TO-Generator aus JPA-Entities

Spezifikation

Jens Ebert

Status: - in Arbeit -

Version: 0.1

23. März 2015

1 Einleitung

Dieses Dokument stellt eine Kurzspezifikation eines Hilfswerkzeugs für die Entwicklung einer JavaEE-Anwendung dar. Es handelt sich dabei um einen Generator, der aus einer gegebenen Java-Klasse, die einer Standard-JPA-Entität entspricht, auf Basis von Templates andere Java-Klassen generieren kann, z.B. „Transport-Objekte“ (TOs) oder „Data Access Objects“ (DAOs). Arbeitstitel des Werkzeuges ist *„TO from JPA entity generator“*.

Grundlegende Anforderungen an das Werkzeug:

- Alle Standard-JPA-2.1-konformen Entitäten können als Input verwendet werden; die zu verwendenden Entitäten werden über Quellpfad und dort befindliche Quellpackages sowie weitere Einschränkungsmöglichkeiten angegeben.
- Alle JPA-relevanten Informationen werden aus der Entität geparst und können dann über *Template-Variablen* in beliebig vielen *Templates* verwendet werden.
- Ein Template ist eine Schablone einer durch den Generator zu erzeugenden Java-Klasse, die im Rahmen des Generierungsprozesses auf Basis der gelesenen Entitäten expandiert wird. Aus einer Schablone können dabei N generierte Java-Klassen entstehen.

Ausschlüsse der Initialversion 0.5:

- Es werden nur annotierte Entitäten unterstützt, XML-Mappings können nicht ausgewertet werden.
- Mapped superclasses oder Entitäts-Superklassen werden bei Generierung nicht berücksichtigt.
- Der Generator navigiert Beziehungen zwischen Entitäten nicht explizit. Ist eine in Beziehung zur Quellentität stehende Entität nicht in den angegebenen Quellpaketen der Generierung enthalten, so wird der Typ der referenzierten Fremd-Entität in die generierten Klassen übernommen. Für die Fremd-Entität erfolgt dann keine Generierung.
- Spezielle Annotationen von Implementierungen (Hibernate, OpenJPA etc.) werden ignoriert.
- Der Generator erzeugt immer komplett neue Klassen, ein Mergen in bereits generierte Klassen ist nicht möglich.
- Annotationen an Accessor-Methoden statt an Attributen in einer Entität werden nicht unterstützt, die JPA-Annotationen müssen ausschließlich an Feldern verwendet werden.
- Eingebettete Klassen werden momentan nicht geparst.

- Eine grundlegende Erweiterbarkeit durch eigene Template-Generierungsregeln wird aktuell noch nicht unterstützt.

2 Grundkonzepte

Hier werden die Grundkonzepte des Generators beschrieben.

2.1 Entities

Der Generator arbeitet auf Basis von JPA-Entities. JPA-Entities sind Java-Klassen, die mit der Annotation @javax.persistence.Entity annotiert worden sind und weiteren Regeln entsprechen, siehe dazu [**?**][JPA21]. Eingabe des Generators sind 0 bis n Entitäten. Der Generator bearbeitet die Entitäten sequentiell (die Reihenfolge der Abarbeitung ist nicht definiert) und liest aus jeder Entität relevante Informationen aus. Diese Informationen werden dann benutzt, um auf Basis gegebener Templates Ziel-Javaklassen zu generiern. Die Quell-Entitäten werden bei diesem Prozess nicht verändert.

Details zu JPA werden hier nicht erläutert, sondern als bekannt vorausgesetzt. Siehe [**?**][JPA21] sowie [**?**][J2EEDocs].

2.2 Templates

Ein **Template** ist eine Text-Datei, die den (Pseudo-)Code einer Java-Klasse enthält. Das Template wird während des Generierungsprozesses geparst, nach bestimmten Kriterien wird eine Kopie des Templates angelegt, und diese auf Basis der gelesenen Quell-Entitäten mit Informationen angereichert. Dieser Prozess wird **Expansion** genannt. Resultat der Expansion eines Templates sind im Idealfall[1] 0 bis N kompilierende Java-Klassen, oder zumindest 0 bis N syntaktisch korrekte Java-Klassen, die in ein konfiguriertes Ausgabeverzeichnis geschrieben worden sind. Aus einem Template können also mehrere Ergebnis-Klassen resultieren.

Ein Template enthält neben normalem Java-Source-Code folgende spezielle Elemente, die in den darauffolgenden Abschnitten näher definiert werden:

- Template-Variablen
- Generator-Direktiven
- Expansions-Blöcke

[1] Wie später erläutert wird, kann es sein, dass nicht alle Template-Variablen während der Expansion ersetzt werden konnten, sodass Syntaxfehler in der generierten Java-Klasse die Folge sind.

2.2.1 Template-Variablen

Template-Variablen sind Strings, die folgender Syntax folgen: {`$varname`}, wobei `varname` der Name der Template-Variablen ist. Außer der schließenden geschweiften Klammer „}“ dürfen alle nur denkbaren Zeichen des Unicode-Zeichensatzes in `varname` vorkommen. Es wird jedoch empfohlen, sich auf ASCII-Zeichen und idealerweise nur auf alphanumerische Zeichen zu beschränken.

Eine an einer bestimmten Stelle in einem Template verwendete Template-Variable mit Namen `varname` wird als *Instanz* der Template-Variable[2] mit dem Namen `varname` bezeichnet. Es dürfen beliebig viele Instanzen einer Template-Variablen in einem Template vorkommen. Zudem dürfen beliebig viele verschiedene Template-Variablen in einem Template verwendet werden. Die Position einer Instanz einer Template-Variablen im Template ist beliebig. Template-Variablen sollten jedoch nicht ineinander geschachtelt werden[3].

Während des Generierungsprozesses werden die Instanzen einer Template-Variablen entsprechend ihres Scopes (Definition des Begriffes siehe im entsprechenden Abschnitt über Scopes) mit Werten belegt, sofern im aktuellen Scope ein Wert für die Variable verfügbar ist. „Mit Werten belegt“ heißt, dass alle Vorkommen des Strings {`$varname`} im gesamten aktuellen Scope durch einen anderen String, den Wert der Variablen im aktuellen Scope, ersetzt werden. Ist für die Template-Variable kein Wert im aktuellen Scope definiert, so findet keine Ersetzung der Template-Variablen statt (d.h. die resultierende Java-Klasse weist dann an jeder Stelle Syntax-Fehler auf, an der die Template-Variable verwendet wird und nicht ersetzt werden konnte).

Der Anwender sollte also durch korrekte Verwendung des Generators dafür Sorge tragen, dass alle verwendeten Template-Variablen auch expandiert werden.

Der Anwender wird durch Warnungen darauf hingewiesen, dass bestimmte Template-Variablen nicht ersetzt werden konnten.

2.2.2 Generator-Direktiven

Generator-Direktiven sind spezielle Hinweise an den Generator innerhalb von Templates, die den Generierungsprozess beeinflussen bzw. bestimmte Bereiche in einem Template hervorheben. Sie werden im Template mittels folgender Syntax definiert: `[$name` `property`$_1$`=value`$_1$ ... `property`$_n$`=value`$_n$`]`. Dabei ist `name` der Identifier der Direktive, und `property`$_1$`=value`$_1$ bis `property`$_n$`=value`$_n$ sind Schlüssel-Wert-Paare, welche bestimmte Eigenschaften der Direktive angeben. Diese Eigenschaften sind fest je Direktive definiert.

[2] Statt von einer Instanz wird häufig vereinfachend einfach von einer Template-Variablen oder einer Variablen gesprochen. Der Begriff Instanz wird nur dann verwendet, wenn sich besonders auf genau ein Vorkommen einer Template-Variablen in einem Template bezogen wird.

[3] Genauer: Bei einem Konstrukt der Form {`$var{$innervarname}name`} wird eine Variable namens `innervarname` expandiert. Die äußere Variable `varname` wird nicht expandiert. Der Name der (neuen) Templatevariablen nach Expansion von `innervarname` wird ggf. expandiert, jedoch nur dann, wenn es sich um eine definierte Variable handelt und sie zufällig nach innervarname expandiert wird. Die Reihenfolge der Expansion ist undefiniert. Daher wird davon abgeraten, Template-Variablen zu verschachteln.

In Version 1.0 wird neben den Expansions-Blöcken (die als spezielle Direktiven angesehen werden können) nur eine weitere Generator-Direktive unterstützt: `[$SCOPE id=`*`scopename`*`, qualifier=`*`qualifier`*`]`. Hierbei gilt:

- *`scopename`* ist ein Platzhalter, für den in einem validen Template einer der folgenden Werte genutzt werden muss: `global`, `subsys`, `entity`, `field`. Erläuterungen des Begriffes Scope siehe Abschnitt über Scopes.

- *`qualifier`* ist ein Scope-abhängiger Parameter, der Scope-Einschränkungen ermöglicht. Für jeden Scope sind bestimmte qualifier zulässig. Diese ermöglichen es, Einfluss darauf zu nehmen, welches Feld, welche Entitäten oder welches Subsystem zu einer Expansion eines Templates oder eines Expansions-Blockes führen, und welche nicht. Mit Version 0.5 ist noch kein qualifier definiert. D.h. dass die Eigenschaft qualifier entweder weggelassen werden muss, oder nur ein leerer String als Wert verwendet werden darf.

Für alle Generator-Direktiven gilt: Sie werden in jedem Fall während des Generierungsprozesses entfernt und gelangen niemals in die generierte Zielklasse.

2.2.3 Expansions-Blöcke

Expansions-Blöcke sind spezielle Generator-Direktiven, die im Wesentlichen den Kontrollstrukturen einer typischen imperativen Programmiersprache entsprechen, jedoch natürlich deutlich weniger mächtig sind. Sie werden in Templates verwendet, um bedingte oder auch wiederholte Expansion des im Inneren des Blocks eingeschlossenen Textes zu ermöglichen. Man spricht also sowohl von der Expansion eines gesamten Templates als auch von der Expansion eines einzelnen Expansions-Blockes.

Mit Version 0.5 werden folgende Expansions-Blöcke unterstützt: Repeat-Blöcke und Conditional-Blöcke.

Repeat-Block

Ein **Repeat-Block** ermöglicht die Definition von Bereichen, die wiederholt in einem Template expandiert werden. Ein Repeat-Block ist immer an einen Scope gebunden. Die Syntax eines Repeat-Blocks ist folgende: `[$REPEAT scope=`*`scopename`* `qualifier=`*`qualifier`*`]` Hierbei gilt:

- *`scopename`* ein Platzhalter, für den in einem validen Template einer der folgenden Werte genutzt werden muss: `global`, `subsys`, `entity`, `field`. Erläuterungen des Begriffes Scope siehe Abschnitt über Scopes.

- *`qualifier`* ist ein Scope-abhängiger Parameter, der Scope-Einschränkungen ermöglicht. Für jeden Scope sind bestimmte qualifier zulässig. Diese ermöglichen es, Einfluss darauf zu nehmen, welches Feld, welche Entitäten oder welches Subsystem zu einer Expansion eines Templates oder eines Expansions-Blockes führen, und welche nicht. Mit Version 0.5 ist noch kein qualifier definiert. D.h. dass die Eigenschaft qualifier entweder weggelassen werden muss, oder nur ein leerer String als Wert verwendet werden darf.

Conditional-Block

Ein **Conditional-Block** ermöglicht die Definition von Bereichen, die bei Eintreten einer Bedingung in einem Template expandiert werden. Die Syntax eines Conditional-Blocks ist folgende: `[$COND condition=`*`condition`*`]` Dabei ist *`condition`* ein boolscher Ausdruck entspricht mit Einschränkungen der Java-Syntax. Es dürfen konkret folgende Symbole verwendet werden:

- `true` sorgt dafür, dass der Conditional-Block immer expandiert wird
- `false` sorgt dafür, dass der Conditional-Block nie expandiert wird
- Verwendbare binäre Operatoren: `==, !=, &&, ||`
- Verwendbare unäre Operatoren: `!`
- Alle Template-Variablen {`$varname`} des aktuellen Scopes
- Das spezielle Literal null bezeichnet eine im aktuellen Scope nicht definierte Variable
- Klammerausdrücke, um mehrere binäre oder unäre Ausdrücke zu verbinden und Vorrangregeln zu umgehen
- In einfachen Hochkommata eingeschlossene Ausdrücke bezeichnen literale Strings, somit ist auch der leere String und der literale String 'null' angebbar.

Da Template-Variablen nur als Strings betrachtet werden, sind andere Operatoren zwar vorstellbar, aber sind für Version 0.5 ausgeschlossen. Insbesondere sind ausgeschlossen:

- Ungleichungsoperatoren `<,>,<=,>=`
- Rechenoperatoren `+, -, *, /, \%`
- Binäre Operatoren `&, |, ^`
- Der tertiäre Operator `? :`
- Sonstige Java-Ausdrücke wie Zuweisungen, Array-Ausdrücke, Methodenaufrufe etc.

2.2.4 Schachteln von Expansions-Blöcken

Expansions-Blöcke dürfen grundsätzlich beliebig tief ineinander geschachtelt werden. Schachtelung heißt, dass ein weiterer Expansions-Block im Inneren eines anderen Expansions-Blocks verwendet wird. Es gelten folgenden Möglichkeiten und Bedingungen:

- Conditional-Blöcke dürfen beliebig tief ineinander verschachtelt werden.
- Conditional-Blöcke dürfen beliebig in Repeat-Blöcken verschachtelt werden.
- Repeat-Blöcke dürfen beliebig in Conditional-Blöcken verschachtelt werden.
- Ein Repeat-Block darf nur dann in einen anderen Repeat-Block verschachtelt werden, wenn er einen niedrigeren Scope hat. Z.B. darf ein Repeat-Block mit Scope `entity`

nicht in einem anderen Repeat-Block mit Scope `field` oder in einem anderen Repeat-Block mit Scope `entity` verwendet werden, wohl aber in Repeat-Blöcken der Scopes `global` und `subsys`.

2.3 Scopes

Ein **Scope** bezeichnet den Gültigkeitsbereich des aktuellen Templates oder Repeat-Blockes. Dieser gibt an, wie oft ein Template oder Repeat-Block expandiert wird. Ein Scope bezieht sich immer auf die Gesamtmenge der für einen Generierungslauf verwendeten Quell-Entitäten. Die folgenden Scopes sind insgesamt verfügbar:

- **Field:** Bezieht sich auf genau ein Feld (=Java-Attribut) genau einer Quell-Entitäts-Klasse
- **Entity:** Bezieht sich auf genau eine Quell-Entitäts-Klasse
- **Subsystem:** Bezeichnet eine nach bestimmten Kriterien umrissene Teilmenge der Quell-Entitäten
- **Global:** Globaler Scope, unabhängig von Subsystemen, Entitäten oder Feldern

Für jeden Scope sind bestimmte qualifier zulässig. Diese ermöglichen es, Einfluss darauf zu nehmen, welches Feld, welche Entitäten oder welches Subsystem zu einer Expansion des Templates oder des Repeat-Blockes führen und welche nicht. Die Liste der verfügbaren Qualifier mit ihren Bedeutungen findet sich oben bei der Beschreibung der Generator-Direktive `SCOPE`.

2.3.1 Template-Scope

Der Template-Scope wird durch die verpflichtende Generator-Direktive `SCOPE` (Beschreibung siehe Abschnitt oben) in der ersten Zeile eines Templates genau einmal definiert. Mit Version 0.5 des Generators gilt:

- **Field:** Das Template wird genau einmal für jedes Feld der Quell-Entität expandiert. Es wird jedes Feld verwendet, unabhängig davon, ob es transient oder persistent ist.
- **Entity:** Das Template wird genau einmal für jede Quell-Entität expandiert. Dabei spielt keine Rolle, ob die Quell-Entität selbst abstrakt oder Basisklasse einer anderen Entität ist.
- **Subsystem:** Das Template wird genau einmal je Subsystem expandiert (sofern in der Konfiguration die verfügbaren Subsysteme definiert worden sind).
- **Global:** Das Template wird genau einmal expandiert.

2.3.2 Scope von Repeat-Blöcken

Der Scope von Repeat-Blöcken wird mittels der Eigenschaft `scope` definiert und kann einen der oben genannten Werte annehmen. Mit Version 0.5 des Generators gilt:

- **Field:** Der Repeat-Block wird genau einmal für jedes Feld der Quell-Entität expandiert. Es wird jedes Feld verwendet, unabhängig davon, ob es transient oder persistent ist.
- **Entity:** Das Repeat-Block wird genau einmal für jede Quell-Entität expandiert. Dabei spielt keine Rolle, ob die Quell-Entität selbst abstrakt oder Basisklasse einer anderen Entität ist.
- **Subsystem:** Das Repeat-Block wird genau einmal je Subsystem expandiert (sofern in der Konfiguration die verfügbaren Subsysteme definiert worden sind).
- **Global:** Der Scope global wird unterstützt, führt aber lediglich dazu, dass der Inhalt des Repeat-Blockes genau einmal expandiert wird.

2.3.3 Scope-Abhängigkeit von Template-Variablen und Hierarchie

Eine Template-Variable ist an einen Scope gebunden. Bindung an einen Scope bedeutet: Der Wert der Variablen ist nur in einem gegebenen Scope gültig bzw. unveränderlich. Ist bei der Definition einer Template-Variablen kein Scope angegeben, so hat sie standardmäßig globalen Scope. Es gilt:

- Template-Variablen mit Scope `global` sind während des gesamten Generierungsvorganges unveränderlich, ändern ihren Wert also nicht. Sie dürfen völlig uneingeschränkt in jedem Template und jedem Scope verwendet werden. Man kann diese als Konstanten betrachten.
- Template-Variablen mit Scope `subsys` haben nur innerhalb von Templates oder Repeat-Blöcken mit Scope `subsys`, `entity` oder `field` einen Wert. Sie können in all diesen Scopes verwendet werden. Werden sie im Scope `global` verwendet, werden sie nicht expandiert. Sie ändern ihren Wert potentiell je Subsystem.
- Template-Variablen mit Scope `entity` haben nur innerhalb von Templates oder Repeat-Blöcken mit Scope `entity` oder `field` einen Wert. Sie können in all diesen Scopes verwendet werden. Werden sie in den Scopes `global` oder `subsys` verwendet, werden sie nicht expandiert. Sie ändern ihren Wert potentiell je Quell-Entität.
- Template-Variablen mit Scope `field` haben nur innerhalb von Templates oder Repeat-Blöcken mit Scope `field` einen Wert. Sie können nur in diesem Scope verwendet werden. Werden sie in den Scopes `global`, `subsys` oder `entity` verwendet, werden sie nicht expandiert. Sie ändern ihren Wert potentiell je Feld einer jeden Quell-Entität.

2.4 Vordefinierte Template-Variablen

Je Scope gibt es einen Satz vordefinierter Template-Variablen. Diese sind während des Generierungsprozesses mit definierten Werten (üblicherweise aus den Quell-Entitäten stammend) belegt. Die folgende Tabelle gibt die vordefinierten Template-Variablen je Scope mit Version 0.5 an:

Template-Variable	**Scope**	**Erläuterung**
gen.name	`global`	Name des Generators, der für die Generierung eingesetzt wird.
gen.version	`global`	Version des Generators, die für die Generierung eingesetzt wird.
gen.date.start	`global`	Datum des Startzeitpunktes des aktuellen Generatorlaufs, formatiert in aktueller locale.
gen.date.curr	`global`	Aktuelles Datum, an dem die Expansion des aktuellen Templates gestartet wurde, formatiert in aktueller locale.
gen.time.start	`global`	Uhrzeit des Startzeitpunktes des aktuellen Generatorlaufs, formatiert in aktueller locale.
gen.time.curr	`global`	Uhrzeit, an dem die Expansion des aktuellen Templates gestartet wurde, formatiert in aktueller locale.
ent.name	`entity`	Name der Quell-Entität (entspricht dem name-Attribut der @Entity-Annotation).
ent.type	`entity`	Simple name der Quell-Entitäts-Klasse.
ent.type.trim	`entity`	Simple name der Quell-Entität-Klasse, gekürzt um Präfix und Suffix.
ent.type.pkg	`entity`	Package der Quell-Entität.
ent.type.full	`entity`	Shortcut für `${ent.type.pkg}.${ent.type}`.
ent.ver.name	`entity`	Name des @Version annotierten Feldes der Quell-Entität (falls vorhanden).
ent.ver.type	`entity`	Simple name des Typs des @Version annotierten Feldes der Quell-Entität (falls vorhanden).

Template-Variable	**Scope**	**Erläuterung**
ent.ver.type.pkg	`entity`	Package des Typs des @Version annotierten Feldes der Quell-Entität (falls vorhanden).
ent.ver.type.full	`entity`	Shortcut für `${ent.ver.type.pkg}.${ent.ver.type.name}`.
field.name	`field`	Name des Feldes.
field.name.u	`field`	Name des Feldes, erster Buchstabe ist Großbuchstabe.
field.type	`field`	Java-Typname des Feldes.
field.type.pkg	`field`	Package des Java-Typnamens des Feldes.
field.type.full	`field`	Shortcut für `${field.type.pkg}.${field.type.name}`.

Tabelle 2.1: Vordefinierte Template-Variablen

2.5 Benutzerdefinierte Template-Variablen

Neben den vordefinierten Template-Variablen dürfen benutzerdefinierte Template-Variablen in der Generator-Konfiguration vorgegeben werden. Bei der Definition benutzerdefinierter Template-Variablen kann der Scope der Variable angegeben werden (default bei fehlender Scope-Angabe: global).

Es gelten folgende Regeln für die Definition benutzerdefinierte Template-Variablen:

- Die benutzerdefinierte Template-Variable darf nicht dem Namen einer der vordefinierten Template-Variablen entsprechen - völlig unabhängig vom Scope. Template-Variablen, die mit reservierten Namensbereichen beginnen, sollten vermieden werden, da sie zukünftig zu vordefinierten Template-Variablen werden könnten. Reservierte Namensbereiche in Version 0.5 sind: `gen.`, `sub.`, `ent.`, `field.`, `constr.`, `assoc.`, `attr.`, `entl.`, `join.`, `pkjoin.`, `sec.`, `namq.`, `namnq.`, `sqlr.`.

- Eine benutzerdefinierte Template-Variable darf nur einem in der Generator-Konfiguration definiert werden.

- Eine benutzerdefinierte Template-Variable darf den Wert einer anderen Template-Variablen (benutzerdefiniert oder vordefiniert) über die Syntax {`$varname`} referenzieren. Es ist dabei irrelevant, ob die verwendeten benutzerdefinierten Variablen vorher oder nachher in der Generator-Konfiguration definiert worden sind. Es dürfen nur existierende Variablen verwendet werden. Natürlich dürfen auch mehrere verschiedene oder mehrere gleiche Instanzen von anderen Template-Variablen im Wert der Variablen genutzt werden. Verwendung des eigenen Variablennamens im eige-

nen Wert ist nicht zulässig. Alle im Wert verwendeten Template-Variablen müssen denselben Scope haben wie die Variable selbst.

- Eine benutzerdefinierte Template-Variable darf unabhängig von ihrem Scope eine beliebige Java runtime property referenzieren. Dies geschieht mit der Syntax {`$prop:propertyname`}. Ist eine solche Java runtime property nicht definiert, so wird der leere String als Wert verwendet.

- Eine benutzerdefinierte Template-Variable darf nicht den Namen einer Konfigurations-Property des Generators verwenden. Folgende Präfixe für Konfigurations-Properties sind aktuell reserviert und sollten daher nicht als Beginn eines Namens einer vordefinierten Variable verwendet werden: `prop.`.

Aufgrund all der oben genannten Einschränkungen wird empfohlen, alle benutzerdefinierten Template-Variablen mit Präfix `usr.` zu beginnen. Es wird garantiert, dass weder Konfigurations-Properties noch vordefinierte Template-Variablen jemals mit diesem Präfix beginnen. Der Generator schreibt eine Warnung, wenn er eine benutzerdefinierte Variable ohne diesen Präfix vorfindet.

Die Syntax und Beispiele für die Definition benutzerdefinierter Variablen findet sich im Abschnitt über die Konfiguration des Generators.

2.6 Subsysteme

Die Variable `sub.name` mit Scope `subsys` ist ein Sonderfall: Es handelt sich um eine Mischung aus vordefinierter und benutzerdefinierter Variablen. Vordefiniert, weil sie - falls definiert - eine besondere Bedeutung und immer Scope `subsys` hat. Sie gibt für das aktuelle Subsystem den Namen des Subsystems an. Sie ist optional, muss also keinen Wert haben. Dies bedeutet: Es gibt keine Subsysteme, oder: Es gibt genau ein namenloses Subsystem unter `global`.

Wie entstehen Subsysteme? Es handelt sich nicht um etwas, das automatisiert aus den Quell-Entitäten extrahiert werden kann. Ein Subsystem ist eine Menge zusammengehörender Entitäten. Gibt es nur ein Subsystem, so muss es alle Entitäten enthalten. Gibt es mehrere Subsysteme, so müssen sie disjunkt sein, ihre Vereinigung muss alle Entitäten überdecken. Die Angabe aller vorhandenen Subsysteme erfolgt durch den Anwender auf genau eine der folgenden Arten:

- Verwendung der Variablen {`$sub.name`} in der Konfigurations-Property `prop.src.pkg` (Details siehe bei der Beschreibung der property). Dabei entspricht ein Subsystem im Wesentlichen einem Package, und alle Entitäten in diesem Package gehören zum Subsystem.

- Definition aller Subsysteme in der Konfigurations-Datei. Dabei müssen Variablen verwendet werden, die mit `sub.` beginnen. Der String nach dem Punkt bezeichnet den Namen des Subsystems. Dieser darf außer „name“ alel Werte annehmen. Ein Subsystem darf allerdings nur einmal als Variable in der Konfigurationsdatei definiert werden. Als Wert eines jeden Subsystems darf eine Semikolon-separierte Liste von Package- und vollqualifizierten Klassennamen (die Entitäten) verwendet werden.

Dabei sind Wildcards erlaubt.

Beide Konfigurationsarten schließen sich gegenseitig aus.

2.7 Konfiguration des Generierungsprozesses

Die Konfiguration des Generierungsprozesses erfolgt über eine properties-Datei namens jtoGenerator.properties. Bei Ausführung des Generators wird eine Datei dieses Namens im aktuellen Classpath gesucht. Konnte keine Datei gefunden werden, wird versucht, die verpflichtenden Konfigurations-Properties in den aktuellen Java runtime properties zu finden. Werden sie dort nicht gefunden, wird der Generator mit einer Fehlermeldung beendet. Werden hingegen mehrere Dateien des Namens im Classpath gefunden, so wird eine (nicht deterministisch) ausgewählt, und der Generator schreibt eine entsprechende Log-Warnung.

Eine Konfigurations-Property ist entweder verpflichtend oder nicht. Werden verpflichtende properties nicht angegeben, so führt dies zum sofortigen Abbruch des Generierungsprozesses mit einer Fehlermeldung. Werden nicht-verpflichtende properties weggelassen (d.h. sie tauchen gar nicht in der properties-Datei auf bzw. werden nicht als runtime property gesetzt), so werden hierfür die definierten Default-Werte verwendet.

In den folgenden Abschnitten ist beschrieben, welche Konfigurations-Properties angegeben werden können.

2.7.1 Generelles zu Konfigurationsproperties

Es gibt mehrere Konfigurations-Properties, die denselben Typ haben bzw. eine ähnliche Art und Weise von Wertangaben haben. Diese Gemeinsamkeiten werden hier zusammenfassend erläutert:

- *Package-Angaben:* Für Packages gelten die Java-Konventionen (Trennung der Packages durch `.`). Package-Angaben dürfen bei manchen Konfigurations-Properties auch Wildcards enthalten (siehe Punkt unten). Ein Package wird im folgenden auch verallgemeinert als Segment bezeichnet.

- *Pfad-Angaben:* Pfade können absolut oder relativ angegeben werden. Als Pfad-Trennsymbol kann
  (unter Windows üblich) oder (unter Unix üblich) verwendet werden. Pfad-Angaben dürfen bei manchen Konfigurations-Properties auch Wildcards enthalten (siehe Punkt unten). Bei relativen Pfadangaben ist das aktuelle Verzeichnis immer das Verzeichnis, in dem die Generator-Anwendung abgelegt ist. Auch bei Pfad-Angaben sprechen wir bei einem Verzeichnis des Pfades von einem Segment. Eine Pfad-Angabe kann sich entweder auf ein Verzeichnis oder eine Datei beziehen.

- *Wildcards*: Siehe nächsten Abschnitt.

- *Mehrwertige und atomare Werte für Properties:* Eine Property kann entweder genau einen (atomaren) Wert verlangen, oder aber mehrere gleichartige Werte (1 bis N) erlauben. Bei mehrwertigen Properties müssen die Einzelwerte mit `;` abgetrennt

werden. Die Verwendung von ; am Ende der mehrwertigen Properties ist erlaubt. Wird bei atomaren Properties ; verwendet, wird dies als Bestandteil des Wertes interpretiert.

- *Leerer Property-Wert:* Ein leerer Wert hinter einer Property wird völlig gleichbedeutend mit dem Nichtvorhandensein der Property interpretiert.

Wir unterscheiden dann entsprechend die Typen Datei-Pfad, Verzeichnis-Pfad, gemischter Pfad (Datei oder Verzeichnis), Package, String, wobei String ein beliebiger Wert ohne Einschränkungen ist. Alle drei Typen können atomar oder mehrwertig sein. Wildcards sind nur für Packages und Pfade wie oben definiert verwendbar und werden in Strings ignoriert.

2.7.2 Wildcards

Wildcards dürfen in Pfad-Angaben und Package-Angaben genutzt werden, um Schreibarbeit zu sparen und mehrere Pfade bzw. Packages mit nur einer Angabe zu verwenden. Die Pfad-Angaben mit Wildcard-Ausdrücken werden dabei mit dem tatsächlichen Dateisystem der aktuellen Maschine abgeglichen, um passende Verzeichnisse, Dateien oder Packages zu finden. Bei Elementen, die dem Wildcardausdruck entsprechen, spricht man von einem *match*. Wildcardausdrcüke sind also eine Art und Weise, existierende Verzeichniss, Dateien und Packages ohne allzu viel Schreibarbeit bequem addressieren zu können.

Es gibt nur zwei Wildcards:

- * zur Angabe einer beliebigen Zeichenkette. Dies matched nur bis zum nächsten Verzeichnis-Trennsymbol bzw. bis zum nächsten Package-Trennsymbol.
- Der Wildcard ** matched hingegen alle Verzeichnisse oder Dateien bzw. Packages rekursiv bis zum nächsten danach angegebenen Verzeichnis bzw. Package.

Wildcards dürfen an beliebiger Stelle stehen und natürlich auch mehrfach in einer Pfad- oder Package-Angabe verwendet werden.

Wird * verwendet, dürfen im gleichen Segment vor oder hinter dem Wildcard noch beliebige (Pfad bzw. Package-valide) Zeichen verwendet werden. Dies kann also zu u.U. sehr vielen Dateien führen. Wird * am Ende des Pfades verwendet, dann werden alle Verzeichnisse bzw. Dateien im zuletzt angegebenen Unterverzeichnis (nicht-rekursiv) eingelesen. * kann auch mehrfach innerhalb desselben Segments verwendet werden, allerdings nicht direkt hintereinander (sonst wird ** erkannt). * kann im Falle von Dateipfaden natürlich auch im letzten Segment verwendet werden, um mehrere Dateinamen zu verwenden (z.B. C:
xyz
log*.txt). * bezieht sich immer nur auf das aktuelle Segment.

Wird ** verwendet, so dürfen im selben Segment keine anderen Zeichen verwendet werden, und es muss sowohl davor (falls nicht direkt zu Beginn des Pfades verwendet) als auch danach (falls nicht direkt am Ende des Pfades verwendet) ein Pfad- bzw. Package-Trennsymbol stehen. Wird ** am Ende des Pfades verwendet, heißt dies, dass ab dem Punkt alle Verzeichnisse bzw. Dateien bzw. Packages rekursiv eingelesen werden. Wird **

mehrfach direkt hintereinander verwendet, so ist dies äquivalent zur Verwendung nur eines `**`.

Werden nach `*` oder `**` noch Segmente angegeben, so matchen diese nur, wenn es im Segment von `*` Ordner gibt, die dem Ausdruck entsprechen und Unterverzeichniss oder -Dateien haben, die den nachfolgenden Segmenten entsprechen. Die nachfolgenden Segmente matchen nicht, wenn das übergeordnete Verzeichnis entsprechende Segmente hat. Beispiel:

- Pfade sind `C:/abc/xyz/hij` sowie `C:/abc/lmn`
- Der Wildcardausdruck `C:/abc/*/lmn` matched keinem der Pfade, denn unter abc gibt es nur das Unterverzeichnis `xyz`, das nur ein Unterverzeichnis `hij` hat.
- Der Wildcardausdruck `C:/abc/**/lmn` matched aus demselben Grund ebenso keinen der Pfade.
- Der Wildcardausdruck `C:/*/lmn` matched den Pfad `C:/abc/lmn`.
- Der Wildcardausdruck `C:/**/lmn` matched ebenso den Pfad `C:/abc/lmn`.
- Der Wildcardausdruck `C:/**/*` matched die Pfade `C:/abc/lmn` und `C:/abc/xyz`.

Für beide Wildcards gilt: Ihre Verwendung zu Beginn des Ausdrucks macht keinen Sinn, weder unter Unix noch unter Windows.

Zudem gilt: Ist ein Segment in einem Ausdruck mit oder ohne Wildcard eine Datei, und folgen danach noch weitere Segmente, dann ist dies natürlich ein Fehler.

2.7.3 Umgang mit Nicht-Existenz von Pfaden bzw. Packages

Bei atomaren Pfad- oder Packageangaben dürfen naturgemäß niemals Wildcards genutzt werden, denn diese könnten zu mehreren Matches führen und damit wäre die Property nicht mehr atomar. Hier gilt: Ist die Property optional, so wird die Nicht-Existenz des Pfades oder Packages im Dateisystem toleriert und nur mit einer Warnung vermerkt.

Bei mehrwertigen Pfad- oder Packageangaben, für die Wildcards nicht erlaubt sind, gilt: Mindestens einer der Pfade bzw. Packages muss existieren, sonst resultiert ein Fehler. Für alle anderen angegeben, jedoch nicht-existenten Pfade bzw. Packages wird eine Warnung vermerkt.

Bei mehrwertigen Pfad- oder Packageangaben, für die Wildcards erlaubt sind, gilt: Existiert nach dem Expandieren der Wildcards ein Teilpfad nicht, wird eine Warnung vermerkt. Auch sonst analog: Mindestens einer der Pfade bzw. Packages muss existieren, sonst resultiert ein Fehler. Für alle anderen angegeben, jedoch nicht-existenten Pfade bzw. Packages wird eine Warnung vermerkt.

2.7.4 Angaben über die Generierungsquelle

Um den Generator zu nutzen, sind Angaben über die Generierungsquelle notwendig. Die folgende Tabelle listet die mit Version 0.5 definierten Properties.

Property	**Eigenschaften**	**Erläuterung**
prop.src.dir	Verpflichtend, gemischter Pfad, Wildcards nicht erlaubt, atomar	Das Quellverzeichnis bzw. die Quell-JAR-Datei, in dem die Quell-Entitäten erwartet werden. Es handelt sich also um deren Classpath.
prop.src.pkg	Verpflichtend, Package, Wildcards erlaubt, mehrwertig	Die Quellpackages, aus dem die Quell-Entitäten gelesen werden. Nur im Wert dieser Property darf die spezielle Variable {`$sub.name`} verwendet werden. Sie darf mehrfach im Wert der Property benutzt werden, jedoch innerhalb desselben Pfades nur einmal. Sie hat die gleichen Auswirkungen und Verwendungsregeln wie die Verwendung des Wildcards `*`. Allerdings wird jedes unterschiedliche Package, dass matched, als Subsystem betrachtet, und alle Entitäten, die sich im gleichen Subsystem befinden, als diesem zugehörig. Die Entitäten in allen Package-Angaben, in denen {`$sub.name`} nicht vorkommt, werden einem intern generierten „Rest“-Subsystem zugeordnet.
prop.src.suff	Optional, String, atomar	Suffix der Entities. Wenn angegeben, erfüllt dieser zwei Funktionen: Erstens werden nur Entity-Klassen verwendet, die diesen Suffix im simple name haben. Zweitens wird der Suffix für die Bestimmung der vordefinierten Template-Variablen `ent.type.trim` verwendet: Der Suffix wird im Wert von `ent.type.trim` weggelassen.
prop.src.pref	Optional, String, atomar	Präfix der Entities. Wenn angegeben, erfüllt dieser zwei Funktionen: Erstens werden nur Entity-Klassen verwendet, die diesen Präfix im simple name haben. Zweitens wird der Präfix für die Bestimmung der vordefinierten Template-Variablen `ent.type.trim` verwendet: Der Präfix wird im Wert von `ent.type.trim` weggelassen.

Property	Eigenschaften	Erläuterung
prop.src.template.files	Verpflichtend, Datei-Pfad, Wildcards erlaubt, mehrwertig	Gibt alle für die Generierung zu verwendenden Template-Dateien an.
prop.src.dep	Optional, Gemischter Pfad, Wildcards erlaubt, mehrwertig	Ein oder mehrere Verzeichnisse oder auch JAR-Dateien, die auf dem Classpath benötigt werden, damit alle Abhängigkeiten der Quell-Entitäten erfüllt sind. Es werden die gleichen Verzeichnisse benötigt, die zum erfolgreichen Kompilieren aller Quell-Entitäten notwendig sind.

Tabelle 2.2: Konfigurations-Properties zur Angabe der Generierungsquelle

2.7.5 Angaben über das Generierungsziel

Außerdem sind Angaben über des Ziel der Generierung erforderlich:

Property	Eigenschaften	Erläuterung
prop.dst.dir	Verpflichtend, Verzeichnis-Pfad, Wildcards nicht erlaubt, atomar	Das Zielverzeichnis der Generierung. In dieses Verzeichnis werden alle generierten Klassen geschrieben.

Tabelle 2.3: Konfigurations-Properties zur Angabe des Generierungsziels

2.7.6 Definition benutzerdefinierte Template-Variablen

Benutzerdefinierte Template-Variablen werden zusammen mit den Konfigurations-Properties in der Konfigurationsdatei oder als Java-Runtime-Properties angegeben.

3 Design der Umsetzung

Hier wird das Grob-Design des Generators definiert und einige Implementierungsfragen geklärt.

3.1 Grundlegende Designentscheidungen

Unter grundlegenden Designentscheidungen sind Richtungsvorgaben für die Architektur der Anwendung zu verstehen, die nach Fertigstellung der ersten stable Version vermutlich nur mit sehr hohem Aufwand rückgängig gemacht werden können bzw. bei grundlegenden Änderungen aufwändige Migrationen und Regressionstests erforderlich machen. D.h. natürlich nicht, dass keine der Entscheidungen nicht doch in Folgeversionen revidiert werden kann.

Designentscheidung	**Begründung**
Als technische Plattform für die Umsetzung wird Java 8 SE verwendet.	TBD
Verwendung der Hibernate JARs für die JPA API.	TBD
Verwendung von Maven (3.2.1) für den Build-Prozess.	TBD
Verwendung von Eclipse (4.4) und M2E plugin als IDE.	TBD
Keine Anpassung bereits generierter Artefakte möglich	TBD

Tabelle 3.1: Grundlegende Designentscheidungen für Version 0.5

3.2 Generierungsprozess

3.2.1 Phasen

Der Generierungsprozess kann in folgende Phasen unterteilt werden:

1. *Einlesen der Konfiguration:* Auffinden, einlesen und Validieren der Konfigurations-Properties und der benutzerdefinierten Template-Variablen.

2. *Parsen der Templates:* Einlesen, Parsen und Validieren der verwendeten Templates. Die Informationen aus den Templates werden in eine interne Repräsentation überführt, welche das spätere Expandieren ermöglicht.

3. *Parsen der Entitäten:* Einlesen, Parsen und Validieren der Quell-Entitäten. Die Informationen aus den Quell-Entitäten werden extrahiert und in eine für das spätere Expandieren sinnvolle Repräsentation überführt.

4. *Expandieren der Templates:* Die Templates werden je Scope mit den entsprechenden Inhalten gefüllt.

5. *Finalisierung Generierungsprozess:* Der Generierungsprozess wird zum Abschluss gebracht: Ressourcen werden freigegeben, die finalen Java-Klassen werden geschrieben.

3.2.2 Potentielle Parallelisierung

Prinzipiell müssen die Phasen nicht vollständig sequentiell ablaufen. Es besteht folgendes Potential für Parallelisierung, um die Generierung der Ziel-Klassen bei vielen Templates und vielen Quell-Entitäten zu beschleunigen:

- Das Einlesen der Konfiguration ist Voraussetzung für alle anderen Schritte und wegen der überschaubaren Anzahl an Properties i.d.R. nicht langlaufend. Einzig das Bestimmen aller Dateien und Packages bei Verwendung von Wildcards erfordert Dateisystemzugriffe. Daher muss diese Phase vollständig und erfolgreich abgeschlossen sein, damit die Folge-Phasen starten können.

- Parsen der Templates und das Parsen der Entitäten sind völlig unabhängig voneinander und können so potentiell parallel durchgeführt werden. Dabei erfordern beide lesende Dateisystemzugriffe. Bei den Entitäten ist dies auf das initiale Laden der Entitäts-Klassen beschränkt. Dieses Laden *aller Entitäts-Klassen* kann ggf. bewusst vor allen anderen Parse-Schritten durchgeführt werden. Nachdem alle Entitäts-Klassen geladen sind, wird mit dem Parsen der ersten Entitätsklasse begonnen. Das Parsen der Entitäten selbst kann bereits in mehreren Threads erfolgen, wobei eine Entität selbst nur wenig Zeit für das Parsen benötigt. Mehrere Threads sollten hier nur dann erzeugt werden, wenn eine gewisse Minimalzahl an Entitäten erreicht ist, und dann sollte aus einer möglichst geringen Anzahl Threads je ein Thread einen Block von N Entitäten nacheinander abarbeiten. Diese Parallelisierung ist aber im Hinblick auf möglicherweise später hinzuzufügende Aktivitäten wegen JPA-Relationen vorsichtig zu designen. Parallel zum Parsen der Entitäten kann bereits das Lesen der Templates in einem weiteren Thread starten. Dieses Lesen sollte in nur einem Thread erfolgen. Das nachfolgende Aufbereiten der Templates (ohne I/O) sollte ebenso nur in einem Thread erfolgen, da nicht mit einer enormen Anzahl an Templates zu rechnen ist (in aller Regel 3 bis maximal 10 Templates).

- Je nach Scope eines Templates kann dessen Expandierung bereits dann erfolgen, wenn es fertig geparst ist, und bevor alle Entitäten geparst worden sind. Dies geht genau dann, wenn das Template selbst Scope `global` oder `subsystem` hat, und kein

darin verwendeter Repeat-Block einen Scope kleiner als `subsystem` hat.

- Auch das finale Schreiben der Ziel-Java-Klassen kann bereits während des Parsens der Entitäten mit den im letzten Punkt genannten Einschränkungen erfolgen.

3.3 Einlesen der Konfiguration

Das Einlesen der Konfiguration kann nochmals in folgende Unterschritte gegliedert werden:

- Bestimmen der Konfigurations-Quelle
- Parsen und validieren aller Konfigurations-Properties
- Parsen und validieren aller benutzerdefinierten Template-Variablen

Alle Schritte der Phase 1 werden von einer Klasse `ConfigPropertyPhaseHandler` umgesetzt. Diese hat darüber hinaus die folgenden Verantwortlichkeiten:

- Verwalten aller verfügbaren Konfigurations-Properties
- Verwalten der für die Konfigurations-Properties vorhandenen Konfigurations-Werte
- Fehlerbehandlung für Fehler, die bei den oben genannten Schritten auftreten

3.3.1 Bestimmen der Konfigurations-Quelle

TBD

3.3.2 Parsen und validieren aller Konfigurations-Properties

Eigenschaften von Konfigurations-Properties

Wir hatten weiter oben die folgenden Typen von Konfigurations-Properties identifiziert: Pfad-Angaben, Package-Angaben, sonstiges. Zudem wurde zwischen atomaren und mehrwertigen Properties unterschieden. Bei Pfad- und Package-Angaben sind noch Wildcards möglich. Zudem kann eine Konfigurations-Property optional oder verpflichtend sein. All diese Eigenschaften können in einer Klasse `ConfigPropertyDesc` zusammengefasst werden:

- name: String
- optional: boolean
- wildcardsAllowed: boolean
- isMultiValued: boolean
- defaultValue: String

- type: PATH|PACKAGE|OTHER

Je konkret definierter Konfigurations-Property (siehe Tabellen **??** und **??**) gibt es dann eine Instanz der Klasse `ConfigPropertyDesc`.

Zieltypen des Parse-Vorganges

Was sind die konkret zu verwendenden Datentypen in `ConfigPropertyDesc`, also die konvertierten Zieldatentypen? Es macht Sinn, für alle Properties mit isMultiValued=true eine `List<T>` zu verwenden, und für solche mit isMultiValued=false eben T. Wofür steht im Einzelnen T?

- Für Konfigurations-Properties, die weder den Typ PATH oder PACKAGE haben, kommt nur String (isMultiValued=false) und `List<String>` (isMultiValued=true) in Frage. Defacto findet also außer einem Splitten bei multivalued keine Konvertierung statt.
- Für Pfad-Angaben gibt es mehrere Alternativen: `java.io.File` (kann sowohl Dateien als auch Verzeichnisse repräsentieren), String oder java.io.Path (neu mit JavaSE 1.7). Wir gehen der Reihe nach auf die Typen ein:
  - String ist zu primitiv, da weitere Verarbeitungsschritte bereits mindestens ein File benötigen, z.B. um Verzeichnisse zu erzeugen, Existenz zu prüfen etc.
  - Path bietet erweiterte und bequeme Funktionen. Allein für die Weiterverarbeitung nach dem Abholen der Property-Werte ist Path nicht unbedingt notwendig, sondern das Zurückliefern eines Files ist ausreichend. Sollten später Path-Funktionen benötigt werden, kann einen Umwandlung in einen Path nach wie vor erfolgen. Die neuen nio-Funktionen auf Paths mit glob-Syntax (einer recht mächtigen Unix-Wildcard-Syntax für Pfadangaben) mit Java 1.7 klingen zuerst vielversprechend, nützen uns aber leider wenig:
    - `FileSystem.getPathMatcher()` wäre mit seiner glob-Syntax fast ideal geeignet, um Pfade mit Wildcards zu implementieren. Der zurückgelieferte PathMatcher als solcher ist aber wenig nützlich. Seine direkte Verwendung würde jedoch erfordern, alle Pfade unter dem letzten Pfad-Segment rekursiv zu iterieren, um dann mit PathMatcher.matches() zu prüfen, ob der Pfad vom Pattern abgedeckt wird. Sehr ineffizient. Eine andere Verwendung scheint es für diesen PathMatcher leider gar nicht zu geben (es gibt keine Methode, die einen PathMatcher akzeptieren würde).
    - Mit `Files.newDirectoryStream()` scheint es eine perfekte Lösung zu geben: Diese Funktion nimmt einen Path und einen glob. Dann kann man über alle Pfade iterieren, die unter dem Start-Pfad dem glob entsprechen. Leider liefert diese aber tätsächlich nur alle Pfade direkt unter dem Start-Pfad, z.B. alle Dateien mit der Endung .java. Rekursive globs wie **/*.java funktionieren nicht.
  - Bleibt noch java.io.File. Da die Funktionalität von Path nicht weiterhilft bzw. nicht benötigt wird, kann diese Klasse für die Repräsentation von Dateien und

Verzeichnissen genutzt werden.

- Auch für Package-Angaben gibt es mehrere Alternativen, die wir der Reihe nach vorstellen:
  - Die Klasse java.reflect.Package kommt nicht in Frage: Sie wird weder für das spätere Einlesen von Entitätsklassen benötigt (denn es gibt keine Methoden wie „getAllClassesInPackage“ o.ä.) noch für die Klärung, ob das Package existiert oder nicht.
  - Wieder kommt eine String-Repräsentation wie „org.de.package“ in Frage. Diese ist für die Weiterverarbeitung jedoch nicht sonderlich nützlich. Denn es müssen letztlich alle Java-Klassen im Package gefunden werden, was nur mit Dateisystemmitteln erreichbar ist. Dennoch wird die String-Repräsentation *auch* benötigt: Nämlich dann, wenn die Zielklasse geladen werden soll. Somit führen wir eine neue Klasse `PackageDesc` ein, welche den vollqualifizierten Dateipfad des Packages enthält sowie den Package-Namen.

Lösungsvariante 1: Parser- und Description-Hierarchie

Das Parsen der properties ist von den Dimensionen type, isMultiValued und wildcardsAllowed abhängig. Die in Rohform als Strings vorhandenen Konfigurationswerte müssen im Verlaufe des Parsens letztlich in Zieldatentypen umgewandelt werden, die für die weitere Verarbeitung am geeignetsten sind. Daher ist es sinnvoll, `ConfigPropertyPhaseHandler` bereits umgewandelte Datentypen liefern zu lassen. Die Umwandlung in Datentypen schließt bereits ein Parsen und Validieren mit ein. Eine mögliche Design-Lösung besteht in Folgendem:

- Statt eines type-Attributes in `ConfigPropertyDesc` kann ein generischer Typ-Parameter T verwendet werden
- Da das Parsen und Validieren stark vom type abhängig, macht es Sinn, dies in spezifischen Klassen zu kapseln
- Als Basisklasse kann man einen `AbstractConfigPropertyParser` definieren, der einige Gemeinsamkeiten sowie abstrakte Methoden für Teilschritte des Parsens anbietet. Diese Klasse wird je type von einer konkreten Parser-Klasse erweitert.
- Der Parser muss die `ConfigPropertyDesc` kennen, für die er parst (die Informationen daraus sind für den Parse- und Validierungsvorgang erforderlich).
- Die Abfrage von Konfigurations-Property-Werten soll idealerweise ohne Casts auskommen. Um dies zu erreichen, sind folgende Designanpassungen notwendig:
  - Man macht `ConfigPropertyDesc` zu einer abstrakten generischen Klasse `AbstractConfigProp`
    Je konkreter Kombination von type und isMultiValued definiert man eine von `AbstractConfigPropertyDesc` abgeleiteten Klasse, also z.B. eine für PATH und multivaled=true, eine andere für PATH und multivalued=false. In diesen abgeleiteten Klasse kann man dann auch spezifische Eigenschaften definieren, die nur für diesen Typ von Konfigurations-Property relevant sind.

  - Zudem muss dann jede `AbstractConfigPropertyDesc` eine Instanz des zugehörigen Parsers erzeugen können. Auch dieser ist generisch.
  - Dann kann man eine get-Methode in `ConfigPropertyPhaseHandler` definieren, welche dazu dient, zu einer gegebenen Instanz des Typs `AbstractConfigPropertyDesc` den jeweiligen (bereits geparsten, validierten und konvertierten) Wert aus der Konfiguration zurückzuliefern.
  - Diese Methode erzeugt den Parser, parst, validiert, loggt das Ergebnis und schmeißt ggf. eine Exception. Die geparsten Werte können in einen Cache geschrieben werden, sodass dieser Prozess nur beim ersten Abfragen (lazy) passiert. Diese Methode kann typsicher implementiert werden, wobei im Fall von Caching zumindest eine „unchecked“ conversion in Java notwendig ist, da der Cache üblicherweise in Form einer Map implementiert wird, die dann gezwungenermaßen mit Wildcards definiert sein muss, da sie unterschiedliche konkrete `AbstractConfigPropertyDesc` mit unterschiedlichsten konkreten Typ-Argumenten speichern muss.

Zusammengefasst sieht das Design in Form eines Klassendiagramms so aus:

TBD

Hier ist ersichtlich:

- Jede Parserklasse speichert sowohl den Rohwert der Property als String, als auch den geparsten Wert des generischen Typs T, als auch Validierungs-Informationen in Form einer Instanz von `PropertyValidationResult`. Dies kann später für das Logging der entsprechenden Validierungs-Fehler und Warnungen genutzt werdne.
- Die Methoden parse() und validate() sind in der abstrakten Basisklasse implementiert und führen einige Basisschritte des Parsings und der Validierung durch, sie delegieren dann an protected extended Methoden, die in der abgeleiteten Klasse implementiert werden müssen.

Lösungsvariante 2: Nur stateless Parser-Hierarchie

An Lösungsvorschlag 1 fällt die duplizierte Hierarchie auf, sowie, dass die Parser den Zustand (Rohwert und konvertierten Wert) halten. Somit ist ein Parser-Instanz je Property notwendig. Dies fällt Performance-technisch natürlich aufgrund der geringen Anzahl an Objekte nicht ins Gewicht. Das Ziel jedoch, Casts beim Lesen der konvertierten Zieldaten zu vermeiden, kann auch einfacher erreicht werden.

Lösungsvariante 2 besteht aus folgendem:

- Aus `AbstractConfigPropertyDesc` wird die generische Klasse `ConfigProperty`, die nun nicht mehr nur die Eigenschaften einer Property repräsentiert, sondern auch deren (Roh- und geparsten Ziel-) Wert enthät. Dieser kann dann direkt mit einem getter gelesen werden, der den generischen Typ als Rückgabetyp hat.
- Die Klasse ist nicht mehr abstrakt und hat keine abgeleiteten Klassen mehr. Die

Unterschiede zwischen den Property-Typen werden ausschließlich durch die Parser-Klassenhierarchie implementiert. Die Klasse muss dazu eine Parser-Instanz im Konstruktor übergeben bekommen.

- Zudem hat sie eine Methode parseAndValidate, die den gelesenen Rohwert erhält. Diese ruft nacheinander parse() und dann validate() auf dem Parser auf.
- Parserhierarchie und -Methoden entsprechen weitgehend denen in Lösungsvariante 1, nur dass die Parser nun stateless sind und daher alle Werte (inkl. die ConfigProperty selbst) als Parameter erhalten.

Hier das zugehörige Klassendiagramm:

TBD

Dieses Design reduziert die Klassenanzahl deutlich und ist dabei noch flexibler einsetzbar. Es bekommt daher den Vorrang.

Hinweise zur Implementierung des Parsens

Das Parsen der Konfigurations-Property-Werte wird durch Wildcards zu einer recht komplexen Aufgabe. Die folgende Tabelle fasst einige Hinweise zusammen, die für die Implementierung des Parsens relevant sind:

Typ	**Hinweise zum Parsen und Validieren**
String, atomar	Trivial
String, multivalued	Trivial: Nur split
Pfad, atomar	Trivial: Nur Prüfung auf Existenz
Pfad, multivalued, keine Wildcards	Trivial: Nur split und Prüfung auf Existenz
Pfad, multivalued, Wildcards	Split, effizientes Spazieren durchs Dateisystem und Prüfung auf Existenz
Package, atomar	Trivial: Nur Prüfung auf Existenz
Package, multivalued, keine Wildcards	Kann Implementierung für Pfad weitegehend wiederverwenden, siehe dort
Package, multivalued, Wildcards	Kann Implementierung für Pfad weitegehend wiederverwenden, siehe dort

Tabelle 3.2: Implementierungshinweise für Parsen und Validieren

3.3.3 Parsen und validieren aller benutzerdefinierten Template-Variablen

3.4 Parsen der Templates

3.5 Parsen der Entitäten

3.6 Expandieren der Templates

3.7 Logging

4 Test

5 Ausblick auf die Folgeversion

Für die Folgeversion 0.6 sind folgende Features geplant:

-