Programmierung von Systemen

Erik Neller — Dozent: Matthias Tichy

30. August 2020

ulm university universität ulm

Inhaltsverzeichnis

1 UML-Klassendiagramme

1.1 Klassen und Schnittstellen

Methoden und Attribute werden wie in Java in der Form `$TYP $NAME` angegeben, ist die Methode vom Typ `void` entfällt der Rückgabetyp.

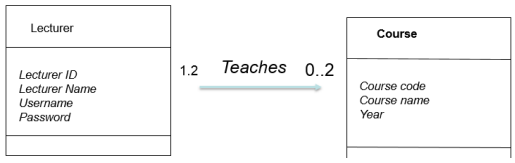
- `private` wird durch "-" symbolisiert
- `package` wird durch "~" symbolisiert
- `protected` wird durch "#" symbolisiert
- `public` wird durch "+" symbolisiert
- `static` wird unterstrichen

{`readonly`} ist ein UML Attribut das für Konstanten verwendet wird.

Der Block einer Klasse / eines Interfaces besteht aus drei Feldern:

1. Name mit Modifikatoren
2. Attribute (Variablen)
3. Methoden

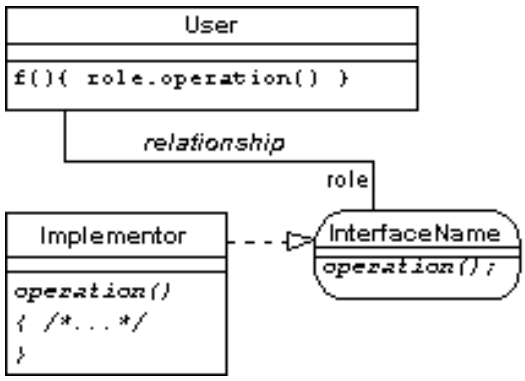
Modifikatoren für die Klasse können sein: «`interface`» oder {`abstract`} bzw. der Name in *kursiv*.
Die Multiplizität wird genutzt um Listen und Mengen von Objekten anzugeben, aber auch in Relationen zwischen Klassen:



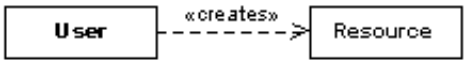
1. Typ
2. Größe begrenzt {1..x} oder unendlich [*]
3. Attribute wie {order} und {unique}

1.2 Relationen

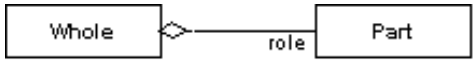
`extends` oder `implements` wird dargestellt durch einen nicht ausgefüllten Pfeil, wobei die Linie bei gleichen Typen (Interface erbt von Interface, Klasse von Klasse) durchgezogen ist, wenn eine Klasse ein Interface implementiert, gestrichelt.



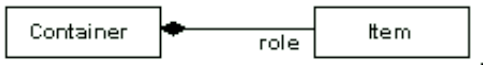
- Abhängigkeit (Dependency): User nutzt Ressource, aber die Ressource ist nicht Teil der User Klasse. Wird die Ressource modifiziert, muss auch der User modifiziert werden.



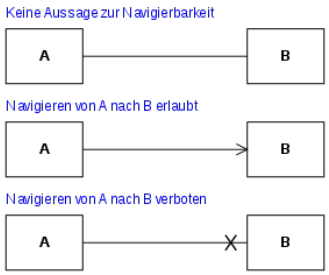
- Aggregation: "ist Teil von", wird symbolisiert durch einen Pfeil mit leerer Raute



- Komposition "besteht aus", wird symbolisiert durch einen Pfeil mit ausgefüllter Raute



- Assoziation: Die Klassen sind auf eine beliebige Weise verbunden, nutzen Methoden der anderen aber nicht auf die oben genannten Weisen



Kommentare werden mit einer gestrichelten Linie mit dem eigentlichen Objekt verbunden.

2 Version Control Systems

2.1 Grundlegende Funktionen

Version Control Systems (VCS) erlauben die Verwaltung von mehreren Teilen und Versionen eines Projekts und damit die Zusammenarbeit von mehreren Teilnehmern.

- Rechteverwaltung (z.B. Entwickler von Front-und Backend, Projektmanagement)
- Archivierung in verschiedenen Versionen (einfach anhand der gemachten Änderung, anstatt vollständige Backups zu machen)
- Speicherung von Metadaten: Historie von Änderungen mit Datum, Autor, etc.
- Backup zur Wiederherstellung lokal gelöschter Daten oder versehentlicher Änderungen
- Zentralisierung auf Server

2.2 Git

Das Git System besteht aus vier Teilen, von denen drei durch Git selbst implementiert werden. Jede Änderung durchläuft sie in dieser Reihenfolge:

1. Der eigentlichen Arbeitsplatz / *Workspace* in dem Änderungen an Dateien vorgenommen werden
2. Die *staging area*, in der *commits* aus einzelnen Änderungen an Dateien feingranular (bis zu einzelnen Zeilen) zusammengesetzt werden
3. Das *lokalen Repository*
4. Das *remote Repository* auf einem Server, beispielsweise GitHub oder GitLab

Auf den einzelnen Bereichen existieren verschiedene Befehle. Für die staging area:

- `git init` erstellt ein neues Git-Repository im aktuellen Verzeichnis
- `git add` um Dateien der staging area hinzuzufügen
- In einer `.gitignore` Datei können Regeln für Dateien angegeben werden, die generell nicht mit in die staging area aufgenommen werden sollen
- `git status` um aktuell geänderte und getrackte Dateien zu sehen
- `git rm --cached` um Dateien aus der staging area zu entfernen

Für das lokale Repository:

- `git commit -m $MESSAGE` um den Inhalt der staging area in das lokale Repository hinzuzufügen
- `git checkout $COMMIT-HASH` erlaubt das Wiederherstellen vorheriger Zustände von commits
- `git reset --soft HEAD~x` erlaubt das Rückgängigmachen von x commits
- `git log` zeigt den Verlauf von commits an
- `git remote add $NAME $ADDRESS` verknüpft das lokale Repository mit einem remote Repository

Und für das remote Repository:

- `git push $REPOSITORY $BRANCH` um den lokalen commit auf den Server zu legen
- `git pull` um das lokale Repository mit den Änderungen aus dem remote Repository zu aktualisieren
- `git clone $REPOSITORY` um das ganze Repository lokal zu speichern

Für verschiedene Themengebiete / Zuständigkeitsbereiche existiert das Konzept von Branches(Verzweigungen), die mit `git branch $NAME` erstellt werden können, um anschließend mit `git checkout $NAME` in den Branch zu wechseln, oder in Kurzform: `git checkout -b $NAME`. Mit `git merge $NAME` kann dann ein Branch in den jeweils Aktuellen integriert, oder mit `git branch -d $NAME` gelöscht werden. Ist keine automatische Vereinigung der Branches möglich, müssen die angezeigten Dateien manuell geändert und anschließend mit `git add $FILE` Alternativ kann der Branch in das Repository aufgenommen werden: `git push $REMOTE $BRANCH`.

3 Objektorientierung

3.1 Interfaces

Analog zur abstrakten Klasse erlaubt ein Interface keine Instanziierung. Es enthält keine tatsächliche Implementierung, sondern nur Methodenrümpfe und evtl. Konstanten und muss dementsprechend in jeder Klasse mit dem Schlüsselwort `implements` implementiert werden. Im Gegensatz zu abstrakten Klassen können mehrere Interfaces von einer Klasse implementiert werden.

3.2 Sichtbarkeit

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

3.3 Annotations

Verschiedene Standardannotationen, eigene definierbar. Durch Reflection zur Laufzeit auslesbar.

- `@Override` gibt an dass hier ein Element überschrieben werden soll
- `@Deprecated` gibt eine Warnung aus dass Element veraltet ist
- `@SuppressWarnings()` unterdrückt die angegebene Warnung
- `@Documented` nimmt nachfolgende Annotationen in die JavaDoc mit auf

JavaDoc:

- `@author`
- `@version`
- `@param` zur Beschreibung von Methodenparametern
- `@return` beschreibt den Rückgabewert
- `@exception`, `@throws` Beschreibt Fehlermeldungen, die diese Methode produzieren kann
- `@link` Verknüpfung zu anderem Symbol

3.4 Enumerations

Mit einem `enum` kann eine Menge von Konstanten definiert werden, entweder auf Klassenlevel oder innerhalb einer Klasse. Sie sind auch selbst eine Klasse, deren Attribute `public static final` definiert sind.
Methoden:

- `int ordinal()` gibt die Position in der Liste des Enums zurück
- `String name()` gibt den Namen der Konstanten zurück
- `int valueOf` gibt den zugeordneten Wert der Variablen zurück

4 Collections

4.1 Datenstrukturen

4.1.1 Array

Statische Größe, wird mit Länge und Datentyp gespeichert, zB int[5]. Objekt x erhalten: array[x]. Länge: array.length

4.1.2 ArrayList

Klasse die zur Speicherung Arrays verwendet, diese aber ersetzt wenn die Methoden `add()` oder `remove()` aufgerufen werden. Mit `toArray()` kann der Array erhalten werden, mit `size()` die Größe, da diese dynamisch ist. Objekt erhalten: arraylist.get(x).

4.1.3 Linked List

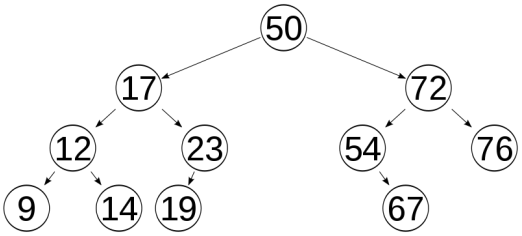
Jedes Listenelement enthält einen Zeiger auf das nächste Element, bei dem letzten Element ist der Zeiger `NULL`.

4.1.4 Double Linked List

Wie linked list, nur dass jedes Element zusätzlich einen Zeiger auf das vorherige Element enthält (beim ersten Element `NULL`).

4.1.5 Sorted Tree

Setzt die Sortierbarkeit der Objekte voraus: Objekte mit kleinerem Wert sind links des Vaterknotens, mit größerem rechts. Beim Einfügen / Entfernen reorganisieren: rot-schwarz Bäume, B Baum, B+Baum



4.1.6 HashTable

Objekte werden gehasht, also ein Wert aus ihnen berechnet, anhand dessen sie in in eine Tabelle eingeordnet werden.

4.1.7 Map

Speichert nicht einzelne Werte, sondern Tupel aus (Key, Value), nutzt dann zB eine Liste aus Tupeln oder eine Hashtabelle anhand der Keys.

4.2 Collections API

Stellt effiziente Datenstrukturen für viele Anwendunge bereit, sodass Datenstrukturen und Algorithmen nicht selbst implementiert werden müssen. Das Collection Interface selbst enthält Methoden wie add(), size(), contains(), remove()

- List ist eine geordnete Collection: add(e), add(index, e), indexOf(e), contains(e)
- Set ist eine Collection ohne Duplikate(inklusive `NULL`): add(), contains()

4.3 Queue

Warteschlange als Datenstruktur bei der Elemente am einen Ende angehängt und am anderen Ende gelesen werden. Dementsprechend existieren die folgenden Methoden, die eine Exception verursachen bei Fehlern:

- add(e) fügt ein Element hinten an der Queue an, falls max. Größe erreicht Exception
- remove() nimmt das vorderste Element der Queue und gibt es zurück, gleichzeitig wird es aus der Queue entfernt (Exception falls Queue leer)
- element() gibt das erste Element aus der Queue zurück aber löscht es nicht (ebenfalls Exception falls leer)

Alternativ existieren die Methoden mit speziellen Rückgabewerten anstelle von Exceptions:

- offer(e) gibt true/false zurück ob Element hinzugefügt wurde
- poll() gibt Element oder `NULL` zurück
- peek() gibt Element oder `NULL` zurück

4.4 Deque

Ist die Erweiterung einer Queue mit den gleichen Operationen an beiden Enden der Datenstruktur, analog zu einem Kartendeck (daher auch oft fälschlich Dequeue geschrieben: Double Ended queue)

4.4.1 Exception und special Element

	throw Exception		**special Element**	
	first	*last*	*first*	*last*
einfügen	addFirst()	addLast()	offerFirst()	offerLast()
löschen	removeFirst()	removeLast()	pollFirst()	pollLast()
lesen(behalten)	getFirst()	getLast()	peekFirst()	peekLast()

4.4.2 Blocking Deque

Die dritte Möglichkeit ist, den Thread zu blockieren wenn die Operation nicht möglich ist. Davon existieren zwei Varianten: mit maximaler Blockierungszeit (Timeout) und ohne. Dieses blocking deque implementiert sowohl die blocking Queue (welche queue und blocking implementiert) als auch das Deque.

	block	**timeout**
einfügen	putFirst()	offerFirst(e,time,unit)
löschen	takeFirst()	pollFirst(e,time,unit)
lesen(behalten)	-	-

4.5 Generics

Collections können *generisch* verwendet werden, indem bei der Initialisierung eine Klasse angegeben wird, deren Instanzen in der

Collection enthalten sein werden. Bei allen anderen Klassen / Datentypen wird der Compiler hier direkt eine Fehlermeldung liefern, was der Alternative, potenzielle Fehler während der Laufzeit, vorzuziehen ist.

4.6 Streams

Streams aus `java.util.stram.Stream<T>` stellen Ströme von Referenzen dar, die es erlauben, verkettete Operationen auf diesem Datenstrom nacheinander oder parallel auszuführen. Bestimmte Methoden erlauben die Verwendung von Prädikaten und klassischen mathematischen Funktionen. Wo vorher eine for-Schleife mit Zählvariable notwendig war um die Anzahl von bestimmten Elementen in einem Array zu erhalten, kann man damit schreiben: `anzahl = Arrays.stream(myArray).filter(x -> x.equals(SString")).count();`

5 Java IO

5.1 Streams

Grundlegende Operationen beim Lesen von Daten:

- read
- skip
- available (prüfen ob Daten verfügbar)
- close

und beim schreiben von Daten:

- write
- flush (prüfen ob richtig geschrieben)
- close

Verschiedene Arten von Daten die in Java gelesen werden: ByteArray, Object, andere Dateien
Ein Stream ist eine "Hülle" für eine Datei und kann wie der utilStream mehrere Dateien als kontinuierlichen Strom bearbeiten.

Für die verschiedene Zwecke existieren auch verschiedene Streams, die einander übergeben werden können:

- InputStream als generelles Interface / Klasse
- FileInputStream zum Lesen von Dateien
- ByteArrayInputStream zum Lesen von ByteArrays
- DataInputStream zum Lesen von Java Datentypen
- ObjectInputStream zum Lesen von Java Objekten (Problem: Versionierung)
- SequenceInputStream zum Verknüpfen mehrerer Inputstreams
- BufferedInputStream zum schnelleren Lesen über einen Pufferspeicher

5.2 Reader/Writer

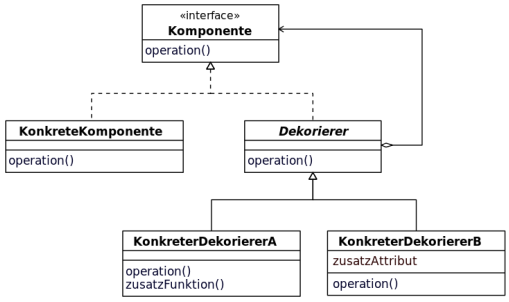
Liest statt Bytes lesbare Zeichen/Characters, also ist ein Reader/writer eine erweiterte Version der Streams der zusätzlich eine Übersetzung von Characters in Bytes macht.

- Reader
- FileReader
- StringReader analog zu ByteArrayInputStream
- BufferedReader
- InputStreamReader um von Stream zu Reader zu übersetzen

Über ein boolean flag kann außerdem bestimmt werden, ob an die existierende Datei beim Schreiben angehängt werden soll. Tatsächlich wahlfreien Zugriff erhält man mit RandomAccessFile(file,mode), das dann auch die Methoden seek() und getFilePointer() bereitstellt. Zur Erleichterung der Nutzung existiert außerdem das try-with-resources-Statement.

5.3 Decorator Pattern

Reader / Writer und Streams nutzen das Decorator Pattern um die Kombination verschiedener Funktionalitäten zu erleichtern. Anstatt BufferedDataFileInputStream zu implementieren, können einfach die verschiedenen Streams einander übergeben werden und so "verschachtelt" verwendet werden. Im Decorator Pattern werden die grundlegenden Methoden, die von der konkreten Komponente und den Decoratorn implementiert werden, durch ein Interface definiert. Jede Decorator Klasse kann dann unter Verwendung der konkreten Komponente zusätzliche Funktionen hinzufügen.



6 Dateien

6.1 java.io.file

In java.io.file sind sämtliche Java-Methoden zum Umgang mit Dateien zusammengefasst.

6.2 java.nio.file.*

Stellt erneuerte, sortierte Methoden zum Umgang mit Dateien bereit.

- FileSystem(s): Wurzelverzeichnisse / Arten von Dateisystemen
- Path(s): Pfade zu Dateien / Verzeichnissen
- Files: zur tatsächlichen Manipulation von Dateien

6.3 Scanner

Kann primitive Typen scannen: Boolean, int, Double, String. Mit den Methoden hasNext kann überprüft werden ob der Scanner eine Eingabe des entsprechenden Typs bereit hält, um diesen dann mit next() abzurufen.

6.4 RegEx

Reguläre Ausdrücke aus der Automaten - / Sprachentheorie können verwendet werden um Übereinstimmung mit einem Format zu prüfen.

6.4.1 Zeichenauswahl

Innerhalb von [] lässt sich eine Zeichenauswahl treffen.

- $[egh]$ wählt eines der Zeichen e,g oder h aus
- $[0-6]$ eine Zifer von 0 bis 6
- $[a-zA-Z0-9]$ lateinischer Buchstabe oder Ziffer
- $[\wedge a]$ negiert a, alles außer a

Vordefinierte Zeichenklassen:

- \d für digit, also Zahlsymbole
- \w word character inklusive Unterstrich $[a-zA-Z_0-9]$, eventuell auch nicht-lateinische Buchstaben
- \s WhiteSpace
- \n newline
- . (Punkt) als Wildcard für alle Zeichen außer Zeilenumbrüchen, im single-line mode auch diese

Vordefinierte Zeichenklassen können mit der großgeschriebenen Variante negiert werden: während \d für [0-9] steht, bedeutet \D $[\wedge 0-9]$. Die Zeichensätze, insbesondere die der Buchstaben, sind in der Regel von der Betriebssystem-locale abhängig.

6.4.2 Quantoren

Ausdruck		?	+	*	*?	{m,n}	{m}
Bedeutung	1	0,1	≥ 1	≥ 0	kleinste passende Zeichenkette	{min,max}	{m,m}

6.4.3 Spezielle Zeichen

- $\wedge$ Anfang der Zeile
- $ Ende der Zeile
- X|Z X oder Z
- XZ X gefolgt von Z
- () zum Gruppieren von Ausdrücken, kann später mit $x wieder aufgerufen werden, Zählung beginnt mit 1; capture group
- (?:) um eine Zeichenkette zu gruppieren, aber nicht zu speichern (kann später nicht aufgerufen werden); non-capturing group
- a(?!b) a, wenn nicht gefolgt von b (negative lookahead)
- \1 um auf Capture Groups zuzugreifen, die dann den tatsächlichen Inhalt referenzieren.

6.4.4 Modi

- (?i) um Groß- und Kleinschreibung zu beachten
- (?s) für single-line Mode (ganze Datei als eine Zeile interpretieren)
- (?m) für multi-line mode

6.4.5 Java

Einige Java Funktionen unterstützen RegEx nativ, zB split(), matches(), replaceFirst(), replaceAll() aber **nicht** replace(). Außerdem existieren java-util.regex.Pattern und java.util.regex.Matcher .

7 XML

Extensible Markup Language zur Darstellung hierarchisch strukturierter Daten innerhalb einer Textdatei. Beginnt in der Regel mit `<?xml version=''1.0'' encoding=''UTF-8''?>` Tags werden mit `<Tag>` *Inhalt* `</Tag>` angegeben und können Kind-Tags und Attribute haben. Kind-Tags werden innerhalb des Inhaltes angegeben, während Attribute im Tag selbst geschrieben werden, also
`<Tag attr="hello">` *Inhalt* `</Tag>`. Um die Gültigkeit von XML Dateien sicherzustellen, muss ein Muster vorgegeben werden, mit dem diese geprüft werden kann. Die Syntax ist durch XML vorgegeben, die Gültigkeit der Kombinationen und des hierarchischen Aufbaus kann / muss jedoch je nach Anwendung angepasst werden und braucht dementsprechend eine variable Überprüfung. Kommentare haben die Form `<!-- Kommentar -->`

7.1 DTD

7.1.1 Verknüpfen

Verwiesen wird auf die DTD in der XML Datei über `<!DOCTYPE` *rootNode* `SYSTEM|PUBLIC` *file.dtd*`>`. Mit SYSTEM wird eine lokale URI angegeben, mit PUBLIC eine vordefinierte, online bereitgestellte DTD verwendet.

7.1.2 Aufbau

- Element `<!ELEMENT Name Inhalt>`
- Attribut `<!ATTLIST referenzELement (Name Typ Wert)>` verschiedene Vorgaben für Attribute sind: #REQUIRED, #IMPLIED (=optional), #FIXED, und ein Standardwert mit "Wert"
- Entity `<!ENTITY name ''Zeichenkette''>` Abkürzung für einen String oder ein Dokument, das dann referenziert werden kann, also #PCDATA

Kindelemente werden nur mit Namen aufgezählt, nicht tatsächlich innerhalb der anderen Tags angegeben, da ein Tag auch mehrere Verwendungen haben könnte.
Für die Angabe von Kindelementen existieren reguläre Ausdrücke:

Außerdem können Auflistungen in Reihenfolge durch Kommata getrennt werden und Elemente mit () gruppiert werden.

7.1.3 Datentypen

In einer DTD existieren nur zwei grundlegende Datentypen:

- #PCDATA, bei dem die Eingabe wie der Rest der Datei geparst wird, also in Text zum Beispiel <FETT> </FETT> verwendet werden kann um verschiedene Formatierungen zu erhalten. Sollen in #PCDATA reservierte Symbole verwendet werden, müssen deren Kodierungen statt der eigentlichen Symbole eingetragen werden.
- #CDATA wird als reiner Text interpretiert

7.2 XSD

XML Schema Definition ist selbst in XML und verweist auf eine "root" XSD Datei, um wiederum die Korrektheit der XSD zu prüfen. XSD erweitert die Funktionalität der DTD um die Möglichkeit:

- die Anzahl der Instanzen eines Elements anzugeben
- das Aussehen der Zeichendaten innerhalb eines Elements zu spezifizieren
- die semantische Bedeutung eines Elements zu beschreiben.
- eigene Datentypen zu erstellen

Das Wurzelelement `<schema>` hat folgende Form:

```
<namespace:schema
xmlns:namespace=''XSDnamespace''
targetNamespace=''https://package.URI.com/file.xsd''
xmlns=''https://defaultnamespace.xsd'' >
```

1. *namespace*, das Präfix für den Namespace , das in der zweiten Zeile definiert wird.
2. wird der Namespace der XSD benannt und ein Namespace(äquivalent zu Package) angegeben, in dem sich die XSD befindet.

3. der targetNamespace, also der Namespace in dem sich die durch die XSD definierten Elemente befinden

4. der default Namespace, in dem sich alle Elemente ohne explizites Präfix befinden

7.2.1 Verknüpfen

7.2.2 Datentypen

XSD enthält einige grundlegende, bereits bekannte Datentypen:
Grundlegende Typen:

- xs:string
- xs:integer
- xs:date
- xs:decimal
- xs:boolean
- xs:time

Auch bei ihnen können den Daten Eigenschaften wie `default` und `fixed` gegeben werden, jeweils mit einem korrespondierenden Wert. Außerdem können für jedes Element `minOccurs` und `maxOccurs` definiert werden.
`<element name=''Hello'' minOccurs=''0'' default=''World''>`
Darüber hinaus gibt es zwei definierbare Datentypen, `<simpleType>` und `<complexType>`, die entweder innerhahlb eines Elements einmalig definiert werden können, oder universell nur innerhalb von `<schema>`.

simpleType Entsteht aus den Basis-Datentypen durch Einschränkung mit `<restriction base=''string''>` mit einem Kindelement der Art `<maxLength value=''10''/>` (maxExclusive, maxInclusive). Kann keine anderen Elemente enthalten. Auch eine `enumeration` zählt als Einschränkung und damit als `simpleType`, da sie nur vorgegebene Werte annehmen kann. Dafür wird innerhalb der restriction tags eine liste von `<enumeration value=''blau''>` erstellt, die dann die validen Zustände darstellen.

complexType Entsteht aus mehreren simpleTypes oder einer Restriktion eines complexType. Am häufigsten ist eine `<sequence>`, oder `<all>`(sequence mit minOccurs=1) von mehreren anderen Typen.