NERZ e.V.
Nutzer der einheitlichen Rechnerzentralensoftware

# Technische Dokumentation

## Konzept zum Konfigurationsmanagement


**Autoren**
Dipl.-Ing. H. C. Kniß

**Ersteller**
FTB des NERZ e. V.
An der Krautwiese 37
53783 Eitorf



**Version:** **2.0** **Stand:** 01.09.2017 **Status:** akzeptiert

# 0 Allgemeines

## 0.1 Lizenzen



### 0.1.1 Ursprüngliche Dokumente und Autoren

Dieses Dokument basiert auf den Inhalten folgender Dokumente (und ggf. Vorgängerversionen):

- TechDoc_RandbedinungenZurEinfuehrungEinesVersionsverwaltungsprogramms_FREI_V2.0_D2014-05-15.docx
- TechDok_UeberblickEinfuehrungEinesVersionsverwaltungsprogramms_FREI_V1.0_D2014-12-04.docx

Beteiligte Autoren an den Vorgängerdokumenten:

- Dipl.-Ing. H. C. Kniß, inovat

## 0.2 Verteiler

| Organisationseinheit | Name | Anzahl Kopien | Vermerk |
|---|---|---|---|
| NERZ e.V. | | 1 | |

*Tabelle 0-1: Dokumentenverteiler*

## 0.3 Änderungsübersicht

| Version | Datum | Kapitel | Bemerkungen | Bearbeiter |
|---|---|---|---|---|
| 1.0 | 20.02.2017 | | Erstellung | Dip.-Ing. H. C. Kniß (FTB) |
| 2.0 | 01.09.2017 | | Einarbeitung der Änderungen auf Basis der Abstimmungen in den Fach-TelKo's und den Anmerkungen der Mitglieder | Dip.-Ing. H. C. Kniß (FTB) |

*Tabelle 0-2: Änderungsübersicht*

## 0.4 Inhaltsverzeichnis

**0 Allgemeines 2**

**0.1 Lizenzen 2**

0.1.1 Ursprüngliche Dokumente und Autoren....2

**0.2 Verteiler 2**

**0.3 Änderungsübersicht 2**

**0.4 Inhaltsverzeichnis 3**

**0.5 Abkürzungsverzeichnis 5**

**0.6 Definitionen 5**

**0.7 Referenzierte Dokumente / URLs 6**

**0.8 Abbildungsverzeichnis 6**

**0.9 Tabellenverzeichnis 6**

**1 Zweck des Dokuments 7**

**2 Zielsetzung / Randbedingungen 7**

**2.1 Beschreibung und Festlegung der Ziele 7**

**2.2 Randbedingungen 8**

**3 Begriffsdefinitionen 8**

**4 Lösungsskizze 9**

**5 Prozesse – Aktivitäten 11**

**5.1 Prozess „Neues Teilprodukt erstellen“ 12**

**5.2 Prozess „Bestehende Teilprodukte ändern“ 12**

**5.3 Prozess „Abnahme AG“ 13**

**5.4 Prozess „(Teil-)Produkt übernehmen“ 13**

**5.5 Prozess „Produkt prüfen“ 13**

**5.6 Prozess „Übernahme GIT“ 14**

**5.7 Prozess „Produkte erzeugen“ 14**

**5.8 Prozess „Produkte publizieren“ 14**

**6 Technische Umsetzung 15**

**6.1 Build Tool 15**

6.1.1 Maven....15

6.1.2 Gradle....17

6.1.3 Einsatzmöglichkeiten....19

**6.2 GIT Workflow 19**

**6.3 Server 20**

6.3.1 Versionsverwaltungssystem / Server für Versionsverwaltungssystem....21

6.3.2 Dependancy Repository Server (Server zur Verwaltung der Abhängigkeiten)....22

6.3.3 Produktserver ..... 22
**6.4 Eigenbetriebene Server versus externe Server 23**
**7 Struktur der Repositories 23**
**7.1 SWE 23**
**7.2 Plug-in 24**
**7.3 KV 24**
**7.4 Spezifikationen (Dokumentation) 25**
**8 Festlegungen zur Versionierung von Teilprodukten 28**
**8.1 Quellcode 28**
**8.2 Spezifikationen (Dokumentation) 28**
**8.3 Distributionspakete 29**
**8.4 Konfigurationsverantwortliche (KV) 29**
**8.5 Konfigurationsbereiche (KB) 29**
**8.6 Datenkatalog 30**
**9 Bereits getroffene Festlegungen / Nächste Schritte 30**
**10 Anhang „Abbildungen“ 30**

## 0.5 Abkürzungsverzeichnis

Siehe [AbkBSVRZ].

Darüber hinaus werden folgende Abkürzungen verwendet:

| | |
|---|---|
| BSVRZ | Basis System VRZ |
| ERZ | Einheitliche Rechnerzentralensoftware |
| KB | Konfigurationsbereich einer Konfiguration |
| KV | Konfigurationsverantwortlicher eines Teils einer Konfiguration |
| KM / KMS | Konfigurationsmanagement / Konfigurationsmanagementsystem |
| NERZ | Nutzer der ERZ, siehe auch www.nerz-ev.de |
| PD | Produkt-Definition |
| POM | Projekt Objekt Modell |

## 0.6 Definitionen

Siehe [GlossarBSVRZ].

Darüber hinaus werden folgende Definitionen verwendet:

| | |
|---|---|
| Eingangsprodukte | Resourcen wie z. B. der Sourcecode, die Spezifikationsdokumente, die Konfigurationsdaten der KV etc., die im Versionsverwaltungsprogramm (GIT) verwaltet werden. |
| Ausgangsprodukte | Produkte, die über die in einer PD (siehe nächste Definition) gemachten Definitionen aus Eingangsprodukten erzeugt werden (z. B. Distributionspakete, Updateseiten für Plug-in, Datenkataloge etc.) |
| PD | Produkt-Definition. Eine PD für ein Produkt in einer bestimmten Version (z. B. KExTLS in der Version 3.7.2) beschreibt dabei, aus welchen Eingangsprodukten einer bestimmten Version sich das Ausgangsproduktzusammensetzt (erzeugt wird) und welche Abhängigkeiten zu anderen Produkten einer bestimmten Version bestehen.<br><br>Die PD ist letztlich das Buildscript, mit dem mittels eines geeigneten Buildtools aus den unterschiedlichen Quelldaten unter Berücksichtigung der Abhängigkeiten zu anderen Produkten die gewünschten Zielprodukte (z. B. Distributionspakete der SWE oder Updateseiten für die Plug-in) erstellt werden.<br><br>Beim Einsatz des Buildtools Maven entspricht das PD dem POM (pom.xml Datei), beim Einsatz von Gradle als Buildtool entspricht das PD der build.gradle Datei |

## 0.7 Referenzierte Dokumente / URLs

Die folgende Tabelle listet die im Dokument verwendeten Referenzen auf. Zum aktuellen Zeitpunkt sind die folgenden Archiv-URLs vorhanden:

- NERZ-Archiv: http://www.nerz-ev.de/ → Dokumente und Software

| | |
|---|---|
| [VMOD97] | Der Bundesminister des Inneren, Entwicklungsstandard für IT-Systeme des Bundes Vorgehensmodell, Juni 1997, KBSt, Koordinations- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. |
| [AbkBSVRZ] | Abkürzungsverzeichnis BSVRZ Gesamt<br>NERZ-Archiv: Abk_BSVRZ-Gesamt_FREI_V4.0_D2006-08-15.doc |
| [GlossarBSVRZ] | Glossar BSVRZ Gesamt<br>NERZ-Archiv: SE-02.0002-Glos-0.4__Glossar__global__.pdf |
| [JDK] | Übersicht über JavaSE inklusive dem Java SE Development Kit<br>http://www.oracle.com/technetwork/java/javase/overview/index.html |
| [JRE] | Übersicht über JavaSE inklusive des Java SE Runtime Environments<br>http://www.oracle.com/technetwork/java/javase/overview/index.html |
| [NERZ] | Homepage des Vereins der "Nutzer der einheitlichen Rechnerzentralensoftware für Verkehrsrechnerzentralen - NERZ e.V."<br>http://www.nerz-ev.de/www.nerz-ev.de/home.php |
| [NERZSoftware] | Softwarearchiv des NERZ e.V.<br>http://www.nerz-ev.de/www.nerz-ev.de/produkte/software/software.php |
| [DatKatHTML] | Datenkatalog der ERZ-Software<br>http://www.nerz-ev.de/datkat/start.html |

## 0.8 Abbildungsverzeichnis

Abbildung 4-1: Lösungsskizze Konfigurationsmanagement NERZ e.V. ....10
Abbildung 5-1: Prozesse – Aktivitäten Konfigurationsmanagement NERZ e.V. ....11
Abbildung 6-1: Server für das Konfigurationsmanagementsystem des NERZ e.V. ....21
Abbildung 7-1: Beispiel Verzeichnisstruktur (hier bei Gradle)....24
Abbildung 7-2: Inhalt einer Beschreibungsdatei *.md für Spezifikationen ....28

## 0.9 Tabellenverzeichnis

Tabelle 0-1: Dokumentenverteiler ....2
Tabelle 0-2: Änderungsübersicht....2

# 1 Zweck des Dokuments

Der NERZ e.V. beabsichtigt, neben der aktuell verfügbaren Bereitstellung der Software / Produkte als herunterladbares Paket auf der Internetseite des NERZ e.V. [NERZSoftware] zusätzlich ein zentrales Konfigurationsmanagement einzuführen.
Das Dokument behandelt die nachfolgenden Teilaspekte im Zusammenhang mit der Einführung eines Konfigurationsmanagements für die ERZ-Software:

- Beschreibung und Festlegung der Ziele und der zu berücksichtigen Randbedingungen → Kapitel 2
- Begriffsdefinitionen → Kapitel 3
- Beschreibung der geplanten Abläufe (Lösungsskizze, Workflow, Prozesse, Aktivitäten) im Zusammenhang mit dem Konfigurationsmanagement des NERZ e.V → Kapitel 4, Kapitel 5
- Technische Umsetzung → Kapitel 6
- Festlegung der Struktur der Repositories → Kapitel 7
- Festlegungen zur Versionierung von Teilprodukten → Kapitel 8
- Bereits getroffene Festlegungen / Nächste Schritte → Kapitel 9
  - In diesem Kapitel sind Entscheidungen, die in den vorhergehenden Kapiteln alternativ betrachtet wurden, dokumentiert, die zum Zeitpunkt des aktuellen Standes dieses Dokuments durch den NERZ e.V. bereits getroffen wurden.
  - Des Weiteren sind hier der aktuelle Stand der Umsetzung sowie die noch offenen Punkte jeweils zum Stand dieses Dokuments dargestellt.

# 2 Zielsetzung / Randbedingungen

## 2.1 Beschreibung und Festlegung der Ziele

Durch die Einführung eines Konfigurationsmanagementsystems (KMS) seitens des NERZ e.V. zur Verwaltung der Quellen und Produkte der ERZ-Software sollen folgende Ziele erreicht werden:

- einfache Bereitstellungsmöglichkeit des aktuellen Quellcodes der Softwaremodule, Konfigurationen, aktueller Distributionspakete und der Dokumentation für Entwickler und Anwender
- Versionierung folgender Eingangsprodukte mit einem Versionsverwaltungssystem[1]:
  - Quellcode
  - Dokumente
  - Konfigurationen (KV, KB)
  - Produktdefinitionen[2] (PD). Die PD ist die formale Beschreibung der Abhängigkeiten zwischen (Teil-)Produkten zwecks eindeutiger Versionierung sowie zur Erzeugung von Ausgangsprodukten (Distributionspaketen, Datenkatalogen und Systemen, …)

- Erzeugung folgender Ausgangsprodukte durch den NERZ e.V.:
  - Distributionspakete für SWE
  - Updateseiten für Plug-in
  - Dokumentationen
  - aktueller Datenkatalog als HTML-Version auf Basis der aktuellen Konfigurationsverantwortlichen (KV) / Konfigurationsbereiche (KB).
  - Konfigurationen (KV, KB)

[1] Als Versionsverwaltungssystem wurde bereits GIT festgelegt.

[2] Die PD ist letztlich das *Buildscript*, mit dem mittels eines geeigneten *Buildtools* aus den unterschiedlichen Quelldaten unter Berücksichtigung der Abhängigkeiten zu anderen Produkten die gewünschten Zielprodukte (z. B. Distributionspakete der SWE oder Updateseiten für die Plug-in) erstellt werden.

Beim Einsatz des Buildtools Maven entspricht das PD dem POM (`pom.xml` Datei), beim Einsatz von Gradle als Buildtool entspricht das PD der `build.gradle` Datei

### 2.2 Randbedingungen

Bei der umzusetzenden Lösung für das KMS sind entsprechend der Abstimmungen im Rahmen des Fachtreffens vom 03.07.2014 sowie der Vorgaben aus den Mitglieder- und Fachtelefonkonferenzen folgende Randbedingungen zu berücksichtigen:

- Die geplante werkzeugunterstütze Versionsverwaltung darf die Firmen (AN) intern nicht zu Änderungen zwingen
- Einsatz des Versionsverwaltungsprogramm GIT in Verbindung mit GitLab[3]
- Seitens der NERZ-Mitglieder ist die Speicherung der Daten auf einem deutschen (extern gemieteten) Server oder auf einem seitens der Auftragsverwaltung bereitgestellten Server zwingend. Aus diesem Grund wird eine Verwaltung unter GitHub ausgeschlossen und stattdessen die Verwendung von GitLab angestrebt, da GitLab auf einem eigenen Server installiert und verwaltet werden kann[4].
- Unabhängig vom genauen Speicherort ist bei einer Umsetzung in jedem Fall dafür zu sorgen, dass durch regelmäßige Spiegelung der relevanten Daten des verwendeten Servers in jedem Fall die Daten zu jedem Zeitpunkt für den NERZ e.V. unabhängig von Dritten verfügbar sind
- Die Erzeugung der Distributionspakete soll durch die NERZ erfolgen. Dazu sind verbindliche Vorgaben für die Anlieferung der zur Erzeugung notwendigen Teilprodukte zu erstellen und als Bestandteil des Vertrages mit dem jeweiligen AN zu vereinbaren.
- Der Test und die Abnahme von neuer oder geänderter Software sollen wie bisher auf Basis des durch den AN erstellen Build der Distributionspakete erfolgen. Nach erfolgter Abnahme werden dann der Quellcode, externe Bibliotheken sowie alle weiteren notwendigen Teilprodukte sowie eine standarisierte Build-Datei (Produktdefinition, PD) an die NERZ übergeben, die für die entsprechende Versionierung der Produkte und die Erstellung und Bereitstellung der „offiziellen“ Distributionspakete zuständig ist.
- Zugriffsrechte auf das / die zur Verfügung gestellten Repositories für die Quelldaten
    - Lesezugriffe
        - Jeder
    - Schreibzugriffe
        - Nur NERZ.
            - Änderungen siehe Punkte „Workflow Änderungen“.
            - Die Prüfung und Übernahme erfolgt ausschließlich durch den NERZ e.V.

## 3 Begriffsdefinitionen

Nachfolgend werden kurz die bei der Darstellung der Abläufe verwendeten Begrifflichkeiten definiert.

Die Beschreibung der Abläufe und Aktivitäten ist u. U. abhängig vom Typ eines (Teil-)Produkts. Einige Produkttypen stellen die Eingangsgrößen der Abläufe dar, andere die Ausgangsgrößen und wieder andere die Informationen, die aus den Eingangsprodukten die Ausgangsprodukte erzeugen.

Folgende Produkttypen werden unterschieden:

- Eingangsprodukte
    - Sourcecode
    - Dokumente
    - Konfigurationsbereich (KB)
    - Konfigurationsverantwortliche (alle KB eines KV)
    - Produktdefinition. Die PD ist die formale Beschreibung der Abhängigkeiten zwischen (Teil-) Produkten zwecks eindeutiger Versionierung sowie zur Erzeugung von Distributionspaketen, Datenkatalogen und Systemen.

[3] Abweichende Alternativen hierzu siehe Kapitel 6.3 und 6.4.

[4] Abweichende Alternativen hierzu siehe Kapitel 6.3 und 6.4.

- Ausgangsprodukte
  - Distributionspakete für SWE
  - Updateseiten für Plug-in
  - Dokumentationen
  - aktueller Datenkatalog als HTML-Version auf Basis der aktuellen Konfigurationsverantwortlichen (KV) / Konfigurationsbereiche (KB).
  - Konfigurationen (KV, KB)
  - (Beispiel-)Systeme

Die Eingangsprodukte im Sinne der vorliegenden Beschreibung sind einzelne, mittels des Versionsverwaltungssystems (GIT) zu versionierende Einheiten aus denen die eigentlichen Ausgangsprodukte bestehen bzw. mit Hilfe des einzusetzenden Buildtools auf Basis der PD erzeugt werden können:

# 4 Lösungsskizze

Die nachfolgende Abbildung[5] zeigt prinzipiell die angestrebte Lösung im Zusammenhang mit dem geplanten Konfigurationsmanagement der ERZ-Software.

Eine formalere Darstellung der notwendigen Prozesse und Aktivitäten enthält Kapitel 5ff.

Die technische Sicht auf das geplante System enthält Kapitel 6ff.

[5] Die Abbildungen sind nochmals in voller Größe im Anhang Kapitel 10 enthalten.

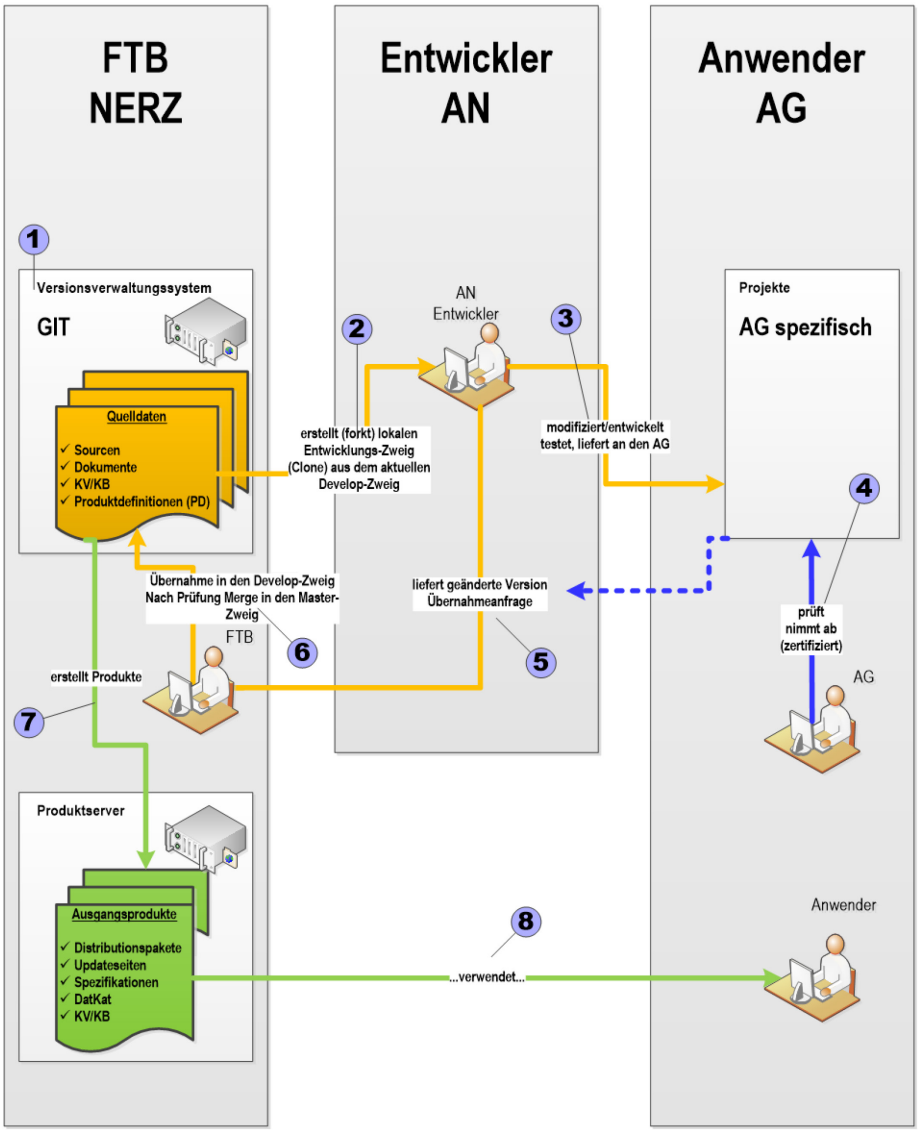



*Abbildung 4-1: Lösungsskizze Konfigurationsmanagement NERZ e.V.*

1. Auf dem Versionsverwaltungsserver werden die Quelldaten in GIT-Repositories gespeichert.
2. Ein Entwickler (AN) kann die gewünschten Quelldaten (Source, Dokumente, KV, …) auschecken, indem er einen Clone (lokale Kopie) des seitens der FTB bereitgestellten Entwicklungszweigs (Develop-Zweig) erstellt.
3. Der AN modifiziert oder erstellt das seitens eines AG gewünschte Produkt, testet dieses und liefert es an den AG (z. B. für ein AG spezifisches Projekt)
4. Der AG prüft das gelieferte Produkt, nimmt dieses ab und zertifiziert dieses ggf.
5. Das geänderte / neue Produkte wird an die FTB zur Übernahme in das GIT-Repository übergeben (Übernahmeanfrage)
6. Die FTB übernimmt die Änderungen in den Develop-Zweig. Die Änderungen werden dort formal geprüft und nach erfolgreicher Prüfung in den Master-Zweig des zentralen GIT-Repository „gemerged“ und mit einer neuen Version gekennzeichnet.
7. Die FTB erstellt aus den geänderten Quelldaten auf Basis der Produktdefinition (PD) mit dem Buildtool (Maven oder Gradle) die eigentlichen Ausgangsprodukte (Distributionspakete, Updateseiten, …) und stellt diese auf dem allgemein zugänglichen Produktserver zur Verfügung. Zudem

wird das versionierte Produkt auf einem Server zur Verwaltung der Abhängigkeiten in einer spezifischen Form (Maven-Repository) bereitgestellt.

8. Anwendern stehen auf dem Produktserver die unterschiedlichen Produkte zum Download bereit.

# 5 Prozesse – Aktivitäten

In Anlehnung an die zuvor dargestellte Lösungsskizze enthält die nachfolgende Abbildung die notwendigen Prozesse und Aktivitäten der verwalteten bzw. erzeugten (Teil-) Produkte im „**Regelbetrieb**“ in einer formaleren Darstellung[6] (der einmalig notwendige Prozess „Ersteinrichtung des Versionsverwaltungssystems/Konfigurationsmanagements“ ist nicht dargestellt und wird als bereits erfolgt vorausgesetzt).

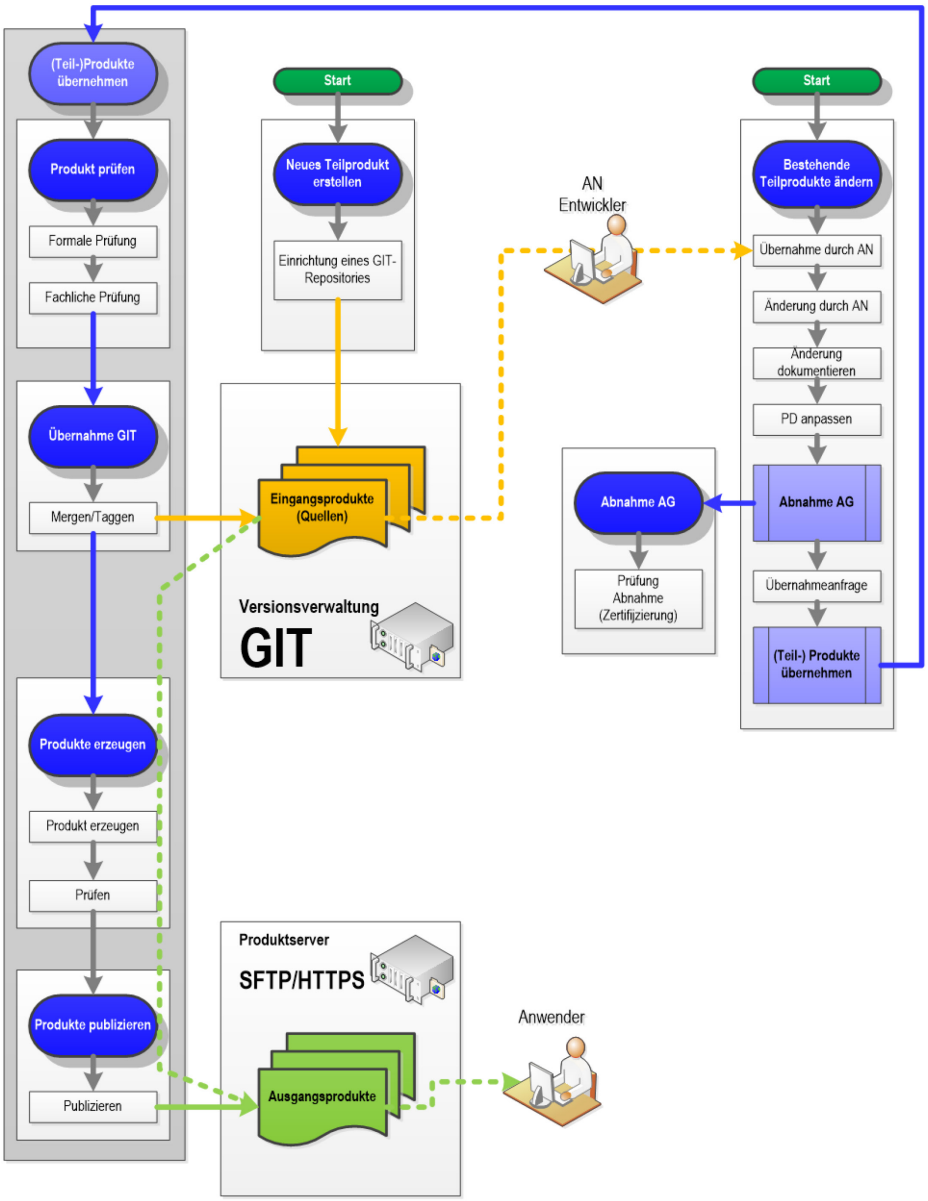



*Abbildung 5-1: Prozesse – Aktivitäten Konfigurationsmanagement NERZ e.V.*

[6] Die Abbildungen sind nochmals in voller Größe im Anhang Kapitel 10 enthalten.

Die einzelnen Prozesse mit ihren Aktivitäten sind in den nachfolgenden Kapiteln dargestellt.

### 5.1 Prozess „Neues Teilprodukt erstellen“

| | |
|---|---|
| **Anwendung** | Der Prozess „Neues Teilprodukt erstellen“ dient dazu, für ein neu zu erstellendes Teilprodukt die notwendigen GIT-Repositories mit den standardisierten Ordnerstrukturen und (Dummy-)Dateien anzulegen und die gewünschte PD (Produktdefinition) anzulegen und in GIT einzuchecken.<br>Anschließend kann dann durch den für die Erstellung zuständigen AN der Prozess „Bestehende Teilprodukte ändern“ ausgeführt werden. |
| **Durchführung** | NERZ e.V. (FTB) |
| **Aktivitäten** | |
| GIT Repositories anlegen | Anlegen eines neuen GIT-Repositories für den gewünschten Produkttyp (siehe Kapitel 7ff) |
| Produktvorlage erstellen | Erstellung einer Produktvorlage (Verzeichnisstruktur, Standard Produktdefinition, etc.) gemäß den Festlegungen in Kapitel 7ff. |
| In GIT einchecken | Produkt in GIT einchecken, zusätzlich den Develop-Zweig anlegen |

### 5.2 Prozess „Bestehende Teilprodukte ändern“

| | |
|---|---|
| **Anwendung** | Der Prozess „Bestehende Teilprodukte ändern“ dient zur Änderung eines bereits versionierten Teilprodukts, i. d. R. durch einen externen AN und der Rückführung der Änderung in das Versionsverwaltungssystem. Soll ein neues Produkt erstellt werden, so ist zuvor der Prozess „Neues Teilprodukt erstellen“ auszuführen. |
| **Durchführung** | AN (Hersteller, Entwickler) im Auftrag eines AG |
| **Aktivitäten** | |
| Übernahme durch AN | Der mit der Änderung beauftragte AN erzeugt für das zu ändernde Teilprodukte ein lokale Kopie auf seinem eigenen Rechner auf Basis des eingerichteten Develop-Zweigs (technisch: Klonen des Develop-Zweigs aus dem Repository) |
| Änderung durch AN | Der AN führt auf seinem lokalen Branch die beauftragten Änderungen durch, testet diese und stellt dem AG die Änderung zur Abnahme zur Verfügung. |
| Änderungen dokumentieren | Der AN dokumentiert die Unterschiede zwischen der Ursprungsversion und dem geänderten Produkte. Die Änderungsdokumentation ist als Bestandteil des dokumentierten (Teil-)Produkts mit zu übernehmen. |
| PD anpassen | Der AN passt die PD (Produktdefinition) an die Änderungen an, insbesondere sind dies die geänderten Angaben zu Version, Stand, Abhängigkeiten zu anderen Produkten etc. Auf Basis dieser PD erzeugt die FTB dann später im Prozess „Produkt erzeugen“ die definierten Zielprodukte (Distributionspakete, Dokumentenpakete etc.) und publiziert diese. |
| Abnahme AG | Der AG prüft die Änderung und erteilt die Abnahme → siehe Pro- |

| | |
|---|---|
| | zess *„Abnahme AG“*. |
| Übernahmeanfrage | Der AN erstellt eine Übernahmeanfrage. Diese Anfrage löst dann den Prozess „Teilprodukte übernehmen“ aus. |
| *(Teil-)Produkte übernehmen* | Das erstellte Produkt ist in das Versionsverwaltungssystem zu übernehmen → Durchführung des Prozesses „*(Teil-)Produkte übernehmen*“ |

### 5.3 Prozess „Abnahme AG“

| | |
|---|---|
| **Anwendung** | Der Prozess „Abnahme AG“ ist durch den AG für ein zu änderndes bzw. neu erstelltes Teilprodukt vor Rückführung dieses Teilprodukts in das Quellproduktverzeichnis durchzuführen. |
| **Durchführung** | AG, der die Änderung (Neuerstellung) eines Produkts beauftragt hat |
| **Aktivitäten** | |
| Prüfung / Abnahme | Der AG hat vor der Rückführung eines geänderten oder neu erstellten (Teil-)Produkts (→ Prozess *„(Teil-)Produkt übernehmen“*) dieses auf Einhaltung der geforderten Spezifikation und auf Konformität gegen die Vorgaben des NERZ e.V. hinsichtlich der Produktqualität zu prüfen und abzunehmen. Insbesondere ist zu prüfen, ob durch die Änderungen die Spezifikationsdokumente anzupassen sind. |
| Zertifizierung | Der AG hat ggf. die Durchführung der (Neu-)Zertifizierung des geänderten oder neu erstellten Produkts durchzuführen. |

### 5.4 Prozess „(Teil-)Produkt übernehmen“

| | |
|---|---|
| **Anwendung** | Der Prozess „(Teil-)Produkt übernehmen“ enthält die zur Übernahme, Prüfung, Versionierung und Erzeugung und Publikation der definierten Zielprodukte für das zu übernehmende Teilprodukt notwendigen Prozesse:<br>• Produkt prüfen<br>• Übernahme GIT (in Versionsverwaltung)<br>• Produkte erzeugen<br>• Produkte publizieren |
| **Durchführung** | NERZ e.V. (FTB) |

### 5.5 Prozess „Produkt prüfen“

| | |
|---|---|
| **Anwendung** | Der Prozess „Produkt prüfen“ dient der formalen und fachlichen Prüfung der zur Übernahme bereitgestellten Produkte vor der eigentlichen Übernahme in das Versionsverwaltungsprogramm und der anschließenden Produkterstellung und –Publikation. |
| **Durchführung** | NERZ e.V. (FTB) |
| **Aktivitäten** | |
| Formale Prüfung | Prüfung des zu übernehmenden Produkts auf Einhaltung der formalen Vorgaben für den jeweiligen Produkttyp (Ordnerstruktur, Vollständigkeit der Daten, korrekte Versionsbezeichnung etc., Korrekt- |

| | |
|---|---|
| | heit der zugehörigen Produktdefinition (PD) etc. |
| Fachliche Prüfung | (Grobe) Prüfung, ob das Produkt die geforderte fachliche Spezifikation erfüllt. Dieser Schritt sollte allerdings bereits vorher im Rahmen des Prozesses „*Abnahme AG*“ vollständig durchgeführt und bestätigt worden sein. |

### 5.6 Prozess „Übernahme GIT“

| | |
|---|---|
| **Anwendung** | Beim Prozess „Übernahme GIT“ wird im ersten Schritt das vom AN geänderte Produkt in den Develop-Zweig übernommen, geprüft und nach der erfolgreichen Prüfung des Produkts dieses in den Masterzweig übernommen und gekennzeichnet. |
| **Durchführung** | NERZ e.V. (FTB) |
| **Aktivitäten** | |
| Übernehmen | Der AN stellt den Merge-Request zur Übernahme der Änderungen in den Develop-Zweig an die FTB. Die Übernahme wird von der FTB durchgeführt. |
| Zusammenführen („Mergen“) | Das im Develop-Zweig geprüfte Produkt wird durch die FTB mit dem Masterzweig zusammengeführt, so dass daraus jetzt eine neue Version des Produkts entsteht. |
| Produktkennzeichnung („Taggen“) | Der neue Stand wird im Masterzweig mit der aktuellen Version und Stand gekennzeichnet (Technisch: Erstellen eines TAG’s in GIT gemäß den Konventionen gemäß Kapitel 0ff). |

### 5.7 Prozess „Produkte erzeugen“

| | |
|---|---|
| **Anwendung** | Der Prozess „Produkt erzeugen“ dient dazu, mittels der Produktdefinition (PD) automatisch aus den in der Produktdefinition definierten Teilprodukten und Abhängigkeiten zu anderen Produkten ein oder mehrere konkrete Produkte in einer neuen Version zu erstellen (Distributionspaket für SWE, Updateseite für Plug-in, Dokumentenzusammenstellung mit PDF-Erzeugung etc.). |
| **Durchführung** | NERZ e.V. (FTB) |
| **Aktivitäten** | |
| Produkt erzeugen | (Automatische) Erstellung einer neuen Version eines Produkts auf Basis der PD. |
| Prüfen | Das erstellte Produkt ist vor der Publikation zu prüfen.<br>Bei erzeugten Produkten, die auf Basis von bei einem AG abgenommenen Produkten erstellt wurden, ist zu prüfen, ob das durch den AG abgenommene Produkt (erstellt durch den AN) identisch mit dem durch den Prozess „Produkt erzeugen“ erstellten Produkt auf Basis der bereitgestellten Teilprodukte und der PD ist. |

### 5.8 Prozess „Produkte publizieren“

| | |
|---|---|
| **Anwendung** | Beim Prozess „Produkte publizieren“ werden die aus den Quellprodukten i.d.R. automatisch auf Basis der PD erzeugten Zielprodukte |

| | |
|---|---|
| | auf den Produktserver hochgeladen. Dort stehen sie dann für alle Anwender zur direkten Verwendung zur Verfügung. |
| **Durchführung** | NERZ e.V. (FTB) |
| **Aktivitäten** | |
| Publizieren | Hochladen (kopieren) der Zielprodukte auf den Produktserver. |

# 6 Technische Umsetzung

## 6.1 Build Tool

Als Build-Tool stehen die alternativen Gradle oder Maven zur Diskussion. Gradle ist die aktuellere und modernere Variante, bei der die Build-Dateien (Produktdefinitionen) in einer Domänen spezifischen Sprache spezifiziert werden können, währenddessen Maven ein schon länger auf dem Markt befindliches, weitestgehend ausgereiftes System ist, bei dem die Build-Dateien (Produktdefinitionen) in XML spezifiziert werden müssen.

Nachfolgend die Zusammenfassung der Beschreibung der beiden Werkzeuge in Wikipedia (in Auszügen):

### 6.1.1 Maven

Text in Auszügen von *https://de.wikipedia.org/wiki/Apache_Maven*

***Maven** ist ein Build-Management-Tool der Apache Software Foundation und basiert auf Java. Mit ihm kann man insbesondere Java-Programme standardisiert erstellen und verwalten.*

***Konzeptionelles***

*Maven versucht, den Grundgedanken „Konvention vor Konfiguration“ (englisch Convention over Configuration) konsequent für den gesamten Zyklus der Softwareerstellung abzubilden. Dabei sollen Software-Entwickler von der Anlage eines Softwareprojekts über das Kompilieren, Testen und „Packen“ bis zum Verteilen der Software auf Anwendungsrechnern so unterstützt werden, dass möglichst viele Schritte automatisiert werden können. Folgt man dabei den von Maven vorgegebenen Standards, braucht man für die meisten Aufgaben des Build-Managements nur sehr wenige Konfigurationseinstellungen zu hinterlegen, um den Lebenszyklus eines Softwareprojekts abzubilden.*

***Der Standard-Lebenszyklus***

*Maven geht von einem Zyklus aus, der bei der Softwareerstellung häufig durchlaufen wird. Dabei wird nicht unterstellt, dass jedes Softwareprojekt alle Phasen des im Folgenden dargestellten Zyklus verwendet:*

- *archetype : Damit kann ein Template für ein Softwareprojekt erstellt werden. Abhängigkeiten werden aufgelöst und bei Bedarf heruntergeladen.*
- *validate (Validieren): Hier wird geprüft, ob die Projektstruktur gültig und vollständig ist.*
- *compile (Kompilieren): In dieser Phase wird der Quellcode kompiliert.*
- *test (Testen): Hier wird der kompilierte Code mit einem passenden Testframework getestet. Maven berücksichtigt dabei in späteren Zyklen, dass Testklassen normalerweise nicht in der auszuliefernden Software vorhanden sind.*
- *package (Verpacken): Das Kompilat wird – ggf. mit anderen nichtkompilierbaren Dateien – zur Weitergabe verpackt. Häufig handelt es sich dabei um eine Jar-Datei.*

- *integration-test (Test der Integrationsmöglichkeit): Das Softwarepaket wird auf eine Umgebung (anderer Rechner, anderes Verzeichnis, Anwendungsserver) geladen und seine Funktionsfähigkeit geprüft.*
- *verify (Gültigkeitsprüfung des Software-Pakets): Prüfungen, ob das Softwarepaket eine gültige Struktur hat und ggf. bestimmte Qualitätskriterien erfüllt.*
- *install (Installieren im lokalen Maven-Repository): Installiere das Softwarepaket in dem lokalen Maven-Repository, um es in anderen Projekten verwenden zu können, die von Maven verwaltet werden. Dies ist insbesondere für modulare Projekte interessant.*
- *deploy (Installieren im fernen Maven-Repository): Stabile Versionen der Software werden auf einem fernen Maven-Repository installiert und stehen damit in Umgebungen mit mehreren Entwicklern allen zur Verfügung.*

*Dieser Lebenszyklus kann noch wesentlich durch Maven-Plugins und Maven-Archetypen (engl. Archetypes) erweitert werden. Mit Maven-Archetypen können Gerüste für unterschiedlichste Softwareprojekte erstellt werden, die der Standardstruktur von Maven entsprechen.*

*Maven-Plugins ermöglichen es, neue Verarbeitungsschritte zu verwenden (z. B. Verteilung auf einen Anwendungsserver) und/oder die Schritte im Standard-Lebenszyklus zu erweitern.*

***Die Konfigurationsdatei: pom.xml***

*Normalerweise werden die Informationen für ein Softwareprojekt, das von Maven unterstützt wird, in einer XML-Datei mit dem Dateinamen pom.xml (für **P**roject **O**bject **M**odel) gespeichert. Diese Datei enthält alle Informationen zum Softwareprojekt und folgt einem standardisierten Format. Wird Maven ausgeführt, prüft es zunächst, ob diese Datei alle nötigen Angaben enthält und ob alle Angaben syntaktisch gültig sind, bevor es weiterarbeitet.*

***Standard-Verzeichnisstruktur***

*Ein wesentliches Element des Prinzips Convention over Configuration ist die Standard-Verzeichnisstruktur von Maven. Sofern ein Projekt sich daran hält, müssen die Pfadnamen nicht spezifiziert werden, was die zentrale Konfigurationsdatei pom.xml stark vereinfacht. Auf oberster Stufe gibt es die beiden Verzeichnisse src und target. src enthält alle Dateien, die als Eingabe für den Verarbeitungsprozess dienen und in target werden automatisch alle erzeugten Dateien abgelegt. Auch weitere Verzeichnisstufen sind standardisiert und Plugins, die neue Arten von Eingabedateien verarbeiten oder neue Arten von Ausgabedateien erzeugen, geben dafür Standardpfade vor.*

*Die folgende Struktur zeigt einige der wichtigsten Verzeichnisse.*

- *src: alle Eingabedateien*
- *src/main: Eingabedateien für die Erstellung des eigentlichen Produkts*
- *src/main/java:* Java-Quelltext
- *src/main/resources: andere Dateien, die für die Übersetzung oder zur Laufzeit benötigt werden, beispielsweise* Java-Properties-Dateien
- *src/test: Eingabedateien, die für automatisierte Testläufe benötigt werden*
- *src/test/java:* JUnit-*Testfälle für automatisierte Tests*
- *target: alle erzeugten Dateien*
- *target/classes: kompilierte Java-Klassen*

***Auflösung von Abhängigkeiten, zentrale Repositorys***

*In der pom.xml werden Softwareabhängigkeiten angegeben, die ein von Maven unterstütztes Softwareprojekt zu anderen Softwareprojekten hat. Diese Abhängigkeiten werden aufgelöst, indem Maven zunächst ermittelt, ob die benötigten Dateien in einem lokalen Verzeichnis, dem lokalen Maven-Repository, bereits vorhanden sind. Sind sie es, verwendet Maven z. B. beim Kompilieren die lokal vorhandene Datei von dort, also ohne sie in das Projektverzeichnis zu kopieren.*

*Kann die Abhängigkeit nicht lokal aufgelöst werden, versucht Maven, sich mit einem konfigurierten Maven-Repository im Intranet oder Internet zu verbinden und von dort die Dateien in das lokale Repository zu kopieren, um sie von nun an lokal verwenden zu können. Bekannte öffentliche Maven-Repositorys sind Apache, Ibiblio, Codehaus oder Java.Net. Firmenweite über das Intranet ansprechbare Maven-Repositorys dienen dazu, selbst entwickelte oder gekaufte Bibliotheken und Frameworks firmenweit allen Projekten zur Verfügung zu stellen. Diese Repositorys werden üblicherweise mittels Software wie Apache Archiva, Nexus, Artifactory, Proximity, Codehaus Maven Proxy oder Dead Simple Maven Proxy realisiert.*

***Individualisierbarkeit***

*Fast alle Vorgaben, die Maven macht, können individuell geändert werden, bis auf die Struktur der Projektdatei (pom.xml): Man kann unterschiedliche Projektpfade wählen, Compiler für andere Sprachen verwenden (sofern von Plugins unterstützt).*

***Unterstützung in Entwicklungsumgebungen***

*Für die gängigsten Entwicklungsumgebungen (z. B. Eclipse, IntelliJ IDEA oder NetBeans) sind Plugins vorhanden, über die sich Maven direkt aus der Entwicklungsumgebung heraus bedienen lässt. Zusätzlich sind Maven-Plugins vorhanden, die Dateien erzeugen, welche den Import eines reinen Maven-Projekts in die Entwicklungsumgebung ermöglichen (siehe auch Beispiele).*

***Design***

*Maven basiert auf einer Plugin-Architektur, die es ermöglicht, Plugins für verschiedene Anwendungen (compile, test, build, deploy, checkstyle, pmd, scp-transfer) auf das Projekt anzuwenden, ohne diese explizit installieren zu müssen. Die Anzahl an Plugins ist mittlerweile sehr umfangreich: Die Bandbreite reicht von Plugins, die es ermöglichen, direkt aus Maven heraus eine Webanwendung zu starten, um sie im Browser zu testen, über welche, die es ermöglichen, Datenbanken zu testen oder zu erstellen, bis hin zu solchen, die Web Services generieren. Die dafür nötigen Tätigkeiten beschränken sich häufig nur darauf, das gewünschte Plugin zu ermitteln und einzusetzen.*

### 6.1.2 Gradle

Text in Auszügen von *https://de.wikipedia.org/wiki/Gradle*

***Konzeption und Plugin-Architektur***

*Gradles Build-Konzept übernimmt die von Maven eingeführten Standardkonventionen („convention over configuration“) für das Verzeichnislayout der Projektquellen, die üblichen Phasen für den Bau eines (Java-) Projekts (Validieren, Kompilieren, Testausführung, Archiv-Erstellung und Report-Generierung, Verteilung). Die Build-Datei kann daher minimal ausfallen und bei einem simplen Java-Projekt aus einer einzigen Zeile (apply plugin: 'java') bestehen. Ebenso übernimmt Gradle weitgehend das Maven-Konzept des Managements der Abhängigkeiten eines Projekts von anderen Projekten oder Fremdbibliotheken. Gradle kann sich hierbei auf die weitverbreiteten Maven Repositories (lokale, Firmen- und Internet-Repositories) stützen. Alternativ kann der Anwender aber auch auf Ivy-Repositories zurückgreifen.*

*Ähnlich wie Maven besteht Gradle aus einem abstrakten Kern und einer Vielzahl von Plug-ins und ist durch diese Struktur vielfältig erweiterbar. Selbst die Implementierung des Java-Builds basiert auf einem Plugin für Java. Mit dieser Architektur bietet Gradle die Möglichkeit, Buildprozesse für beliebige Software-Plattformen bewerkstelligen zu können und liefert dem Anwender die Möglichkeit, seine „nicht-konventionellen" Vorstellungen dem Tool beizubringen. Gradle liefert von Hause aus Plug-ins mit, die neben Java Groovy-, Scala- und C++- Projekte bauen können. Daneben wird der Build von Java Enterprise Archiven (WAR, EAR) unterstützt. Weitere Plug-ins erlauben die Überwachung der Softwarequalität (beispielsweise FindBugs, PMD, Checkstyle) durch automatisierte Checks und Generierung entsprechender Reports.*

*Die mit Gradle mitgelieferten Plug-ins sind zwar hauptsächlich für die Entwicklung und das Deployment von Java-, Groovy- und Scala-Projekten gemacht. Gradle kann aber auch für andere Programmiersprachen und Workflows eingesetzt werden. Seit der Entscheidung des Android-Teams für Gradle als Build-System wird von den Entwicklern hauptsächlich die Unterstützung eines Buildmodells für native Programmierumgebungen vorangetrieben.*

***Gradle DSL***

*Im Gegensatz zu den Konventionen von Apache Maven und dessen XML-Deklarationen arbeitet der Anwender mit Gradles Domänenspezifischer Sprache, die er – da eine Gradle-Build-Datei immer ein Groovy-Skript darstellt – erweitern oder in den Standardeigenschaften ändern kann. Ebenso kann er mit Groovy-Code eigene Build-Änderungen schreiben oder vordefinierte Standards überschreiben und eigenen Belangen anpassen. Der Gradle-Anwender kann beide Stile verwenden: den deklarativen, auf Standardkonventionen beruhenden Ansatz von Maven und den eher imperativen Ansatz von Ant, bei dem der Anwender aber auch alles im Detail definieren muss.*

*Der Anwender ist auf Basis dieser DSL-Sprache nicht gezwungen, zuerst einmal Groovy lernen zu müssen, bevor er sich an Gradle-Buildskripte heranwagt.*

***Der Gradle-Build***

*Gradle kennt zwei Hauptphasen der Buildverarbeitung, die immer durchlaufen werden: Konfiguration und Ausführung. Während des Konfigurations-Zyklus wird die gesamte Build-Definition durchlaufen, um den Abhängigkeitsgraphen (DAG) zu erzeugen, der die Reihenfolge aller abzuarbeitenden Schritte enthält. Im zweiten Teil wird dieser Graph für die gewünschten Tasks durchlaufen. Sowohl die Konfiguration als auch die Ausführung sind dem Anwender durch eine offene API zugänglich.*

*Der Buildprozess, der durch den Task-Graphen beschrieben wird, besteht aus einer Abfolge von Tasks, die hierarchisch voneinander abhängen, und wo ein Nachfolger nur ausgeführt wird, wenn seine Vorgänger erfolgreich durchlaufen wurden. So wird beispielsweise der Task ‚test' nur ausgeführt, wenn zuvor die Tasks ‚compile', ‚processresources', ‚classes', ‚testCompile' ohne Fehler durchlaufen wurden.*

***Build-Dateien***

*Gradle nutzt für einen einfachen Build hauptsächliche drei benutzerdefinierte Dateien:*

- *build.gradle – die auf der Gradle-DSL beruhende Definition des Builds mit allen Tasks und Abhängigkeiten eines Projekts (ein Multiprojekt hat pro Projekt eine solche Build-Datei, die durch Vererbung der Eigenschaften von ihrem „Vater"-Buildskript kurz gehalten werden können).*
- *settings.gradle (optional) – bei einem Multiprojekt werden hier die teilnehmenden Unterprojekte festgelegt.*
- *gradle.properties (optional) – eine Liste von Properties, die für die projektspezifische Gradle-Initialisierung eines Builds gültig sind.*

*Gradle-Skripte können unmittelbar Groovy-Code enthalten oder durch eine Groovy-Klasse implementiert werden. Alternativ lassen sie sich als Build-Abhängigkeit aus einem Maven-Repository laden.*

### 6.1.3 Einsatzmöglichkeiten

Die Einsatzfähigkeit von Maven ist sowohl für die Anforderungen hinsichtlich der Software serverseitig (SWE → Distributionspakete) als auch hinsichtlich der Anforderung der Plug-in Entwicklung gewährleistet.[7].

Hinsichtlich Gradle wurden grundsätzliche Prüfungen für die Serverseite durchgeführt, so dass auch hier davon ausgegangen werden kann, dass die entsprechende Umsetzung gewährleistet ist.

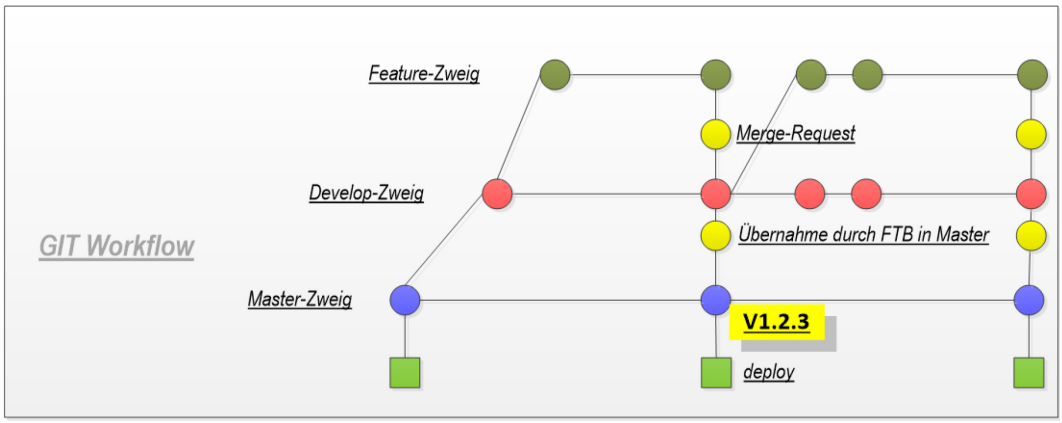
Die Einsatzfähigkeit von Gradle für die Plug-in Entwicklung müsste im Detail noch verifiziert werden.

Allerdings gibt es auch für Eclipse (was faktisch die Standardentwicklungsumgebung für die Entwicklung von Plug-ins ist) diverse Plug-ins für die Gradle-Integration, sodass auch hier grundsätzlich von der Einsetzbarkeit von Gradle ausgegangen werden kann.

Alternativ wäre auch vorstellbar, dass die Vorgaben für die Plug-in Entwicklung auf Maven basieren, die für die übrigen Produkte (SWE, Dokumente, Datenkatalog, KV/KB) auf Gradle.

## 6.2 GIT Workflow

Für die Anforderungen des NERZ e.V. werden für den GIT-Workflow der nachfolgend dargestellte, einfache Workflow vorgeschlagen. Spätere Änderungen am Workflow hin zu deutlich komplexeren Abläufen (wie z. B. dem „GitFlow“) lassen sich bei Bedarf auch noch später etablieren.




- Es gibt einen *Master-Zweig*, dessen Inhalt jederzeit *deploybar* sein muss (aus dem sich jederzeit stabile Produkt-Versionen erstellen lassen müssen).
- Parallel dazu gibt es einen *Develop-Zweig*, in den alle Änderungen über eine Merge-Request (nach Prüfung durch die FTB) durch die FTB übernommen werden.
- Um eine Änderung zu machen, wird das Repository durch den AN der Änderung geklont und vom *Develop-Zweig*[8] ein *Feature-Zweig* erstellt (z.B. „ÄM4711-DUA“), auf dem die eigentlichen Änderungen durchgeführt.

[7] Entsprechende Einrichtungen, die in wesentlichen Grundsätzen den hier aufgeführten Konzepten entsprechen, werden seit einiger Zeit seitens der Firma BitCtrl für die von BitCtrl bereitgestellten SWE und Plug-ins auf Basis von Maven realisiert

- Bei Umfangreichen Änderungen, bei denen ggf. der allgemeine Zugriff auf die laufenden Änderungen gewünscht ist, kann der *Feature-Zweig* bereits vor dem Klonen im Repository erstellt werden und dieser Zweig dann regelmäßig aktuell gehalten und zum Server übertragen werden.
- Sind die Arbeiten abgeschlossen, erstellt man einen *Merge-Request* (Übernahmeanfrage) auf den *Develop-Zweig*.
- Nach Prüfung der Änderungen (durch die FTB) wird die Änderung in den *Develop-Zweig übernommen* und nach abschließender Prüfung anschließend in den *Master-Branch* integriert und eindeutig *getaggt* (gekennzeichnet, versioniert).
- Im Anschluss daran werden die aktualisierten Zielprodukte aus dem *Master-Branch* erzeugt und publiziert.

### 6.3 Server

Die folgende Abbildung zeigt die vereinfacht das Zusammenspiel und den Zugriff auf die benötigten Server für das geplante Konfigurationsmanagementsystem. Die Details zu den einzelnen Servern beschreiben die nachfolgenden Unterkapitel.

In der Abbildung ist auch das einfache GIT-Workflow-Konzept dargestellt.

[8] Oder von einem anderen Entwicklungsstand, z. B. im *Master-Zweig*

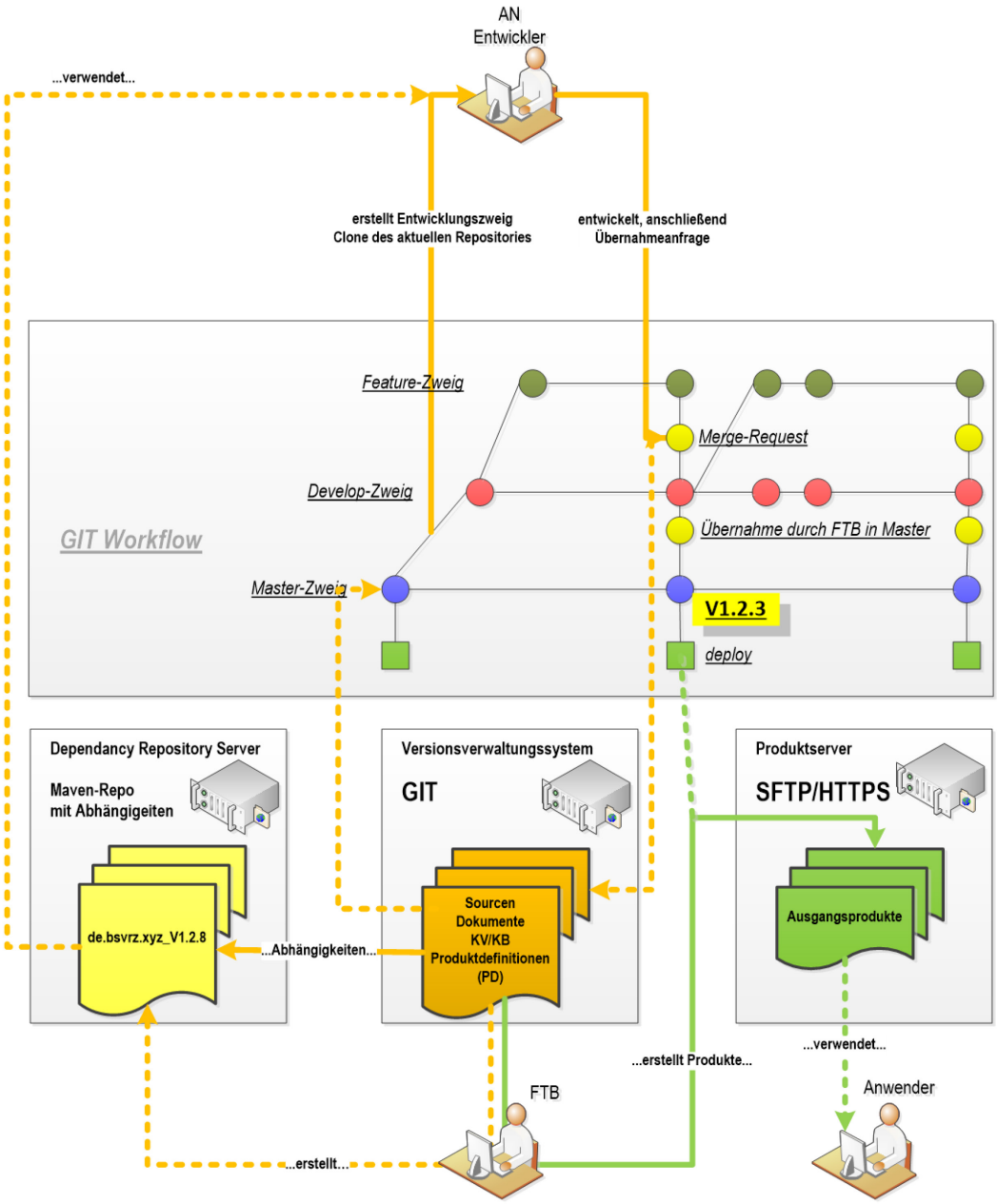



*Abbildung 6-1: Server für das Konfigurationsmanagementsystem des NERZ e.V.*

### 6.3.1 Versionsverwaltungssystem / Server für Versionsverwaltungssystem

Als Versionsverwaltungssystem wurde bereits im Vorfeld GIT ausgewählt. Für die technische Umsetzung zur Verwaltung der GIT-Repositories stehen verschiedene Möglichkeiten zur Verfügung (Auswahl weitverbreiteter Lösungen, nicht vollständig):

- GIT-Hub
- Bitbucket
- GIT-Lab

Die beiden ersten beiden Varianten (GIT-Hub und Bitbucket) sind frei verfügbare bzw. kostenlose Serverprozesse, im Internet bereitgestellt werden. Bei der Nutzung dieser Produkte werden die Daten auf i. d. R. auf einem nicht deutschem oder europäischem Recht unterstehenden Server gespeichert. Die notwendigen Wartungsarbeiten wie z. B. Backups, Update und Pflege der Installation etc. werden durch die entsprechenden Betreiber dieser Produkte durchgeführt. Die Nutzung ist z. T. kostenlos für Projekte mit offener Software, so wie dies bei der NERZ Software der Fall ist.

Die dritte Variante GIT-Lab entspricht im Wesentlichen der ersten Variante GIT-Hub, mit dem Unterschied, dass die Software auf einem eigenen Server eingerichtet und betrieben werden kann (muss). In diesem Fall kann die Software bzw. das Versionsverwaltungsprogramm auf einem Server gehostet werden, der die Vorgaben des Bundes an datenrechtliche Vorgaben erfüllt. Eine Installation und der Betrieb auf einem in Deutschland betriebenen Server sind damit möglich.

Der Vorteil dieser Lösung, dass das System vollständig unter der Kontrolle des NERZ e.V. steht, wird natürlich mit dem Nachteil erkauft, dass die gesamte Pflege des Servers einschließlich der notwendigen Sicherheitsupdates, Zugriffsbeschränkungen, Backups, Updates etc. aktiv durch den NERZ e.V. durchgeführt werden muss.

### 6.3.2 Dependancy Repository Server (Server zur Verwaltung der Abhängigkeiten)

Für die Erstellung der Zielprodukte, hier insbesondere der Distributionspakete der SWE bzw. der Updateseiten der Plug-ins, mittels der Buildtools Maven oder Gradle, benötigen beide einen „Dependancy Repository Server“ (Server zur Verwaltung der Abhängigkeiten). Auf diesem Server werden die für einen Build notwendigen Produkte, von denen das zu erstellende Distributionspaket abhängig ist, in einer speziellen Form gespeichert. Unabhängig vom verwendeten Werkzeug (Maven oder Gradle) werden die Daten in Form eines sogenannten „Maven-Repositories“ abgelegt.

Vereinfacht ausgedrückt werden hier alle erzeugten Pakete mit Versionsnummer vorgehalten, so dass u.a. beim Kompilieren einer SWE die zum Kompilieren notwendigen Abhängigkeiten zu anderen SWE in einer spezifischen Version aufgelöst werden können. Der Zugriff auf dieses Repository erfolgt während des Builds vom jeweiligen Entwicklungsrechner nur lesend. Da die Zugriffe sowohl durch die NERZ e.V. bei Erstellen der offiziellen Distributionspakete erfolgt als auch durch die Entwickler der AN, muss der lesende Zugriff über das Internet sichergestellt werden.

Das Ablegen von spezifischen Abhängigkeiten (also z. B. der SWE-xyz in der Version 3.6.4) erfolgt ausschließlich durch den NERZ (FTB).

Für die technische Umsetzung des Dependancy Repository Servers, stehen ebenfalls zwei unterschiedliche Varianten zur Verfügung.

- Speicherung auf einem extern betriebenen Server (z. B. Sonatype Nexus (https:/bintray.com))
- Eigenbetriebener Server durch den NERZ e.V.

Hinsichtlich der Vor- und Nachteile der beiden Varianten gilt sinngemäß dasselbe wie für den Server des Versionsverwaltungsprogramms.

### 6.3.3 Produktserver

Der Produktserver, auf welchem die fertigen Ausgangsprodukte

- Distributionspakete für SWE
- Updateseiten für Plug-in
- Dokumentationen
- aktueller Datenkatalog als HTML-Version auf Basis der aktuellen Konfigurationsverantwortlichen (KV) / Konfigurationsbereiche (KB).
- Konfigurationen (KV, KB)
- (Beispiel-)Systeme

für Anwender der ERZ bereitgestellt werden, sollte hinsichtlich des Zugriffs durch die Nutzer keine spezielle Software erfordern[9].

Vorgeschlagen wird die Bereitstellung über einen (S)FTP-Server und/oder durch einen Webserver (Bereitstellung im Prinzip ähnlich wie jetzt im Produktbereich des NERZ-Servers).

Der Zugriff (nur lesend) ist dabei alle für Anwender frei und ohne irgendeine Registrierung möglich.

Physisch kann dies durchaus derselbe Server sein, auf dem auch die Versionsverwaltung mit GIT-Lab bzw. das Dependancy Repository betrieben wird[10].

### 6.4 Eigenbetriebene Server versus externe Server

In der ursprünglichen Vorgabe für den Aufbau des Versionsverwaltungssystems wurde von den Mitgliedern des NERZ e.V. beschlossen, dass die Daten zwingend auf einem deutschen Server betrieben werden müssen und die Nutzung von ausländischen Servern nicht zulässig ist.

Dies würde bedeuten, dass beim Versionsverwaltungsserver die ersten beiden Varianten (GIT-Hub und Bitbucket) und beim Dependancy Repository Server die Variante mit z. B. Sonatype Nexus nicht genutzt werden können.

Zu klären wäre, ob sich diese Anforderung aus grundsätzlichen datenschutzrechtlichen Vorgaben oder einer zwingenden Vorgabe des Bundes ergibt, oder ob sich dies hier lediglich unter dem Gesichtspunkt der sicheren Verfügbarkeit der Daten ergeben hat.

Für den letzteren Fall kann man davon ausgehen bzw. wird man dies so einrichten, dass alle Repositories auf den drei Servern (Versionsverwaltungssystem, Produktserver und Dependancy Repository Server) regelmäßig (bei jeder Änderung) auf einen lokalen Server, der unter der Kontrolle des NERZ e.V. steht, kopiert werden (bzw. automatisch ein Clone erzeugt wird).

Somit würde zu jedem Zeitpunkt ein vollständiges Backup der Daten zur Verfügung stehen, so dass auch im Ausfall der kommerziellen Versionen die Daten weiterhin uneingeschränkt zur Verfügung stehen. Beim dauerhaften Ausfall (der allerdings aus heutiger Sicht nicht sehr realistisch erscheint) könnte man dann mit diesen Daten kurzfristig eine neue Server Umgebung aufbauen, die dann tatsächlich komplett durch den NERZ e.V. administrieren wird.

## 7 Struktur der Repositories

Für die Repositories auf dem Versionsverwaltungsserver unter GIT werden die nachfolgend beschrieben Strukturen für die unterschiedlichen Quelldaten SWE, Plug-in (Updateseiten), KV und Spezifikationen (Dokumente) vorgeschlagen.

### 7.1 SWE

Analog zur bisherigen Struktur der Distributionspakete der SWE wird je SWE ein eigenes GIT-Repository angelegt, wobei dem Paketnamen der Präfix „SWE_“ vorangestellt wird:

- *SWE_de.bsvrz.segment.swe*[11]

Die interne Struktur entspricht der vom eingesetzten Build-Tool (Maven oder Gradle) geforderten Verzeichnisstruktur.

---

[9] Im Gegensatz zum Zugriff auf den Versionsverwaltungsserver durch Entwickler, bei dem i.d.R. eine geeignete Entwicklungsumgebung mit GIT und Maven oder Gradle installiert sein muss.

[10] Sofern die Entscheidung für einen durch den NERZ e.V. betriebenen Server mit GIT-Lab getroffen wurde.

[11] Bezeichnung analog zur Package-Bezeichnung der SWE gemäß Vorgaben des NERZ e.V.

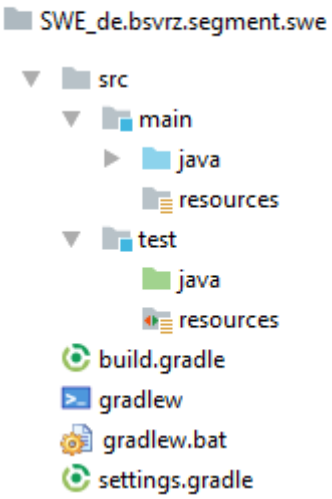



*Abbildung 7-1: Beispiel Verzeichnisstruktur (hier bei Gradle)*

Neben dem eigentlichen Quellcode (`src/main/java/…`) und dem Quellcode der (JUnit) Tests (`src/test/java/…`) werden weiterhin die benötigten Resourcen und die Dateien, die letztlich die Produktbeschreibung (PD) ausmachen (in der Abbildung die Dateien `settings.gradle` und `build.gradle`) im GIT-Repository verwaltet.

Weitere als verbindlich vereinbarte Zusätze wie z.B. Festlegungen zur automatischen Codeanalyse (Checkstyle etc.) werden ebenfalls verwaltet.

Bei der Verwaltung ist (durch die FTB) darauf zu achten, dass weitere Verzeichnisse und Dateien, die lediglich zur Erstellung der Ausgangsprodukte (Distributionspakete etc.) benötigt werden bzw. beim Build-Prozess entstehen, nicht verwaltet werden und somit beim Übernehmen (clonen) eines Entwicklungszweigs in ein Entwickler-Repository auch nicht erzeugt werden[12].

## 7.2 Plug-in

Es gilt hier sinngemäß die gleiche Vorgabe, wie bei den SWE beschrieben, wobei die einzelnen Repositories wie folgt benannt werden

- UpdateSeite_xyz
- UDS_xyz (alternativ, noch festzulegen)

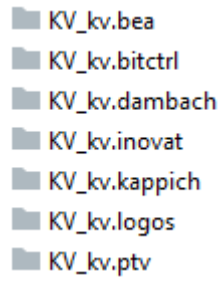
Allerdings sollten bei den Plug-in die einzelnen GIT-Repositories nicht je Plug-in erstellt werden. Stattdessen sollte sich die Festlegung der Repositories an der zu erstellenden Updateseite bzw. Features orientieren, wobei die Updateseiten / Features nach sinnvollen und nicht trennbaren Bestandteilen (Plug-ins) gruppiert werden sollten.

Die Gruppierung sollte also so erfolgen, dass gegenseitig abhängige Plug-ins, die ohnehin nur sinnvoll als Einheit verwendet werden können, zusammengefasst werden. Die Gruppierung sollte so klein wie möglich und so groß wie nötig erfolgen (also nicht ein Repository für alle Plug-in der Firma xyz oder eines bestimmten Projekts).

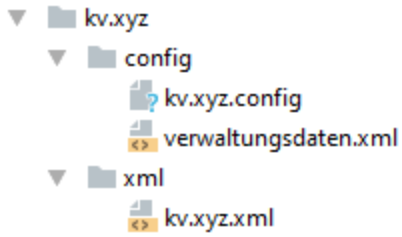
## 7.3 KV

Analog zur bisherigen Struktur der Bereitstellung der KV wird je KV ein eigenes GIT-Repository angelegt, wobei dem KV-Namen der Präfix „KV_“ vorangestellt wird::

[12] Diese Dateien und die zugrundeliegende Struktur sind i.d.R. abhängig von der eingesetzten Entwicklungsumgebung des jeweiligen Entwicklers / AN und somit nicht allgemein verwendbar.




Die interne Struktur entspricht dabei den bisherigen Konventionen:




Versioniert werden sowohl die *.config-Dateien als auch die *.xml-Dateien[13] sowie die aktuelle Verwaltungsdatei[14].

Zusätzlich sollte zukünftig auch ein Dokument mit den aktuellen Release-Notes zu den Änderungen am jeweiligen KV gepflegt und versioniert werden (neben der Änderungsdokumentation in den einzelnen KB.

### 7.4 Spezifikationen (Dokumentation)

Zu jeder SWE und zu jeder Plug-in-Einheit, für die entsprechend der Kapitel 7.1 und 7.2 ein eigenes GIT-Repository angelegt wurde, existiert auch eine vollständige Spezifikation gemäß V-Modell [VMOD97] nebst ggf. zusätzlichen Dokumenten[15]. Diese Spezifikation ist zudem bei Änderung der zugeordneten SWE bzw. Plug-in-Einheit zu aktualisieren.

Zu einer SWE (analog für Plug-in-Einheiten)

- *de.bsvrz.segment.swe*

wird jeweils ein eigenes GIT-Repository mit analoger Namensgebung und dem Präfix **SPEZ_** verwaltet:

- ***SPEZ_****de.bsvrz.segment.swe*

Die interne Struktur bildet dabei die V-Modell-Struktur der Dokumente ab[16]:

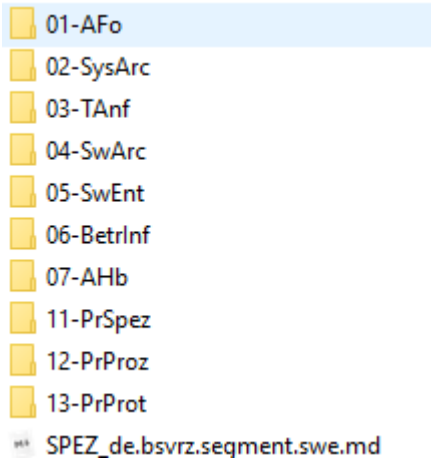
---

[13] Bei den *.xml-Dateien sind jeweils die beim Export der Konfiguration erzeugten Daten zu verwenden, damit diese von Version zu Version aufgrund der einheitlichen Formatierung besser mit entsprechenden Diff-Tools verglichen werden können.

[14] Auch wenn die Verwaltungsdatei ebenso wie die *xml-Dateien für die eigentliche Verwendung eines KV nicht benötigt werden, ermöglich diese den schnellen Überblick über die Versionen der einzelnen KB des KV und ggf. auch, gegen welche Versionen anderer KB eines anderen KV dieser aktuelle KV erzeugt wurde.

[15] Zu mindestens sollte diese existieren.

[16] Grundsätzlicher Aufbau: Details zur Struktur und Namensgebung sind im Rahmen der konkreten Umsetzung noch festzulegen.




Zusätzlich ist jeweils ein gleichnamiges Dokument

- ***SPEZ***_*de.bsvrz.segment.swe.md*

als Beschreibung enthalten (Format: Markdown, *.md), welches die für eine bestimmte Version zugeordneten Dokumente aufführt.

Sind ein oder mehrere Spezifikationen (z.B. AFo, SysArc, ...) nicht genau für diese SWE vorhanden, sondern sind diese in einem (oder mehreren) übergeordneten Dokument(en) z. B. für das gesamte Segment vorhanden, so werden diese bei der Beschreibung referenziert und in einem eigenen Repository verwaltet.

Für die übergeordneten Spezifikationen werden ebenfalls eigene GIT-Repositories angelegt, wobei die Bezeichnung entsprechend modifiziert wird (z. B. für übergeordnete Dokumente die systemweit gültig sind oder die übergeordnete Dokumente zum einem speziellen Segment oder Projekt enthalten):

- *SPEZ_de.bsvrz* (systemweit gültige Dokumente)
- *SPEZ_de.bsvrz.dua* (übergreifend für Segment DUA)
- *SPEZ_LosC1C2-VRZ3* (übergreifend aus dem Projekt LosC1C2-VRZ3)

Der Inhalt der *.md Datei ist nachfolgend beispielhaft für die

- `SWE_de.bsvrz.dua.plformal, FREI_V2.0.2_D2016-07-30`

dargestellt (Markdown-Syntax in der Darstellung bereits interpretiert):

Inhalt
Version: FREI_V2.0.2_D2016-07-30

# Inhalt

Diese Dokument enthalt die Aufstellung der Spezifikationsdokumente für die

- **SWE_de.bsvrz.dua.plformal**

# Version: FREI_V2.0.2_D2016-07-30

**Übergeordnete Dokumente**

| Typ | Repository | Dokument(e) |
|---|---|---|
| AFo | SPEZ_de.bsvrz.dua | AFo_DUA_FREI_V8.0_D2017-04-28 |
| SysArc | SPEZ_de.bsvrz | SysArc_ERZ-Gesamt_FREI_V7.0_D2016-09-30 |
| | SPEZ_LosC1C2-VRZ3 | SysArc_LosC1C2-VRZ3*FREI_V7.0*D2006-01-30 |
| TAnf | SPEZ_de.bsvrz | TAnf_BSVRZ-Gesamt_FREI_V1.0_D2006-04-12 |
| | SPEZ_de.bsvrz.dua | TAnf_DUA_FREI_V3.0_2006-01-21 |
| | SPEZ_LosC1C2-VRZ3 | TAnf_LosC1C2-VRZ3*FREI_V7.0*D2006-01-30 |
| SwArc | SPEZ_de.bsvrz.dua | SE-02.04.00.00.00-SwArc-2.0 [SwArc DUA] |
| SwEnt | --- | --- |
| BetrInf | --- | --- |
| AHb | --- | --- |
| PrSpez | SPEZ_de.bsvrz.dua | PrSpez_DUA_FREI_V5.0_2017-04-28 |
| PrProz | SPEZ_de.bsvrz.dua | PrProz_DUA_FREI_V7.0_2017-04-24 |
| PrProt | SPEZ_de.bsvrz.dua | PrProt_DUA_FREI_V3.0_2017-04-24 |

**Spezifische Dokumente**

| Typ | Dokument(e) |
|---|---|
| AFo | --- |
| SysArc | --- |
| TAnf | --- |
| SwArc | --- |
| SwEnt | SwEnt_DUA-PLFormal_FREI_V5.0_D2016-05-23 |
| BetrInf | BetrInf_DUA-PLFormal_FREI_V8.0_D2017-07-17 |
| AHb | entfällt, da Softwarebibliothek |
| PrSpez | --- |
| PrProz | --- |
| PrProt | --- |

*Abbildung 7-2: Inhalt einer Beschreibungsdatei *.md für Spezifikationen*

# 8 Festlegungen zur Versionierung von Teilprodukten

## 8.1 Quellcode

Der Quellcode für jedes einzelne Distributionspaket (also i. d. R. je SWE, Plug-in) wird jeweils in einem eigenen Repository verwaltet.

Bei der Versionsnummer ist diese in der Form mit Hauptversionsnummer (major release), Nebenversionsnummer (minor release) und Revisionslevel (patch level) anzugeben.

Wird eine neue Version in den Masterzweig des Repositories übernommen (durch die FTB), so wird der entsprechende Stand mit einem TAG aus Versionsnummer, Status und Stand in der Form

STATUS_Vx.y.z_DJJJJ-MM-TT versehen.

Beispiel:

- FREI_V2.10.5_D2017-02-04

Hauptversionsnummer: Die Hauptversionsnummer sollte bei signifikanten Änderungen / Erweiterungen erhöht werden oder wenn sich keine Abwärtskompatibilität mehr herstellen lässt[17].

Nebenversion: Die Nebenversion wird bei funktionalen Erweiterungen erhöht, wobei aber die Abwärtskompatibilität bzw. die Schnittstellenkompatibilität zu den älteren Versionen mit derselben Hauptversionsnummer erhalten bleibt.

Revisionslevel: Der Revisionslevel wird bei Änderungen (i.d.R. Fehlerbehebungen) erhöht, die ansonsten keinerlei funktionale Änderungen beinhalten.

Jede dieser Versionsnummern kann auch aus mehreren Ziffern bestehen. Zum Beispiel folgt nach Version 0.9, wenn sich nur die Nebenversion erhöht, 0.10 und nicht 1.0.

## 8.2 Spezifikationen (Dokumentation)

Zu jeder SWE und zu jeder Plug-in-Einheit, für die entsprechend der Kapitel 7.1 und 7.2 ein eigenes GIT-Repository angelegt wurde, existiert auch eine vollständige Spezifikation gemäß V-Modell nebst ggf. zusätzlichen Dokumenten[18]. Diese Spezifikation ist zudem bei Änderung der zugeordneten SWE bzw. Plug-in-Einheit zu aktualisieren.

Die Version einer Spezifikation sollte deshalb i.d.R. der Version der spezifizierten SWE / Plug-in-Einheit entsprechen. Dies bedeutet, dass die Spezifikation mit der gleichen Versionsnummer wie die SWE / Plug-in-Einheit diese korrekt dokumentiert.

Bei Änderungen der SWE / Plug-in-Einheit und daraus resultierender neuer Version sollte die Version der zugeordneten Spezifikation deshalb

- ebenfalls angepasst werden, wenn die Änderungen an der SWE / Plug-in-Einheit **KEINE** Anpassung der Spezifikation erfordern. Dadurch wird ersichtlich, dass Spezifikation und SWE /

[17] Insbesondere die Erhöhung der Hauptversionsnummer sollte genau überlegt werden. Bei Aktualisierung des RW2 auf die aktuelle Eclipse-Version wurde z.B. das RW auf die Version 3 erhöht (RW3), obwohl hier die Aufwärtskompatibilität gewährleistet ist. Dies führt jetzt bei der Verwendung von Updateseiten, die nur als kompatibel mit Version 2.x.x gekennzeichnet sind zu Problemen. Diese können nicht mehr ohne Anpassungen der entsprechenden Kompatibilitätseinstellungen im RW3 verwendet werden, obwohl dies eigentlich problemlos möglich wäre.

[18] Zu mindestens sollte diese existieren.

Plug-in-Einheit zueinander konform sind (in diesem Fall wird lediglich ein zusätzliches TAG mit der neuen Versionsnummer vergeben).

- NICHT angepasst werden, wenn die Änderungen an der SWE / Plug-in-Einheit **KEINE** Anpassung der Spezifikation erfordern. Dadurch wird ersichtlich, dass Spezifikation und SWE / Plug-in-Einheit zueinander nicht mehr bzw. nur bis zur entsprechenden Version konform sind. In diesem Fall ist die Spezifikation aber unbedingt zu aktualisieren.

### 8.3 Distributionspakete

Da die Distributionspakete jeweils aus dem Quellcode der SWE/Plug-ins erzeugt werden (in Abhängigkeit von bereits erzeugten und versionierten Distributionspaketen (die auf dem speziellen Dependancy Repository Server veröffentlicht sind, siehe Kapitel 6.3.2) erhalten diese jeweils die identische Versionsnummer wie der zugehörige Sourcecode.

### 8.4 Konfigurationsverantwortliche (KV)

Für jeden KV (kv.bitctrl, kv.inovat, …) wird jeweils ein eigenes Repository angelegt. Mit jeder Bereitstellung eines aktualisierten KV (also aller KB, die von diesem KV verwaltet werden), wird dieser Stand als aktuelle Version für diesen KV versioniert. Da ein solcher KV als vielen einzelnen KB mit jeweils unterschiedlichen Versionen[19] vorliegt, erfolgt die Versionsbezeichnung über den Stand Datum) des KV und zusätzlich, sofern durch den jeweils Verantwortlichen benannt, eine Versionsangabe[20].

Wird eine neue Version in den Masterzweig des Repositories übernommen (durch die FTB), so wird der entsprechende Stand mit einem TAG aus Status (optional), Versionsnummer (optional) und Stand in der Form

*STATUS_Vx.y.z*_DJJJJ-MM-TT versehen.

Beispiele:

- FREI_V2.10.5_D2017-02-04
- V3.8.0_ D2017-02-04
- D2017-02-04

### 8.5 Konfigurationsbereiche (KB)

Optional:

Für jeden KB wird zusätzlich jeweils ein eigenes Repository angelegt. Mit jeder Bereitstellung eines aktualisierten KV (also aller KB, die von diesem KV verwaltet werden), werden die einzelnen KB dieses KV, sofern sich ihre Version[21] geändert hat, versioniert (jeweils die *config und die exportierte *.xml Datei)

Dadurch kann im Bedarfsfall ein KB in einer spezifischen (i. d. R. älteren) Version in eine Gesamtkonfiguration integriert werden. Dies kann notwendig sein, weil man z. B. die aktuelle Version eines KB wegen Inkompatibilitäten (noch) nicht verwenden kann.

[19] Gemeint ist hier die Versionsnummer für einen KB, wie sie von der Konfigurationsapplikation vergeben wird und in der verwaltungsdaten.xml eingetragen ist.

[20] Wie dies z. B. bisher beim kv.kappich erfolgt, da dieser KV i. d. R. mit der Version des aktuellen Kernsystemrelease verbunden ist.

[21] Gemeint ist hier die Versionsnummer für einen KB, wie sie von der Konfigurationsapplikation vergeben wird und in der verwaltungsdaten.xml eingetragen ist.

**Der naheliegende „Trick“, die gewünschte Version händisch in die verwaltungsdaten.xml einzutragen, funktioniert wegen eines Fehlers in der Konfigurationsapplikation nicht[22]!**

Versionsbezeichnung (Beispiel):

- kv.tmVerkehrGlobal, Version 49

### 8.6 Datenkatalog

Optional:

Für jeden Datenkatalog, also eine konkrete Zusammenstellung aus diversen KV und KB einschließlich der zugehörigen verwaltungsdaten.xml wird ein eigenes Repository angelegt und verwaltet.

Datenkataloge können z. B. für unterschiedliche Beispielsysteme (z. B. „minimales Basissystem basierend auf Kernsoftware Version 3.9.1“, „Beispielsystem XYZ auf Basis Kernsoftware 3.8.0“, etc.) angelegt und verwaltet werden.

## 9 Bereits getroffene Festlegungen / Nächste Schritte

- Festlegung des zu einzusetzenden Build-Tools (Maven oder Gradle)
  - Gemäß FachTelko vom 19.05.2017 wurde die Verwendung von Gradle beschlossen
- Festlegungen zu den Servern (siehe dazu auch Kapitel 6.4)
  - GIT-Lab / GIT-Hub / Bitbucket
    - Gemäß MitgliederTelko vom 19.05.2017 wurde die Verwendung von GitLab auf einem durch die GS verwalteten Server festgelegt.
    - Der Server ist bereits eingerichtet
  - Produktserver
    - Noch keine abschließende Festlegung erfolgt.
  - Dependancy Repository Server
    - Noch keine abschließende Festlegung erfolgt.
- Einrichtung Versionsmanagement
  - Die Repositories für die SWE wurden bereits eingerichtet.
  - Die Repositories für die KV wurden bereits eingerichtet.
  - Die Repositories für die Spezifikationen müssen noch eingerichtet werden (es wurden bisher nur einzelne Repositories erstellt).
  - Die Repositories für die Updateseiten (BuV-Plug-ins) müssen noch eingerichtet werden.
  - Die Erstellung der Gradle-Umgebung nebst Einrichtung des Produktserver und des Dependancy Repository Servers ist noch nicht erfolgt.

## 10 Anhang „Abbildungen“

- Lösungsskizze Konfigurationsmanagement NERZ e.V
- Prozesse – Aktivitäten Konfigurationsmanagement NERZ e.V.
- Server für das Konfigurationsmanagementsystem des NERZ e.V.

[22] abgesehen von der Problematik, dass dann auch die Funktionen zur „Konsistenzprüfung“ und zur „Aktivierung“ sowie der „Export“ nicht angewendet werden können.