura, Telon
- Einsatz von Git, Gerrit, Atlassian SourceTree, Jenkins, Atlassian Jira, Eclipse, Maven und Sonatype Nexus in der Ziellandschaft

**Primärziele beim Design der Gesamtlandschaft:**

- Identifizierbarkeit der Softwarestände in den Test- und Produktionsumgebungen zu jeder Zeit „auf Knopfdruck“
- Beibehaltung des bestehenden Freigabeprozesses bei Softwareänderungen
- Optimale Unterstützung von Parallelentwicklung (Branching) und Zusammenführung verteilter Änderungen (Merging)
- Optimale Unterstützung der Softwareentwicklung an verschiedenen Standorten
- Schaffung der Basis für ein durchgängiges und technologieübergreifendes Abhängigkeitsmanagement
- Keine Gefährdung bestehender Build-Prozesse
- Ablösung der Eigenentwicklungen im IT-Betrieb durch moderne Industriestandards

Auf die Einführung einer Komplettlösung für das Application-Life-Cycle-Management wurde verzichtet, da aktuelle Open-Source-Werkzeuge einen sehr hohen Reifegrad haben und sich sehr gut integrieren lassen. Der Vorteil: Einzelne Komponenten der Gesamtlandschaft lassen sich separat austauschen, was bei einer Komplettlösung nicht möglich wäre. Ein Vendor-Lock-in wird so vermieden. Die Anwendung der eingesetzten Tools Git, Gerrit, Nexus und Jenkins ist darüber hinaus in der Entwicklerszene weitverbreitet.

Als grundlegendes Paradigma für die Strukturierung der Systemlandschaft wird ein einheitliches Staging- und Promotion-Konzept vorgegeben, das den Lebenszyklus eines Projekts oder einer neuen Funktion vom Source-Code bis hin zu den produktiven Laufzeitumgebungen abdeckt. Die Bausteine dazu sind im Source-Code-Managementsystem hinterlegte, einzelne Entwicklungszweige (Branches) für jede Laufzeitumgebung, in die einzelne Features für die jeweiligen Releases übernommen werden.

Aus dem jeweiligen Branch wird durch den Build ein Artefakt erzeugt, das im Artifact Repository dann wiederum in den zugehörigen Bereich (Snapshot für Entwicklung, Staging für Test und Qualitätssicherung und Release für Produktion) übernommen wird. Aus dem Artifact Repository kann dann aus dem jeweiligen Bereich das Deployment auf die Laufzeitumgebungen erfolgen.
Alle Build-, Promotion- und Deployment-Jobs werden zentral im Continuous-Integration-System verwaltet, das eine einheitliche Schaltstelle für die Nutzung, Einplanung und die revisionssichere Protokollierung dieser Aufgaben darstellt.

Die Rollendefinition und -zuweisung, Authentifizierung und Autorisierung werden systemübergreifend im zentralen Active Directory der KfW vorgenommen, in dem sämtliche Systeme der Landschaft integriert sind. Durch Projektnamen und Versionsnummern werden

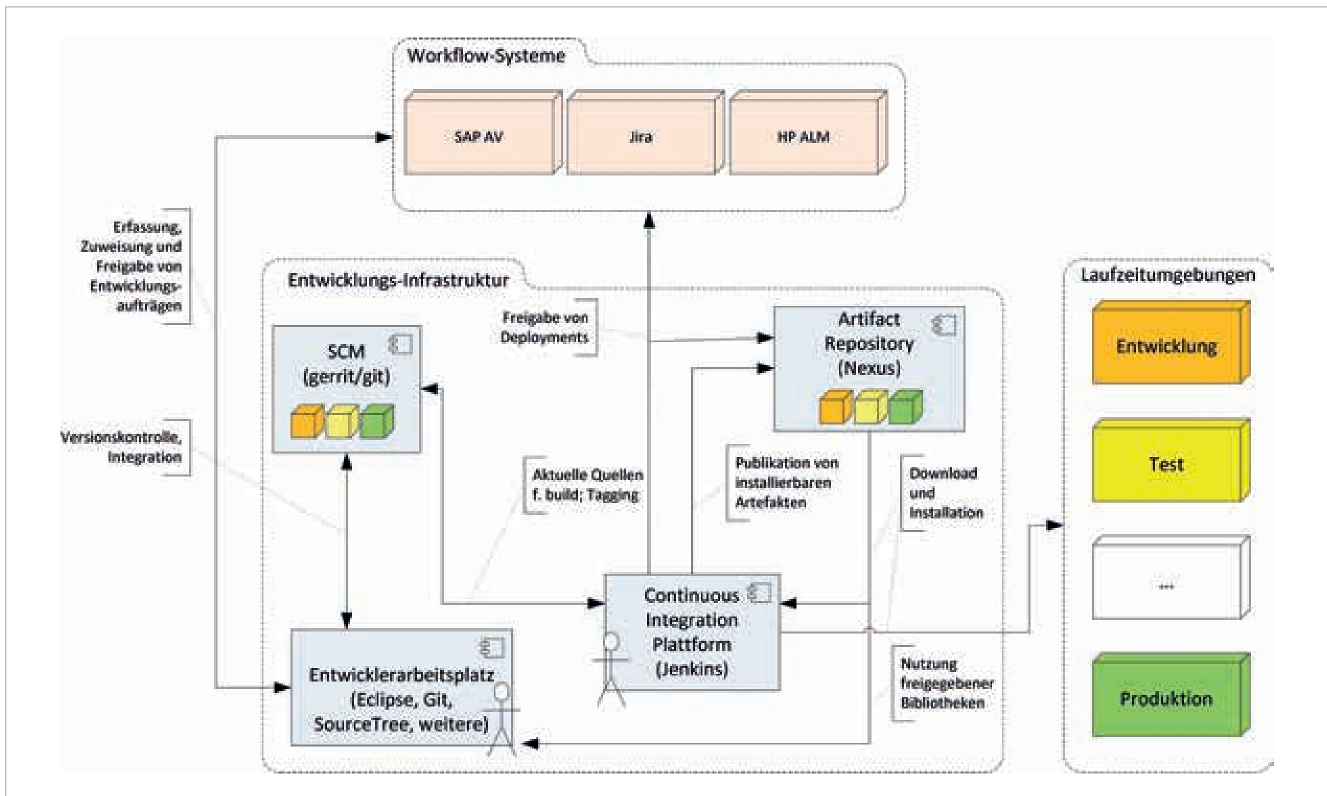


Abbildung 1: Zielbild der Entwicklungsinfrastruktur (Bereiche innerhalb der Einzelsysteme für die jeweiligen Laufzeitumgebungen sind farbig hervorgehoben)

Releases so eindeutig systemübergreifend identifiziert, der Installationsstand jeder Laufzeitumgebung wird unmittelbar auf Binärartefakt- und Source-Code-Ebene nachvollziehbar. Zusätzlich wird jede zu entwickelnde Funktionalität eindeutig identifiziert und mit Auftragsnummern aus den benachbarten Workflow-Systemen (Rational Team Concert, HP Service Manager, SAP AV) verknüpft.

Durch diese Konzepte kann eine Requirement-to-Source-Traceability hergestellt werden. Das bedeutet, dass zu jeder Fachanforderung der entsprechende Source-Code, der diese umsetzt, über Entwicklungsprojekte hinweg einfach gefunden werden kann. Andererseits wird durch die Verknüpfung dieser IDs zu Releases transparent gemacht, welche Funktionalität in welcher Version enthalten ist und in welchem Status (Entwicklung, Test, Produktion) sich diese befindet. Die einheitlichen Versionierungsschemata in den verschiedenen Systemen stellen darüber hinaus sicher, dass historische Stände – beispielsweise zu A-/B-Tests oder als Unterstützung bei Fehleranalysen – auf den unterschiedlichen Laufzeitumgebungen wiederhergestellt werden können und dem Stand des Source-Codes entsprechend erneut identisch erzeugbar sind.

## Versionskontrolle und Parallelentwicklung

In Softwareentwicklungsprojekten nimmt die Versionsverwaltung eine zentrale Rolle ein und hat damit eine Reihe wichtiger Schnittstellen. Das Version Control System (VCS) dient als Quelle für das Build Management. Es muss das angestrebte Release Management technisch manifestieren, ist ein wichtiger Baustein der Team Collaboration und im Rahmen des Projektmanagements eng mit dem Issue Tracking verzahnt. So ist es kein Wunder, dass ein unternehmensweit ausgerolltes VCS möglichst lange unverändert genutzt wird, frei nach dem Prinzip: „Never change a running system." Der Wechsel auf ein modernes Versionskontrollsystem, das den heutigen Ansprüchen der Softwareentwicklung Rechnung trägt – Stichwort „agile Entwicklung" – erscheint oft als hochgradig riskant und außerordentlich aufwendig. Die Beibehaltung des Status quo kann jedoch bedeuten, dass auf betriebswirtschaftliche oder regulatorische Anforderungen nicht mit der notwendigen Dynamik und Effizienz reagiert werden kann. Wettbewerbsnachteile wären die Folge.

So auch bei der KfW. Seit über zehn Jahren wird IBM Rational Synergy (Synergy) in allen Softwareprojekten eingesetzt. Versioniert werden hauptsächlich Java-, C-, C++- und C#-Sources, ebenso wie alle Host-Projekte. Synergy ist ein sehr mächtiges und komplexes Werkzeug. Um die hohe Tool-Komplexität zu kapseln, wurde innerhalb der KfW eine Reihe von Werkzeugen programmiert. Die unterschiedlichen Nutzer können damit zielgerichtet ihre rollenspezifischen Aktivitäten durchführen, ohne Synergy direkt bedienen zu müssen. Das Arbeiten mit Synergy wurde so erleichtert, und Fehler, die durch die hohe Tool-Komplexität entstehen könnten, wurden verhindert.

Die KfW hat schließlich entschieden, Synergy abzulösen und Git als unternehmensweites Versionskontrollsystem einzuführen. Parallel zu Synergy ein weiteres VCS einzuführen, hätte zur Folge gehabt, dass die angestrebte Verschlankung und Standardisierung der Entwicklungsinfrastruktur nicht erreicht worden wären. Im Gegenteil: Zur bestehenden Infrastruktur wäre noch ein weiteres Versionierungswerkzeug hinzugekommen, mit all seinen Schnittstellen.

Git ist ein Produkt aus dem Open-Source-Umfeld und gehört zur Familie der verteilten (distributed) Versionsverwaltungssysteme (DVCS). Die Tool-Evaluierung hat gezeigt, dass mit Git insbesondere Branching, Merging und verteilte Entwicklung optimal adressiert werden können. Die Identifikation von Softwareständen und die restlichen Migrationsprämissen werden eher durch eine adäquate Tool-Nutzung gewährleistet als durch werkzeugspezifische Funktionalitäten.

Eine der zentralen Fragen, die sich im Zuge der Feinkonzeption stellt, ist, ob die gesamte Historie migriert werden soll oder nur ausgewählte Softwarestände. Nochmals zur Erinnerung: Über 20.000 aktive Entwicklungsprojekte mit ca. 65 Millionen Source Lines of Code sind zu migrieren. Und in diesem Mengengerüst sind die historischen Stände noch nicht enthalten.

Wird die gesamte Historie migriert, die bei einigen Projekten zehn Jahre in die Vergangenheit reicht, ist damit zu rechnen, dass der chronologische Aufbau der Historie nicht ohne Probleme automatisiert durchgeführt werden kann. Es drohen manuelle Nacharbeiten, die den Aufwand schnell explodieren lassen können. Ein hohes Risiko, das schwer kalkulierbar ist und den Erfolg des gesamten Projekts gefährdet. Denn die aktuell verwendeten Stände in Entwicklung, Test und Produktion können erst dann migriert werden, wenn alle vorherigen Stände im neuen VCS konsistent verfügbar sind. Aus diesen Überlegungen lässt sich folgende Best Practice ableiten: Je Entwicklungsprojekt werden die letzten zehn produktiven Stände (falls vorhanden), der aktuelle Teststand und der aktuelle Entwicklungsstand migriert. Das Altsystem bleibt für den Zugriff auf die komplette Historie im Read-only-Modus verfügbar. Es gibt zwei Migrationsvarianten:

> Die Anzahl der produktiven Stände, die migriert werden, kann je nach Anforderung nach oben oder unten angepasst werden.

> Kann das Altsystem nicht zur Verfügung gestellt werden, wird die komplette Historie in ein separates Repository migriert und steht im Read-only-Modus zur Verfügung. Die Migration der gesamten Historie in ein separates Repository kann damit abgekoppelt vom eigentlichen Migrationsprojekt durchgeführt werden.

Diese Best Practice wurde bei der KfW so konzipiert, dass neben aktuellem Entwicklungs- und Teststand nur der letzte produktive

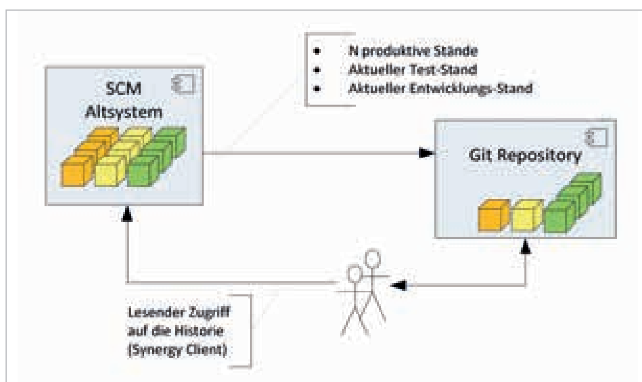


Abbildung 2: Best Practice zur Migration der Historie – Standardvorgehen

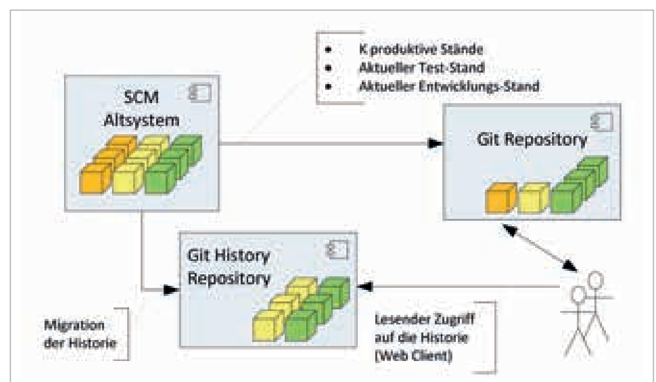


Abbildung 3: Best Practice zur Migration der Historie – Variante (separates Repository)

Stand migriert wurde und das Altsystem in Zukunft lesend zur Verfügung steht. Damit ergibt sich eine deutliche Aufwands- und Risikominimierung. Nichtsdestotrotz verblieben über 60.000 zu migrierende Source-Stände. Eine vollständige Automatisierung war daher unverzichtbar. Diese beinhaltete nicht nur den Export aus dem Altsystem und den Import in das neue VCS, sondern auch die Tests zur Überprüfung der Konsistenz und Fehlerfreiheit. Für die erfolgreiche Migration eines Softwarestands gilt daher:

- Identifikation und Export eines spezifischen Source-Standes aus dem Altsystem
- Import dieses Source-Standes in das neue VCS
- Identifikation und Export des Source-Standes aus dem neuen VCS und Vergleich mit dem ursprünglichen Stand
- Anstoßen eines Build sowohl mit dem Stand aus dem Altsystem als auch mit dem Stand des neuen VCS
- Vergleich, ob die beiden Build-Ergebnisse identisch sind

Damit wird nicht nur sichergestellt, dass alles konsistent migriert wurde, sondern auch gewährleistet, dass jeder Build-Prozess nach der Migration noch genauso funktioniert wie vorher.

Die Migration der Sources nach Git schafft die Basis, den nächsten großen Projektabschnitt anzugehen: das zentrale Build- und Deployment-Management. Die strukturierte Verwaltung des Source-Codes ist die Grundvoraussetzung für eine stabile und dennoch flexible Anwendungslandschaft. Durch Git kann die Softwareentwicklung über das optimal unterstützte Branching und Merging einfach parallelisiert werden. Allein damit ist schon eine Effizienzsteigerung zu erwarten. Wird die Parallelisierung auch in den folgenden Prozessschritten und Systemen weitergeführt, fällt diese Steigerung deutlich höher aus.

## Zentrales Build- und Deployment-Management

Die Gesamtanzahl der Projekte lässt die Zentralisierung der Builds zunächst als hochgradig aufwendig erscheinen. Jedoch ist der überwiegende Teil der Projekte stark strukturiert; etwa 90 Prozent lassen sich jeweils einem von rund 35 Projekttypen zuordnen. Für jeden dieser Typen ist ein eindeutiges Build- und Deployment-Modell definiert. Tatsächlich werden die konkreten Build-Skripte anhand des Typs und weiterer Metainformationen projektspezifisch generiert. Diese Struktur ermöglicht die Handhabung der Projektanzahl mit vertretbarem Aufwand – sei es im Tagesgeschäft oder bei der Migration hin zu einer neuen Landschaft.

Um die Flexibilität der Builds für die kommenden Modernisierungsschritte zu erhöhen, wird diese statische Erzeugung durch ein parametrisierbares Build-Skript pro Projekttyp ersetzt. So ist es möglich, eine Build-Implementierung pro Projekttyp vorzuhalten und zentral zu pflegen. Das Kernstück der zentralen

KFW

Die KfW wurde 1948 gegründet und beschäftigt ca. 5.200 Mitarbeiter. Heute ist die KfW eine der führenden Förderbanken der Welt. Mit ihrer jahrzehntelangen Erfahrung setzt sich die KfW im Auftrag des Bundes und der Länder dafür ein, die wirtschaftlichen, sozialen und ökologischen Lebensbedingungen weltweit zu verbessern. Allein 2012 hat sie dafür ein Fördervolumen von 73,4 Milliarden Euro zur Verfügung gestellt. Davon flossen 40 Prozent in Maßnahmen zum Klima- und Umweltschutz.

Die KfW besitzt keine Filialen und verfügt nicht über Kundeneinlagen. Sie refinanziert ihr Fördergeschäft fast vollständig über die internationalen Kapitalmärkte. Im Jahr 2012 hat sie zu diesem Zweck 78,7 Milliarden Euro aufgenommen. Die Bilanzsumme lag 2012 bei 511,6 Milliarden Euro.

In Deutschland ist die KfW-Bankengruppe mit Standorten in Frankfurt/Main, Berlin, Bonn und Köln vertreten. Weltweit gehören mehr als 70 Büros und Repräsentanzen zu ihrem Netzwerk.

Build- und Deployment-Verwaltung ist eine auf Jenkins basierende Continuous-Integration-Plattform. Dort werden je Projekttyp entsprechende Build- und Deployment-Jobs definiert. Bei der Ausführung eines Jobs müssen benötigte Parameter wie Projektname und Zielumgebung angegeben werden. Diese Parameter werden an das generische Build-Skript für den jeweiligen Projekttyp übergeben. Regelmäßig eingeplante Builds werden schließlich über einen weiteren Build-Job aufgerufen, in dem die o. g. Parameter persistiert sind. Dieses Vorgehen bringt eine Reihe von Vorteilen:

- Einzelne Builds oder Teile davon können flexibel über Jenkins (z. B. über das Build-Flow-Plug-in) miteinander kombiniert werden, ohne die Implementierung der Build-Skripte zu ändern.
- Parametrisierte Builds lassen sich einfacher über externe Schnittstellen aufrufen, da keine spezifischen Jobs je Projekt bekannt sein müssen. Es ist ausreichend, Projektname, Projekttyp und Build-Aufgabe (z. B. Build, Deployment, Analyse) zu kennen und zu übergeben. So wird die Integration mit den umgebenden Workflow-Systemen vereinfacht.
- Die bestehenden Build-Skripte können zu großen Teilen bestehen bleiben; einzig die Möglichkeit der Parametrisierung wird aufgenommen. Das durch diese Modifikation entstehende Risiko, die Funktionalität der Builds zu gefährden, ist weitaus geringer als bei einer vollständigen Re-Implementierung.
- Änderungen an der Implementierung der Build-Skripte sind sofort für alle Projekte des entsprechenden Typs aktiviert, es müssen keine neuen, projektspezifischen Build-Skripte erzeugt werden.
- Berechtigungen für Builds und Deployments können an zentraler Stelle in Jenkins vergeben werden. Die Integration mit AD/LDAP ermöglicht die Nutzung unternehmensweit definierter Benutzer und Rollen.
- Die Protokollierung kann zentral gesteuert werden. Das ist beispielweise bei Aktionen wie Produktiv-Deployments relevant, die revisionssicher dokumentiert werden müssen.

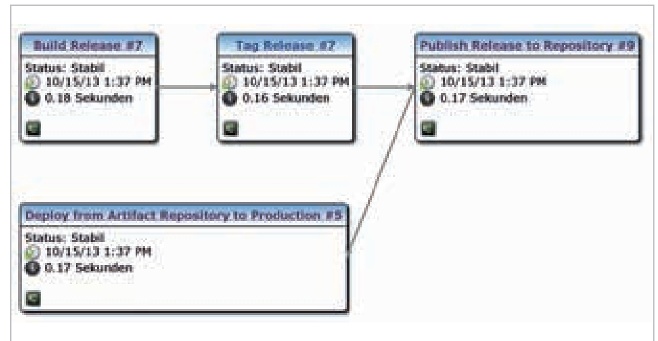


Abbildung 4: Release-Durchführung (beispielhaft)

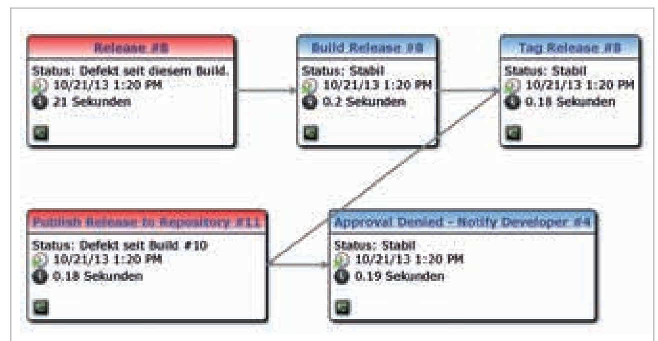


Abbildung 5: Release-Durchführung mit verweigertem Approval (beispielhaft)

Die aus den Builds entstehenden Softwareartefakte werden neben der Installation auf den Laufzeitumgebungen auch in einem zentralen Repository abgelegt. Dieses Repository dient der Historisierung der Softwarestände auf den Laufzeitumgebungen und der Abbildung von Abhängigkeiten zwischen Einzelprojekten.

## Abhängigkeitsmanagement

Zwischen Einzelprojekten besteht eine Vielzahl von logischen Abhängigkeiten, z. B. zwischen Batch-Jobs, die zwar in voneinander getrennten Projekten entwickelt werden, aber in der zeitlichen Einplanung und in Bezug auf Ein- und Ausgabedaten aufeinander abgestimmt sein müssen. Abhängigkeiten können zwischen verschiedenen Versionen der gleichen Komponenten unterschiedlich sein. Aktuell sind die Abhängigkeiten nicht als Metadaten erfasst und auch nicht maschinell auswertbar. Dies bedeutet

einerseits, dass Einzelkomponenten nicht risikolos zwischen Systemumgebungen wie Test und Produktion transportiert werden können, da nicht ohne weitgehende Tests sichergestellt werden kann, dass alle benötigten Abhängigkeiten dort in der entsprechenden Version vorhanden sind. Andererseits müssen beim Build der entsprechenden Komponenten alle potenziellen Abhängigkeiten ebenfalls gebaut werden, um einen konsistenten Stand zu erstellen. Das führt zu einer langen Laufzeit der Builds, was wiederum die Reaktionszeit auf kritische Änderungen/Bug-Fixes sowie den Hardwarebedarf für die Build-Server erhöht.

Weiterhin gilt auch, dass durch unbekannte Abhängigkeiten nicht nur die Deployments neuer Versionen, sondern ebenso die Wiederherstellung eines Last-known-good-Status mit großen Risiken behaftet sind. Wird beispielsweise in einer neuen Version einer Komponente ein Fehler festgestellt, so kann nicht ohne Weiteres die vorhergehende Version wieder eingespielt werden, da bereits weitere Komponenten von der neuen Version abhängig sein könnten. Diese Problematik wird weiter verschärft, je komplexer und tiefer verschachtelt transitive Abhängigkeiten sind.

Das Vorhandensein unbekannter Abhängigkeiten macht auch die Trennung von Test- und Produktionsumgebungen unsicher. So kann beispielsweise Komponente A, die auf Test in Version 1.1 und in Produktion erst in Version 1.0 vorhanden ist, Seiteneffekte auf Komponente B haben. Wird jetzt eine neue Version von Komponente B vor Komponente A produktiv genommen, so entsteht der Fall, dass die Kombination A(v1.1)+B getestet wurde, in Produktion aber A(v1.0)+B deployed ist.

Solange der Betrieb durch ein gut eingespieltes Kernteam erfolgt, ist es möglich, dass keine ernsthaften Probleme auftreten – man „weiß ja", welche Komponenten Seiteneffekte hervorrufen. Dass dieser Zustand weder verlässlich ist noch aktuellen Risikomanagementvorgaben entspricht, liegt auf der Hand. Diesen Zustand auf einmal zu bereinigen, ist aus mehreren Gründen nicht möglich: Erstens bindet das Erfassen aller Abhängigkeiten nahezu alle Entwicklungsbereiche für einen langen Zeitraum. Zweitens kann während der Bereinigung keine Entwicklung stattfinden, und drittens müssen alle Build-Verfahren auf einmal umgestellt werden.

Das in der KfW entwickelte und angewandte Vorgehen ermöglicht es, durch einen iterativen Ansatz des Abhängigkeitsmanagements für neue Projekte zu nutzen, und macht existierende Projekte im Rahmen des Abhängigkeitsmanagements adressierbar. Dazu werden während der SCM-Migration die Projektkoordinaten Gruppe, Artefakt-ID und Version als automatisiert verarbeitbare Metadaten in jedem Projekt abgelegt. Diese Metadaten können aus Projekttyp und Projektname erzeugt werden. Die Version wird für alle Artefakte initial auf 1.0.0 gesetzt. Weiterhin wird für jeden Projekttyp definiert, wie Distributionen eines Projekts zusammengestellt werden. Mit diesen Informationen ist es möglich, Build-Artefakte in einem Artefact Repository – in vorliegenden Projekt wurde Sonatype Nexus mit Apache Maven verwendet – zu versionieren. Zwar wird durch diesen Schritt noch kein Abhängigkeitsmanagement eingeführt, die grundlegenden Voraussetzungen dafür werden aber geschaffen:

- Projekte sind in einer bestimmten Version eindeutig adressierbar.
- Ein Repository, in dem versionierte Artefakte abgelegt werden können, ist vorhanden und kann genutzt werden, um Abhängigkeiten aufzulösen.
- Ein Build-Verfahren, das das Abhängigkeitsmanagement unterstützt, ist etabliert.

Neue Projekte nutzen diese Möglichkeit unmittelbar; auf bestehende Projekte hat das Vorgehen zunächst keine Auswirkung. So kann das Entwicklungsteam eines Legacy-Projekts selbst im Einzelfall entscheiden, ob und wenn ja wann die formelle Abbildung der Abhängigkeiten erfolgen soll.

Um die Nachvollziehbarkeit von Deployments auf Laufzeitumgebungen wie Test, QA oder Produktion zu gewährleisten, wird im Artefact Repository je Umgebung eine Stage definiert. „Promotions" von einer Stage zur anderen, also z. B. der Transport von Test auf Produktion, können so mit entsprechenden Berechtigungen oder ei-

nem Freigabe-Workflow versehen werden. Zukünftige Deployments beziehen die zu installierenden Artefakte immer aus diesem Repository. So wird sichergestellt, dass alle Artefakte auch nur auf die Umgebungen ausgerollt werden können, für die eine Freigabe vorliegt.

## Fazit und Ausblick

Für ein Migrationsprojekt dieser Größe, bei dem zentrale Werkzeuge wie das VCS ausgetauscht, neue Prozesse und neue Werkzeuge (Dependency Management) eingeführt sowie bestehende Build-Prozesse durch Modifikation der Build-Skripte verändert werden, gibt es einen Schlüsselfaktor: das Scope-Management. Es entscheidet über Erfolg oder Misserfolg des Modernisierungsprojekts. Von zentraler Bedeutung sind die klare Fokussierung auf unbedingt notwendige Aufgaben sowie die Identifikation und der Ausschluss verzichtbarer Aktivitäten.

Bei der Migration der Sources ist zu entscheiden, ob tatsächlich die gesamte Historie in das zukünftige VCS übernommen werden muss. Durch die Einbeziehung der Entwicklungsprojekte in den Entscheidungsprozess kann die ganze Tragweite dieser Entscheidung transparent gemacht werden. Dann sind Entwickler auch eher bereit, während einer gewissen Übergangszeit eine umständliche Historienrecherche zu akzeptieren.

Der bei der Modernisierung der Builds und Deployments gewählte Ansatz, möglichst viele Teile der bestehenden Skripte beizubehalten, gleichzeitig aber neue Möglichkeiten der Automatisierung bereitzustellen, minimiert Risiken und die Last auf den Wartungsteams. Artifact Repositories und die Referenzierbarkeit der Projekte schaffen die Basis für die Einführung eines Dependency Managements – und das ohne zu versuchen, alle Abhängigkeiten in einem Big Bang formal zu erfassen und beim Build zu berücksichtigen.

Generelles Ziel ist es, eine tragfähige Basis zu schaffen. Neue Projekte können sofort von den Vorteilen der neuen Entwicklungsinfrastruktur profitieren. Bestehende Projekte sind in der Lage, eigenverantwortlich notwendige Anpassungen vorzunehmen, um in den Genuss dieser Vorteile zu kommen. Der Zeitpunkt dieser Anpassungen ist dabei frei wählbar.

Ein weiterer Vorteil der neuen Infrastruktur ist die mögliche Messung von Source-Code-Metriken. Über die etablierte CI-Plattform können automatisierte Messungen nach Projekttyp zentral eingerichtet ausgeführt werden. Auch die Integration der Entwicklungsumgebung mit der weiteren Build-Infrastruktur bietet Vorteile. Die bereits eingeführten Werkzeuge Eclipse als IDE und Jira als Issue-Tracking-System lassen sich über das Eclipse-Mylyn-Plug-in verknüpfen. Der Entwickler kann so alle Issue-Tracking-Tätigkeiten ausführen, ohne seine Entwicklungsumgebung verlassen zu müssen. Dies wird insbesondere interessant, wenn die Prozessablaufsteuerung zukünftig stärker in Jira abgewickelt wird.

Die Nutzung von Git als verteiltes Versionskontrollsystem bietet zudem neue Möglichkeiten, die Zusammenarbeit zwischen externen Mitarbeitern in der Softwareentwicklung elegant abzubilden. So wird bei der KfW nicht nur der aktuelle Stand der Softwareentwicklung modernisiert und effizienter gestaltet. Die neue Entwicklungsinfrastruktur legt auch einen soliden Grundstein für weitere Optimierungsmaßnahmen und sorgt so für eine hohe Zukunftssicherheit.

**Autoren**

**Jürgen Biebl**
Lead IT Consultant, msg Applied Technology Research
> +49 (0) 89 / 96101 - 2093
> juergen.biebl@msg-systems.com

**Wolfgang Werner**
Lead IT Architect, CoC IT-Architekturen, msgGillardon AG
> +49 (0) 89 / 943011 - 1854
> wolfgang.werner@msg-gillardon.de