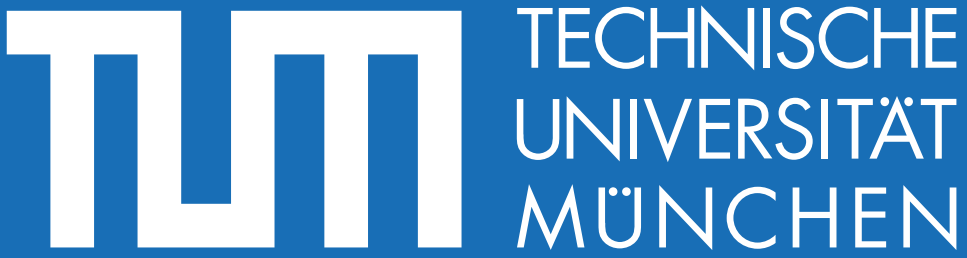
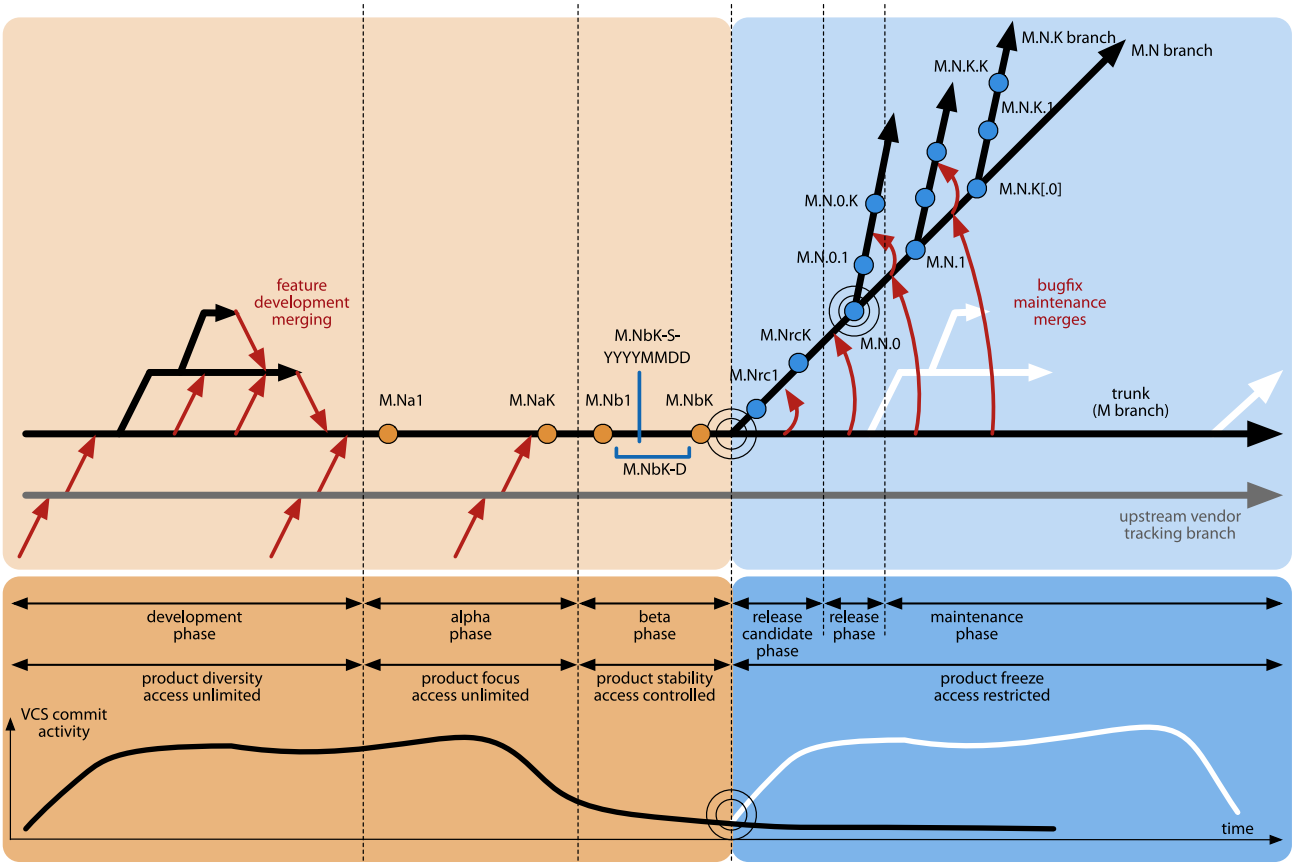


Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

Version Control Architecture



Beim **Version Control** geht es um die konsequente Versionierung aller Quellcode-Artefakte in einem **Version Control System (VCS)**, wie Subversion oder Git. Man nutzt im VCS fünf verschiedene Arten von **Branches**: **Vendor Branch**, **Trunk** (bzw. **M Branch**), **Feature Branch**, **Release Branch** (bzw. **M.N Branch**) und **Hotfix Branch** (bzw. **M.N.K Branch**).

Der **Vendor Branch** bzw. **Upstream Vendor Tracking Branch** hält die unveränderten Kopien von fremden ("third-party") Quellen. Seine Daten werden regelmäßig auf den Trunk integriert, um dort ggf. verändert zu werden. Er existiert so lange, wie das Produkt selbst. Ein klassischer Anwendungsfall besteht aus fremden Bibliotheken, die verändert werden müssen.

Der **Trunk** bzw. **M Branch** (bei Subversion `trunk` und bei Git `master` genannt) hält zu jedem Zeitpunkt den aktuell integrierten Stand der nächsten Hauptversion ("M") des Produkts. Er existiert so lange, wie das Produkt selbst.

Ein **Feature Branch** wird jeweils vom Trunk (oder ggf. sogar von einem anderen Feature Development Branch) abgezweigt. Er enthält die Änderungen während der Entwicklung eines neuen Features, falls dieses umfangreicher ist, über einen längeren Zeitraum oder von mehr als einer Person entwickelt werden muss. Er wird regelmäßig mit den Änderungen des Trunk versorgt (bei Git durch "Merging" oder "Rebasing"), am Ende der Feature-Entwicklung auf den Trunk integriert und danach üblicherweise wieder gelöscht.

Ein **Release Branch** bzw. **M.N Branch** wird jeweils vom Trunk abgezweigt, üblicherweise nach der Beta-Phase ("M.NbK") und vor der Release-Candidate-Phase ("M.Nrc1"). Er erhält regelmäßig vom Trunk Bugfixes durch sogenanntes "Cherry Picking" von Änderungen. Auf ihm werden die Release-Versionen "M.N.K" erstellt. Der "M.N Branch" existiert so lange, wie die Version "M.N" des Produkts noch nicht "end-of-life" ist.

Ein **Hotfix Branch** bzw. **M.N.K Branch** wird jeweils vom "M.N branch" bei Bedarf abgezweigt, üblicherweise direkt vor der Notwendigkeit eines ersten **Hotfixes** "M.N.K.1". Er existiert so lange, wie **Hotfixes** für die Release-Version "M.N.K" bereitgestellt werden müssen.

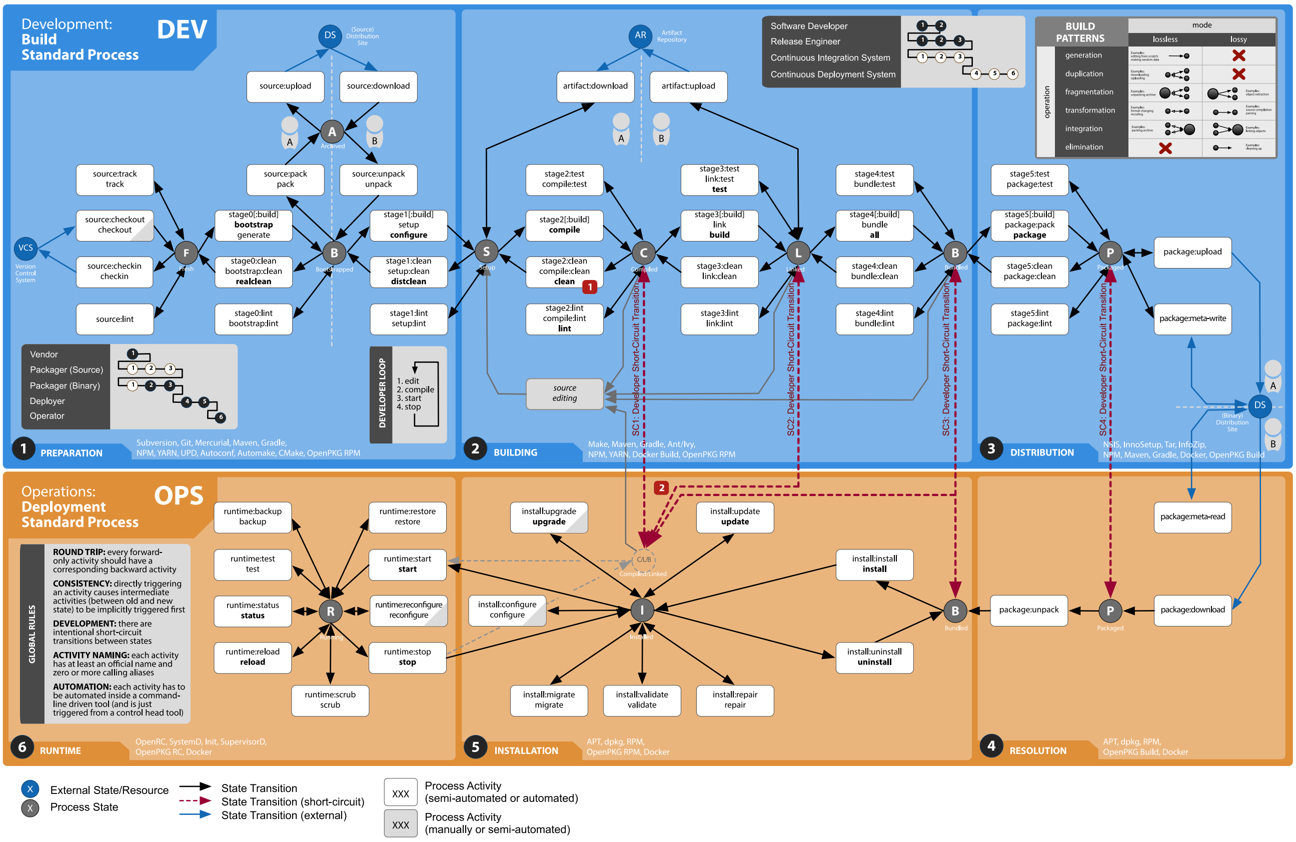
Fragen

- Welche fünf **Arten von Branches** kennt man in einem **Version Control System**"?
- Welcher **Art von Branch** in einem **Version Control System** wird üblicherweise nach der erfolgreichen Integration wieder gelöscht?
- Zwischen welchen beiden zeitlichen **Phasen** des Release-Managements wird in einem **Version Control System** üblicherweise der **Release-Branch** vom **Trunk** abgezweigt?

ARCHITECTURE FUNDAMENTALS

Assembly Process Architecture

TECHNISCHE UNIVERSITÄT MÜNCHEN



Bei der **Assembly Process Architecture** wird eine **DevOps Pipeline** genutzt, um eine Version eines Software-Produkts automatisiert von den Quellen im **Version Control System** bis zur laufenden Instanz im Betrieb zu bringen.

Der Dev(elopment)-Anteil der **DevOps Pipeline**, der sog. **Build Standard Process**, wird üblicherweise über **Continuous Integration (CI)** automatisiert. Der Op(eration)s-Anteil der **DevOps Pipeline**, der sog. **Deployment Standard Process**, wird üblicherweise über **Continuous Deployment (CD)** automatisiert. Alle Aktivitäten der **DevOps Pipeline** werden über spezialisierte Build- und Deployment-Werkzeuge automatisiert und diese Aktivitäten in einem CI/CD-System nach jeder Änderung im **Version Control System** automatisch ausgeführt.

Der **Assembly Process** sollte insbesondere einen sinnvollen **Round Trip** ermöglichen, indem der Prozess als Zustandsautomat verstanden wird und zu allen (Vorwärts-)Aktivitäten sinnvolle zugehörige Rückwärts-Aktivitäten existieren. Wenn von außen ein bestimmter Ziel-Zustand gefordert wird, werden implizit alle Zwischen-Aktivitäten zwischen dem Quell- und dem Ziel-Zustand ausgeführt.

Für die Produktivität bzw. die möglichst effiziente "Developer Loop" in der Software Entwicklung sollten bis zu vier **Short-Circuit Transitions** unterstützt werden, bei dem eine bewusste "Abkürzung" vom **Build Standard Process** zum **Deployment Standard Process** vorgenommen werden kann.

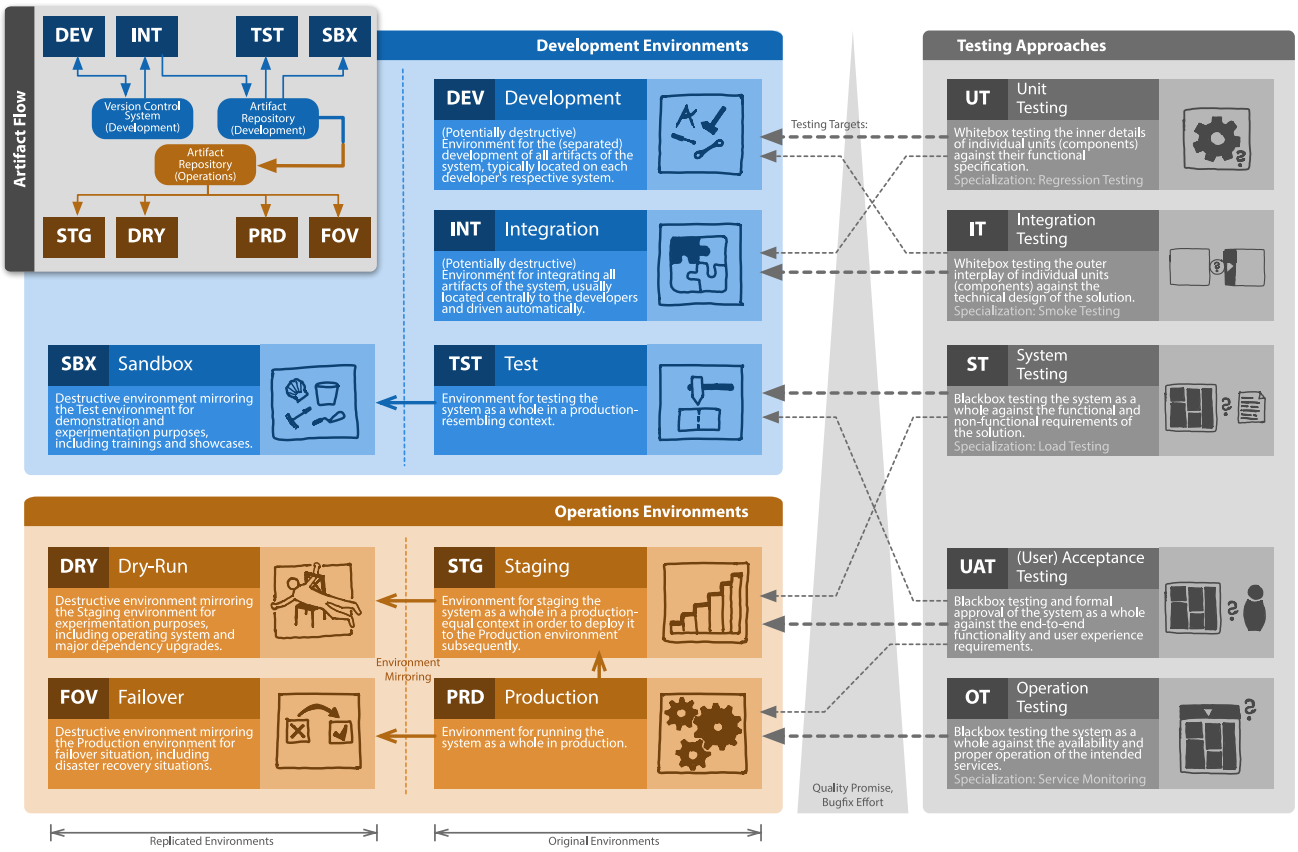
Die **Assembly Process Architecture** nutzt vier unterschiedliche Arten von externen Speicherorten: das **Version Control System** speichert die "nackten" Quell-Dateien, die **(Source) Distribution Site** speichert die für den Build-Prozess vorbereitete "Source Distribution" der Quell-Dateien, das **Artifact Repository** speichert wiederverwendete Build-Artefakte (vor allem Bibliotheken) und die **(Binary) Distribution Site** speichert die für das Deployment vorgesehene "Binary Distribution" des Produkt.

Die ersten drei Speicherorte dienen vor allem der Übergabe von Daten zwischen unterschiedlichen Personen. Letzterer dient vor allem der Übergabe von Daten zwischen Dev(elopment) und Op(eration)s.

Fragen

- Wieso sollten in der **Assembly Process Architecture** bis zu vier sogenannte **Short-Circuit Transitions** zur Abkürzung vom **Build Standard Process** zum **Deployment Standard Process** unterstützt werden?
- Welche vier **Arten von externen Speicherorten** kennt die **Assembly Process Architecture**?

Environments & Quality Assurance



In der Praxis unterscheidet man üblicherweise 4 **Development Environments**, 4 **Operations Environments** und 5 zugehörige **Testing Approaches**.

Die **Development Environments** sind **Development** (z.B. der Rechner des Entwicklers), **Integration** (z.B. ein zentraler Server mit Continuous Integration (CI) System), **Test** (z.B. ein zentraler Server, der dem **Production** Environment ähnelt, aber keine Kopie davon ist) und ggf. **Sandbox** (z.B. ein Server, der als Kopie von **Test** für Schulungen und Show-Cases zur Verfügung steht).

Die **Operations Environments** sind **Staging** (eine Kopie von **Production**), **Dry-Run** (eine 1:1 Kopie von **Staging**), **Production** (die reguläre Produktionsumgebung) und **Failover** (eine 1:1 Kopie von **Production**).

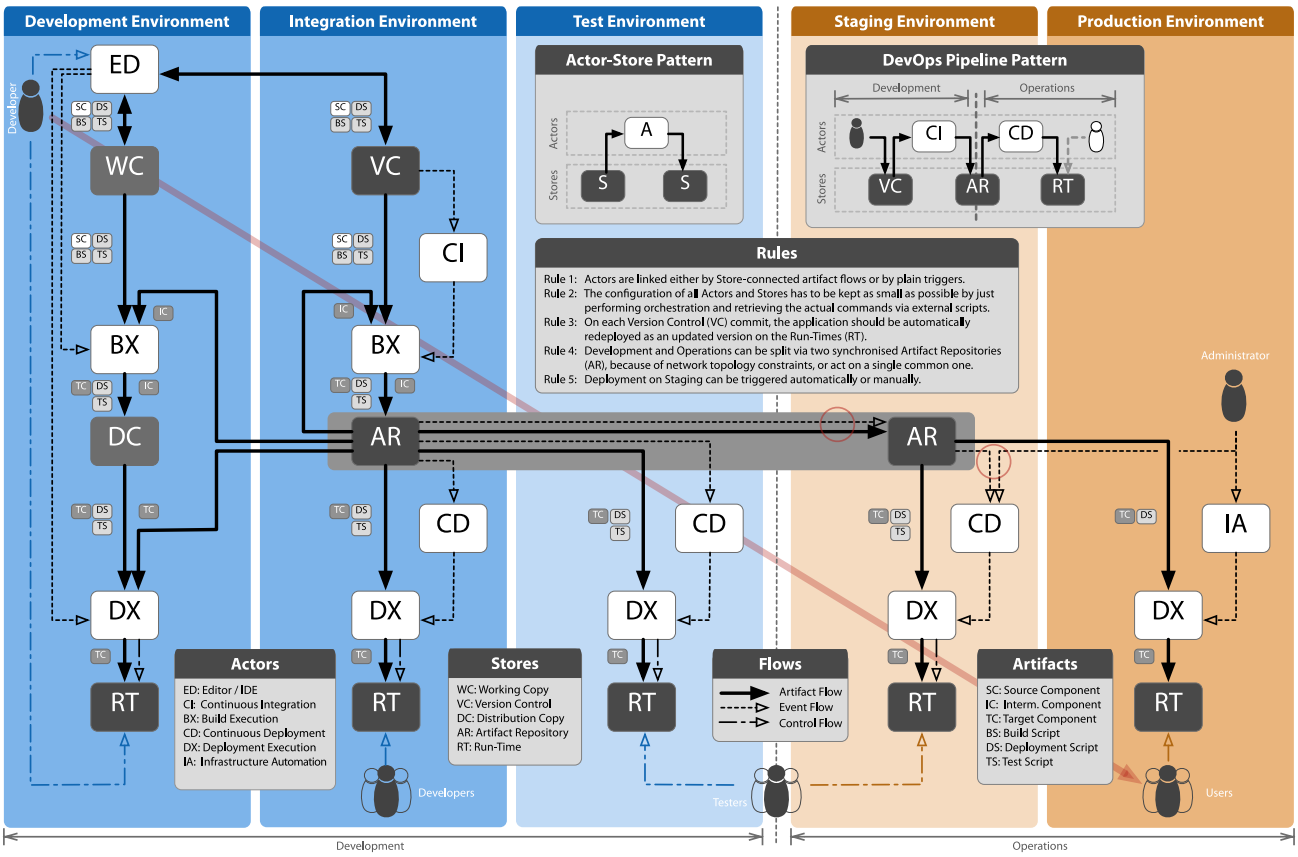
Die **Testing Approaches** sind **Unit Testing** für die Funktionalität von Komponenten auf dem **Development** (und ggf. **Integration**) Environment, **Integration Testing** für das Zusammenspiel von Komponenten auf **Integration** (und ggf. **Development**) Environment, **System Testing** für die funktionalen und nicht-funktionalen Eigenschaften des Gesamtsystems auf dem **Test** (und ggf. **Staging**) Environment, **(User) Acceptance Testing** für die "End-To-End" Funktionalität des Gesamtsystems auf **Staging** (oder ggf. **Production**) Environment und **Operation Testing** für die Verfügbarkeit des Gesamtsystems auf dem **Production** Environment.

Als Übergabepunkte der Artefakte zwischen den 8 Environments dienen ein **Version Control System** und ein **Artifact Repository** auf Seite der **Development Environments** und ein zugehöriges **Artifact Repository** auf Seite der **Operations Environments**.

Fragen

- Wie heißt die Kopie des **Production** Environments, auf dem u.a. **(User) Acceptance Testing** durchgeführt wird?

DevOps Toolchain



Um einen DevOps-Ansatz auch Werkzeug-seitig zu unterstützen, bietet sich eine **DevOps Toolchain** an. Diese basiert im Kern auf einem Muster, bei dem ein **Actor** zwischen jeweils zwei **Stores** agiert, indem er ein oder mehrere **Artifacts** als Eingabe von einem **Store** liest, diese verarbeitet und als Ausgabe ein oder mehrere **Artifacts** in einen anderen **Store** schreibt. Zusätzlich können **Actors** über **Events** angestoßen werden oder direkt von verschiedenen Personenkreisen durch Interaktionen gesteuert werden.

Dieses Grundmuster wird nun zu einem **DevOps Pipeline Pattern** kombiniert, bei dem bei jedem "Commit" eines **Developer** in das **Version Control** (VC) System ein automatischer Übersetzungs- und Integrations-Prozess einer Anwendung in einem **Continuous Integration** (CI) System angestoßen wird. Dessen Ergebnisse werden in einem **Artifact Repository** (AR) zentral aufgesammelt, wodurch wiederum ein automatischer Installations-Prozess in einem **Continuous Deployment** (CD) System angestoßen wird. Dessen Ergebnis ist die installierte Anwendung auf einem **Run-Time** (RT) System, auf welches durch **Tester** und **User** zugegriffen werden kann.

Logisch gehören die Systeme VC und CI zum Bereich **Development**, während die Systeme CD und RT zum Bereich **Operations** gehören. Das System AR wird dagegen von beiden Bereichen als gemeinsamer Übergabepunkt verwendet. Da in der Praxis die **DevOps Toolchain** nicht in einem einzigen **Environment** existiert, sondern üblicherweise auf die logisch (oder sogar physikalisch) getrennten **Environments Development**, **Integration**, **Test**, **Staging** und **Production** aufgeteilt ist, existieren einige Systeme mehrfach.

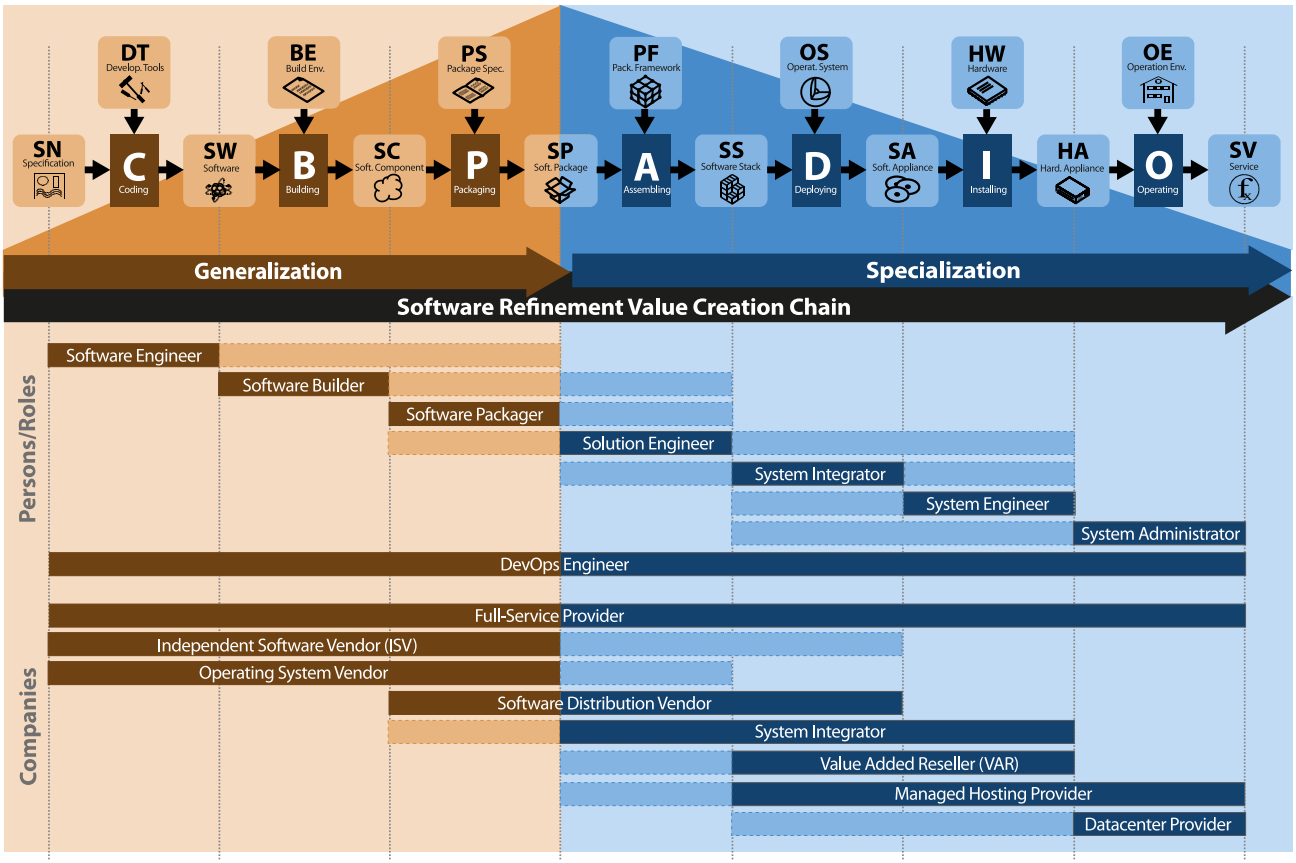
Bei den **Artifacts** wird einerseits zwischen **Source Component** (`foo.java`), **Intermediate Component** (`foo.jar`) und **Target Component** (`foo.exe`) und andererseits zwischen **Build Script** (`foo.make`), **Deployment Script** (`foo.spec`) und **Test Script** (`foo-test.java`) unterschieden.

Beachte absichtliche Sonderfälle: Das **Development Environment** ist unterschiedlich, da es eine schnelle "Edit-Build-Install-Start-Stop"-Schleife unterstützt. Das AR-System kann 1 oder 2 Mal existieren, abhängig davon, wie stark Development und Operations miteinander verzahnt sind. Das Deployment im **Production Environment** sollte üblicherweise manuell über ein **Infrastructure Automation** System angestoßen werden.

Fragen

- Welche beiden **Actor**-Systeme steuern beim **DevOps Pipeline Pattern** den automatisierten Integrations- und Installations-Prozess?

Software Refinement Process



Software-Entwicklung ist prinzipiell ein Verfeinerungsprozess ("Refinement"), bei dem von der **Specification** bis zum **Service** verschiedene Schritte vollzogen werden, welche eine Art Wertschöpfungskette ("Value Creation Chain") darstellen.

In jedem Schritt der Wertschöpfungskette wird das bisherige Artefakt mit Hilfe von hinzugefügten Mitteln "verfeinert". Die ersten Schritte verallgemeinern zuerst die sehr konkrete **Specification** hin zum maximal wiederverwendbaren **Software Package**. Danach spezialisieren die folgenden Schritte das **Software Package** wieder bis zum konkreten **Service**.

Die Wertschöpfungskette wird von verschiedenen Personenkreisen bzw. Rollen und Firmen unterstützt.

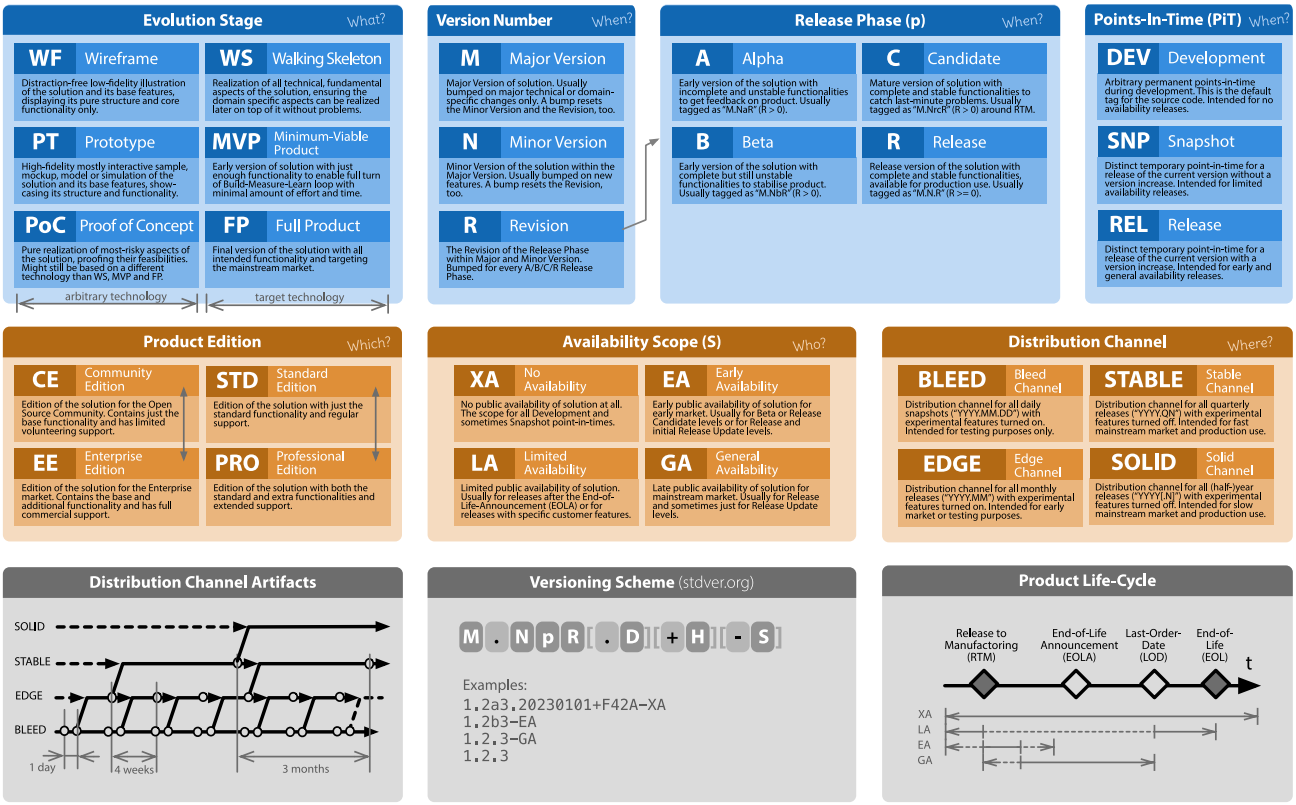
Fragen

- Software-Entwicklung kann als Wertschöpfungskette verstanden werden. Welches Artefakt ist das maximal wiederverwendbare?

ARCHITECTURE FUNDAMENTALS

Software Release Management

TECHNISCHE UNIVERSITÄT MÜNCHEN



Beim **Software Release Management** werden **Releases** von Software-Produkten über 7 Dimensionen gesteuert, welche über ein wohl-definiertes **Version Schema** das Release identifizieren.

Bei der Dimension **Evolution Stages** geht es um die Art und Ausbaustufe des Produkts. Man unterscheidet zwischen den (teilweise noch nicht auf der späteren Technologie basierenden) Vor-Stufen **Wireframe**, **Prototype** und **Proof of Concept** und den (auf der späteren Ziel-Technologie basierenden) produktionsreifen Stufen **Walking Skeleton**, **Minimum-Viable Product** und **Full Product**.

Bei der Dimension **Version Number** wird das Produkt über drei Zahlen identifiziert: **Major Version**, **Minor Version** und **Revision**. Die ersten beiden beziehen sich inhaltlich auf das Produkt. Letztere bezieht sich auf den jeweiligen **Maturity Level** des Produkts. Die Dimension **Maturity Levels** selbst legt die Reife des Produkts innerhalb **Major/Minor Version** fest: **Alpha** (a, unvollständig, instabil), **Beta** (b, vollständig, instabil), **Release Candidate** (rc, vollständig, stabil), **Release** und **Release Update**.

Die Dimension **Points-In-Time** sagt aus, ob das aktuelle Release den Stand **Development** besitzt (so wie das Produkt im VCS liegt), ob es sich um einen speziellen **Snapshot** handelt (z.B. für extrem zeitkritische Hotfixes oder Early Feedbacks) oder ob es sich um eine ganz normale **Release** handelt.

Die Dimension **Product Editions** legt die Ausgabe bzw. Variante des Produkts fest: hier wird üblicherweise **Community/Enterprise Edition** (mit dem Fokus auf den Zielmarkt) oder **Standard/Professional Edition** (mit dem Fokus auf den Funktionsumfang) unterschieden.

Die Dimension **Availability Scopes** legt fest, für wen das Release des Produkts verfügbar ist: **No Availability** (nur für den Hersteller intern), **Limited Availability** (für Spezialsituationen), **Early Availability** (für "Early Adopters" unter den Kunden) und **General Availability** (für alle Kunden). Üblicherweise werden die Availability Scopes für konkrete **Maturity Level** genutzt, es besteht aber kein notwendiger Zusammenhang.

Die Dimension **Distribution Channels** legt fest, über welche Kanäle ein Release verfügbar ist: **Bleed Channel** (für "Visionaries" und "Die Hards"), **Edge Channel** (für "Early Adopters"), **Stable Channel** und **Solid Channel** (für alle Kunden). In jedem Fall sollten die **Distribution Channels** direkt mit den Branches des VCS gekoppelt sein.

Fragen

- Bei welchem **Maturity Level** im **Software Release Management** hat man eine noch unvollständige und instabile Funktionalität?
- Bei welchem **Maturity Level** im **Software Release Management** hat man eine bereits vollständige aber üblicherweise noch instabile Funktionalität?