HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Scala als Generatorensprache

Julian Müller

Konstanz, [Datum]

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang [Software-Engineering/Technische Informatik/Wirtschaftsinformatik]

Thema: **Scala als Generatorensprache**

Bachelorkandidat: Julian Müller, Mondrauteweg, 78467Konstanz

1. Prüfer: Prof. Doc. Marko Boger
2. Prüfer: Titel, Markus Gerhart

Ausgabedatum: [Datum]
Abgabedatum: [Datum]

Zusammenfassung (Abstract)

Thema:	Scala als Generatorensprache
Bachelorkandidat:	Julian Müller
Firma:	HTWG Konstanz
Betreuer:	Prof. Doc. Marko Boger Titel, Markus Gerhart
Abgabedatum:	[Datum]
Schlagworte:	[Platz, für, spezifische, Schlagworte, zur, Ausarbeitung]

[Text der Zusammenfassung etwa 150 Worte. Es soll der Lösungsweg beschrieben sein.]

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Julian Müller, geboren am 07.07.1991in Donaueschingen*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Scala als Generatorensprache

bei der HTWG Konstanz unter Anleitung von Prof. Doc. Marko Boger selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, [Datum]

(Unterschrift)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seitdem Rechner mit Maschinencode programmiert werden, steigt das Abstraktionslevel der Programmiersprachen kontinuierlich. Siehe Abbildung 1.1. Wer die Assemblersprachen kennt und fürchtet, weiß Programmiersprachen wie Fortran, Pascal oder C zu schätzen, da sie Kontrollstrukturen einführen, welche leicht verstanden und eingesetzt werden können. Wer stets wiederkehrende Muster programmiert, bedient sich der Objektorientierten Sprachen, wie zum Beispiel Java oder C++. Indem die Logik hinter Programmiersprachen den Paradigmen angeglichen werden, die die Menschen intuitiv(oder am einfachsten) beherrschen, wird programmieren erheblich effizienter, sowohl im Lernprozess, als auch in der Umsetzung. Der prozentuale Anteil an wiederverwendbarem Code in einem Projekt steigt equivalent zu dem Abstraktionslevel der verwendeten Programmierparadigmen. Code, der wiederverwendet werden kann spart Zeit und reduziert die Chance neue Fehler in das Projekt einzuarbeiten. Die Objektorientierung wurde eingeführt um wiederkehrende Problemstellungen einheitlich behandeln zu können (z.B. graphische Elemente), somit die Komplexität des Codes zu verringern und dessen Wartbarkeit zu erleichtern. Die Programmiersprache *Scala* ist ebenfalls objektorientiert, erfüllt jedoch ebenso die Kriterien einer funktionalen Programmiersprache und bietet zudem einige Features, die sie besonders dafür qualifizieren sogenannte *Domain Specific Languages* zu erstellen. Dies begünstigt die Unterstützung des nächsten Schrittes in der Evolution der Programmierparadigmen. Nicht zuletzt der Markt bekräftigt den Wunsch nach eben dieser Effizienz und fordert diese nächste Abstraktionsstufe. Entwickler auf der ganzen Welt benutzen Bibliotheken und Frameworks, um bereits entwickelte Lösungen wiederverwenden zu können und um bekannte Lösungsansätze für bekannte Problemstellungen zu verwenden. Der hierbei entwickelte Code richtet sich also nach einem bereits vorgelegten Plan. Jener Quellcode, der manuell ergänzt werden muss, lässt sich nun teilweise automatisiert ergänzen. Dieses Ziel verfolgt das **M**odel **D**riven **S**oftware **D**evelopment (**MDSD**). Schematischer Code kann in domänenspezifischen Modellen verarbeitet werden. Nachdem Modelle konzipiert sind, ist es möglich automatisiert auf domänenspezifische Problemstellungen einzugehen. Im Falle der Industrie bedeutet dies Massenproduktion, ermöglicht von Robotern. Im Falle der Softwareetwicklung, bedeutet dies Code, der Code ge-

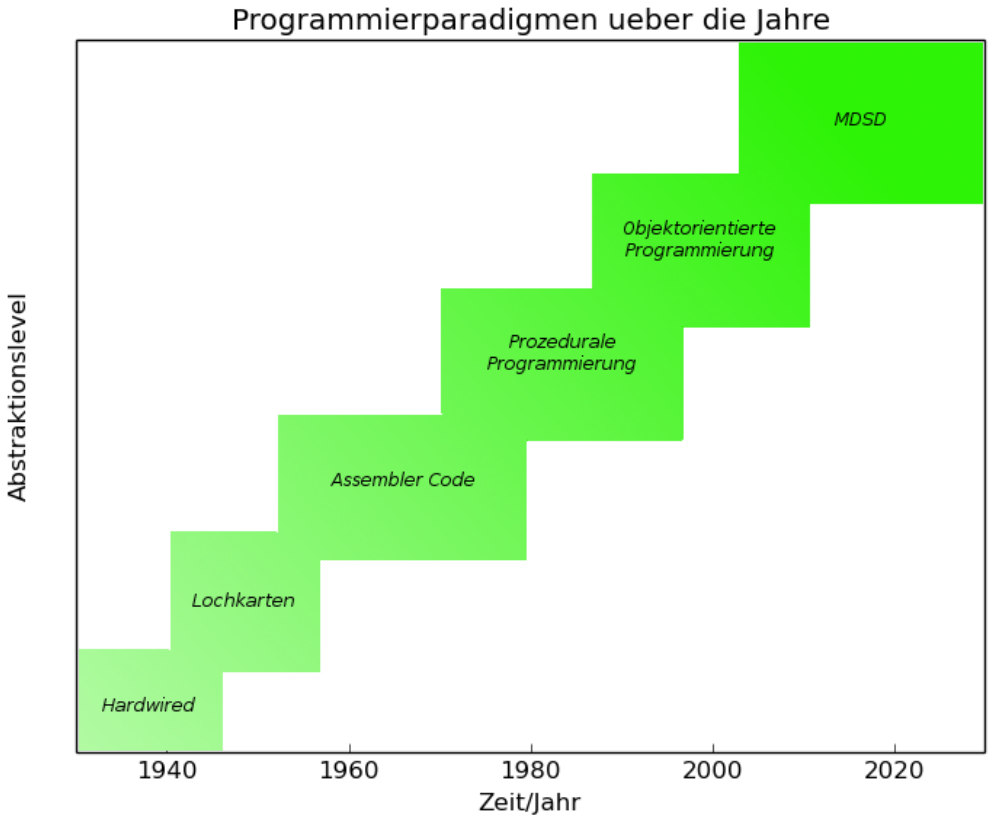

Abbildung 1.1: Programmierparadigmen im Verlauf der Zeit

neriert. Die Idee ist, variable Teile des Codes über ein Modell zu beschreiben. Das Programm ist selbstständig in der Lage, das Modell in den gewünschten Code umzuwandeln. Das besagte Modell kann hierbei drastisch vereinfacht, oder auf eine bestimmte Benutzerzielgruppe(Domäne) angepasst sein. Es lässt sich also problemlos einrichten einem Banker über seine gewohnten Fachbegriffe eine **D**omain **S**pecific **L**anguage (**DSL**)zu bieten, welche syntaktisch auch noch einer seiner Tabellen oder beispielsweise Buchungssätzen ähnelt und somit für ihn unkompliziert und sofort erlernbar ist. Für die Umwandlung eines Modells in konkreten Code (Generat) sind sogenannte **Generatoren** notwendig. Siehe auch Bettin [1]

1.1 Was Ist MoDiGen

Domänenspezifische Modellierung erfreut sich wachsender Beliebtheit in der Softwareentwicklung. Projekte, die für ein domänenspezifisches Problem graphische Modellierungstools benötigen, können entweder ein sehr spezifisches Tool benutzen, was nur dann möglich ist, wenn das Problem exakt durch das Tool beschrieben werden kann. Ansonsten muss ein generischer Editor heange-

zogen werden, doch diese sind dann in der Regel dermaßen komplex, dass es nicht mehr möglich ist, sein Problem schnell und unkompliziert darstellen zu können. Das MoDiGen (**Mo**del **Di**agram **Gen**erator) Projekt will es ermöglichen durch kurze und schnell umsetzbare textuelle DSLs graphische Editoren erzeugen zu lassen, deren Entwicklung sonst überaus Zeitaufwendig sein kann. Hierfür wurde einerseits ein hoch abstrahiertes Metamodell erstellt, welches komplexe Klassenhierarchien darstellen kann, außerdem wurden einige textuelle DSLs erstellt, über welche die Modellklassen beschrieben werden können (Diagram, Shape, Connection, Style). So soll dem Anwender (Domain Expert) die Möglichkeit geboten werden, eigene graphische Elemente zu erzeugen. Dabei kann es sich auch um Verbindungselemente handeln, denn diese werden im Metamodell von MoDiGen getrennt und ebenbürtig zu den eigentichen Elementen gehandhabt und werden nicht wie bei etwa Ecore nur als Attribut eines Modells gehalten (vgl. Gerhart and Boger [6]).

1.2 Problemstellung

Hinsichtlich der Flexibilität der Modelle und der Skalierbarkeit bieten bestehende Lösungen wie Xtext und Ecore derzeit keine Ideale Lösung. Die relativ junge Programmiersprache **Scala** ist, wie der Name schon andeuten soll, mit einem besonderen Fokus auf die Skalierbarkeit entwickelt worden. Wie bereits in 1.1 erläutert wurde, stützt sich das Projekt auf ein neues selbst entwickeltes Metamodell, welches hinsichtlich der Skalierbarkeit wesentlich bessere Ergebnisse erzielt. So wie das Metamodell sollen nun auch die Parser und Generatoren in Scala umgesetzt werden. Es soll untersucht werden, inwiefern Scala (als allzweckstaugliche Programmiersprache) sich dafür eignet, die Rolle bestehender spezialisierter Technologien zu übernehmen. Wo kann Scala seine Stärken spielen lassen und wo ist es den bisherigen Lösungen unterlegen? Dafür sollen u.a. MoDiGen's Modellklassen Style, Shape und Diagram in Scala umgesetzt werden und die entsprechenden Parser und Generatoren erstellt werden. Konkreter: Es müssen zunächst die geforderten Objektstrukturen geschaffen werden um Style, Shape und Diagram abbilden zu können. Hierbei muss darauf geachtet werden, dass ein Style ein weiteres Style assoziieren kann um eine Vererbung zu ermöglichen. Shapes können derzeit aufgrund der Komplexität der Modellklasse nicht vererbt werden. Sowohl die Shapevererbung als auch Mehrfachvererbung im allgemeinen ist mit bestehenden Technologien nicht möglich und soll daher auf Umsetzbarkeit mit Scala überprüft werden. Desweiteren Sind die Beziehungen der Modelle untereinander genau geklärt - siehe Abbildung 1.2. So kann jedes Modell auf ein Style verweisen. Diagrams können außerdem auf Shapes verweisen. Shapes können Styles referenzieren. Außerdem muss ein Parser erstellt werden, der in der Lage ist Spray's **DSL** zu interpretieren und entsprechend der Modelle zu instanziieren. Zuletzt kommen Generatorenklassen, die schließlich entsprechende Generate aus den erzeugten Objekten erstellen können. Hierbei liegt die größte Herausforderung dabei, einen Parsingmechanismus zu erstellen, der mit rekursiven DSL-Definitionen umgehen kann und der unabhängig davon um welche Modellklasse es sich handelt, in der Lage ist Vererbungshierarchien zu

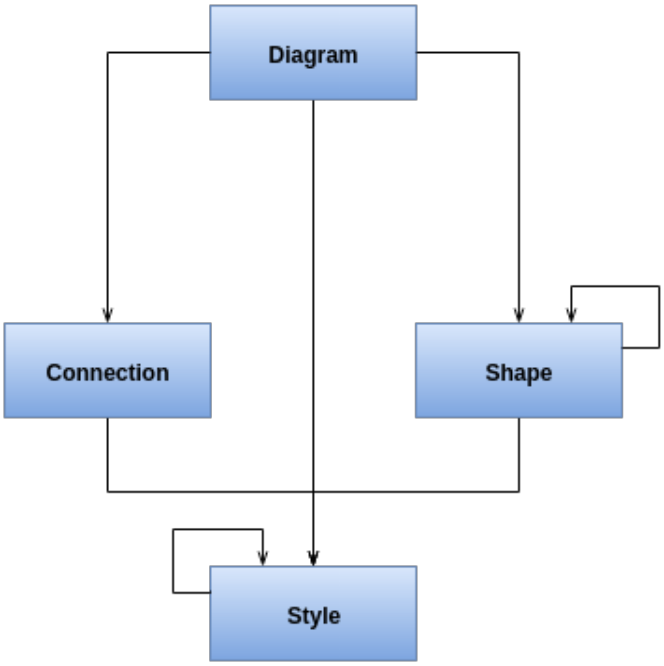

Abbildung 1.2: Klassendiagramm Diagram, Shape, Style (Sehr stark vereinfacht)

erkennen und umzusetzen. Da der Author Scala erst für dieses Projekt erlernt, kann davon ausgegangen werden, dass alle verwendeten Techniken Anfängerfreundlich sind.

Kapitel 2

Grundlagen: Scala/Xtext

2.1 Scala

Im folgenden soll zum Verständnis zunächst die Programmiersprache Scala vorgestellt werden. Die Entwicklung von Scala (Scalable Language) begann bereits 2001 an der École polytechnique fédérale de Lausanne in der Schweiz von einem Team um Professor Martin Odersky. Designziel war eine elegante, typsichere Programmierung, vollkommene Kompatibilität mit der JVM und eine Zusammenführung der Objektorientierung und der funktionalen Programmierung (vgl. [2]). Scala ist hierbei objektorientierter als z.B. Java, da in Scala *alles* ein Objekt ist. Odersky selber bezeichnet Scala als postfunktionale Sprache, da immernoch diskutiert wird, ob Scala als funktionale Programmiersprache betitelt werden darf. Fest steht jedoch, dass man in Scala funktional programmieren kann - man muss es allerdings nicht. Funktionen können Argumente oder Ergebnisse anderer Funktionen sein und sind somit *Higher Order Functions*. Trotz der Typsicherheit bietet Scala das *Feeling* einer dynamisch typisierten Sprache, da Typen beim Überetzen inferiert werden können. Der Compiler erkennt also anhand des initialisierten Wertes, um was für einen Typ es sich handelt, so kann fast immer auf manuelle Typkennzeichnung verzichtet werden.

```
val name = "Julian"
val alter = 24
```

sind vollkommen korrekte Ausdrücke und werden ebenso kompiliert wie:

```
val name:String = "Frederik"
val alter:Int = 33
```

Scala ist einfacher zu erlernen, als beispielsweise Java, da Anweisungen auch alleinstehend als Skript oder direkt in einem interaktiven Scalainterpreter ausgeführt und ausprobiert werden können. Komplexe Problemstellungen können stark abstrahiert in wenigen Zeilen Code beschrieben werden. Über entsprechende Methoden, kann man sogar eigene Operatoren und Kontrollstrukturen erstellen *„Damit ist Scala prädestiniert zur Erstellung von Domain Specific Languages* [...]*“*. Braun [2, p. 2]

2.2 Xtext

Nachdem nun Scala erörtert wurde, soll im folgenden die zu ersetzende Technologie *Xtext* erklärt werden. *„Xtext ist ein Framework zum Erstellen von Programmiersprachen und **D**omain-**S**pecific **L**anguages. Mit Xtext wird deine Programmiersprache durch eine mächtige Gramatiksprache definiert“* ∼ Xtext Homepage [8]. Xtext ist ein Framework mit dem u.a. Domänenspezifische Sprachen entwickelt werden können und ergänzt das *Eclipse-Modeling-Framework*[3]. Dafür wird eine komplexe Grammatiksprache zur Vergügung gestellt, über die Entitäten deklariert und assoziiert werden können. Anhand der Grammatik erzeugt Xtext Parser, Generatoren und sogar ein Klassenmodell für die beschriebenen Klassen (vgl. goodbyexml [7]). Um die beschriebene *DSL* ausführbar zu machen wird außerdem ein Texteditor generiert, welcher entsprechendes Syntax Highlighting automatisch programmiert. Der große Vorteil von Xtext ist, dass die Definition der beschriebenen Sprache gleichzeitig die Objektstruktur der zu generierenden Modelle beschreibt. Wird die Objektstruktur der Modelle verändert, ändert sich die DSL ebenfalls und so bleibt die Abbildung von DSL auf Modell immer konsistent.

Kapitel 3

Ansatz

Nachdem die Problemstellung geklärt ist und die konkurrierenden Technologien erläutert wurden, sollen nun die zugrundeliegenden Gedanken zur Problemlösung behandelt werden.

3.1 Objektstruktur

Zunächst würden die entsprechenden Modundellklassen aus der Xtext Grammatik in Scala Quellcode übernommen werden müssen. Style-, Shape-, Connection- und Diagramklassen mit entsprechenden Feldern würden zunächst als sogenannte *case classes* erstellt werden. Dies würde den Vorteil haben im Falle einer später notwendigen verifizierung der Instanzen vorimplementierte hash- equals- und toString Methoden zur Verfügung zu haben. Da Style am flexibelsten ist (nur eingehende Abhängigkeiten) würde Style.scala zuerst erstellt werden und entsprechend der Assoziationsreihenfolge dann Shape.scala und Diagram.scala. 1.2.

3.1.1 ClassHierarchy

Da es für die Instanzen mancher Modellklassen möglich sein soll, Objekte der selben Klasse zu erweitern, war die Idee des Authors, eine Collection namens *ClassHierarchy* zu erstellen, welche in der Lage sein sollte, ähnlich eines Baumes Vererbungshierarchien in Form von Knoten (*Nodes*) darzustellen 3.1. So würden die Modellklassen, für die die Vererbung möglich sein soll, über diesen Baum in Relation gesetzt werden können. Im Falle einer gänzlich neuen Modellklasse (z.B. ein neuer Style) würde die Instanz als neue Basisklasse eingefügt werden, ansonsten über eine Methode, die beispielsweise *inheritsFrom* heißen könnte. *InheritsFrom* müsste entweder von der ClassHierarchy oder den Nodes selber zur Verfügung gestellt werden und würde über seine Argumente vermittelt bekommen, welche Knoten erweitert werden. *InheritsFrom* würde dann eigenständig die *parent*- und *children* Listen eines Knotens um die neue Instanz der Modellklasse ergänzen.

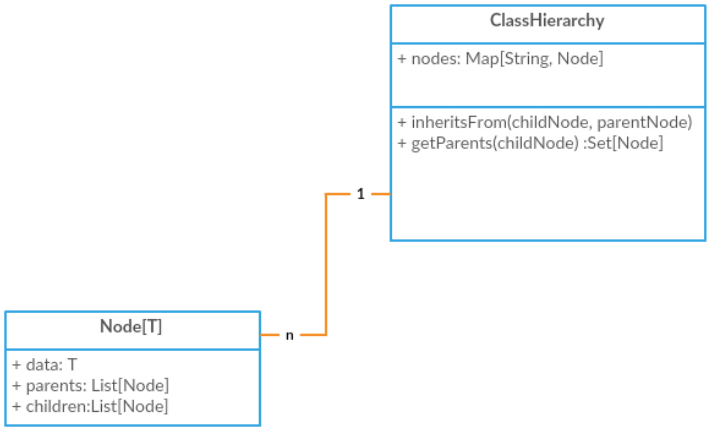

Abbildung 3.1: Klassenmodell von ClassHierarchy

Wie in 3.1 zu sehen ist würden die Nodes über Listen verfügen, welche auf Elternknoten und auf Kindknoten verweisen. So könnte man bequem von jedem Style erfahren, welche Styles er erweitert und welche Styles ihn erweitern. Außerdem würde ClassHierarchy eine Map enthalten, welche Namen(strings) auf beliebige Klassen abbilden kann. Über dieses mapping wäre ein schneller Zugriff auf die gewünschte Klasse möglich, ohne mühsam durch eine Baumstruktur der Nodes navigieren zu müssen. Um dieses Prinzip auf alle möglichen Klassen anwenden zu können sollte ClassHierarchy generisch sein. Vorausgehend waren die Anforderungen für Vererbung nur auf Style bezogen, da die Vererbungsprinzipien aber auch bei Shape hilfreich sein könnten, wäre so gewährleistet, dass man sich diese Tür nicht unnötig verschließt. Benötigte Enumerationen und andere nicht komplexe Hilfsobjekte der Klassen würden in dessen *Companionobject* definiert werden. Über entsprechende Apply-Methoden sollte der Aufruf der ClassHierarchy so einfach wie möglich gestaltet werden. Um Aufrufe wie

```
classHierarchy.nodes.get("styleName").data
```

zu vermeiden, würde eine apply-Methode benutzt werden um den Aufruf auf:

```
classHierarchy("styleName")
```

zu verkürzen. Da der Aufruf in diesem Fall sehr eindeutig ist, kann ruhigen Gewissens auf die apply-Methode zurückgegriffen werden. Esser [4, p. 76 Fazit]

3.2 Parser

Neben den Gedanken zur Vererbung der Modellklassen, ist es auch wichtig zu klären, wie die Modelle überhaupt aus der Stringform in Objektform umzuwandeln sind. Entsprechende Mechanismen werden allgemein als Parser bezeichnet. Dieser Parser ist verantwortlich dafür die in Stringform enthaltenen Styles, Shapes, Connections und Diagrams als diese zu identifizieren und in Objekte umzuwandeln. Die Struktur eines Styles (ähnlich auch bei den anderen Modellen) ist in Listing 3.1 beispielhaft dargestellt.

```
style BpmnDefaultStyle {

        description = "The default style of the petrinet
           diagram type."

        transparency = 0.95

        background-color = black

        line-color = black

        line-width = 1

        font-color = black

        font-name = "Tahoma"

        font-size = 6

        font-bold = yes

        font-italic = yeshttp://www.macwrench.de/wiki/
           Kurztipp_-_Quellcodelistings_in_LaTeX

        gradient-orientation = horizontal
        und
}
```

Listing 3.1: Beispielhafte Style Definition über die entsprechende Style DSL

Shapes und Diagrams fangen entsprechend mit

```
shape/diagram <shapeName> {
```

an. Die Xtext Grammatik sollte außerdem um die Vererbungskomponente erweitert werden können, z.B.:

```
shape SomeShape extends StandardShape {
```

Style erlaubt zwar bereits erweitert zu werden, doch was die Vererbung von Shapes und Mehrfachvererbung generell angeht besteht noch Bedarf, da Xtext nicht in der Lage ist, diese Features umzusetzen.

3.2.1 Reguläre Ausdrücke

Da Stringevaluierung, wie vorangehend erörtert, nicht ergebnisführend war, soll im Folgenden ein weiterer Ansatz vorgestellt werden, die *Regulären Ausdrücke*. Über diese *regulären Audrücke* könnte der Style gefiltert werden um so an an der Syntax der DSL vorbei zu kommen und an die eigentlichen Attribute zu gelangen. Anhand des ersten Wortes soll der Parser erkennen, um welchen Modelltyp (Style, Shape, Diagram) es sich handelt. Weiter würde ein vordefiniertes

Schlenter code hereüsselwört *extends* eine zu erweiternde Instanz kennzeichnen. Im Gegensatz zu diesen hartkodierten Schlüsselwörtern würden die restlichen Attribute wie

```
line-width = 1
```

einheitlich geparsed werden als

```
attributeName = attributeValue
```

und erst in einer tieferen Parserklasse oder dem entsprechenden Konstruktor aufgelöst werden. Auf diese Art und Weise würden wenige Regeln ausreichen um komplexe Ausdrücke abbilden zu können. So geparste Attribute bleiben zunächst in der Stringform. Ziel dieses Vorgehens ist es zunächst nur Attributsname und Attributswert zu ermitteln, sicherzustellen, dass Syntaxregeln eingehalten wurden und die Weitergabe eines einheitlichen Datentyps an tieferliegende Parserklassen. Um die Strings zu parsen, werden *reguläre Ausdrücke* verwendet. Die wenig komplexen Styles sollten so einfach zu parsen sein. Shapes stellen den schwierigsten Teil beim Parsing dar, da geometrische Figuren(*geometricModel*), wie Ellipsen oder Polygone ineinander geschachtelt werden können.

3.2.2 Factory Klassen

Es wird davon ausgegangen, dass Modellklassen durch Umwandlung aus einem bereitgestellten Stringinput erzeugt werden. Dieser Stringinput enthält alle notwendigen Informationen um eine einsatzfähige Instanz einer Modellklasse erzeugen zu können. Es ist wünschenswert die Erzeugung der Instanzen so konsistent wie möglich zu halten. Dies bedeutet möglichst wenige und klar definierte Wege eine Klasse zu erzeugen. Dies kann effektiv über eine sogenante *Fabrik* (vgl. englisch *Factory*) Klasse erreicht werden. Zur Erzeugung der Exemplare der Modellklassen wird so wenn möglich nur eine einzige Methode definiert, die schlussendlich auf den Konstruktor der Modellklasse verweist. Dies schließt allerdings noch nicht aus, dass Modellklassen auch anders erzeugt werden können. Scala stellt einen von Haus aus eingebauten Mechanismus zur Verfügung, die *Companion Objects*. Scala hat sich von den in Java typischen statischen Variablen (Klassenvariablen) getrennt. Um das Prinzip der Objektorientierung konsequenter durchzusetzen stellt es hieirfür die Definition von Singleton Objekte zur Verfügung. Über das Schlüsselwort *object* kann anstatt einer Klassen-, eine Objektdefinition erfolgen. Jegliche Felder eines solchen Objekts sind (in Java manier ausgedrückt) Instanzvariablen, jedoch importierbar. Allein die Möglichkeit auf diese Art und Weise Singleton Objekte zu erstellen deckt bereits eine Funktionalität der Fabrik Klassen ab. Wird ein solches Objekt in der selben Quelldatei wie eine Klassendefinition erstellt und haben Klasse und Objekt außerdem den gleichen Namen handelt es sich um *Companion Object* und *Companion Class*. Auf den ersten Blick sind *Companion Objects* nur der Ansatz Klassenvariablen bereitzustellen und trotzdem dem objektorientierten Paradigma treu zu bleiben. Ein entscheidender Vorteil ist jedoch, dass so definierte Objekte auch auf die privaten Member der *Companion Class* zugriff haben. Auf unseren Fall bezogen bietet sich also die Möglichkeit den Konstruk-

tor der Modellklasse mit dem Schlüsselwort *private* zu kennzeichnen. Folglich kann nurnoch das *Companion Object* darauf zugreifen und wir können exklusiven Zugriff über eine Fabrik Methode des Companion Objekts anbieten.

3.2.3 Parserlogik

Da in den bisherigen Gedanken geklärt worden ist, wie Strings eingelesen werden, die Vererbung unter den Modellklassen dargestellt werden kann und wie Modellklassen konkret erzeugt werden, kann nun anhand des Sequenzdiagramms 3.2 beschrieben werden wie ein Parsing Vorgang prinzipiell abzulaufen hat. Erhält die Applikation einen String um beispielsweise eine Shape einzulesen,

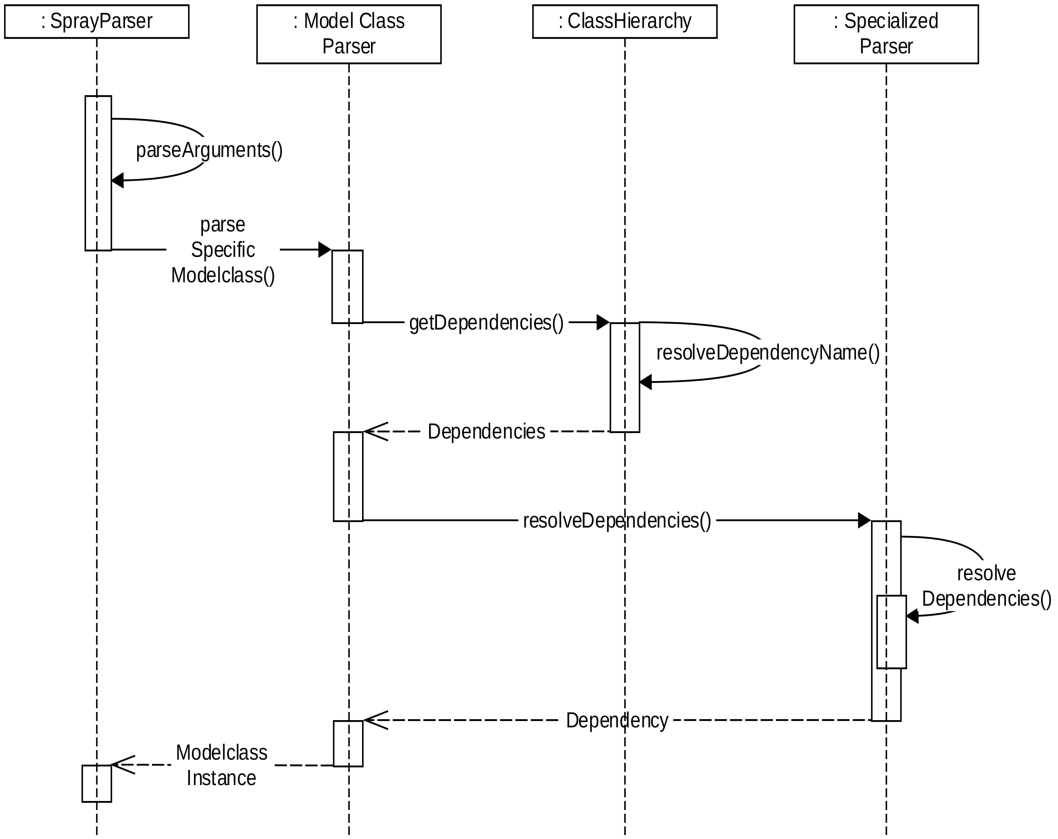

Abbildung 3.2: Sequenzdiagramm für den Ablauf eines Parsing Vorgangs

werden zunächst sämtliche Attribute aus dem String gefiltert. Hierzu zählen auch Name und erweiterte Shapereferenzen. Dies wird über eine Methode namens *parseArguments*(vgl. 3.2) erfolgen. Egal um welche Modellklasse es sich handelt, werden alle ausgelesenen Attribute in Stringform weiterverarbeitet. So wird die Verantwortung der Auflösung auf die einzelnen Parser der Modellklassen ausgelagert und diesen gleichzeitig ein einheitliches Medium geliefert. Auf diese Weise können ähnliche oder gleiche Attribute in unterschiedlichen Parsern auf die selbe Weise aufgelöst werden. Wie bereits angedeutet geht es von hier

an mit einem auf die Modellklasse spezialisierten Parser weiter *parseSpecificModelClass*(vgl. 3.2). Der spezifische Modellklassenparser muss nun zunächst prüfen, ob weitere Instanzen, des Zieltyps erweitert werden sollen. Hierfür gibt er die in den Attributen enthaltene Liste an Eltern (in Form von Strings) an die *ClassHierarchy* weiter, die den Überblick über die Vererbungshierarchie hat und die Namen über ein internes Mapping auf bestehende Referenzen auflösen kann (*resolveDependencyName*, vgl. 3.2). Sind alle Elternteile aufgelöst, kann der Modellklassen spezifische Parser nun anhand einer kompletten Liste an Attributen beginnen die in Stringform enthaltenen Informationen der Abhängigkeiten aufzulösen. Falls es sich bei den in Stringform enthaltenen Informationen nicht um primitive Datentypen, sondern um komplexe evtl. ineinander geschachtelte Elemente handelt, werden diese wiederum an ihren speziell definierten Parser weitergegeben (*resolveDependencies*, vgl. 3.2). Da die Elemente wie gesagt evtl ineinander verschachtelt sind, können viele weitere rekursive Aufrufe auf spezialisierte Parsereinheiten erfolgen, bis schließlich alle Argumente aufgelöst sind und eine fertige Instanz an den SprayParser zurück geliefert werden kann. Die Parserlogik teilt sich also auf viele kleine spezialiserte Parser auf. Dabei gibt es einerseits den SprayParser, welcher die Schnittstelle zu allen vorhandenen Parsern darstellt, die vier spezialisierten Style-, Shape-, Connection- und Diagramparser, welche die Modellklassen erzeugen und einige weitere kleinere Parser, welche dafür verantwortlich sind eigens erstellte Datentypen parsen und erzeugen zu können, wie zum Beispiel *Anchor*s.

3.3 Auflösung von Vererbungshierarchien

Ist eine Technik gefunden, mit der sich die DSL in die Objektstruktur der Modellklassen umwandeln lässt, muss direkt die Problematik der Vererbung betrachtet werden, da die hierfür benötigten Techniken schon beim Parsen beachtet werden müssen. Es soll ein Vererbungsmechanismus implementiert werden. Entgegen der geläufigen Definition der Vererbung handelt es sich hierbei nicht um das Erweitern von Klassen, wie man es aus diversen Programmiersprachen kennt. Die Anforderung definiert, dass einzelne Instanzen der Modellklassen in der Lage sein sollen Objekte der selben Klasse erweitern zu können. Genauer bedeutet dies, dass eine Instanz **nicht** in der Lage sein soll dynamisch komplett neue Attribute zu definieren, sondern **bekannte** Attribute von Elterninstanzen zu übernehmen, beziehungsweise zu überschreiben. Erweitern soll in diesem Kontext also zwei Situationen beschreiben:

- Elternteil verfügt über Attribute, die in der erweiternden Klasse nicht vorhanden sind ->Wert wird geerbt
- Elternteil verfügt über Attribute, die in der erweiternden Klasse vorhanden sind ->ignoriere/überschreibe den Wert der erweiterten Klasse

Das folgende Beispiel, soll verdeutlichen was gemeint ist: Es wird eine vereinfachte Styleklasse definiert

```
class Style(val name:String, val transparency:Double, val
    line-width:Int)
```

Nun wird eine Instanz (S1) der Modellklasse über die DSL definiert, welche sowohl einen Namen, als auch eine Angabe zur *transparency* hat:

```
style S1 {
    transparency = 0.5
}
```

Eine weitere Instanz (S2) wird definiert, die ihrerseits über einen Namen und eine Angabe zur *line-width* verfügt. Außerdem soll S2 die Instanz S1 erweitern:

```
style S2 extends S1{
    line-width = 10
}
```

Nach diesen Definitionen sollten Objekte folgender Struktur entstanden sein (aus Gründen der Lesbarkeit aufgeführt in der Java Script Object Notation):

```
{
 "name": "S1",
 "transparency": 0.5,
 "line-width":"notDefined"
}

{
 "name": "S2",
 "transparency": 0.5,
 "line-width": 10
}
```

Wie man sieht verfügt S2 über eine *transparency* angabe, die von S1 übernommen wurde. Wenn an dieser Stelle jedoch Überlegungen zur Implementierung unternommen werden, wird man mit folgenden Fragen konfrontiert:

- Wie weiß ein Objekt, dass es das Attribut des Elternteils übernehmen oder überschreiben soll?
- Übernimmt S2 tatsächlich alle Attribute des Elternteils, oder hält es nur eine Referenz auf S1?

Die meisten der Attribute, die in der Definition einer Modellklasse berücksichtigt werden können, sind optional. Wie man bei der Definition von S1 sieht, wurde keine Angabe zur *line-width* gemacht. Da die Definition jedoch auf den Konstruktor der Style Klasse abgebildet wird, muss irgendwo eine konkrete Angabe zu dem *line-width* Attribut erfolgen. Man könnte sich überlegen, jedem Feld der Modellklassen einen Defaultwert zuzuweisen. Das würde bedeuten, dass die *line-width* beispielsweise standardmäßig auf 10 gesetzt wird, wenn in

der Definition keine Angabe darüber gemacht wird. Hat aber folglich jede Styleinstanz zu jedem möglichen Attribut eine valide Angabe, fragt sich, woher später bekannt ist, ob auf Attribute einer Elterninstanz oder auf die eigenen zugegriffen wird. Immerhin gibt es zu jedem Feld der Instanz gültige Werte.

3.3.1 Kennzeichnung nicht definierter Felder in Modellklassen

Da Integerwerte ausschließlich gültige Werte haben wäre es sinnlos 0 als *nicht gesetzt* zu verwenden und 1 - MAX_INT als *gesetzte Werte*. Am Beispiel der *linewidth* Angabe in S1 wird also klar, dass es bekannt sein muss, ob die Definition den Wert beschreibt, oder nicht. Deshalb werden alle Felder der Modellklassen (Style, Shape, Diagram), die ein konkretes Attribut der Modellklasse darstellen, als sogenannte *Options* definiert. *Options* sind dem Paradigma der funktionalen Programmierung zu verdanken(z.B. in Haskell ->Maybe). Sie lösen ein Problem, welches in Java meist mit einer Exception gelöst wird. Eine Funktion gibt nicht immer für jedes Argument ein gültiges Ergebnis zurück, Esser [4, p. 102]. Im Falle der optional setzbaren Attribute der Modellklassen bieten Options daher die passende Lösung. Man kann sie entweder auf *None* initialisieren oder auf ein *Some*[*T*] . Representativ steht also in diesem Fall *None* für: nicht gesetzter Wert (schau in den Elternklassen nach entsprechenden Werten) und *Some*[*T*] für: gesetzter Wert, ignoriere/überschreibe den Wert der Elternklasse.

3.3.2 Vererbungsmechanismus über Referenz auf Elterninstanzen

Als nächstes ist es derzeit auch noch unklar an welcher Stelle dieser Gedanke Anwendung findet. Es besteht einerseits die Möglichkeit erst beim unmittelbaren Zugriff auf die einzelnen Felder zu prüfen, ob sie definiert wurden, andererseits könnten jegliche Elterninstanzen schon bei der Erzeugung einer neuen Instanz nach fehlenden Werten durchsucht werden. Genauer ist also die Frage, ob Modellklassen eine zusätzliche Referenz auf Elternteile erhalten und nicht definierte Werte erst bei Bedarf in den Attributen der Elternliste gesucht werden, oder ob Modellklassen schon bei der Instanziierung über sämtliche Attribute der Eltern bescheid wissen, diese auf sich übertragen und anschließend nicht mehr auf ihre Eltern angewiesen sind. Beide Möglichkeiten bieten Vor- und Nachteile. Referenzen erlauben es, die Extraktionslogik der Attribute auf eine beliebige Stelle im Code auszulagern, je nach dem wann der Aufruf eben erfolgen soll. Außerdem erscheinen sie zunächst Speicherschonender. Die Überlegung wäre ja, dass nicht definierte Werte auch keinen zusätzlichen Speicher benötigen, somit kein zusätzlicher Speicher allokiert werden muss. Dies ist allerdings ein Trugschluss. Die Felder einer Styleinstanz werden immer initialisiert. Im Falle, dass der Wert in der Definition nicht angegeben wurde, eben mit besagtem *None* Wert. Im Endeffekt ist also leiglich eine weitere Referenz als Attribut hinzugekommen, was die Speicherauslastung also (wenn auch unmerklich) erhöht.

3.3.3 Vererbungsmechanismus durch direkte Weitergabe der Attribute bei Instanziierung

Werden die Attribute der Elternklasse jedoch direkt bei der Objekterzeugung an die neue Instanz weitergegeben, hat dies zwei besondere Vorteile. Je nach Größe der Vererbungshierarchie ist es wesentlich performanter die Werte im aktuell benutzten Element zu suchen, als über eine Referenz in den vielen Elternknoten zu suchen. Außerdem beseitigt diese Vorgehensweise die Rekursivität der Problemstellung. Dazu ein stark vereinfachtes Beispiel:
Werden mehrere Modellklassen (hier Style) definiert, welche den jeweiligen Vorgänger erweitern und wird davon ausgegangen, dass die Attribute der zu erweiternden Instanzen direkt bei der Erzeugung übernommen werden,

```
style S1 {}
style S2 extends S1 {}
style S3 extends S2 {}
style S4 extends S3 {}
```

kann davon ausgegangen werden, dass S2 jedes Element von S1 erbt, S3 jedes Element von S2 erbt und somit auch implizit jedes Element von S1 usw. Diese Eigenschaft lässt sich auf beliebig viele erweiternde Instanzen anwenden, wodurch in dem Beispiel S4 garantiert jede Eigenschaft von S1, S2 und S3 enthält, oder überschreibt. Da die Rekursivität der Problemstellung so entfällt und den Code (ausschlaggebender Faktor ist die Größe des durch Vererbung erstellten Baumes) performanter gestaltet, wird versucht werden die Vererbungshierarchie auf diese Weise aufzulösen.

3.4 Generatoren

Ausgehend davon, dass nun Strings eingelesen und in entsprechende Instanzen der Modellklassen umgewandelt werden können, ist der nächste Schritt über Generatoren den nun eigentlich gewünschten Code (Generat) erzeugen zu lassen. Hierfür gibt es entsprechend der Modellklassen einige Generatorenklassen wie z.B. *StyleGenerator*, welcher die Style Modellklassen abbildet oder auch den *ShapeGenerator*, der die Shape Modellklassen abbildet. Die Generatorenklassen bilden die entstandenen Objektinstanzen auf *javascript* code ab. Um der Vererbung entsprechend immer die richtigen (also evtl die überschriebenen) Werte zu bekommen, müssen stets erst die Eltern-Knoten besucht werden, um nachzusehen, ob ein Elternteil evtl einen Wert ausfüllt.

Kapitel 4

Umsetzung

Basierend auf den Gedankengängen, die im Ansatz erläutert wurden 3, wird in diesem Kapitel die Umsetzung beschrieben. Hierbei werden zuvor getroffene Entscheidungen oftmals revidiert und neue Entschlüsse getroffen.

4.1 Objektstruktur

4.1.1 Modellklassen

Da bevor jeder anderer Logik zunächst die zu verwendenden Klassen definiert werden müssen, sollen Diese auch zuerst vorgestellt werden. Die Modellklassen Style, Shape, Connection und Diagram sind entgegen des Ansatzes, diese als *case classes* umzusetzen, als normale Klassen implementiert worden. Dies hängt damit zusammen, dass die Parserlogik auf verschiedene Einzelparser aufgeteilt wurde. Entgegen der Objektstruktur, die durch die DSL in Xtext beschrieben wird, mussten einige Beziehungen geändert werden (siehe Abbildungen 4.1, 4.2, 4.3). Die Xtext Grammatik definiert zwar, welche Elemente nur Definitionsregeln sind und was in die Objektstruktur aufgenommen wird, jedoch liegt es letztlich beim Entwickler zu entscheiden, was in die endgültige Objektstruktur übernommen wird und was nicht.

4.1.2 Besonderheiten bei Style

Bei der Umsetzung der einzelnen Modellklassen aus der Xtext Grmmatik in die Objektstruktur in Scala, kam es regelmäßig zu Situationen, in denen die Objektstruktur nicht 1:1 auf Scala übernommen werden konnten, da die verschiedenen Sprachen auch verschiedene Features bedienen, oder eben nicht (z.B. Xtext ->Schlüsselwort dispatch). Außerdem galt es bestehende komplexe Strukturen möglicherweise vereinfacht umzusetzen. Beispielsweise enthält die Definition der Styleklasse in Xtext eine sogenanntes Stylelayout, welches die eigentlichen Attribute der Styleklasse auflistet (siehe Listing 4.1).

```
Style:
    {Style}
```

```
    [...]
    layout=StyleLayout
    [...]
;

StyleLayout:
    {StyleLayout}
    (
        ("transparency" "=" transparency=DOUBLE)? &
        [...]
    )
;
```

Listing 4.1: stark vereinfachter Auszug aus der Xtxt Grammatik, die Styles beschreibt

In diesem Fall wurden alle Attribute eines Styles direkt in der Styleklasse aufgelistet. Dies vereinfacht einerseits die Objektstruktur, außerdem erleichtert es den Zugriff auf die Attribute, da die Navigation durch die referenzierte Instanz eines Layouts entfällt. Des weiteren definiert die Xtext Grammatik nicht eindeutig um welche Abstraktionsform es sich bei manchen Definitionen handelt. So wird beispielsweise definiert:

```
ColorOrGradient: Color | Transparent | GradientRef
```

Also kann an Stelle eines *ColorOrGradient* sowohl eine *Color*-, *Transparent*- oder *GradientRef*definition stehen. Für das Einbetten in eine Objektstruktur sind diese Regeln grausam, da Color, Transparent und GradientRef keine gemeinsamen Eigenschaften haben. Sie lassen sich zwar unter einer gemeinsamen Abstraktion zusammenfassen, dies bereitet jedoch später im Generator Probleme. *Highlighting*attribute sind beispielsweise vom Typ ColorOrGradient. Später im Generator werden für verschiedene Untertypen von ColorOrGradient einheitliche getterMethoden definiert (dies funktioniert in XText über eine sogenannte dispatch kennzeichnung), jedoch geht Gradient dabei leer aus. Wird also ein Gradient definiert, muss der Entwickler blind vertrauen, dass es nicht an der falschen Stelle eingesetzt wurde. Gibt man einem highlighting Attribut ein Gradient, was laut Abstraktion möglich ist läuft der Generator über Code, der versucht eine Methode aufzurufen, die nicht definiert ist.

```
if(s.layout.highlighting.selected != null){
                        [...].layout.highlighting.
                            selected.createColorValue;
                            }'''
                }
```

In diesem Fall ist die Methode createColorValue für ein Gradient **nicht** beschrieben. Entwickler müssten also jedes mal wenn die entsprechende Methode aufgerufen wird, zunächst prüfen um was für eine konkrete Unterklasse von ColorOrGradient es sich handelt. Für die Vereinheitlichung der Klassen wurden traits benutzt. Ausdrücke wie folgender

```
ColorOrGradient: Color | Transparent | GradientRef;
ColorWithTransparency: Color | Transparent;
```

zeigen, dass Xtext eine Art der Mehrfachvererbung zulässt. Color kann sowohl als ColorOrGradient, als auch ColorWithTransparency benutzt werden. Traits haben die Möglichkeit in eine Klasse *hineingemischt* zu werden und sollen in diesem Sinne in der Regel weitere Eigenschaften hinzufügen. Hierüber wird das Problem der vermeintlichen Mehrfachvererbung gelöst, da beliebig viele Traits in die Klasse hineingemischt werden können. Einige Elemente wie zum Beispiel Farbkonstanten ("light-orange" ->LIGHT_ORANGE, "blue" ->BLUE), wurden nicht wie in Xtext als Enumeration verwirklicht, sondern als *case objects*.

```
case object WHITE      extends ColorConstant {def
   getRGBValue = """#ffffff""" }
case object LIGHT_GRAY extends ColorConstant {def
   getRGBValue = """#d3d3d3""" }
case object GRAY       extends ColorConstant {def
   getRGBValue = """#808080""" }
case object DARK_GRAY  extends ColorConstant {def
   getRGBValue = """#a9a9a9""" }
...
```

Da Farbkonstanten (*ColorConstant*) so von Superklassen erben können, können einheitliche Methoden für sie definiert werden. Beispielsweise ist die Methode createOpacityValue in einer Superklasse einheitlich beschrieben worden.

```
trait ColorOrGradient{
   def getRGBValue:String
   def createOpacityValue:String = "1.0"
}
```

Da ColorOrGradient ein Trait ist, erlaubt es, die Methoden vorzuimplementieren. So ist automatisch jede Farbkonstante bedient und es muss nicht erneut wie in Xtext auf eine Weise wie *dispatch* getrickst werden.
In Abbildung 4.1 ist eine vereinfachte Version der entstandenen Objektstruktur zu sehen, die die wichtigsten Abhängigkeiten der Style Modellklasse darstellt.

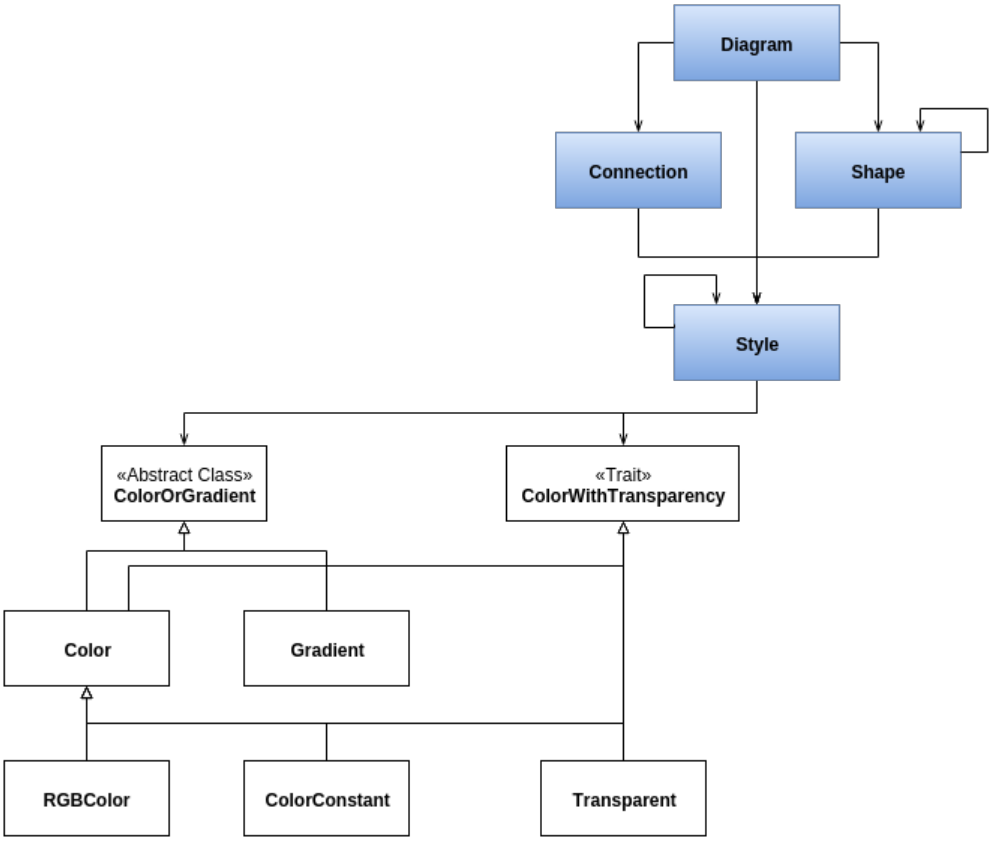

Abbildung 4.1: Vereinfachte Objektstruktur. Im Fokus Style und seine wichtigsten Dependencies.

4.1.3 Besonderheiten bei Shape

Für Shapes muss zunächst ein Namensproblem aufgelöst werden. Mit einer Shape wird der Wrapper definiert, der sowohl Metainformationen zu der Shape enthält und außerdem **eigentliche Shapes** in diesem Fall geometrische Figuren (*geometricModel*) enthält. Ist also die Rede von *geometricModel(s)*, sind die tatsächlichen Implementierungen beispielsweise einer Ellipse gemeint. Ist die Rede von einer Shape, ist die Wrapperklasse gemeint. Da die verschiedenen geometrischen Figuren einheitlich behandelt und verwaltet werden können müssen, bildet *GeometricModel* ihre gemeinsame Abstraktion **??**. Die konkreten geometrischen Figuren haben zwar untereinander hin und wieder Gemeinsamkeiten, doch es lassen sich tatsächlich keine allgemeinen Ähnlichkeiten feststellen. Beispielsweise haben einige zwar Angaben zur Größe, jedoch nicht alle. Aus diesem Grund definiert die abstrakte Klasse *GeometricModel* lediglich eine Gemeinsamkeit, die für die Distribution von vererbten Informationen wichtig ist: Die Referenz auf ein eventuell existierendes Elternteil. Ein Rechteck, kann ein Polygon beinhalten, welches wiederum ein weiteres Rechteck beinhalten kann.

```
ellipse style StandardStyle {
    rectangle {
        ...
    }
}
```

In diesem Falle muss garantiert werden können, dass auch das verschachtelte Rechteck über die Style-Attribute der Ellipse verfügt. Die Layouts der einzelnen geometrischen Formen wurden als Traits realisiert. Der große Vorteil der Traits als *rich-Interfaces* liegt darin, dass mehrere Eigenschaften(Traits) geerbt werden können. So kann jede geometrische Figur von der abstrakten Klasse *GeometricModel* erben und gleichzeitig die Eigenschaft eines spezifischen Layouts besitzen. Die Layouts selber haben wiederum nur eine Gemeinsamkeit, nämlich die Möglichkeit einen Style zu referenzieren. Diese Eigenschaft wurde in in dem Trait *Layout* bestimmt, von dem **jedes** weitere Layout erbt **??**. Da die geometrischen Figuren nun keine Referenz mehr auf Layoutobjekte haben, sondern entsprechende Felder selber erben, ist auch hierbei der Navigationsaufwand deutlich verringert. Wenn ein *roundedRectangle* zuvor auf seine Positionskoordinaten zugreifen wollte, gelang dies auf folgende Art und Weise:

```
val x_coord = rr.layout.commom.x
```

Durch die Vereinfachung der Objektstruktur können diese Informationen direkt abgerufen werden, sind aber logisch weiterhin von der erbenden Klasse getrennt und können einheitlich gewartet werden. In Abbildung 4.2 ist eine vereinfachte Version der entstandenen Objektstruktur zu sehen, die die wichtigsten Abhängigkeiten der Shape und Connection Modellklassen darstellt.

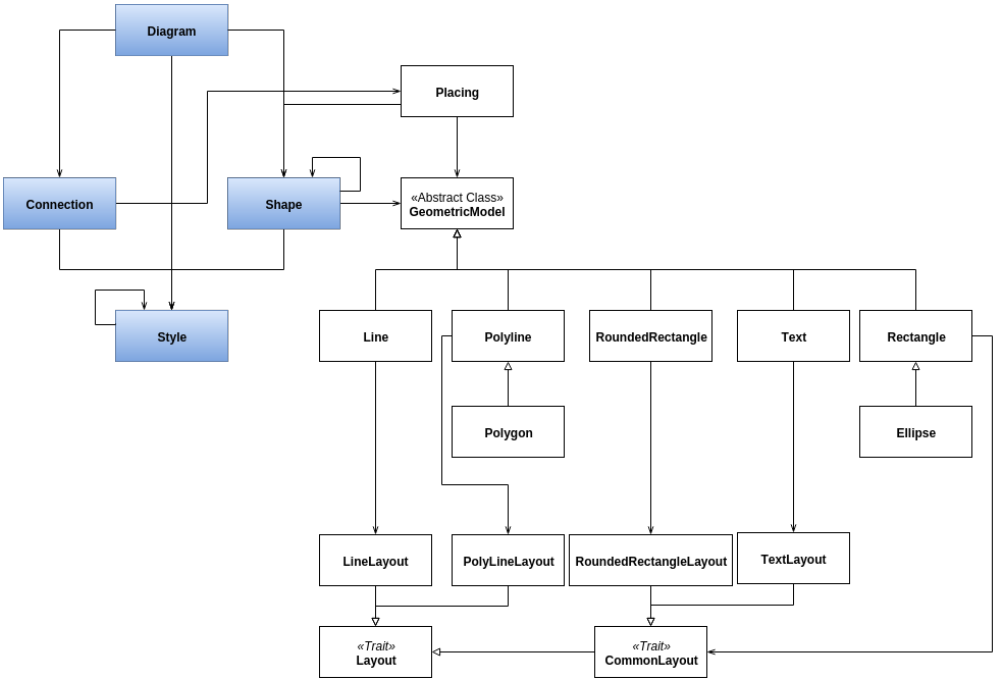

Abbildung 4.2: Vereinfachte Objektstruktur. Im Fokus Shape und seine wichtigsten Dependencies.

4.1.4 Besonderheiten bei Diagram

Die Spray Sprache, in der Diagrams geparst werden, verknüpfen das Metamodell und die Modellklassen. Die Definition eines Diagrams sieht (vereinfacht) aus wie folgt:

```
diagram diagramName for mcoreElement (style: styleName){
        ...
}
```

Diagrams und einige seiner Attribute verweisen auf ein Metamodell, oder dessen Felder, aus dem MoDiGen Projekt. Da das Metamodell zur Zeit der Entwicklung noch nicht bereit war, wurden diese Elemente mit Mockups versehen. Ein Auszug aus dem SprayParser:

```
case identifier => identifier +" is a Mock, change this
   line!!!!"
//TODO this is only a mock, actually Metamodel Attribute
```

Bemerkenswert an dieser Stelle ist jedoch, dass Zugriffe auf einzelne Elemente des Metamodells im Gegensatz zu *Xtext* sehr einfach gestaltet werden können. Dieser Vorteil gegenüber *Xtext* rührt nicht allein durch Scala, sondern der Entscheidung Metamodelle in der ***J**ava **S**cript **O**bject **N**otation* in einer *NoSQL* Datenbank zu hinterlegen. *Xtext* benutzt *XML* (*E**X**tensible **M**arkup **L**anguage*)

Dateien als Speichermedium. Zugriff auf spezifische Elemente erfolgt in *Xtext* über das Laden der gesamten *XML* Datei. Da die Entitäten der Metamodelle in *MoDiGen* einzeln im *JSON* Format in eine *NoSQL* Datenbank eingespeist werden, kann auch einzeln auf entsprechende Elemente zugegriffen werden. Diese Möglichkeit des Datenzugriffs vermeidet (abhängig von der Dateigröße) evtl riesigen Overhead und ist auf Grund der geringeren Bandbreitenauslastung auch wesentlich performanter.
In Abbildung 4.3 ist eine vereinfachte Version der entstandenen Objektstruktur zu sehen, die die wichtigsten Abhängigkeiten der Diagram Modellklasse darstellt.

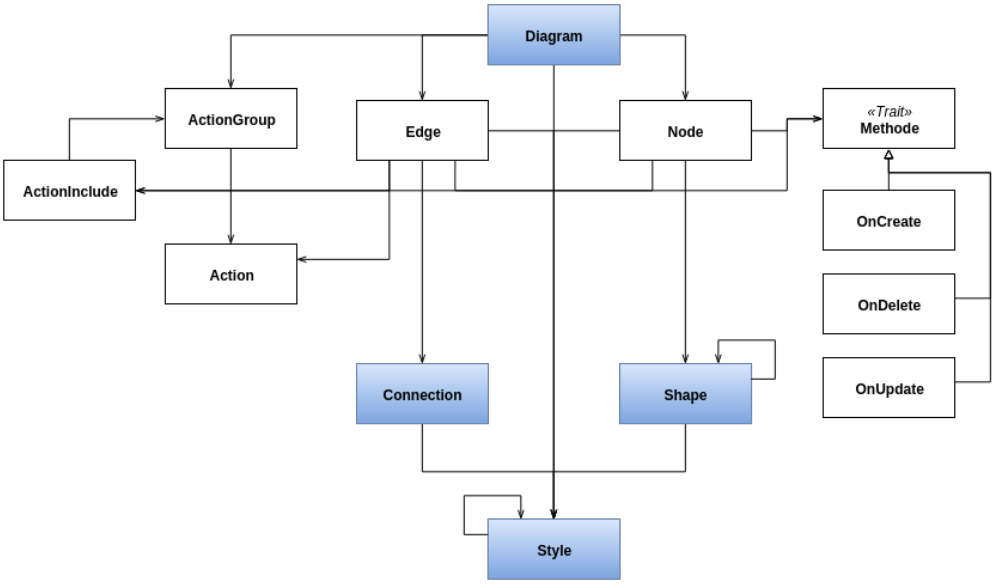

Abbildung 4.3: Vereinfachte Objektstruktur. Im Fokus Diagram und seine wichtigsten Dependencies.

4.1.5 Vererbung/ClassHierarchy

Nachdem die Modellklassen definiert worden sind, müssen diese wie bereits beschrieben auch hinsichtlich der Vererbungshierarchie verwaltet werden können. Entsprechend des Ansatzes wurde - wie ursprünglich geplant - eine generische Klasse namens *ClassHierarchy* erstellt, welche sowohl Knoten definiert,

```
sealed case class Node(data: T,
                       var parents:List[Node]= List(),
                       var children:List[Node]= List(),
                       var depth:Int= 0)
```

als auch eine Map führt, die Namen (Strings) auf entsprechende Knoten abbildet. Da diese Klasse im Stande sein soll, die Namen, also die tatsächlichen String Attribute der Style- und Shapeklassen abzubilden, wurde in in der Typparameterliste folgender Viewbound definiert:

```
sealed class ClassHierarchy[T <% {val name:String}](
   rootClass:T){
```

Die Kennzeichnung *X <: Y* sagt aus, dass der übergebene Typ X implizit auf den Typ Y umgewandelt werden können muss, Braun [2, p. 158]. In Diesem Fall wurde ein anonymer Typ erstellt, der ein Feld *name:String* besitzt. Wenn der angegebene Typ also über ein solches Feld verfügt, kann er implizit darauf abgebildet werden und wird akzeptiert. Desweiteren wurden mehrere apply-Methoden definiert, die den Umgang mit ClassHierarchy erleichtern sollten.

```
def apply(parent:Node, className:T) =
        parent inheritedBy className
def apply(parent:T, className:T) =
        nodeView(parent.name) inheritedBy className
def apply(parent:String, className:T) =
        nodeView(parent) inheritedBy className
def apply(className:T) =
        nodeView(className.name)
def apply(className:String) =
        nodeView(className)
```

Apply Methoden sind spezielle Methoden und können ohne den eigentlichen Methodennamen aufgerufen werden. Die Argumente werden einfach nach der Instanz in Klammern angegeben, Esser [4, p. 74]. Dies ist sehr komfortabel. Eltern- und Kindbeziehungen werden in den Knotenklassen gespeichert. Leider ist ClassHierarchy so jedoch, wie sich herausstellt nicht wie gewünscht einsetzbar. Wird ein neuer Style oder eine neue Shape erzeugt, müssen die Eigenschaften der Elternknoten verfügbar sein, noch bevor der/die eigentliche Style/Shape instanziiert wurde, da diese Eltern und deren Felder noch vor dem Instanziieren des neuen Objekts aufgelistet und abgerufen werden. Das companion Objekt der entsprechenden Klassen sammelt erst alle Informationen an neu geparsten und an geerbten Attributen und gibt diese dann an den richtigen Konstruktor weiter (4.3). Bevor also ein Style/Shape instanziiert ist, kann in ClassHierarchy auch nicht danach gesucht werden, es existiert ja noch nicht. Entsprechende Elternknoten des noch nicht existierenden Knotens sind so ebenfalls nicht ausfindig zu machen. Dieser Denkfehler reduziert die Brauchbarkeit der ClassHierarchy-Klasse auf die Map[String, T], welche diese noch enthält. Da die Namen der Elternklassen über die Definition der Modellklasse mitgeliefert werden, werden die Entsprechenden Elternteile über die Abbildung String=>T der Map gefunden. Die komplette gebrauchte Funktionalität der ClassHierarchy ist also bereits durch eine herkömmliche Map der scala.collections beschrieben. Lediglich die zusätzlichen apply-Methoden der ClassHierarchy, welche den Gebrauch der Map vereinfachen, sorgen derzeit dafür, dass die Klasse einen Nutzen erfüllt. Schließlich behalten die erzeugten Styles/Shapes eine Liste mit ihren Eltern. Da sie in sich alle Merkmale der Superklassen behalten, wäre dies nichteinmal nötig. Jedoch bleibt es zu Testzwecken nützlich eine solche Referenz zu haben, um schnell überprüfen zu können welche Werte übernommen wurden, welche

nicht und warum nicht (z.B. *latest-bound principle*).

4.2 Implementierung der Parserklassen

Sind die Beziehungen unter den Modellklassen definiert, muss als nächstes dafür gesorgt werden, dass entsprechende Klassen auch instanziiert werden können. Da die benötigte Information für die Erstellung einer Modellklasse in Stringform bereitgestellt wird, muss also nun ein Parser eingerichtet werden, der die benötigte Umwandlung der in Stringform enthaltenen Attribute zu den richtigen Datentypen vollziehen kann.

4.2.1 Factory Klassen

Der Parser muss, dem Metamodell entsprechend, sowohl Style-, Shape-, als auch Diagramdefinitionen verarbeiten können (u.a) 1.2. Entsprechend wurden Diagram.scala, Shape.scala und Style.scala erstellt. Zunächst waren diese Klassen als *case classes* geplant, um bestimmte Methoden vorimplementiert benutzen zu können. Allerdings wurde schlussendlich entschieden, Teile der Parserlogik auszulagern. Gemäß dem *Single Responsibility Principle*, lag es nahe sowohl einen Style-, Shape- und Diagramparser zu haben, auf die der eigentliche Parser zurückgreifen kann. Im Idealfall würden diese spezialisierten Parser wie Factorys agieren, um die zugehörige Klasse möglichst entkoppelt instanziieren zu können. Die Factorys wurden als companion Objekte realisiert, da diese sich hierfür besonders eignen. Diese Absicht kreuzt sich leider mit den case classes, da case classes bereits implizit über ein companion Objekt verfügen. Diese companion Objekte können nicht partiell überschrieben werden. Jede der (normalen) Modellklassen hat nun ein companion Objekt, welches eine *parse* Methode enthält, um seine companion Klasse zu instanziieren. Die companion Objekte, können hierbei als *Factory* angesehen werden, da ihre *apply*-Methoden den Parsevorgang ebenfalls initiieren und anschließend eine neue Instanz ihrer companion Klasse zurückliefern. Vergleich Esser [4, p. 80] Eine Modellklasse kann also auf diese Weise (Bsp. Style)

```
val newStyle = Style(argList)
```

erzeugt werden, wobei die Factorymethode Argumente in String Form auflöst (z.B. “line-width = 4“ ->Integer 4 wird aufgelöst und anhand des Attributnamens an der richtigen Stelle im Konstruktor platziert) und der eigentliche Konstruktor nur aufgelöste Werte als Argumente akzeptiert. Das oben beschriebene Prinzip, der Aufteilung der Parserlogik in mehrere kleine Parser, wird konsequent durchgeführt. So haben eigene Datentypen wie z.B. *GradientAlignment* ebenfalls kleine Parsereinheiten in ihren companion Objekten:

```
sealed abstract class GradientAlignment private ()
  case object HORIZONTAL extends GradientAlignment
  case object VERTICAL extends GradientAlignment

object GradientAlignment {
```

```
  def ifValid(s: String) = {
    s match {
      case "horizontal" => Some(HORIZONTAL)
      case "vertical" => Some(VERTICAL)
      case _ => None
    }
  }
}
```

Um sicherzustellen, dass auf eigene Faust keine Modellklasse erzeugt werden kann, wird der Konstruktor auf *private* gesetzt und die Klasse mit *sealed* gekennzeichnet. So ist es ausschließlich über das companion Objekt möglich eine Modellklasse zu erzeugen. Dem companion Objekt ist der Zugriff auf den privaten Konstruktor möglich, da es Zugriff auf alle privaten Member der companion Klasse hat. Ansonsten könnten im schlimmsten Fall sogar abstrakte Klassen anonym instanziiert werden:

```
val gradient_alignment = new GradientAlignment(){}
```

sealed verhindert, dass die gekennzeichnete Klasse außerhalb ihrer Quelldatei erweitert werden kann. So wäre es erneut möglich die Klasse um eine Ecke selber zu instanziieren:

```
class AnotherGradient extends GradientAlignment

&

val anonymous_gradient = new AnotherGradient()
```

Da davon ausgegangen werden kann, dass die Modellklassen nur über geparsten Input erstellt werden können, ist es sicherer auch ihre Instanziierung **ausschließlich** den Parsern zu überlassen. Deshalb sollten alle Modellklassen auf diese Weise definiert werden.

4.2.2 Problematik der rekursiven regulären Ausdrücke

Das tatsächliche Parsen der Strings, erfolgte zunächst über reguläre Ausdrücke. Warum dies in der Umsetzung scheitert und wie das Problem zu lösen ist, wird in diesem Abschnitt erläutert. Zunächst erscheint es einleuchtend die Strings über *Regular Expressions* einzulesen und auszuwerten. Ist die zugrundeliegende Grammatik simpel genug, funktioniert dies auch einwandfrei. Simple *match*-funktionen lassen sich in Scala sehr viel komfortabler anwenden als beispielsweise in Java. Als einfaches Beispiel:

```
"type identifier { val a = 10 }".matches("type [a-z]+
    \\{ (val [a-z]+ = [0-9]+)* \\}")
```

Nun wurde zunächst der Styleparser komplett und fehlerfrei fertig gestellt. Die bestehende Lösung definierte viele undurchsichtige Regeln, nicht zuletzt da

komplexe reguläre Ausdrücke einfach nicht schön zu lesen sind. Entsprechend ist die Skalierbarkeit einer solchen Lösung nicht sehr effizient, da um eine ohnehin schon komplexe Regel zu erweitern, erst einmal die bestehende Regel verstanden werden muss. Dies kann einige Zeit in Anspruch nehmen. Beim erstellen der Regeln für die Shape Klassen stießen die regulären Ausdrücke endgültig an ihre Grenzen. Vor Allem die Anforderung der DSL, ineinander geschachtelte geometrische Figuren abbilden zu können, ist mit regulären Ausdrücken nicht machbar. Eine Shape definition wie:

```
shape <identifier> {
    ellipse {
        size (width=50, height=50)
        ellipse {
            size (width=38, height=38)
        }
    }
}
```

Ist einerseits sehr komplex, da man um sie parsen zu können sehr komplizierten und unschönen Code erzeugen muss. Das Problem ist, dass beim Parsen zwar ein size Attribute ermittelt werden kann, die Zuweisung zur richtigen Ellipse aber höchstens auf einer Indexvariable basieren könnte. Das weitaus größere Problem ist, dass normale reguläre Ausdrücke an komplexen Ausdrücken scheitern, da Scala (Stand Version 2.11.7) keine rekursiven regulären Ausdrücke unterstützt. Setzt die Grammatik also voraus, dass ein Element **beliebig** oft in sich selbst gekapselt werden kann (wird zum Beispiel eine Grammatik beschrieben mit der russische Babuschka Puppen beschrieben werden können),

```
Babuschka {
    Babuschka {
        Babuschka {}
    }
}
```

sind reguläre Ausdrücke dem Problem nicht mehr gewachsen. In diesem Fall muss auf eine komplexere Technologie zurückgegriffen werden.

4.2.3 Lösung Parser Combinators

Hierbei bieten sich die *Parser Combinator* Klassen an. Wie der Name schon vermuten lässt, kann man hier Parser zusammenschalten, um so komplexe Ausdrücke auswerten zu können. Hierfür werden Methoden definiert, welche als Rückgabewert einen *Parser*[*T*] liefern. Für *Parser* sind (u.a.) die Methoden , ¿ und ¡ angegeben, welche sich wie normale Operatoren anfühlen und verwenden lassen.

- $a \sim b$ erzeugt einen Parser, der zuerst a, dann mit b parst und das Gesamtergebnis zurückgibt.

- $a \sim> b$ parst genauso wie $a \sim b$, gibt aber nur das Ergebnis des Parsers a zurück.
- $a <\sim b$ gibt analog nur das Ergebnis von b zurück.

(vgl. Braun [2, p. 182]) Zunächst ein einfaches Beispiel (in der interaktiven scala REPL):

```
scala> import scala.util.parsing.combinator.
   JavaTokenParsers
import scala.util.parsing.combinator.JavaTokenParsers
scala> class NameParser extends JavaTokenParsers {
     | def name = "hello my name is" ~> ident ^^ {_.
        toString}
     | }
defined class NameParser
scala> val p =new NameParser()
p: NameParser = NameParser@2ecaa79e
scala> p.parse(p.name, "hello my name is Julian")
res0: p.ParseResult[String] = [1.24] parsed: Julian
```

Mit relativ wenig Aufwand können sogar sehr komplexe Grammatiken beschrieben werden:

```
scala> import scala.util.parsing.combinator.
   JavaTokenParsers
import scala.util.parsing.combinator.JavaTokenParsers
scala> class BabuschkaParser extends JavaTokenParsers{
        def babuschka:Parser[String]=("babuschka" ~>
           ident) ~ ("{" ~> (babuschka|"") <~ "}") ^^ {_.
           toString}
}
defined class BabuschkaParser
scala> val p = new BabuschkaParser()
p: BabuschkaParser = BabuschkaParser@23a1ef14
scala> p.parse(p.babuschka, "babuschka b1{ babuschka b2 {
   babuschka b3 {babuschka b4{}}}}")
res3: p.ParseResult[String] = [1.36] parsed: (b1~(b2~(b3~
   (b4~))))
```

Umgesetzt wurden so beispielsweise die Rekursiven *GeometricModels*- Geometrische Formen - welche in sich gekapselt erzeugt werden können:

```
private def geoModel: Parser[GeoModel] =
    geoIdentifier ~
    ((("style" ~> ident)?) <~ "{") ~
    rep(geoAttribute) ~
```

```
    (rep(geoModel) <~ "}") ^^ {
      case name ~ style ~ attr ~ children =>
        GeoModel(name, style, attr, children, cache)
      }
```

Wie man sieht ist die Methode *geoModel* rekursiv, da sie sich selbst aufruft. Doch zunächst erst ein paar einfachere Parser.

4.2.4 Prinzip der Parserlogik

Die gewählte Technologie zum Parsen sind also *Parser Combinators*, über welche komplexe Problemstellungen gelöst werden können. Wie die einzelnen Parser untereinander assoziiert sind, was ihre jeweiligen Aufgaben sind und welche Gedanken dahinter stecken wird nun beschrieben. Wie bereits erwähnt, sollte der Parser möglichst generisch funktionieren, um das Parsen der verschiedenen Modellklassen im Idealfall möglichst einheitlich gestalten zu können. Konkret heißt das, dass versucht wurde so wenig expliziten Inhalt der Modellklassen abzufragen wie möglich. Anstatt eine Regel aufzustellen, die aussehen könnte wie folgt:

```
def parseLineWidth:Parser[Int] = "line-width = " ~> int
```

wurden Regeln gewählt, die allgemeiner auf die Attribute eingehen:

```
def attributePair: Parser[(String, String)] =
        variable ~ arguments ^^ { case v ~ a => (v, a) }
def variable:Parser[String] =
    "[a-züäöA-ZÜÄÖ]+([-_][a-züäöA-ZÜÄÖ]+)*".r <~ "\\s*".r
def arguments: Parser[String] =
    argument_classic | argument_advanced_explicit |
    argument_advanced_implicit | argument_wrapped
```

Es wird also zunächst definiert, dass beim Parsen von Attributen ein *attributePair* erwartet wird. Dieses attributePair besteht aus einer *variable* und (~) *arguments*. Eine Variable wird in der nächsten Regel bzw. Methode definiert. Die vielen verschiedenen Möglichkeiten den Variablenwert zu ermitteln führt darauf zurück, dass dieser Teil nun eben sehr variabel ist. An dieser Stelle kann jeder dem System bekannte Datentyp folgen. Während ein *argument_classic* erwartet, dass ein '=' Zeichen und anschließend irgendein Int, double oder String folgt, deckt ein *argument_advanced_explicit* die Möglichkeit eines Arguments in folgender Form:

```
someSize ( width = 10, height = 12)
```

Da diese und weitere Parserregeln öfter gebraucht werden, sind diese in einem Trait namens *CommonParserMethods* zusammengefasst. Die companion Objekte der Modellklassen, die gleichzeitig den Parser bilden erweitern diesen Trait. Blickt man zurück auf *attributePair*, sieht man außerdem, dass diese Methode ein *Tupel* zurück gibt, welches eben zum einen den Variablennamen und außerdem den Wert zurückliefert. Da An dieser Stelle des Parsers noch keiner der Werte in seinen eigentlichen Typ konvertiert wird, kann nun auch ein

einheitlich typisiertes Set an Tupel[String, String] an die tiefer liegenden Parser weitergegeben werden. Wird also gerade beispielsweise ein Style geparsed, werden die erhaltenen Tupel nun an das companion Objekt **Style** übergeben, welches sich selber um seine Attribute zu kümmern hat. Bisher wurde also nur darauf geachtet, dass der Inputstring möglichst mit keiner Konvention der zugrundeliegenden Grammatik bricht, außerdem wird so ein einheitliches Medium garantiert, nach dem sich die spezialisierten Parser der Modellklassen richten können. Da diese grundlegenden Regeln noch keinen abstrakten Syntaxbaum bilden, sondern nur dem Identifizieren der einzelnen Wörter der DSL dient bildet diese logische Einheit (*SprayParser.scala*) den *Lexer* der Anwendung (vgl. Ferg [5]). Da sich die spezialisierten Parser nun ihr Inputformat teilen, können sie folglich auch mit denselben Methoden weiterarbeiten. So muss das Rad nicht für jeden Parser neu erfunden werden. Ab jetzt passiert im spezialisierten Styleparser die eigentliche Magie. Bevor die übergebenen Werte in den Tupeln nun ihre endgültige Form erhalten, widmen sich die Factorys zunächst der Vererbung. Dazu mehr in 4.3. Nachdem die vererbten Werte berücksichtigt sind, werden nun die Tupel mit den Attributen auseinandergenommen und vordefinierten Variablen zugewiesen, welche später an den Konstruktor der Modellklasse weitergegeben werden. Die Factory erhält also ein Set mit Attributen. Diese Attribute sind in Form eines Tupels[String, String] und enthalten jeweils den Namen des zuzuweisenden Feldes und den Attributswert(aber immer in String Form). Um Die Werte aufzulösen und zuzuweisen, wird eine mächtige Technik namens *Pattern Matching* eingesetzt.

```
attributes.foreach{
    case ("description", x) =>
        description = Some(x)
    case ("transparency", x) =>
        transparency = ifValid(x.toDouble)
    case ("line-style", x) =>
        line_style= LineStyle.getIfValid(x)
    ...
}
```

So wird immer zunächst geprüft um welches Feld es sich handelt. Anschließend wird der String falls nötig auf den gewünschten Typ konvertiert und zugewiesen. Da Fehleingaben wie Buchstaben an Stelle einer erwarteten Zahl möglich sind, wird über eine Hilfsmethode *ifValid* sichergestellt, dass es nicht zum Programmabsturz kommt, sondern im Fehlerfall ein Defaultwert eingesetzt wird.

```
def ifValid[T](f: => T):Option[T] = {
      var ret:Option[T] = None
      try { ret = Some(f)
      ret
      }finally {
        ret
```

```
        }
}
```

ifValid erwartet einen Typparamter und einen Codeblock, der ein Ergebnis entsprechend des Typparameters zurückliefert. Der gewünschte Codeblock wird in einer sicheren *try* umgebung ausgeführt. Bei erfolgreicher Ausführung wird das Ergebnis des Codeblocks zurückgeliefert, ansonsten *None*. Der Aufruf von *ifValid* kann wie man sieht, aber sogar ohne Typparameter erfolgen, da Scala diesen auch hier anhand der Zuweisung erkennen kann und ihn inferiert (Scala ist klasse!). Im Falle der Styles werden größtenteils nur primitive Datentypen geparsed, daher ist das konvertieren der Werte in die eigentlichen Datentypen auch eher einfach. Im Falle der Shape Modellklasse werden unter anderem mehrere ineinander gekapselte geometrische Formen geparsed. Beim Parsen der Ellipsen, Rechtecke etc. sind mehrere Parser beteiligt. Beim Pattern Matching des Shape companion Objekts wird daher in der Regel auf weitere spezialisiertere Parser verwiesen.

4.3 Vererbung

Da Styles und Shapes auch mit einem *extends* Schlüsselwort definiert werden können, muss auch ein Mechanismus vorhanden sein, der sich um die Vererbungshierarchie kümmert und die Werte der erweiterten Klassen abrufen kann. Bisher wurde die Vererbung über eine Referenz zum Elternknoten geregelt. Wenn also ein gefragter Wert in der aktuellen Klasse nicht gefunden wurde, suchte man rekursiv in den Elternklassen danach. Dies erfordert nun einerseits bei größeren Hierarchien einige Zeit, außerdem eventuell komplizierte rekursive Funktionen. Ein leichterer und performanterer Weg wurde darin gefunden, die Werte eines Elternknotens direkt beim Erzeugen der neuen Instanz auf das Kind zu übertragen. Das Prinzip wird in einem Beispiel deutlich(Bsp. Style): Man erzeugt einen Style A, mit dem Attribut *line-width = 10*

```
style A {
    line-width = 10
}
```

anschließend einen Style B, der von A erbt.

```
style B extends A {
    transparency = 0.5
}
```

Nun werden die beiden Definitionen dem Parser übergeben, der die Attribute filtert und an spezialisierte Parser weiter gibt. In dem *Companion Object* von Style, in dem die Attribute für einen neuen Style aufgelöst werden, wird nun zuerst die Liste an Eltern erstellt.

```
val extendedStyle = parents.getOrElse(List()).foldLeft(
```

```
    List[Style]())((styles, s_name) =>
      if(cache.styleHierarchy.contains(s_name.trim))
        s_name.trim :: styles else styles)
```

Dafür wird in der vom Parser mitgegebenen *ClassHierarchy* (*styleHierarchy*) nach den Namen der Eltern gesucht. Anschließend werden die verschiedenen Variablen, die später an den Konstruktor übergeben werden wie folgt vordefiniert:

```
var description:Option[String]=relevant{_.description}
var transparency:Option[Double]=relevant{_.transparency}
var background_color:Option[ColorOrGradient]=relevant{_.
    background_color}
var line_color:Option[Color]=relevant{_.line_color}
var line_style:Option[LineStyle]=relevant{_.line_style}
var line_width:Option[Int]=relevant{_.line_width}
var font_color:Option[ColorOrGradient]=relevant{_.
    font_color}
var font_name:Option[String]=relevant{_.font_name}
var font_size:Option[Int]=relevant{_.font_size}
var font_bold:Option[Boolean]=relevant{_.font_bold}
var font_italic:Option[Boolean]=relevant{_.font_italic}
```

Auffällig ist hierbei, dass die Zuweisungen beinahe auch für nicht Programmierer lesbar sind. Wie zum Beispiel die erste Zuweisung des obigen Codeausschnitts: *"Weise der description die relevante bestehende description zu"*. Hinter diesem Aufruf verbirgt sich eine Funktion, die in ClassHierarchy definiert wurde, um einfach auf die Attribute der Elternklasse zugreifen zu können.

```
def mostRelevant[T, C](stack:List[C])(getter: C => Option
    [T]):Option[T] = {
    for (parent <- stack) {
      if(getter(parent).isDefined)
        return f(parent)
    }
    None
}
```

mostRelevant macht gebruach von mehreren nützlichen Eigenschaften von Scala. Zunächst wir die Funktion mit Typparametern beschrieben. Was diese jeweils beschreiben sollen wird gleich klar. Sinn und Zweck der *mostRelevant* Funktion ist es dem Benutzer zu ermöglichen, einen generischen *getter* auf beliebige Felder der Oberklassen zu bieten. Sowohl Style als auch Shape führen eine Liste mit den Instanzen, die sie erweitern. Genauer, führen Style und Shape eine Art *Stack*, denn bei der Vererbung soll hier das *latest-Bound-Principle* gelten. Dieser Stack wird als erster Parameter übergeben. Als nächstes sieht man eine weitere ParameterListe, die eine anonyme Funktion erwartet. Da die Funktion komplett typunabhängig funktionieren soll, wurden Typparameter benutzt. C

stellt in diesem Fall den Typ der Modellklasse dar, T hingegen ist der Typ des gewünschten Elternattributs. Die anonyme Funktion bildet eine Modellklasse (C) auf den gewünschten Typ (T) ab und ist somit als Getter zu verstehen. Die Funktion iteriert also über die Elternklassen, wobei die Relevanz beziehungsweise die Priorität vom ersten Element der Liste, bis zum Letzten absteigend ist. Da alle Felder der Modellklassen *Options* sind, kann nun für jede Superklasse geprüft werden, ob das gewünschte Feld definiert ist. Dieses gewünschte Feld wird wiederum über die mitgelieferte Funktion ermittelt. Wird in Keiner der Elternklassen ein passendes Feld gefunden, wird *None* zurückgegeben. Da der zweite Parameter in einer eigenen Parameterliste angegeben ist, kann das entsprechende Argument anstatt in runden Klammern, in eckigen Klammern angegeben werden und fügt sich somit wie eine neue Kontrollstruktur in den Code ein. Momentan müsste ein Aufruf aus dem Style companion Objekt wie folgt aussehen:

```
var description: Option[String] =
    ClassHierarchy.mostRelevant(extendedStyle){ _.
       description }
```

Um selbst diesen Aufruf noch zu verkürzen wird sich einfach einer weiteren Hilfsmethode, namens *relevant* bedient:

```
def relevant[T](f: Style => Option[T]) =
    ClassHierarchy.mostRelevant(extendedStyle) {f}
```

Diese setzt die Elternliste schon einmal voraus und erwartet ab hier nur noch eine Funktion, welche aus einem Style element ein beliebiges T extrahiert. Selbst jetzt müsste der Aufruf eigentlich noch wie folgt aussehen:

```
var description: Option[String] = relevant[String]{ _.
   description }
```

Doch auch hier kann Scala den Typparameter selbst ermitteln und inferieren. Somit reduziert sich das Abrufen des am meisten relevanten Feldes eines beliebigen Typs einer Elternklasse auf den Aufruf:

```
var description: Option[String] = relevant{ _.description
    }
```

Da Scala Funktionen als *First-Class Objects* behandelt werden, können sie eben auch als Argumente vergeben werden Esser [4, p. 244]. In Sprachen, in denen Funktionen anders behandelt werden wie zum Beispiel Java, wäre dies so nicht möglich. Um bei der Stylegenerierung den Wert einer Elternklasse für das gewünschte Feld zu ermitteln, müsste dort für jedes einzelne Attribut eine eigene Funktion erstellt werden. Diese iteriert durch die Elternklassen und führt dann eine nicht variable eindeutige Getterfunktion aus. Scala kommt hier mit einer einzigen Funktion und einer weiteren Hilfsfunktion (relevant), welche nur

dem Komfort dient, aus.

4.3.1 Vererbung bei Shapes

Die Vererbungslogik hat sich bisher allgemein auf alle Felder der Modellklassen bezogen. Im Falle einer erweiterten Shape, sind zunächst zwei Fragen prinzipiell zu klären:

- Werden geerbte geometrische Figuren referenziert, oder müssen tiefe Kopien davon erstellt werden?
- Werden geometrische Figuren an geometrische Figuren oder an Shapes vererbt?

Referenzierung oder tiefe Kopie geerbter Felder?

Eine Referenzierung der geerbten geometrischen Figuren wäre natürlich wünschenswert, da insbesondere große geschachtelte Bäume aus geometrischen Figuren sehr viel speicherschonender behandelt werden würden. Gefahrlos umsetzen lässt sich dies aber nur wenn garantiert wird, dass eben jene geometrischen Figuren konstant sind. Der Ansatz ist natürlich alle Shape Member über ein **val**, also konstant zu definieren. Dies setzt jedoch voraus, dass die Anforderung einer späteren Änderung der Felder trotz der Shape vererbung nicht nötig ist. Daraus ergibt sich die nächste Frage.

Geerbte geometrische Figuren

Wird beispielsweise also ein Shape A definiert, welche eine geometrische Figur enthält

```
shape A { rectangle{...}}
```

und wird nun eine Shape B definiert, welche selber eine geometrische Figur enthält und zusätzlich Shape A erweitert,

```
shape B extends A { ellipse {...}}
```

so ergeben sich nun zwei Möglichkeiten, die Shapevererbung umzusetzen: Möglichkeit 1

```
shape B extends A {
    ellipse{
        rectangle {...}
    }
}
```

und Möglichkeit 2

```
shape B extends A {
    ellipse{...}
```

```
    rectangle{...}
}
```

Würde die Shapevererbung bedeuten, dass die bereits bestehenden geometrischen Figuren z.B. ausgehend von der Tiefe erweitert werden (Möglichkeit 1), würde dies ebenso bedeuten, dass geometrische Figuren auch nach ihrer Erzeugung verändert werden können müssen. Damit wären sie nicht immutabel und könnten nicht gefahrlos referenziert werden. Ergo wäre die Speicherauslastung höher. Möglichkeit zwei schlägt vor, vererbte geometrische Figuren als seperaten Baum in die neu erzeugte Shape einzufügen. Die im Beispiel beschriebene Ellipse würde nicht nachträglich verändert werden müssen und kann somit als immutabel definiert werden. Nun reicht eine einfache Referenz auf die vererbten Felder und senkt die Speicherauslastung.

4.3.2 Transitivität der Styleeigenschaften

Eine wichtige Anforderung an die Objektstruktur ist, dass Styleinformationen in Shapes an die beinhalteten geometrischen Figuren weiter gegeben werden. So wird erreicht, dass geometrische Figuren mit gleichen Farb-, Font- und anderen Eigenschaften erstellt werden. Wenn also einer Shape A ein Style S zugewiesen wird und ferner, A einen Baum aus geometrischen Figuren enthält

```
rectangle{
    ellipse{
        rectangle{...}
    }
    line{...}
}
```

muss gewährleistet werden, dass sowohl rectangle, ellipse, line und polygon über die selben Styleinformationen von S verfügen, wie ihr Eltern Shape A. Dabei hört es noch nicht auf, denn sollte nun die beschriebene Ellipse ebenfalls einen Style (S1) zugeordnet bekommen, so müssen nun rectangle und line über die Styleinformationen von S verfügen. Ellipse und alle seine Kinder müssten hingegen sowohl über die von S, als auch über die von S1 etc. verfügen. Dabei hört es ebenfalls noch nicht auf. Denn entgegen einer Stylereferenz, ist es geometrischen Figuren außerdem auch möglich einen anonymen Style innerhalb ihres Scopes zu definieren. Durch diesen anonymen Style, werden die vorherigen Styleinformationen jedoch **nicht** revidiert, sondern wenn überhaupt erweitert. Es muss also aufgepasst werden, dass Styleinformationen in die Tiefe weitergegeben werden und dabei von tieferen Styles, ob anonym oder nicht, gegebenenfalls erweitert werden. Ein vereinfachtes Beispiel demonstriert anhand zweier vordefinierter Styles und einer Shape mit mehreren geometrischen Figuren was gemeint ist:

```
style S1 {
    color = blue
}
```

```
style S2 {
        transparence = 0.5
}

shape A style S1 {
    rectangle {
        ellipse {
            rectangle style S2 {
                style {
                    color = red
                }
            }
        }
    }
}
```

Diese Definitionen fertig umgesetzt also aussehen wie 4.4

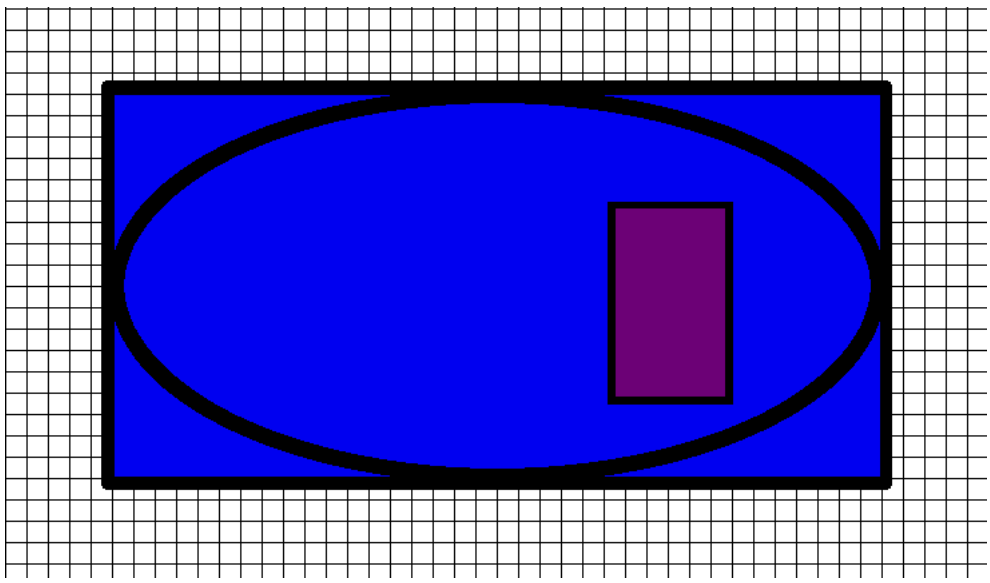

Abbildung 4.4: Beispiel transitiver Styleinformation

In 4.4 ist zu sehen, dass die Ellipse die Farbeigenschaften des äußeren Rechtecks übernimmt, das innere Rechteck, jedoch sowohl seine eigene Farbeigenschaft (rot) definiert und außerdem die Transparenz aus S2 umgesetzt ist. Bei der Vererbung wurde bisher wie bereits erwähnt darauf geachtet, Felder der Superklassen direkt bei der Erzeugung der neuen Instanzen weiter zu geben, um rekursives Suchen der entsprechenden Attribute in Listen zu vermeiden. Ebenso sind hier die Prinzipien der transitiven Styleeigenschaften über Vererbung gelöst und auch in diesem Fall wird darauf verzichtet nur Referenzen zu speichern, über die Elternknoten erreicht werden können. Die Parserregel für

ein *GeometricModel* beziehungsweise dessen "Skizze" dem GeoModel erkennt anonyme Styles während dem Parsen. Da Anonyme Styles **keine** Informationen über Elternpaare besitzen zu brauchen, werden sie direkt instanziiert und in einem Cache hinterlegt. Der Attributesliste des entsprechenden GeoModel wird lediglich der Name des anonymen Styles mitgegeben (ja, der anonyme Style bekommt intern einen generierten Namen, den der Benutzer nicht kennt). So kann weiterhin das einheitliche Medium einer Argumentenliste aus Strings an die Factorys weitergegeben werden. Die entsprechende Factory löst das Argument schlussendlich über den Namen auf und erhält wieder die eigentliche Styleinstanz. Styles werden wie andere Attribute auch in den entsprechenden Parserklassen aufgelöst und an die Konstruktoren der Modellklassen weitergegeben. Am Beispiel des *CommonLayout*s wird deutlich wie die Stylevererbung funktioniert. Wie die anderen Parser auch, erzeugt das *CommonLayout* zuerst Variablen, welche ausgehend von geparster und geerbter Information initialisiert und später an den Konstruktor überreicht wird. Für den Styleparameter erzeugt das *Commonlayout* also zunächst:

```
var styl:Option[Style] = Style.makeLove(cache,
   parentStyle, geoModel.style)
```

Hier ist zu sehen wie die Styleinformationen der nächst höheren Instanz und der eigenen Stylereferenz ausgewertet werden. Mehr zu der Funktion *makeLove* im weiteren Verlauf beschrieben. Wird unter den Attributen nun beim Pattern Matching ein String gefunden, auf den ein Style abgebildet werden kann, handelt es sich um einen anonymen Style, der ebenfalls in die bestehenden Styleinformationen eingearbeitet wird.

```
case anonymousStyle:String if cache.styleHierarchy.
   contains(anonymousStyle) =>
        styl = Style.makeLove(cache, styl, Some(
           anonymousStyle))
```

Was genau macht nun *makeLove*? MakeLove ist eine Funktion des Style companion Objektes und wird wie folgt definiert.

```
def makeLove(cache: Cache, parents:Option[Style]*):Option
   [Style] ={
    val parentStyles = parents.filter(_.isDefined)
    if(parentStyles.length == 1) return parentStyles.head
    else if(parentStyles.isEmpty) return None
    val childName =
        "(child_of -> "+parentStyles.map( p => p.get.name
           +{if(p != parentStyles.last)" & "else ""}).
           mkString+")"
    Some(Style(childName, Some(parentStyles.toList.map(i
       => i.get.name)), List[(String, String)](), cache))
```

```
}
```

Die Funktion erwartet eine variable Anzahl an Styles (Scala erlaubt diese *varargs* durch die Kennzeichnung *), prüft anschließend, ob es sich um gültige Argumente handelt. Im Falle, dass mehr als ein gültiges Argument unter den Eltern ist, wird ein **neuer** Style über die Factory erzeugt, der die entsprechenden Elternteile mitbekommt. Die Stylefactory kümmert sich nun wieder darum, die Felder des neuen Styles anhand des *latest Bound Prinzips* zu erben, wodurch ein neuer "Kind" Style entsteht.
Im Folgenden wird nun ein weiteres Problem adressiert.

4.3.3 Modell und Skizze; Problematik der Instanziierungsreihenfolge

Über Style, Connection und Shape ist es möglich in der Definition des allumfassenden Diagrams ebenfalls eine Stylereferenz anzugeben. Effektiv soll hierduch ein *Corporate Design* realisiert werden können, welches sämtliche Eigenschaften an unterliegende Connections, Shapes und somit auch geometrische Figuren überträgt, Gerhart and Boger [6, 3 Approach]. Um wie bereits beschrieben die Möglichkeit zu bieten, die Modellklassen in Zukunft parallelisiert bearbeiten zu können, wurden alle Attribute der Modellklassen als *vals* gekennzeichnet und sind somit Konstant. So können ruhigen Gewissens funktionale Programmierparadigmen darauf angewendet werden und entsprechender Code kann hoch parallel arbeiten. Entsprechend ist es nicht möglich eine (beispielsweise) Shape zu erzeugen, anschließend ein Diagram zu erstellen, das auf die bereits fertige Shape referenziert und ihr somit nachträglich Styleinformationen übertragen zu wollen. Die Shape ist ja bereits erstellt und alle Felder, inklusive der Styleinformationen, sind bereits aufgelöst. Hier wird nun die selbe Taktik angewandt, wie zuvor bei den geometrischen Figuren 4.4. Da die Erzeugung der Shapeinstanzen folglich erst erfolgen darf, wenn mögliche Corporate Styleinformationen bereits gegeben sind, werden Shapes zunächst über eine Methode namens *shapeSketch* geparsed.*ShapeSketch* sammelt für eine angehende Shape, wie das *GeoModel* für angehende *GeometricModel*s alle geparsten Informationen, löst diese aber noch **nicht** auf. Beim Einlesen eines Diagrams, werden sowohl Style als auch Shape referenziert. Erst also wenn alle nötigen Informationen vorhanden sind, wird während des Parsing Prozesses des Diagrams die Shape Skizze(*ShapeSketch*) mit dem Corporate Style zu der eigentlichen Shape aufgelöst. Hierfür verfügt die *ShapeSketch* Klasse über eine Methode namens *toShape*, die die gespeicherten Attribute an die Shape Factory übergibt. So erzeugte "Skizzen" sind somit nur Container um angehende Modellklassen zu einem beliebigen Zeitpunkt auflösen zu können.

```
case class ShapeSketch(name:String,
                       parents:Option[List[String]],
                       style:Option[Style],
                       attrs:List[(String, String)],
                       geos:List[GeoModel],
```

```
                        descr:Option[(String, String)],
                        anch:Option[String],
                        cache: Cache){
  def toShape(corporateStyle:Option[Style]) =
      Shape(name, parents, Style.makeLove(cache,
         corporateStyle, style),
            attrs, geos, descr, anch, cache)
}
```

Außerdem wird der nun vorhandene Corporate Style und der Style der Shape, über die bereits vorgestellte Funktion *Style.makeLove* kombiniert. Da der Corporate Style auf diese Weise ganz oben in der Shape eingesetzt wird, verteilt er sich wie bereits beschrieben (siehe 4.3.2) automatisch auf die unterliegenden geometrischen Formen, da die entsprechenden Factorys die überliegenden Styleinstanzen analysiert und entsprechend erweitert. Das Problem der Erzeugungsreihenfolge zieht sich durch das gesamte Projekt. Die Abhängigkeit zu Styleinstanzen lässt es nicht zu, irgendeine Modellklasse vor dem Diagram zu erzeugen, da Styleinformationen bis hin zum Diagram immer erweitert werden können. Folglich ist es zwingend erforderlich für sämtliche Modellklassen, die von Style abhängen, Container zu erstellen, die die unaufgelösten Attribute der Modellklasse beinhalten und es ermöglichen die Erzeugung der eigentlichen Instanz auf einen späteren Zeitpunkt zu verschieben. So findet sich zu jeder Modellklasse und jeder Hilfsklasse, die von Style abhängt, eine entsprechende Containerklasse, die mit *Sketch* gekennzeichnet ist:

Diagram Node und NodeSketch; Edge und EdgeSketch

Shape Shape und ShapeSketch; GeometricModel und GeoModel(einzige Ausnahme, endet nicht mit Sketch)

Connection Connection und ConnectionSketch

Diese Lösung resultiert aus der Vorgabe, dass alle Felder der Modellklassen immutabel sein sollen und dem gewählten Lösungsweg, vererbte Eigenschaften bei der Objekterzeugung zu übergeben und nicht über Referenzen auf Elternelemente beim Aufrufzeitpunkt zu suchen. Der so gewonnene Vorteil während der Benutzung der Modelle mit besseren *Response Times* rechnen zu können, da die Rekursivität des Problems aufgelöst wurde, wird also im Endeffekt durch einen zusätzlichen Overhead an Speicherauslastung wieder wet gemacht. Erzeugte Container/Skizzen können nach Auflösung zur entsprechenden Instanz der Modellklasse nämlich nicht verworfen und dem *Garbage Collector* überlassen werden, da weitere Diagrams ebenfalls noch in der Lage sein müssen unberührte Skizzen der Modellklassen benutzen zu können. Die Klasse *SprayParser* stellt die eigentliche Schnittstelle zum Parser dar. Sie enthält alle notwendigen Methoden um das Parsen eines Strings einzuleiten.

parseStyle Zum parsen von n Styles

parseShape Zum parsen von n Shapes

parseAbstractShape Zum parsen von n abstrakten Shapes

parseConnection Zum parsen von n Connections

parseAbstractConnection Zum parsen von n abstrakten Connections in einem String

parseDiagram Zum parsen von n Diagrams in einem String

Diverse weitere Methoden, sind privat um den SprayParser nach außen hin möglichst simpel und benutzerfreundlich zu gestalten. Wird eine Methode mit der Kennzeichnung *Abstract* zum Parsen einer Shape oder Connection benutzt, werden tatsächliche Shape bzw. Connection Instanzen zurückgelifert, jedoch auch für diesen Prozess die entsprechenden Container Skizzen angelegt. Das dient dem Zweck, dass es für Diagrams auch möglich ist abstrakte Shapes zu referenzieren hat aber eine sehr wichtige Konsequenz. Referenziert ein Diagram eine abstrakte Shape (oder Connection), wird die entsprechende *ShapeSketch* verwendet, damit corporate Styles beachtet werden können. Die bereits bestehende Shape Instanz wird im *Cache* sodurch aber überschrieben, da der entsprechende Container neu umgewandelt wird und der bestehende Eintrag im Cache mit dem entsprechenden *Key* (Name der Shape) überschrieben wird. Abstrakte Shapes sollten also nur dann von Diagrams referenziert werden, wenn absichtlich darauf verzichtet wird, dass Unterklassen der abstrakten Shape den corporate Style des Diagrams dadurch **nicht** implizit übernehmen. Deren Styleinformationen sind ja bereits aufgelöst und können wie bereits erklärt nicht nachträglich aktualisiert werden. Um dieses Prinzip zu verdeutlichen ein kleines Beispiel: Es wird eine Shape *A* definiert, welche einen bestimmten Style benutzt und als abstrakte Shape geparst wird.

```
val abstractShape = """shape A style DefaultStyle{...}"""
parser.parseAbstractShape(abstractShape)
```

Eine weitere Shape *B*, die *A* erweitert, wird definiert und als normale Shape geparst.

```
val normalShape = """shape B extends A{...}"""
parser.parseShape(normalShape)
```

Wird nun ein Diagram definiert, welches umbedingt die abstrakte Shape *A* referenzieren soll und ein corporate Style namens *CorporateStyle* benutzt

```
val diagram = """diagram D ... (style:CorporateStyle){
                node ... {
                        shape:A(...)
                }
        }"""
parser.parseDiagram(diagram)
```

und anschließend eine neue Shape *C* definiert, welche die abstrakte Shape *A* erweitern soll,

```
val normalShape2 = """shape C extends A{...}"""
parser.parseShape(normalShape2)
```

dann hat nun auch die Shape C die Styleinformationen von sowohl dem *CorporateStyle* und dem *DefaultStyle*, da die Shape, die sie referenziert von der Diagram Definitin neu erzeugt worden ist. Da abstrakte Shapes in der Regel nur erzeugt werden um tests mit Shapes durchzuführen, ohne ein Diagram erstellen zu müssen ist der beschriebene Fall normalerweise nicht von Belang, da Diagrams normalerweise auf keine abstrakte Shape verweisen sollten. Hier muss überlegt werden, ob sich aus dem Referenzieren von abstrakten Shapes ein zusätzlicher nützlicher Effekt ergeben kann, oder ob dadurch eher eine potenzielle Fehlerquelle eingeführt wird. Je nach dem muss über eine Prüfung verhindert werden, dass Shapeinstanzen überschrieben werden können. Wird eine Shape bzw. Connection über *parseShape* bzw. *parseConnection*, also ohne die Abstract Kennzeichnung, geparst werden zunächst **keine** tatsächlichen Instanzen der Modellklassen erzeugt, sondern nur die der Container Klassen *ShapeSketch* bzw. *ConnectionSketch*. Jetzt erscheint es natürlich an dieser Stelle paradox, dass die Methoden mit der *Abstract* Kennzeichnung tatsächliche Instanzen der Modellklassen erzeugen und die ohne *Abstract* Kennzeichnung nur Container Klassen. Aus Anwendersicht ergibt dies schon mehr Sinn. Als Anwender definiere ich Shapes und Connections nämlich eigentlich nur dann, wenn ich sie später auch in einem Diagram referenziere. Mit anderen Worten sind die tatsächlich benutzten Shapes und Connections die, die über die Methoden *parseShape* bzw. *parseConnection* eingelesen werden. Shapes und Connections, die ich nur dafür einlesen will, um ähnliche Eigenschaften zusammenzufassen und durch Unterklassen zu diversifizieren (sinngemäß also abstrakt), werden über die Methoden *parseAbstractShape* bzw. *parseAbstractConnection* eingelesen.

Kritik an gewählter Taktik

Im Falle, dass mit Referenzen auf Styles der Elternobjekte gearbeitet werden würde, wäre es möglich die Styleinformationen der Elterninstanz und die eigenen erst zu erzeugen, wenn sie benötigt werden. So wäre kein Bedarf für die vorgestellten Container Klassen. Je nach dem wie tief die Vererbungshierarchie ist, würde dies jedoch beträchtliche Nachteile im Bezug auf die *Responsetime* des später erzeugten graphischen Editors mit sich bringen. Die Skizzen Klassen zu den jeweiligen Modellklassen wurden in der *SprayParser.scala* Datei definiert, da andere Parsingstrategien evtl nicht mehr abhängig von ihnen sind.

4.4 Ausführliches Beispiel

Nun folgt ein ausführliches Beispiel anhand einer Shapedefinition, zur Verdeutlichung der Schritte, die beim Parsen einer Modellklasse, durchlaufen werden:

```
shape EClassShape style B{
    size-min (width=4, height=6)
    rectangle {
        style (line-width=2)
        position (x=2, y=0)
```

```
        size (width=10, height=3)
        ellipse {
            position (x=0, y=36)
            size (width=30, height=30)
        }
    }
}
```

Soll eine Shapedefinition wie die obige eingelesen und direkt in eine Shapeinstanz umgewandelt werden, wird sie zunächst im **SprayParser** (der Standard Parserklasse) durch folgende Regeln/Parser überprüft:

```
[ 1]private def shapeSketch:Parser[Shape] =
[ 2]    ("shape" ~> ident) ~
[ 3]    (("extends" ~> rep(("(?!style)".r ~> ident)<~ ",?
   ".r))?) ~
[ 4]    (("style" ~> ident)?) ~
[ 5]    ("{" ~> rep(shapeAttribute)) ~
[ 6]    rep(geoModel) ~
[ 7]    (descriptionAttribute?) ~
[ 8]    (anchorAttribute?) <~ "}" ^^
[ 9]    {
[10]      case name ~ parent ~ style ~ attrs ~ geos ~
   desc ~ anch =>
[11]      ShapeSketch(name, parent, style, attrs, geos,
   desc, anch, cache)
[12]    }
```

Warum die Methode/Regel in Zeile 1 *shapeSketch* heißt wurde in 4.3.3 erklärt). Zeile 2 filtert den identifier, der Shape. Zeile 3 prüft auf eine optionale Angabe erweiterter Shapes. Hierbei dürfen diese nicht „style“ heißen, da die Regel sonst verwirrt wäre, erwartet sie doch in Zeile 4 eine optionale Angabe für einen benutzten Style. Zeile 5 sucht nun n mal nach einer Regel namens shapeAttribute

```
private def shapeAttribute = shapeVariable ~ arguments ^^
    {case v ~ a => (v, a)}
private def shapeVariable = ("""("""+Shape.
   validShapeVariables.map(_+"|").mkString+""")""").r ^^
   {_.toString}
```

validShapeVariables ist eine Sammlung von Strings, welche die bekannten Felder einer Shape darstellen. Diese Sammlung wird zu einem regulären Ausdruck umgewandelt und unter der Regel *shapeVariable* benutzt. Die Regel *arguments* entstammt dem Trait *CommonParserMethods*, welches jedem Parsr zur Verfügung stehet. So kann *shapeAttribute* Tupel von Attribut & Wert Paaren bilden. Shapeattributes sind lediglich primitive Datentypen und daher nicht interessant weiterzuverfolgen. In Zeile 6 wird ebenfalls n mal die Regel *geoModel* angewandt, welche die einzelnen Regeln einer geometrischen Figur abbildet und ein

Objekt zurückliefert, das als Container(4.3.3) fungiert:

```
private def geoModel: Parser[GeoModel] =
    geoIdentifier ~
    ((("style" ~> ident)?) <~ "{") ~
    rep(geoAttribute|anonymousStyle) ~
    (rep(geoModel) <~ "}") ^^
    {
      case name ~ style ~ attr ~ children =>
        GeoModel(name, {if(style.isDefined) Some(style.
           get) else None }, attr, children, [...])
    }
```

Hierbei werden nicht direkt Instanzen der eigentlich gewünschten *GeometricModel* Klasse erzeugt, da es für die geometrischen Figuren zwingend erforderlich ist einerseits ihre entsprechende Eltern Shape Instanz zu kennen. Diese existiert zu diesem Zeitpunkt jedoch noch nicht. Außerdem müssen sie ihre Eltern *GeometricModel* Instanzen kennen, welche ebenfalls erst erzeugt werden können, wenn ihre Kinder existieren. Alle Felder sollen ja konstanten sein und nicht nachträglich gesetzt werden. Diese Teufelsspirale wird gelöst, indem zunächst Container Objekte erstellt werden, die die Informationen speichern und umgewandelt werden, sobald ihre Abhängigkeiten geklärt sind. Sind alle Regeln überprüft worden und keine Fehleingaben oder fehlende Eingaben identifiziert worden, gibt es von nun an zwei Möglichkeiten weiter zu verfahren. Einer der Gründe, warum Container erstellt werden müssen wird in 4.3.2 erklärt. Wird der Container jetzt bereits der Fabrik für Shapes übergeben und somit eine fertige Shape Instanz erzeugt, können der Shape Instanz nachträglich **keine** weiteren Informationen mehr gegeben werden. Wird die Shape bereits jetzt erzeugt, existiert sie zwar als valide Shape, ist aber nun mehr als abstrakte Shape zu betrachten, da neue Shapes sie zwar erweitern können, sie aber nicht selber in einer Diagram Instanz referenziert wird, denn das Diagramm würde versuchen nachträglich Styleinformationen in die Shape einzuarbeiten. Hierzu mehr in 4.3.3. Um die Bearbeitungsschritte einer Shape zu verdeutlichen reicht es hier, den Container, welcher die Information der angehenden Shape enthält, nun direkt an die Shape Fabrik (das *Companion Object* der Shape Klasse) zu übergeben. Wie man in Zeile 11 sieht, wird beim Parsen einer Shape der Container ShapeSketch zurückgeliefert. Um entweder zu Testzwecken, oder um eine abstrakte Shape zu erzeugen kann ein String mit Shapedefinition deshalb über die Methode *abstractShape* geparsed werden:

```
private def abstractShape:Parser[Shape] = shapeSketch ^^
   {case sketch => sketch.toShape(None)}
```

Wie man sieht greift abstractShape auf die Regel/Methode *shapeSketch* zurück, erzeugt jedoch anschließend direkt über *ShapeSketch*'s Methode *toShape*, welche lediglich ein Aufruf an die *Shapefactory* weiterleitet, eine valide Shape. Ab dem Aufruf *toShape* übernimmt der spezialisierte Shape Parser im *Compan-*

ion Object der Shape Klasse das Ruder und bereitet die erhaltenen Argumente so auf, dass diese an den eigentlichen Shapekonstruktor weitergegeben werden können. Dabei werden zunächst die Superklassen aufgelöst und erweitert (siehe 4.3). Anschließend werden über ein Pattern Matching die erhaltenen Attribut & Wert Tupel aufgelöst. Beispielsweise das *"size-min"* Attribut, das zuvor in der beispielhaften Shapedefinition gegeben ist:

```
attributes.foreach{
      case ("size-min", x) =>
        val opt = parse(width_height, x).get
        if(opt.isDefined){
          size_width_min = Some(opt.get._1)
          size_height_min = Some(opt.get._2)
        }
        ...
```

Auch hierbei wird an weitere Regeln delegiert (z.B. *width_height*). Herausgefilterte Werte werden vordefinierten Variablen zugewiesen, welche später an den eigentlichen Konstruktor übergeben werden. Sind so alle Attribute der Shape aufgelöst wird der private Konstruktor der Shape companion Klasse aufgerufen. Es ist darauf zu achten, dass alle Felder der erweiterbaren Modellklasse Konstanten sind (siehe 4.3). Dazu müssen dem Shapekonstruktor die "Skizzen" (die Container) der geometrischen Figuren übergeben werden. *GeometricModel(s)* sollen ebenfalls immutabel sein und müssen daher bereits bei ihrer Erzeugung ihre Eltern Shape kennen. Würden die geometrischen Figuren vor der Shape instanziierung erzeugt werden, müsste die Referenz auf die Eltern Shape nachgereicht und die entsprechende Referenz als *var* definiert werden. Da die Erzeugung der *GeometricModel(s)* im Konstruktor erfolgt, kann die umschließende Shape über *this* referenziert werden. Wir befinden uns also jetzt im privaten Konstruktor der Shape Modellklasse. Die wohl wichtigste Aufgabe des Shape Konstruktors ist es die geometrischen Figuren zu erzeugen. Hierbei muss der Konstruktor jedoch auch beachten, dass von einer möglichen Elternshape weitere geometrische Figuren geerbt werden können (siehe 4.3.1). Deshalb werden die geometrischen Figuren wie folgt aufgelöst:

```
val shapes = {
    val inherited_and_new_geometrics = parentShapes.
       getOrElse(List()) ::: parseGeometricModels(geos,
       style)
    if(inherited_and_new_geometrics nonEmpty)Some(
       inherited_and_new_geometrics) else None
  }
```

So werden die geometrischen Figuren der Eltern Shape und die neu geparsten in einer Liste zusammengefasst. Die hier benutzte Methode *parseGeometricModels*, iteriert durch die Container der geometrischen Figuren und erzeugt vollwertige *GeometricModel(s)*:

```
private def parseGeometricModels(geoModels:List[GeoModel
    ], parentStyle:Option[Style]) =
     geoModels.map{_.parse(None, parentStyle)}.
       foldLeft(List[GeometricModel]())((list, c:Option[
          GeometricModel])=>if(c.isDefined) c.get :: list
          else list)
```

Hier wird die *parse* Methode der Container aufgerufen, welche anhand eines Matchings entscheidet um welches geometrisches Model es sich explizit handelt (z.B. Ellipse oder Rechteck etc.). Anhand dieses Matchings wird dann der spezialisierte Parser der vielen verschiedenen Klassen aufgerufen, die eine geometrische Figur beschreiben. Im Falle der obigen Beispielshape wurde ein Rechteck definiert, welches eine Ellise beinhaltet. Ab hier geht es also in der *Rectangle Fabric* weiter (dem *Companion Object* der Rectangle Klasse). Hier werden wie in allen anderen Parsern die mitgegebenen Attribute über ein Matching aufgelöst, wobei hierfür wieder evtl. auf weitere spezialisiertere Parser zurückgegriffen wird. Zum Beispiel wird das Rectangle seine Attribute an die Parser für das *CommonLayout* und die *CompartmentInfo* weitergeben. Erst wenn deren Fabrikmethoden valide Ergebnisse zurück liefern wird das Rechteck erzeugt. Der private Konstruktor der *Rectanlge* Klasse bekommt wiederum den Container der im Beispiel beschriebenen Ellipse mit und so geht der Vorgang in der *Ellipse Fabric* von vorne los.

4.5 Cache

Bisher wurde bereits einige male erwähnt, dass sich Modellklassen untereinander referenzieren. Diese Referenzierungen können aber Aufgrund der Problematik der Instanziierungsreihenfolge nicht während des Parsingvorgangs erfolgen. Sonst würde auf Objekte verwiesen werden, die selber bereits erst in der Mache sind. Oder es geht schlicht um die Problematik, dass Styles und Shapes unabhängig voneinander definiert werden können, Shapes jedoch namentlich auf Styles verweisen. Selbst wenn es also nur darum geht Referenzen auf erstellte Styles zu behalten um sie nicht an den *Garbage Collector* zu verlieren braucht es einen Mechanismus, der sich entsprechende Referenzen behält und wiederum bereitstellt. Da sich Diagrams, Shapes und Styles untereinander referenzieren können sollen, muss der Parser also in der Lage sein Instanzen zu erzeugen und gleichzeitig auch zu verwalten. Hierbei wurde stets darauf geachtet, dass der SprayParser keine statischen Zustände hält. Ausgehend davon, dass der Parser zukünftig in einer Multi-Client-fähigen Umgebung eingesetzt wird, ist so garantiert, dass sich verschiedene Clients niemals die selben Ressourcen teilen und somit eventuell inkonsistente Daten produzieren. Schon eine gemeinsame Abhängigkeit eines Namensraums wäre nicht wünschenswert, da Modellklassen mit gleichen Namen sonst je nach implementierung entweder Exceptions auslösen würden oder sich gegenseitig überschreiben könnten. Wenn aber keine statischen Zustände gehalten werden kann jeder Client mit einer eigenen SrpayParser Instanz kommunizieren. Außerdem kann so auf komplizierte synchronisationsmechanismen verzichtet werden. Es wird beispielsweise zunächst

eine Shape geparsed, auf die ein Diagram verweisen soll. Jedoch kann das Diagram die Shape nicht referenzieren, wenn es die entsprechende Referenz gar nicht kennt. Um die Referenz auf die Shape also nicht zu verlieren, muss sie lokal gespeichert und über eine Eigenschaft erreichbar sein, welche dem Diagram (besser gesagt dem Benutzer) bekannt ist und möglichst eindeutig sein sollte. Hierfür bietet sich der Name der Shape an. Anstatt aber nun IDs in Form von Strings zu halten, sondern auf bestehende Objekte verweisen zu können, müssen die Instanzen der generierten Shapes usw. lokal in einer Map gesichert werden. Hierfür wurde eine Hilfsklasse *Cache* erstellt, die Container für alle relevanten Datentypen bereitstellt. So können fertig instanziierte Objekte zwischengespeichert und bei Bedarf über die entsprechende Map der Cache Instanz referenziert werden.

4.5.1 Implicits

Hier wird Gebrauch von einer sehr mächtigen Eigenschaft von Scala gemacht, den **implicits**. Wenn man sich beispielsweise im Shapekonstruktor befindet und auf ein parentShape-element zugreifen will, bräuchte man normalerweise einen Aufruf wie den folgenden:

```
val someShape:Shape =
someCacheInstance.ContainerOfType[T].get("identifier").
   data
```

Oder weniger abstrakt:

```
val someShape:Shape= cache.shapeHierarchy(parent_ID).data
```

Selbst Aufrufe, die wie im obigen Beispiel recht kurz ausfallen, zerstören die Lesbarkeit ohnehin schon komplexer Codezeilen. Scala erlaubt es *implicits* zu definieren, welche u.a. implizite Typumwandlungen nach angegebener Methode durchführen können. Wenn in einem sogenannten *package object* besagte implicits definiert werden, ist es sogar möglich das entsprechende package object in einer Quelldatei zu importieren und so die impliziten Definitionen überall in den gewünschten Scope zu bringen. Im Falle der häufig benutzten Cache Klasse, sehen Zugriffe auf den Cache nun wie folgt aus:

```
val parentShape:Shape = parent_ID
```

parent_ID ist tatsächlich immernoch ein String, wird aber vom Compiler als unpassend erkannt, daraufhin sucht er nach einer impliziten Methode, welche in der Lage ist, den String in den gewünschten Typ (hier Shape) umzuwandeln. Der Programmierer, kann seine Cache-Instanz also sogar ohne explizit auf sie zugreifen zu müssen - eben implizit - benutzen. So kann man sich auf die wesentlichen Aspekte der Problemstellung konzentrieren. In diesem Fall ist die manuelle Typisierung der *parentShape* Konstante allerdings zwingend, da der Compiler sonst eine valide Zuweisung eines Strings zu einer nicht typisierten Konstante feststellen würde und daraufhin den Typ String inferieren würde.

Die entsprechende Definition des implicits im package object *model.parser* sieht aus wie folgt:

```
implicit def IDtoShape(id:String)(implicit c:Cache):
   Shape =
c.shapeHierarchy(id).data
```

Die Umwandlung der mit implicit gekennzeichneten Funktion nimmt dem Programmierer nun den Zwischenschritt ab über die Cacheinstanz die richtige Collection ausfindig zu machen und den gewünschten String aufzulösen. Auffällig ist, dass gleich **zwei implicit**s vorkommen. Ersteres kennzeichnet die Methode für den Compiler, das zweite ist ein *implicit parameter*. Da der besagte Cache pro Parserinstanz erzeugt wird um statische Bindungen prinzipiell vermeiden zu können, kann in der Methodendefinition im Grunde auf keine Cache-Instanz zugegriffen werden. Das liegt daran, dass der Compiler ja nicht implizit weiß, in welcher Cache-Instanz er nach der ID suchen soll. Genau das, wird dem Compiler durch das zweite implicit mitgeteilt. Man sagt ihm quasi: '*Ich kennzeichne eine Cache-Instanz, nach der sollst du suchen*'. Diese zwei einfachen Tricks, verkürzen die Verschachtelung und steigern die Lesbarkeit des Codes. Zwar kann der Code für jemanden, der nicht weiß was implicits sind durchaus verwirrend aussehen, da wir der Shape variable eindeutig einen String zuweisen. Doch der Code selbst, liest sich beinahe wie pseudo Code und der eigentliche Sinn erschließt sich dadurch besser. Entgegen der *dispatch* Kennzeichnung in Xtext, handelt es sich hierbei **nicht** um eine Art dynamische Erweiterung einer Klasse, sondern um eine implizite/automatisierte Typumwandlung. Anstatt also Methoden für einen Typ zu definieren, den dieser eigentlich garnicht besitzt, nur um ein Argument in der entsprechenden Parameterliste zu sparen, bleiben die Klassen hier konsistent. Im package object *parser* sind so für alle collections, die der Cache benutzt implizite Typumwandlungen bereitgestellt. So bleibt der Cache bis auf wenige Ausnahmen vollkommen unsichtbar für den Entwickler (wenn gewünscht). Als weiteren Vorteil der implicits könnte die Entkopplung genannt werden. Wird beispielsweise entschieden, die aktuelle Style Modellklasse mit einer neuen Klassendefintion zu erweitern, so muss nur die implizite Umwandlung abgeändert werden und die Polymorphie erledigt den Rest. Da eine Erweiterung einer Klasse auch neue Eigenschaften und Methoden mit sich bringt fragt sich natürlich nun, was es bringt Instanzen der neuen Klasse auf Variablen zu initialisieren, deren Typ noch die Oberklasse hat. Immerhin können die neuen Methoden so nicht auf die neuen Instanzen angewandt werden, da sie ja nur als Superklassenelement angesprochen werden können. Allerdings kann hier wieder eine neue implizite Umwandlung abhelfen, die entweder ein implizites Casting oder ein erneutes Einleiten einer Suche nach dem passenden Element in einer Map vollzieht. Man sieht, dass so relativ leicht komplett neue Objektstrukturen auf unveränderte Codepassagen anwenden lassen können. Hier muss man jedoch vorsichtig sein. Denn wie schon Martin Odersky sagt, sind *implicits* zwar unwahrscheinlich mächtig, „sind die Implicit Geister jedoch einmal beschworen, sind sie schwer wieder los zu werden“. In diesem Fall müsste man also vorsichtig überlegen, ob diese Vorgehensweise der späteren Objektstruktur

hilft oder sie eher undurchsichtig macht.

Kapitel 5

Fazit

5.1 Probleme

Die Arbeit war sehr Zeitaufwendig, da die bestehende Objektstruktur, die durch die Xtext grammatik beschrieben wurde auf Scala bezogen gelegentlich keinen oder wenig Sinn machte. Hin und wieder wurden auch Fehler in der definierten DSL entdeckt. Da das Umsetzen einer DSL in Scala ausgehend von einer Xtext Grammatik voraussetzt, dass entsprechende Grammatik verstanden worden ist. Dauerte es außerdem eine Weile, bis alle Unklarheiten bezüglich der Grammatik beseitigt wurden. Hin und wieder machten sich Fehler erst lange Zeit später bemerkbar, so mussten ein paar male bestehende Klassen komplett neu überarbeitet werden. Auch Xtext Sprachkenntnisse waren nötig, da z.B. die Generatoren der Xtext Sprache sonst unklar erscheinen. Methodenaufrufe wie createColorValue, die wie in 4.1.2 beschrieben worden sind,

```
s.layout.highlighting.unallowed.createColorValue
```

sind ähnlich den implicits in Scala für Neulinge absolut unverständlich. Auch bis die Xtext Quelldateien richtig interpretiert werden konnten, verging einige Zeit. Da das benötigte Metamodell zur Zeit der Entwicklung noch nicht einsatzbereit war, musste sich an einigen Stellen mit Mockups geholfen werden. Da bestehende Schnittstellen, des MoDiGens nicht verletzt werden durften, musste die Objektstruktur möglichst derer entsprechen, die durch die Xtext Grammatik eingeführt wurde. Dies ist dahingehend problematisch, dass so auch unsaubere Beziehungen, wie die ColorOrGradient, oder ColorWithTransparency übernommen werden mussten **??**specialstyle).

5.1.1 Scalaspezifische Probleme

Scalaspezifische Probleme gab es tatsächlich nicht wirklich. Scala ist einfach zu erlernen, bietet komfortable Collections, sowie eine ausgezeichnete und einheitliche Collections API. Außerdem bietet Scala Features wie *case classes* und *Pattern Matching*, um nur ein paar zu nennen. Diese Features sind schnell verstanden und umsetzbar. Einzig und allein die Tatsache, dass hier ein Scalaneuling am Werk war, bremste die Arbeit erheblich aus. So wurde zunächst mit

langen und sehr komplexen regulären Ausdrücken versucht, beliebige Shapedefinitionen zu parsen, was aber aufgrund der Rekursion der in sich geschachtelten geometrischen Figuren unmöglich ist. Ein Scalakenner, hätte sofort zu den *Parser Combinator* klassen gegriffen, diese waren dem Author zunächst aber noch nicht bekannt. Scala bietet viele Lösungen für viele Probleme, jedoch ist es insbesondere als Scalaneuling zunächst ein Labyrinth aus Möglichkeiten und oft wird erst später bemerkt, dass eine alternative Lösung besser passt.

5.2 Schlussfolgerung

5.2.1 Was wurde erreicht

5.2.2 Was wurde nicht erreicht

5.2.3 Fazit

Literaturverzeichnis

[1] Jorn Bettin. Mda journal. *bptrends*, 2004.

[2] Oliver Braun. *SCALA Objektfunktionale Programmierung*. Carl Hanser Verlag München, 2011.

[3] Eclipse Modeling Framework (EMF). Eclipse modeling framework. `https://eclipse.org/modeling/emf/`. Accessed: 2016-02-06.

[4] Friedrich Esser. *Scala für Umsteiger*. Oldenburg Verlag München, 2011.

[5] Stephen Ferg. Notes on how parsers and compilers work. `http://parsingintro.sourceforge.net/"`. Accessed: 2016-02-06.

[6] Markus Gerhart and Prof Doc Marko Boger. Modigen-concept of a set of textual dsl to define graphical dsls based on a meta-model.

[7] goodbyexml. Goodbye xml - befreiungsakt mit xtext. `https://jaxenter.de/goodbye-xml-befreiungsakt-mit-xtext-2-7369`. Accessed: 2016-01-22.

[8] Xtext Homepage. Language engineering for everyone. `https://eclipse.org/Xtext/`. Accessed: 2016-01-31.