Konzeption und Implementierung eines generischen Alphabet-Editors für visuelle Sprachen

Manuel Schwarze

Diplomarbeit

am Fachbereich Informatik der Technischen Universität Berlin,
betreut von Prof. Dr. Hartmut Ehrig und Dr. Gabriele Taentzer

18. März 1999

Zusammenfassung

In dieser Arbeit wird die Entwicklung eines constraint-basierten Alphabet-Editors für visuelle Sprachen beschrieben. Visuelle Sprachen werden hauptsächlich in Form von Modellierungstechniken in Prozessen der Software-Modellierung eingesetzt. Die Hauptbestandteile eines visuellen Alphabets bilden Symbole, die durch eine Kombination verschiedener primitiver Grafiken erstellt werden. Durch die Erweiterung des Alphabets um Datentypen können neben statischen auch dynamische Daten ausgedrückt werden. Außerdem werden einstellige Operationen zwischen Symbolen und Datentypen definiert, um zwischen ihnen grafische Verbindungen mit Hilfe von Constraints festzulegen.

Als Werkzeug für das Lösen von grafischen Constraints wird der externe Constraint-Solver ParCon benutzt, der an der Universität-Gesamthochschule Paderborn entwickelt wurde. Um einfacher mit den von ihm zur Verfügung gestellten Constraint-Formen arbeiten zu können, wird eine objektorientierte High-Level-Constraint-Sprache beschrieben.

Neben einem Alphabet enthält eine visuelle Sprache eine Grammatik, deren Entwicklung jedoch nicht Inhalt dieser Arbeit ist. Die Konzeption und Implementierung eines Grammatik-Editors sowie eines syntaxgesteuerten Grafik-Editors zum Zeichnen von Diagrammen sind Thema einer weiteren Diplomarbeit. Um dennoch die Bestandteile des Alphabets, insbesondere die grafischen Verbindungen zwischen Symbolen bzw. Datentypen anzuwenden, wird ein constraint-basiertes Test-System entwickelt. Damit können Diagramme erzeugt werden, die zwar auf grafischen Constraints, jedoch auf keiner Grammatik aufbauen.

An dieser Stelle möchte ich all denen danken, die mich in den letzten sieben Monaten geistig, seelisch, moralisch, kritisch, löblich, grammatisch, rhetorisch, genüßlich, spielerisch und freundlich unterstützt haben. Da wären besonders zu erwähnen ...

Rosi Bardohl ...	für hilfreiche Diskussionen, Kommentare, LATEX-Tips, ihre Geduld und ihr mutmachendes Lob
Magnus Niemann ...	für gute Diskussionen, Ideen und sein geduldiges Warten auf den Alphabet-Editor
Bernhard Bardohl ...	für sein Wissen über Skripte, Java, PARCON, Solaris und Linux sowie die Fähigkeit, es an mich weiterzugeben
Thorsten Shultzke ...	für die gute Vorarbeit im GENGED-Projekt und die dadurch entstandenen Ideen
Gabi Taentzer ...	für die ermunternden Worte zu meinem Vortrag nach schlafloser Nacht
Michael Rudolf ...	für seine Diplomarbeit, die mir als Richtlinie für Stil und Layout diente
Mario Schuster ...	für die großartige Zusammenarbeit im GENGED-Projekt, ohne die ich wahrscheinlich diese Diplomarbeit nicht angenommen hätte
Jens Hünerberg ...	für einen anspornenden „Wettkampf“
meine Eltern ...	für die Entlastung in der letzten Zeit und für unermüdliches Probelesen meiner vielen Worte

---

Kolosser 3,17

Die selbständige und eigenhändige Anfertigung
dieser Diplomarbeit versichere ich an Eides statt.

Berlin, den 18. März 1999

Unterschrift

Inhaltsverzeichnis

1 Einführung **1**
1.1 Visuelle Sprachen . . . 1
1.2 Ziele dieser Arbeit . . . 2
1.3 Gliederung . . . 5

2 Anforderungen und Konzepte **7**
2.1 Alphabet . . . 7
2.1.1 Symbole . . . 7
2.1.2 Veränderliche Daten . . . 9
2.1.3 Constraints . . . 10
2.1.4 Grafische Verbindungen . . . 11
2.1.5 Instanziierung von Objekten . . . 12
2.1.6 Zusammenfassung der Konzepte . . . 12
2.1.7 Beispiel: Nassi-Shneiderman-Diagramme . . . 13
2.2 Alphabet-Editor . . . 21

3 Werkzeuge und Hilfsmittel **24**
3.1 Unified Modeling Language . . . 24
3.1.1 Klassendiagramme . . . 24
3.1.2 Sequenzdiagramme . . . 27
3.2 Design Pattern . . . 28
3.2.1 Observer . . . 29
3.2.2 Listener . . . 30
3.2.3 Whole-Part . . . 32
3.2.4 Command-Processor . . . 32
3.2.5 Model-View-Controller . . . 35
3.2.6 Singleton-Pattern . . . 36
3.3 Programmiersprache Java . . . 37
3.3.1 Vorteile . . . 37
3.3.2 Nachteile . . . 38
3.3.3 Ausblick . . . 39
3.4 Constraint-Solver PARCON . . . 39
3.4.1 Variablen . . . 39
3.4.2 Formeln . . . 40
3.4.3 Verarbeitung des Constraint-Systems . . . 41
3.4.4 Kommunikation . . . 42

4 Systemstruktur **45**
4.1 Basis und Komponenten 45
4.2 Packages 46
4.3 Editoren-Management 49

5 Datenstrukturen eines Alphabets **53**
5.1 Überblick über verschiedene Bestandteile 53
5.1.1 Grafische Objekte 53
5.1.2 Verbindungen 53
5.2 Templates und Instanzen 54
5.3 Primitive Grafiken 56
5.3.1 Allgemeine Funktionalitäten 58
5.3.2 Bounding-Box und Koordinatenberechnungen 59
5.3.3 Transformationen 61
5.3.4 Knotenpunkte 63
5.3.5 Verbindungspunkte 65
5.3.6 Darstellungs- und Bearbeitungsmodi 68
5.3.7 Bearbeitungsfähigkeiten und -rechte 70
5.3.8 Darstellung graphischer Objekte 70
5.4 Gruppen von Primitiven 73
5.4.1 Verwaltung der Gruppenmitglieder 74
5.4.2 Zeichenreihenfolge 74
5.4.3 Bearbeitungsfähigkeiten und -rechte 75
5.5 Alphabet-Objekte 76
5.5.1 Symbole 76
5.5.2 Datentypen 76
5.6 Operationen 81
5.6.1 Interne und externe Operationen 81
5.6.2 Source und Target 81
5.6.3 Temporäre Instanzen 82
5.6.4 Verwaltung von Constraints 82
5.7 Constraints 84
5.8 Alphabet als Datenbasis 88
5.8.1 Funktionalitäten 88
5.8.2 Benachrichtigung bei Änderungen 89
5.8.3 Speichern und Laden 94

6 Allgemeine Mechanismen **96**
6.1 Internationalisierung 96
6.2 Zeichenfläche 98
6.2.1 Verwaltung der Grafiken 100
6.2.2 Zeichen-Ebenen 101
6.2.3 Arbeitsmodi 102
6.3 Strukturansicht 104
6.3.1 Modelle 104
6.3.2 Einträge der Strukturansicht 104
6.4 Benutzeraktionen 106

6.4.1 Eigenschaften von Benutzeraktionen 106
6.4.2 Arbeiten mit Benutzeraktionen 108
6.4.3 Definitionsdateien 110
6.5 Undo-Funktionalität 112
6.5.1 Befehlsobjekte und Befehlsgruppen 112
6.5.2 Undo-Manager 113

7 Constraint-Datenbank 116
7.1 High-Level-Constraint-Sprache 116
7.1.1 Definitionsdateien 116
7.1.2 Sprachelemente 117
7.1.3 Deklaration von Typen 117
7.1.4 Definition von Constraints 118
7.2 Auswertung der Definitionsdateien 121
7.2.1 Parsieren einer Typ-Deklaration 122
7.2.2 Parsieren einer Constraint-Definition 122
7.2.3 Datenstrukturen zur Typisierung 123
7.2.4 Datenstrukturen für Constraints 124
7.2.5 Verwaltung von Typen und Constraints 126
7.3 Anbindung an PARCON 128
7.3.1 Objekte im Constraint-Solver 129
7.3.2 Constraints im Constraint-Solver 131
7.3.3 Verbindung zu PARCON 131

8 Symbol- und Verbindungs-Editor 135
8.1 Symbol-Editor 135
8.1.1 Benutzeraktionen 135
8.1.2 Strukturansicht 137
8.1.3 Zeichenfläche 137
8.1.4 Arbeitsmodi 140
8.1.5 Definition der Bestandteile des Alphabets 141
8.1.6 Verwendung des Constraint-Solvers 143
8.1.7 Beispiel: Nassi-Shneiderman-Diagramme 143
8.2 Verbindungs-Editor 146
8.2.1 Benutzeraktionen 148
8.2.2 Strukturansicht 148
8.2.3 Zeichenfläche 148
8.2.4 Arbeitsmodi 150
8.2.5 Externe Operationen 150
8.2.6 Verwendung des Constraint-Solvers 151
8.2.7 Beispiel: Nassi-Shneiderman-Diagramme 151

9 Test-System 156
9.1 Modell des Diagramms 156
9.2 Struktur des Test-Systems 158
9.3 Ansichten zu Alphabet und Diagramm 158
9.4 Kontrollstrukturen 158

9.4.1 Steuerung von Aktionen 158
9.4.2 Erstellen der Bestandteile des Diagramms 161
9.4.3 Verwendung des Constraint-Solvers 161
9.4.4 Laden und Speichern eines Diagramms 161
9.5 Beispiel: Nassi-Shneiderman-Diagramme 162

10 Zusammenfassung und Ausblick **163**
10.1 Zusammenfassung 163
10.2 Erweiterungen 164
10.3 Verbesserungen 165

Literaturverzeichnis **168**

Abbildungsverzeichnis **172**

Tabellenverzeichnis **174**

Benutzerhandbuch **176**

Kapitel 1

Einführung

„Zeichnen ist Sprache für die Augen, Sprache ist Malerei für das Ohr.“ (Joseph Joubert)

Der Begriff der „Sprache“ ist ein Begriff, über den man im allgemeinen nicht nachdenkt. Sprache ist etwas Alltägliches. Sie existiert, sie wird benutzt – keiner könnte sich ein Leben ohne sie vorstellen. Sie dient der Kommunikation untereinander. Ohne sie würde es keinerlei Kulturen auf diesem Planeten geben. Wie schrecklich ist es für ältere Mitmenschen, wenn sie feststellen müssen, daß die Hörqualität der Ohren nachläßt. Wir brauchen die Sprache, um einander zu verstehen, um Meinungen auszutauschen. Mitunter können wir natürlich auch aneinander vorbeireden ... Joubert[1] jedoch dachte bei der *„Malerei für das Ohr“* wohl eher an ein Gedicht, dem er lauschen konnte.

Die vorliegende Arbeit zielt jedoch auf eine andere Art von Sprache ab, die mit dem ersten Teil des Zitats zusammenhängt – *„Zeichnen ist Sprache für die Augen“*. Nicht immer ist es sinnvoll, Zusammenhänge durch Worte zu erklären. Obwohl Joubert wahrscheinlich Gemälde im Sinn hatte, als er diesen Ausspruch von sich gab, möchten wir dieses Zitat ein wenig für unsere Zwecke mißbrauchen.

Die Verdeutlichung eines Sachverhalts benötigt oft viel weniger Zeit, wenn ein Blatt Papier und ein Bleistift zur Hand sind. Zusammenhänge werden durch Benutzung verschiedener Symbole, einfacher Kästchen, durch Linien, Pfeile und ein paar geschriebenen Worten skizziert. Meist erfolgt das Zeichnen intuitiv. Schon vor vielen Jahrzehnten haben Wissenschaftler erkannt, wie effektiv solche Zeichnungen sind. Mit den Jahren wurden verschiedenste Techniken entwickelt, die „Zeichnungen“ systematisieren. Der Begriff der „Visuellen Sprache“ wurde geprägt.

1.1 Visuelle Sprachen

Visuelle Sprachen [VLXX] haben tatsächlich alle Merkmale, die auch eine menschliche Sprache besitzt. Sie besitzen sowohl ein Alphabet als auch eine Grammatik. Diese Unterteilung wurde bereits von Costagliola et.al. eingeführt [CODL95]. Das Alphabet legt fest, welche Sprachmittel zur Verfügung stehen. Die Grammatik definiert dagegen, wie

[1] Joseph Joubert (1734-1824) – Französischer Essayist und Moralist

diese einzelnen Bestandteile miteinander zusammenspielen, um sinnvolle „Sätze“ zu erzeugen. Jeder wird einsehen, daß ein Alphabet ohne Grammatik eigentlich wenig Sinn macht. In solchem Fall könnten beliebige Zusammenstellungen von Sätzen gebildet werden. Was wäre das für ein Chaos!

Als die Software-Entwicklung noch in den Kinderschuhen steckte, wurden bereits visuelle Sprachen zur Modellierung von Programmen benutzt. Mit Struktogrammen[2], die heute in der Informatik nur noch auf der Programmierebene angewendet werden, modellierten die damaligen Entwickler ihre Systeme. Grady Booch startete Mitte der achtziger Jahre den Versuch, objektorienientierte Denkweisen zum Entwurf von Software salonfähig zu machen. Mit seiner Veröffentlichung „Objektorientierte Entwicklung mit Anwendungen“ [Boo86], die noch stark von der Programmiersprache Ada geprägt war, erregte er erstmals größeres Aufsehen. Erst zu Beginn der neunziger Jahre begannen sich jedoch objektorienierte Programmiersprachen durchzusetzen. Ursache dafür waren die immer größer werdenden Software-Systeme. Unter Objektorientierung verstehen wir den Zusammenschluß mehrerer diskreter Objekte zu der Gesamtheit einer Software. Diese Objekte besitzen verschiedene beschreibende Eigenschaften sowie von diesen Eigenschaften abhängende Verhaltensweisen [RBP+91].

Zur Analyse und zum Entwurf von Softwaresystemen wurde nun verstärkt an verschiedenen Modellierungstechniken und -methoden gearbeitet. Jede dieser Methoden stellt letztlich eine visuelle Sprache dar. Durch Anwendung solcher visueller Sprachen kann der Modellierer Diagramme zeichnen, die verschiedene Sichtweisen auf ein Software-System eröffnen.

Neben vielen eher unbedeutenden Methoden zur objektorientierten Modellierung existieren drei Techniken bzw. Methoden, die bis vor kurzem führend waren:

- BOOCH [Boo94]
- Object Modeling Technique (OMT) [RBP+91]
- Objectory Method [JCJv92]

Aus diesen drei Methoden entwickelte sich durch Zusammenarbeit von Grady Booch (BOOCH), James Rumbaugh (OMT) und Ivar Jacobsen (Objectory) die *Unified Modeling Language* (UML) [UML99], eine Modellierungstechnik, die auch innerhalb dieser Diplomarbeit zur Analyse und zum Entwurf der zugrundeliegenden Software benutzt wird.

1.2 Ziele dieser Arbeit

Wie schon angedeutet wurde, existieren viele verschiedene Modellierungstechniken, die jeweils auf einer oder mehrerer visuellen Sprachen beruhen. Obwohl es inzwischen die Unified Modeling Language gibt, ist absehbar, daß bestehende Techniken weiterentwickelt und damit verändert werden. Neue Modelle werden folgen.

[2] auch als Nassi-Shneiderman-Diagramme bekannt

Um Diagramme unter Nutzung einer visuellen Sprache zu erstellen, benötigt der Anwender eine spezielle Modellierungs-Software. Ohne ein solches Werkzeug wäre es nicht möglich, größere Projekte zu realisieren. Daher existiert in der Regel zu jeder einsatzfähigen Modellierungstechnik ein Programm zur Erstellung und Bearbeitung von Diagrammen. Es basiert auf einem fest implementierten Alphabet und der zur visuellen Sprache gehörenden Grammatik.

Ein normales Zeichenprogramm ist dagegen nicht in der Lage, die inhärenten Strukturen der Diagramme zu verwalten. Die innerhalb eines beliebigen Diagramms verwendeten Grafiken müssen in einer durch die visuelle Sprache definierten Weise zusammenhängen. Die Abbildung 1.1 zeigt ein Beispiel für ein sehr einfaches Diagramm, das Schema eines Netzplans. Es besteht ausschließlich aus Kreisen, die durch Pfeile verbunden sind. Benutzt der Anwender ein einfaches Zeichenprogramm wie z.B. CorelDraw, wird nach dem Verschieben eines Kreises die Linie in der Luft hängen. Sie ist nicht mehr mit dem Kreis verbunden. Eine Software dagegen, die speziell auf diese Art von Diagrammen zugeschnitten ist, bewegt sowohl den Kreis als auch die daran hängende Linie. Sie weiß um die inhärente Verbindung zwischen Kreis und Linie.

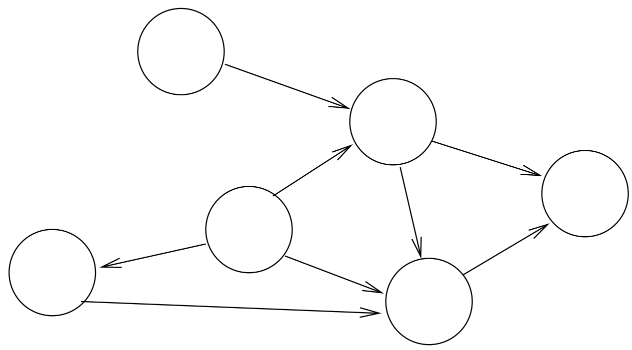


Abbildung 1.1: Einfache Diagrammstruktur – Ein Netzplan

Was heißt das ganz praktisch? Für *jede* Technik muß letztlich eine neue Anwendungssoftware implementiert oder zumindest ein bestehendes Programm angepaßt werden. Dieser Prozeß kostet neben viel Geld auch sehr viel Zeit – Zeit, die eigentlich keiner hat. Der Wettbewerb in der Marktwirtschaft ist hart: Die Firma, die zuerst ein funktionierendes Produkt zu einer neuen Modellierungstechnik auf den Markt bringt, sichert sich die ersten Marktanteile. Das Sprichwort *„Wer zu spät kommt, den bestraft das Leben.“* heißt in diesem Fall *„Wer zu spät kommt, den bestraft die Kundschaft.“*.

Im Rahmen des Projektes *„Generierung von Grafik-Editoren zur objektorientierten Informationsmodellierung“* haben sich Studenten der TU Berlin bereits im Wintersemester 1997/98 mit diesem Problem befaßt. Geleitet wurde das Projekt von Roswitha Bardohl, die sich zur Zeit in ihrer Dissertation ebenfalls mit diesem Thema auseinandersetzt. Die beteiligten Studenten hatten es sich zum Ziel gesetzt, ein System zu entwickeln, in das der Benutzer interaktiv eine visuelle Sprache eingeben kann, um später unter Nutzung dieser Sprache Diagramme zu zeichnen. Die Modellierungstechniken mit ihren grafischen Mitteln und entsprechenden Grammatiken werden hier nicht fest implementiert, sondern können von dem Benutzer definiert werden. Basierend auf dieser

Definition wird ein Grafik-Editor generiert, der im Gegensatz zu aktuellen Modellierungswerkzeugen flexibel einsetzbar ist. Auch mit zukünftigen visuellen Sprachen sollte er keine Probleme haben. Zur Beschreibung visueller Sprachen werden algebraische Spezifikationstechniken und Graphgrammatiken benutzt. Die ersten Ideen für dieses Projekt wurden in [BT97] beschrieben.

Das Resultat des Projekts war der Prototyp eines Systems, das GENGED[3] getauft wurde. Allerdings enthielt es noch zahlreiche Unzulänglichkeiten und Fehler. Während des Projekts traten einige Probleme zu Tage, die nicht alle befriedigend gelöst werden konnten. Aufbauend auf dem Projekt wurden deshalb zwei Diplomarbeiten ausgeschrieben, die eine Neukonzeption sowie eine Reimplementierung des Systems vorsahen. Die Erfahrungen des GENGED-Projekts werden in [Bar97] diskutiert.

Ziel der vorliegenden Diplomarbeit ist es, einen Alphabet-Editor zu entwerfen, mit dem der Benutzer das Alphabet einer visuellen Sprache definieren kann. Dieser Editor muß neben dem Zeichnen von Grafiken in der Lage sein, auch die inhärenten Strukturen zu verwalten. Diese Strukuren, die die Beziehungen verschiedener Grafiken beschreiben, können durch Graphen[4] dargestellt werden. Darüberhinaus kann der Benutzer spezielle Komponenten an Knoten oder Kanten als Attribute anhängen. In diesem Fall sprechen wir von attributierten Graphen. Beispielsweise könnten die Kreise in der Abbildung 1.1 verschiedene Zahlen enthalten. Diese Definitionen bestimmen das Alphabet einer visuellen Sprache. Desweiteren soll ein prototypischer Grafik-Editor erstellt werden, der die visuelle Manipulation von Sätzen über dem Alphabet ermöglicht. Da die Grammatik der visuellen Sprache nicht Inhalt dieser Diplomarbeit ist, kann der Benutzer uneingeschränkt mit dem Alphabet arbeiten. Aus Sicht der Grammatik kann er daher auch syntaktisch falsche Diagramme erstellen. Erst durch die Erweiterung des Systems um einen Grammatik-Editor, erhalten wir eine vollständige visuelle Sprache.

Berücksichtigt werden sollen in der vorliegenden Diplomarbeit die folgenden Punkte:

- Vereinfachung der Strukturen und vollständige Neuimplementierung des bereits in GENGED enthaltenen Alphabet-Editors
- Entwurf des gesamten Alphabet-Editors unter Verwendung von verschiedenen Design Patterns [GHJV95]
- Entwicklung einer High-Level-Constraint-Sprache zur Definition von Layout-Constraints[5]
- Vorhandene Grafiken innerhalb eines Alphabets sollen austauschbar sein
- Implementierung eines Undo-Mechanismus
- Berücksichtigung von Strukturen und Arbeitsweisen des nachfolgenden Grammatik-Editors durch Schnittstellen zu verschiedenen Datenstrukturen

[3] Abkürzung für „**Gen**erierung von **G**rafik-**Ed**itoren“

[4] Ein Graph besteht aus einer Menge von Elementen (Knoten), zwischen denen gewisse Beziehungen (Kanten) bestehen. Die Abbildung 1.1 zeigt – obwohl es sich um ein Beispieldiagramm handelt – auch einen Graphen. Die Kreise stehen für die Knoten und die Pfeile für die gerichtete Kanten des Graphen.

[5] Constraints (engl. Zwänge) beschreiben eine Beziehung zwischen verschiedenen Grafiken, die sich im Layout wiederspiegelt.

Eine weitere Diplomarbeit [Nie99] beschäftigt sich mit der Konzeption und Reimplementierung des zu GENGED gehörenden Grammatik-Editors. Mit diesem Editor wird es möglich sein, aufbauend auf dem Alphabet einer visuellen Sprache Grammatikregeln zu definieren. Erst diese Grammatik vollendet die Definition der visuellen Sprache und erlaubt es, korrekte Diagramme entsprechend der Grammatikregeln zu zeichnen.

1.3 Gliederung

Mit dem **ersten Kapitel** wurde eine kurze Einführung gegeben, die die Benutzung visueller Sprachen und den Einsatz eines Alphabet-Editors motiviert.

Im **zweiten Kapitel** wollen wir uns anschauen, welche Anforderungen an ein Alphabet gestellt werden. Wie ist es aufgebaut, welche Merkmale sind wichtig? Es werden verschiedene Konzepte vorgestellt, die diese Fragen beantworten. Abschließend veranschaulichen wir die Zusammenhänge anhand eines Beispiels. Nachdem die Bestandteile eines Alphabets herausgearbeitet wurden, analysieren wir die Anforderungen an einen Alphabet-Editor und beschreiben dessen grundlegenden Aufbau. Wir stellen fest, daß es sinnvoll ist, den Alphabet-Editor in mehrere einzelne Editoren aufzuteilen, die verschiedene Aufgaben zur Definition eines Alphabets übernehmen.

Das **dritte Kapitel** beschäftigt sich ausschließlich mit Werkzeugen und Hilfsmitteln, die in dieser Diplomarbeit zum Einsatz kommen. Zuerst stellen wir die Unified Modeling Language vor. Sie wird zur Analyse und zum Entwurf des Alphabet-Editors benutzt. Danach wird der Begriff *Design Pattern* beleuchtet und einige Beispiele, die bei der Konzeption des Alphabet-Editors angewendet wurden, vorgestellt. Nachdem wir kurz auf die Programmiersprache Java eingegangen sind, widmen wir uns einem externen Programm namens PARCON, mit dem der Alphabet-Editor zusammenarbeitet.

In dem **vierten Kapitel** wird die Grobstruktur des Alphabet-Editors beschrieben. Die verwendeten Packages des Software-Systems werden in groben Zügen erläutert. Dadurch wird ein Gesamtüberblick über den Alphabet-Editor vermittelt und bereits einige Zusammenhänge geklärt.

Die verschiedenen Editoren unter dem Dach des Alphabet-Editors arbeiten alle auf ein und denselben Datenstrukturen. Daher widmen wir uns im **fünften Kapitel** ausschließlich dem Entwurf und der Implementierung eines Alphabets.

Neben den Datenstrukturen benutzen die Editoren noch weitere grundlegende Mechanismen, auf die wir im **sechsten Kapitel** eingehen. Das betrifft beispielsweise verschiedene Bildschirmelemente. Außerdem werden der Umgang mit Benutzeraktionen und die Undo-Funktionalität erläutert.

Das **siebte Kapitel** geht auf einen weiteren allgemeinen Bestandteil des Alphabet-Editors ein – eine Constraint-Datenbank. Hier wird der Aufbau der entwickelten High-Level-Constraint-Sprache sowie der Entwurf und die Implementierung der Datenstrukturen und Zugriffsfunktionen beschrieben. Außerdem erläutern wir die Anbindung des Alphabet-Editors an den externen Constraint-Solver PARCON.

Im **achten Kapitel** beschäftigen wir uns ausgiebig mit dem Entwurf und der Implementierung des Symbol- und Verbindungs-Editors. Beide Editoren zusammen bilden die wichtigsten Bestandteile des Alphabet-Editors. Dieses Kapitel erläutert außerdem ausführlich den internen Aufbau des Alphabet-Modells. Abermals wird das Beispiel aus dem zweiten Kapitel aufgegriffen.

Das **neunte Kapitel** beschreibt die Datenstrukturen und Funktionalitäten des Test-Systems. Mit diesem System können Diagramme auf der Basis eines zuvor erstellten Alphabets gezeichnet werden. Bei diesem Test-System handelt es sich letztlich um eine Vereinfachung des schon erwähnten Grafik-Editors, der hier jedoch ohne Grammatikregeln arbeitet.

Wie es mit dem Alphabet-Editor weitergeht, stellen wir im **zehnten Kapitel** dar. Außerdem machen wir diverse Verbesserungsvorschläge und klären, wie vorhandene Funktionalitäten noch erweitert werden könnten.

Der **Anhang** enthält unter anderem ein Benutzerhandbuch, welches sehr ausführlich die Handhabung des Alphabet-Editors beschreibt. Dieses Handbuch wurde so konzipiert, daß es auch losgelöst von dieser Diplomarbeit verwendet werden kann. Es enthält daher jegliches Wissen, das ein Benutzer benötigt, um mit dem Alphabet-Editor arbeiten zu können. Aus diesem Grund wird innerhalb der eigentlichen Diplomarbeit einige Male auf den Anhang verwiesen, um ein gewisses Maß an Redundanz nicht zu überschreiten.

Kapitel 2

Anforderungen und Konzepte

Dieses Kapitel widmet sich der Analyse der Anforderungen an einen Alphabet-Editor. Zuvor diskutieren wir den Aufbau eines Alphabets für visuelle Sprachen. Dazu werden verschiedene Konzepte vorgestellt. An dem Beispiel der Nassi-Shneiderman-Diagramme wollen wir diese Konzepte weiter veranschaulichen.

2.1 Alphabet

Ein Alphabet besteht aus einer festen Anzahl von Bestandteilen. In der deutschen Sprache sprechen wir von Buchstaben, innerhalb einer visuellen Sprache von Symbolen. Sowohl Buchstaben als auch Symbole haben ein festes vorgegebenes Aussehen. Buchstaben können zu Sätzen zusammengefaßt werden, Symbole dagegen zu Diagrammen. Dabei werden Buchstaben von links nach rechts angeordnet, zweidimensionale Symbole auf verschiedene Weise. Diesen Prozeß können wir schon als Anwendung eines Alphabets verstehen. In den folgenden Abschnitten setzen wir uns intensiv mit den Bestandteilen des visuellen Alphabets auseinander. Wir werden feststellen, daß es nicht ausreicht, nur von Symbolen zu sprechen. Weitere Anforderungen verlangen weitere Bestandteile.

2.1.1 Symbole

Ein Symbol stellt zweifellos den wichtigsten Bestandteil des Alphabets dar. Es wird durch einen eindeutigen Namen und eine Symbolgrafik repräsentiert. Dadurch findet eine Typisierung der in einem Alphabet definierten Symbole statt.

Bestandteile: Die Symbolgrafik selbst besteht aus beliebig vielen primitiven Grafikobjekten. Dazu zählen wir beispielsweise Rechtecke, Kreise, Ellipsen, Linien, Pfeile, unveränderlichen Text, Polygone und Formen, die frei gezeichnet werden können. Das Zusammenspiel dieser Primitiven ist letztlich für das Aussehen des Symbols verantwortlich. Dabei ist es wichtig, daß jede der beteiligten Primitiven eine bestimmte Position und Größe innerhalb des gesamten Symbols einnimmt.

Bounding-Box: Um jede primitive Grafik können wir eine sogenannte Bounding-Box legen. Darunter verstehen wir das kleinstmögliche Rechteck, das die jeweilige Primitive vollständig umschließt. Durch Nutzung einer solchen Bounding-Box können wir für *jedes* Grafikobjekt auf die gleiche Art und Weise seine Position und Größe festlegen. Auch das Symbol selbst besitzt eine Bounding-Box. Sie wird jedoch bestimmt durch die Bounding-Boxen der enthaltenen primitiven Grafikobjekte. Die Abbildung 2.1 verdeutlicht diese Sachverhalte grafisch. In den eckigen Klammern ist als erstes Zahlentupel die Position und als zweites Tupel die Größe einer Primitive (Breite und Höhe) angegeben.

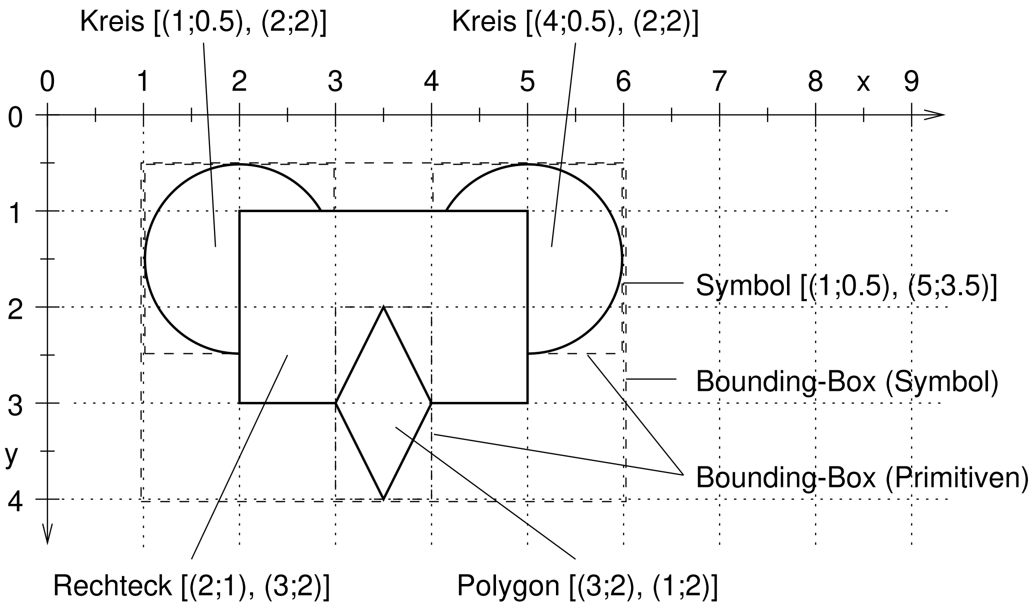


Abbildung 2.1: Beispielhafter Aufbau eines Symbols

Tiefenposition: Beim Betrachten dieser Abbildung fällt noch eine weitere wichtige Eigenschaft der Primitiven ins Auge. Die Kreise sind vom Rechteck verdeckt, das Rechteck dagegen vom Polygon. Für jede Primitive muß festgelegt werden, an welcher Tiefenposition sie gezeichnet wird. Würde das Rechteck zuerst gezeichnet, dann läge es im Hintergrund und würde sowohl von den Kreisen als auch von dem Polygon verdeckt. Die Zeichenreihenfolge der Primitiven des dargestellten Symbols lautet: Kreis, Kreis, Rechteck, Polygon.

Eigenschaften der Primitiven: Die primitiven Grafikobjekte sollten verschiedene Eigenschaften zugewiesen bekommen, die über Form, Position und Größe hinausgehen. Eine Primitive besitzt meist nur einen Teil der folgenden Eigenschaften:

- Umriß-/Linieneigenschaften:
  - Farbe oder Transparenz
  - Linienanfang und -ende für offene Formen
  - Liniendicke

- Fülleigenschaften:
  - Farbe oder Transparenz
  - Füllmuster
  - Farbverlauf
- Texteigenschaften:
  - Text- und Hintergrundfarbe
  - Schriftart
  - Schriftgröße
  - Schriftstil

Default-Layout: Durch die Art und Eigenschaften der Primitiven ergibt sich für jedes Symbol ein Default-Layout, das später bei der Anwendung in Diagrammen durch den Benutzer oder durch Constraints in einem gewissen Rahmen verändert werden kann.

2.1.2 Veränderliche Daten

Beim Zeichnen von Diagrammen reicht ein Alphabet von statischen, also unveränderlichen Symbolen[1], nicht aus. Es soll möglich sein, an jedes Symbol veränderliche Attribute anzuhängen. Dabei kann es sich beispielsweise um Zahlen, Namen von Objekten oder um Listen handeln. Innerhalb eines Symbols kann nur unveränderlicher Text als eine der vielen Primitiven eingesetzt werden.

Datentypen: Das Alphabet wird deshalb um variable Elemente erweitert, die wir im folgenden als *Datentypen* bezeichnen. Sie stellen die Attribute innerhalb eines Diagramms dar. Ein Datentyp besitzt wie ein Symbol einen eindeutigen Namen innerhalb des gesamten Alphabets. Außerdem wird durch einen bestimmten Typ festlegt, welche Art von Daten dieser Datentyp darstellt. Die Analyse verschiedener Modellierungstechniken hat gezeigt, daß oft folgende Daten benötigt werden:

- Zeichenketten, z.B. für Namen von Objekten
- Ganze Zahlen, z.B. für Kardinalitäten
- Gebrochene Zahlen, z.B. für Geldbeträge
- Listen von Zeichenketten, z.B. zur Auflistung von Eigenschaften

Default-Wert: Neben seinem Typ enthält ein Datentyp auch immer einen konkreten Wert. Dieser Wert stellt jedoch nur einen Default-Wert dar. Er muß verändert werden können, zum einen durch den Benutzer, zum anderen aber auch durch die Anwendung später definierter Grammatikregeln. Ein Datentyp besitzt wie ein Symbol eine Bounding-Box, die Aussagen zur Größe und Position macht. Die Größe dieses umschließenden Rechtecks hängt hier jedoch von dem zugewiesenen Wert sowie von speziellen Eigenschaften des Datentypen ab.

[1]Ein Symbol besitzt durch verschiedene Grafiken ein bestimmtes Aussehen, das durch die Anwendung von Constraints zwar beeinflußt, aber nicht grundlegend geändert werden kann.

Eigenschaften: Die Eigenschaften eines Datentypen sind typspezifisch. Eine Zeichenkette enthält beispielsweise die Eigenschaften Schriftart, Schriftgröße und Schriftstil. Ein Datentyp für gebrochene Zahlen könnte darüber hinaus noch die Anzahl der Nachkommastellen verwalten.

2.1.3 Constraints

Die Abbildung 2.1 des vorletzten Abschnitts zeigte einige Primitiven, die ein Symbol bilden. Man könnte den Eindruck gewinnen, Positionen und Größe der primitiven Grafikobjekte seien konstant. Dieser Eindruck ist falsch. Erst im nächsten Abschnitt beschäftigen wir uns mit Verbindungen, die zwischen zwei Symbolen definiert werden können. Geht ein Symbol eine Verbindung mit einem anderen Symbol ein, kann sich sein Layout verändern. Position und Größe einer Primitive sind nicht automatisch fest definiert.

Um einen Zusammenhang zwischen verschiedenen Primitiven herzustellen, müssen wir Constraints nutzen. Unter einem Constraint verstehen wir ein Gleichungs- oder Ungleichungssystem, das numerische Eigenschaften von Objekten in Beziehung setzen kann. Durch Lösen von Constraints können beispielsweise Position, Breite oder Höhe eingeschränkt werden. In der Abbildung 2.2 sind einige Beispiele für Constraints dargestellt.

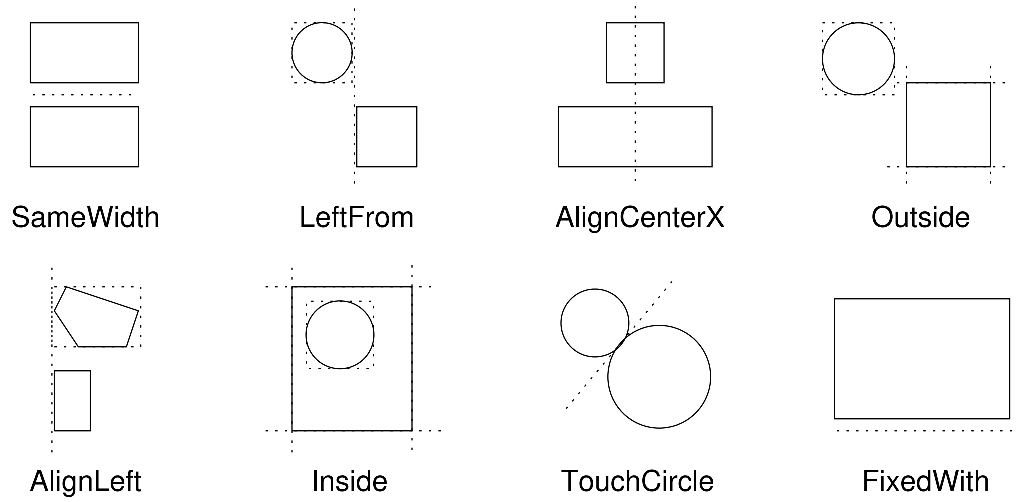


Abbildung 2.2: Beispiele für Constraints

Wir müssen dementsprechend unser Verständnis von einem Symbol erweitern. Es besitzt neben den beliebig vielen primitiven Grafiken außerdem beliebig viele Constraints, die zwischen diesen Grafiken definiert sind. Sie dienen dazu, die Primitiven zusammenzuhalten und einen Rahmen für das gewünschte Layout sicherzustellen.

2.1.4 Grafische Verbindungen

Beim Zeichnen eines Diagramms werden verschiedene Symbole und Datentypen eines Alphabets miteinander verbunden. So verbindet beispielsweise ein Symbol, das einen Pfeil darstellt, zwei Symbole miteinander, die Rechtecke darstellen. Über eine Grammatik wird später festgelegt, unter welchen Bedingungen diese grafischen Verbindungen auftreten dürfen. Dennoch definieren wir schon innerhalb des Alphabets alle möglichen Zusammenschlüsse von Symbolen bzw. Datentypen. Wir können das Bestehen einer solchen Verbindung durch eine einstellige *Operation* ausdrücken.

Beteiligte Objekte: An einer Verbindung sind immer genau zwei Objekte (Symbole oder Datentypen) beteiligt. Wir bezeichnen sie als *Source* und *Target*. Das Source-Objekt stellt den Parameter und das Target-Objekt den Rückgabewert der Operation dar. Weil das Source-Objekt eine Beziehung zu dem Target-Objekt aufbaut, muß das Target-Objekt bereits vorhanden sein. Eine zweite Folge der einstelligen Operation: Das Source-Objekt darf nur solange wie das Target-Objekt existieren. Die Grammatik, die später auf diesen Verbindungen aufbaut, fordert diese Regeln.

Layout: Gehen die für Source und Target festgelegten Symbole eine der definierten Verbindungen ein, drückt sich das (normalerweise) durch ein verändertes Layout aus. Diese Veränderung wird unter Anwendung von Constraints zwischen Source- und Target-Objekt erreicht. Oft wird ein Symbol an ein anderes „angekoppelt“, oder es wird ein Datentyp in ein Symbol integriert.

Verbindungspunkte: Um die Constraint-Definition zu vereinfachen, wird jede Primitive mit sogenannten Verbindungspunkten ausgestattet. Diese Punkte können bei der Definition von Constraints direkt verwendet werden. Es wird gewünscht, daß eine Anzahl von neun Verbindungspunkten bereits standardmäßig für jede Primitive sowie für das Symbol selbst vorhanden ist. Diese neun Punkte sollen – wie in der Abbildung 2.3 dargestellt – auf der Bounding-Box verteilt sein. Der Benutzer soll jedoch beliebig viele weitere Verbindungspunkte zu einer Primitiven hinzufügen und auch wieder entfernen können.

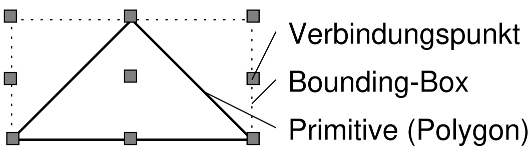


Abbildung 2.3: Standard-Verbindungspunkte einer Primitive

Platzhalter: Bei einigen Verbindungen wird es notwendig sein, Platzhalter als Source oder Target einzusetzen. Dabei handelt es sich um unsichtbare Symbole, die jedoch eine durch eine Bounding-Box definierte Position und Größe besitzen. Auch eine Primitive kann innerhalb eines Symbols als Platzhalter dienen. Auf einem Platzhalter sollen wie auf einem normalen Symbol Constraints definiert werden können.

2.1.5 Instanziierung von Objekten

Nachdem ein vollständiges Alphabet definiert wurde, kann der Benutzer mit den definierten Symbolen, Datentypen und Verbindungen Diagramme erstellen. Vorerst jedoch ohne Grammatik. Die Elemente des Alphabets dienen hierbei jedoch nur als Templates[2]. Sie müssen instanziiert werden. Durch diesen Prozeß wird eine originalgetreue Kopie von einem Symbol oder Datentyp geschaffen, die innerhalb des Diagramms eingesetzt werden kann. Auch grafische Verbindungen werden bei ihrer Anwendung instanziiert. Eine Verbindungsinstanz arbeitet demzufolge auch nicht mehr auf Templates, sondern auf den entsprechenden Instanzen. Von einem Template-Objekt – egal ob Symbol, Datentyp oder Verbindung – können beliebig viele Instanzen gebildet werden. Ändern sich die Template-Objekte des Alphabets, müssen jedoch auch die Instanzen geändert werden.

2.1.6 Zusammenfassung der Konzepte

In diesem Abschnitt wollen wir die wichtigsten der bisher diskutierten Anforderungen als Konzepte definieren, um einige von ihnen danach an einem ausführlicheren Beispiel zu demonstrieren.

Konzept 1 – Symbole: Ein grafisches Symbol besteht aus einem im Kontext eindeutigen Symbolnamen und einer Symbolgrafik. Diese Symbolgrafik besteht aus n primitiven Grafiken und m Constraints zwischen diesen Primitiven ($n, m \in \mathbb{N}$).

Konzept 2 – Datentypen: Ein Datentyp repräsentiert veränderliche Daten. Er besitzt einen eindeutigen Namen, einen bestimmten Typ und einen Default-Wert.

Konzept 3 – Bounding-Box: Jede Primitive, ein Symbol selbst und ein Datentyp besitzen eine Bounding-Box, die zur Positionierung und zur Bestimmung ihrer Größe dient.

Konzept 4 – Verbindungspunkte: Ein Verbindungspunkt kennzeichnet eine bestimmte Position auf einer Primitive, einem Symbol oder einem Datentyp. Diese Position wird in Abhängigkeit von der linken oberen Ecke der Bounding-Box relativ bezogen auf die Breite bzw. Höhe angegeben.

Konzept 5 – Grafische Verbindungen: Eine grafische Verbindung definiert durch eine einstellige Operation einen Zusammenhang zwischen zwei Alphabet-Objekten (Symbol oder Datentyp). Neben der Operation existieren k Constraints zwischen den beteiligten Symbolgrafiken bzw. Datentypen ($k \in \mathbb{N}$). In den folgenden Kapiteln sprechen wir von *Operationen*.

Konzept 6 – Templates und Instanzen: Bei der Definition des Alphabets werden Templates für Symbole, Datentypen und grafische Verbindungen erstellt. Diese können später bei der Benutzung des Alphabets beliebig oft instanziiert werden.

[2] engl. Schablone

Innerhalb eines Diagramms können ausschließlich Instanzen benutzt werden. Eine grafische Verbindung definiert in diesem Fall eine einstellige Operation auf der Instanz eines Symbols oder Datentyps.

Konzept 7 – Default-Layout: Jedes Symbol und jeder Datentyp eines Alphabets besitzt ein sogenanntes Default-Layout. Es entsteht durch die aktuellen Positionen und Ausmaße der Primitiven sowie durch die Anwendung aller Constraints, die innerhalb eines Symbols definiert sind. Bei der Instanziierung eines Alphabet-Objekts wird dieses Default-Layout vorerst übernommen. Es kann sich jedoch später verändern.

2.1.7 Beispiel: Nassi-Shneiderman-Diagramme

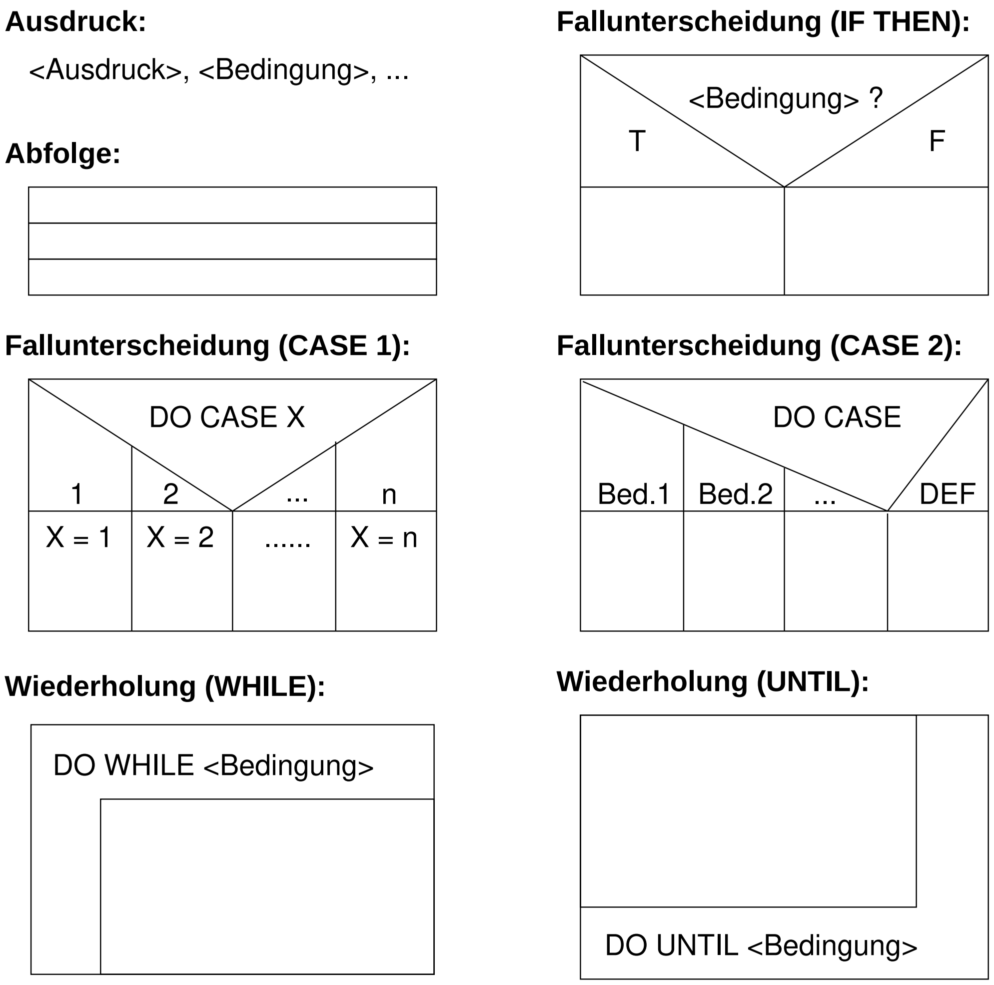


Abbildung 2.4: Nassi-Shneiderman-Diagramme: Bestandteile

An dem Beispiel der Struktogramme – sie werden auch als Nassi-Shneiderman-Diagramme bezeichnet – soll der Aufbau des Alphabets einer visuellen Sprache beschrieben werden. Nassi-Shneiderman-Diagramme dienen der Strukturierung eines sequentiellen Programms bzw. eines Programmteils und sind in [DIN79] standardisiert. Die Abbildung 2.4 gibt einen Überblick über die Bestandteile dieser Modellierungstechnik. Sie dürfen jedoch nicht mit einzelnen Symbolen verwechselt werden. Einige von ihnen bestehen bereits aus mehreren Symbolen.

Hauptbestandteile eines Programms sind *Ausdrücke*, die Zuweisungen von Werten an Variablen oder den Aufruf von Befehlen enthalten. *Bedingungen* können durch einen besonderen Ausdruck formuliert werden. In der Regel besteht ein Programm aus vielen Ausdrücken, die eine *Abfolge* bilden. Dabei wird jeweils ein Ausdruck in ein Rechteck eingerahmt und untereinander gesetzt. *Fallunterscheidungen* und dadurch entstehende Verzweigungen innerhalb des Programms sind ebenfalls typisch. Insgesamt existieren drei Arten von Fallunterscheidungen. Bei der ersten wird nur abgeprüft, ob eine bestimmte Bedingung erfüllt ist (IF THEN). Die zweite Art von Fallunterscheidung prüft eine beliebige Anzahl (>1) von Fällen, in denen Zahlen vom Typ Integer verglichen werden (CASE 1). Die letzte Art erweitert die Case-Anweisung des zweiten Falls um eine Default-Verzweigung (CASE 2). Außerdem können auf zweierlei Weise *Wiederholungen* modelliert werden. Es existiert eine WHILE- und eine UNTIL-Schleife, die jeweils von einer bestimmten Bedingung abhängen.

Zwischen den Primitiven der Symbole werden Constraints benutzt, die in der Tabelle 2.1 kurz beschrieben werden. Die Erklärung stellt keine Definition, sondern nur eine informelle Beschreibung dar.

Symbole

In den folgenden Abschnitten werden wir die Symbole zu Nassi-Shneiderman-Diagrammen erstellen, die aus Primitiven zusammengesetzt sind und durch Anwenden von Constraints zusammengehalten werden.

Programmteil: Ein Programmteil wird durch ein Rechteck dargestellt. Durch zwei Constraints stellen wir sicher, daß das Rechteck eine Mindesthöhe von 20 EH[3] und eine Mindestbreite von 50 EH hat. Das Symbol ist in der Abbildung 2.5 dargestellt.

Name: `Part`

Constraints: `MinWidth(Part.Rect1, 50)` `MinHeight(Part.Rect1, 20)`

Symbolgrafik: 1 Rechteck

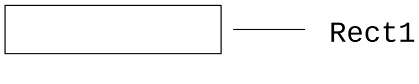


Abbildung 2.5: Nassi-Shneiderman-Diagramme: Programmteil (`Part`)

Abfolge: Eine Abfolge von Programmteilen stellt eigentlich bereits eine Verbindung dar. Trotzdem benötigen wir ein Symbol, das eine *leere Abfolge* definiert. Dabei handelt es sich um einen Platzhalter. Wir erstellen ein unsichtbares Rechteck, das

[3]EH = Einheiten

Constraint	Erklärung
`AnchorBottom`	Ein Objekt schließt unter einem anderen Objekt nahtlos an.
`AnchorRight`	Ein Objekt schließt rechts an einem anderen Objekt nahtlos an.
`AlignCenter`	Zwei Objekte sind zentriert zueinander ausgerichtet.
`AlignCenterX`	Zwei Objekte sind horizontal zentriert zueinander ausgerichtet.
`AlignLeft`	Zwei Objekte sind linksbündig zueinander ausgerichtet.
`AlignRight`	Zwei Objekte sind rechtsbündig zueinander ausgerichtet.
`DiffLocationX`	Positioniert ein Objekt in einer bestimmten Distanz zu einem anderen Objekt, die x-Koordinate betreffend.
`DiffSize`	Skaliert ein Objekt, so daß es sich um eine bestimmte Differenz von der Größe eines anderen Objekts unterscheidet.
`Inside`	Ein Objekt liegt innerhalb eines anderen Objekts.
`MinDiffSize`	Ein Objekt hat mindestens die Höhe eines anderen Objekts zu/abzüglich einer bestimmten Differenz.
`MinHeight`	Ein Objekt hat eine minimale Höhe.
`MinWidth`	Ein Objekt hat eine minimale Breite.
`RelLocation`	Die Position eines Objekts wird relativ zur Position und Größe eines anderen Objekts bestimmt.
`RelLocationY`	Die Y-Koordinate eines Objekts wird relativ zur Y-Koordinate und Größe eines anderen Objekts bestimmt.
`SameLocation`	Zwei Objekte haben dieselbe Position.
`SameSize`	Zwei Objekte haben dieselbe Breite und Höhe.
`SameWidth`	Zwei Objekte haben dieselbe Breite.
`SizePercent`	Die Größe eines Objekts hängt relativ von der Größe eines anderen Objekts ab.

Tabelle 2.1: Nassi-Shneiderman-Diagramme: Verwendete Constraints

eine minimale Breite und Höhe besitzt. Dieses Rechteck wird später Programmteile aufnehmen können. Es ist in der Abbildung 2.6 dargestellt.

Name: `Sequence`

Symbolgrafik: 1 Rechteck (unsichtbar)

Constraints: `MinWidth(Sequence.Rect1, 50)` `MinHeight(Sequence.Rect1, 20)`

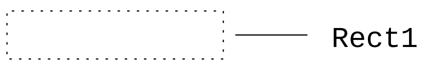
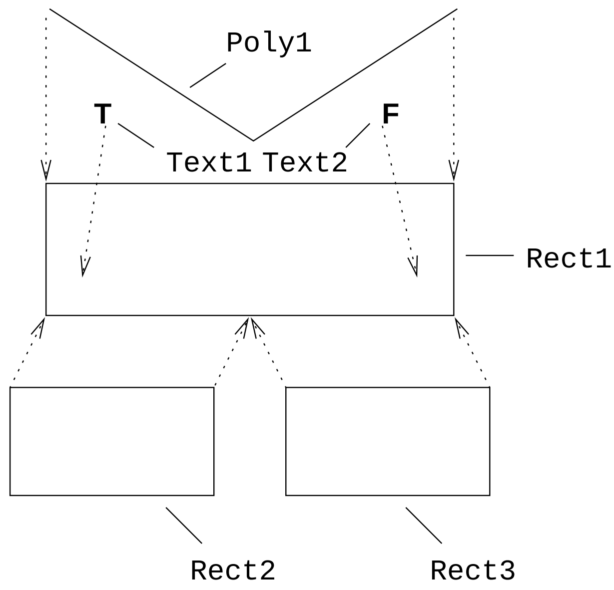


Abbildung 2.6: Nassi-Shneiderman-Diagramme: Leere Abfolge (`Sequence`)

Fallunterscheidung (IF THEN): Eine einfache Fallunterscheidung (IF THEN) können wir mit Hilfe von Rechtecken, einem Polygon und statischem Text erzeugen. Über Constraints wird sichergestellt, daß die Einzelteile des Symbols richtig angeordnet werden, wie Abbildung 2.7 zeigt.

Name: `Selection`

Symbolgrafik: 3 Rechtecke,
1 Polygon,
2 Statische Texte



Constraints:

```
MinWidth(Selection.Rect1, 100)
MinHeight(Selection.Rect1, 30)
SameLocation(Selection.Rect1,
  Selection.Poly1)
SameSize(Selection.Rect1,
  Selection.Poly1)
AlignLeft(Selection.Rect1,
  Selection.Rect2)
AlignRight(Selection.Rect1,
  Selection.Rect3)
AnchorBottom(Selection.Rect2,
  Selection.Rect1)
AnchorBottom(Selection.Rect3,
  Selection.Rect1)
WidthPercent(Selection.Rect2,
  Selection.Rect1, 0.5)
WidthPercent(Selection.Rect3,
  Selection.Rect1, 0.5)
RelLocation(Selection.Text1,
  Selection.Rect1, 0.15, 0.6)
RelLocation(Selection.Text2,
  Selection.Rect1, 0.83, 0.6)
SameHeight(Selection.Rect2,
  Selection.Rect3)
MinHeight(Selection.Rect2, 20)
```

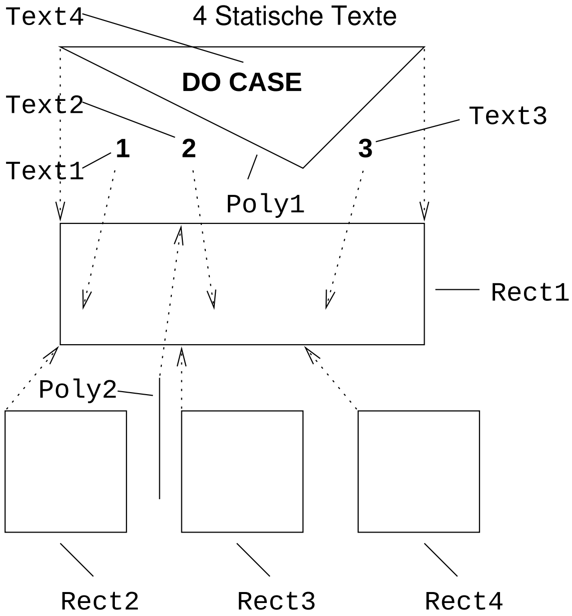
Abbildung 2.7: Nassi-Shneiderman-Diagramme: Fallunterscheidung (`Selection`)

Fallunterscheidung (CASE): Die anderen Fallunterscheidungen (CASE) sind etwas schwieriger zu erzeugen, da es keine feste Anzahl von Fällen gibt. Zur Vereinfachung entscheiden wir uns, für jede Anzahl ein separates Symbol zu erstellen. Wir greifen uns das Symbol für drei verschiedene Fälle heraus. Die Fallunterscheidung, in der als letzter Fall ein Default-Wert auftritt, wird hier vernachlässigt, da sie der hier beschriebenen sehr ähnlich ist. Um das Symbol weiter zu vereinfachen, definieren wir die unteren Rechtecke so, daß sie gleich groß sind. Die Abbildung 2.8 zeigt den Symbolaufbau.

Wiederholung: Offen bleibt noch die Schleifenmodellierung. Da die WHILE- und UNTIL-Schleife fast identisch sind, beschränken wir uns auf letztere. Der Aufbau ist in der Abbildung 2.9 dargestellt.

Name: `Case3`

Symbolgrafik: 4 Rechtecke,
2 Polygone,
4 Statische Texte



Bemerkung: Die Primitive Poly1 ist weiß gefüllt und muß vor Poly2 gezeichnet werden. Dadurch verdeckt sie teilweise Poly2.

Constraints:

```
MinWidth(Case3.Rect1, 100)
MinHeight(Case3.Rect1, 30)
SameLocation(Case3.Rect1,
  Case3.Poly1)
SameSize(Case3.Rect1,
  Case3.Poly1)
AlignLeft(Case3.Rect1,
  Case3.Rect2)
AnchorRight(Case3.Rect3,
  Case3.Rect2)
AnchorRight(Case3.Rect4,
  Case3.Rect3)
AnchorBottom([Case3.Rect2,
  Case3.Rect3, Case3.Rect4],
  Case3.Rect1)
SameHeight([Case3.Rect2,
  Case3.Rect3], Case3.Rect4)
SameWidth([Case3.Rect2,
  Case3.Rect3], Case3.Rect4)
RelLocation(Case3.Poly2,
  Case3.Rect1, 0.33, 0)
SameHeight(Case.Poly2,
  Case3.Rect1)
RelLocation(Case3.Text1,
  Case3.Rect1, 0.05, 0.8)
RelLocation(Case3.Text2,
  Case3.Rect1, 0.38, 0.8)
RelLocation(Case3.Text3,
  Case3.Rect1, 0.71, 0.8)
RelLocation(Case3.Text4,
  Case.Rect1, 0.4, 0.1)
MinHeight(Case3.Rect2, 20)
```

Abbildung 2.8: Nassi-Shneiderman-Diagramme: Fallunterscheidung (`Case3`)

Datentypen

Die Bestandteile der Nassi-Shneiderman-Diagramme, die in der Abbildung 2.4 aufgelistet sind, zeigen eigentlich nur einen Kandidaten für einen Datentyp – einen *Ausdruck*. Hinsichtlich der Verbindungen, die jedoch noch definiert werden müssen, erzeugen wir insgesamt zwei Datentypen (siehe Tabelle 2.2).

Name	**Typ**	**Default-Wert**	**Bedeutung**
`Expression`	String	<Ausdruck>	Verwendung in Programmteilen, Variable in einer Fallunterscheidung (CASE 1)
`Condition`	String	<Condition>	Verwendung in Fallunterscheidungen (IF THEN, CASE 2) und Wiederholungen

Tabelle 2.2: Nassi-Shneiderman-Diagramme: Datentypen (`Expression`, `Condition`)

Name: `DoUntil`

Symbolgrafik: 3 Rechtecke,
1 Polygon,
2 Statische Texte

Constraints:
```
MinWidth(DoUntil.Rect1, 100)
MinHeight(DoUntil.Rect1, 50)
SameLocation(DoUntil.Rect1,
  DoUntil.Rect2)
DiffSize(DoUntil.Rect2,
  DoUntil.Rect1, -20, -20)
AnchorBottom(DoUntil.Text1,
  DoUntil.Rect2)
DiffLocationX(DoUntil.Text1,
  DoUntil.Rect1, 10)
```

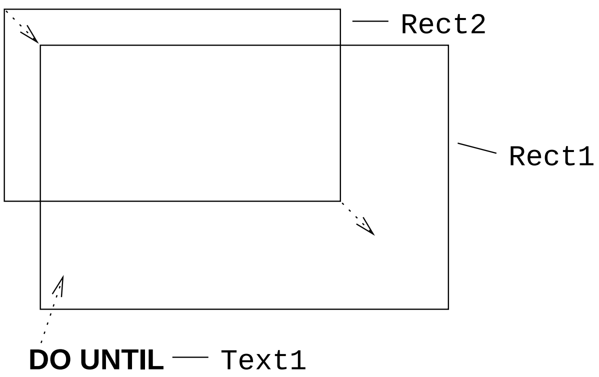


Abbildung 2.9: Nassi-Shneiderman-Diagramme: Wiederholung (UNTIL) (`DoUntil`)

Verbindungen

Die Verbindungen stellen Möglichkeiten von Beziehungen zwischen Objekten (Symbole oder Datentypen) dar. Erst durch die im Grammatik-Editor definierten Grammatikregeln werden die Umstände und Bedingungen des Auftretens dieser Verbindungen festgelegt.

Die Tabelle 2.3 enthält alle Verbindungen mit jeweiligem Source- und Target-Objekt, die bei Berücksichtigung der bisher erstellten Symbole und Datentypen definiert werden müßten. In Anbetracht der großen Anzahl beschränken wir uns auf einige exemplarische Fälle, die in den Abbildungen 2.10 bis 2.14 näher beschrieben werden.

Name: `ExprInPart`

Source: `Part`

Target: `Expression`

Constraints:
```
Inside(Expression, Part)
AlignCenter(Expression, Part)
```

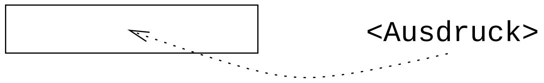


Abbildung 2.10: Nassi-Shneiderman-Diagramme: Programmteil – Ausdruck

Verbindung	**Source**	**Target**	**Bedeutung**	**Abb.**
`ExprInPart`	`Part`	`Expression`	Darstellung eines Ausdrucks in einem Programmteil	2.10
`ExprInCase`	`Case3`	`Expression`	Variable für eine Fallunterscheidung (CASE)	
`CondInSel`	`Selection`	`Condition`	Bedingung für eine If-Then-Fallunterscheidung	2.13
`CondInDoUntil`	`DoUntil`	`Condition`	Bedingung für eine Do-Until-Wiederholung	
`PartInSeq`	`Part`	`Sequence`	Hinzufügen eines Programmteils zu einer Abfolge	2.12
`PartAppPart`	`Part`	`Part`	Festlegen des Nachfolgers für einen Programmteil	2.11
`SeqInTrueSel`	`Sequence`	`Selection`	Abfolge im True-Teil einer Fallunterscheidung (IF THEN)	2.14
`SeqInFalseSel`	`Sequence`	`Selection`	Abfolge im False-Teil einer Fallunterscheidung (IF THEN)	
`SeqInCase3_1`	`Sequence`	`Case3`	Abfolge im Fall 1 einer Fallunterscheidung (CASE)	
`SeqInCase3_2`	`Sequence`	`Case3`	Abfolge im Fall 2 einer Fallunterscheidung (CASE)	
`SeqInCase3_3`	`Sequence`	`Case3`	Abfolge im Fall 3 einer Fallunterscheidung (CASE)	
`SeqInDoUntil`	`Sequence`	`DoUntil`	Abfolge in einer Wiederholung (UNTIL)	

Tabelle 2.3: Nassi-Shneiderman-Diagramme: Grafische Verbindungen

Name: `PartAppPart`

Source: `Part`

Target: `Part`

Constraints: `AnchorBottom(Part, Part)`
`SameWidth(Part, Part)`
`AlignLeft(Part, Part)`

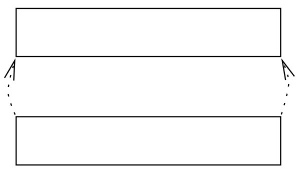


Abbildung 2.11: Nassi-Shneiderman-Diagramme: Programmteil – Programmteil

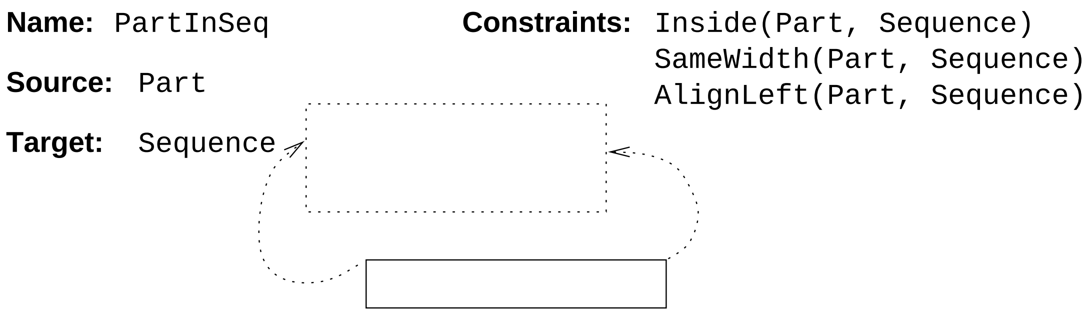


Abbildung 2.12: Nassi-Shneiderman-Diagramme: Programmteil – Abfolge

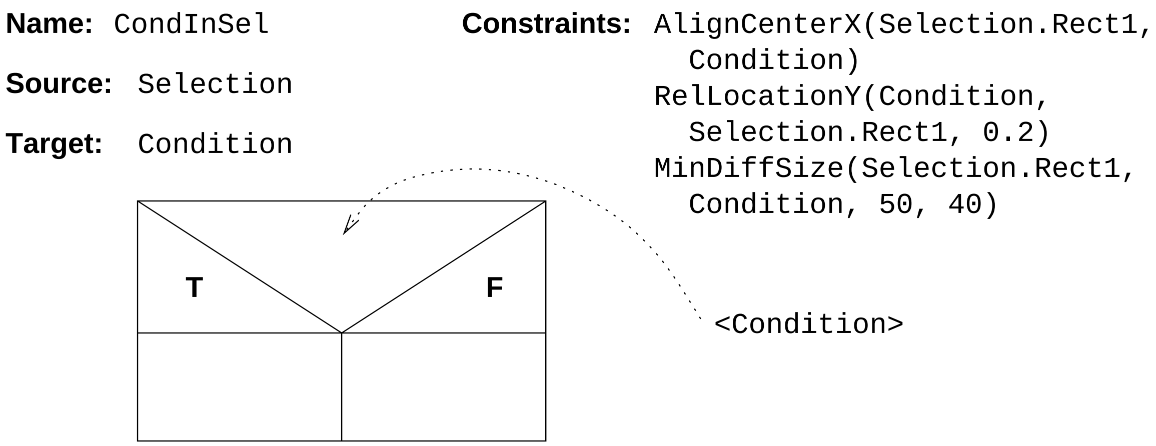


Abbildung 2.13: Nassi-Shneiderman-Diagramme: Bedingung – Fallunterscheidung

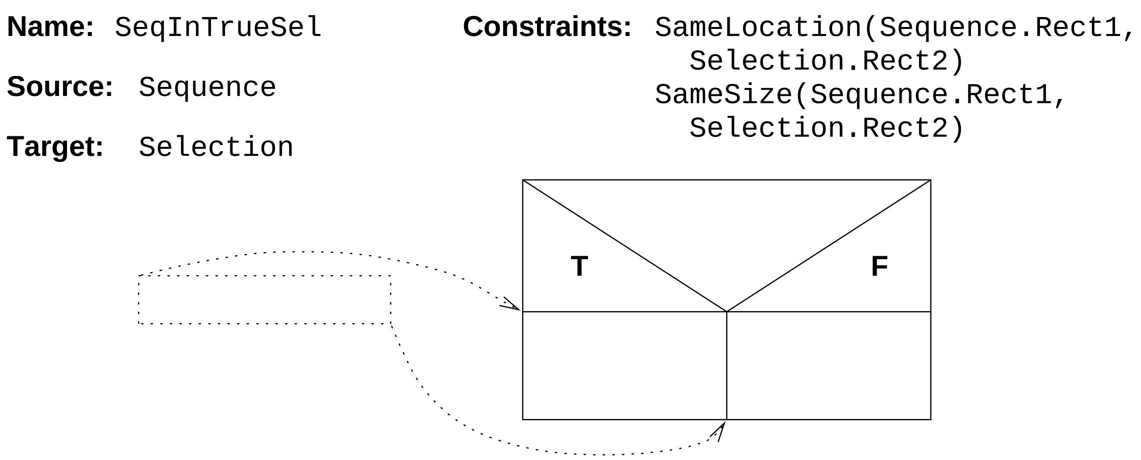


Abbildung 2.14: Nassi-Shneiderman-Diagramme: Abfolge – Fallunterscheidung

2.2 Alphabet-Editor

Der Alphabet-Editor soll Funktionen anbieten, mit denen der Anwender ein Alphabet erstellen und verändern kann. Die Struktur des Editors folgt idealerweise der Struktur eines Alphabets. Folgende grundlegende Funktionen müssen vorhanden sein:

- Definition von Symbolen:
  - Typisierung (Namensgebung)
  - Definition der Primitiven
  - Festlegen von Eigenschaften für die Primitiven
  - Definition von Verbindungspunkten
  - Angabe von Constraints zwischen den Primitiven bzw. Verbindungspunkten
- Definition von Datentypen:
  - Typisierung (Namensgebung)
  - Auswählen der Art des Datentyps (Typzuweisung)
  - Festlegen von Eigenschaften
  - Definition von Verbindungspunkten
- Definition von grafischen Verbindungen:
  - Definition der Operation (Source, Target)
  - Angabe von Constraints zwischen den beteiligten Grafiken

Jegliche Daten sollten nach Möglichkeit auch nachträglich veränderbar sein. Das betrifft insbesondere Symbole oder Datentypen, die bereits an grafischen Verbindungen beteiligt sind. Werden sie nachträglich verändert, indem man Primitiven hinzufügt, entfernt oder editiert, müssen sich die Änderungen auch innerhalb der Verbindungen bemerkbar machen.

Das auf diese Art erstellte Alphabet soll auf einen Datenträger gesichert und auch wieder geladen werden können. Um zukünftige Erweiterungen zu berücksichtigen, sollte für jeden Bestandteil des Alphabets eine Versionsnummer vorhanden sein.

Der Alphabet-Editor wird nachgeschaltete Komponenten besitzen, die mit dieser Diplomarbeit nur indirekt zu tun haben. Dabei handelt es sich vor allem um den Grammatik-Editor, mit dem der Anwender die visuelle Sprache um Grammatikregeln erweitern kann. Der Grammatik-Editor wird auf das erstellte Alphabet vorerst nur unter Nutzung einer Datei zugreifen. Es wird keine direkte Kopplung zwischen beiden Editoren geben. Zur Anwendung der gesamten visuellen Sprache wird ein Grafik-Editor implementiert, mit dem im Sinne des Alphabets syntaktisch korrekte Diagramme erstellt werden können. Dieser Editor folgt jedoch erst nach dem Grammatik-Editor und ist ebenfalls nicht Thema dieser Arbeit.

Symbol-Editor: Der Symbol-Editor bildet den ersten Teil des Alphabet-Editors. Mit ihm werden die Symbole und Datentypen erstellt. Dazu gehören natürlich deren primitive Grafikobjekte sowie verschiedenste Constraints. Zum Erstellen von grafischen Verbindungen zwischen den Symbolen muß der Benutzer in einen anderen Editor umschalten, der parallel zum Symbol-Editor innerhalb des Alphabet-Editors läuft. Beide Editoren arbeiten auf demselben Alphabet.

Verbindungs-Editor: Mit dem Verbindungs-Editor, dem zweiten Teil des Alphabet-Editors, kann der Anwender grafische Verbindungen zwischen Symbolen und Datentypen definieren. Durch Constraints kann für jede Beziehung das Layout der beteiligten grafischen Objekte angepaßt werden. Um weitere Symbole oder Datentypen zu erstellen oder schon bestehende zu verändern, kann wiederum der Symbol-Editor benutzt werden.

Test-System: Um das erstellte Alphabet auch ohne Grammatik testen zu können, wird ein Test-System entwickelt, welches das erstellte Alphabet ausschließlich aus einer Datei bezieht. Dadurch wird die Unabhängigkeit des Test-Systems vom Alphabet-Editor demonstriert. Mit dem Test-System kann der Benutzer Diagramme unter Benutzung aller Symbole, Datentypen und grafischen Verbindungen zeichnen. Es ist komplett constraint-, jedoch nicht regelbasiert. Daher ist der Benutzer selbst für die syntaktische Korrektheit verantwortlich, da es noch keine Grammatik gibt. Ohne Regeln kann zu jeder Zeit jede beliebige grafische Verbindung angewendet werden, die zwischen Symbolen bzw. Datentypen definiert wurde. Das Test-System ist zwar optisch in den Alphabet-Editor integriert, besitzt aber vollkommen eigene Datenstrukturen. Diese sind im Symbol- und Verbindungs-Editor nicht sichtbar.

Constraint-Solver: Ein Constraint-System stellt ein (Un)gleichungssystem dar, das mitunter auf viele hundert (Un)gleichungen anwachsen kann. Eine optimale Lösung für solch ein komplexes System zu errechnen, erfordert spezielle Algorithmen, die nicht in den Alphabet-Editor integriert sind. Alle Teilsysteme des Alphabet-Editors, einschließlich des Test-Systems, benutzen daher ein externes Programm zum Lösen der definierten Constraints – einen Constraint-Solver.

Systemaufbau: Die Abbildung 2.15 zeigt den rudimentären Aufbau des Alphabet-Editors und sein Zusammenwirken mit anderen Komponenten.

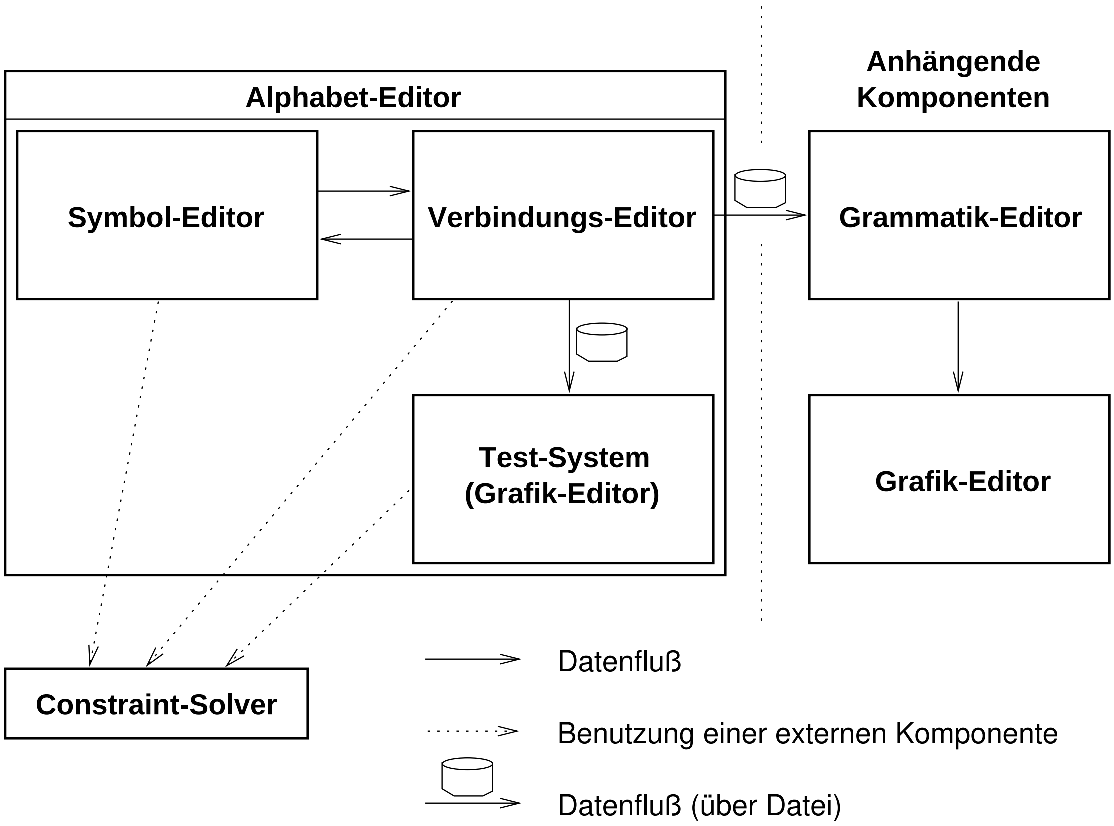


Abbildung 2.15: Rudimentärer Aufbau des Alphabet-Editors

Kapitel 3

Werkzeuge und Hilfsmittel

In diesem Kapitel wollen wir einige Werkzeuge vorstellen, die in dieser Diplomarbeit benutzt wurden. Wir betrachten einige Aspekte der *Unified Modeling Language*, eine Sammlung mehrerer visueller Sprachen, die bei der Analyse und dem Entwurf des Alphabet-Editors zum Einsatz kommen. Danach gehen wir auf den Begriff der *Design Patterns* ein und stellen vier ausgewählte Patterns vor, die bei der Implementierung berücksichtigt wurden. Nachdem wir in groben Zügen Vor- und Nachteile der verwendeten *Programmiersprache Java* diskutiert haben, widmen wir uns einem externen Programm namens PARCON , das zum *Constraint-Solving* eingesetzt wird.

3.1 Unified Modeling Language

Der Alphabet-Editor wurde objektorientiert entworfen und mit Hilfe einer objektorientierten Programmiersprache implementiert. Zur Beschreibung des Entwurfs verwenden wir die Unified Modeling Language (UML), die zur Zeit den Standard in Sachen objektorientierter Modellierung darstellt. UML versteht sich als Nachfolger der zuvor führenden Techniken BOOCH [Boo91], OMT [RBP$^+$91] und Objective [JCJv92]. Durch Zusammenarbeit von Grady Booch, James Rumbaugh und Ivar Jacobsen wurde die UML [UML99] entwickelt. Ziel der UML ist eine Vereinheitlichung der für verschiedene Aspekte des Entwurfs verwendeten Notationen. Dennoch werden viele Optionen für die konkrete Darstellung offengelassen.

Innerhalb der vorliegenden Arbeit werden wir nur zwei der vielen verschiedenen Notationen verwenden: Für die Modellierung statischer Strukturen benutzen wir *Klassendiagramme*, für dynamische Aspekte dagegen *Sequenzdiagramme*. Im folgenden soll ein kurzer Überblick zur Notation dieser beiden Diagrammarten gegeben werden. Zum genaueren Studium sei an dieser Stelle auf [Oes97] verwiesen.

3.1.1 Klassendiagramme

Klassendiagramme gehören zur Gruppe der *Static-Structure-Diagramme* und modellieren die statische Struktur eines Systems und der dazugehörigen Komponenten. Es werden sowohl Klassen als deren Beziehungen modelliert, wie beispielsweise Vererbung oder Aggregation. Die Abbildung 3.1 gibt einen Überblick über den Aufbau eines Klassendiagramms.

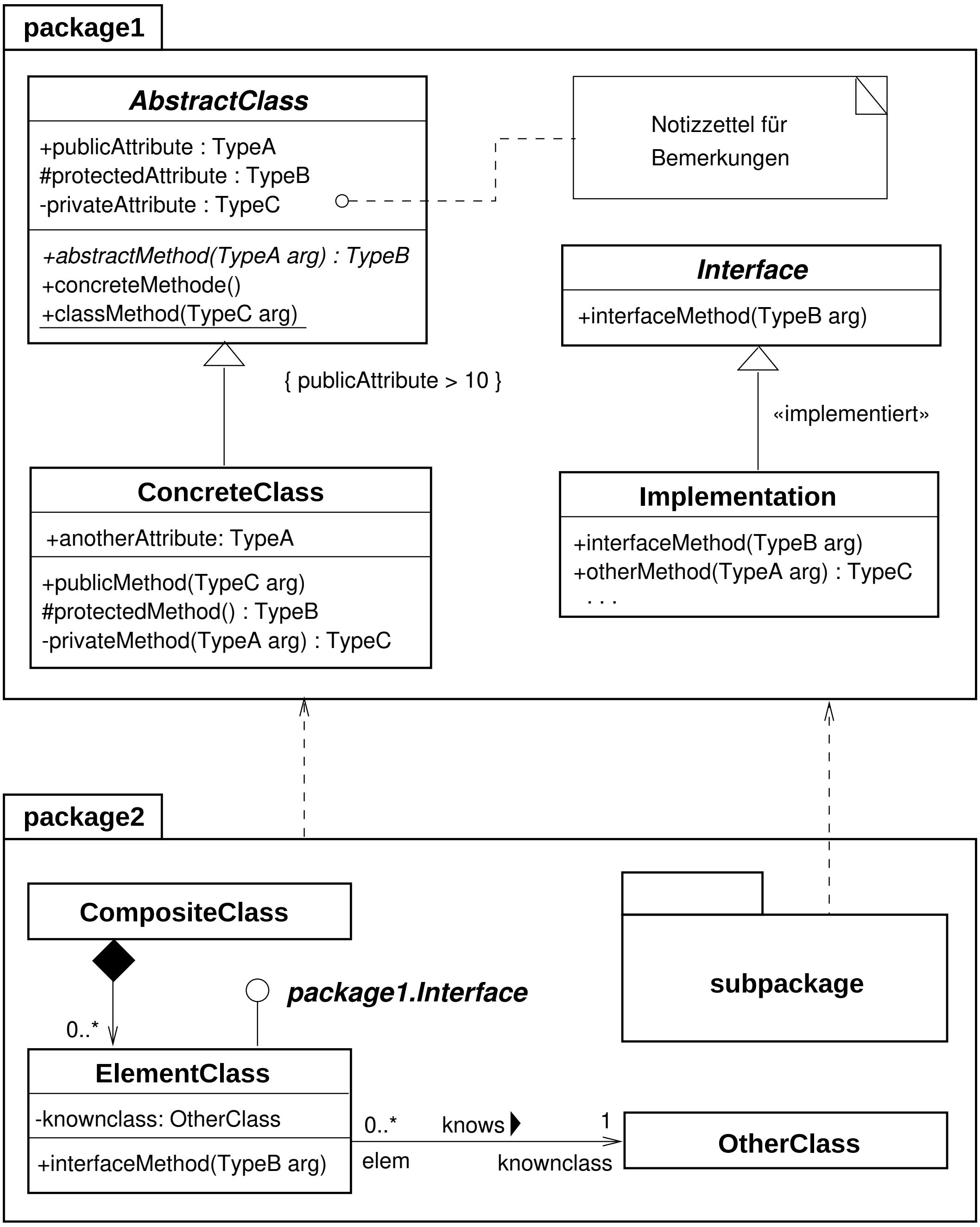


Abbildung 3.1: Unified Modeling Language: Notation von Klassendiagrammen

Bei dieser Art von Notation stehen die Begriffe *Package*, *Klasse* und *Interface* im Mittelpunkt. Eine **Klasse** und ein **Interface** werden durch ein Rechteck dargestellt, das mit Hilfe von horizontalen Linien in drei Teile aufgeteilt wird. Der obere Teil enthält fettgedruckt den Klassennamen. Die Namen von abstrakten Klassen und Interfaces werden kursiv gesetzt. Der mittlere Teil kann eine Liste von Attributen und der untere Teil eine Methodenliste enthalten. Beide Teile können leer sein. Sowohl bei den Attributen als auch bei den Methoden unterscheiden wir drei verschiedene Arten, die Aussagen über Zugriffsmöglichkeiten von außen machen: Es gibt öffentliche (public), geschützte (protected) und private Einträge. Die Art eines Attributs oder einer Methode wird durch ein vorangestelltes $+$, # oder $-$ gekennzeichnet. Abstrakte Methoden werden kursiv dargestellt. Statische Attribute und Methoden, die auch unter dem Namen Klassenattribute bzw. -methoden bekannt sind, werden unterstrichen. Die Typisierung der Attribute und der Rückgabewert von Methoden wird durch einen Doppelpunkt getrennt nach dem jeweiligen Namen angegeben. Da es nicht immer sinnvoll ist, alle Bestandteile einer Klasse aufzuführen, können Attribute oder Methoden „verborgen“ werden. Durch ein „...“ wird darauf hingewiesen. Der Blick des Betrachters soll dadurch auf das Wesentliche gelenkt werden. Es ist üblich, einer Klasse Zusicherungen zuzuordnen, die sich auf die abgeleiteten Instanzen beziehen. Sie werden in geschweifte Klammern eingeschlossen. Durch die Benutzung von *Stereotypen* kann einem Notationselement eine spezielle Bedeutung gegeben werden. Ein solcher Stereotyp wird in französische Anführungszeichen ($\ll \gg$) eingeschlossen.

Beliebig viele Klassen und Interfaces können zu einem **Package** zusammengefaßt werden. Dazu werden alle beteiligten Klassen durch ein Rechteck umschlossen, das links oben ein kleines Rechteck mit dem eingeschlossenen fettgedruckten Package-Namen enthält. Durch ein Package wird expizit der Namensraum für die enthaltenen Klassen definiert. Ein Package kann neben Klassen und Interfaces auch Subpackages enthalten. Zur eindeutigen Identifkation einer Klasse müssen die Package- und Subpackage-Namen in Form eines *Pfades* dem Klassennamen vorangestellt werden. Dasselbe gilt auch für die Interfaces. Als Trennzeichen für die Bestandteile eines Pfades wird ein Punkt benutzt (z.B. `package1.Interface`). Oft wird ein Package von den Klassen eines anderen Packages benutzt. Um eine generelle „Benutzt-Beziehung“ zu modellieren, kann ein gestrichelter Pfeil zwischen zwei Packages angegeben werden, dessen Spitze zu dem benutzen Package zeigt. Innerhalb eines Packages müssen nicht unbedingt alle enthaltenen Klassen oder Interface sichtbar sein. Um eine gewisse Übersichtlichkeit zu wahren, verbergen wir die Klassen, die nicht von Belang sind.

Zwischen zwei Klassen kann eine **Vererbungsbeziehung** bestehen, die durch eine Linie mit einem Dreieck dargestellt wird. Die Dreiecksspitze zeigt auf die Klasse, von der Attribute und Methoden übernommen werden. Auch die **Implementierung** eines Interfaces wird durch diese Notation ausgedrückt. Wird ein Interface von mehreren Klassen implementiert, ist das Benutzen dieser Notation nicht immer sinnvoll. Deshalb kann ein Interface auch durch einen kleinen Kreis repräsentiert werden, an dem nur der Name des Interfaces angegeben wird. Hierbei ist auf Eindeutigkeit zu achten.

Beziehungen zwischen den Instanzen zweier Klassen können durch **Assoziationen** ausgedrückt werden. Sie werden durch einen einfachen Pfeil oder eine Linie zwischen beiden Klassen angegeben, die mehrfach beschriftet sein kann. So können auf beiden Seiten der Linie Rollenbezeichnungen und Kardinalitäten angegeben werden. In der Mitte der Linie können ein oder zwei Namen vergeben werden. Oft wird der Name jedoch weggelassen. In der Abbildung 3.1 besteht zwischen den Klassen `package2.ElementClass` und `package2.OtherClass` eine solche Assoziation. Durch die Kardinalität 0..* wird ausgedrückt, daß jede Instanz von `OtherClass` in Beziehung zu beliebig vielen Instanzen von `ElementClass` stehen kann. Den Instanzen von `ElementClass` wird die Rollenbezeichnung „elem“ zugeordnet. Andererseits hat jede Instanz von `ElementClass` genau eine Beziehung zu einer Instanz von `OtherClass` (Kardinalität 1). Der Pfeil der Assoziation gibt an, daß `ElementClass` durch eine Variable (in unserem Fall `knownclass`) direkten Zugriff auf die Instanz von `OtherClass` besitzt. Wäre das in beiden Richtungen der Fall, würden wir statt eines einfachen Pfeils eine einfache Linie benutzen. Oft existieren zwischen zwei Klassen mehrere Assoziationen. Zur Vereinfachung verwenden wir oft nur einen Assoziationspfeil, der jedoch mit mehreren, durch Kommata getrennten Rollennamen beschriftet ist. In einem solchen Fall entspricht die angegebene Kardinalität jedoch nicht etwa der Summe der Einzelkardinalitäten, sondern der Kardinalität jeder einzelnen Assoziation.

Die Bedeutung einer Assoziation kann in Form einer **Aggregation** oder einer **Komposition** verstärkt werden. Beide Notationen sind Ausdruck einer „besteht aus“-Beziehung. Die Aggregation wird durch einen Assoziationspfeil, der auf einer Seite eine ungefüllte Raute besitzt, dargestellt. Die Kardinalität auf dieser Seite ist generell „1“. Eine Aggregation könnte auch durch eine entsprechende Assoziation dargestellt werden. Anders verhält es sich mit der Komposition. Sie wird durch eine gefüllte Raute gekennzeichnet. Auch für eine Komposition gilt immer die Kardinalität „1“. Allerdings besteht im Gegensatz zur Aggregation eine zusätzliche Abhängigkeit, die die Lebensdauer der Instanzen betrifft: Sobald die Instanz des Kompositums stirbt, sterben auch alle durch Komposition gebundenen Elemente.

3.1.2 Sequenzdiagramme

Mit Sequenzdiagrammen kann die Interaktion zwischen Instanzen verschiedener Klassen beschrieben werden. Die Abbildung 3.2 zeigt eine Beispiel-Notation für ein Sequenzdiagramm. Eine **Instanz** wird in einem Rechteck dargestellt, das den fettgedruckten Klassennamen enthält. Diesem Namen kann – durch einen Doppelpunkt getrennt – ein eigener Instanzenname vorangestellt werden. Im Unterschied zu den Klassendiagrammen wird der Name in einem Sequenzdiagramm unterstrichen, um deutlich zu machen, daß es sich um Objekte handelt. Entlang der vertikalen Achse, die nicht extra gezeichnet wird, verläuft die Zeit. Unter dem Rechteck einer Instanz wird entweder ein weißer Balken oder eine gestrichelte Linie angegeben. Der Balken besagt, daß die Instanz gerade „benutzt wird“, die gestrichelte Linie dagegen stellt die Lebensdauer der Instanzen dar. Ein Pfeil notiert den **Aufruf einer Methode** oder eines **Konstruktors**. Durch die Angabe einer Bedingung in eckigen Klammern können Verzweigungen auftreten. Die aufgerufene Methode kann in anderen Instanzen weitere Methoden aufrufen. Das **Terminieren der Methode** wird durch einen Pfeil zurück zur aufrufenden Instanz ausgedrückt. Dieser Pfeil kann mit einem Rückgabewert beschriftet sein. Übrigens besteht die Möglichkeit, daß eine Instanz eine **eigene Methode** aufruft (siehe Methodenaufruf `callOwn(arg)` in Abbildung 3.2). Wird ein Objekt aus dem Speicher entfernt (C++) bzw. wird es von keinem anderen Objekt referenziert (Java), endet seine **Lebensdauer**. Dieser Umstand wird durch ein Kreuz am Ende der gestrichelten Linie ausgedrückt.

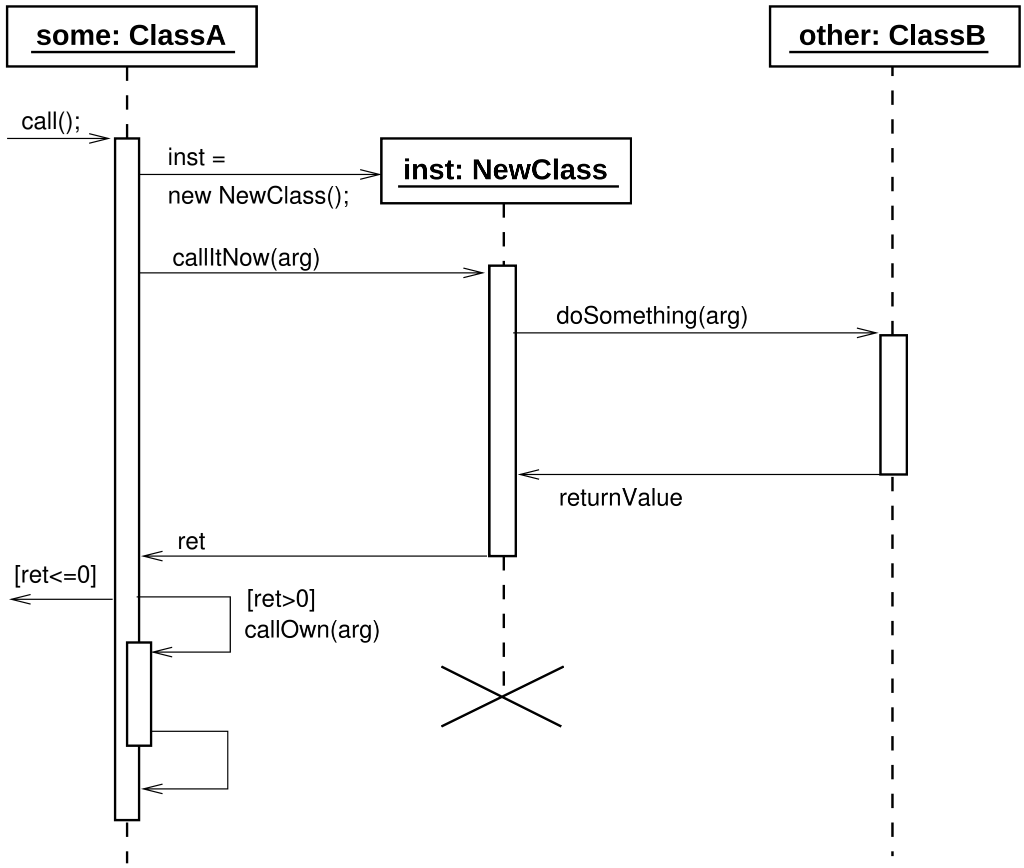


Abbildung 3.2: Unified Modeling Language: Notation von Sequenzdiagrammen

3.2 Design Pattern

Software mit Hilfe von Design Patterns[1] zu entwerfen, ist ein neuer Ansatz in der Softwareentwicklung. Muster beschreiben erprobte Lösungen für viele häufig wiederkehrende Probleme in dem Entwurf und in der Implementierung von Software-Systemen. Ein Design Pattern wird dabei gewissermaßen als „Kochrezept“ formuliert, das jedoch unabhängig von einer bestimmten Programmiersprache oder einem Programmierparadigma ist. Es existieren verschiedene Abstraktionsstufen von Mustern.

Architekturmuster: Architekturmuster werden zur grundsätzlichen Strukturierung von Software verwendet. Sie bilden die höchste Abstraktionsstufe. Laut [BMR+98] beschreiben sie vordefinierte Sub-Systeme. Dabei werden deren Zuständigkeitsbereiche spezifiziert und Regeln zur Organisation der Beziehungen formuliert. Das wahrscheinlich bekannteste Beispiel eines Architekturmusters ist das Model-View-Controller-Pattern.

[1] engl. Entwurfsmuster

Entwurfsmuster: Mit einem Entwurfsmuster wird die Verfeinerung der Subsysteme, der Komponenten oder der Beziehungen in einem Software-System beschrieben, z.B. die Strukturen zur Zerlegung komplexer Dienste oder Komponenten. Andere wiederum beschreiben die Kooperation zwischen verschiedenen Komponenten. Das Observer-Pattern, das in [GHJV95] vorgestellt wird, ist eines der verbreitetsten Entwurfsmuster.

Idiome: Die niedrigste Abstraktionsstufe stellen die Idiome dar. Sie machen Aussagen zur Implementierung von speziellen Entwurfsmustern. Idiome sind – abweichend von den anderen beiden Abstraktionsstufen – abhängig von einer bestimmten Programmiersprache. Sie beschreiben, wie ein Entwurf mit den Mitteln einer speziellen Sprache implementiert werden kann. Beispiel für ein Idiom ist das Singleton-Pattern.

Jedes dieser Muster bildet letztlich ein Problem-Lösungs-Paar und unterliegt einem dreiteiligen Schema. Ein *Kontext* beschreibt die Situation, in der ein Problem auftritt. Diesen Kontext vollständig zu beschreiben, ist meist unmöglich. Idealerweise müßte man alle Situationen aufzählen, in denen ein Muster angewendet werden kann. Die Beschreibung eines *Problems*, das im Kontext häufig auftritt, bildet das zweite Schema. Dabei werden die Randbedingungen sowie Anforderungen und wünschenswerte Eigenschaften für eine mögliche Lösung beschrieben. Der letzte Teil einer Musterbeschreibung gibt die Beschreibung einer *Lösung* zu dem Problem an. Sie versucht, die verschiedenen Anforderungen zu erfüllen und insbesondere entgegenwirkende Kräfte auszugleichen.

Durch die Verwendung von Design Patterns wird eine bessere Wartbarkeit und Wiederverwendbarkeit von Teilen eines Software-Systems erreicht. Inzwischen sind durch den großen Erfolg einiger Bücher, wie z.B. [GHJV95] oder [BMR+98], viele Patterns unter den Softwareentwicklern bekannt. Dadurch kann die Beschreibung von Software zum Teil auf einer abstrakteren Ebene stattfinden. Um Mißverständnissen vorzubeugen: Es gibt nur sehr selten ein Entwurfsmuster, das direkt auf ein Problem anzuwenden ist. Meist müssen entsprechende Anpassungen vorgenommen werden. Nicht zuletzt hängen diese Anpassungen auch von der verwendeten Programmiersprache ab.

Im folgenden sollen sechs Design Patterns vorgestellt werden, die in das System des Alphabet-Editors eingeflossen sind. Da die Beschreibungen kurz gehalten wurden, verweisen wir für ausführlichere Informationen auf [BMR+98].

3.2.1 Observer

Kontext: In einem System existieren verteilte Komponenten, die Daten einer ausgezeichneten Komponente benutzen.

Problem: Die Änderung der Daten in einem System kann zu Inkonsistenzen zwischen kooperierenden Komponenten führen. Es wird ein Mechanismus benötigt, der alle interessierten Komponenten über Änderungen der Daten informiert. Die Komponenten sollen jedoch nur lose gekoppelt sein. Die Komponente, die die Daten verwaltet und zur Verfügung stellt, soll außerdem unabhängig von den anderen Komponenten sein.

Lösung: Zwischen der Komponente, die Daten bereitstellt, – dem `Subject` – und den von den Daten abhängigen Komponenten – den `Observern` – wird ein Mechanismus implementiert, der über Änderungen der Daten informiert. Jeder Beobachter kann sich dynamisch bei diesem Mechanismus anmelden und auch wieder abmelden. Ändern sich die Daten des Subjekts, werden alle angemeldeten Beobachter durch Aufruf einer speziellen Funktion namens `update` davon in Kenntnis gesetzt. Die Informationen über die Art der Änderungen an den Daten kann entweder als Parameter an die `update`-Funktion übergeben (Push-Modell) oder später von den Beobachtern separat erfragt werden (Pull-Modell). Meist wird eine Kombination beider Modelle implementiert.

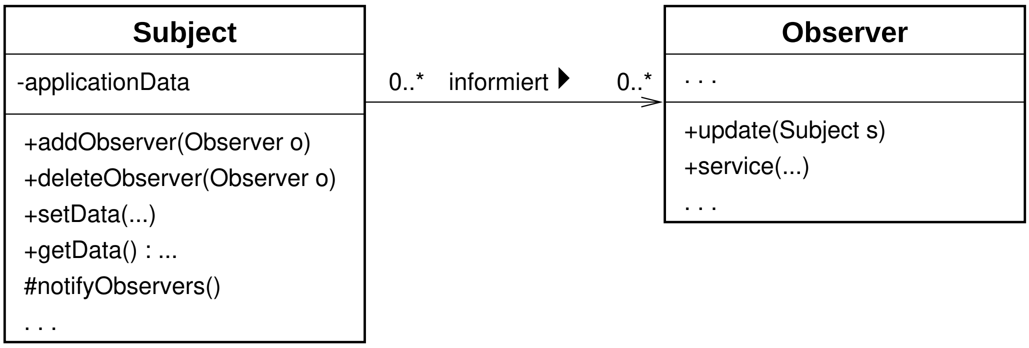


Abbildung 3.3: Design Pattern: Observer

Das Observer-Pattern wird bereits durch die Programmiersprache Java unterstützt. Es existiert eine Klasse `java.util.Observable`, die der Klasse `Subject` entspricht und vom Benutzer erweitert werden kann. Dazu gibt es das Interface `Observer`.

3.2.2 Listener

Kontext: Wie beim Observer-Pattern existiert auch hier ein System mit verteilten Komponenten, die Daten einer ausgezeichneten Komponente benutzen.

Problem: Das Observer-Pattern wird schnell unübersichtlich, wenn der Umfang der Daten des Subjekts sehr groß ist und zusätzlich zwischen verschiedenen Daten unterschieden werden soll. Das Problem vom Observer wird erweitert: Verschiedene Komponenten möchten darüber informiert werden, daß sich *ein bestimmter Teil* der Daten einer Komponente geändert hat. Würde jegliche Kommunikation über die `update`-Methode abgewickelt, würde die Implementierung der Observer-Klassen schnell unübersichtlich.

Lösung: Der Mechanismus des Observer-Patterns wird erweitert. Die Komponente, die Daten bereitstellt, unterscheidet verschiedene Arten von Änderungen. Zu dieser Klasse wird ein Interface bereitgestellt, das *verschiedene* Methoden deklariert, um nur über bestimmte Änderungen informiert zu werden. Wie diese Methoden aussehen (Methodennamen, Parameter), bleibt dem Modellierer überlassen. Der Name dieses Interfaces bekommt den Suffix `Listener`[2]. Alle Klassen, die über Datenänderungen informiert werden möchten, implementieren dieses Interface und melden sich über eine Methode `addXListener(XListener l)`[3] bei der Komponente an, die die Daten enthält. Die Abmeldung erfolgt über eine Methode namens `removeXListener(XListener l)`. Es ist möglich, das verschiedene Listener-Interfaces angeboten werden. Dadurch können verschiedene Kategorien von Beobachtern generiert werden. Zum anderen können innerhalb eines Listeners beliebig viele Methoden zur Verfügung stehen, um zwischen verschiedenen Arten von Änderungen zu unterscheiden. Wie beim Observer-Pattern können wir das Pull- oder Push-Modell zur Übermittlung von Daten verwenden.

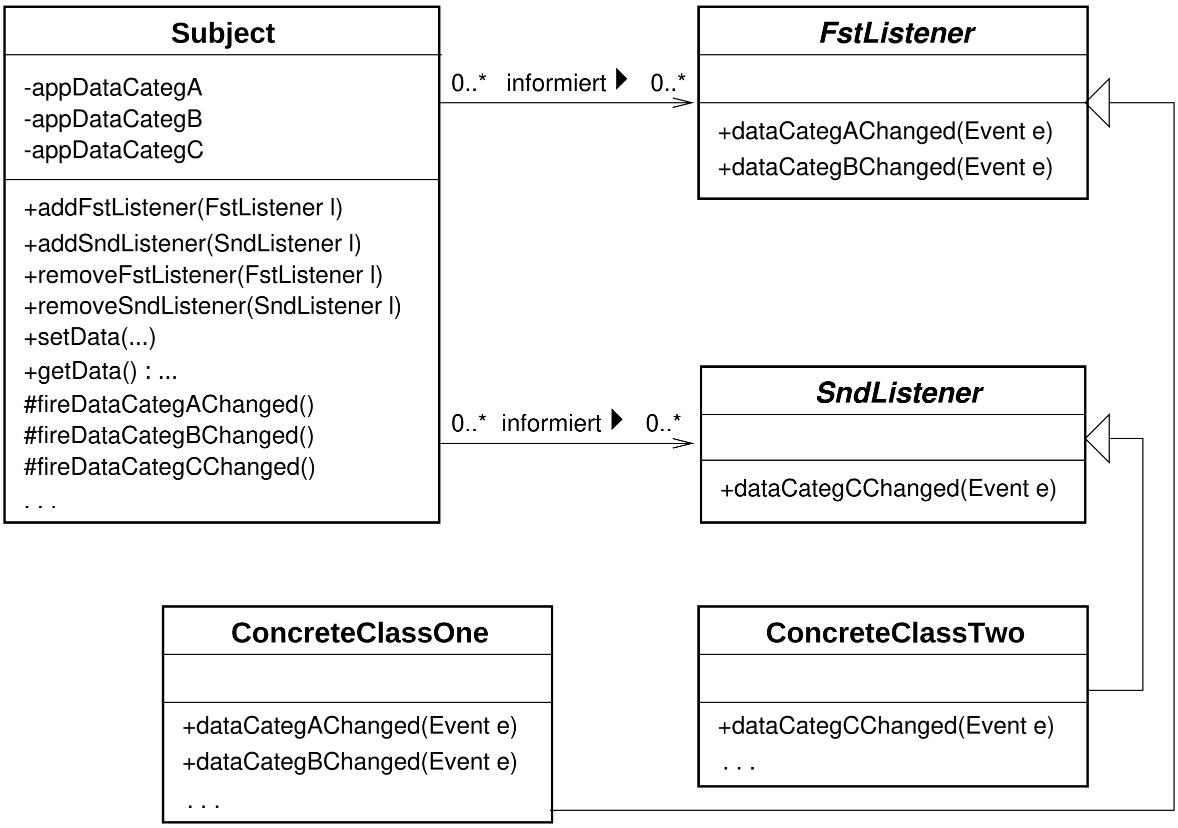


Abbildung 3.4: Design Pattern: Listener

In der Programmiersprache Java wird dieses Design Pattern sehr oft benutzt. Jegliche Kommunikation, die zum einen Ereignisse, zum anderen aber auch „normale" Daten betrifft, wird über das Implementieren verschiedener Listener-Interfaces abgewickelt.

[2]engl. Lauschender (frei übersetzt)

[3]X wird hier durch ein intelligentes Wort ersetzt, das den Listener charakterisiert

3.2.3 Whole-Part

Kontext: In einem System sollen zusammengesetzte Objekte implementiert werden. Im Alphabet-Editor betrifft das beispielsweise den Aufbau eines Symbols aus verschiedenen Primitiven.

Problem: Oft bilden zusammengesetzte Objekte Einheiten, die mehr sind als eine bloße Ansammlung ihrer Teile. Es existieren Eigenschaften des Objektverbundes, die abhängig sind von den beteiligten Objekten. Die Kombination von einzelnen Objekten läßt ein neues Verhalten[4] entstehen. Ein komplexes Objekt sollte entweder in kleinere Teile zerlegt oder aus bereits existierenden Teilen zusammengesetzt werden. Dadurch wird die Wiederverwendung und Änderbarkeit der Objekte unterstützt. Außerdem wird dadurch ermöglicht, daß sich die Objekte auch in anderer Art und Weise zusammensetzen lassen. Zusätzlich wird gefordert, daß das Gesamtobjekt wie ein atomares Objekt aussieht und keinen direkten Zugriff auf seine Teilobjekte erlaubt.

Lösung: Wir führen eine Komponente ein, die als Gesamtobjekt fungiert. Sie kann kleinere Objekte – die Teilobjekte – kapseln und verhindert, daß andere Klassen direkten Zugang zu diesen Teilobjekten haben. Außerdem könnte eine Schnittstelle für das Gesamtobjekt definiert werden, um den direkten Zugriff auf die Funktionalität der Teilobjekte zu unterbinden. Dadurch wird sichergestellt, daß das Gesamtobjekt mit seinen gekapselten Teilobjekten eine semantische Einheit bildet. Diese vorgestellte Lösung ist sehr allgemein gehalten und muß in der Praxis an die jeweilige Situation angepaßt werden. In [BMR+98] sind verschiedene Varianten dieses Design Patterns erläutert. Im Alphabet-Editor verwenden wir die Variante des Composite-Pattern.

Variante Composite-Pattern: Bei dieser Abwandlung des Whole-Part-Pattern stellt das Gesamtobjekt einen Behälter dar, der seine Bestandteile selbst verwalten muß. Teilobjekte können hinzugefügt und entfernt werden. Der Hauptunterschied zum normalen Whole-Part-Pattern besteht jedoch in der Behandlung aller beteiligter Komponenten. Im Composite-Pattern werden alle Objekte, sowohl das Gesamtobjekt als auch alle Teilobjekte gleich behandelt, weil alle dieselbe abstrakte Schnittstelle implementieren oder von einer abstrakten Klasse abgeleitet sind. Die Abbildung 3.5 zeigt das von uns verwendete Composite-Pattern in einer allgemeinen Form.

Das Whole-Part-Pattern wird im Alphabet-Editor zur Modellierung der Datenstrukturen eines Symbols benutzt. Konkrete Informationen dazu sind im Kapitel 5 zu finden.

3.2.4 Command-Processor

Kontext: Dieses Pattern betrifft Anwendungen, die im Zusammenhang mit Benutzeraktionen Dienste zur Verfügung stellen, welche das Bearbeiten von Operationen oder das Zurücksetzen von Änderungen unterstützen.

[4] im Englischen als "Emergent Behavior" bezeichnet

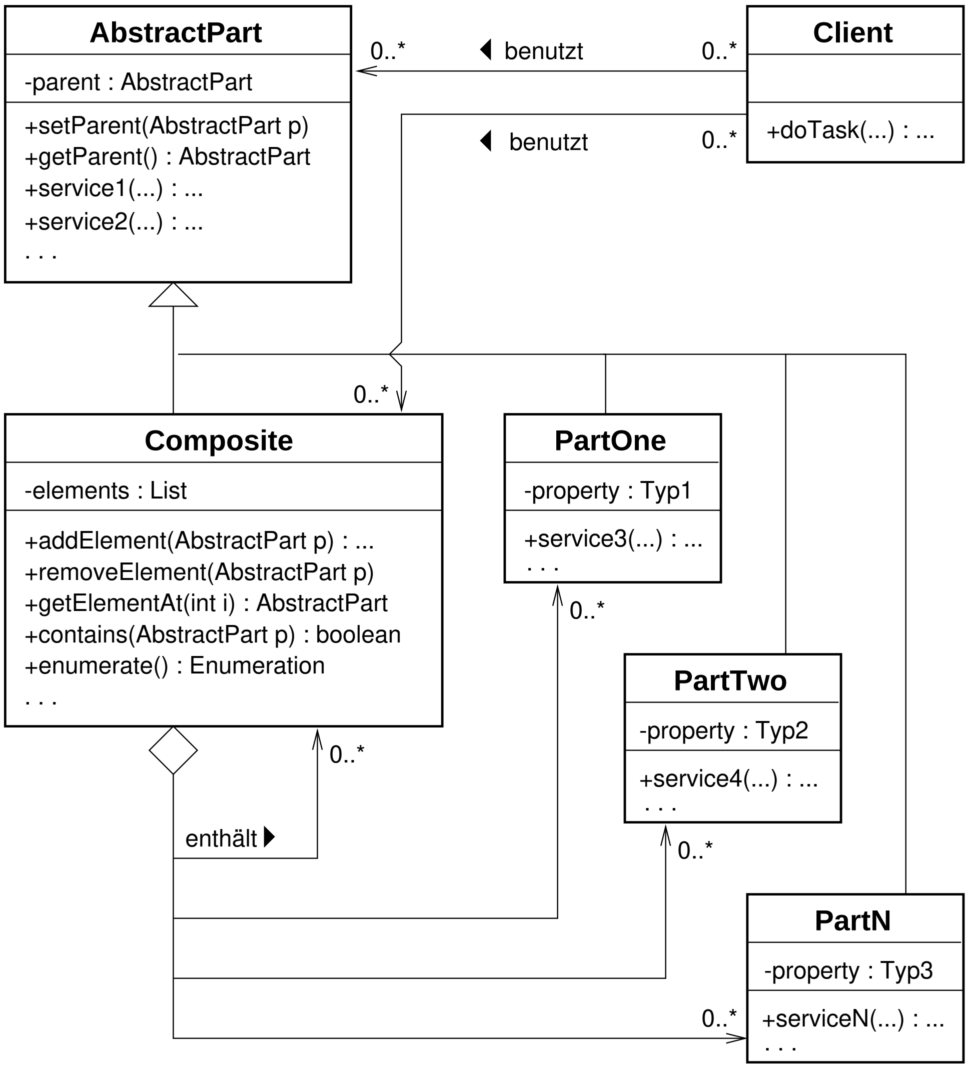


Abbildung 3.5: Design Pattern: Whole-Part, Variante des Composite-Pattern

Problem: Bei größeren Systemen müssen oftmals Dienste zum Ausführen von Operationen implementiert werden, die über die Kernfunktionalität des Systems hinausgehen. Gewünscht wird beispielsweise, daß Änderungen zurückgesetzt oder erneut ausgeführt werden können. Außerdem ist es sinnvoll, Benutzeraktionen zu protokollieren. Dabei soll beachtet werden, daß Erweiterungen der Anwendung den bereits bestehenden Code nicht zerstören. Zusätzliche Dienste wie die Undo-Funktionalität sollten konsistent für alle Operationen implementiert werden.

Lösung: Sobald ein Benutzer eine Operation anstößt, wird ein neues Befehlsobjekt erstellt, das diese Anforderung kapselt. Für jede Operation, die der Benutzer auf den Daten eines Systems ausführen kann[5], muß ein entsprechendes Befehlsobjekt

[5] dadurch wird der interne Zustand des Systems verändert

existieren, welches zu gegebener Zeit instanziiert wird. Alle Befehlsobjekte sind von einem abstrakten Befehlsobjekt abgeleitet, das zwei wichtige Funktionen zur Verfügung stellt: `redo` und `undo()`. An den Konstruktor des Befehlsobjekts werden alle relevanten Dienstleister und Daten übergeben, die zur Ausführung der Operation benötigt werden. Die Informationen, die zum Zurücksetzen der Operation nötig sind, können beispielsweise von den Dienstleistern angefordert werden. Die Methode `redo` ist für das Ausführen der Operation verantwortlich, die Methode `undo` dagegen für das Zurücksetzen. Da alle Befehlsobjekte denselben Aufbau haben, wird dadurch eine für alle Operationen konsistente Undo-Funktionalität implementiert. Ein Befehlsverwalter – der `CommandProcessor` – verwaltet alle erstellten Befehlsobjekte. Er speichert sie in einer Liste, um ein späteres Zurücksetzen zu ermöglichen. Außerdem stellt er andere Dienste zur Verfügung, z.B. das Protokollieren der einzelnen Schritte des Benutzers.

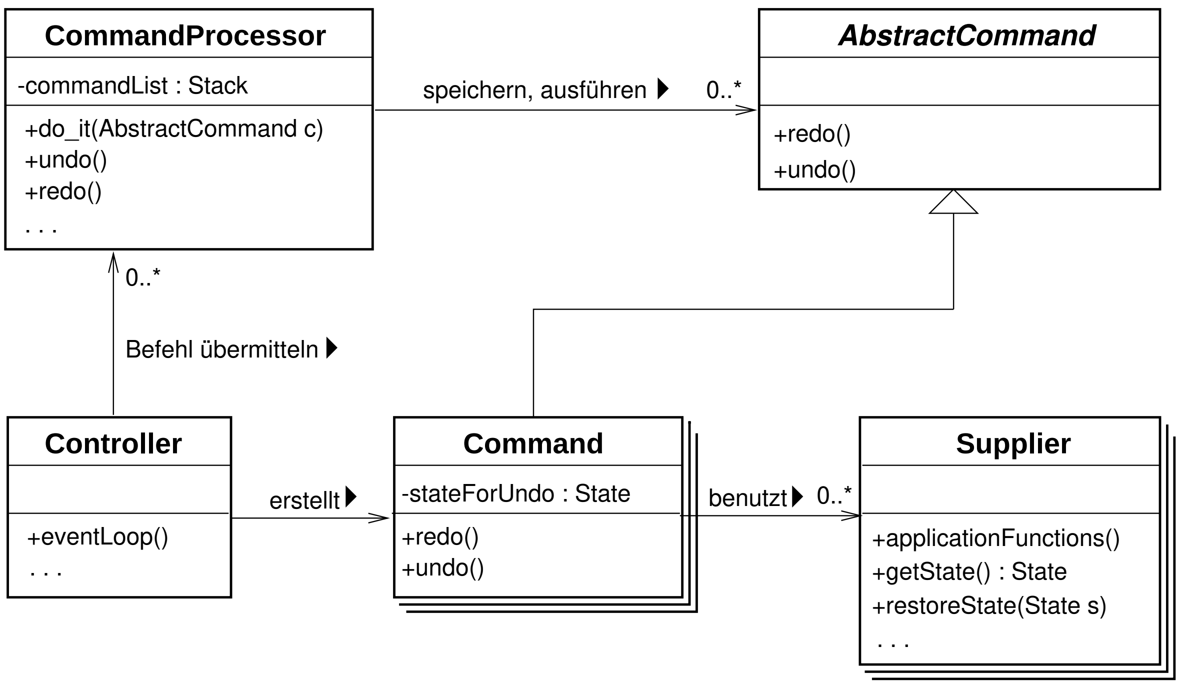


Abbildung 3.6: Design Pattern: Command-Processor

Die Swing-Klassen der Java-Version 2.0 enthalten unter anderem ein Package `javax.swing.undo`, in dem einige Interfaces und einfache Klassen für das Command-Processor-Pattern vorhanden sind. Für den Alphabet-Editor wurden dennoch teilweise neue Klassen implementiert, da die Funktionalität nicht ausreichend war. Auf die konkrete Implementierung wird im Abschnitt 6.5 eingegangen.

3.2.5 Model-View-Controller

Kontext: Dieses Pattern kann für alle interaktiven Anwendungen, die eine flexible Mensch-Maschine-Schnittstelle erfordern, benutzt werden.

Problem: Die Benutzerschnittstelle eines Systems ist die Komponente, die am häufigsten geändert werden muß, um Anpassungen vorzunehmen. Verschiedene Anwender ein- und desselben Systems stellen mitunter widersprüchliche Anforderungen an die Bedienoberfläche. Dementsprechend sollten verschiedene Paradigmen leicht integrierbar sein. Je enger die Benutzerschnittstelle jedoch mit dem funktionalen Kern des Systems verbunden ist, desto fehleranfälliger und teurer wird eine Überarbeitung bzw. Anpassung der Software. Weiterhin wird gefordert, daß dieselbe Information in mehreren Sichten dargestellt wird, eventuell auf verschiedene Art und Weise. Wird in der einen Sicht etwas an den Daten geändert, müssen alle anderen Sichten sofort auf die Änderung reagieren. Außerdem sollten Anpassungen der Bedienoberfläche leicht durchführbar sein – nach Möglichkeit sogar zur Laufzeit des Systems. Eine Portierung der Bedienoberfläche auf ein anderes System sollte den funktionalen Kern des Systems nicht beeinflussen.

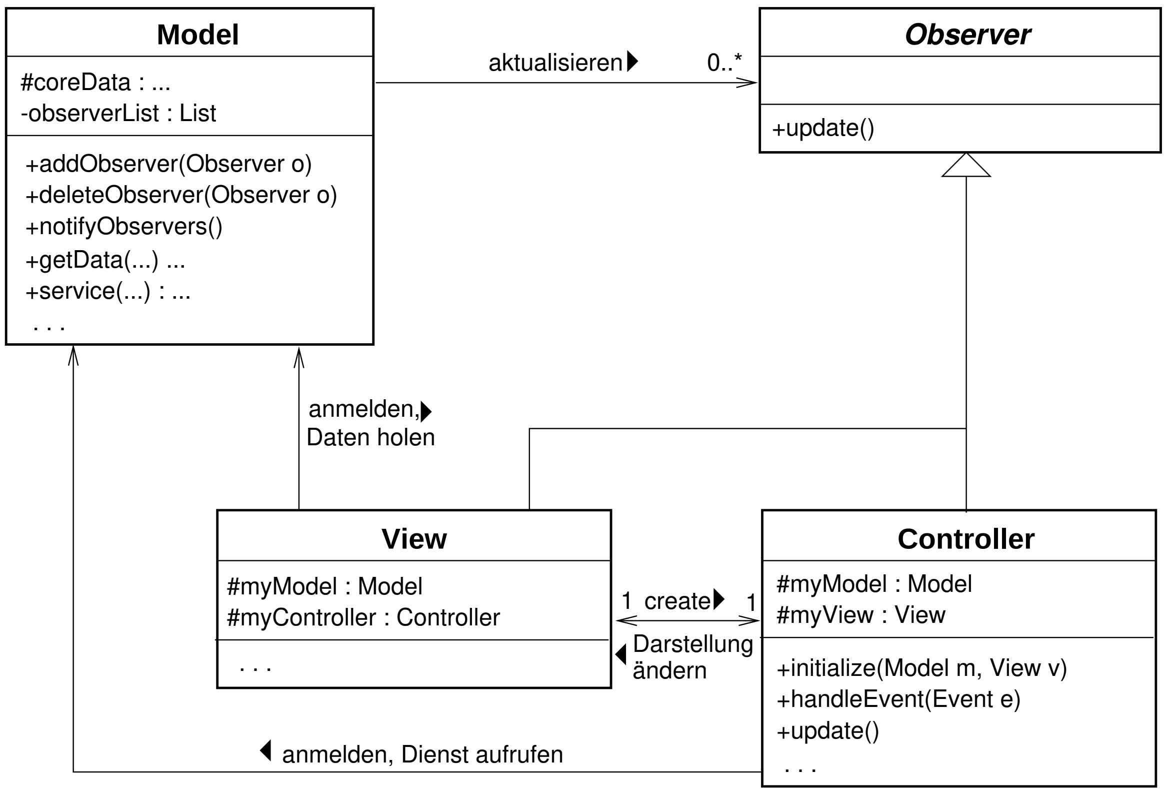


Abbildung 3.7: Design Pattern: Model-View-Controller

Lösung: Wir unterteilen eine interaktive Anwendung in drei Teile: Verarbeitung, Ausgabe und Eingabe. Die *Verarbeitungskomponente* – das Modell – verwaltet die Daten und die Kernfunktionalität des Systems. Sie ist im Grunde genommen unabhängig von speziellen Ausgabeformaten. Zur Repräsentation der Daten werden *Ansichten*, sogenannte Views, verwendet. Eine Ansicht erhält die Daten, die sie darstellt, von dem Modell. Sie ist jedoch für die Aufbereitung und Darstellung der Daten verantwortlich. Jede Ansicht ist gekoppelt mit einer eigenen *Kontrollkomponente*, die Benutzereingaben entgegennimmt und verarbeitet. Zur Verarbeitung eines Eingabeereignisses wird dieses in eine Dienstleistungsanforderung an das Modell (oder in einigen Fällen an die Ansicht) übersetzt. Ändert sich der interne Zustand des Modells, werden durch Nutzen des Observer-Patterns alle Ansichten informiert. Nachdem sich jede Ansicht die neuen relevanten Daten aus dem Modell geholt hat, wird die Darstellung angepaßt. Im Alphabet-Editor wurde statt dem Observer-Mechanismus das Listener-Pattern benutzt.

3.2.6 Singleton-Pattern

Kontext: Dieses Pattern kann für Systeme mit Mechanismen eingesetzt werden, die nur einmal innerhalb des gesamten Systems existieren sollen, aber von verschiedensten Komponenten genutzt werden.

Problem: In der Programmiersprache Java soll gewährleistet werden, daß es von einer bestimmten Klasse zur Laufzeit genau eine Instanz gibt. Das Pattern ist somit ein Idiom, weil es von der eingesetzten Programmiersprache abhängt.

Lösung: Wir machen den Konstruktor der betreffenden Klasse zu einer privaten Operation. Weiterhin wird ein privates statisches Attribut `sharedInstance` deklariert, das eine Referenz auf die einzige Instanz der Klasse enthält. Dieses Attribut wird mit null initialisiert, um anzuzeigen, daß zu Beginn keine Instanz der Klasse existiert. Zusätzlich definieren wir eine statische Methode `getSharedInstance()`, die den Wert des Attributs zurückgibt. Wird diese Methode das erste Mal aufgerufen, legt sie die einzige Instanz dieser Klasse mit dem Operator `new` an und setzt das Attribut `sharedInstance` auf diese Instanz.

```
public class Singleton {
  private static Singleton sharedInstance = null;
  // weitere Attribute

  private Singleton() {
    // Initialisierung
  }

  public Singleton getSharedInstance {
    if(sharedInstance==null)
      sharedInstance = new Singleton();
    return sharedInstance;
  }
}
```

3.3 Programmiersprache Java

Das System des Alphabet-Editors ist komplett in der von *Sun Microsystems* entwickelten Programmiersprache *Java* geschrieben. Die Wahl fiel nicht zufällig auf diese inzwischen weitverbreitete Sprache. Die dem Alphabet-Editor folgende Komponente – der Grammatik-Editor – arbeitet mit dem AGG-System zusammen. Dabei handelt es sich um ein Programm zur allgemeinen Graphtransformation sowie zur Verwaltung von Attributen. Da das AGG-System in Java implementiert wurde, mußte auch das gesamte GENGED-System, das Alphabet-, Grammatik- und Grafik-Editor einschließt, diesem Beispiel folgen. Die für uns relevanten Vor- und Nachteile von Java sollen in den folgenden zwei Abschnitten kurz diskutiert werden.

3.3.1 Vorteile

Der Hersteller von Java beschreibt in *"The Java Language: A Whitepaper"* die Sprache wie folgt: *„Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded and dynamic language."*

Nicht alle von diesen Schlagwörtern entsprechen der Wahrheit. Einige von ihnen sind inzwischen überholt. Was sind nun aber tatsächlich die Vorteile dieser weit verbreiteten Sprache?

Einfach: Java wurde bewußt schlankgehalten, was die fest in die Sprache integrierten Befehle und Möglichkeiten anbetrifft. Vieles ist aus den Sprachen C und C++ übernommen worden, einiges wurde weggelassen. So gibt es beispielsweise keine Pointer, d.h. der Programmierer muß sich nicht um die Speicherverwaltung kümmern – die beliebten Fehler „Segmentation fault" existieren in Java nicht. Trotz ihrer Einfachheit ist die Sprache dennoch mächtig genug, um anspruchsvolle Programme zu schreiben. Nicht zuletzt durch die bereits mitgelieferten Klassenbibliotheken existieren Lösungen für viele Probleme.

Objektorientiert: Die Objektorientierung bietet einen entscheidenden Vorteil für Entwickler, die mit dieser Art der Modellierung und Programmierung vertraut sind. Durch sie ist es möglich, auf sehr schnelle Weise Sachverhalte und Theorien als Objekte mit Eigenschaften und Verhaltensweisen zu modellieren und in eine Implementierung umzusetzen. Durch Vererbungsmechanismen können mitunter viele Programmzeilen eingespart werden.

Verteilt: Java unterstützt von Hause aus Applikationen auf Netzwerken. Dazu existieren verschiedenste Klassen, die einfach anzuwenden sind. Für uns besonders von Vorteil ist die Nutzung der Klasse `java.net.Socket`, über die auf sehr einfache Weise eine Socket-Verbindung zu einem Daemon realisiert werden konnte, der nicht unter Java läuft.

Robust: Die starke Typsisierung von Java sowie das zuverlässige Speichermodell verhindern einige Arten von Programmierfehlern. Der Programmierer muß sich beispielsweise keinerlei Gedanken mehr zur Freigabe von benutztem Speicher machen, da Java über einen Mechanismus namens Garbage Collection verfügt, der automatisch nicht mehr benötigten Speicher freigibt. Durch die Möglichkeit, zur Laufzeit auftretende Fehler und Ausnahmen abzufangen und zu behandeln, wird ein Programm ebenfalls robuster.

Portabilität: Java ist plattformunabhängig. Das heißt: Java-Applikationen laufen auf jedem System, auf dem eine Java Virtual Machine implementiert ist. Dadurch ist es möglich, das der Alphabet-Editor sowohl unter Linux, Solaris, Windows 98, Windows NT als auch auf einem Macintosh läuft. Wird die grafische Benutzeroberfläche mit Hilfe der Swing-Klassen[6] implementiert, kann sogar sichergestellt werden, daß die Arbeitsfläche auf allen Plattformen gleich aussieht. Diesen Vorteil zu nutzen, fällt allerdings schwer, da wir einen externen Constraint-Solver benutzen, der in Objective C geschrieben wurde und zur Zeit nur unter Solaris und Linux läuft.

Multi-Thread-Unterstützung: Java unterstützt die Ausführung mehrerer Threads, die verschiedene Aufgaben wahrnehmen können.

Dynamik: Java ist eine dynamische Sprache. Unter anderem ermöglicht sie, Klassen in ein laufendes System einzubinden. Außerdem haben alle verwendeten Klassen und Objekte eine sogenannte Laufzeit-Repräsentation, die spezielle interne Angaben zu einem Objekt enthält.

3.3.2 Nachteile

Neben den vielen Vorteilen, die Java bietet, existieren auch einige Nachteile, die nicht verschwiegen werden sollen.

Interpretersprache: Java ist eine Interpretersprache, die zwar in einen plattformunabhängigen Byte-Code kompiliert, aber erst zur Laufzeit in Maschinen-Code übersetzt wird. Das Übersetzen kostet viel Zeit. Vergleiche haben gezeigt, daß ein Java-Programm ungefähr zehn bis zwanzigmal langsamer läuft als ein vergleichbares C++-Programm.

Portabilität: Dieser Punkt wurde bereits als Vorteil aufgeführt. Leider gibt es einige „Nebenwirkungen“, die vom Hersteller gern verschwiegen werden. Besonders bei der Benutzung ausgefeilter grafischer Benutzeroberflächen gibt es mitunter Probleme, wenn ein Java-System auf unterschiedlichen Plattformen läuft und der Programmierer während der Entwicklung nicht über ein Höchstmaß an Einsicht in die verwendete Klassenbibliothek verfügt hat.

Geschwindigkeit: Wie schon erwähnt, ist Java natürlich nicht so schnell wie eine vollständig übersetzte Sprache. Besonders unter der Anwendung von Swing-Klassen leidet die Geschwindigkeit eines Java-Programms sehr stark. Betrachtet man dagegen andere interpretierte Sprachen, muß allerdings eingeräumt werden, daß Java aus dieser Kategorie eine der schnellsten Sprachen darstellt.

[6] auch Java Foundation Classes (JFC) genannt

3.3.3 Ausblick

Seit Sun Microsystems die Programmiersprache Java auf den Markt brachte, sind einige Jahre vergangen. Java erfreut sich inzwischen immer größerer Beliebtheit, nicht zuletzt wegen der großen Möglichkeiten, die die Sprache für das Internet bietet. Aus diesem Grunde gehen wir davon aus, daß die Sprache noch eine lange Zeit weiter gepflegt, entwickelt und damit verbessert wird. Da auch die Computer immer schneller werden, sind vielleicht einige der Nachteile bald nicht mehr relevant. Seit einiger Zeit wird intensiv an der Entwicklung von *Just-In-Time-Compilern* (JIT-Technik) gearbeitet, die Java-Code in plattformabhängigen Maschinencode übersetzen. Damit könnten laut Hersteller Ausführungszeiten erreicht werden, die C++ entsprechen. Bisherige eigene Erfahrungen mit den ersten Entwicklungen von JIT können diese Aussage jedoch nicht bestätigen.

3.4 Constraint-Solver PARCON

Der Constraint-Solver PARCON[7] ist ein in Objective C geschriebenes System, das bereits 1995 von Peer Griebel an der Universität-Gesamthochschule Paderborn entwickelt wurde [Gri96]. Er ist auf das parallele Lösen zweidimensionaler grafischer Constraint-Systeme spezialisiert. Wir benutzen den Constraint-Solver in allen Teil-Editoren des Alphabet-Editors – im Symbol- und Verbindungs-Editor sowie im Test-System. Auch nachfolgende Komponenten, wie beispielsweise der Grammatik-Editor, stehen über eine von uns entwickelte Schnittstelle mit PARCON in Verbindung.

Wird PARCON ohne diese Schnittstelle benutzt, können Constraints ausschließlich über die Standardeingabe durch ASCII-Text eingegeben werden. Dabei ist einer speziellen Syntax zu folgen, die im folgenden kurz beschrieben werden soll.

3.4.1 Variablen

Ein Constraint-System stellt ein (Un)gleichungssystem dar, für das eine optimale Lösung ermittelt werden soll. Dazu muß PARCON alle an diesem System beteiligten Variablen so mit Werten belegen, daß das System lösbar wird. Der Constraint-Solver versucht dabei, möglichst wenige Variablen zu ändern. Da es sich um einen Solver für grafische Constraints handelt, enthalten die Variablen meist Koordinaten, Breite oder Höhe eines Objekts, den Radius oder eine andere spezielle Eigenschaft einer geometrischen Figur. PARCON selbst unterscheidet zwischen zwei Arten von Variablen.

Punktvariable: Um die Aufstellung bestimmter Constraints zu vereinfachen, kann PARCON mit Punkten umgehen. Ein Variablenname, der eine Kombination aus Buchstaben und Ziffern darstellt, wird von PARCON standardmäßig für eine Punktvariable gehalten. Die x- und y-Koordinate eines Punktes kann durch

[7] Im World Wide Web kann der Constraint-Solver PARCON als Linux- oder Solaris-Version von der Seite `http://www.uni-paderborn.de/fachbereich/AG/szwillus/forschung/parcon/parcon.html` heruntergeladen werden.

Anfügen von `.x` oder `.y` dereferenziert werden. Intern wird ein Punkt auf zwei einfache Variablen abgebildet.

Einfache Variable: Sie enthält einen normalen numerischen Wert, der als ganze Zahl ausgegeben wird. Jeder geometrische Wert stellt letzlich eine einfache Variable dar. Der Name einer einfachen Variable besteht generell aus einem frei gewählten Namen, einem Punkt und einem beliebigen Buchstaben. Wird als Buchstabe ein `x` oder `y` angegeben, hält PARCON die Variable für die dereferenzierte x- bzw. y-Koordinate einer Punktvariablen.

Einfache Variablen können über die Eingabe von `set Variable Wert` mit einem Wert vorbelegt werden. Um dem Solver später die Änderung von Variablen bekanntzugeben, müssen zwei Schritte befolgt werden. Zuerst werden PARCON die Variablen mitgeteilt, deren Wert sich ändern soll: `new` $\{$ `Variablenliste` $\}^+$. Danach folgt eine Liste mit den entspechenden Werten: `val` $\{$ `Wert` $\}^+$. Die Anzahl der übergebenen Variablennamen und Werte muß gleich sein. Nach Abschluß der letzten Eingabe versucht PARCON, das Constraint-System unter Berücksichtigung der neuen Werte zu lösen. Sollte das System einen Widerspruch enthälten, wird eine Fehlermeldung ausgegeben. Andernfalls gibt PARCON alle geänderten Variablen mit ihren aktuellen Werten zurück. Punktvariablen werden als zwei einfache Variablen (p.x und p.y) behandelt.

3.4.2 Formeln

PARCON kann mit vier verschiedenen Formelarten umgehen, die nach einer fest definierten Syntax eingegeben werden müssen. Im folgenden werden diese Arten kurz beschrieben und an einem Beispiel erläutert. Stehen für einen Formelteil mehrere Varianten zur Verfügung, sind diese voneinander durch das Zeichen | getrennt.

Lineare Formel: Eine lineare Formel drückt einen linearen Zusammenhang zwischen mehreren einfachen Variablen aus. Dabei können die Relationen $<=$, $=$ und $>=$ verwendet werden. Die Relationen $<$ und $>$ werden dagegen nicht unterstützt. Die allgemeine Eingabe einer linearen Formel lautet:
`add` $\{$ `Wert Variable` $\}^+$ `<=|=|>= Wert`

Die Formel $a_x + 4b_x - 0.5b_y < 11$ würde man für PARCON folgendermaßen definieren:
`add 1 a.x 4 b.x -0.5 b.y <= 10`

Produkt: Durch Angabe eines Prouktes zweier einfacher Variablen, kann die Linearität verlassen werden. Werden die Zeichen =: als Relation benutzt, versucht PARCON beim Lösen der Gleichung vorrangig, die Variable der rechten Seite mit einem neuen Wert zu belegen. Die allgemeine Syntax lautet:
`add Variable Variable =|=: Variable|Wert`

Die Produktformel $a_x * b_x = 42$ lautet in der Schreibweise von PARCON:
`add a.x b.x = 42`

Euklidischer Abstand: Der euklidische Abstand drückt aus, daß ein Punkt einen bestimmten Abstand von einem anderen Punkt hat. Hier können neben einer einfachen Variable auch Punktvariablen angegeben werden:
`add Punktvariable R= Variable|Wert Punktvariable`

Die Tatsache, daß der Punkt P_1 vom Punkt P_2 23 Einheiten entfernt sein soll, wird folgendermaßen ausgedrückt:
`add p1 R= 23 p2`

Punktgleichheit: Soll die Gleichheit zweier Punkte in eine Formel gefaßt werden, wird folgende Variante angeboten:
`add Punktvariable P= Punktvariable`.

So verarbeitet PARCON beispielsweise für zwei Punkte P_1 und P_2 folgendes:
`add p1 P= p2`.

Für jede Formel kann der Benutzer eine Priorität festlegen. Dabei handelt es sich um einen Wert zwischen 0 und 9, wobei 0 für die höchste und 9 für die niedrigste Priorität steht. Auf diese Weise kann die Berechnungsreihenfolge angepaßt werden. Die Zahl wird ohne Leerzeichen hinter dem Schlüsselwort `add` angehängt.

Normalerweise sind alle eingegebenen Formeln UND-verknüpft. Das heißt, jede Formel muß unbedingt erfüllt sein, um das gesamte Constraint-System zu lösen. PARCON bietet außerdem eine ODER-Verknüpfung mehrerer hintereinander geschriebener Formeln an. Wird an das Schüsselwort `add` das Zeichen | angehängt, verknüpft PARCON diese Formel mit der zuletzt eingegebenen disjunkt. Durch Angabe von ODER-Verknüpfungen wird jedoch die Rechenzeit dramatisch erhöht. Ursache dafür ist folgende: PARCON kopiert intern für jede gefundene Disjunktion das gesamte Constraint-System und errechnet für jede dieser Kopien eine eigene Lösung. Liegen alle möglichen Lösungen vor, wird daraus die optimalste bestimmt. Dieser Vorgang kostet erheblich viel Zeit, so daß von der umfangreichen Benutzung disjunkter Beziehungen abzuraten ist.

Jede angemeldete Formel erhält von PARCON automatisch eine eindeutige Formelnummer. Durch Eingabe von `del Formelnummer` kann eine Formel aus dem Constraint-System entfernt werden.

3.4.3 Verarbeitung des Constraint-Systems

Nachdem der Constraint-Solver die verschiedenen Formeln erhalten hat, werden PARCON die Werte der einzelnen verwendeten Variablen mitgeteilt. Für jede Variable speichert PARCON den Zeitpunkt der letzten Änderung. So gibt es „alte“ und „junge“ Variablen. Ändern sich die Werte aufgrund von Benutzeraktionen, geht PARCON auf der Suche nach der optimalen Lösung davon aus, daß der Benutzer am Bildschirm so wenig Veränderungen wie möglich wünscht. Außerdem sollen natürlich die Werte, die als letztes geändert wurden, möglichst beibehalten werden. Der Constraint-Solver versucht daher zuerst, die „ältesten“ Variablen zu verändern. Wurde dadurch noch keine Lösung erzielt, müssen mitunter auch die zuletzt vom Benutzer geänderten Variablen wieder auf einen anderen Wert gesetzt werden.

Oft kommt es vor, daß innerhalb einer Formel alle Variablen gleich alt sind. In einem solchen Fall ändert PARCON zuerst die Variablen, die am weitesten links in der Formel stehen. Das betrifft alle Formelarten gleichermaßen. Nur bei einer Produktgleichung kann durch die Relation =: festgelegt werden, daß zuerst der rechten Seite ein neuer Wert zugewiesen wird.

3.4.4 Kommunikation

Der Constraint-Solver PARCON läßt sich ausschließlich über die Standardeingabe ansprechen und gibt die errechneten Variablenwerte sowie andere Ausgaben über die Standardausgabe zurück.

Bernhard Bardohl, Inhaber der Berliner Firma MSB Software, hat sich bereits 1997 mit PARCON beschäftigt. Er stellte der TU Berlin für das Projekt GENGED einen Daemon zur Verfügung, mit dem über TCP/IP eine Socket-Verbindung zu PARCON realisiert wurde. Unter Benutzung eines speziell entwickelten Protokolls kann aus Java heraus mit PARCON gearbeitet werden. In der Abbildung 3.8 ist eine schematische Darstellung der Verbindung des Alphabet-Editors mit PARCON dargestellt. Es können beliebig viele Verbindungen gleichzeitig existieren. Für jede Verbindung startet der Daemon einen separaten PARCON-Prozeß.

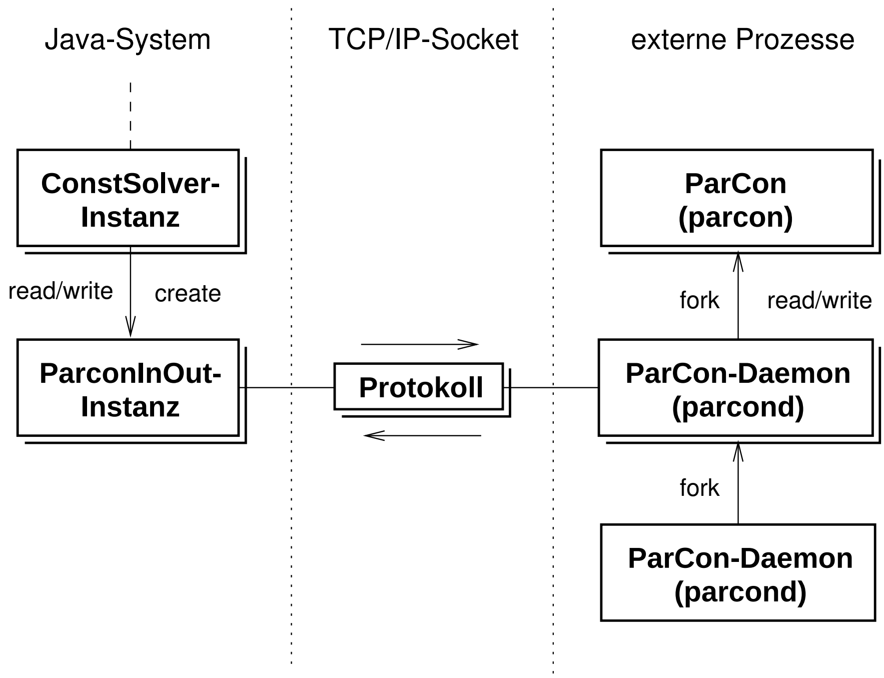


Abbildung 3.8: Anbindung des Alphabet-Editors an PARCON

Der Daemon für PARCON existiert als Solaris- und Linux-Version. Er ist in C geschrieben und könnte leicht an andere Plattformen angepaßt werden. Da PARCON selbst jedoch nur auf diesen beiden Betriebssystemen läuft, ist eine Portierung des Daemons zur Zeit unnötigt.

Die Kommunikation mit PARCON erfolgt mit Hilfe eines einfachen Protokolls, das vom PARCON-Daemon und zum Teil von PARCON selbst bereitgestellt wird. Es setzt auf den bereits vorhandenen Protokollen von Java[8] und auf TCP/IP auf. Diese Protokolle unterstützen alle nur einen Bytestrom als Daten. Aus diesem Grund besteht das Protokoll für PARCON aus zwei Teilen. Die Abbildung 3.9 gibt einen Überblick über die verschiedenen Protokollschichten. Die Darstellung ist an das OSI-Schichten-Modell[9] angelehnt.

Teil 1: Den zu übertragenen Daten wird ein generierter Header vorangestellt, der die Länge der Daten enthält. Durch diesen Mechanismus wissen wir beim Lesen der Daten aus einem InputStream[10], wieviele Daten wir zu erwarten und zu lesen haben. Der Header hat folgenden Aufbau: `.[00123]` – er beginnt mit einem Punkt, gefolgt von einer eckigen Klammer, und endet ebenso mit einer solchen. Zwischen den zwei angegebenen eckigen Klammern wird die Länge der folgenden Daten in Bytes angegeben. Diese Zahl kann maximal fünfstellig sein. Hat sie weniger als fünf Stellen, werden links die restlichen Stellen mit Nullen aufgefüllt. Die Headerlänge beträgt somit generell 8 Bytes.

Teil 2: In diesem Teil werden die Daten interpretiert, die PARCON liest oder schreibt. Der Daemon übersetzt die Daten, die PARCON ausgibt, in eine geeignete Protokoll-Sprache, die vom Alphabet-Editor ausgewertet wird. Die Daten, die aus dem Editor an PARCON geschickt werden, bleiben dagegen unverändert. In dem folgenden Abschnitt soll daher nur geklärt werden, wie PARCON auf verschiedene Eingaben reagiert.

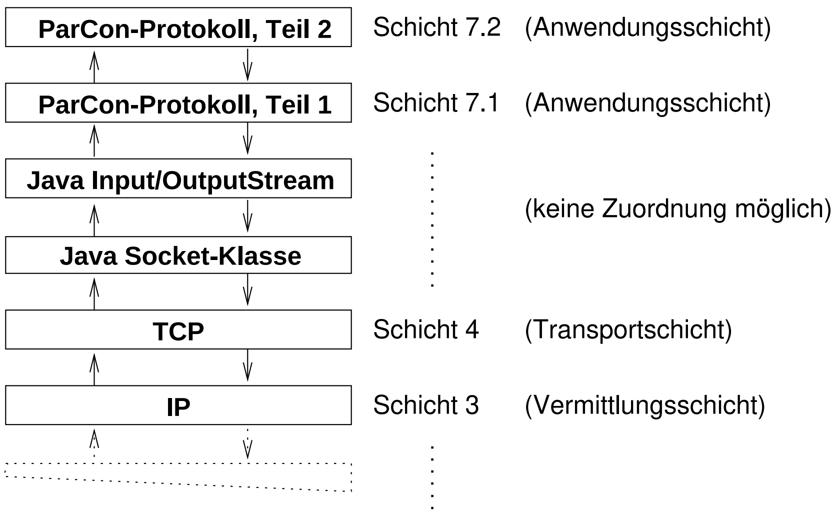


Abbildung 3.9: Ausschnitt eines angepaßten OSI-Schichten-Modells

[8] die internen Protokoll-Spezifikationen von Socket und Input/OutputStream sind uns nicht bekannt, aber auch nicht von Interesse

[9] Referenzmodell für Offene Kommunikationssysteme, in ISO7498 festgeschrieben

[10] wird von Java bereitgestellt

Wir unterscheiden drei Aktionen, die mit PARCON durchgeführt werden können:

Anmelden einer Formel: Nachdem die Formel an PARCON geschickt wurde, antwortet der Constraint-Solver mit einer Ausgabe, die mit dem Zeichen `$` beginnt. Direkt danach folgt eine Zahl. Ist sie größer null, handelt es sich um die Nummer, die PARCON der Formel zugewiesen hat. Ist sie dagegen `-1`, dann wurde die Formel nicht akzeptiert.

Löschen einer Formel: Nachdem an PARCON der Befehl zum Löschen einer Formel ergangen ist, bestätigt der Constraint-Solver den Erfolg der Aktion mit der Ausgabe `$0`. Kann die Formel nicht gelöscht werden, z.B. weil sie nicht existiert, wird der Fehlercode `$-1` zurückgegeben.

Übertragen neuer Daten und Lösen des Constraint-Systems: Das Übertragen der Daten erfolgt mit den Befehlen `new` und `val`, die separat an PARCON geschrieben werden. Ist das System bereits unlösbar, wird schon nach dem Befehl `new` die Ausgabe `$-1` zurückgegeben. Nach dem Befehl `val`, mit dem die konkreten Werte von Variablen übergeben werden, reagiert PARCON entweder mit `$-1`, wenn das System widersprüchlich ist und nicht gelöst werden kann, oder mit einer Liste von neuen Variablenwerten. Für letzteren Fall beginnt jede Zeile mit einer Tilde (~), direkt gefolgt von dem Variablennamen. Danach folgt ein Leerzeichen und eine Zahl – der Wert der angegebenen Variable. Es können weitere Zeilen mit neuen Variablenwerten folgen. Abgeschlossen wird die Ausgabe mit dem Wort `end`. Sollte die Berechnung von PARCON ungewöhnlich lange dauern (z.B. infolge von ODER-Verknüpfungen), werden in unregelmäßigen Abständen in eckigen Klammern die Anzahl der zur Zeit ermittelten möglichen Lösungen zurückgegeben, z.B. `[7100]`.

Die Ausgabe `$0` erscheint übrigens immer dann, wenn eine Leerzeile an PARCON übermittelt wurde. Dieses Verhalten benutzen wir, um zu testen, ob eine Verbindung zu PARCON besteht oder ob sie gestört ist.

Kapitel 4

Systemstruktur

In diesem Kapitel wollen wir einen kurzen Überblick über die Struktur des Systems geben. Es enthält mit dem Stand dieser Diplomarbeit sowohl Basiskomponenten als auch den Alphabet-Editor. Der Editor selbst ist wiederum unterteilt in verschiedene Systeme, auf die wir ebenfalls eingehen.

4.1 Basis und Komponenten

Es wurde versucht, die einzelnen Klassen für den Entwurf des Gesamtsystems nach Aufgabengebieten zu ordnen. Dadurch sind verschiedene Packages entstanden, die zum Teil voneinander abhängen. Der Package-Begriff der Unified Modeling Language, die wir als Modellierungstechnik benutzen, kann direkt in die Programmiersprache Java umgesetzt werden. Die Abbildung 4.1 gibt einen Überblick über die Struktur des Systems. Einige Packages werden von einer gepunkteten Linie umschlossen. Viele der Klassen in diesen Packages sind unabhängig von den Klassen des konkreten Alphabet-Editors, so daß sie auch von Komponenten genutzt werden können, die später das System erweitern. Die verschiedenen Pfeile machen Aussagen darüber, welche Beziehungen zwischen den Packages bestehen. Besteht eine Beziehung zwischen zwei Packages, dann gilt sie auch für alle Subpackages.

Unter Nutzung des Model-View-Controller-Patterns (siehe Abschnitt 3.2.5) wurde das System in mehrere Teile aufgeteilt. Das *Modell* stellt mit dem Package `genged.language` die Basis des gesamten Systems dar. Es enthält sämtliche Datenstrukturen des Alphabets sowie einen Mechanismus, der interessierte Komponenten über Änderungen informiert. Hier kommt vehement das Listener-Pattern (siehe Abschnitt 3.2.2) zum Einsatz. Die Konzeption und Implementierung dieses Modells stellt einen wesentlichen Bestandteil der vorliegenden Diplomarbeit dar. Verschiedene *Ansichten* auf die Daten sowie *Kontrollstrukturen* bilden den zweiten wichtigen Teil des Systems. Alle verschiedenen Teil-Editoren befinden sich in Subpackages vom Package `genged.alphabeteditor`. Ein weiterer wichtiger Bestandteil des Gesamtsystems ist das Package `genged.constsolver`. Es implementiert die Anbindung an den Constraint-Solver PARCON und funktioniert völlig unabhängig von allen anderen Komponenten. Dazu bietet es die Funktionalitäten zur Anwendung einer High-Level-

Constraint-Sprache, deren Entwicklung einen weiteren Gegenstand dieser Arbeit darstellt. Die restlichen Packages innerhalb von GENGED bilden eine Art Bibliothek für verschiedenste andere Funktionalitäten. Sie können wie das Modell und die Constraint-Solver-Anbindung von den zukünftigen Erweiterungen des Alphabet-Editors genutzt werden.

In vielen Packages befindet sich ein Subpackage mit dem Namen `commands`. Dieses Package enthält verschiedene Befehlsklassen, die die Funktionalität des Systems, speziell von dem jeweils übergeordneten Package, kapseln. Sie dienen der Umsetzung des Command-Processor-Patterns (siehe Abschnitt 3.2.4).

4.2 Packages

In diesem Abschnitt sollen nun kurz die Aufgaben aller verwendeten Packages aus der Strukturübersicht vorgestellt werden, mit Ausnahme der `commands`-Packages. Wir verweisen jeweils auf das Kapitel, in dem näher auf das jeweilige Package eingegangen wird.

`genged`: Dieses Package stellt die Wurzel der ganzen Struktur dar und enthält eine Klasse `GenGEd`, die viele statische Attribute besitzt, welche verschiedene Eigenschaften des Systems beschreiben. Fast alle Klassen des Systems besitzen eine Referenz auf diese Klasse. Die Beschreibung erfolgt in Kapitel 6.

`genged.alphabeteditor`: Hier wird das Rahmenwerk des Alphabet-Editors implementiert. Dazu gehört unter anderem eine Klasse zur einheitlichen Verarbeitung von Benutzeraktionen. Alle Klassen werden ebenfalls im Kapitel 6 näher vorgestellt.

`genged.alphabeteditor.symeditor`: Dieses Package implementiert den Symbol-Editor. Dazu gehören die Kontrollstrukturen, eine Zeichenfläche sowie Adapterklassen für die Strukturansicht des Editors. Diese Klassen werden in Kapitel 8 erläutert.

`genged.alphabeteditor.coneditor`: Hier wird der Verbindungs-Editor implementiert. Auch er wird in Kapitel 8 behandelt.

`genged.alphabeteditor.testsystem`: Das Test-System zum Zeichnen von Diagrammen gleicht im Package-Aufbau dem letzten und vorletzten Package. Es wird in Kapitel 9 beschrieben.

`genged.alphabeteditor.consteditor`: Dieses Package existiert eigentlich noch nicht. Es soll später einen Constraint-Editor implementieren, mit dem auf textueller Basis High-Level-Constraints definiert werden können.

`genged.alphabeteditor.helpsystem`: Auch das Online-Hilfe-System existiert noch nicht. Das Package `helpsystem` wird später einmal Funktionen zum Aufbau und zur Bedienung einer Online-Hilfe bieten. Es ist geplant, das im Anhang dieser Diplomarbeit befindliche Handbuch als HTML-Variante einzubinden.

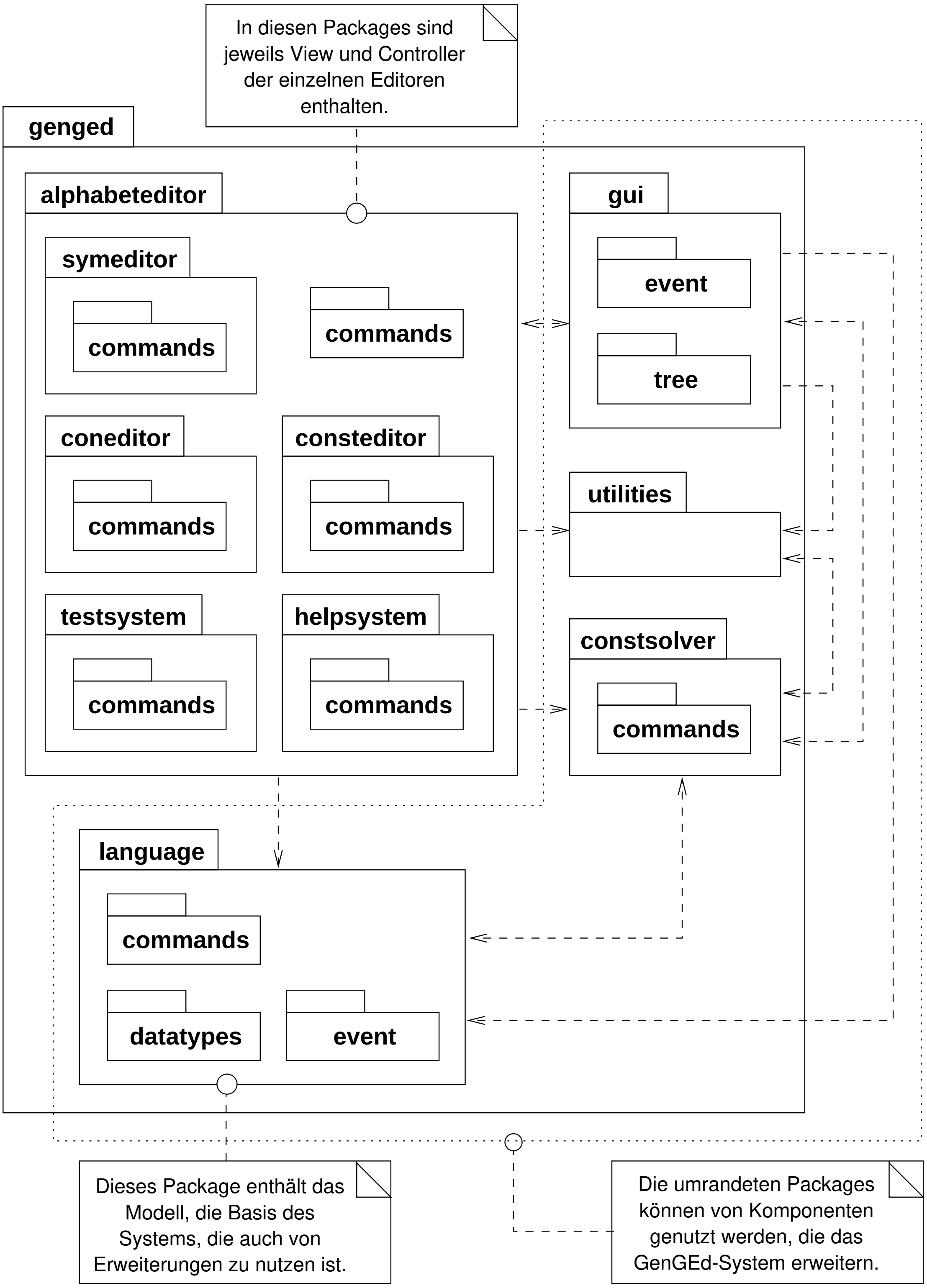


Abbildung 4.1: Struktur des Alphabet-Editors und seiner Komponenten

`genged.language`: Dieses Package bildet den Kern des Systems. Hier werden die Bestandteile eines Alphabets bzw. Diagramms implementiert. Außerdem wird die gesamte Funktionalität zum Umgang mit Alphabeten und Diagrammen zur Verfügung gestellt. Wir beschäftigen uns ausgiebig in Kapitel 5 mit dem Aufbau des Alphabets. Mit Diagrammen setzen wir uns in Kapitel 9 auseinander.

`genged.language.datatypes`: Möchte der Anwender des Alphabet-Editors das System erweitern, indem er eigene Datentypen programmiert, dann werden diese Klassen in dem Package `datatypes` abgelegt. Dort befinden sich außerdem die Standardklassen zum Thema Datentypen. Auch Datentypen werden in Kapitel 5 behandelt. Außerdem verweisen wir an dieser Stelle auf das letzte Kapitel des Benutzerhandbuchs, welches sich im Anhang dieser Arbeit befindet.

`genged.language.event`: In diesem Package werden die Mechanismen des Listener-Patterns für das Alphabet umgesetzt. Es werden u.a. verschiedene Interfaces angeboten, welche von Klassen implementiert werden können, die an bestimmten Änderungen des Alphabets interessiert sind. Die Nutzung des Mechanismus wird in Kapitel 5 erläutert.

`genged.gui`: Die Benutzeroberfläche des Alphabet-Editors ist trotz der verschiedenen Teil-Editoren einheitlich gestaltet. Alle Komponenten, die Klassen aus den Packages `java.awt` oder `javax.swing` erweitern, sind hier abgelegt. Eigene GUI-Komponenten ergänzen das Angebot. Teile dieses Packages werden in Kapitel 6 aufgegriffen.

`genged.gui.event`: In diesem Package sind alle Klassen und Interfaces enthalten, die den Benachrichtigungs-Mechanismus von GUI-Komponenten implementieren. Viele dieser Interfaces werden in Kapitel 6 erwähnt.

`genged.gui.tree`: Einen wichtigen Teil der Benutzeroberfläche stellt die Strukturansicht dar. Zu dieser Ansicht gehören viele Klassen, die aufgrund ihrer Anzahl separat in dem Package `tree` implementiert wurden. Kapitel 6 stellt die Konzepte ausführlich vor.

`genged.utilities`: Das `utilities`-Package beinhaltet viele Hilfsklassen, die meist unabhängig von anderen Klassen funktionieren, aber von mehreren Komponenten benutzt werden. Dazu gehört beispielsweise eine Klasse, die die Mehrsprachigkeit des Systems unterstützt. Außerdem wird hier der Undo-Mechanismus der Swing-Klassen erweitert und an eigene Anforderungen angepaßt. Diese Klassen gehören zu den allgemeinen Mechanismen des Systems, mit denen wir uns in Kapitel 6 auseinandersetzen.

`genged.constsolver`: Dieses Package realisiert die Anbindung an den externen Constraint-Solver PARCON und stellt Mechanismen zur Anwendung einer High-Level-Constraint-Sprache zur Verfügung. Diesen beiden Themen ist das gesamte Kapitel 7 gewidmet.

4.3 Editoren-Management

Der Alphabet-Editor, dessen Konzipierung und Implementierung Aufgabe dieser Arbeit ist, unterteilt sich in mehrere Teil-Editoren. Sie werden ab Kapitel 8 erläutert. Der Alphabet-Editor selbst bildet nur eine Art Rahmenwerk, um diese Teil-Editoren zu verwalten. Die dazu notwendigen Klassen sind im Package `genged.alphabeteditor` implementiert.

Durch Anwendung des Model-View-Controller-Pattern haben wir das Gesamtsystem in verschiedene Modelle (Model), Ansichten (View) und Kontrollstrukturen (Controller) aufgeteilt. Views und Controller zu einem bestimmten Modell sind jeweils die Bestandteile eines einzelnen Teil-Editors. Um die verschiedenen Editoren betreffs der Oberfläche einheitlich zu gestalten, benutzen alle eine bestimmte Aufteilung des Arbeitsfensters. Sie ist in Abbildung 4.2 dargestellt. Menü- und Symbolleiste, Strukturansicht sowie die eigentliche Arbeitsfläche müssen von jedem einzelnen Teil-Editor bereitgestellt werden. Die Klasse `AlphabetEditor` ist für die korrekte Darstellung – das Layout – und für das Umschalten zwischen den Editoren zuständig. Die Abbildung 4.3 zeigt ein Klassendiagramm, das den Zusammenhang der Klassen `AlphabetEditor` und `AbstractEditor` sowie des Interfaces `EditorManager` veranschaulicht. Im folgenden werden die wichtigsten Methoden der Komponenten erläutert.

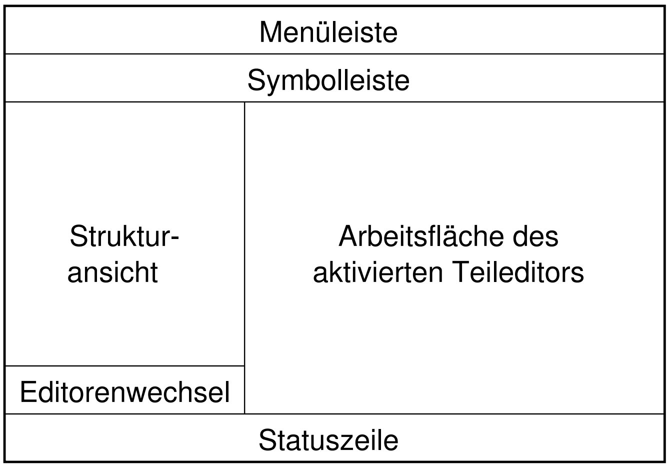


Abbildung 4.2: Struktureller Aufbau des Arbeitsfensters vom Alphabet-Editor

Interface `EditorManager`: In diesem Interface werden allgemeine Methoden definiert, die zur Verwaltung mehrerer Teilsysteme – der einzelnen Editoren – benötigt werden. Diese müssen jeweils von der Klasse `AbstractEditor` abgeleitet sein, die beispielsweise eine Methode `getName()` für die Identifikation eines Editors zur Verfügung stellt. Mit der Methode `registerEditor()` kann ein Editor beim Manager angemeldet werden. Ist ein Editor registriert, kann er mit den `getEditor()`-Methoden abgerufen werden. Es wird davon ausgegangen, daß die Editoren in einer indizierten Liste, z.B. einem Vector, abgelegt sind. Deshalb kann ein Editor entweder aufgrund seines Indizes oder seines eindeutigen Namens an-

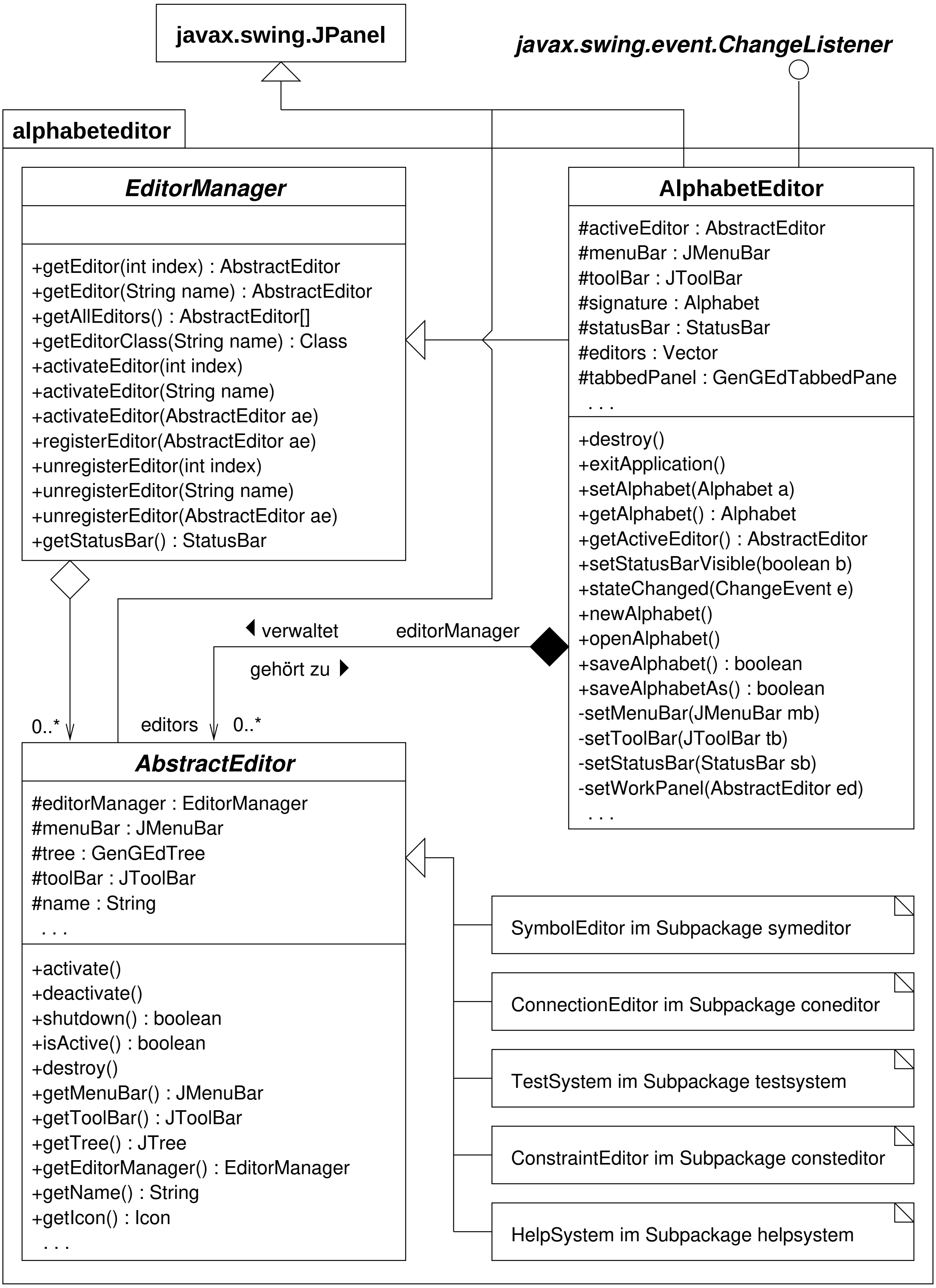


Abbildung 4.3: Klassendiagramm zur Organisation des Editoren-Managements

gesprochen werden. Durch Aufruf der Methode `getAllEditors()` werden alle registrierten Editoren zurückgegeben. Ein Editor kann im Gesamtsystem durch Anwenden der `activateEditor()`-Methode aktiviert werden. Alle anderen Editoren werden dadurch deaktiviert. Die gesamte grafische Benutzeroberfläche sollte durch diese Methode auf den aktivierten Editor umgeschaltet werden. Das betrifft die Menü- und Symbolleiste, die Strukturansicht und die Arbeitsfläche des Gesamtsystems. Alle Eingaben, die der Benutzer macht, werden immer von dem gerade aktivierten Editor entgegengenommen und bearbeitet. Nur dieser Editor ist sichtbar, alle anderen werden „versteckt“. Mit Hilfe der `unregisterEditor()`-Methoden wird ein Editor wieder abgemeldet.

Klasse `AlphabetEditor`: Diese Klasse erweitert die Klasse `JPanel` aus den Swing-Klassen von Java und implementiert das Interface `EditorManager`. Sie enthält neben den Methoden des Interfaces zahlreiche Funktionalitäten zum Verwalten der grafischen Benutzeroberfläche. Außerdem verwaltet diese Klasse das Alphabet, das im Symbol- und Verbindungs-Editor bearbeitet werden kann. Da die gesamte Benutzeroberfläche auf einem Panel aufbaut, ist der Alphabet-Editor flexibel einsetzbar und damit offen für spätere Erweiterungen und Anpassungen. Im *Konstruktor* werden verschiedene Initialisierungen vorgenommen. So werden u.a. die Struktur der Benutzeroberfläche aufgebaut, die verschiedenen Editoren erzeugt und der Symbol-Editor aktiviert. Außerdem wird ein neues Alphabet generiert. Die Abbildung 4.4 gibt einen Überblick über die Verschachtelung der verschiedenen eingesetzten GUI-Komponenten.

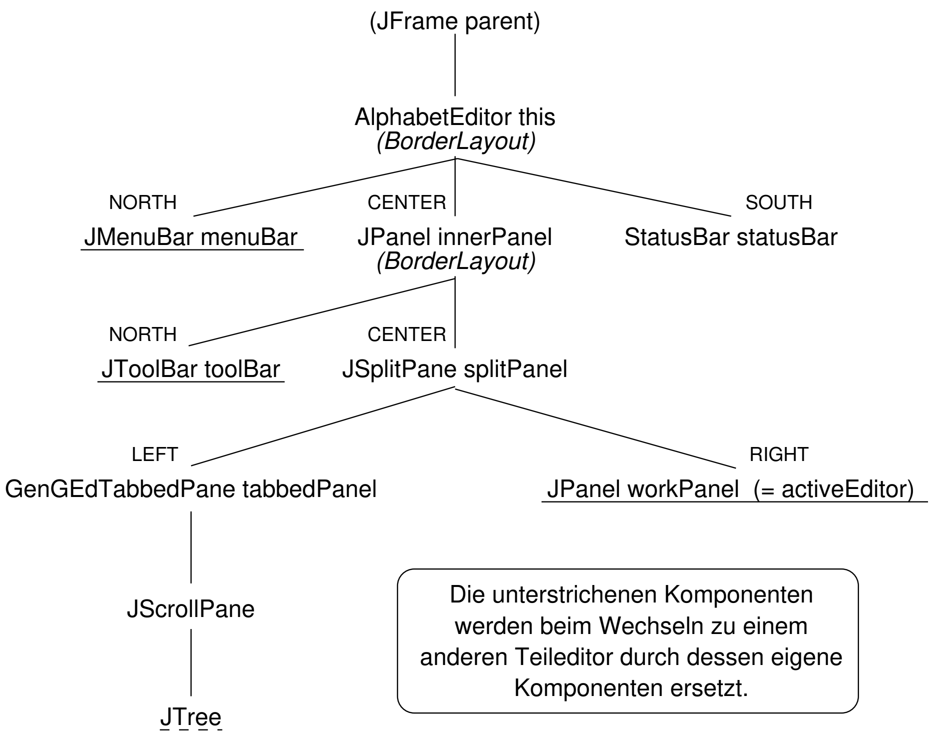


Abbildung 4.4: GUI-Komponenten der Benutzeroberfläche des Alphabet-Editors

Beim Registrieren der Editoren werden die Strukturansichten (`JTree`), die in allen Teil-Editoren enthalten sind, einem Tabulator-Panel hinzugefügt. Klickt der Benutzer mit der Maus auf ein Register des Tabulator-Panels, wird ein Ereignis (`ChangeEvent`) ausgelöst, das den Prozeß zum Aktivieren eines Teil-Editors startet. Durch das Ereignis wird die Methode `stateChanged()` aufgerufen, die wiederum die Methode `activateEditor()` benutzt. Diese sorgt dafür, daß Menü- und Symbolleiste des aktivierten Teil-Editors in das Panel des Alphabet-Editors übernommen werden. Außerdem wird das Editor-Panel der rechten Seite des Split-Panels hinzugefügt. Die Strukturansicht erscheint im Tabulator-Panel automatisch.

Klasse `AbstractEditor`: Von dieser Klasse sind alle Teil-Editoren abgeleitet, die von den Subpackages von `genged.alphabeteditor` implementiert werden. Jeder dieser Editoren enthält neben seinen spezialisierten Attributen eine eigene Menü- und Symbolleiste sowie eine Strukturansicht. Außerdem wird eine Referenz auf den Editor-Manager gehalten, bei dem der Editor registriert ist. Jeder Editor erhält einen eindeutigen Namen, der an den Konstruktor übergeben wird. Aktiviert der Benutzer einen Editor, dann wird in diesem Editor die Methode `activate()` aufgerufen. Mit ihr kann ein initialer Zustand hergestellt werden. Beim Deaktivieren erfolgt ein Aufruf von `deactivate()`. Die Methode `isActive()` gibt Auskunft darüber, ob ein Editor gerade aktiviert ist. Weitere Methoden dienen der Übergabe von GUI-Elementen an den Alphabet-Editor (`getMenuBar()`, `getToolBar()` und `getTree()`). Die Methode `shutdown()` wird vom Alphabet-Editor in jedem Teil-Editor aufgerufen, wenn der Benutzer versucht, das System zu beenden. Jeder der Editoren kann durch ein Veto den Shutdown-Prozeß stoppen. Dazu gibt er einfach `false` zurück. Diese Methode führt bei einigen Editoren eine Sicherheitsabfrage aus, wenn bearbeitete Daten noch nicht gesichert wurden.

Kapitel 5

Datenstrukturen eines Alphabets

In diesem Kapitel wollen wir uns intensiv mit dem Aufbau eines Alphabets befassen. Das Alphabet bildet das Modell – die Basiskomponente – des gesamten Systems. Sowohl der Symbol- als auch der Verbindungs-Editor arbeiten auf diesem Modell. Die Anforderungen, die an ein Alphabet gestellt werden, haben wir bereits in Kapitel 2 formuliert. Hier soll nun die Umsetzung in konkrete Datenstrukturen vorgestellt werden.

5.1 Überblick über verschiedene Bestandteile

Die Bestandteile eines Alphabets gliedern sich in zwei grundsätzliche Kategorien: Es existieren sichtbare grafische Objekte und unsichtbare nicht-grafische Verbindungen zwischen diesen Objekten.

5.1.1 Grafische Objekte

Die einfachsten sichtbaren Bestandteile eines Alphabets sind die *Primitiven*. Dabei handelt es sich um vektororientierte Grafiken wie z.B. einen Kreis oder ein Rechteck. Jede Art einer Primitive wird durch eine eigene Klasse repräsentiert. Mehrere Primitiven lassen sich zu einer *Gruppe von Primitiven* zusammensetzen. Diese Funktionalität wurde nicht explizit in den Anforderungen formuliert. Dennoch ist sie hilfreich, da eine solche Gruppe durch den Anwender wie ein einziges Objekt behandelt werden kann. Es können auch weitere Untergruppen gebildet werden.

Eine beliebige Kombination aus Primitiven und Gruppen bildet die Symbolgrafik eines *Symbols* – sie stellt eine spezielle Gruppe dar. Für veränderliche Daten können *Datentypen* erzeugt werden. Sie werden wie eine Primitive behandelt, haben jedoch denselben Stellenwert wie ein Symbol.

5.1.2 Verbindungen

Neben Symbolen und Datentypen gibt es Verbindungen zwischen den grafischen Objekten. Diese Verbindungen drücken sich in einem bestimmten Layout der beteiligten

Grafiken aus. Wir unterscheiden Verbindungen, die innerhalb eines Symbols zwei oder mehrere Primitiven in Beziehung setzen, und Verbindungen, die zwischen den Primitiven von Symbolen oder Datentypen aufgebaut werden. Eine grafische Verbindung wird im folgenden als *Operation* bezeichnet. Zu jeder Operation gehören beliebig viele *Constraints*, die konkret das Layout der an der Operation beteiligten Primitiven und damit auch das Layout eines Symbols oder Datentyps beeinflussen.

In den Anforderungen, die in Kapitel 2 aufgestellt wurden, definierten wir für jede Operation ein Source- und Target-Objekt – ein Ausdruck für eine Beziehung zwischen Symbolen oder Datentypen. Diese Beziehung bezeichnen wir als *externe Operation*. Source- und Target-Objekt sind in diesem Fall verschiedene Objekte. Eine *interne Operation* arbeitet dagegen auf ein und demselben Source- und Target-Objekt und definiert Constraints zwischen dessen Primitiven. Diese Design-Entscheidung wurde getroffen, weil die Eigenschaften einer internen und externen Operation völlig identisch sind. Zu beiden Arten gehören Constraints, die auf den Primitiven des zugewiesenen Source- und Target-Objekts definiert sind.

Die Abbildung 5.1 zeigt beispielhaft die hierarchische Zusammensetzung der sichtbaren und unsichtbaren Bestandteile eines Alphabets anhand der gerade erläuterten Begriffe.

5.2 Templates und Instanzen

Jeder der aufgezählten Bestandteile stellt innerhalb des Alphabets lediglich eine Art Schablone – ein Template – dar, mit der später in einem Grafik-Editor Diagramme gezeichnet werden sollen. Dazu müssen Symbole, Gruppen, Primitiven, Datentypen, Operationen und Constraints instanziiert werden können. Um diesem Umstand gerecht zu werden, statten wir jede Klasse dieser Objekte mit folgenden Attributen und Methoden aus:

Attribute:

- `public final static int TEMPLATE = 0;`
  Diese Konstante wird für das Attribut `kind` verwendet, wenn es sich bei dem Objekt um ein Template handelt.
- `public final static int INSTANCE = 1;`
  Diese Konstante wird für das Attribut `kind` verwendet, wenn es sich bei dem Objekt um eine Instanz handelt.
- `protected int kind;`
  Dieses Attribut gibt Auskunft darüber, ob dieses Objekt ein Template (`TEMPLATE`) oder eine Instanz (`INSTANCE`) ist. Jedes Objekt, das mit seinem Konstruktor erzeugt wurde, ist automatisch ein Template. Eine Instanz von einem Template wird durch Aufruf der Methode `createInstance()` abgeleitet.

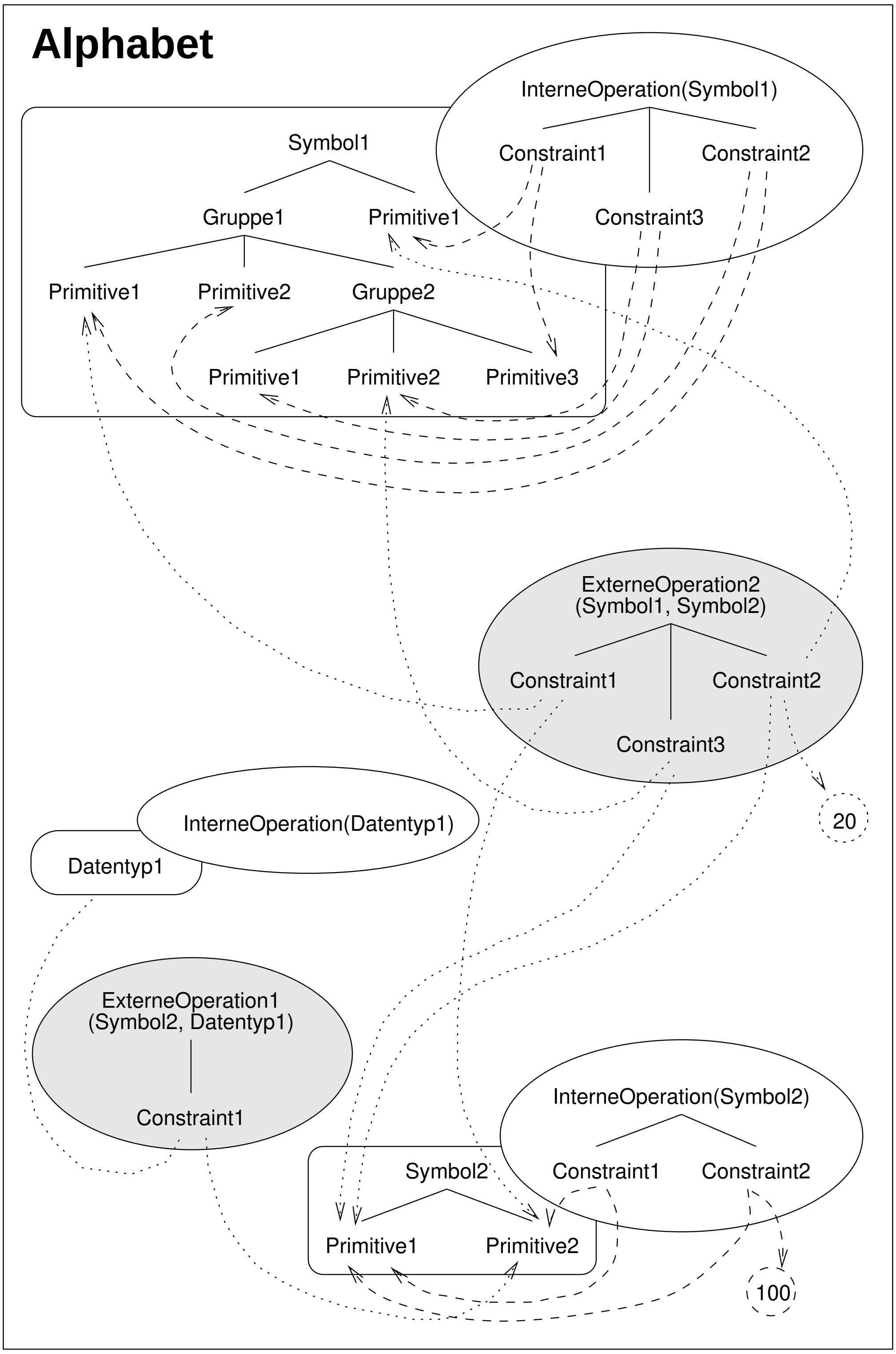


Abbildung 5.1: Hiearchische Zusammensetzung eines Alphabets

- `protected Klasse template;`
  Handelt es sich bei diesem Objekt um eine Instanz, enthält dieses Attribut eine Referenz auf das Template-Objekt, von dem dieses Objekt abgeleitet wurde. Andernfalls hält es eine Referenz auf sich selbst.

Methoden:

- `public boolean isInstance();`
  Gibt `true` zurück, wenn es sich bei diesem Objekt um eine Instanz handelt.
- `public boolean isTemplate();`
  Gibt `true` zurück, wenn es sich bei diesem Objekt um ein Template handelt.
- `public Klasse getTemplate();`
  Gibt im Fall einer Instanz deren Template-Objekt zurück, andernfalls sich selbst (`this`).
- `public Klasse createInstance(...);`
  Wird diese Methode innerhalb eines Template-Objekts aufgerufen, wird daraus ein Instanzen-Objekt abgeleitet und zurückgegeben. Die Parameter dieser Methode variieren. Ist das Objekt in eine Hierarchie von Objekten integriert, muß oft das später übergeordnete Instanzen-Objekt als Parameter angegeben werden.

Eine Instanz ist ein eigenständiges Objekt, das nach dem Ableiten in der Regel völlig identisch ist mit seinem Template – abgesehen von den oben erwähnten nichtstatischen Attributen. Im Falle einer Grafik sprechen wir auch von einem Default-Layout. Da eine Instanz eigenständig ist, könnte sie jedoch unabhängig von ihrem Template verändert werden. Das betrifft beispielsweise die Position oder Größe der Primitiven innerhalb eines Symbols.

Was passiert jedoch mit den Instanzen eines Alphabets, wenn sich deren Templates nachträglich ändern? Die Instanzen wissen nichts von dieser Änderung, da ein Template keine Referenzen auf seine Instanzen besitzt. Wenn sich die Templates und auch die dazugehörigen Instanzen während der Änderung im Arbeitsspeicher befinden, könnte durch einen unabhängigen Update-Mechanismus sichergestellt werden, daß Änderungen automatisch übernommen werden. Sind jedoch die Instanzen auf einem Datenträger gespeichert, funktioniert dieser Ansatz nicht. In einem solchen Fall benötigen wir einen Algorithmus, der während eines Ladevorgangs überprüft, welche Änderungen stattgefunden haben, und die Instanzen gegebenenfalls anpaßt. Dieser Algorithmus wurde bisher noch nicht implementiert. Wir setzen uns jedoch ansatzweise im Kapitel 10 mit diesem Problem auseinander.

5.3 Primitive Grafiken

Die Primitiven sind die Grundbestandteile von jedem Symbol. Sie stellen genau ein grafisches Grundobjekt zur Verfügung. Alle grafischen Objekte, einschließlich des Symbols und des Datentyps, sind von der abstrakten Klasse `GraphicObject` abgeleitet. Diese Klasse implementiert bereits alle Methoden, die für jegliche vektororientierte Grafiken benötigt werden. Die Abbildung 5.2 stellt die Hierarchie aller von der Klasse `GraphicObject` abgeleiteten Komponenten dar.

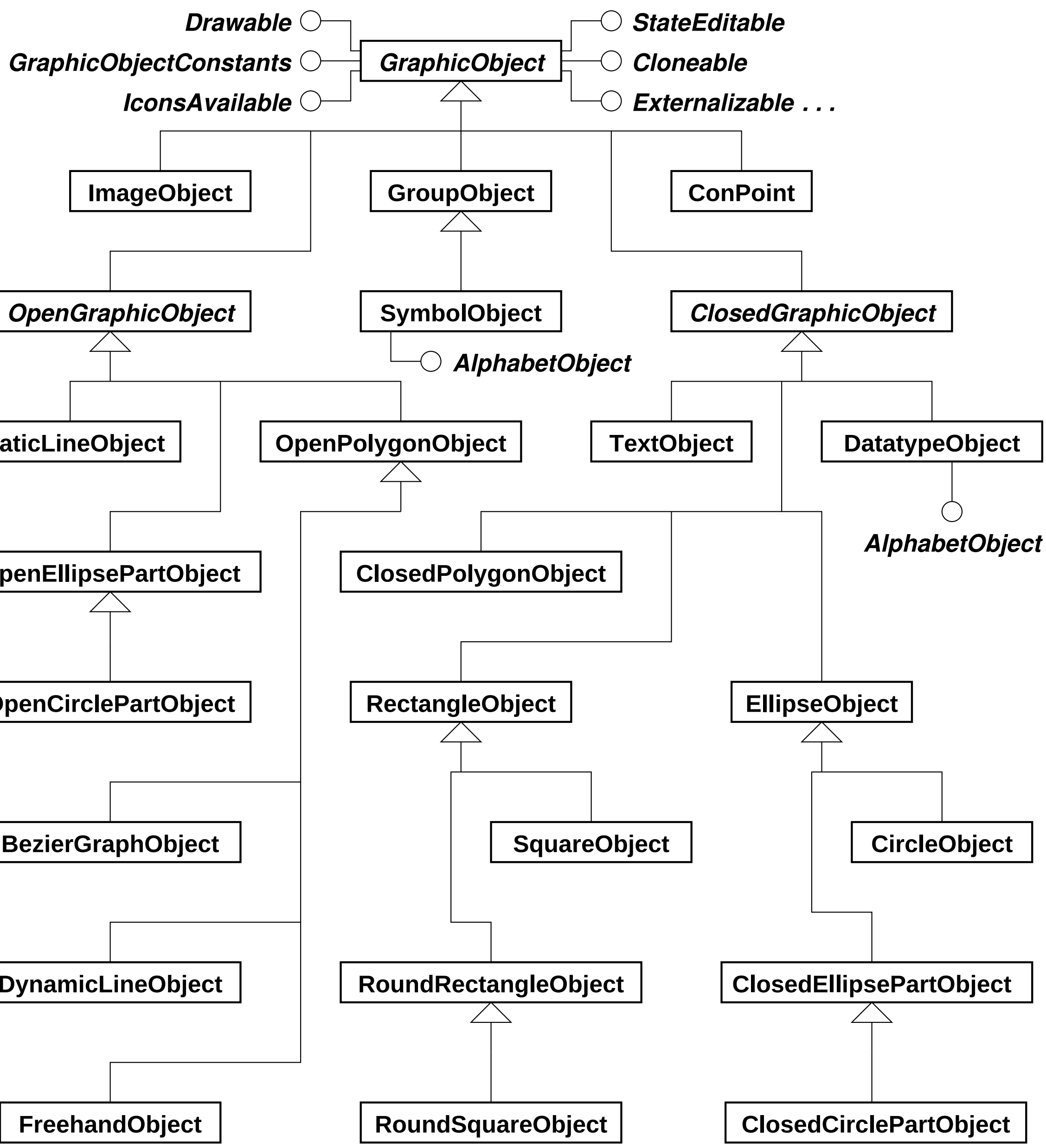


Abbildung 5.2: Klassendiagramm: `GraphicObject`, Ableitungen

Da Java keine Mehrfachvererbung unterstützt, mußten in einigen Klassen gleiche Attribute implementiert werden. Beispielsweise betrifft das die Klassen `OpenPolygonObject` und `ClosedPolygonObject`. Es wäre zweckmäßiger gewesen, die Klasse `ClosedPolygonObject` von den Klassen `OpenPolygonObject` *und* `ClosedGraphicObject` abzuleiten, da diese bereits alle Methoden besitzen, die in `ClosedPolygonObject` benötigt werden.

5.3.1 Allgemeine Funktionalitäten

Die Klasse `GraphicObject` implementiert das Interface `GraphicObjectConstants`, das eine Sammlung von vielen Konstanten darstellt, die von den verschiedenen Primitiven genutzt werden.

Als interne Einheit für Objektpositionen, Ausmaße eines Objekts oder Distanzen wird 1/100pt benutzt. Sie bietet eine zufriedenstellende Auflösung und wird durch die Klasse `Measure` repräsentiert. Eine Referenz findet sich in der Konstante `MEASURE`, die in dem Interface `GraphicObjectConstants` definiert wurde. Diese Klasse stellt viele verschiedene Einheiten sowie Methoden zu beliebigen Umrechnungen zur Verfügung. Da ein grafisches Objekt letztlich auf den Bildschirm gezeichnet wird, benötigen wir eine Umrechnung von 1/100pt nach Pixel. Sie wird unter Einbeziehung eines Zoom-Faktors von der geschützten Methode `getPixel()` ausgeführt.

Jede Primitive besitzt ein übergeordnetes Objekt, das in dem Attribut `parent` gespeichert wird. Einzige Ausnahmen: Die Klassen `SymbolObject` und `DatatypeObject` sind die Wurzeln in der späteren Objekt-Hierarchie des Alphabets. Bei ihnen ist `parent` mit dem Wert `null` belegt. Dafür existiert ein Verweis auf das Alphabet. Über die Methoden `setParent()` und `getParent()` kann auf das Attribut `parent` zugegriffen werden.

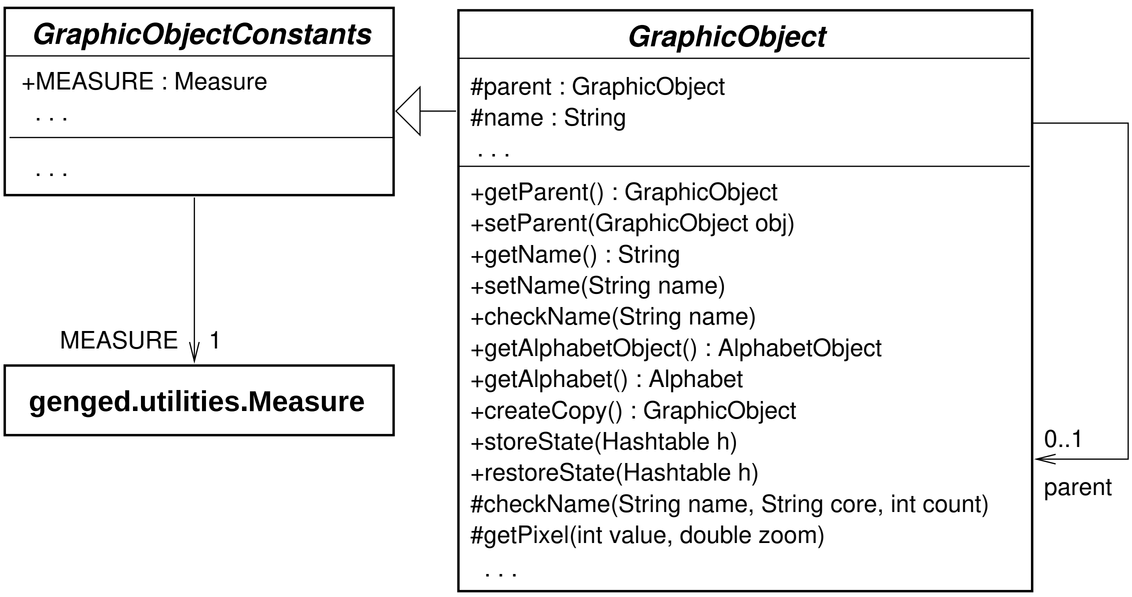


Abbildung 5.3: Klassendiagramm: `GraphicObject`, Allgemein

Der Name einer Primitive kann mit der Methode `setName()` verändert und mit `getName()` abgefragt werden. Er muß jedoch innerhalb des Kontextes eindeutig sein. Das bedeutet, daß alle Kindobjekte eines übergeordeten Objekts (`parent`) verschiedene Namen haben müssen. Beim Setzen des Namens wird er automatisch mit Hilfe der Methode `checkName()` auf Eindeutigkeit geprüft. Existiert der Name bereits, wird eine Zahl an den Namen angehängt, die so lange inkrementiert wird, bis er eindeutig ist. Neben dem Namen kann noch ein sogenannter Identifier abgefragt werden. Dabei handelt es sich um den Pfad vom Symbol über eventuell vorhandene Gruppen bis hin zu einer speziellen Primitive. Der Identifier besteht aus den Namen der verschiedenen übergeordneten Objekte, die jeweils durch einen Punkt voneinander getrennt sind. Da alle Objekte eindeutige Namen innerhalb ihres Kontextes besitzen, ist der Identifier im gesamten Alphabet eindeutig. Ein Name lautet beispielsweise `Rechteck5`, der dazugehörige Identifier `Sequence.Gruppe2.Rechteck5`.

Wurde eine Primitive bereits von einem Template abgeleitet, sieht der Name etwas anders aus. Dem Templatenamen wird ein neuer eindeutiger Name für die Instanz (`Inst<Zahl>`), gefolgt von einem Doppelpunkt, vorangestellt. Entsprechend angepaßt wird auch der Identifier. Eine Instanz könnte z.B. `Inst2:Rechteck5` heißen, der Identifier beispielsweise `Inst2:Sequence.Inst2:Gruppe2.Inst2:Rechteck5`. Um den Identifier des dazugehörigen Templates zu erhalten, kann die Methode `getTemplateIdentifier()` genutzt werden.

Bei der Wurzel des Pfades, der durch den Identifier bezeichnet wird, handelt es sich generell um ein Objekt, welches das Interface `AlphabetObject` implementiert. In Frage kommen ein Symbol oder ein Datentyp. Die Methode `getAlphabetObject()` gibt daher ein Objekt der Klasse `SymbolObject` oder `DatatypeObject` zurück. Eine Referenz auf das Alphabet selbst liefert die Methode `getAlphabet()`.

Mit der Methode `createCopy()` wird eine originalgetreue Kopie einer Primitive erzeugt. Vor den Namen wird der Prefix `CopyOf` angehängt. Eine solche Kopie sollte allerdings nur für temporäre Zwecke genutzt werden.

Wir benutzen innerhalb des Alphabet-Editor einen Undo-Mechanismus, der es erlaubt, Veränderungen von Objekten rückgängig zu machen. Die Klasse `GraphicObject` implementiert die Methoden `storeState()` und `restoreState()` des Interfaces `StateEditable`. In diesen Methoden wird *ein Teil* des Zustands in eine Hashtable gespeichert bzw. wieder rekonstruiert.

Die Abbildung 5.3 zeigt Beziehungen der Klasse `GraphicObject` auf und stellt eine kleine Auswahl von Attributen und Methoden dar. In den folgenden Abschnitten werden weitere Abbildungen dieser Klasse folgen, die jeweils nur die Methoden enthalten, die in einem Abschnitt erläutert werden.

5.3.2 Bounding-Box und Koordinatenberechnungen

Die Klasse `GraphicObject` stattet jede Grafik mit einer Bounding-Box aus, die durch zwei Punkte beschrieben wird. `p1` liegt in der linken oberen Ecke des Rechtecks, `p2` rechts unten. Diese Bounding-Box existiert unabhängig von der Form einer Grafik. Sie bietet die Möglichkeit, alle Objekte grundsätzlich gleich zu behandeln, selbst wenn einige Grafiken noch zusätzliche Möglichkeiten bieten. Dadurch können Methoden wie `setLocation()`, `setSize()` und `setBounds()`, die in einem der folgenden Abschnitte näher erläutert werden, für alle Primitiven gleichermaßen angewendet werden. Ändert sich die Form der Grafik, wird die Methode `updateBoundingBox` aufgerufen, welche die Punkte der Bounding-Box aktualisiert.

Die erwähnten Punkte `p1` und `p2` enthalten allerdings nicht die „echten" Koordinaten der Bounding-Box. Jede Primitive enthält, um Translation, Skalierung, Rotation und Verzerrung zu unterstützen, zusätzlich zu diesen Punkten eine Transformationsmatrix der dritten Dimension. Diese Matrix wird durch die Klasse `Matrix3D` zur Verfügung gestellt und in dem Attribut `transformationMatrix` gespeichert. Erst die Multiplikation eines Punktes mit dieser Matrix liefert die „echten" Punktkoordinaten, die in 1/100pt angegeben werden. Die Transformationsmatrix wird mit der Einheitsmatrix initialisiert und akkumuliert alle durchgeführten Transformationen. Diese Berechnungsweise bietet zwei große Vorteile:

- Da vor der Anzeige eines Objekts für jeden Punkt nur *eine* Matrizenmultiplikation bzw. die Berechnung zweier Gleichungen (x,y) erfolgen muß, wird die Anzahl der Berechnungsschritte gegenüber einer sequentiellen Ausführung der Transformationen erheblich reduziert. [Fel92]
- Es werden Rundungsfehler als Folge von Transformationen verhindert, da die Koordinaten in Variablen der Typen `int` (Punkte) *und* `double` (Matrix) gespeichert werden. Die Fließkommazahlen der Matrix bieten die geforderte hohe Genauigkeit, wodurch Rundungsfehler praktisch ausgeschaltet werden.

Wir bezeichnen die Punkte `p1` und `p2` als *Originale*. Die Punkte, die sich aus den Originalen nach einer Matrizenmultiplikation ergeben, heißen *Bildpunkte*. Die Berechnung eines Bildpunkts aus seinem Original führt die geschützte Methode `origToPic()` aus. Um zu einem Bildpunkt den entsprechenden Originalpunkt zu erhalten, kann die Methode `picToOrig()` verwendet werden, die unter Nutzung der inversen Matrix die Berechnung durchführt. In dem Attribut `inverseTransformationMatrix` wird die inverse Matrix zwischengespeichert. Ändert sich die Transformationsmatrix, wird automatisch die Inverse neu berechnet. Die folgenden zwei Gleichungen werden zur Umrechnung zwischen Original- und Bildpunkten genutzt:

$$\texttt{Bild} = \texttt{origToPic(Orig)} \implies \begin{bmatrix} x_{Bild} & y_{Bild} & 1 \end{bmatrix} = \begin{bmatrix} x_{Orig} & y_{Orig} & 1 \end{bmatrix} \cdot \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

$$\texttt{Orig} = \texttt{picToOrig(Bild)} \implies \begin{bmatrix} x_{Orig} & y_{Orig} & 1 \end{bmatrix} = \begin{bmatrix} x_{Bild} & y_{Bild} & 1 \end{bmatrix} \cdot \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}^{-1}$$

Die Variablen a bis f stehen für verschiedene Fließkommazahlen (`double`), die sich aufgrund der angewendeten Transformationen ergeben.

Die Klasse `GraphicObject` arbeitet ausschließlich intern mit Originalpunkten. Alle Methoden, die Koordinaten als Parameter empfangen oder zurückgeben, benutzen Bildpunkte, die – wie schon erwähnt – in der Einheit 1/100pt angegeben werden.

Über die Methode `getBounds()` wird die Bounding-Box unter Nutzung der Java-Klasse `Rectangle` zurückgegeben. Sollen die Koordinaten in Pixel und unter Einbeziehung eines Zoom-Faktors zurückgegeben werden, kann die Methode `getDrawingBounds()` genutzt werden. Sie erhält als Parameter die Instanz einer Klasse, welche das Interface `GraphicObjectView` implementiert. Dieses Interface repräsentiert eine Ansicht, in der das Grafikobjekt dargestellt wird.

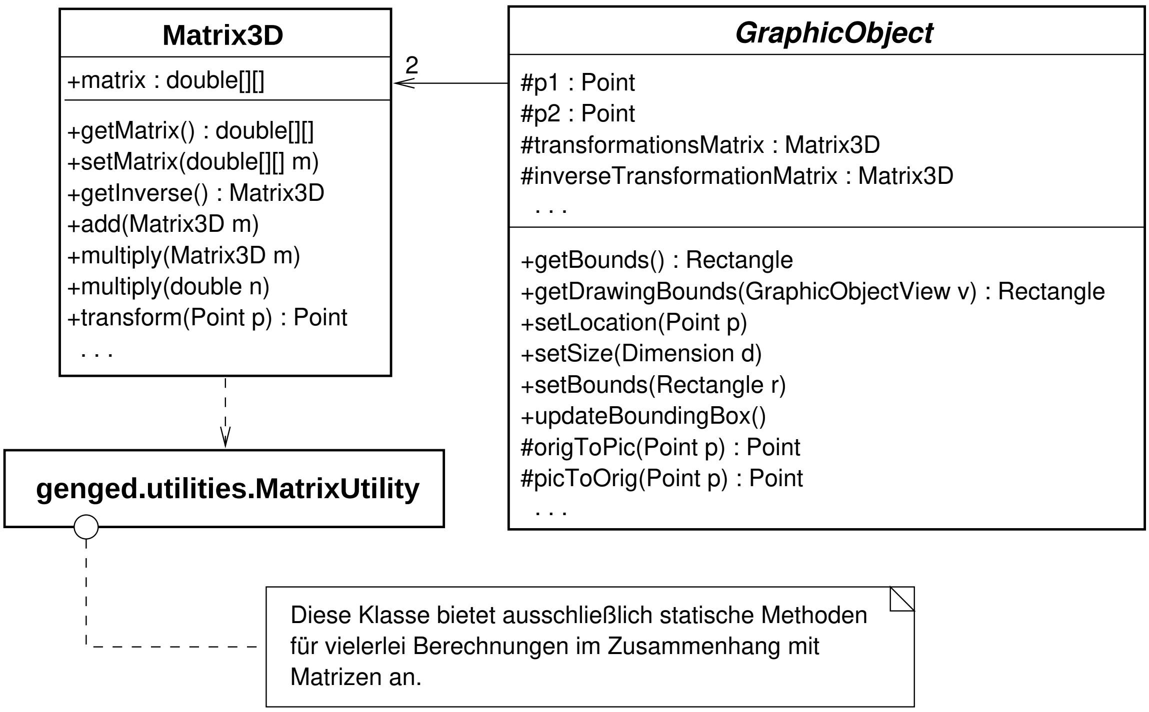


Abbildung 5.4: Klassendiagramm: `GraphicObject`, Bounding-Box

Die Abbildung 5.4 zeigt die erläuterten Attribute und Methoden in einem Klassendiagramm. Außerdem werden wieder Beziehungen zu verschiedenen Klassen aufgezeigt.

5.3.3 Transformationen

Es existieren vier verschiedene Transformationen, mit denen eine Primitive verändert werden kann: Translation, Skalierung, Rotation und Verzerrung. Die beiden zuletzt genannten können allerdings nur für Klassen angewendet werden, die Eigenschaften der Klassen `OpenPolygonObject` oder `ClosedPolygonObject` besitzen. Die Einschränkung ergibt sich durch die Benutzung von Constraints im Zusammenhang mit dem von uns eingesetzten Constraint-Solver. Constraints, die die Beziehung eines Objekts zu einem schräg in der Ebene liegenden Rechteck ausdrücken, werden zu komplex. In einem solchen Fall müßte jede Seite des Rechtecks als eigenständige Linie aufgefaßt werden.

Bei jeder der im folgenden beschriebenen Transformationen wird die Transformationsmatrix mit einer speziellen Matrix multipliziert [Fel92], die abhängig ist von der gewählten Transformation. Dazu wird jeweils die geschützte Methode `multiplyWithTransformationMatrix` aufgerufen. Sie berechnet außerdem die Inverse der neuen Transformationmatrix.

Translation: Zum Verschieben einer Primitive existieren verschiedene Methoden. `setLocation()` bietet die Möglichkeit, absolute Koordinaten für die neue linke obere Ecke der Bounding-Box anzugeben. Ebenso die Methode `setBounds()`, hier müssen allerdings zusätzlich neue Ausmaße übergeben werden. Beide Methoden errechnen die Translationen t_x und t_y in x- und y-Richtung, die an die Methode `translate()` übergeben werden. Diese Methode kann auch direkt mit relativen Koordinaten aufgerufen werden. Die Transformationsmatrix wird mit folgender Translationsmatrix multipliziert:

$$M_{Translation} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Skalierung: Um die Größe eines Objekts zu ändern, können die Methoden `setSize()`, `setBounds` sowie `scale()` genutzt werden. Die ersten beiden Methoden benutzen wiederum die Methode `scale()`, nachdem Sie die Skalierungsfaktoren s_x und s_y für die x- und y-Richtung errechnet haben. Außerdem kann ein Fixpunkt (x_{fix}, y_{fix}) angegeben werden. Standardmäßig wird die linke obere Ecke der Bounding-Box als Fixpunkt verwendet. Die Skalierungsmatrix errechnet sich unter Einbeziehung des Fixpunktes durch:

$$M_{Skalierung} = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_{fix} & -y_{fix} & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_{fix} & y_{fix} & 1 \end{bmatrix} \right)$$

Rotation: `rotate()` ist die einzige Methode, mit der eine Primitive rotiert werden kann. Hierzu wird ein Winkel α (in Radiant) sowie der Fixpunkt (x_{fix}, y_{fix}), um den das Objekt rotiert werden soll, übergeben. Standardmäßig wird ein Objekt um seinen Mittelpunkt rotiert. Die Rotationsmatrix enthält trigonometrische Funktionen und ergibt sich bei Berücksichtung des Fixpunktes aus:

$$M_{Rotation} = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_{fix} & -y_{fix} & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_{fix} & y_{fix} & 1 \end{bmatrix} \right)$$

Verzerrung: Auch unter dem Begriff Scherung bekannt, dient diese Transformation dem gleichmäßigen Verzerren einer Primitive in x- oder y-Richtung. Dazu wird die Methode `stretch()` benutzt, der neben einem Fixpunkt zwei Faktoren (x-, y-Richtung) übergeben werden. Standardmäßig wird der Mittelpunkt des Objekts als Fixpunkt genommen. Die Scherungsmatrix errechnet sich folgendermaßen:

$$M_{Verzerrung} = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_{fix} & -y_{fix} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & Sch_y & 0 \\ Sch_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_{fix} & y_{fix} & 1 \end{bmatrix} \right)$$

Die Abbildung 5.5 veranschaulicht die Beziehungen der Klasse `GraphicObject` zu verschiedenen anderen Klassen und listet die erläuterten Attribute und Methoden auf.

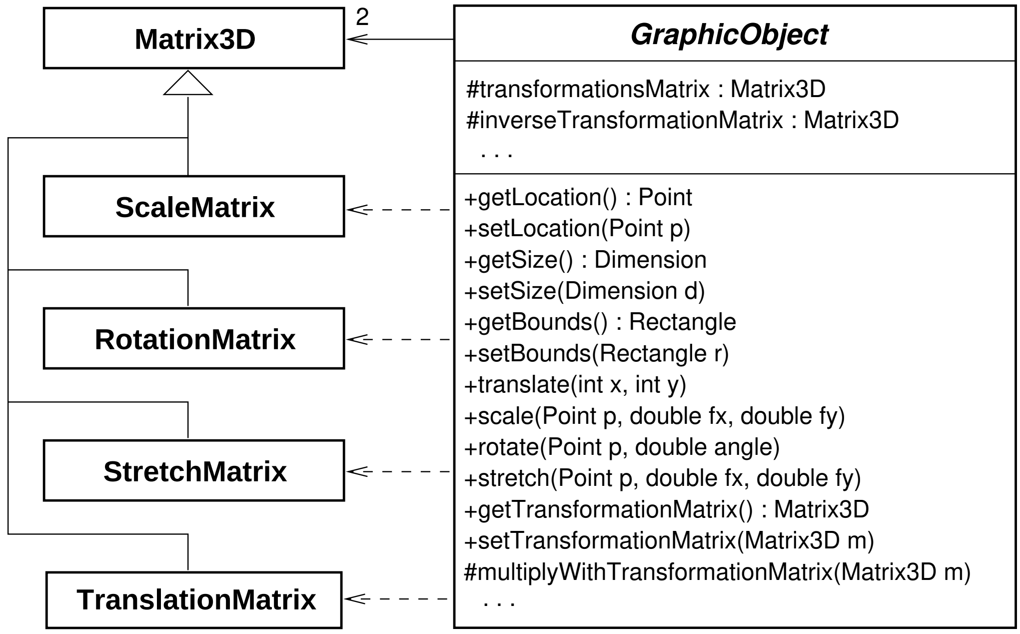


Abbildung 5.5: Klassendiagramm: `GraphicObject`, Transformationen

5.3.4 Knotenpunkte

Viele Primitiven enthalten Knotenpunkte, die in Zusammenhang mit Bearbeitungsmechanismen der Zeichenfläche zur Veränderung der Form dienen. Je nach Art der Primitive befinden sich diese Knoten an verschiedenen Stellen. Jeder Knoten kann verschoben werden. Bei Objekten, die von den Klassen `OpenPolygonObject` oder `ClosedPolygonObject` abgeleitet sind, können außerdem Knoten hinzugefügt und entfernt werden. Die Abbildung 5.6 zeigt einige Beispiele für Knotenpositionen.

Im Zusammenhang mit den Mechanismen der Zeichenfläche wurden verschiedene Methoden entworfen. Ziel ist es, interaktiv mit den Knotenpunkten zu arbeiten. Jeder Knoten wird durch einen bestimmten Punkt (Java-Klasse `Point`) identifiziert. Knotenpunkte werden jedoch nicht separat in einer Liste gespeichert. Stattdessen werden sie innerhalb der Bearbeitungsmethoden auf verschiedene Attribute einer Primitive abgebildet. Beispielsweise wird der Startwinkel eines Ellipsenausschnitts durch einen Knoten repräsentiert, der auf der Ellipse liegt (siehe Abbildung 5.6).

Um alle innerhalb einer Primitve vorhandenen Knoten zu erhalten, kann die Methode `getNodes()` ohne Parameter aufgerufen werden. Sie gibt ein `Point`-Array zurück, das die Knotenpunkte in der Einheit 1/100pt enthält. Um dieselben Werte in der Einheit Pixel und in Abhängigkeit von einem bestimmten Zoom-Faktor zu erhalten, kann als Parameter eine Ansicht `GraphicObjectView` angegeben werden. Die Me-

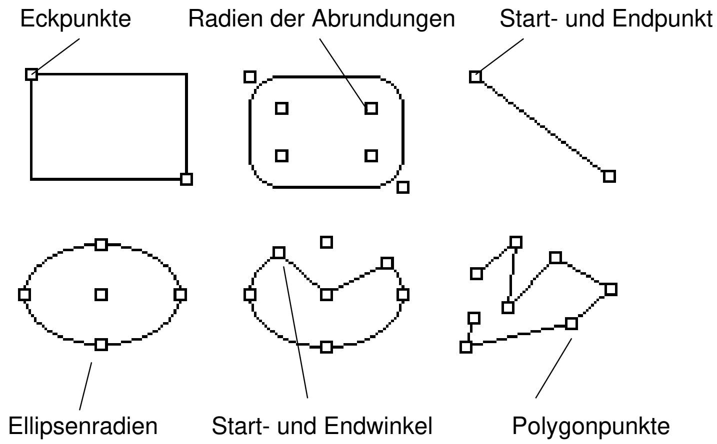


Abbildung 5.6: Beispiele für Knoten in verschiedenen Primitiven

thode `getNode()` gibt zu einem übergebenen Punkt den nächstliegenden Knotenpunkt zurück. Dabei markiert ein Toleranzbereich, der über die Konstante `TOLERANCE` festgelegt wurde, die maximale Abweichung. Sollte kein Punkt in der Nähe der übergebenen Position gefunden werden, wird `null` zurückgeliefert.

Es kann mit Hilfe der drei Methoden `canNodeBeAdded()`, `canNodeBeMoved()` und `canNodeBeRemoved()` überprüft werden, ob der übergebene Knotenpunkt hinzugefügt, verschoben oder entfernt werden kann. Zusätzlich existiert die Methode `shallSmartNodeBeRemoved()`, die feststellt, ob ein Knoten genau zwischen zwei anderen Knoten liegt und entfernt werden soll.

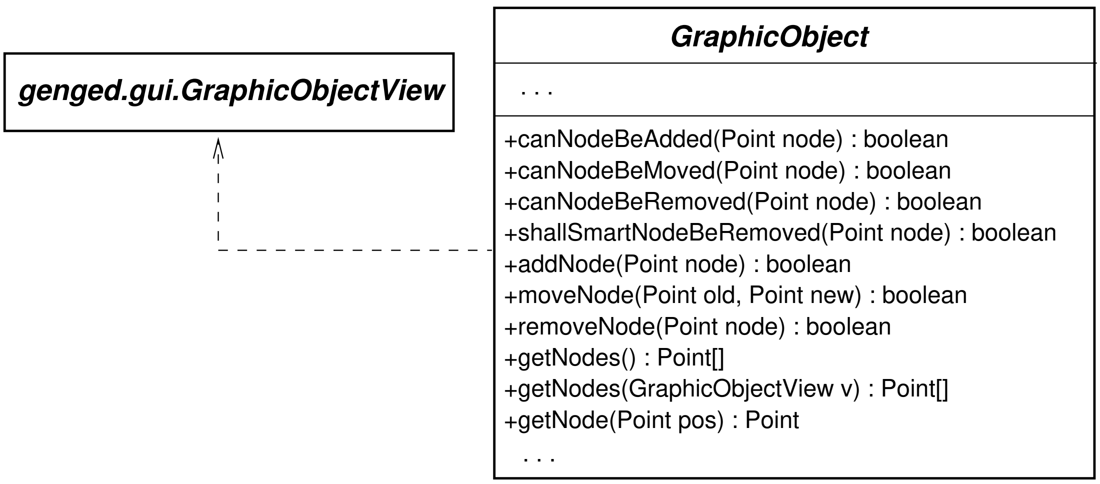


Abbildung 5.7: Klassendiagramm: `GraphicObject`, Knotenpunkte

Um Knoten hinzuzufügen, zu verschieben oder zu entfernen, können die Methoden `addNode()`, `moveNode()` und `removeNode()` benutzt werden. Allerdings ist zu beachten, daß Knoten bei den meisten Primitiven ausschließlich verschoben werden können. Jede dieser Methoden gibt `true` zurück, wenn die jeweilige Aktion erfolgreich ausgeführt werden konnte. Eine Übersicht aller Methoden ist in Abbildung 5.7 zu finden.

5.3.5 Verbindungspunkte

Jeder Primitive können Verbindungspunkte zugewiesen werden. Es sind Instanzen der Klasse `ConPoint`. Verbindungspunkte werden in einem Vector mit Namen `conpoints` gespeichert. Zur Verwaltung existieren verschiedene Methoden.

Mit den Methoden `addConPoint()` und `insertConPointAt()` können Verbindungspunkte hinzugefügt werden. Für sie wird automatisch die `setParent()`-Methode aufgerufen, um die Abhängigkeit zu definieren. Jeder Verbindungspunkt enthält einen Index, der seiner Position im Vector entspricht. Mit Hilfe der Methoden `removeConPoint()` und `removeConPointAt()` kann ein Verbindungspunkt wieder entfernt werden.

Durch Aufruf der Methode `containsConPoint()` kann festgestellt werden, ob der übergebene Verbindungspunkt bereits in der Primitive gespeichert wurde. Mit `hasConPoints()` und `countConPoints()` wird schließlich erfragt, ob und wieviele Verbindungspunkte in dem Grafikobjekt existieren.

Der Index eines Verbindungspunkts kann mit der Methode `indexOfConPoint()` ermittelt werden. Die Methode `getConPointAt()` gibt zu einem bestimmten Index den entsprechenden Verbindungspunkt zurück. Um eine Liste aller Punkte zu erhalten, kann `getConPoints()` aufgerufen werden. Diese Methode liefert einen Vector zurück. Um eine Enumeration zu erhalten, verwendet man `conPoints()`. Mitunter soll ein Verbindungspunkt anhand seiner Position gefunden werden. Dies kann man durch Aufruf der Methode `getConPoint()` erreichen. Eine gleichnamige Methode existiert schließlich noch, um einen Verbindungspunkt über seinen Namen zu erhalten. Weiterhin existiert die Methode `getAllSubObjects()`, die alle Primitiven zurückgibt, die zu einem Objekt gehören. Dazu zählen insbesondere auch alle Verbindungspunkte.

Wird eine neue Primitive erzeugt, enthält sie keine Verbindungspunkte. Diese Tatsache widerspricht etwas den Anforderungen. Ursprünglich sollten automatisch neun verschiedene Verbindungspunkte in Abhängigkeit der Bounding-Box vorhanden sein. Beim Entwurf des Systems hat sich jedoch herausgestellt, daß diese vielen Verbindungspunkte den verwendeten Constraint-Solver überlasten würden. Jede Primitive erhält einen Initialisierungs-Constraint[1], der aus mehreren (Un)gleichungen besteht. Auch Verbindungspunkte erhalten einen solchen Constraint zugewiesen. Bei vier (Un)gleichungen pro Initialisierungs-Constraint und neun Verbindungspunkten pro Objekt käme man bei einem Einsatz von nur 50 grafischen Primitiven bereits auf 1800 (Un)gleichungen. Daher wurde folgendes festgelegt:

- Verbindungspunkte werden ausschließlich durch den Benutzer definiert.
- Jeder Primitive können beliebig viele Verbindungspunkte zugeordnet werden.

[1] Auf Constraints wird im Kapitel 7 noch genauer eingegangen.

- Verbindungspunkte können völlig frei positioniert werden.
- Die gewünschte Position wird in Abhängigkeit der linken oberen Ecke und relativ zur Breite und Höhe der Bounding-Box durch einen speziellen Constraint beibehalten.
- Jede Primitive besitzt besonders geeignete Positionen für Verbindungspunkte, die nicht unbedingt auf der Bounding-Box liegen müssen.
- In einem speziellen Fangmodus kann ein Verbindungspunkt einer vordefinierten Position zugewiesen werden.

Mit der Methode `createConPoint()` kann jede Primitive einen Verbindungspunkt erzeugen. Er wird allerdings nicht automatisch hinzugefügt. Als Parameter wird die gewünschte Position übergeben. Außerdem kann angegeben werden, ob der Fangmodus aktiviert ist. Wenn er eingeschaltet ist, ruft diese Methode `catchPredefinedPosition()` mit der übergebenen Position auf. Es wird versucht, unter Berücksichtigung eines gewissen Toleranzbereiches die nächstgelegene vordefinierte Position zu finden. Für jede Primitive werden in der Methode `getPredefinedConPointsList()` alle besonderen Positionen definiert und als `Point`-Array zurückgegeben.

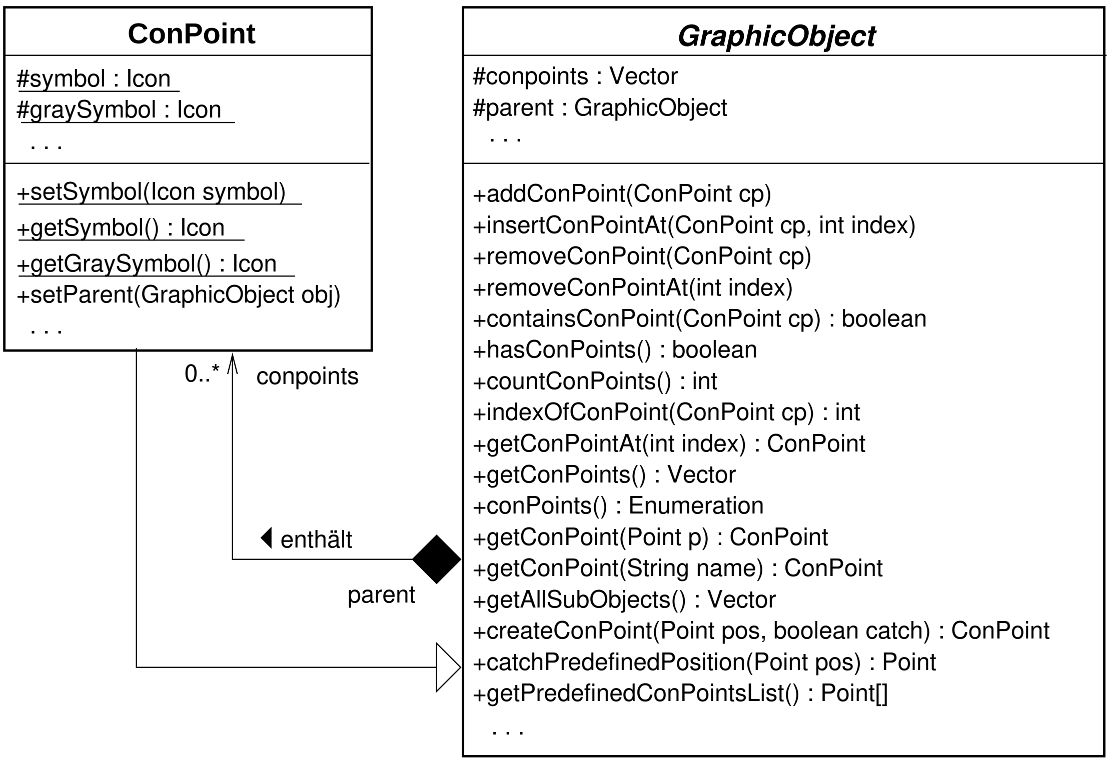


Abbildung 5.8: Klassendiagramm: `GraphicObject`, Verbindungspunkte

Verbindungspunkte, Objekte der Klasse `ConPoint`, besitzen generell eine Breite und Höhe von 1 1/100pt. Nur ihre Position ist variabel. Um sie dennoch sichtbar zu machen, wird ein kleines Symbol verwendet (▣). Die vordefinierten Positionen werden auf einem Grafikobjekt durch heller erscheinende Symbole dargestellt. Das entsprechende Symbol kann mit der statischen Methode `setSymbol()` der Klasse `ConPoint` geändert werden.

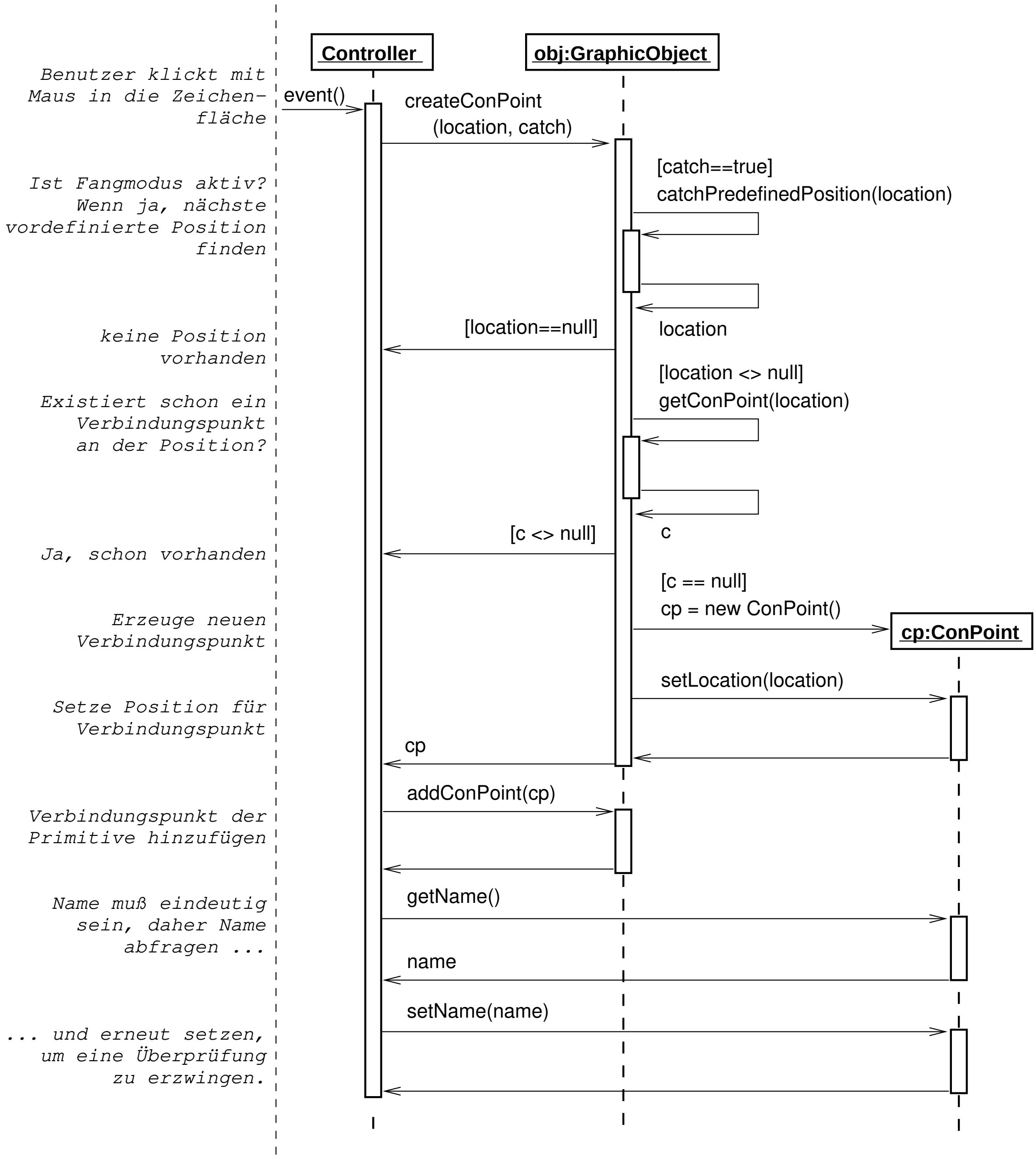


Abbildung 5.9: Sequenzdiagramm: Definition eines Verbindungspunktes

Die Abbildung 5.8 zeigt Attribute, Methoden und Verbindungen der Klasse `GraphicObject`, die in Zusammenhang mit der Verwendung von Verbindungspunkten stehen. In der Abbildung 5.9 wird ein Sequenzdiagramm gezeigt, das den typischen Ablauf der Definition eines neuen Verbindungspunktes auf einer bereits vordefinierten Position darstellt.

5.3.6 Darstellungs- und Bearbeitungsmodi

Eine Primitive kann in verschiedener Weise dargestellt werden. Normalerweise zeichnet sie sich einfach als ein Grafikobjekt. Mitunter müssen jedoch neben der eigentlichen Grafik noch andere Hilfsmittel angezeigt werden. Wird beispielsweise eine Primitive durch den Benutzer selektiert, muß eine Markierung dargestellt werden. Werden die Verbindungspunkte einer Primitive bearbeitet, müssen diese mit angezeigt werden, genauso die vordefinierten Positionen.

Um diesem Umstand gerecht zu werden, haben wir verschiedene Darstellungsmodi eingeführt, zwischen denen umgeschaltet werden kann. Jeder Darstellungsmodus hat eine bestimmte Auswirkung auf die Darstellung der Primitive. Es kann immer nur *ein* Modus aktiv sein. Die Tabelle 5.1 gibt einen Überblick. Bei den Darstellunsmodi handelt es sich um Konstanten, die in dem Interface `GraphicObjectConstants` definiert werden. Dieses Interface wird von der Klasse `GraphicObject` implementiert.

Darstellungsmodus	Zusätze zur normalen Grafik
`NORMAL_MODE`	—
`NODE_MODE`	Knotenpunkte
`SELECTED_MODE`	Markierung
`SELECT_ALL_MODE`	Markierung, Verbindungspunkte
`ROTATION_MODE`	Markierung
`STRETCH_MODE`	Markierung
`CONPOINT_MODE`	Verbindungspunkte, evtl. vordefinierte Positionen

Tabelle 5.1: Darstellungsmodi der Klasse `GraphicObject`

Die Klasse `GraphicObject` stellt zum Umschalten der Darstellungsmodi die Methode `setDrawMode()` zur Verfügung. Der aktuell eingestellte Modus kann mit der Methode `getDrawMode()` abgefragt werden. Außerdem existieren folgende Methoden, um zu prüfen, ob der Darstellungsmodus auf einen bestimmten Wert gesetzt ist: `isNormalMode()`, `isNodeMode()`, `isSelectedMode()`, `isRotationMode()`, `isStretchMode()` und `isConPointMode()`. Die Methode `isChoosed()` gibt `true` zurück, wenn ein anderer Modus als der `NORMAL_MODE` aktiviert wurde.

Neben dem Darstellungsmodus existiert noch ein sogenannter Bearbeitungsmodus. Er ähnelt den Arbeitsmodi der Zeichenfläche, die wir in einem der nächsten Kapitel kennenlernen werden. Über den Bearbeitungsmodus wird festgelegt, welche Bestandteile der Grafik gerade editiert bzw. wie sie editiert werden. Arbeitet der Benutzer beispielsweise mit den Knoten einer Primitive, so muß festgelegt werden, ob er Knoten hinzufügt, verschiebt oder löscht. Der Bearbeitungsmodus muß daher noch detaillierter sein als der Darstellungsmodus. In der Tabelle 5.2 sind wieder die entsprechenden Konstanten aus dem Interface `GraphicObjectConstants` aufgeführt.

Bearbeitungsmodus	Zugehöriger Darstellungsmodus	Bedeutung
NO_EDIT	NORMAL_MODE	Keine Bearbeitung
ADD_NODE REMOVE_NODE MOVE_NODE SMART_NODES	NODE_MODE	Hinzufügen von Knoten Entfernen von Knoten Verschieben von Knoten Hinzufügen, Verschieben und Entfernen von Knoten in einem Modus
SELECT_OBJECT TRANSLATE_OBJECT SCALE_OBJECT RESIZE_MOVE_OBJECT	SELECTED_MODE	Markieren von Objekten Verschieben von Objekten Skalieren von Objekten Verschieben oder Skalieren
ROTATE_OBJECT	ROTATION_MODE	Rotieren von Objekten
STRETCH_OBJECT	STRETCH_MODE	Verzerren von Objekten
ADD_CATCHED_CONPOINT ADD_FREE_CONPOINT REMOVE_CONPOINT MOVE_CONPOINT MOVE_CATCHED_CONPOINT SELECT_CONPOINT	CONPOINT_MODE	Hinzufügen eines Verbindungspunktes im Fangmodus Hinzufügen eines frei positionierten Verbindungspunktes Entfernen eines Verbindungspunktes Verschieben eines Verbindungspunktes an eine beliebige Stelle Verschieben eines Verbindungspunktes an eine vordefinierte Position Markieren von Objekten und Verbindungspunkten
CREATE_OBJECT	NORMAL_MODE	Erstellen eines neuen Objektes (nur während der Erstellungsphase)

Tabelle 5.2: Bearbeitungsmodi der Klasse `GraphicObject`

Die Bearbeitungsmodi werden mit der Methode `setEditMode()` gesetzt. Die Methode `getEditMode()` liefert den aktuellen Bearbeitsmodus zurück. Um Darstellung- und Bearbeitungsmodus gleichzeitig zu setzen, kann die Methode `setModes()` benutzt werden.

5.3.7 Bearbeitungsfähigkeiten und -rechte

Ein Rechteck hat andere Eigenschaften als ein Polygon. Beide Objekte sind jedoch Primitiven und von der Klasse `GraphicObject` abgeleitet. Dennoch kann ein Rechteck beispielsweise nicht rotiert oder verzerrt werden.

Um für jede einzelne Primitive festzulegen, zu welchen Aktionen sie fähig ist, wurde das Attribut `abilities` definiert. Es ist vom Typ `int` und speichert bitweise die Bearbeitungsfähigkeiten einer Primitive. Diese werden durch verschiedene Konstanten definiert. Dabei handelt es sich um dieselben Konstanten, die schon im Zusammenhang mit dem Bearbeitungsmodus behandelt wurden. Die Konstante `MIRROR_OBJECT` ergänzt die Tabelle 5.2. Ist sie in dem Attribut `abilities` gesetzt, kann das Objekt während eines Skalierungsprozesses gespiegelt werden. Die Methode `getAbilities()` gibt die gesamten Fähigkeiten als ein Wert zurück. Dagegen kann mit der Methode `isAbleToDo()` geprüft werden, ob eine Primitive einzelne bestimmte Bearbeitungsfähigkeiten besitzt. Ein Aufruf könnte beispielsweise folgendermaßen aussehen: `isAbleToDo(MIRROR_OBJECT | TRANSLATE_OBJECT`. Es wird nur `true` zurückgegeben, wenn *alle* angegebenen Fähigkeiten möglich sind.

Die Fähigkeiten einer Primitiven können weiter eingeschränkt werden, indem sogannte Bearbeitungsrechte vergeben werden. So kann man einer Primitive untersagen, an einer bestimmten Benutzeraktion teilzunehmen. Nachdem eine Grafik erstellt wurde, darf sie entsprechend ihrer Fähigkeiten an *allen* Aktionen teilnehmen. Das Attribut `authority` enthält die aktuellen Befugnisse. Es kann mit der Methode `getAuthority()` ausgelesen und mit der Methode `setAuthority()` neu gesetzt werden. Allerdings können keine Rechte für eine Aktion erteilt werden, wenn eine Primitive nicht die Fähigkeit für diese Aktion besitzt. Um festzustellen, ob ein einzelnes Recht vergeben oder untersagt wurde, werden die Methoden `isAllowed()` und `isForbidden()` benutzt. Ein oder mehrere Befugnisse können mit der Methode `allow()` erteilt oder mit `forbid()` verboten werden.

Die Bearbeitungsrechte müssen von einem Kontrollmechanismus, der die Aktionen auf einem Alphabet koordiniert, abgefragt und beachtet werden. Die Klasse `GraphicObject` sorgt ausschließlich für die Verwaltung der Rechte, jedoch nicht für deren Durchsetzung.

In der Abbildung 5.10 sind Attribute, Methoden und Beziehungen abgebildet, die in Zusammenhang mit den verschiedenen Darstellungs- und Bearbeitungsmodi sowie den Bearbeitungsfähigkeiten und -rechten einer Primitive stehen.

5.3.8 Darstellung graphischer Objekte

Jegliche Grafikobjekte, die von der Klasse `GraphicObject` abgeleitet sind, könnten auf mehreren Ansichten[2] gleichzeitig dargestellt werden. Alle diese Ansichten müssen jedoch das Interface `GraphicObjectView` implementieren, das unter anderem Funktionalitäten zum Zoomen spezifiziert. So liefert die Methode `getZoom()` den aktuellen Zoom-Faktor der Ansicht. Er muß beim Zeichnen des Objekts berücksichtigt werden.

[2] Dabei handelt es sich um verschiedene Arten von Zeichenflächen. Innerhalb des Alphabet-Editors wird die Klasse `genged.gui.AbstractDrawArea` benutzt. Es könnten aber in Erweiterungen eigene Zeichenflächen implementiert werden.

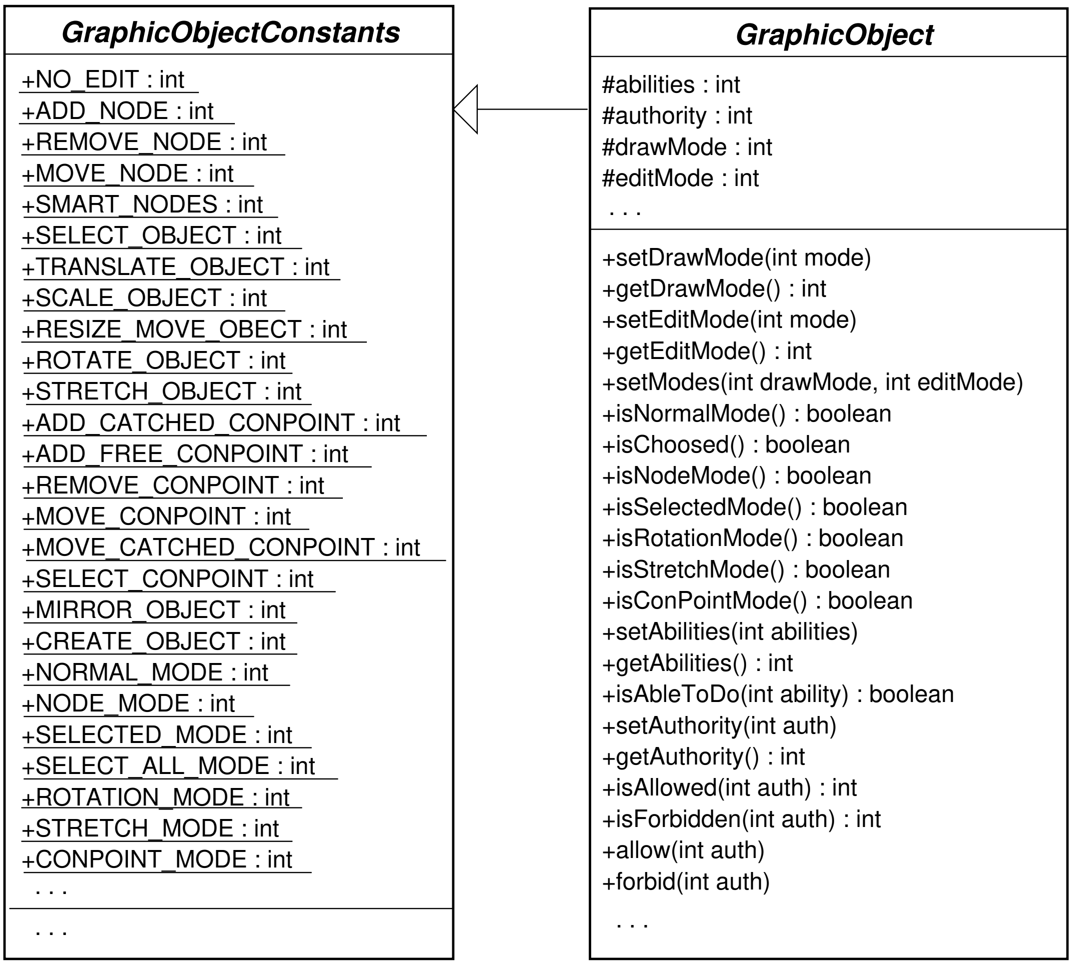


Abbildung 5.10: Klassendiagramm: `GraphicObject`, Bearbeitung

Um eine Primitive zu zeichnen, ruft man die `paint()`-Methode auf, die als Parameter einen `GraphicObjectView` sowie einen grafischen Kontext (`Graphics`) verlangt. Diese Methode ruft in Abhängigkeit des aktuellen Darstellungsmodus verschiedene andere `paint()`-Methoden in folgender Reihenfolge auf:

- `paintBackground()`
  Diese Methode sollte normalerweise nichts tun, da der Hintergrund einer Grafik bereits von anderen Objekten belegt sein kann. Dennoch könnte sie für Debug-Zwecke genutzt werden.

- `paintComponent()`
  Dies ist die wichtigste Methode, da hier die eigentliche Grafik des Objekts dargestellt wird.

- `paintChildren()`
  Besitzt das grafische Objekt Kind-Objekte, werden aus dieser Methode heraus deren `paint()`-Methoden aufgerufen.

- paintConPoints()
  Im CONPOINT_MODE werden durch diese Methode die Verbindungspunkte über die Grafik des Objekts gezeichnet.

- paintNodes()
  Im NODE_MODE stellt diese Methode alle Knotenpunkte des Objekts dar.

- paintMark()
  Diese Methode zeichnet die Markierung einer Primitiven.

- paintText()
  Mit dieser Methode kann ein Informationstext an den verschiedenen Seiten einer Grafik angezeigt werden. Genauere Informationen dazu liefert die Online-Dokumentation.

Alle bisher aufgeführten Methoden erhalten als Parameter ebenfalls einen GraphicObjectView sowie einen grafischen Kontext (Graphics). Es kann auch noch ein dritter Parameter angegeben werden, der die Grafik unabhängig von ihrer derzeitigen Position positioniert. Dabei handelt es sich um einen Punkt (Point), der in der Einheit 1/100pt angegeben wird.

Eine Primitive kann als unsichtbar gekennzeichnet werden; dann wird sie trotz Aufrufs von paint() nicht dargestellt. Die Methode setVisible() dient diesem Zweck. Der aktuelle Status kann mit der Methode isVisible() abgefragt werden. Sollen innerhalb einer Ansicht auch unsichtbare Objekte angezeigt werden, muß die Ansichts-Methode getShowInvisible() den Wert true zurückliefern.

Das Attribut views speichert in einem Vector alle Ansichten, die in der Primitive registriert wurden. Ein GraphicObjectView kann mit der Methode addView() hinzugefügt und mit removeView() wieder entfernt werden. Um alle registrierten Ansichten zu entfernen, wird die Methode removeAllViews() aufgerufen. getViews() gibt alle angemeldeten Ansichten in einem Vector zurück. Ändert ein Objekt sein Aussehen, müssen alle registrierten Ansichten informiert werden. Dazu genügt es, die Methode fireObjectIsDirty() aufzurufen. In der Regel funktioniert dieser Mechanismus automatisch. Allerdings wird dadurch das Objekt nicht sofort neu gezeichnet. In den angemeldeten Ansichten wird nur jeweils der Bereich als „unsauber“ markiert, der durch die Grafik und deren Zusätze beansprucht wurde und wird. Meistens werden mehrere Grafiken auf einmal verändert. Erst wenn alle Grafiken ihre Ansichten informiert haben, sollte von einer der Kontrollstrukturen die repaint()-Methode der Ansicht aufgerufen werden. Um aus einem GraphicObject heraus alle registrierten Ansichten neu zu zeichnen, kann auch die Methode repaintAllViews() genutzt werden.

Die Abbildung 5.11 zeigt in einem Klassendiagramm die Attribute und Methoden, die im Zusammenhang mit der Darstellung einer Primitive benutzt werden.

Der Benutzer des Systems sollte fähig sein, eine Grafik innerhalb der Zeichenfläche durch einen Mausklick auszuwählen. Da die eigentliche Form der Primitive variiert, genügt es nicht abzufragen, ob ein Punkt innerhalb der Bounding-Box ausgewählt wurde. Es sollte vielmehr überprüft werden, ob die Mauskoordinate auf der (Umriß)linie des Objekts liegt. Handelt es sich bei der Primitive um eine geschlossene Grafik, der eine

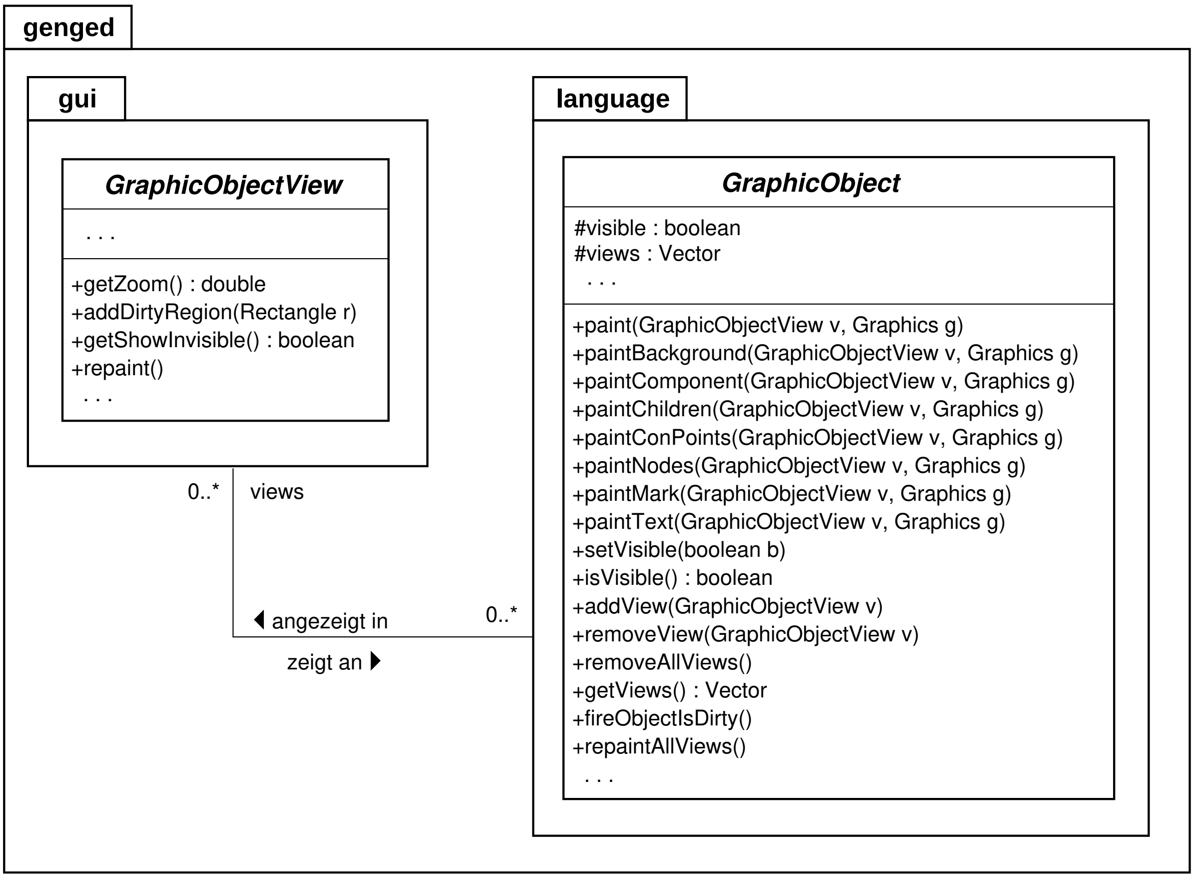


Abbildung 5.11: Klassendiagramm: `GraphicObject`, Darstellung

Füllfarbe zugewiesen wurde, sollte auch ein Klick in die Füllung erkannt werden. Die Methode `contains()` überprüft für eine übergebene Ansicht, ob ein spezifizierter Punkt (in Pixel) innerhalb der Bounding-Box des Objekt liegt. Mit der Methode `touchs()` kann genau ermittelt werden, ob ein Punkt (in Pixel) auf der *Form* einer Primitive liegt. Sollen auch Zusätze wie z.B. eine Markierung des Objekts mit einbezogen werden, kann `touchsAccessories()` aufgerufen werden. Die Methode `getTouchedObject()` liefert schließlich das Objekt selbst oder eines seiner Unterobjekte.

5.4 Gruppen von Primitiven

Durch Anwendung der Klasse `GroupObject` wird das Whole-Part-Pattern in der Variante des Composite-Pattern umgesetzt. Diese Klasse ist direkt von der Klasse `GraphicObject` abgeleitet und kann mehrere Primitiven zu einem Objekt zusammenschließen. Auch eigene Instanzen können in den Verbund mit einbezogen werden, so daß dadurch eine beliebige Schachtelung von Gruppen möglich wird. Somit wird durch Verwendung von Gruppen eine Objekt-Hierarchie aufgebaut.

Die Position und Ausmaße einer Gruppe ergeben sich aus der Vereinigungsmenge der Bounding-Boxen aller Gruppenmitglieder. Wird eine Aktion auf einer Gruppe von Objekten ausgeführt, z.B. Skalierung, werden alle Objekte innerhalb der Gruppe beeinflußt. Dennoch ist es möglich, auch weiterhin auf einzelne Primitiven einer Gruppe zuzugreifen und sie zu verändern.

5.4.1 Verwaltung der Gruppenmitglieder

Eine neu erstellte Gruppe enthält keinerlei Gruppenmitglieder. Sie ist leer. Um einer Gruppe Mitglieder hinzuzufügen, können die Methoden `addObject()` und `insertObjectAt()` genutzt werden, zum Entfernen `removeObject()` oder `removeObjectAt()`. Die Objekte werden in einer indizierten Liste gespeichert (Vector), so daß über einen Index auf jedes Element der Gruppe zugegriffen werden kann. Die Methode `getObjectAt()` liefert die Primitive des spezifizierten Index. Durch Anwendung von `indexOfObject()` kann der Index eines Grafikobjekts ermittelt werden. Die Methode `containsObject()` überprüft schließlich, ob sich eine bestimmte Primitive in der Gruppe befindet. Alle Gruppenmitglieder können durch die Methode `getObjects()` als Vector oder mit `objects()` als Enumeration abgerufen werden. Die Anzahl der Objekte einer Gruppe wird von der Methode `countObjects()` zurückgegeben. Die Methode `isEmpty()` ermittelt, ob überhaupt irgendwelche Gruppenmitglieder vorhanden sind.

Zum Traversieren der gesamten Hierarchie, die durch Gruppen und Untergruppen sowie durch die Verwendung von Verbindungspunkten entsteht, können die Methoden `traverseObjectTree()` und `objectTree()` benutzt werden. Über die Konstanten `PREORDER` und `POSTORDER` wird die Reihenfolge der Traversierung festgelegt.

Um eine Gruppe aufzulösen, kann die Methode `disintegrateGroup()` aufgerufen werden. Die Anwendung ist allerdings nur erlaubt, wenn die Gruppe selbst Mitglied einer Gruppe ist und demzufolge das Attribut `parent` gesetzt ist. Für diesen Fall werden alle Gruppenmitglieder in die übergeordnete Gruppe „verschoben“. Die Gruppe selbst entfernt sich automatisch aus dem übergeordneten Objekt und wird dadurch (normalerweise) nicht weiter referenziert.

5.4.2 Zeichenreihenfolge

Die Gruppenobjekte werden in einem Vector mit dem Namen `objects` gespeichert. Gezeichnet werden alle Primitiven durch Aufruf der `paintChildren()`-Methode. Sie stellt die Objekte in genau der Reihenfolge dar, wie sie in `objects` vorliegen. Dadurch kann es vorkommen, daß sich Objekte teilweise überlagern oder gänzlich verdecken.

Mit den Methoden `setNewPosition()`, `setForeground()`, `setBackground()`, `setLower()` und `setHigher()` kann für eine Primitive die Position innerhalb der Liste geändert werden. Alle genannten Methoden werden auf `setNewPosition()` abgebildet, die die eigentliche Positionsänderung – die Änderung der Zeichenreihenfolge – vornimmt.

Die Abbildung 5.12 zeigt einige Attribute und Methoden der Klasse `GroupObject`.

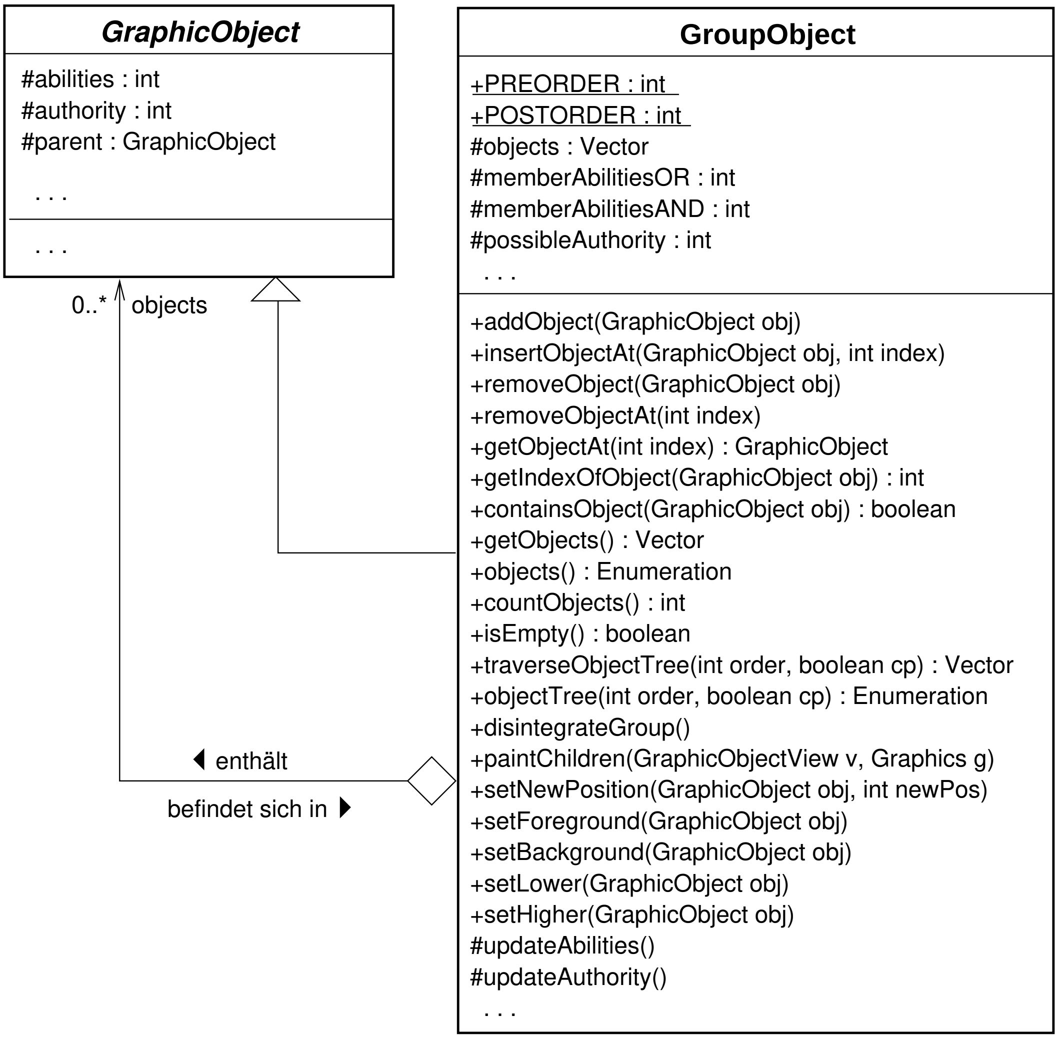


Abbildung 5.12: Klassendiagramm: `GroupObject`

5.4.3 Bearbeitungsfähigkeiten und -rechte

Die Vergabe von Bearbeitungsfähigkeiten an eine Gruppe unterscheidet sich grundsätzlich von einer einzelnen Primitive. Sie ist abhängig von den Gruppenmitgliedern. Wir unterscheiden zwischen zwei Arten von Fähigkeiten:

- In dem Attribut `memberAbilitiesOR` werden die Konstanten aller Bearbeitungsfähigkeiten verknüpft, die bei *mindestens einem* Gruppenmitglied vorhanden sein müssen, um als Gruppen-Fähigkeit zu gelten.

- Die meisten Fähigkeiten sind jedoch in dem Attribut `memberAbilitiesAND` enthalten. In diesem Fall müssen *alle* Gruppenmitglieder eine Fähigkeit vorweisen, damit sich diese auf die Gruppe überträgt.

Das Attribut `abilities` speichert die zur Zeit geltenden Bearbeitungsfähigkeiten der Gruppe. Von diesem Attribut hängt ab, an welchen Aktionen *alle* Objekte der Gruppe teilnehmen können. Wie bei einer einfachen Primitive können auch für eine Gruppe Bearbeitungsrechte vergeben werden, die die Fähigkeiten weiter einschränken. Die vergebenen Rechte werden in dem Attribut `possibleAuthority` vermerkt, um die tatsächlich aktuellen Rechte (Attribut `authority`) zu aktualisieren, sobald sich die Mitglieder und damit die Fähigkeiten der Gruppe ändern. Die tatsächlichen Fähigkeiten und Rechte werden durch Aufruf der geschützten Methoden `updateAbilities()` und `updateAuthority()` automatisch neu bestimmt.

5.5 Alphabet-Objekte

Primitiven und Gruppenobjekte haben normalerweise ein übergeordnetes Objekt, das im Attribut `parent` referenziert wird. Die Wurzel einer Hierarchie von Primitiven bildet jedoch immer ein Objekt, welches das Interface `AlphabetObject` implementiert. Das `parent`-Attribut wird in diesem Fall auf `null` gesetzt. Stattdessen existiert in dem Attribut `signature` eine Referenz auf das Alphabet, zu dem das Objekt gehört.

Es gibt zwei Arten von Alphabet-Objekten: Symbole (Klasse `SymbolObject`) und Datentypen (Klasse `DatatypeObject`). Neben vielen bereits bekannten Methoden fordert das Interface `AlphabetObject` die Methoden `getKind()`, `isSymbol()` und `isDatatype()`, um zwischen den beiden Arten zu unterscheiden. Mit der Methode `setAlphabet()` kann einem Symbol oder Datentypen ein Alphabet zugewiesen werden. Die Methode `getAlphabet()` gibt das aktuell gesetzte Alphabet zurück.

5.5.1 Symbole

Hinter einem Symbol verbirgt sich nichts weiter als ein Objekt der Klasse `GroupObject`, die um die Methoden des Interfaces `AlphabetObject` erweitert wurde. Die Abbildung 5.13 gibt einen Überblick über die Beziehungen zwischen den verschiedenen Klassen der Alphabet-Objekte.

5.5.2 Datentypen

Datentypen werden zur Attributierung von Diagrammelementen verwendet. So werden beispielsweise Zeichenketten für Namen oder Listen für Aufzählungen benötigt. Im Gegensatz zum Symbol enthalten Datentypen der Klasse `DatatypeObject` keine Primitiven, dafür jedoch einen vom verwendeten Datentyp abhängigen veränderlichen Wert.

Das System kann um eigene Datentypen erweitert werden. Die Vorgehensweise ist sehr ausführlich im letzten Kapitel des Benutzerhandbuchs dargestellt, das im Anhang dieser Diplomarbeit abgedruckt ist.

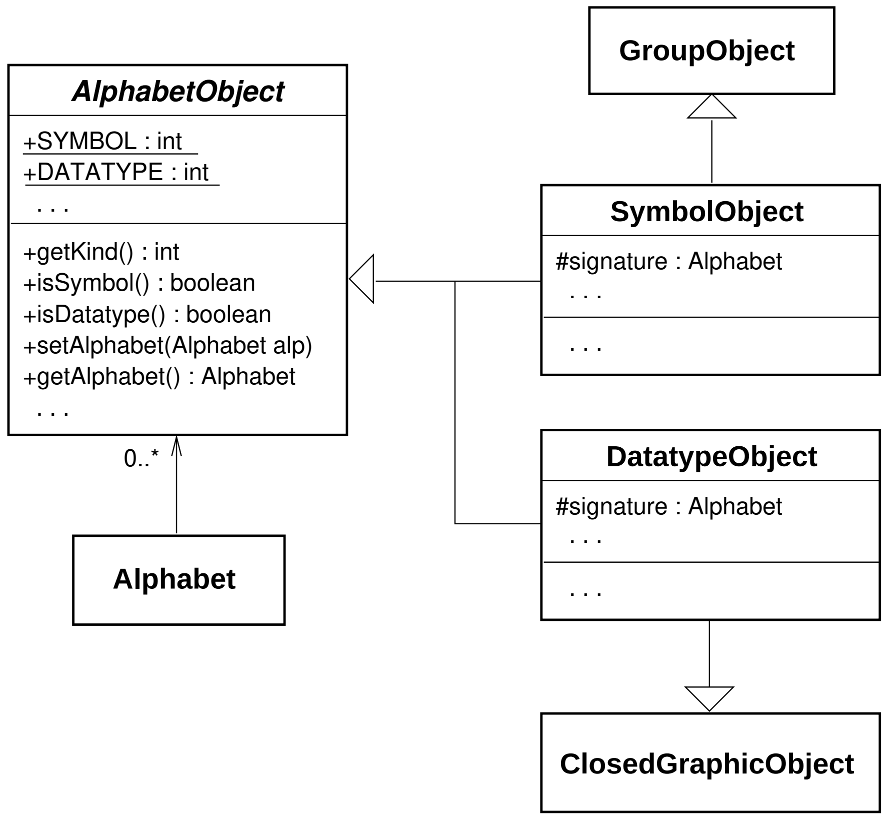


Abbildung 5.13: Klassendiagramm: `AlphabetObject`, Ableitungen

Kapselung

Die Klasse `DatatypeObject` dient dazu, den eigentlichen „Datentypen“ zu kapseln. Dieser Datentyp muß das Interface `Datatype` implementieren. Er ist vergleichbar mit einer Variablen, die den Wert zu einem bestimmten Typ enthält. Die Klasse des Datentyps – eine Klasse, die `Datatype` implementiert – bildet den Typ. Eine Instanz dieser Klasse enthält dazu einen konkreten Wert, der später beliebig oft verändert werden kann. Die bisher verfügbaren Datentypen enthalten Zeichenketten, Zahlen oder Listen. Es wäre aber durchaus denkbar, eine Baumstruktur oder ein Bild als Datentypen zu definieren. Alle dem System bekannten Datentypen müssen sich im Package `genged.language.datatypes` befinden.

In der Klasse `DatatypeObject` existieren die Methoden `setDatatype()` und `getDatatype()`. Damit kann das Attribut `datatype` auf einen konkreten Datentypen gesetzt bzw. abgefragt werden. Die Methode `paintComponent()` überschreibt die gleichnamige Methode von `GraphicObject` und ruft die `paint()`-Methode des gekapselten Datentyps auf.

Der konkrete Wert – Das Interface `Datatype`

Das Interface erweitert einige schon vorhandene Interfaces. Die eigenen Methoden des Interfaces `Datatype` dienen einerseits zum Setzen der Daten, anderseits zum Formatieren. Unter Formatieren verstehen wir das Zuweisen verschiedener Eigenschaften. Weitere Methoden betreffen das Zeichnen des Datentyps innerhalb einer Ansicht.

Andere Interfaces: Das Interface `Cloneable` bietet eine Funktion zum Klonen eines Objekts. Sie wird weiter unten beschrieben. `IconsAvailable` ist ein Interface im Package `genged.gui`, das für die Darstellung des Datentypen in einer Baumstruktur Symbole (Icons) bereitstellt. Dazu gehören die Methoden `getOpenFolderIcon()`, `getClosedFolderIcon()` und `getLeafIcon()`. Das Interface `Serializable` benötigt keine Methoden. Es dient ausschließlich dazu mitzuteilen, daß der Datentyp gespeichert werden kann. Das letzte Interface namens `StateEditable` stammt aus dem Swing-Package `javax.swing.undo` und bietet die Methoden `storeState()` und `restoreState()` an, die den aktuellen Zustand des Datentyps in einer übergebenen Hashtable speichern bzw. wieder daraus rekonstruieren. Diese Methoden werden von dem Undo-Mechanismus benutzt.

Identifikation: In jeder Klasse, die das Interface implementiert, muß die Methode `getName()` einen Namen zurückgeben, der den Typ des Datentypen bezeichnet. Beispiele sind `String`, `Integer` oder `StringList`. Auch die Methode `getDescription()` dient der Beschreibung eines Datentyps. Sie gibt einen kurzen Informationstext zurück, der abhängig von der eingestellten System-Sprache ist.

Wertzuweisung: Durch Aufrufen der Methode `editValue()` erscheint ein Dialogfenster, in dem der aktuelle Wert des Datentyps verändert werden kann. Wie dieses Dialogfenster aussieht und welche Funktionalitäten es bietet, hängt von jedem Datentyp selbst ab. Zur Bearbeitung einer Zeichenkette genügt ein einfacher Dialog, zum Erstellen einer Liste wird dagegen eine komplexere Verwaltung gefordert.

Formatierungen: Jeder Datentyp verfügt über verschiedene Eigenschaften, die sein Format betreffen. Dazu zählen standardmäßig Textfarbe, Hintergrundfarbe, Schriftart und ein Rand. Alle Eigenschaften werden in einer Hashtable gespeichert und sind durch einen Schlüssel ansprechbar. Mit der Methode `getPropertyKeys()` erhält man eine Liste mit allen verfügbaren Schlüsseln. Über die Methoden `setProperty()` und `getProperty()` können Eigenschaften gesetzt oder abgefragt werden. Die Methode `editProperties()` wurde zum Aufruf eines Dialogfensters vorgesehen, mit dem diese Eigenschaften gleichzeitig bearbeitet werden können. Sie ist zur Zeit allerdings in keinem konkreten Datentyp umgesetzt. Geplant ist ein Eigenschafts-Editor, wie er bei Java-Beans [Fla97] verwendet wird. Anhand der Klassen verschiedener Eigenschaften wird ein spezieller Editor automatisch generiert. Die Methode `getPropertyClass()` gibt die Klasse zu einer bestimmten Eigenschaft zurück.

Darstellung: Die Größe der Bounding-Box, die der Wert eines Datentyps in Anspruch nimmt, ist abhängig von dem Wert selbst und seiner Formatierung. Durch Ein-stellen einer größeren Schrift wächst auch die Größe der Bounding-Box. Die Methode `getSize()` bestimmt die Dimension des Datentyps und gibt sie in der Einheit 1/100pt zurück. Um sie in Abhängigkeit einer bestimmten Ansicht in Pixel zu erhalten, existiert die Methode `getDrawingSize()`. In diesem Fall wird der Zoom-Faktor der Ansicht berücksichtigt. Jeder Datentyp muß über eine `paint()`-Methode verfügen, die den Wert des Datentyps an eine bestimmte Position der spezifizierten Ansicht zeichnet.

Transfermöglichkeiten: Im Grammatik-Editor, der das Alphabet um eine Grammatik erweitern soll, wird mit dem an der TU Berlin entwickelten AGG-System [TER][Mel97] zusammengearbeitet. Mit diesem System können Attribute von Diagrammen verwaltet und umgewandelt werden. Jegliche Bearbeitungsmechanismen sind jedoch String-basiert. Daher muß für jeden Datentyp eine Repräsentation geschaffen werden, die sowohl den Wert als auch die Eigenschaften des Datentyps durch eine Zeichenkette ausdrückt. Eine weitere statische Methode, die nicht im Interface aufgenommen werden konnte[3], erstellt anhand dieser Repräsentation einen neuen Datentypen. Die entsprechenden Methoden heißen `getStringRepresentation()` und `createDatatype()`.

Vergleiche und Kopien: Die Methode `equals()` vergleicht den Datentypen mit einem anderen Objekt. Durch `hashCode()` wird ein Hash-Code für einen Datentypen generiert. Da ein Datentyp wie ein primitiver Datentyp in Java, z.B. `int` oder `double`, behandelt werden soll, muß er originalgetreu kopiert werden können. Dazu wird die Methode `clone()` implementiert. Neben dem eigentlichen Wert des Datentyps müssen auch alle Eigenschaften berücksichtigt werden. Für Debug-Zwecke liefert die Methode `toString()` eine Repräsentation des Wertes und aller Eigenschaften des Datentyps.

Abstrakte Oberklasse: Das Interface `Datatype` wird bereits teilweise durch die abstrakte Klasse `AbstractDatatype` implementiert. Alle Datentypen, die sich im Package `genged.language.datatypes` befinden, sind von dieser Klasse abgeleitet. Damit später komplexe Arbeiten, speziell Umwandlungen, mit den Datentypen durchgeführt werden können, implementieren die abgeleiteten Klassen von `AbstractDatatype` verschiedene Methoden, die datentypspezifisch sind. Dabei ist zu beachten, daß später im AGG-System nur die Methoden verwendet werden können, die wieder eine Instanz von *demselben* Datentyp zurückgeben.

Java-Ausdruck

Die Klasse `DatatypeObject` enthält neben dem konkreten Datentypen im Attribut `expression` eine Zeichenkette, die einen Java-Ausdruck darstellen kann. Dabei handelt es sich um direkten Java-Code, z.B. um einen Methodenaufruf. Im Alphabet-Editor wird dieser Ausdruck nicht benötigt. Er muß erst innerhalb des Grammatik-Editors bestimmte Funktionen erfüllen, auf die hier jedoch nicht weiter eingegangen werden soll. Der Java-Ausdruck kann mit der Methode `setJavaExpression()` gesetzt und mit `getJavaExpression()` abgefragt werden. Unter Verwendung der Methode

[3]Java läßt nicht zu, daß innerhalb eines Interfaces statische Methoden existieren.

`setExpressionFont()` kann dem Ausdruck eine Schriftart für dessen Darstellung zugeordnet werden.

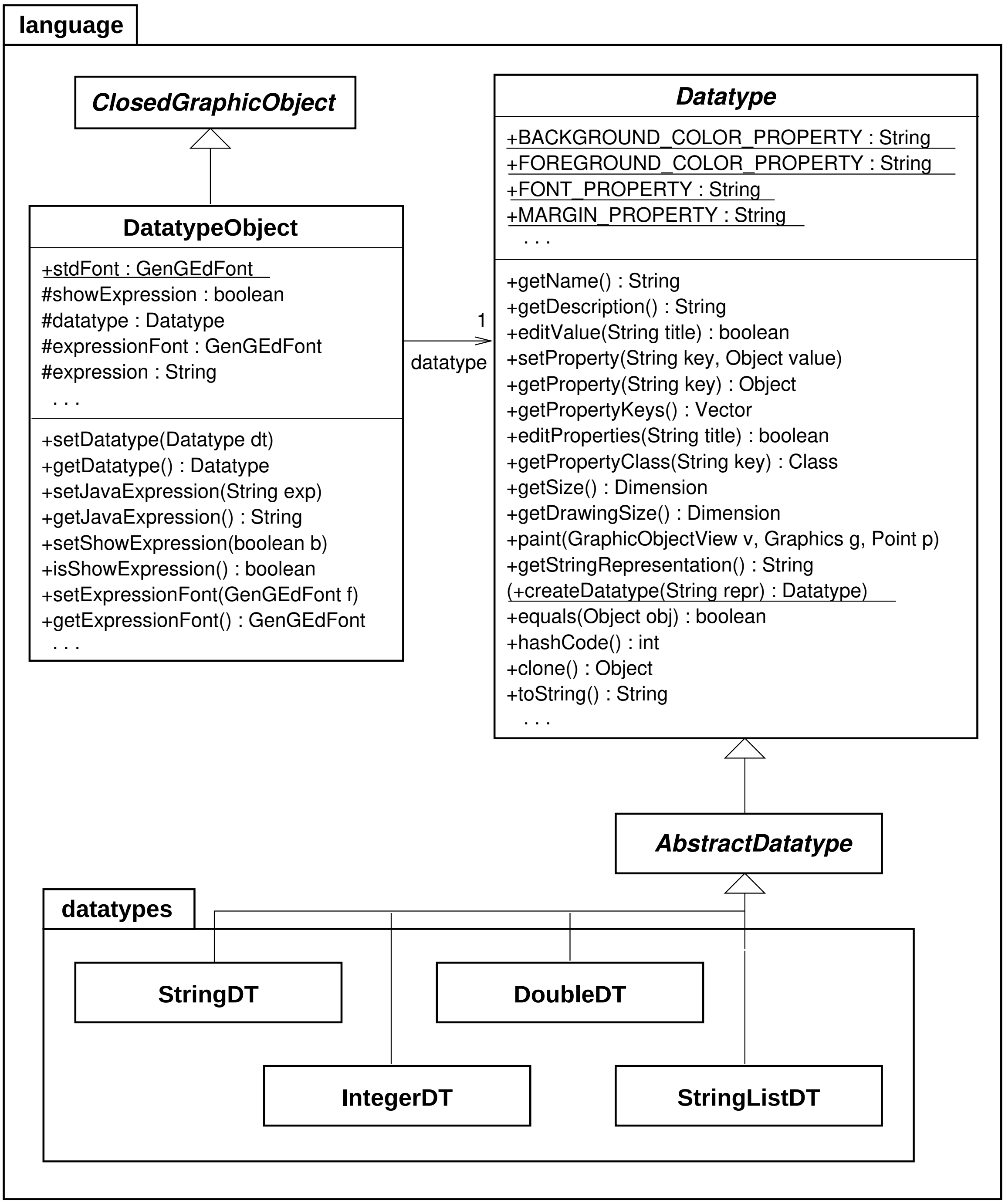


Abbildung 5.14: Klassendiagramm: `DatatypeObject`, Beziehungen

Normalerweise wird bei der Darstellung des Datentyp-Objekts der Wert des Datentyps dargestellt. Mittels der Methode `setShowExpression()` kann das Attribut `showExpression` auf den Wert `true` gesetzt werden. Das hat zur Folge, daß statt dem Datentyp-Wert der Java-Ausdruck angezeigt wird. Die Methode `isShowExpression()` liefert den aktuellen Status zurück.

Die Abbildung 5.14 gibt einen Überblick über die erläuterten Attribute und Methoden und zeigt die Beziehungen zwischen den verschiedenen Klassen auf.

5.6 Operationen

Operationen dienen dazu, grafische Verbindungen zwischen Alphabet-Objekten zu definieren. Dazu wird die Klasse `Operation` verwendet. Sie kann beliebig viele Constraints enthalten, die bei einem Symbol auf dessen Primitiven oder auf einem Datentyp definiert werden.

5.6.1 Interne und externe Operationen

Wir unterscheiden interne und externe Operationen. Eine *interne* Operation gehört immer genau zu einem Alphabet-Objekt. Jedes Alphabet-Objekt besitzt automatisch eine solche Operation. Hier werden alle Constraint-Objekte zusammengefaßt, die *innerhalb* eines Alphabet-Objekts definiert werden. Sie sind für das Layout eines Symbols zuständig. Eine *externe* Operation betrifft dagegen nicht die Darstellung eines einzelnen Objekts, sondern den Zusammenhang zwischen zwei verschiedenen Alphabet-Objekten.

Trotz der beiden Arten gibt es nur *eine* Klasse – die Klasse `Operation`. Sie definiert zwei Konstanten – `INTERN` und `EXTERN` – die dem Attribut `responsibility` zugewiesen werden. Zum Erzeugen einer internen Operation wird dem Konstruktor nur *ein* Alphabet-Objekt übergeben, für eine externe Operation wird er mit *zwei* Objekten aufgerufen. Über die Methoden `getResponsibility()`, `isIntern()` und `isExtern()` kann die Art bzw. die Verantwortlichkeit einer Operation festgestellt werden.

Jede Operation trägt einen im gesamten Alphabet eindeutigen Namen. Der Name kann mit der Methode `setName()` festgelegt werden. Diese Methode testet mit Hilfe der Methode `checkName()`, ob der Name innerhalb des Alphabets schon vorhanden ist. Existiert der Name bereits, wird eine Zahl angehängt, die so lange inkrementiert wird, bis der Name eindeutig ist. Über die Methoden `getName()` und `getIdentifier()` wird der Name der Operation zurückgegeben. Im Falle einer internen Operation wird der Name automatisch vergeben und kann vom Benutzer des Alphabet-Editors nicht verändert werden. Er lautet `Intern(<Name des Alphabet-Objekts>)`.

5.6.2 Source und Target

Die zwei Alphabet-Objekte, zwischen denen eine Operation definiert wird, bezeichnen wir als Source und Target. Bei einer internen Operation sind beide Objekte gleich. Bei einer externen Operation müssen dagegen einige Regeln betreffs der Festlegung von Source und Target beachtet werden. Der Alphabet-Editor dient der Erstellung eines Alphabets, auf dem später mit dem Grammatik-Editor Grammatikregeln definiert werden.

Damit es beim Aufstellen dieser Regeln keine Probleme gibt, dürfen wir einige Dinge im Alphabet-Editor nicht außer acht lassen, die eigentlich erst durch eine Grammatik zum Tragen kommen:

- Ein Source-Objekt stellt ein Objekt dar, das eine Beziehung zu einem Target-Objekt aufbaut.
- Ein Source-Objekt kann später (im Grammatik- und Grafik-Editor) nur *abhängig* vom Target-Objekt existieren.
- Ein Target-Objekt kann dagegen später (im Grammatik- und Grafik-Editor) auch *unabhängig* vom Source-Objekt vorhanden sein.

Source und Target können nicht nachträglich geändert werden. Über die Methoden `getRealSource()` und `getRealTarget()` kann man die Alphabet-Objekte von Source und Target abfragen. Ursprünglich waren dafür die Methoden `getSource()` und `getTarget()` vorgesehen. Diese erfüllen jedoch eine mehrdeutige Funktion, die im nächsten Abschnitt erläutert wird.

5.6.3 Temporäre Instanzen

Bei der Definition von externen Operationen sind wir auf ein Problem gestoßen: Es kann für eine externe Operation als Source und Target dasselbe Objekt verwendet werden, z.B. ein Symbol, das eine Linie darstellt. Nun sollen Constraints zwischen Source und Target definiert werden. Beispielsweise soll die Linie (Source) an die andere Linie (Target) angehangen werden. Da beide Source und Target dasselbe Objekt referenzieren, könnten die Objekte in einem solchen Fall nicht dargestellt werden.

Wir benötigen zur Definition externer Operationen Instanzen. Da diese einzig und allein der Definition einer Operation dienen, bezeichnen wir sie als temporäre Instanzen. Die Klasse `Operation` besitzt einen Modus zur Verwaltung temporärer Instanzen. Er kann – sofern es sich bei der Operation um ein Template handelt – mit der Methode `setTemporaryInstanceMode()` ein- und ausgeschaltet werden. Wird er eingeschaltet, muß man zusätzlich zwei Instanzen von Alphabet-Objekten übergeben. Sie werden in den Attributen `instSource` und `instTarget` abgelegt. Darüber hinaus speichern die Attribute `source` und `target` die Template-Objekte der Instanzen. Die Methoden `getSource()` und `getTarget()` liefern bei aktiviertem Temporary-Instance-Modus Instanzen-Objekte statt der erwarteten Templates.

5.6.4 Verwaltung von Constraints

Erst durch das Hinzufügen von Constraints füllen wir eine Operation mit Inhalt. Ohne Constraints würde eine Operation nur theoretisch existieren, hätte jedoch keine sichtbaren praktischen Auswirkungen. Durch die Verwendung von Constraints kann jedoch das Erscheinungsbild – das Layout – der beteiligten Alphabet-Objekte verändert werden.

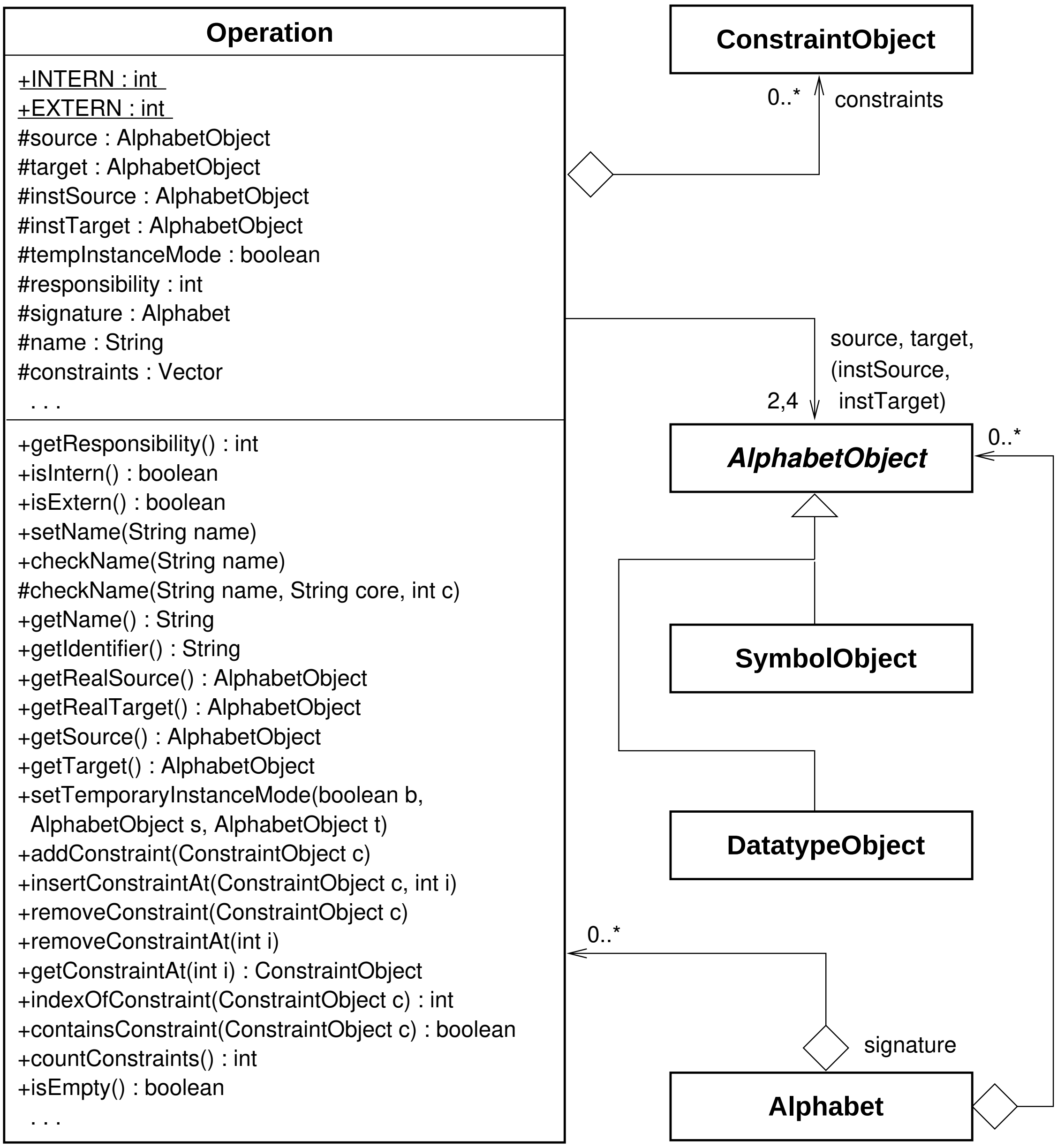


Abbildung 5.15: Klassendiagramm: `Operation`

Mit dem Aufbau eines Constraint-Objekts beschäftigt sich der nächste Abschnitt. Es soll vorerst genügen zu wissen, daß eine Klasse `ConstraintObject` existiert, die konkrete Constraints definiert, welche einer Operation zugewiesen werden können. Dazu existieren die Methoden `addConstraint()` und `insertConstraintAt()`. Die Methoden `removeConstraint()` und `removeConstraintAt()` sind dagegen für das Entfernen eines Constraints zuständig. Alle Constraints werden in dem Attribut `constraints` – einem Vector – abgelegt und besitzen daher einen eindeutigen Index. Die Methode `getConstraintAt()` gibt den Constraint eines bestimmten Indizes zurück. Der Index selbst kann mit der Methode `indexOfConstraint()` ermittelt werden. Über die Methode `containsConstraint()` wird festgestelt, ob einer Operation bereits ein bestimmtes Constraint zugewiesen wurden. Die Methoden `countConstraints()` und `isEmpty()` können genutzt werden, um die Anzahl der Constraints innerhalb einer Operation zu bestimmen, bzw. um zu testen, ob eine Operation überhaupt Constraints enthält.

Die Abbildung 5.15 gibt eine Übersicht zu den vorgestellten Attributen und Methoden sowie zu den Beziehungen verschiedener Klassen.

5.7 Constraints

Constraints stellen durch ein (Un)gleichungssystem Beziehungen zwischen primitiven Grafiken her, indem sie deren Koordinaten, Ausmaße oder andere spezifische Werte benutzen. Bei diesen Constraints handelt es sich um High-Level-Constraints, die mehrere Low-Level-Constraints[4] vereinigen und den Gebrauch von Parametern ermöglichen. Die Definition eines High-Level-Constraints wird ausführlich im Kapitel 7 betrachtet.

Zur Definition einer Beziehung zwischen zwei oder mehreren Primitiven, wird ein Constraint benötigt, der als Parameter die beteiligten Primitiven und numerische Konstanten erhält. Für jeden Constraint ist durch seine Definition festgelegt, welche Arten von Parametern benötigt werden. Jedem Parameter ist ein bestimmter Typ zugeordnet. Alle vorhandenen Typen werden in einer Constraint-Datenbank gespeichert, deren Aufbau ebenfalls in Kapitel 7 erläutert wird.

Jeder Primitive ist ein bestimmter Typ zugeordnet. Einige Primitiven besitzen mitunter denselben Typ. Die Abbildung 5.16 gibt einen Überblick über die objektorientierte Typisierungshierarchie. In der Abbildung sind weitere Typen aufgeführt, die zu keiner Primitive gehören. Als Parametertyp wird allerdings nur noch der Typ `ConstValue` benutzt. Er steht für einen konstanten numerischen Wert vom Typ Integer oder Double.

Die Klasse `ConstraintObject` befindet sich im Package `genged.constsolver`, da sie eng mit dem Constraint-Solver verbunden ist. Jedes Constraint enthält automatisch einen Namen, der innerhalb einer Operation eindeutig sein muß. Er kann mit den Methoden `setName()` und `getName()` geändert bzw. abgefragt werden. Mit der Methode `getIdentifier()` wird der Name um den Namen der zugehörigen Operation erweitert. Er ist dadurch im gesamten Alphabet eindeutig. Die Methode `getDescription()` liefert eine kurze Beschreibung, die aus dem Namen des verwendeten High-Level-Constraints und den spezifizierten Parametern besteht. Soll zusätzlich noch der Name der Operation angegeben werden, kann die Methode `getLongDescription()` aufgerufen werden.

Mit der Methode `setOperation()` wird dem Constraint-Objekt eine Operation zugewiesen. Jeder Constraint gehört zu genau *einer* Operation. Die Methode `getOperation()` gibt diese Operation zurück. In Abhängigkeit der Operation ist auch ein Constraint entweder *intern* oder *extern*. Die bereits aus der Klasse `Operation` bekannten Methoden `getResponsibility()`, `isIntern()` und `isExtern()` existie-

[4]eine einzelne Formel, z.B. eine Ungleichung

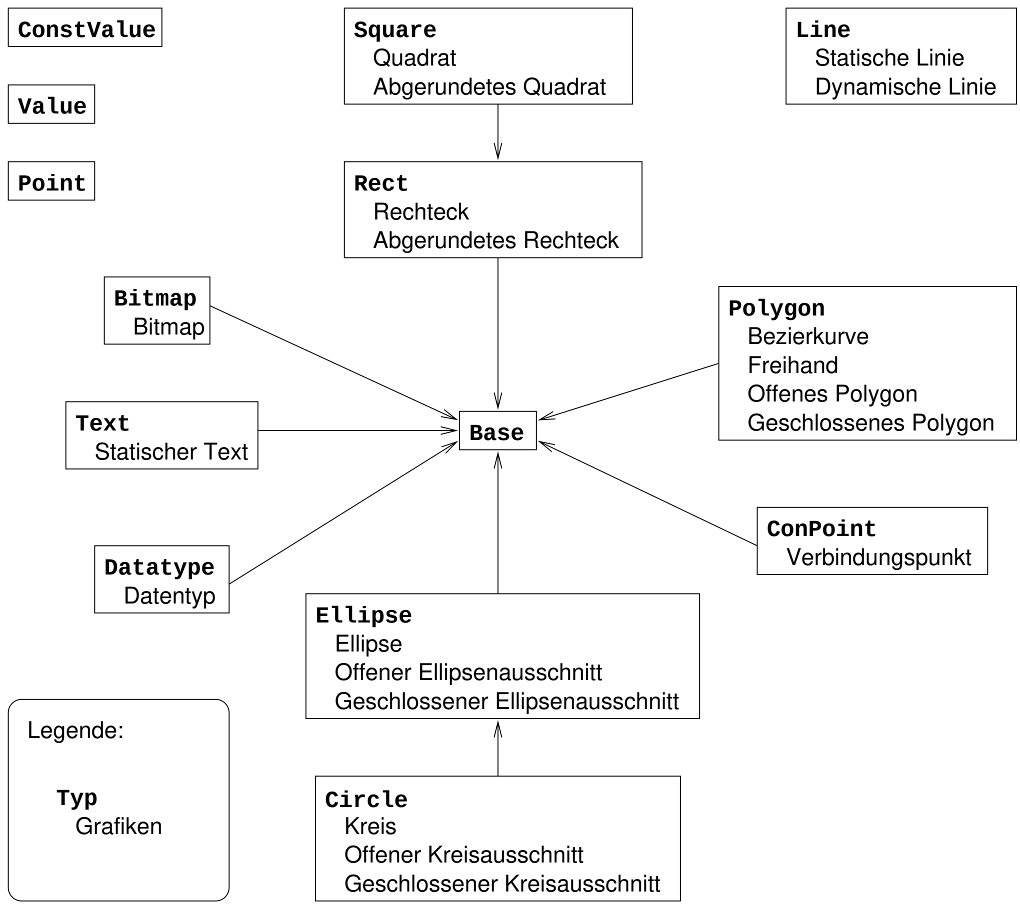


Abbildung 5.16: Objektorientierte Typisierungshierarchie der Primitiven

ren auch in der Klasse `ConstraintObject`. Ähnlich ist auch die Verwendung eines Temporary-Instance-Modus. Ist dieser Modus aktiviert, werden den Parametern temporäre Instanzen von Primitiven zugewiesen.

Um einem Constraint-Objekt einen High-Level-Constraint der Constraint-Datenbank zuzuweisen, wird die Methode `setConstraint()` benutzt. Der aktuelle Constraint kann durch Aufruf von `getConstraint()` ermittelt werden. Zum Speichern der Parameter eines Constraint-Objekts existieren mehrere Vector-Attribute:

- In dem Attribut `types` werden Referencen auf den Typ eines Parameters gespeichert. Ein Typ wird durch die Klasse `Type` im Package `genged.constsolver` beschrieben.
- Das Attribut `parameters` speichert die eigentlichen Parameter-Objekte. Es gibt drei grundsätzlich verschiedene Arten von Parametern:
  - Primitiven: Sie werden als Instanzen von `GraphicObject` gespeichert.
  - Numerische Werte: Eine Zahl wird als `String` gespeichert.

  - Parametermenge: Eine Menge wird als ein Vector gespeichert, der entweder Primitiven *oder* numerische Werte enthält. Verschachtelte Parametermengen sind nicht möglich.

- In `origins` wird der Ursprung eines Parameters gespeichert. Dadurch kann bei externen Operationen zwischen Source- und Target-Parametern unterschieden werden. Wir verwenden drei Konstanten:

  - `BELONGS_TO_NOTHING`: Bei dem zugehörigen Parameter handelt es sich entweder um einen konstanten numerischen Wert oder um einen nicht spezifizierten Parameter.
  - `BELONGS_TO_SOURCE`: Der zugehörige Parameter ist eine Primitive des Source-Objekts.
  - `BELONGS_TO_TARGET`: Der zugehörige Parameter ist eine Primitive des Target-Objekts.

  Für eine Parametermenge wird ein Vector angelegt, der für jedes Element der Menge den Ursprung speichert.

- Das Attribut `instParameters` ist vergleichbar mit `parameters`. Befindet sich das Template eines Constraint-Objekts im Temporary-Instance-Modus, werden hier die Instanzen für die Parameter abgelegt. Deren Templates werden jedoch in dem Attribut `parameters` referenziert.

Zum Setzen von Parametern dienen die Methoden `setParameterAt()` und `setParameters()`. Jeder Parameter wird ausführlich auf seine Gültigkeit getestet. Dazu existiert die Methode `isParameterValid()`. Abgefragt werden können die verschiedenen Parameterbestandteile unter anderem mit den Methoden `getParameter()`, `getRealParameter()`, `getParameterAt()` und `getOrigin()`. Die Methode `getRealParameter()` liefert auch im Template-Instance-Modus das Template eines Parameters.

Ein Constraint-Objekt kann nur im Constraint-Solver verwendet werden, wenn alle Parameter spezifiziert wurden und gültige Objekte enthalten. Um die Gültigkeit eines Constraint-Objekts zu testen, wird die Methode `checkValidity()` aufgerufen. Ist der Constraint ungültig, setzen wir das Attribut `valid` auf den Wert `false`.

Eine wichtige Methode für die Instanz eines Constraint-Objekts ist `matchParameters()`. Ihr werden als Parameter Source und Target der zugehörigen Operation übergeben. Dabei muß es sich ebenfalls um Instanzen handeln. Die Methode ermittelt nun die Instanzen für alle Constraint-Parameter, indem es die Templates der Primitiven mit den Instanzen der Primitiven des übergebenen Source- und Target-Objekts vergleicht.

Für die Beschreibung der restlichen Methoden verweisen wir auf die Online-Dokumentation. Die Abbildung 5.17 zeigt die erläuterten Attribute und Methoden in einem Klassendiagramm.

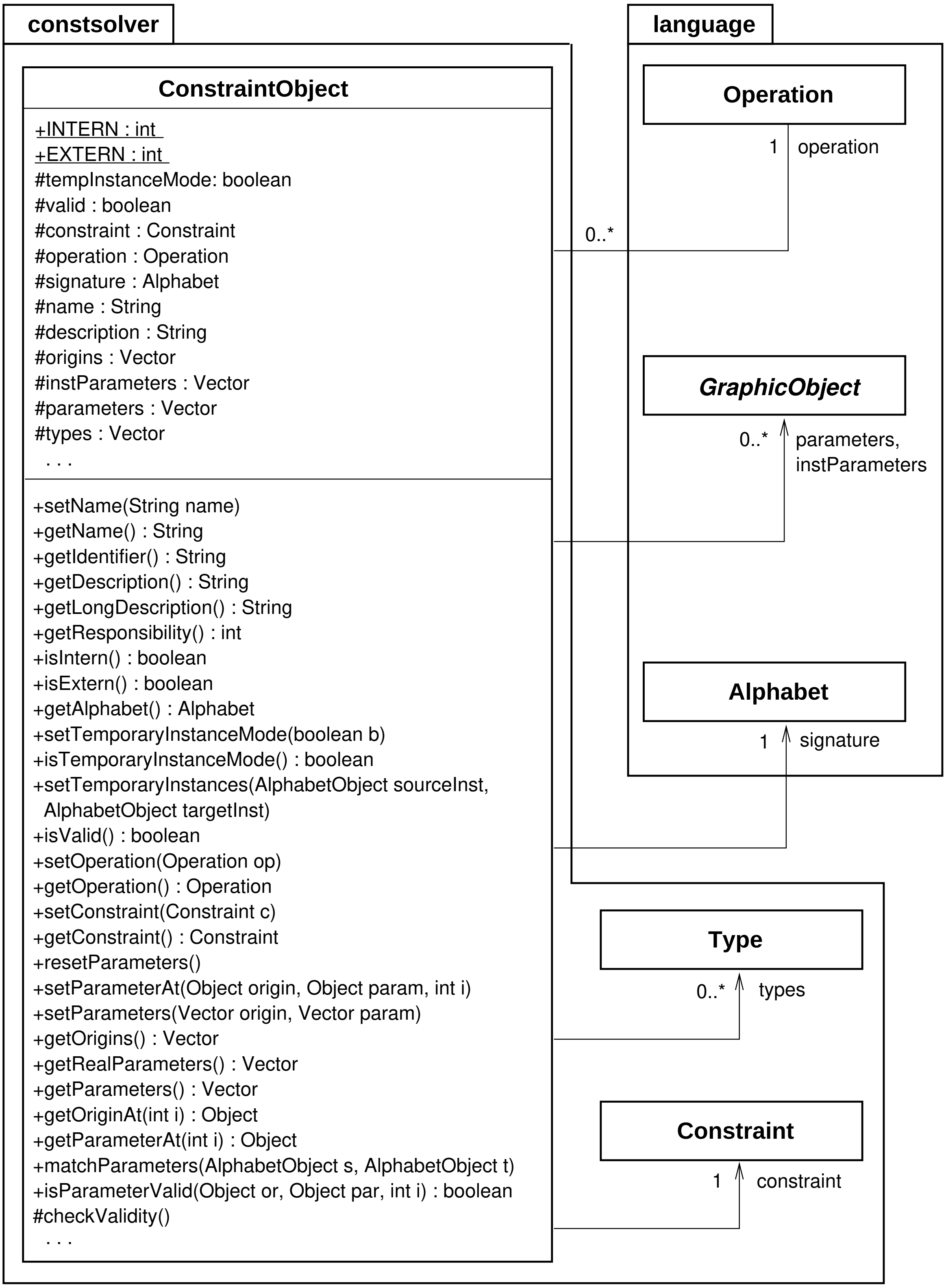


Abbildung 5.17: Klassendiagramm: `ConstraintObject`

5.8 Alphabet als Datenbasis

Das Gesamtmodell des Alphabet-Editors bildet die Klasse `Alphabet`. Es enthält alle definierten Alphabet-Objekte – Symbole und Datentypen – sowie die darauf definierten Operationen mit ihren Constraints. Jedes Alphabet wurde zu einer bestimmten Zeit von einem bestimmten Benutzer erstellt. Um diese Angaben festzuhalten, werden sie in den Attributen `createTime` und `owner` gespeichert und können mittels der Methoden `getOwner()` und `getCreateTime()` abgefragt werden.

Zwei weitere Attribute, `initTime` und `changeTime`, enthalten ebenfalls Timestamps. Beide Attribute werden bei dem Erstellen, Laden oder Speichern eines Alphabets neu initialisiert. Bei jeder Änderung wird das Attribut `changeTime` geändert. Dafür sorgt die Methode `setChanged()`. Durch Vergleich beider Attribute stellt die Methode `hasChanged()` fest, ob sich das Alphabet seit dem letzten Speichern verändert hat. Wird eine Änderung durch den Undo-Mechanismus rückgängig gemacht, setzt die Methode `setChangeTime()` den Timestamp zurück auf den letzten Wert.

5.8.1 Funktionalitäten

Die Klasse `Alphabet` bietet Funktionalitäten zur Verwaltung von Symbolen, Datentypen und Operationen. Verschiedenste Methoden dienen dem Abfragen und Filtern spezieller Objekte. Außerdem sind Funktionen zum Speichern und Laden von Alphabeten enthalten.

Symbole und Datentypen: Alphabet-Objekte werden in dem Attribut `alphabet` – einem Vector – referenziert. Mit den Methoden `addAlphabetObject()` und `insertAlphabetObjectAt()` können Symbole oder Datentypen hinzugefügt werden. Die Methoden `removeAlphabetObject()` und `removeAlphabetObjectAt()` dienen dagegen dem Entfernen von Objekten. Jedes Alphabet-Objekt besitzt einen eindeutigen Namen. Die Methode `existAlphabetObject()` überprüft anhand des Namens, ob ein bestimmtes Alphabet-Objekt bereits im Alphabet vorhanden ist; die Methode `getAlphabetObject()` gibt das Objekt mit dem spezifizierten Namen zurück. Der Index eines Symbols oder Datentypen kann mit der Methode `indexOfAlphabetObject()` ermittelt werden, die Anzahl aller registrierter Objekte mit `countAlphabetObjects()`. Soll eine Liste mit den Namen aller Alphabet-Objekte zurückgegeben werden, ist die Methode `getAlphabetObjectNames()` aufzurufen. Um Referenzen auf alle Objekte zu erhalten, können die Methoden `getAlphabetObjects()` und `enumerateAlphabetObjects()` genutzt werden. Durch Aufruf der Methode `setNewPosition()` kann einem Alphabet-Objekt ein neuer Index zugeordnet werden.

In dem Attribut `alphabet` werden sowohl Objekte der Klasse `SymbolObject` als auch Objekte der Klasse `DatatypeObject` gespeichert. Mit den Methoden `getSymbols()`, `enumerateSymbols()`, `getSymbolNames()`, `getDatatypes()`, `enumerateDatatypes()` und `getDatatypeNames()` können diese Alphabet-

Objekte getrennt nach Kategorien zurückgegeben werden. Interessant ist die Methode `getGraphicObject()`, die anhand eines im Alphabet eindeutigen Identifiers das dazugehörige Grafikobjekt ermittelt und zurückgibt.

Interne und externe Operationen: Operationen werden in dem Attribut `operations` gespeichert. Zur Verwaltung existieren ähnliche Methoden wie für Alphabet-Objekte. Die Methoden `addOperation()`, `insertOperationAt()` sowie `removeOperation` und `removeOperationAt()` dienen dem Hinzufügen und Entfernen von Operationen. Das betrifft sowohl interne als auch externe. Alle Objekte müssen auch hier einen eindeutigen Namen besitzen. Die Existenz einer Operation wird mit der Methode `existOperation()` überprüft. Um eine Liste aller bekannten Operationsnamen aufzurufen, kann die Methode `getOperationNames()` genutzt werden. Um zwischen internen und externen Operationen zu unterscheiden, existieren `getInternOperationNames()` und `getExternOperationNames()`. Der Name einer internen Operation enthält automatisch den Namen des dazugehörigen Alphabet-Objekts. Wird der Name des Alphabet-Objekts geändert, muß die Methode `updateNames()` aufgerufen werden. Sie sorgt für eine Anpassung verschiedener Operationsnamen.

Anstatt alle Methoden zum Thema Operationen aufzuzählen, verweisen wir einmal mehr auf die Online-Dokumentation. So gibt es beispielsweise viele verschiedene Methoden zum Abfragen von Operationen anhand spezieller Eigenschaften, z.B. des Source- oder Target-Objekts.

In der Abbildung 5.18 sind einige der Attribute und Methoden des Alphabets sowie Beziehungen verschiedener Klassen dargestellt.

5.8.2 Benachrichtigung bei Änderungen

Im Zusammenhang mit einem Alphabet wurde konsequent das in Kapitel 4 beschriebene Listener-Pattern umgesetzt, um über Änderungen zu informieren. Das betrifft zum einen Änderungen, die direkt mit der Klasse `Alphabet` in Zusammenhang stehen, beispielsweise das Hinzufügen eines Symbols oder einer Operation. Andererseits können Änderungen auch innerhalb eines Alphabet-Objekts oder einer Operation ausgeführt werden, z.B. wenn eine Primitive oder ein Constraint eingefügt wird. In diesem Fall muß das Alphabet informiert werden, um gezielt Änderungsinformationen an interessierte Objekte schicken zu können.

Es werden insgesamt drei verschiedene Listener eingerichtet. Dabei handelt es sich um Interfaces, die von interessierten Klassen implementiert werden können. Alle Interfaces befinden sich in dem Package `genged.language.event`. An die einzelnen Methoden der Listener wird jeweils eine Instanz der Klasse `AlphabetEvent` übergeben, welche die Änderungen beschreibt, die stattgefunden haben.

Dazu generiert das Objekt, das verändert wurde oder die Veränderung ausgelöst hat, einen `AlphabetEvent` und ruft damit in dem betreffenden Alphabet die Methode `processAlphabetEvent()` auf. Diese Methode verteilt den Event an alle angemeldeten Listener der entsprechenden Kategorie.

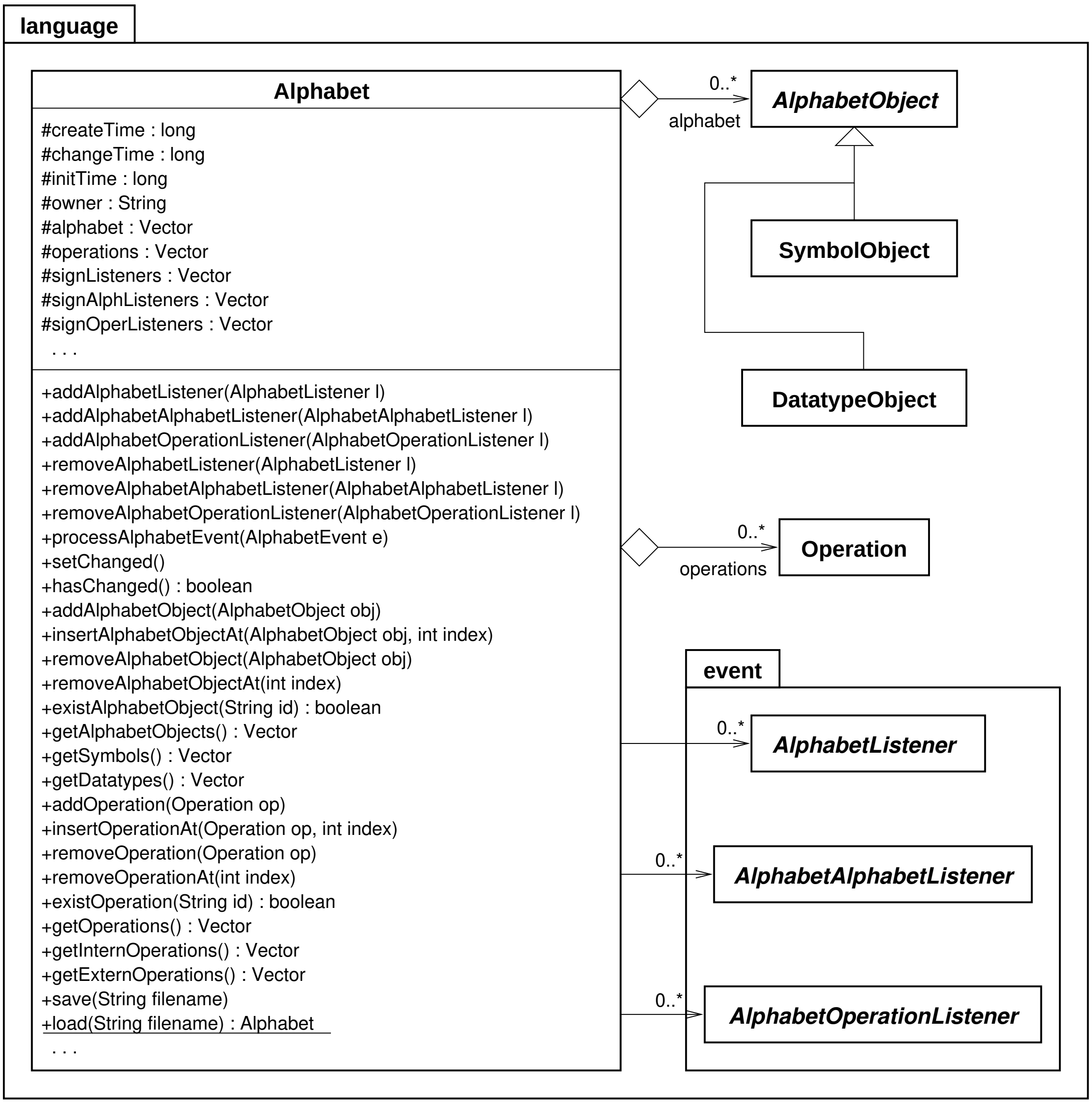


Abbildung 5.18: Klassendiagramm: `Alphabet`, Allgemein

Interface `AlphabetListener`**:** Dieses Interface enthält nur zwei Methoden. `signatureInstalled()` wird aufgerufen, wenn ein neues Alphabet im Alphabet-Editor erstellt oder geladen wurde. Bei `jeder` Änderung der Signature wird die Methode `signatureChanged()` aufgerufen, allerdings ohne Angaben zur Änderung selbst.

Jedes Objekt, das den Listener implementiert, kann sich mit der Methode `addAlphabetListener()` beim Alphabet registrieren bzw. mit der Methode `removeAlphabetListener()` wieder abmelden.

Interface `AlphabetAlphabetListener`**:** Listener dieser Art werden über jegliche Änderungen von Alphabet-Objekten informiert. Das betrifft Symbole mit ihren Primitiven genauso wie Datentypen. Die Methoden `symbolInserted()`, `symbolRemoved()`, `datatypeInserted()` und `datatypeRemoved()` werden aufgerufen, wenn Symbole oder Datentypen in das Alphabet eingefügt oder entfernt wurden. Ändern sich dagegen interne Attribute der Symbole oder Datentypen, werden die Methoden `symbolChanged()` und `datatypeChanged()` benutzt. Innerhalb eines Symbols kann es im Gegensatz zum Datentypen noch weitere spezielle Änderungen geben. Werden in einem Symbol Primitiven hinzugefügt oder entfernt, ruft das Alphabet die Methoden `graphicInserted()` und `graphicRemoved()` auf, bei internen Änderungen eines Grafikobjekts die Methode `graphicChanged()`.

Jedes Objekt, das diesen Listener implementiert, kann sich mit der Methode `addAlphabetAlphabetListener()` beim Alphabet registrieren bzw. mit der Methode `removeAlphabetAlphabetListener()` wieder abmelden.

Interface `AlphabetOperationListener`**:** Hier wird zwischen Änderungen von internen und externen Operationen und ihren Constraints unterschieden. Die Methoden `internalOperationInserted()`, `internalOperationRemoved()`, `externalOperationInserted()` und `externalOperationRemoved()` werden direkt vom Alphabet aufgerufen, wenn eine Operation hinzugefügt oder entfernt wurde. Ändern sich dagegen Attribute einer Operation, werden die Methoden `internalOperationChanged()` und `externalOperationChanged()` benutzt. Für Änderungen, die Constraints innerhalb der Operationen betreffen, existieren die Methoden `internalConstraintInserted()`, `internalConstraintRemoved()`, `externalConstraintInserted()` und `externalConstraintRemoved()`. Diese Methoden werden aufgerufen, sobald ein Constraint in eine Operation eingefügt oder aus einer Operation entfernt wird. Ändern sich die Eigenschaften eines Constraints, werden die Methoden `internalConstraintChanged()` und `externalConstraintChanged()` benutzt. In einer zukünftigen Programmversion soll es möglich sein, zur Laufzeit des Programms die Constraint-Datenbank eines Alphabets zu wechseln. Diese Aktion würde den Aufruf der Methode `databaseChanged()` auslösen.

Jedes Objekt, das diesen Listener implementiert, kann sich mit der Methode `addAlphabetOperationListener()` beim Alphabet registrieren bzw. mit der Methode `removeAlphabetOperationListener()` wieder abmelden.

Klasse `AlphabetEvent`**:** An jede der oben beschriebenen Listener-Methoden wird eine Instanz dieser Klasse übergeben. In diesem Event sind alle Daten enthalten, die die Änderung eines Objektes betreffen. Dadurch wird das Push-Modell für das Listener-Pattern verwirklicht. In der Klasse ist eine Referenz auf das Objekt enthalten, das diesen Event ausgelöst hat. Neben dem Alphabet selbst kann dieser Event auch von Grafikobjekten, Symbolen, Datentypen, Operationen oder Constraints erzeugt worden sein. Zur Übermittlung der Informationen wird dennoch generell die Klasse `Alphabet` benutzt. Das auslösende Objekt muß nicht unbedingt identisch sein mit dem Objekt, das verändert wurde. Die jeweilige „Quelle" des Events wird beim Aufruf der Event-Methode `getSource()` zurückgegeben. Der Event enthält außerdem eine Referenz auf das Alphabet, das geändert wurde, sowie eine Konstante, die die Art der Änderung beschreibt. Die Methoden `getAlphabet()` und `getID()` dienen der Abfrage dieser Informationen.

Bei den meisten Events wird das Objekt, das verändert wurde, in dem Attribut `concObject` gespeichert – es wird bei der Benutzung der Methode `getConcernedObject()` zurückgegeben. Wird ein Objekt in ein anderes Objekt eingefügt oder aus einem entfernt, liefert die Methode `getParent()` das neue bzw. alte übergeordnete Objekt.

Zusätzlich sind mitunter noch ein oder zwei weitere Zahlen enthalten, auf die über die Methoden `getNumber1()` und `getNumber2()` zugegriffen werden kann. Die erste der beiden Zahlen wird im Zusammenhang mit Konstanten verwendet. Durch sie lassen sich Events näher beschreiben, die bei internen Änderungen eines Objekts erzeugt werden. Die zweite Zahl wird dagegen frei vergeben und meistens dazu benutzt, einen Index zu speichern. Die folgende Tabelle 5.3 listet die Konstanten auf, die einem `AlphabetEvent` als „Code“ übergeben werden können:

Code	Bedeutung
POSITION	Die Position eines Objekts innerhalb eines übergeordneten Objekts wurde verändert. Das kann eine Primitive innerhalb eines Gruppenobjekts bzw. eines Symbols, ein Constraint innerhalb einer Operation oder ein Alphabet-Objekt bzw. eine Operation innerhalb eines Alphabets betreffen.
SELECTION	Ein Grafikobjekt wurde selektiert oder deselektiert. Dabei kann es sich um eine Primitive, ein Symbol oder einen Datentypen handeln.
NAME	Der Name eines Objekts wurde geändert.
RESET	Die Parameter eines Constraint-Objekts wurden zurückgesetzt.
DATABASE	Die Referenz auf eine Constraint-Datenbank, die von einer Operation oder einem Constraint gehalten wird, hat sich verändert.
CONSTRAINT	Der verwendete High-Level-Constraint eines Constraint-Objekts wurde verändert.
VALIDITY	Die Gültigkeit eines Constraints wurde verändert.
ACTIVITY	Ein Constraint wurde am Constraint-Solver an- oder abgemeldet.
PARAMETERS	Die Parameter eines Constraints wurden geändert.
OPERATION	Die Operation, zu der ein Constraint gehört, wurde neu gesetzt.
LINE_ATTRIB	Das Linien-Attribut einer Primitiven wurde geändert.
FILL_ATTRIB	Das Füll-Attribut einer Primitiven wurde geändert.
SOURCETARGET	Source und Target einer Operation wurden geändert. Das betrifft Objekte, die im Temporary-Instance-Modus gesetzt werden.
VISIBILITY	Die Sichtbarkeit einer Primitive wurde verändert.

Tabelle 5.3: Konstanten zur näheren Beschreibung einer Alphabet-Änderung

Die Abbildung 5.19 zeigt die kompletten Attribute und Methoden aller Klassen, die den Benachrichtungsmechanismus im Zusammenhang von Änderungen am Alphabet bereitstellen. Die Abbildung 5.20 stellt anhand eines Sequenzdiagramms dar, wie verschiedene Listener über eine Namensänderung einer Primitive informiert werden.

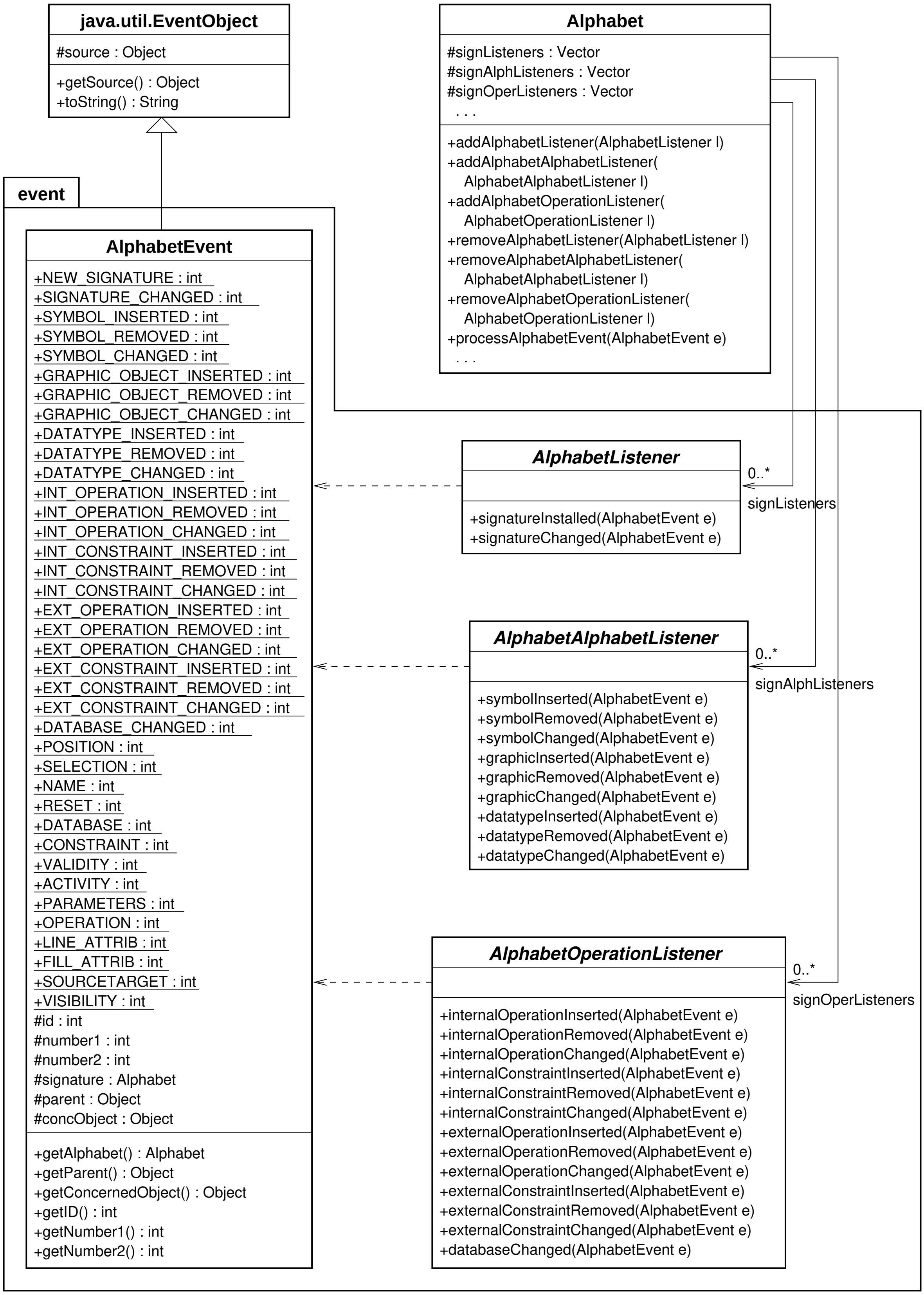


Abbildung 5.19: Klassendiagramm: `Alphabet`, Benachrichtigungsmechanismus

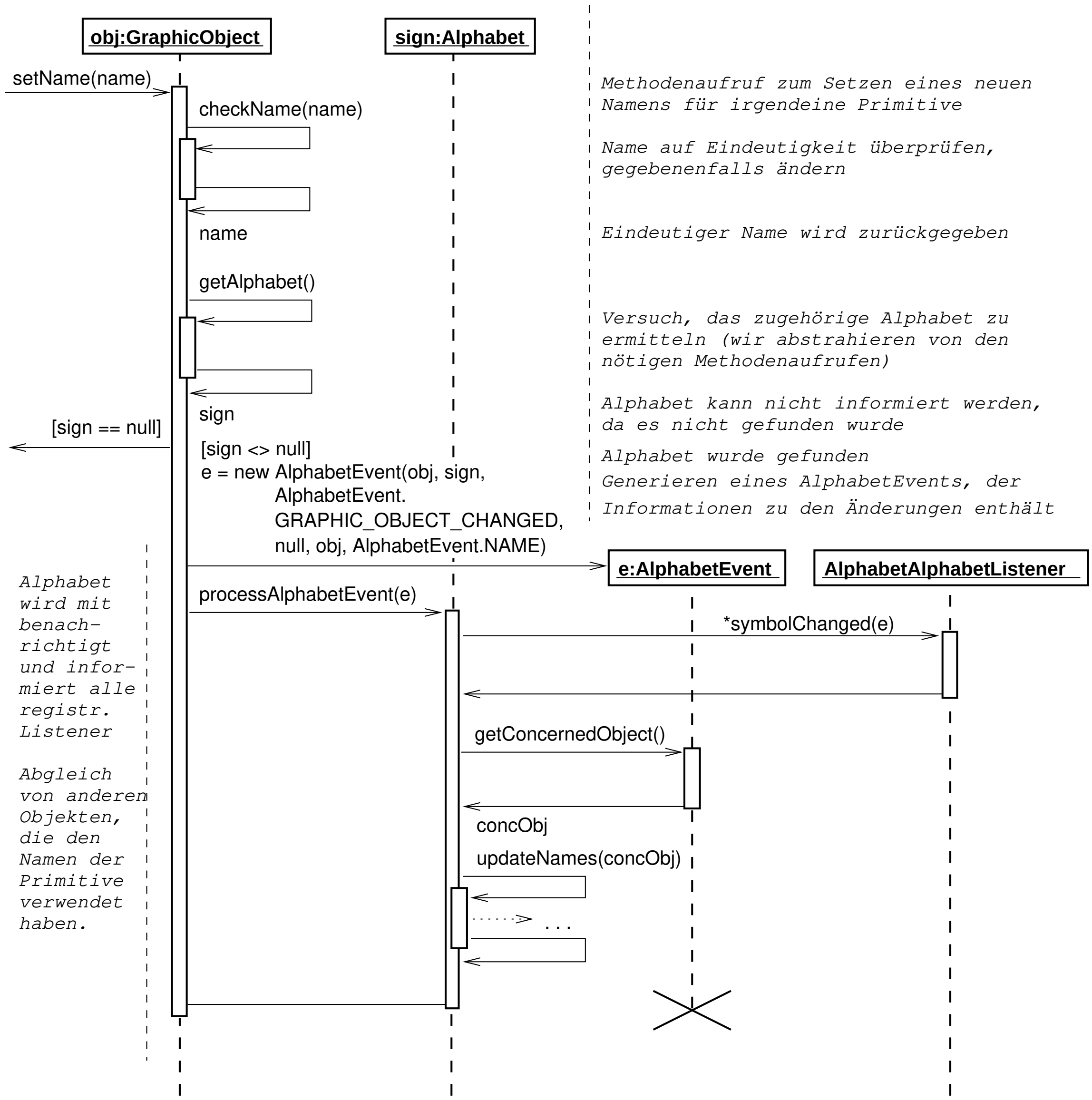


Abbildung 5.20: Sequenzdiagramm: Benachrichtigung von Alphabet-Listenern

5.8.3 Speichern und Laden

Jede Klasse, die in irgendeiner Weise an dem Aufbau des Alphabets beteiligt ist, implementiert das Interface `java.io.Externalizable`. Es enthält zwei Methoden: `writeExternal()` und `readExternal()`. Durch die Implementierung dieser Methoden legt jede Klasse selbst das Format fest, in welchem sie ihre Attribute speichert bzw. von einem Datenträger liest.

Ein Alphabet enthält ziemlich komplexe Strukturen, die sich durch viele „Querverweise“ zwischen Primitiven, Constraints, Symbolen, Datentypen und Operationen ergeben. Beim Speichern der einzelnen Bestandteile werden diese Verweise in eindeutige

Namen umgewandelt. Dazu werden die Identifier der Objekte verwendet. Wird das Alphabet wieder eingelesen, muß die Struktur neu aufgebaut werden. Die Identifier werden wieder auf konkrete Objekte abgebildet. Diese Aufgabe übernimmt eine zusätzliche Methode `postReadExternal()`, die in Abhängigkeit der jeweiligen Klassen verschiedene Parameter erhält.

Um zukünftige Versionen verschiedener Klassen zu unterstützen, enthält jede Klasse in dem statischen Attribut `serialVersionUID` eine Seriennummer. Diese Nummer vom Typ `long` kann mit dem Programm `serialver`, das zusammen mit Java ausgeliefert wird, generiert werden [Fla97]. Wird eine Klasse grundlegend verändert, sollte die alte Seriennummer in das Array `oldSerialVersionUID` eingefügt und eine neue Nummer erzeugt werden. In der Methode `readExternal()` kann die beim Speichern verwendete Seriennummer ausgelesen und entsprechend reagiert werden. Dadurch ist Versions-Management für die einzelnen Bestandteile des Alphabets möglich.

Kapitel 6

Allgemeine Mechanismen

Es gibt einige allgemeine Mechanismen, die von verschiedenen Komponenten des Systems benutzt werden. Daher fällt es schwer, sie in den Kapiteln unterzubringen, die sich speziell mit einer Hauptkomponente beschäftigen. Dennoch sind sie sehr wichtig, so daß ihnen ein eigenes Kapitel gewidmet wurde.

6.1 Internationalisierung

Bei dem Alphabet-Editor handelt es sich um ein mehrsprachiges System. Zur Zeit werden die Sprachen Deutsch und Englisch unterstützt. Weitere könnten in Zukunft folgen. Java selbst ermöglicht die Verwendung mehrerer Sprachen durch sogenannte Resource-Bundles. Störend ist jedoch die dezentrale Plazierung der Text-Ressourcen. Daher wurde mit der Klasse `LanguageText` ein eigener Mechanismus implementiert, der Text-Ressourcen, die in einem speziellen Format vorliegen, beim Programmstart einliest. Über statische Methoden der Klasse `GenGEd` kann durch Verwendung eines Schlüssels der entsprechende Text in der aktuell eingestellten Sprache erhalten werden. Abbildung 6.1 zeigt die Zusammenhänge.

Klasse `LanguageText`: Diese Klasse wurde nach dem Singleton-Pattern implementiert. Sie kann daher nur einmal instanziiert werden. Sie dient als Datenbank für jegliche Text-Ressourcen des GENGED-Systems. Initialisiert wird `LanguageText` durch die Klasse `GenGEd`. Dazu wird zuerst die Methode `getSharedInstance()` aufgerufen. Danach sorgt die Methode `readStringResources()` der Klasse `GenGEd` dafür, daß alle Ressourcen-Dateien, die das Gesamtsystem betreffen, eingelesen werden. Diese Dateien befinden sich alle in dem Verzeichnis `genged/resources/strings`. Für jede Datei wird die Methode `addLanguageText()` aufgerufen. Beim Erzeugen der einzelnen Editoren werden jeweils weitere zwei Dateien hinzugefügt (`guiGlobal.res` und `guiLocal.res`). Für jeden Editor existiert ein separates Ressourcen-Verzeichnis im Verzeichnis `genged/resources/alphabeteditor`. `LanguageText` erweitert die von Java zur Verfügung gestellte Klasse `java.util.Observable`. Alle Klassen, die bei einer Änderung der aktuellen Sprache informiert werden sollen, müssen das In-

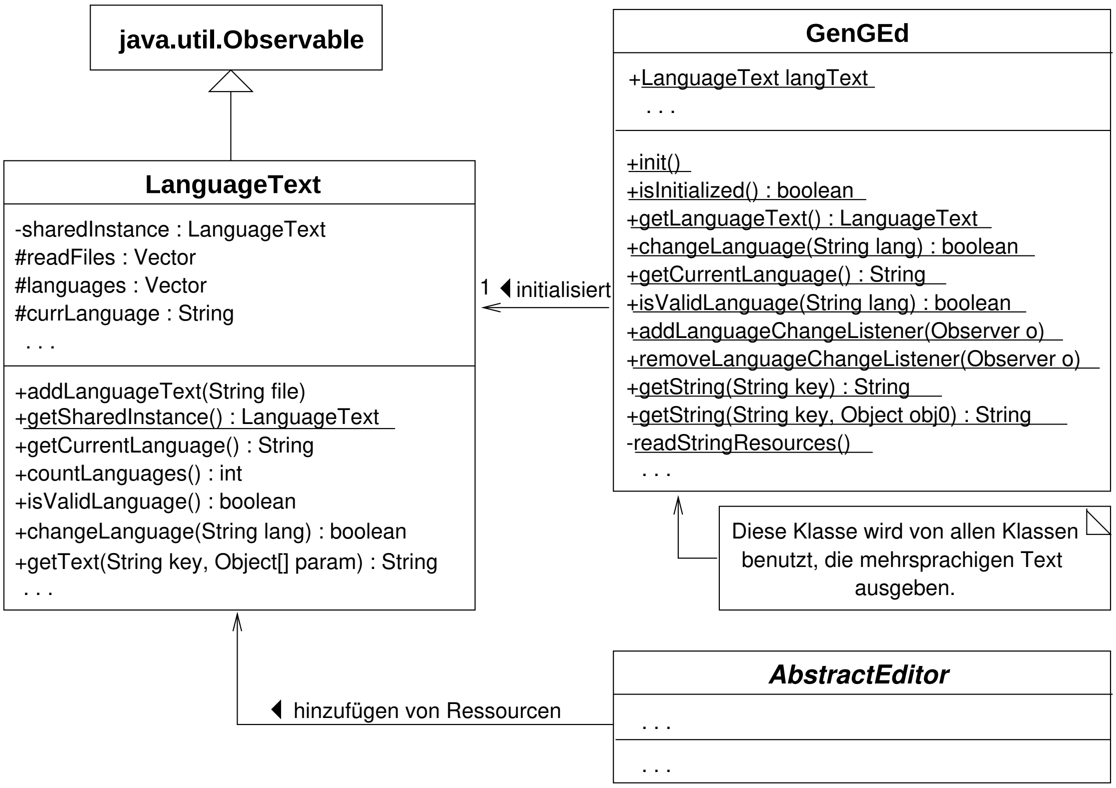


Abbildung 6.1: Klassendiagramm: `LanguageText` und `GenGEd`

terface `java.util.Observer` implementieren und sich anmelden. Dieser Mechanismus entspricht dem Observer-Pattern. Die entsprechenden Klassen können bei ihrer Benachrichtigung dafür sorgen, daß textuelle GUI-Komponenten aktualisiert werden. Die Klasse `GenGEd` bildet alle interessanten Methoden auf entsprechende Methoden der Klasse `LanguageText` ab.

Klasse `GenGEd`: Wir wollen in diesem Abschnitt ausschließlich die Funktionalität betrachten, die mit der Internationalisierung zu tun hat. Jede Klasse, die mehrsprachigen Text zur Ausgabe benötigt, nutzt die statischen Methoden der Klasse `GenGEd`. Beim Starten des Programms müssen daher die Ressourcen-Datenbanken in den Speicher geladen werden. Dazu wird die Methode `init()` aufgerufen. Über `isInitialized()` kann abgefragt werden, ob `GenGEd` bereits initialisiert wurde. Am häufigsten wird die Methode `getString()` aufgerufen, die in elf verschiedenen Varianten vorliegt. Ihr wird der Schlüssel des gewünschten Textes übergeben. Außerdem können bis zu zehn Objekte als weitere Parameter dienen. In jedem Ressourcen-Text dürfen bis zu zehn Platzhalter vorkommen, die zur Laufzeit durch die übergebenen Objekte ersetzt werden. Um die Sprache zu wechseln, wird die Methode `changeLanguage()` benutzt. Zur Zeit sind als Parameter „Deutsch“ und „English“ erlaubt. Ist der Wechsel erfolgreich, gibt die Methode `true` zurück. Klassen, die über den Sprachenwechsel informiert werden möchten, müssen das Interface `java.util.Observer` implementieren. Sie können sich über die Methoden `addChangeLanguageListener()` und `removeChangeLanguageListener()` als Interessenten an- und abmelden.

Format der Ressourcen-Dateien: Am Anfang einer Ressourcen-Datei werden in Anführungszeichen die verwendeten Sprachen – durch Kommata getrennt – aufgeführt. Die Angabe endet mit einem Semikolon. Danach folgen die Ressourcen-Einträge. Jeder Eintrag besteht aus einem Schlüssel und den verschiedensprachigen Texten. Dabei müssen die Texte in derselben Reihenfolge angegeben werden wie die Sprachen zu Beginn der Datei. Die Bestandteile stehen jeweils in Anführungszeichen und werden durch Kommata getrennt. Der Eintrag endet wieder mit einem Semikolon. Ist die Anzahl der Texte kleiner als die Anzahl der angegebenen Sprachen, wird automatisch der erste Texteintrag wiederholt. Innerhalb eines Textes können bis zu zehn Platzhalter verwendet werden, deren Inhalt erst zur Laufzeit übergeben wird. Sie werden von 0 bis 9 durchnummeriert in geschweifte Klammern geschrieben. Für Bemerkungen können Kommentare benutzt werden, die durch ein Doppelkreuz (#) eingeleitet werden und automatisch mit dem Zeilenende enden. Die folgenden Zeilen geben ein kurzes Beispiel:

```
# Dies ist eine Beispieldatei für String-Ressourcen
"Deutsch", "English";

# Hier folgen die Ressourcen-Einträge:
# "Schlüssel", "Deutscher Text", "Englischer Text";

"Yes", "Ja", "Yes";
"No", "Nein", "No";

"ERR_100",
"Die Datei {0} wurde nicht gefunden.", # {0} ist ein Platzhalter
"File {0} doesn’t exist.";

"NotReady", "Dieser Eintrag wird auch für Englisch benutzt.";
```

6.2 Zeichenfläche

Der Symbol- und Verbindungs-Editor und das Test-System benötigen eine Zeichenfläche, um die Symbolgrafiken darzustellen. Bei den Grafiken handelt es sich um eine Zusammenstellung von Vektorgrafiken, die alle von der abstrakten Klasse `GraphicObject` im Package `genged.language` abgeleitet sind. Mehrere dieser Grafiken können durch Anwendung der Klasse `GroupObject` gruppiert werden. Der Aufbau des Alphabets und der dazugehörigen Grafiken wurde bereits ausführlich in Kapitel 5 behandelt.

Die Klasse `AbstractDrawArea` implementiert eine sehr mächtige Zeichenfläche, von der die einzelnen Editoren eine spezialisierte Zeichenfläche ableiten. `AbstractDrawArea` ist trotz ihres Namens keine abstrakte Klasse, sondern verfügt bereits über sämtliche Funktionalitäten. Der Name soll darauf hinweisen, daß jeder Editor eigene Zeichenflächen von dieser Klasse ableitet. Da sehr viele Attribute und Methoden vorhanden sind, beschränken wir uns auf die wichtigsten und auf die Darstellung von Zusammenhängen. Für mehr Informationen sei auf die Online-Dokumentation verwiesen, die mit dem Programm `javadoc` erzeugt wurde. Die Abbildung 6.2 stellt aus Platzgründen nur die Beziehungen des `AbstractDrawArea` zu verschiedenen Komponenten dar.

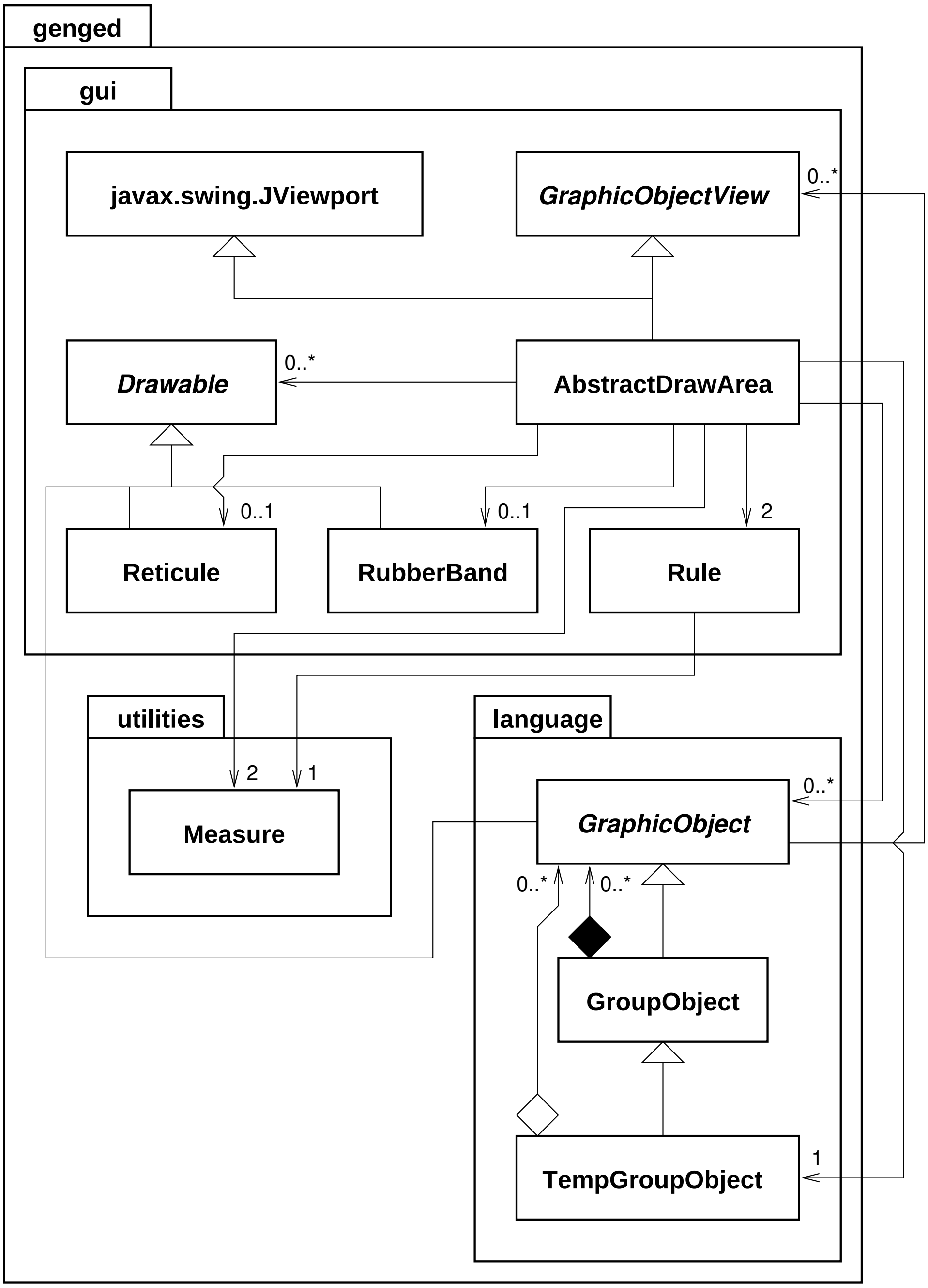


Abbildung 6.2: Klassendiagramm: `AbstractDrawArea`

`AbstractDrawArea` erbt alle Eigenschaften der Klasse `JViewport`, die auf den ersten Blick für die Implementierung einer Zeichenfläche völlig unbrauchbar erscheint. Auf den zweiten Blick ist sie jedoch die einzige Klasse, mit der wir folgende Anforderungen erfüllen können:

1. Scrollbar-Benutzung (es wird immer nur ein Ausschnitt dargestellt)
2. Beliebige Größe (nur begrenzt durch den Definitionsbereich von `int`)
3. Möglichkeit zum Verkleinern/Vergrößern (Maßstab)

Scrollbare Arbeitsflächen sind nichts Ungewöhnliches. Die Swing-Klassen von Java unterstützen bereits den Bildlauf für beliebige grafische Komponenten wie die Klassen `Canvas` oder `Image`. Java hält in diesem Fall jedoch die gesamte Ansicht im Arbeitsspeicher, obwohl nur ein kleiner Ausschnitt am Bildschirm erscheint. Je größer die unterliegende Fläche wird, desto langsamer arbeitet der Bildlauf. Abgesehen davon ist die Größe der Gesamtfläche stark beschränkt. Berechnungen haben ergeben, daß bei einem Arbeitsspeicher von 64 MB und einer Farbtiefe von 32 Bit nur Bildgrößen von ca. 4000x3000 Pixeln möglich wären. Das hört sich viel an, wenn man bedenkt, daß eine Arbeitsfläche auch mit der halben Größe auskommen würde. Wir fordern jedoch, daß ein Maßstab festgelegt werden kann, beispielsweise um bestimmte Bereiche zu vergrößern. In diesem Fall wäre jedoch nur eine maximale Vergrößerung um den Faktor 2 möglich. Da wir vektororientierte Grafiken benutzen, muß in unserem Fall auch nicht die gesamte Zeichenfläche im Speicher gehalten werden. Um den Scrollmechanismus, den Java bereits anbietet, dennoch nutzen zu können, wurde daher die Klasse `JViewport` als Superklasse benutzt. Viele Methoden wurden überschrieben und mit „neuem“ Inhalt gefüllt, der an unsere Anforderungen angepaßt ist.

Als *interne Einheit* der Zeichenfläche benutzen wir 1/100pt (Punkte). Sie bietet selbst bei einem Maßstab des Faktors 10 noch eine zufriedenstellende Auflösung und wird auch im CAD-Bereich verwendet. Die Koordinaten und Größen der dargestellten Vektorgrafiken werden ebenfalls in dieser Einheit angegeben. Die maximalen Ausmaße einer Grafik werden zur Zeit durch den java-internen Datentyp `int` auf 2.147.483.647 1/100pt begrenzt. Das entspricht ungefähr 757,5km und sollte auch für zukünftige Anwendungen ausreichen.

6.2.1 Verwaltung der Grafiken

Normale Grafiken: Mit den Methoden `add()` und `insertAt()` können der Zeichenfläche Grafiken hinzugefügt werden. Diese Grafiken müssen von der Klasse `GraphicObject` abgeleitet sein. Jede Grafik hat eine bestimmte Position und Größe. In diesem Bereich zeichnet sie sich selbst, nachdem ihre `paint()`-Methode durch die Klasse `AbstractDrawArea` aufgerufen wurde. Als Parameter werden die Zeichenfläche selbst (`GraphicObjectView`) sowie der zu verwendende grafische Kontext (`Graphics`) übergeben. Beim Zeichnen muß der aktuelle Zoom-Faktor der Zeichenfläche berücksichtigt werden. `GraphicObject` kann diesen

Faktor über die Methode `getZoom()` des übergebenen `GraphicObjectView` abfragen. Über die Methoden `remove()` und `removeAll()` können eine oder alle Grafiken von der Zeichenfläche entfernt werden.

Temporäre Grafiken: Neben den normalen Grafiken werden mitunter noch sogenannte temporäre Grafiken dargestellt, die auf einer höheren „Ebene“ gezeichnet werden. Dabei handelt es sich normalerweise um bewegliche Objekte, die wenig Platz auf der Zeichenfläche benötigen, jedoch sehr oft aktualisiert werden müssen. Beispiele dafür sind ein Fadenkreuz (Klasse `Reticule`), das die aktuelle Mausposition anzeigt, oder ein Gummiband (Klasse `RubberBand`), welches mitunter bei Drag-And-Drop-Aktionen erscheint. Über die Methoden `addTemp()` und `removeTemp()` können eigene temporäre Objekte hinzugefügt oder entfernt werden. Alle diese Objekte müssen das Interface `Drawable` implementieren. Auch hier muß beim Zeichnen der Zoom-Faktor beachtet werden.

6.2.2 Zeichen-Ebenen

Die Abbildung 6.3 zeigt die Reihenfolge, in der die Objekte der Zeichenfläche gezeichnet werden. Dazu existieren verschiedene Ebenen. Die Objekte verschiedener Ebenen werden nur unter einer bestimmten Bedingung dargestellt. So müssen beispielsweise das Raster (Ebene 3) oder das Fadenkreuz (Ebene 6) mit den Methoden `setGridVisible()` und `setReticuleVisible()` eingeschaltet werden. Ein Gummiband wird nur dargestellt, wenn der Benutzer auf der Zeichenfläche die Maus mit gedrückter Maustaste bewegt und sich die Zeichenfläche in einem bestimmten Arbeitsmodus (interner Zustand) befindet.

Die meiste Zeit benötigt aufgrund ihrer Anzahl das Zeichnen der normalen Grafiken, die mit der `add()`-Methode hinzugefügt wurden. Diese Grafiken werden deshalb zuerst auf ein Image gezeichnet, das exakt die Ausmaße des Viewports besitzt und extra im Speicher gehalten wird. Ändert sich eine der Grafiken, wird ausschließlich der betroffene Bereich über die Methode `addDirtyRegion()` als „dirty“[1] erklärt. In der Regel wird diese Methode durch die Klasse `GraphicObject` automatisch aufgerufen. Dieser Aufruf bewirkt allerdings kein Neuzeichnen der Zeichenfläche. Meistens sind mehrere Grafiken und damit mehrere Bereiche der Zeichenfläche vom Neuzeichnen betroffen. Daher werden die „schmutzigen“ Bereiche zuerst gesammelt. Die Klasse, die Veränderungen an den Grafiken vorgenommen hat, muß am Ende dafür sorgen, daß die Methode `repaint()` in der Klasse `AbstractDrawArea` aufgerufen wird. Sie bewirkt, daß nur die „schmutzigen“ Bereiche neu gezeichnet werden. Durch Aufruf der Methode `validate()` wird dagegen die gesamte Zeichenfläche neu erstellt.

Haben sich nur Objekte der Ebenen 3 bis 7 geändert, genügt es, die Methode `repaintTemp()` aufzurufen. Sie nutzt zum Zeichnen der Ebenen 1 und 2 das oben erwähnte Image, das bereits fertig im Speicher vorliegt.

[1] engl. schmutzig, unsauber

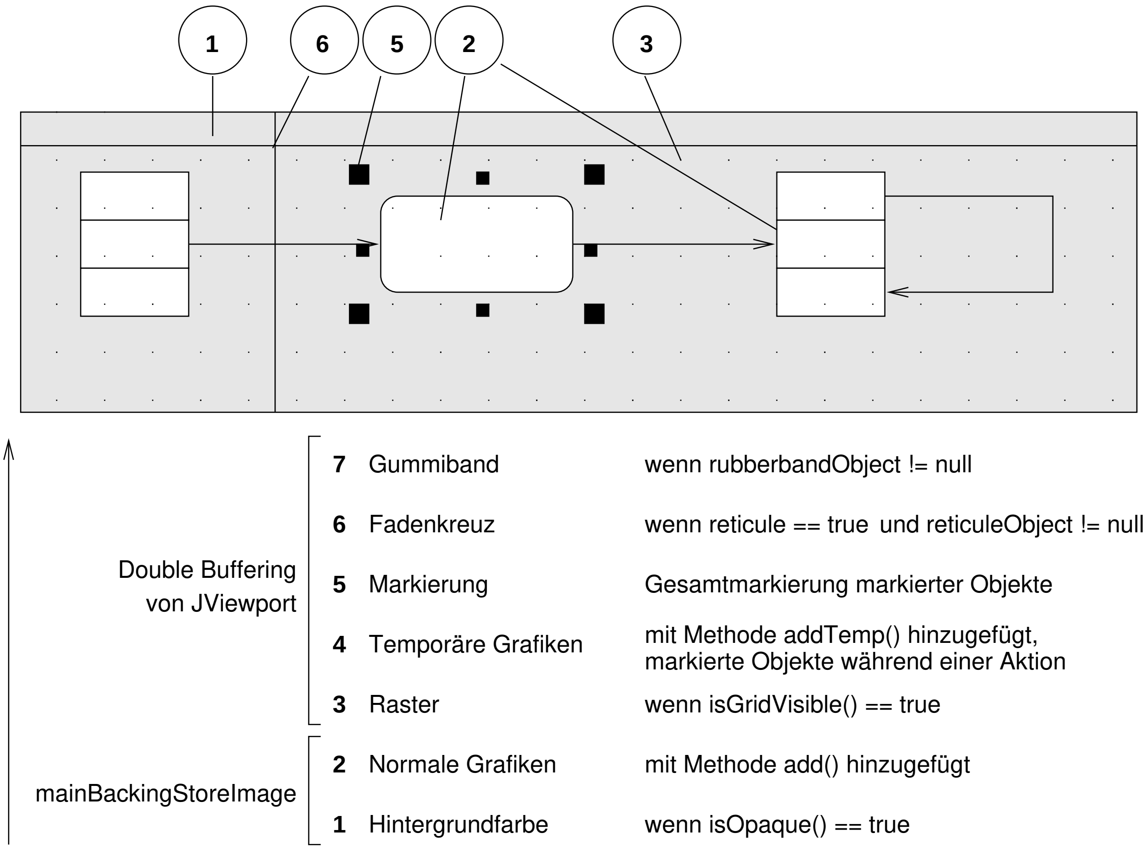


Abbildung 6.3: Zeichenreihenfolge der Objekte einer Zeichenfläche

6.2.3 Arbeitsmodi

Die Klasse `AbstractDrawArea` kennt verschiedene Zustände, sogenannte *Arbeitsmodi*. Sie orientieren sich stark an den bereits bekannten Bearbeitungsmodi der grafischen Primitiven. Zwischen den Arbeitsmodi kann mit der Methode `setWorkingMode()` umgeschaltet werden. Der aktuelle Zustand wird mit `getWorkingMode()` abgefragt. Je nach Arbeitsmodus reagiert die Zeichenfläche unterschiedlich auf Mausaktionen des Benutzers. Sie arbeitet dabei sehr eng mit der Klasse `GraphicObject` bzw. deren Unterklassen zusammen. Aufgabe der Zeichenfläche ist es nicht nur, die Objekte dieser Klasse darzustellen, sondern sie auch zu manipulieren. Besonders ist hier die Klasse `TempGroupObject` zu erwähnen, die temporäre Gruppen von Objekten verwalten kann, um verschiedene Transformationen durchzuführen. Fast könnte man von einer Symbiose beider Klassen sprechen. Die Klasse `AbstractDrawArea` definiert für jeden ihr bekannten Arbeitsmodus eine Konstante, die der oben genannten Methode übergeben werden muß. Die Tabelle 6.1 listet alle Arbeitsmodi auf und gibt jeweils eine kurze Erläuterung. Zur genaueren Beschreibung aus Benutzersicht verweisen wir auf das ausführliche Benutzerhandbuch im Anhang dieser Arbeit.

Arbeitsmodus	Erläuterung
CHOICE_MODE	Dieser Modus dient der Auswahl von Grafiken. Durch einen Mausklick wird eine Grafik markiert. Zum Markieren mehrerer Objekte kann ein Gummiband-Rechteck aufgezogen werden. Die markierten Grafiken können verschoben, skaliert, rotiert oder verzerrt werden.
ZOOM_MODE	Es kann ein Gummiband-Rechteck aufgezogen werden, das die neuen Ausmaße des Viewport festlegt. Dadurch kann in die Zeichenfläche hineingezoomt werden.
ADD_NODE_MODE	In diesem Modus können einer Primitiven Knoten hinzugefügt werden, sofern sie das zuläßt.
MOVE_NODE_MODE	Die Knoten einer Primitiven können verschoben werden, um die Form der Grafik zu beeinflussen.
REMOVE_NODE_MODE	Dieser Modus dient dem Löschen von Knoten innerhalb einer Primitiven.
SMART_NODES_MODE	In diesem Modus können Knoten „intelligent“ bearbeitet werden. Es können Knoten hinzugefügt, verschoben und entfernt werden. Dabei werden jedoch die Bearbeitungsfähigkeiten und -rechte der Primitiven berücksichtigt.
ADD_NORMAL_CONPOINT_MODE	Einer Primitiven können frei positionierte Verbindungspunkte hinzugefügt werden.
ADD_CATCHED_CONPOINT_MODE	Dieser Modus bietet die Möglichkeit, Verbindungspunkte im Fangmodus an vordefinierte Positionen einzufügen.
MOVE_CONPOINT_MODE	In diesem Modus können Verbindungspunkte an eine beliebige Position verschoben werden.
MOVE_CATCHED_CONPOINT_MODE	Hier können Verbindungspunkte im Fangmodus an eine vordefinierte Position verschoben werden.
REMOVE_CONPOINT_MODE	Dieser Modus dient dem Löschen von Verbindungspunkten.
CHOICE_ALL_MODE	Dieser Modus unterscheidet sich von dem Modus CHOICE_MODE dadurch, daß Verbindungpunkte von selektierten Objekten angezeigt und ebenfalls ausgewählt werden können.
OTHER_MODE	In diesem Modus werden keinerlei Aktionen ausgeführt. Stattdessen können Mauslistener von außen an der Zeichenfläche angemeldet werden, die selbst Aktionen initiieren.

Tabelle 6.1: Arbeitsmodi der Klasse AbstractDrawArea

6.3 Strukturansicht

Bei der Strukturansicht handelt es sich um eine Baumstruktur, die auf der linken Seite des Arbeitsfensters als Inhalt eines Tabulator-Panels dargestellt wird. Mit dieser Ansicht, die Teile des Modells enthält, bekommt der Benutzer einen Überblick über alle Daten, die zur Zeit in dem jeweiligen Editor bearbeitet werden können. Der konkrete Inhalt ist jedoch von Editor zu Editor verschieden. Wir wollen daher nur auf grundlegende Entwurfsentscheidungen eingehen.

6.3.1 Modelle

Die Klasse `JTree`, die zu den Swing-Klassen von Java gehört, diente uns als Basis für die Strukturansicht. Sie wurde durch die Klasse `GenGEdTree` erweitert, um selbst formatierte Tooltips auf den verschiedenen Einträgen anzuzeigen. Die Klasse `JTree` benutzt zwei Modelle zur Darstellung der Baumstruktur: Ein Datenmodell und ein Modell, das die Markierungen von Einträgen verwaltet. Um die Daten unseres Alphabets in einer Baumstruktur darzustellen, boten sich uns zwei Vorgehensweisen an:

1. Unser eigenes Modell, das Alphabet mit seinen Komponenten, müßte verschiedene Interfaces implementieren, die von `JTree` gefordert wären. Der Vorteil dieser Variante wäre der direkte Kontakt zwischen der Strukturansicht und der Daten. Nachteil wäre, daß die Daten nicht mehr unabhängig von einer Ansicht sind, und daß durch die vielen neuen Methoden die schon vorhandene Komplexität der Klassen weiter steigt.

2. Es müßten für das Daten- und das Selektierungsmodell jeweils Adapterklassen implementiert werden, die ein Modell enthalten, das speziell auf das Alphabet und seine Komponenten zugeschnitten ist, ohne diese jedoch zu beeinflussen. Einer der Vorteile wäre, daß dadurch die Daten unabhängig von ihrer Darstellung sind und deren Darstellung an jeden Editor angepaßt werden kann.

Wir haben die zweite Variante umgesetzt, da sie die Wiederverwendbarkeit einzelner Komponenten unterstützt und dadurch flexibler einsetzbar ist. Für jeden einzelnen Editor wurden daher zwei Adapterklassen implementiert: ein Daten- und ein Selektierungsmodell. Beide Klassen befinden sich in dem jeweiligen Package des Teil-Editors.

6.3.2 Einträge der Strukturansicht

In dem Package `genged.gui.tree` sind verschiedene Klassen zu finden, die die von Java zur Verfügung gestellten Klassen um eigene Funktionalitäten erweitern. So werden hier für jede Teilkomponente des Alphabets Knoten – das sind Einträge der Baumstruktur – implementiert, die die Daten einer bestimmten Komponente darstellen. Dazu zählen die Klassen `GraphicObjectNode`, `GroupNode`, `ConPointNode`, `SymbolNode` und `DatatypeNode`. Sie veranschaulichen die grafischen Bestandteile eines Alphabets. Die unsichtbaren Bestandteile werden durch die Knoten `OperationNode` und `ConstraintNode` abgebildet.

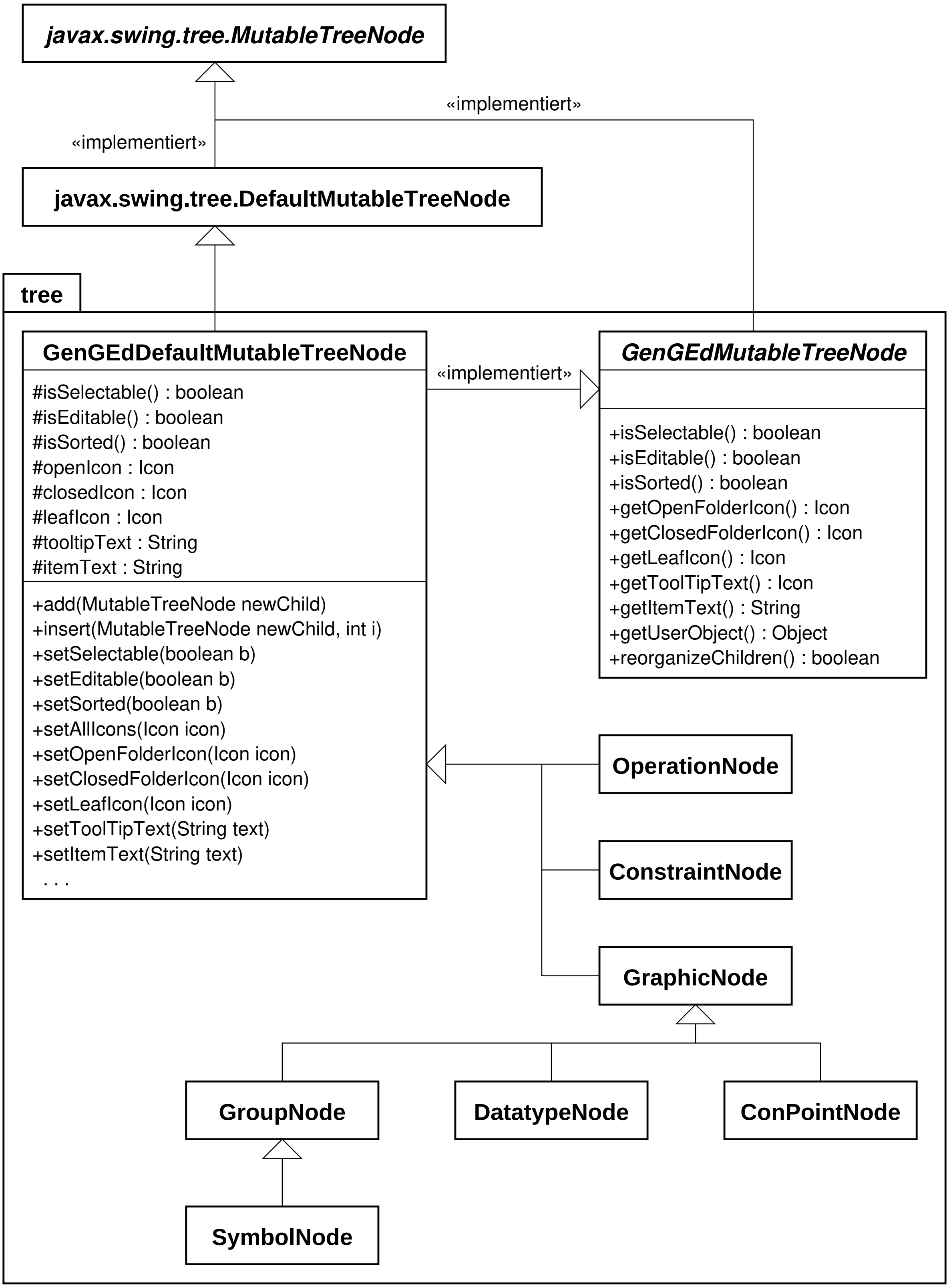


Abbildung 6.4: Klassendiagramm: Knotenklassen der Strukturansicht

Alle Klassen, die wir bisher aufgezählt haben, erweitern die von uns implementierte Klasse `GenGEdDefaultMutableTreeNode`. Sie stellt die von unserem System geforderten Grundfunktionalitäten zur Verfügung. Unter anderem unterstützt sie sortierte Knoten. Alle Untereinträge, die in einen solchen Knoten eingefügt werden, sortiert die Methode `insert()` zwischen die bestehenden Einträge ein. Die Abbildung 6.4 bietet einen Überblick über Vererbungsbeziehungen der verschiedenen Knoten. Außerdem werden einige Attribute und Methoden dargestellt.

6.4 Benutzeraktionen

Die Aktionen eines Benutzers können in zwei verschiedene Kategorien eingeteilt werden. Es gibt Funktionalitäten, die über ein Menü, über eine Symbolleiste oder durch Tastendruck ausgewählt werden können. Andere Aktionen werden durch einen oder mehrere Mausklicks in die Strukturansicht oder Zeichenfläche eingeleitet bzw. ausgeführt.

Für alle Aktionen der ersten Kategorie wurde ein Mechanismus entwickelt, der fähig ist, verschiedenste Merkmale von Aktionen zu verwalten. Moderne Software-Systeme fordern, daß ein und dieselbe Benutzeraktion in der Regel mehrere Möglichkeiten der Benutzerinteraktion bietet. Ein und dieselbe Funktion kann so beispielsweise über einen Menüpunkt *oder* einen Symbol-Button aufgerufen werden. Unser Mechanismus kapselt daher eine Benutzeraktion und stellt beliebig viele Menüeinträge und Symbol-Buttons sowie viele andere Eigenschaften zur Verfügung. Eine sehr wichtige Eigenschaft betrifft die Internationalisierung, die Verwendung mehrerer Sprachen. Außerdem kann an zentraler Stelle festgelegt werden, ob beispielsweise alle Bedienelemente der Aktion verfügbar (enabled) oder nicht verfügbar (disabled) sein sollen, oder ob eine Checkbox selektiert oder deselektiert ist. Alle Menüeinträge und Symbol-Buttons, die zu einer Aktion gehören, reagieren auf solche Änderungen. In der Klasse `EditorAction` im Package `genged.alphabeteditor` wurde dieser Mechanismus umgesetzt, um das System offen und leicht änderbar zu gestalten. `EditorAction` erweitert die Klasse `AbstractAction`, die zu den Swing-Klassen von Java gehört.

6.4.1 Eigenschaften von Benutzeraktionen

In einer Ressourcen-Datei können folgende Eigenschaften definiert werden:

- Name
- Menüeintrag
- Tooltip-Text
- Text für die Statuszeile
- Verschiedene Icons für Symbol-Buttons und Menüeinträge, die abhängig sind vom Status der Aktion (normal, selektiert, disabled, ...)

- Art der Aktion (normale Aktion, Checkbox, Radio-Button[2])
- Methode, die bei Ausführung der Aktion aufgerufen werden soll

Alle genannten Eigenschaften werden in einer `LanguageText`-Datei definiert, die bereits am Anfang des Kapitels vorgestellt wurde. Sie bietet die Möglichkeit, mehrsprachigen Text zu verwenden. Für jeden Editor existieren zwei Dateien, die in dem Verzeichnis `genged/resources/alphabeteditor/...` unter den Namen `guiGlobal.res` und `guiLocal.res` zu finden sind. Für jede Aktion, die definiert werden soll, muß ein eindeutiger Name festgelegt werden, der als Teil eines Schlüssels dient. Wir haben die verschiedenen Aktionen nach Kategorien geordnet, um das System übersichtlicher zu gestalten. Die verschiedenen Kategorie-Namen und der eigentliche Aktionsname werden durch Punkte getrennt, wie z.B. `SymbolEditor.SystemAction.Cut`. Diese Aktion gehört zum Symbol-Editor in die Kategorie System-Aktion und dient dem Ausschneiden von Objekten. Der eindeutige Name einer Aktion wird – gefolgt von einem bestimmten Suffix – für alle Eigenschaften als Schlüssel verwendet. Die Tabelle 6.2 gibt einen Überblick über alle Suffixe und die dazugehörigen Eigenschaften.

Es müssen nicht unbedingt alle Eigenschaften definiert werden. Im Grunde genommen würde die Definition eines Menüeintrags ausreichen. Wird kein normales Icon definiert, erscheint in einem Symbol-Button ebenfalls Text. Die Definition der besonderen Icons kann weggelassen werden, wenn das normale Icon verwendet werden soll.

Um eine Checkbox zu definieren, muß der Name der Aktion mit dem Wort `CHECK` enden. Ein Radio-Button gehört zu einer Gruppe mit mehreren Buttons. Zur Definition wird an den Namen `RADIO(xxx)` angehängt, wobei `xxx` für den Namen einer Gruppe steht, der von allen Radio-Buttons dieser Gruppe verwendet wird. Im folgenden ist ein Beispiel abgedruckt, das einen Radio-Button definiert:

```
# ====================================================================
# ==== SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw)
# ====================================================================
"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).MENU",
"Offener &Kreisausschnitt",
"Open circle part";

"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).TOOLTIP",
"Offener Kreisausschnitt",
"Open circle part";

"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).STATUS",
"Aktiviert das Zeichenwerkzeug für offene Kreisausschnitte.",
"Activate the tool for open circle parts.";

"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).ICON",
"circlepart_open";

"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).ENTRY",
"SymbolEditor.SetDrawAction";
```

[2] eigentlich müßten wir in diesem Zusammenhang von Radio-Actions sprechen

Suffix	Eigenschaft
.MENU	Menüeintrag (steht vor einem Buchstaben das Zeichen &, kann der Menüeintrag auch über die Tastatur ausgewählt werden, indem auf den jeweiligen Buchstaben gedrückt wird)
.TOOLTIP	Tooltip-Text, der erscheint, wenn die Maus über einen Symbol-Button dieser Aktion bewegt wird
.STATUS	Statuszeilen-Text, der erscheint, wenn die Maus über einen Menüeintrag oder Symbol-Button dieser Aktion bewegt wird
.ICON	Dateiname für ein normales Icon, das in einem Symbol-Button und ggf. neben einem Menüeintrag verwendet wird (diese Datei muß als GIF-Format im Verzeichnis `genged/resources/icons` vorliegen, die Endung GIF wird in der Beschreibung jedoch weggelassen)
.DISABLEDICON	wie `.ICON`, allerdings für eine Aktion, die zur Zeit nicht ausgewählt werden kann
.PRESSEDICON	wie `.ICON`, allerdings für einen Symbol-Button, der mit der Maus heruntergedrückt wird
.ROLLICON	wie `.ICON`, allerdings für einen Symbol-Button, über den der Benutzer die Maus bewegt
.SELECTICON	wie `.ICON`, allerdings für einen Symbol-Button, der eine Checkbox oder ein Radio-Button darstellt und selektiert wurde
.ENTRY	Name der Eintrittsmethode bei der Ausführung der Aktion (Editorname, gefolgt von einem Punkt und dem Methodennamen)
.CLASS	Zusatzinformation, die den Klassennamen eines Objekts enthält
.PARENT	Zusatzinformation, die den Namen einer übergeordneten Gruppe enthält

Tabelle 6.2: Suffixes für Aktions-Eigenschaften innerhalb einer Definitionsdatei

```
"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).CLASS",
"genged.language.OpenCirclePartObject";

"SymbolEditor.DrawAction.CirclePartOpenRADIO(SymbolEditor.Draw).PARENT",
"SymbolEditor.GroupAction.ShowOvalGroupRADIO(SymbolEditor.Mode)";
```

Die Definitionsdateien müssen durch die Instanz der Klasse `LanguageText` eingelesen werden. Sie verwaltet die mehrsprachigen Datensätze. Jeder Text kann nach dem bereits bekannten Mechanismus über einen Schlüssel abgerufen werden. Die Klasse `EditorAction` erwartet die Existenz der gewünschten Aktionseigenschaften.

6.4.2 Arbeiten mit Benutzeraktionen

Um eine neue Aktion zu erstellen, muß an den Konstruktor unter anderem der Name der Aktion übergeben werden. Dabei handelt es sich um den Schlüssel, der für die verschiedenen Eigenschaftsdefinitionen verwendet wurde, allerdings ohne den jeweiligen Suffix. Alle bekannten Eigenschaften werden ausgewertet und für die aktuell eingestellte Sprache vorbereitet. Unter anderem werden im Konstruktor die Weichen zur Verwendung von Radio-Buttons und Checkboxen gestellt. Um verschiedene Gruppen von Radio-Buttons zu verwalten, wurde das statische Attribut `actionGroups` eingeführt. Dabei handelt es sich um eine Hashtable, die den Namen einer Radio-Button-Gruppe auf eine Liste mit allen dazugehörigen `EditorAction`-Instanzen abbildet. Über verschiedenste Methoden können die Eigenschaften abgefragt oder unabhängig von der Datenbank eingestellt werden.

Menüeintrag: Mit der Methode `createMenuItem()` kann aufgrund der Eigenschaften ein Menüeintrag erstellt werden. Dieser Eintrag wird in der Hashtable `menuItems` gespeichert. Er dient als Schlüssel und wird abgebildet auf das Menü, in dem er später enthalten sein wird. Die Methode fügt den Menüeintrag jedoch nicht in das Menü ein. Die Methode `removeMenuItem()` entfernt einen Menüeintrag aus der Aktion und aktualisiert das Menü, in dem er enthalten war.

Symbol-Button: Die Methode `createToolButton()` erstellt entweder mit einem Icon oder einem Text einen Symbol-Button, der in einer Symbolleiste dargestellt werden könnte. Gespeichert werden alle erstellten Buttons in der Hashtable `toolButtons`. Um einen Button aus der Aktion zu entfernen, kann die Methode `removeToolButton()` aufgerufen werden. Sie aktualisiert gleichzeitig die Komponente, in der er enthalten war.

Menüs: Nicht nur die Menüeinträge, sondern auch die Menüs und Untermenüs selbst müssen in mehreren Sprachen vorliegen. Zur Erstellung und Verwaltung bietet die Klasse `EditorAction` statische Methoden an. In dem statischen Attribut `menus` werden alle generierten Menüs gespeichert. Um ein Menü zu erstellen, wird die Methode `EditorAction.createMenu()` aufgerufen. Übergeben werden muß unter anderem ein Name, der wie bei den einzelnen Aktionen den Schlüssel zur Text-Datenbank darstellt. Es können einige der bekannten Eigenschaften definiert werden: Menütitel (`.MENU`), Menü-Icon (`.ICON`), ein nicht selektierbares Icon (`.DISABLEDICON`) und ein Statuszeilen-Text (`.STATUS`). Durch Aufruf der ebenfalls statischen Methode `removeMenu()` kann ein Menü wieder entfernt werden.

Enable/Disable: Der größte Vorteil bei der Verwendung von Aktionen ist, daß der aktuelle Zustand `aller` Menüeinträge und Symbol-Buttons gleichzeitig geändert werden kann. Das betrifft beispielsweise das An- bzw. Abschalten einer Benutzeraktion. Dazu dient die Methode `setEnabled()`. Mit `isEnabled()` kann der aktuelle Zustand abgefragt werden.

Selektieren/Deselektieren: Handelt es sich bei einer Aktion um eine Checkbox oder um einen Radio-Button, so kann sie durch Aufrufen der Methode `setSelected()` selektiert oder deselektiert werden. Den aktuellen Zustand liefert die Methode `isSelected()`. Während bei einer Checkbox ausschließlich der eigene Zustand wechselt, werden bei einem Radio-Button alle anderen Gruppenmitglieder ebenfalls beeinflußt.

Ausführen: Sobald der Benutzer mit der Maus oder per Tastatur auf einen der registrieren Menüeinträge oder Symbol-Buttons klickt, wird die Aktion ausgeführt. Das gilt auch für Checkboxen und Radio-Buttons. Außerdem wird jedoch deren Zustand betreffs ihrer Selektierung geändert. Eine Aktion kann auch über die Methode `invoke()` aufgerufen werden. Bei jedem Aufruf wird die ActionListener-Methode `actionPerformed()` ausgeführt. Sie generiert anhand der Eigenschaft für die Eintrittsmethode (`.ENTRY`) einen dynamischen Methodenaufruf. Dazu wird unter Verwendung des bereits vorgestellten Interfaces `EditorManager` der Editorname ausgewertet und dessen Klasse angefordert. Dafür ist die Methode `getEntryMethod()` zuständig. In der jeweiligen Editor-Instanz wird nun die Eintrittsmethode aufgerufen. Sie `muß` immer genau zwei Parameter besitzen. Ihr werden Instanzen von `EditorAction` (die aufgerufene Aktion) und `ActionEvent` (Ereignis) übergeben. In dem oben beschriebenen Beispiel wurde als Eintrittsmethode "`SymbolEditor.SetDrawAction`" definiert. Innerhalb des Symbol-Editors muß dazu die folgende Methode implementiert sein:

```
public void SetDrawAction(EditorAction action, ActionEvent event) {
  // hier folgt der Code, der bei Aufruf der Aktion ausgeführt wird
}
```

Verwaltet werden die verschiedenen Benutzeraktionen durch die Klasse `ActionManager`, die eine Erweiterung der Klasse `java.util.Hashtable` darstellt. Die Abbildung 6.5 zeigt die wichtigsten Attribute und Methoden sowie die Klassenbeziehungen der Klasse `EditorAction`.

6.4.3 Definitionsdateien

Die verschiedenen Teileditoren innerhalb des Alphabet-Editors definieren in jeweils drei Dateien, welche Benutzeraktionen existieren und wie die Menüs und Symbolleisten aus diesen Aktionen zusammengesetzt werden. Die drei Dateien befinden sich in dem Ressourcen-Verzeichnis des jeweiligen Editors: `genged/resources/alphabeteditor/...`. Das Format dieser Dateien entspricht dem Format der Property-Dateien [Fla97], die von Java unterstützt werden. Sie werden zur Laufzeit unter Nutzung der Klasse `java.util.Properties` eingelesen und ausgewertet.

Datei `actions.ini`: In dieser Datei werden alle Namen der verwendeten Aktionen festgelegt. Über diesen Namen werden später die Eigenschaften aus der mehrsprachigen Text-Datenbank ausgewertet. Wir unterstützen die Bildung verschiedener Kategorien, die eine Hierarchie bilden können. Die oberste Kategorie besitzt den Namen des Editors, zu dem die Benutzeraktionen gehören, z.B. `SymbolEditor`. Ein Aktionsname unterteilt sich in verschiedene Kategorienamen, die jeweils durch einen Punkt voneinander getrennt sind, und in den eigentlichen Namen der Aktion. Der schematische Aufbau der Datei ist folgendermaßen:

```
Editor = Editor.Kateg1 Editor.Kateg2 Editor.Aktion1
Editor.Kateg1 = Editor.Kateg1.Aktion1 Editor.Kateg1.Aktion2
Editor.Kateg2 = Editor.Kateg2.Aktion1 Editor.Kateg2.Kateg1
Editor.Kateg2.Kateg1 = Editor.Kateg2.Kateg1.Aktion1
```

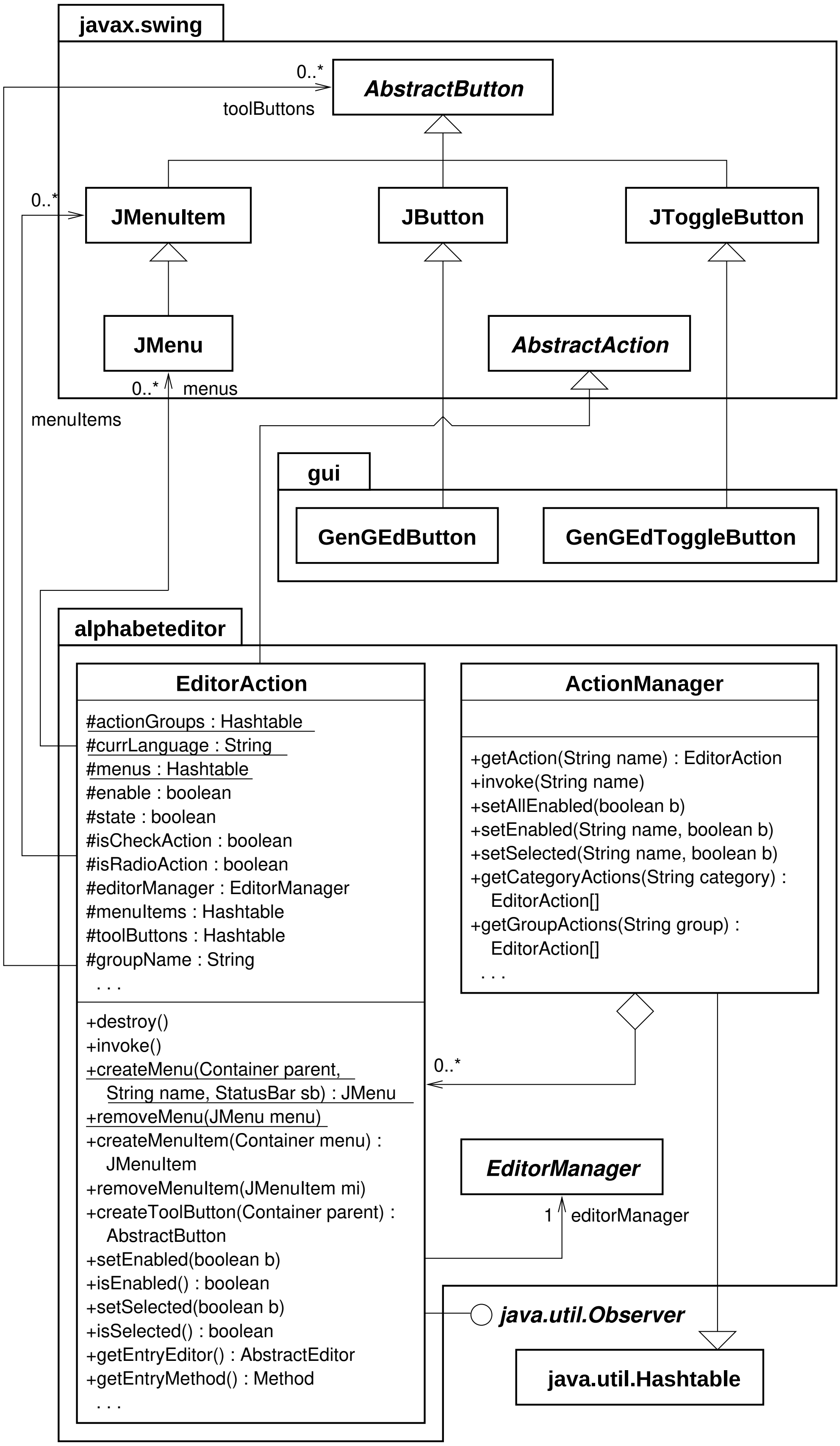


Abbildung 6.5: Klassendiagramm: `EditorAction`

Kategorie- und Aktionsnamen müssen nur in ihrem Kontext eindeutig sein. Jeder Name, der ausschließlich rechts von einem Gleichheitszeichen steht und nicht Ausgangspunkt für eine neue Definition ist, wird als Aktionsname betrachtet. Dieser Name wird im Zusammenhang mit den verschiedenen Suffixes als Schlüssel in den Dateien `guiGlobal.res` und `guiLocal.res` verwendet.

Datei `menubar.ini`: Um eine Menüleiste für einen Editor zusammenzustellen, wird die Datei `menubar.ini` ausgewertet. In ihr tauchen alle Aktionsnamen wieder auf. Außerdem werden hier die Namen (Schlüssel) für verschiedene Menüs verwendet. Soll zwischen zwei Menüeinträgen ein Separator erscheinen, muß dafür das Zeichen „–“ benutzt werden. Der schematische Aufbau der Datei ist folgendermaßen:

```
Editor = Editor.Menü1 Editor.Menü2
Editor.Menü1 = Editor.Kateg1.Aktion2 - Editor.Kateg1.Aktion1
Editor.Menü2 = Editor.Kateg2.Kateg1.Aktion1 Editor.Menü2.Submenü1
Editor.Menü2.Submenü1 = Editor.Kateg2.Aktion1
```

Auch hier müssen die einzelnen Namen in ihrem Kontext eindeutig sein. Jeder Name, der rechts vom Gleichheitszeichen steht, muß entweder ein Aktions- oder ein Menüname sein, der weiter spezifiziert wird. Alle Namen werden im Zusammenhang mit den verschiedenen Suffixes als Schlüssel in den Dateien `guiGlobal.res` und `guiLocal.res` verwendet.

Datei `toolbar.ini`: Zum Definieren einer Symbolleiste wird genauso vorgegangen wie bei einer Menüleiste. Allerdings existieren für die globale Symbolleiste keine „Untermenüs“. Der Symbol-Editor definiert allerdings zusätzlich zur normalen Symbolleiste noch weitere Werkzeug-Panels, die in verschiedene Kategorien unterteilt sind. Auch zwischen zwei Symbol-Buttons kann durch das Zeichen „–“ ein Separator eingefügt werden.

6.5 Undo-Funktionalität

Der Undo-Mechanismus ermöglicht es dem Benutzer, ausgeführte Aktionen rückgängig zu machen. Unter einer Aktion verstehen wir in diesem Zusammenhang jede Änderung des Datenmodells, des Alphabets. Die Funktionalität des Undo-Mechanismus beruht auf dem Command-Processor-Pattern, das bereits vorgestellt wurde. Allerdings wurde es ein wenig modifiziert und an unsere Anforderungen angepaßt. Wir haben bestehende Java-Klassen erweitert, die sich nicht vollständig nach dem Pattern richten. Insgesamt existieren drei Klassen in dem Package `genged.utilities`, die die Grundlagen bilden. Sie werden in den folgenden Abschnitten kurz erläutert.

6.5.1 Befehlsobjekte und Befehlsgruppen

Die Klasse `Command` bildet die Basisklasse für alle Befehlsobjekte. Sie kapselt jeden Befehl und bietet die Möglichkeit, ihn rückgängig zu machen. Dazu werden an den

Konstruktor alle Informationen übergeben, die benötigt werden, um eine Aktion auszuführen. In der Regel gehört dazu eine Referenz auf das Alphabet, meistens weitere Referenzen auf dessen Teilobjekte. Generell wird dem Konstruktor eine Referenz des Undo-Managers übergeben, der alle ausgeführten Befehlsobjekte verwaltet. Der Konstruktor sorgt dafür, daß alle Daten der Datenbasis, die verändert werden sollen, zuvor gesichert werden. Danach wird aus dem Kontruktor heraus die Methode `redo()` aufgerufen, die die Änderung mit Hilfe der Funktionalitäten verschiedener Dienstleister ausführt. Als letztes meldet sich das Befehlsobjekt durch Aufruf der Methode `finishCreation()` beim Undo-Manager an und wird dort gespeichert. Wurde kein Undo-Manager übergeben, wird die Aktion zwar ausgeführt, kann jedoch nicht rückgängig gemacht werden. Zum Wiederrufen der Aktivität wird durch den Undo-Manager die Methode `undo()` aufgerufen.

Die Erfahrung hat gezeigt, daß wir eine Möglichkeit benötigen, Befehlsobjekte zu schachteln. Daher wurde die Klasse um die Methode `addEdit()` erweitert, mit der beliebig viele Befehlsobjekte hinzugefügt werden können. Da diese ebenfalls über eine solche Methode verfügen, kann theoretisch eine beliebige Schachtelungstiefe erreicht werden. Soll ein Befehlsobjekt in ein anderes eingefügt werden, muß es vom Konstruktor der Klasse `Command` erstellt werden, noch bevor die eigene `redo()`-Methode benutzt wird. Statt den Undo-Manager als Parameter zu übergeben, wird direkt das `Command`-Objekt benutzt (`this`). Die Befehle innerhalb eines anderen Befehls werden immer zuerst ausgeführt bzw. zuletzt wieder rückgängig gemacht.

Es existieren noch einige weitere Methoden, die bedeutend sind. Die Methode `getPresentationName()` liefert eine Kurzbeschreibung des Befehls, die dem Benutzer über die grafische Oberfläche angezeigt wird. Der Rückgabewert der Methode `isSignificant()` legt fest, ob der Befehl so bedeutsam ist, daß er in der „History“-Liste der Befehle sichtbar sein soll. Ist er eher unbedeutsam, wird er zwar im Undo-Manager gespeichert, ist für den Benutzer jedoch nicht sichtbar und kann nur im Zusammenhang mit anderen Aktionen rückgängig gemacht werden.

Um mehrere Befehlsobjekte in einer Gruppe zusammenzuschließen, existiert die Klasse `CommandGroup`, die die Swing-Klasse `CompoundEdit` erweitert. Diese Klasse wird als `Command` benutzt. Es werden keine neue Subklassen gebildet. Damit die Kurzbeschreibung des Befehls dynamisch ist, wird an den Konstruktor ein Schlüssel für einen mehrsprachigen Text (aus der Klasse `LanguageText`) und optional ein Array mit Parametern übergeben.

6.5.2 Undo-Manager

Die Klasse `GenGEdUndoManager` erweitert den von Java zur Verfügung gestellten `UndoManager` der Swing-Klassen. An dieser Stelle sollen nur zwei der Erweiterungen vorgestellt werden. Für weitere Informationen verweisen wir auf die Online-Dokumentation von Java.

Dem Undo-Manager können mit der Methode `addEdit()` Befehlsobjekte hinzugefügt werden. Er verfügt dabei über die Fähigkeit, unabhängig von dem Befehlsobjekt eine Verschachtlung von Befehlen aufzubauen. Dazu nutzt er die Klasse `CommandGroup` im Zusammenhang mit einem Stack. Mit der Methode `beginGroup()` kann der Nutzer eines Undo-Managers eine neue Gruppe initiieren. Alle Befehle, die dem Undo-Manager hinzugefügt werden, leitet er nun an die Gruppe weiter, bis die Gruppe mit der Methode `endGroup()` geschlossen und gleichzeitig dem Undo-Manager als *ein* Befehl hinzugefügt wird. Eine Verschachtelung der Gruppen kann erreicht werden, indem mehrmals hintereinander die Methode `beginGroup()` aufgerufen wird.

Neben der Befehlsverwaltung erfüllt die Klasse `GenGEdUndoManager` noch eine Nebenfunktion. Sie protokolliert alle vom Benutzer ausgeführten Aktionen. Dazu wird mit der Methode `createProtocolFile()` eine Protokolldatei auf dem Datenträger angelegt. Es kann ein Dateiname sowie ein Header übergeben werden. Sobald ein Befehlsobjekt dem Undo-Manager hinzugefügt wird, schreibt er einen entsprechenden Eintrag an das Ende der Prokolldatei. Mit der Methode `closeProtocolFile()` kann die Datei geschlossen und die Protokollierung beendet werden. Die Methode `protocol()` bietet auch von „außen“ die Möglichkeit, der Datei einen eigenen Eintrag hinzuzufügen.

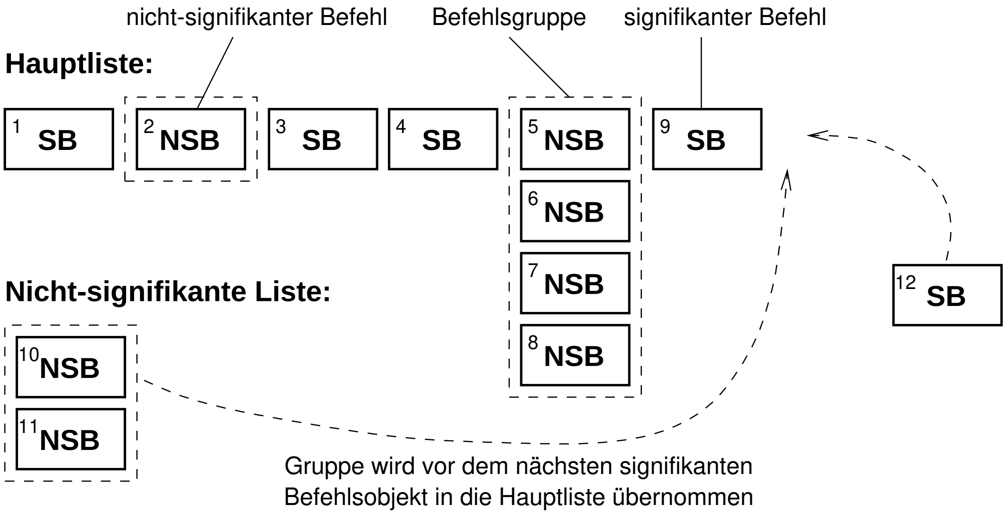


Abbildung 6.6: Verwaltung der Befehlsobjekte im Undo-Manager

Bei der Benutzung des Undo-Managers trat ein Problem auf, dessen Ursachen in den Wurzeln des Java-eigenen Undo-Mechanismus liegen: Ein Benutzer führt verschiedene Aktionen aus, die im Undo-Manager gespeichert werden. Der Benutzer läßt einige Aktionen rückgängig machen und führt daraufhin eine nicht-signifikante Aktion aus. Obwohl die Aktion nicht im Undo-System angezeigt wird, wurde sie dennoch hinzugefügt. Das hat zur Konsequenz, daß die vorher rückgängig gemachten Befehle nicht wiederhergestellt werden können. Der Rest der Undo-Liste wurde „abgeschnitten“. Dieser Vorgang entspricht vollkommen der Spezifikation der entsprechenden Swing-Klassen. Dennoch wurde er als sehr störend empfunden. Daher haben wir den Undo-Manager um eine Befehlsgruppe (Attribut `insignCommands`) erweitert, die nicht-signifikante Befehlsobjekte zwischenspeichert, ohne sie der Hauptliste hinzuzufügen. Wird ein signifikantes Befehlsobjekt hinzugefügt, wird die nicht-signifikante Gruppe zuvor in der Hauptliste gespeichert. Die Methoden `undo()` und `redo()` wurden entsprechend angepaßt, damit auch die letzten nicht-signifikanten Befehle rückgängig gemacht werden können. Die Abbildung 6.6 gibt einen Einblick in diese Strukturen.

In der Abbildung 6.7 werden Zusammenhänge des Undo-Mechanismus als Klassendiagramm dargestellt.

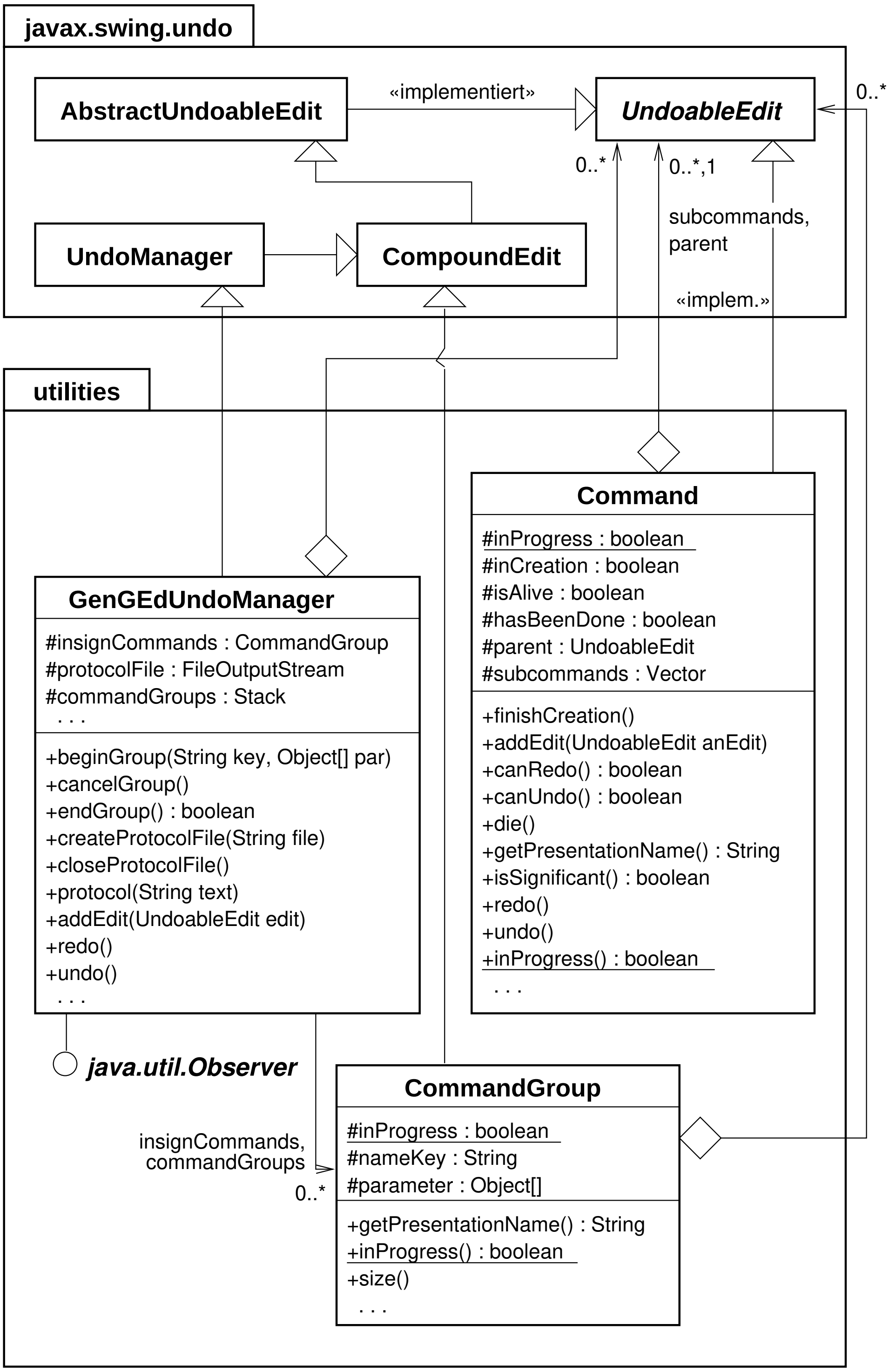


Abbildung 6.7: Klassendiagramm: `GenGEdUndoManager`

Kapitel 7

Constraint-Datenbank

Die Operationen, die im Alphabet definiert werden, enthalten Constraints. Sie dienen der Layout-Anpassung von verschiedenen primitiven Grafiken und werden durch (Un)gleichungssysteme formuliert. Diese Formelsysteme werden durch den externen Constraint-Solver PARCON gelöst, der bereits in Kapitel 3 vorgestellt wurde. In diesem Kapitel wird nun die Anbindung des Alphabet-Editors an diesen Constraint-Solver beschrieben. Außerdem wird darauf eingegangen, wie das Constraint-System im Alphabet-Editor aufgebaut ist und wie es verwaltet wird.

7.1 High-Level-Constraint-Sprache

Dem Constraint-Solver PARCON können nur einzelne (Un)gleichungen übergeben werden, sogenannte Low-Level-Constraints. Für den Alphabet-Editor und seine nachfolgenden Komponenten wurde daher eine High-Level-Constraint-Sprache entwickelt, mit der Objekttypen und Constraints definiert werden können, die parametrisiert und typabhängig sind und mehrere PARCON-Formeln zusammenfassen. Die Entwicklung dieser Sprache stellt einen wichtigen Bestandteil dieser Arbeit dar.

Der Benutzer kann mit Hilfe der High-Level-Constraint-Sprache eigene Constraints definieren, die zu einer Constraint-Datenbank zusammengestellt werden. Da es sich bei der Benutzung von Constraints nicht um eine interne Angelegenheit handelt, wird sie im fünften Kapitel des Benutzerhandbuchs[1] sehr ausführlich beschrieben. Deshalb soll in den folgenden Abschnitten nur eine sehr kurze Zusammenfassung zur Anwendung der High-Level-Constraint-Sprache gegeben werden. Wir werden in diesem Kapitel mehr auf interne Mechanismen eingehen.

7.1.1 Definitionsdateien

Die High-Level-Constraints – im folgenden nur noch Constraints genannt – werden in zwei Dateien definiert:

[1]Das Benutzerhandbuch ist im Anhang der Diplomarbeit zu finden. Es wird jedoch auch losgelöst von dieser Arbeit benutzt.

- Die Datei `genged/resources/constraints/system/system.hlcl` enthält systeminterne Typen und Constraints. Sie darf nicht verändert werden. Unter anderem werden in dieser Datei die Initalisierungs-Constraints für alle Primitiven definiert.
- In dem Verzeichnis `genged/resources/constraints` können sich benutzerdefinierte Constraint-Definitionen befinden. Normalerweise wird die Datei `user.hlcl` verwendet, in der standardmäßig verschiedene Constraints definiert sind, die immer wieder benötigt werden.

7.1.2 Sprachelemente

Die High-Level-Constraint-Sprache setzt sich aus den folgenden verschiedenen Sprachelementen zusammen:

- Schlüsselwörter (`abs acos acos°  and asin asin° atan atan° cos cos° constraint deg description dialog e exp extends log max of oo operationMode or pi rad round set sin sin° sqrt tan tan° trim trim° type`)
- Bezeichner (Typbezeichner, Variablenbezeichner, Constraintbezeichner)
- Zeichenketten (in Hochkommata eingeschlossen)
- Zahlen (Integer, Double, jedoch keine Exponentendarstellung)
- Sonderzeichen (`{ } ( ) [ ] + - * > = < : , ; "` #)
- Kommentare (werden durch das Zeichen # eingeleitet)

7.1.3 Deklaration von Typen

In der High-Level-Constraint-Language werden unter anderem Typen deklariert, die später als Constraint-Parameter genutzt werden. Typen beziehen sich – abgesehen von einigen Ausnahmen – auf eine oder mehrere grafische Primitiven. Bereits im Abschnitt 5.7 wurde die Typisierung der Grafiken angesprochen. Es können nur zusammengesetze Typen deklariert werden. Diese bauen auf Grundtypen bzw. auf bereits bestehenden anderen zusammengesetzten Typen auf.

Grundtypen: Es existieren die Grundtypen `Value`, `ConstValue` und `Point`. Der Typ `Value` stellt einen numerischen Wert dar und kann entweder innerhalb von zusammengesetzten Typen oder für lokale Variablen innerhalb eines Constraints verwendet werden. `ConstValue` stellt einen konstanten numerischen Wert dar, der ausschließlich für Parameter von Constraints eingesetzt werden kann. Auf diesem Typ können mit der High-Level-Constraint-Sprache verschiedene mathematische Berechnungen durchgeführt werden. Bei dem Typ `Point` handelt es sich bereits um einen zusammengesetzten Typ, der x- und y-Koordinaten vom Typ `Value` enthält.

Zusammengesetzte Typen: Ein zusammengesetzter Typ hat einen eindeutigen Namen und kann mehrere Variablen enthalten, die als Eigenschaften oder Attribute bezeichnet werden. Der Typ einer Eigenschaft ist entweder einer der vorgestellten Grundtypen oder ein bereits definierter zusammengesetzter Typ. Zusammengesetzte Typen können andere Typen erweitern und dadurch deren Eigenschaften nutzen. Darauf beruht der objektorientierte Ansatz der Typendeklaration. Eigenschaften eines Typs können dereferenziert werden, indem nach dem Variablen- bzw. Parameterbezeichner ein Punkt und der Name der Eigenschaft angegeben wird.

Beispiel: Die folgenden Zeilen zeigen verschiedene Typendeklarationen:

```
type Base {
  Point lt;     # point in upper left corner
  Value w,h;    # width and height
}

type Ellipse extends Base {
  Point m;      # centre
  Value rx;     # radius x (horizontal)
  Value ry;     # radius y (vertical)
}

type Circle extends Ellipse {
  Value r;      # radius
}

type Line {
  Point s,t;    # start and target point
}
```

7.1.4 Definition von Constraints

Ein Constraint wird im Alphabet-Editor über seinen Namen und die Anzahl und Typen seiner Parameter identifiziert. Diese Bestandteile müssen auch in seiner Definition vorkommen. Namen können dabei mehrfach auftreten. Allerdings müssen sich die Parameter unterscheiden. Wir sprechen in diesem Fall von einem Überladen[2] von Constraints.

Ein Parameter ist eine spezielle Variable, die beim Aufruf des Constraints mit einem bestimmten Wert initialisiert wird. Bei diesem Wert kann es sich je nach Typ des Parameters um eine Zahl oder eine grafische Primitive handeln. Für jeden Constraint werden Formeln definiert, in denen der Parameter über seinen Namen angesprochen wird. Es können neben einfachen Parametern auch Parametermengen verwendet werden, die zur Laufzeit mehrere Elemente eines Typs aufnehmen und verarbeiten können. Für Parameter vom Typ `ConstValue` können Intervalle festgelegt werden, in denen sich der übergebene Zahlenwert bewegen muß.

Da bei der Anwendung der Constraints zur Laufzeit des Alphabet-Editors Parameter angegeben werden müssen, kann für jeden Constraint und jeden seiner Parameter

[2] Overloading

eine Beschreibung definiert werden, die sogar Mehrsprachigkeit unterstützt.

Innerhalb eines Constraints können verschiedene Formeln zur Definition eines (Un)gleichungssystems genutzt werden. Wir unterscheiden insgesamt vier verschiedene Formelarten. *Lineare (Un)gleichungen* stellen einen linearen Zusammenhang zwischen beliebigen Variablen her. Eine *Produktgleichung* setzt ein Produkt zwischen höchstens zwei Variablen zu einem konstanten Wert oder einer anderen Variablen in Beziehung. Mit Hilfe des *Euklidischen Abstands* kann auf leichte Weise ein bestimmter Abstand zwischen zwei Punkten definiert werden. Auch die letzte Formelart – *Punktgleichheit* – arbeitet auf Punkten. Sie stellt sicher, daß sich zwei Punkte an ein und derselben Position befinden. Alle Formeln sind dabei durch den Operator UND verknüft – sie alle müssen für die ermittelte Lösung gelten. PARCON bietet zusätzlich an, einzelne Formeln miteinander durch ein ODER zu verknüpfen. Dadurch genügt es, daß eine der betreffenden Formeln erfüllt ist. Deshalb ist es möglich, einen speziellen ODER-Modus ein- und auszuschalten. Die Reihenfolge der Auswertung einer Formel kann durch Prioritätenvergabe beeinflußt werden. Dazu stehen die Zahlen 0 bis 9 zur Verfügung. Die Zahl 0 steht für die höchste, die Zahl 9 dagegen für die niedrigste Priorität. Die Auswertungsreihenfolge ist immer dann wichtig, wenn mehrere Beziehungen zwischen den Variablen und Parametern definiert werden, die aufeinander aufbauen.

Jeder Constraint stellt in sich ein (Un)gleichungssystem dar, das mit anderen (Un)gleichungssystemen in Beziehung steht. Die Beziehung wird immer dann hergestellt, wenn bei der Anwendung mehrerer Constraints dieselben Primitiven als Parameter verwendet werden. Über lokale Variablen hat man die Möglichkeit, Hilfswerte zu definieren, die nur innerhalb *eines* (Un)gleichungssystems gelten. Innerhalb einer Formel müssen Parameter und Variablen generell auf die Grundtypen dereferenziert werden. PARCON versteht nur einfache numerische Variablen `Value`. In einigen Formeln kann er auch mit Punkten (`Point`) umgehen. Konstanten dagegen, z.B. Parameter vom Typ `ConstValue`, werden vor der Übergabe an den Constraint-Solver in Zahlen umgewandelt. Konstanten können wie Zahlen addiert, subtrahiert oder multipliziert werden. Die Division ist zur Zeit nicht definiert – sie kann jedoch über Multiplikation ausgedrückt werden. Stattdessen ist eine eingeschränkte Nutzung einstelliger mathematischer Funktionen möglich.

Aus einer Constraint-Definition heraus können andere Constraints verwendet werden. Sie werden über ihren Constraint-Namen, gefolgt von den erforderlichen Parametern aufgerufen. Existiert kein Constraint, dessen Parametertypen direkt mit den übergebenen Parametern übereinstimmen, wird nach einem gleichnamigen Constraint gesucht, dessen Typen aufgrund der objektorientierten Vererbungshierachie am besten dazu passen. Alle Formeln des aufgerufenen Constraints werden der aktuellen Constraint-Definition mit einer UND-Verknüpfung hinzugefügt. Sollte innerhalb des aufgerufenen Constraints an einer Stelle der ODER-Modus verwendet werden, wird er lokal berücksichtigt. Um einen ganz speziellen Constraint aufzurufen, können Variablen oder Parameter gecastet werden. Darunter verstehen wir eine Typumwandlung, die ausschließlich innerhalb eines Constraint-Aufrufs nötig und deshalb auch nur dort möglich ist.

Wir benutzen innerhalb einer Constraint-Definition eine etwas übersichtlichere Schreibweise für die Formeln als PARCON. Daher muß die von uns verwendete Syntax in die Schreibweise des Constraint-Solvers umgewandelt werden. Besonders die linearen Formeln, die in der High-Level-Constraint-Sprache fast völlig frei gestaltet werden können, bedürfen einer Umwandlung in die strengen Formen von PARCON. Im folgenden werden kurz die Schritte skizziert, die zur Umwandlung erforderlich sind:

- Die Relationszeichen $<$ oder $>$ werden in die Relationen $<=$ bzw. $>=$ umgewandelt, indem die Zahl 1 zur Formel addiert oder von ihr substrahiert wird.
- Es werden alle mathematischen Funktionen berechnet.
- Alle Konstanten, die nicht als Faktor für eine Variable dienen, werden zusammengefaßt und auf die rechte Seite der Formel gebracht.
- Alle gleichen Variablen werden zusammengefaßt und zusammen mit ihren Faktoren auf die linke Seite der Formel gebracht, indem sie hinter die bereits dort vorhandenen Variablen geschrieben werden (weiter nach rechts).
- Alle anderen Variablen der rechten Seite werden mit ihren Faktoren ebenfalls auf die linke Seite gebracht, indem auch diese hinter die bereits vorhandenen Variablen geschrieben werden (noch weiter nach rechts).

Beispiel: Die folgenden Zeilen zeigen verschiedene Constraintdefinitionen:

```
constraint Outside(Line a, Base b) {
  Dialog(a, "English", "Line");   # Parameter-
  Dialog(a, "Deutsch", "Linie");  # und
  Dialog(b, "English", "Object"); # Constraint-
  Dialog(b, "Deutsch", "Objekt"); # beschreibungen
  Description("English", "The line is outside of the object.");
  Description("Deutsch", "Die Linie liegt außerhalb von Objekt.");

  Value w, h;                # lokale Variablen

  [0] w = a.t.x - a.s.x;     # w und h sollen zuerst
  [0] h = a.t.y - a.s.y;     # berechnet werden (=> Priorität 0)
  OperationMode(OR);         # Einschalten des ODER-Modus
  [1] a.s.x + w < b.lt.x;    #      a liegt links von b
  [1] a.s.y + h < b.lt.y;    # oder a liegt über b
  [1] a.s.x > b.lt.x + b.w;  # oder a liegt rechts von b
  [1] a.s.y > b.lt.y + b.h;  # oder a liegt unter b
}

constraint MinLength(Line l, ConstValue minLength) {
  Dialog(l, "English", "Line");
  Dialog(l, "Deutsch", "Linie");
  Dialog(minLength, "English", "Minimal length");
  Dialog(minLength, "Deutsch", "Minimale Länge");
  Description("English", "The line has a minimal length.");
  Description("Deutsch", "Die Linie hat eine minimale Länge.");

  Value length;              # lokale Variable
```

```
  length >= minLength;        # Eingrenzung des Wertebereichs für length
  l.s R= length * l.t;        # Euklidischer Abstand
}

constraint SameLocation(Set Of Base a, Base b) {
  a.lt P= b.lt;               # Verwendung einer Parametermenge (Set Of)
}
```

7.2 Auswertung der Definitionsdateien

Beim Starten des Alphabet-Editors werden die Definitionsdateien der Constraints eingelesen, ausgewertet und in eigenen Datenstrukturen gespeichert. Dazu dient die Klasse `DatabaseHLCL`. Sie und alle anderen Komponenten, die zur Arbeit mit Constraint verwendet werden, befinden sich in dem Package `genged.constsolver`. Zum Erstellen der Datenbank wird der Konstruktor mit dem Dateinamen der benutzerdefinierten Constraint-Datei aufgerufen. Er sorgt dafür, daß zuerst die systemeigene Definitionsdatei eingelesen wird. Danach wertet er unter Nutzung der Methode `addDatabaseHLCL()` die angegebene Datei aus. Dieser Prozeß erfolgt in sechs Schritten:

1. Die Datei wird durch Aufruf der Methode `readFile()` zeilenweise eingelesen und in einem Vector zwischengespeichert.

2. Dieser Vector wird an die Methode `createTokens()` übergeben, die alle Zeilen der Datei in Tokens zerlegt. Jedes Token entspricht einem der bereits vorgestellten Sprachelemente, z.B. einer Zahl, einem Schlüsselwort oder einem Bezeichner. Hierbei kommt die Klasse `Token` zur Anwendung, die jeweils ein Token kapselt. Alle Tokens werden in einem weiteren Vector in der richtigen Reihenfolge gespeichert.

3. Nun werden Typ-Deklarationen von Constraint-Definitionen getrennt. Die Methode `getTypeDefinitions()` kopiert alle Tokens, die jeweils zu einer Typ-Deklaration gehören, in einen neuen Vector. Am Ende entsteht ein Vector, der wiederum mehrere Vectoren enthält, die jeweils eine vollständige Typ-Deklaration enthalten.

4. Derselbe Prozeß läuft mit Hilfe der Methode `getConstraintDefinitions()` für die Constraint-Definitionen ab.

5. Durch Aufruf der Methode `makeTypes()` wird nun mit der Parsierung aller Tokens begonnen, die zu den Typ-Deklarationen gehören. Dazu wird die Klasse `TypeParser` benutzt. Es werden für jeden Typ Datenstrukturen angelegt, die auf der Klasse `Type` beruhen. Alle erstellten Typ-Instanzen werden in dem Vector `types`, einem Attribut der Klasse `DatabaseHLCL`, gespeichert.

6. Die Methode `makeConstraints()` parsiert alle Constraint-Definitionen mit Hilfe eines für diesen Zweck erstellten Parsers (Klasse `ConstraintParser`) und legt für jeden Constraint eine Instanz der Klasse `Constraint` an. Alle eingelesenen

Constraints werden in dem Vector `constraints` abgelegt, ebenfalls ein Attribut der Klasse `DatabaseHLCL`.

Die letzten beiden Schritte sollen für eine Typ-Deklaration bzw. eine Constraint-Definition in den nächsten beiden Abschnitten etwas näher betrachtet werden.

7.2.1 Parsieren einer Typ-Deklaration

Der Typ-Parser (Klasse `TypeParser`) bekommt eine Liste mit allen Tokens einer Typ-Deklaration übergeben. Er besitzt verschiedene Methoden, die jeweils einen bestimmten Teil der Typ-Deklaration auswerten und das Attribut `type` – eine Instanz der Klasse `Type` – nach und nach aufbauen. Dabei verzweigen die Methoden ineinander in Abhängigkeit von der Syntax der High-Level-Constraint-Sprache. Es existieren die in der Tabelle 7.1 dargestellten Methoden.

Methode	Funktionalität
`readType()`	Liest das Schlüsselwort `type`.
`readName()`	Liest den Typbezeichner.
`readBaseType()`	Liest den Basis-Typ, sofern einer definiert wurde.
`readVariableType()`	Liest den Typ einer oder mehrerer Eigenschaften.
`readVariableNames()`	Liest die Namen der Eigenschaften.

Tabelle 7.1: Methoden der Klasse `TypeParser`

Tritt ein Syntax-Fehler auf, wird die Exception `FileFormatException` geworfen, die die Beschreibung des Fehlers sowie Zeilen- und Spaltennummer der Fehlerstelle enthält.

7.2.2 Parsieren einer Constraint-Definition

Zum Parsieren von Constraint-Definitionen wird die Klasse `ConstraintParser` verwendet. Die unterschiedlichen Teile einer Constraint-Definition werden durch diverse Methoden ausgewertet, die ineinander anhand der jeweiligen Syntax verzweigen. Jede Methode gibt die gelesenen Informationen an das Attribut `constraint` weiter, das eine Instanz der Klasse `Constraint` referenziert. Es enthält am Ende alle Daten, die zur Anwendung eines High-Level-Constraints nötig sind.

Constraints sind sehr komplexe Objekte. Daher genügt es nicht, die Daten in primitiven Objekten zu speichern. Wir benutzen verschiedene weitere Klassen aus dem Package `genged.constsolver`, um Teilbereiche der Datenstruktur zu kapseln: `EquationPart`, `Interval`, `MathFunction`, `ParconEquation` und `Variable`. Zum Parsieren existieren die in der Tabelle 7.2 dargestellten Methoden.

Methode	Funktionalität
`readConstraint()`	Liest das Schlüsselwort `constraint`.
`readName()`	Liest den Constraint-Bezeichner.
`readParameter()`	Liest einen Parameter, bestehend aus Typbezeichner und Parametername.
`readInterval()`	Liest Intervalle für Parameter vom Typ `ConstValue`.
`readBody()`	Wertet einen Anfang eines Teilbereich der Constraint-Definition aus und verzweigt zu einer der folgenden Methoden.
`readDescription()`	Liest die Beschreibung des Constraints.
`readDialog()`	Liest die Beschreibung zu einem Parameter.
`readOperationMode()`	Liest den Befehl zum Umschalten zwischen UND- und ODER-Modus zur Verknüpfung von Formeln des Constraints.
`readPriority()`	Liest die Priorität einer Formel.
`readVariableType()`	Liest den Typbezeichner einer lokalen Variable.
`readVariableName()`	Liest den Bezeichner einer lokalen Variable.
`readConstraintCall()`	Liest einen Constraint-Aufruf.
`readCallParameter()`	Liest den Parameter eines Constraint-Aufrufs inklusive Casting.
`readEquation()`	Liest eine Formel des Constraint-Systems und teilt sie in eine linke und rechte Formelseite.
`readRelationEquation()`	Wertet die Formelseiten einer linearen Formel oder einer Produktformel aus.
`readPointEquation()`	Wertet die Formelseiten für Punktgleichheit aus.
`readEuclidEquation()`	Wertet die Formelseiten für den Euklidischen Abstand zweier Punkt aus.
`readMaxEquation()`	Wertet die Formelseiten für eine Maximum-Formel[3] aus.
`readMathFunction()`	Liest ein mathematische Funktion, die auch verschachtelt sein kann.

Tabelle 7.2: Methoden der Klasse `ConstraintParser`

Tritt ein Syntax-Fehler auf, wird auch hier die Exception `FileFormatException` geworfen, die die Beschreibung des Fehlers sowie Zeilen- und Spaltennummer der Fehlerstelle enthält.

7.2.3 Datenstrukturen zur Typisierung

Jeder Typ wird durch die Klasse `Type` repräsentiert. Das betrifft alle zusammengesetzten Primitiven-Typen und die Grundtypen gleichermaßen. Jeder Typ bekommt

[3]Diese Formelart wurde in dieser Diplomarbeit nicht erläutert, da deren Funktionalität – im Widerspruch zur Dokumentation – in ParCon nicht zur Verfügung steht. Sie könnte dennoch benutzt werden, sofern ParCon in einer zukünftigen Version diese Funktion anbietet.

einen eindeutigen Namen, der in dem Attribut `name` abgelegt ist und von der Methode `getName()` zurückgegeben wird. Durch Aufruf der Methode `isSystemType()` kann bestimmt werden, ob es sich um eine Grundtypen handelt, den das System selbst bereitstellt. Eine Liste aller Typen, deren Eigenschaften durch Vererbung übernommen werden, liefert die Methode `getBasetypes()`. Ob ein Typ ein Basistyp von einem anderen Typ ist, entscheidet die Methode `isBasetypeOf()`. Jede Eigenschaft eines Typs wird durch einen Typbezeichner und einen Typen definiert. Eigenschaften werden bei der Auswertung durch den Typ-Parser mit der Methode `addVariable()` hinzugefügt. Die Abbildung 7.1 zeigt die Attribute und die wichtigsten Methoden der Klasse `Type`. Die Klasse ist von anderen Klassen unabhängig, sie enthält ausschließlich Referenzen auf Instanzen derselben Klasse, um Basistypen und Typen von Eigenschaften zu referenzieren.

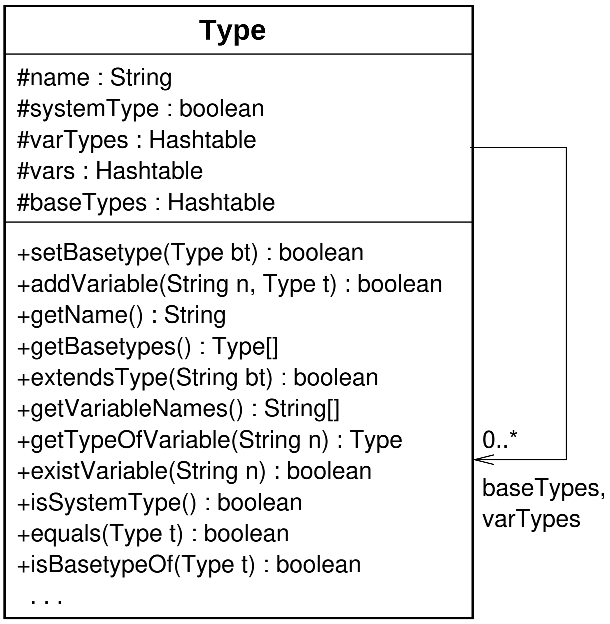


Abbildung 7.1: Klassendiagramm: `Type`

7.2.4 Datenstrukturen für Constraints

Die Datenstrukturen von Constraints sind wesentlich komplexer als bei Typen. Daher existieren verschiedene Klassen, die Teile eines Constraints kapseln.

Klasse `Constraint`: Die Klasse `Constraint` verwaltet alle Bestandteile eines Constraints: Parameter und Variablen, Formeln, Aufrufe anderer Constraints sowie Beschreibungen zum Constraint und zu seinen Parametern.

Jeder Parameter und jede Variable erhält eine Nummer. Diese Nummer dient unter anderem in Formeln als Platzhalter. Während der Verwendung des Constraints mit dem Constraint-Solver wird für jeden Parameter ein konkreter Name vergeben, der auf eine Primitive verweist. Lokale Variablen erhalten ebenfalls einen im gesamten Constraint-System eindeutigen Namen. Beide Arten werden durch die Klasse `Variable` repräsentiert.

Formeln werden durch die Klasse `ParconEquation` repräsentiert. Sie wurden bereits durch den Constraint-Parser in eine Vorstufe der PARCON-eigenen Schreibweise umgewandelt. Die Klasse `ConstraintCall` kapselt einen Constraint-Aufruf. Die wichtigste Methode innerhalb der Klasse `Constraint` ist `getEquations()`. Sie liefert bei der Übergabe von konkreten Parameter- und Variablennamen alle Gleichungen, die direkt an den Constraint-Solver PARCON übergeben werden können. Auch die Gleichungen, die sich aufgrund von Constraint-Aufrufen ergeben, sind enthalten.

Beschreibungen des Constraints bzw. seiner Parameter können mit den Methoden `getDescription()` und `getDialog()` abgefragt werden. Sie finden ihre Anwendung in den Dialogfenstern zur Constraint-Eingabe, die sowohl im Symbol- als auch im Verbindungs-Editor benutzt werden.

Klasse `Variable`: Durch diese Klasse werden Parameter und lokale Variablen beschrieben. Sie enthält neben einem eindeutigen Namen den Typ der Variable, der von der Methode `getType()` zurückgegeben wird. Um einen Parameter von einer lokalen Variable zu unterscheiden, existiert das Attribut `local`, das mit der Methode `isLocal()` abgefragt werden kann. Außerdem muß bei Parametern zwischen normalen Parametern und Parametermengen unterschieden werden. Die Methode `isSet()` liefert den Wert `true` zurück, wenn es sich um letzteres handelt. Numerische Parameter vom Typ `ConstValue` können Intervalle definieren, die durch die Klasse `Interval` gekapselt sind. Mit der Methode `isInInterval()` wird überprüft, ob sich ein konkreter Zahlenwert in einem der definierten Intervalle befindet.

Klasse `ParconEquation`: Diese Klasse repräsentiert eine Formel innerhalb eines Constraint-Systems. Die Priorität kann mit der Methode `getPriority()` erfragt werden. Um eine oder mehrere Formeln in der Schreibweise von PARCON zu erhalten, wird die Methode `getEquationsWithNames()` aufgerufen, der eine Hashtable übergeben wird, die Nummern von Parametern bzw. Variablen auf konkrete Namen abbildet. Handelt es sich bei einem Parameter um eine Parametermenge, wird statt einem Namen ein Vector von Namen referenziert. Eine Formel teilt sich in drei Teile auf: In eine linke und rechte Formelseite und das Relationszeichen. Die beiden Formelseiten werden jeweils durch eine oder mehrere Instanzen der Klasse `EquationPart` gekapselt, das Relationszeichen durch ein Token.

Klasse `EquationPart`: Diese Klasse repräsentiert den Teil einer Formel, der sich – je nach Formelart – verschiedenartig zusammensetzt. Daher werden verschiedene Arten zur Verfügung gestellt. Jeder Formelteil teilt sich in vier Bestandteile auf, die zusammen immer ein Produkt bilden:

Faktor: Ein Faktor wird durch eine Zahl dargestellt.

Konstanten: Konstanten ergeben sich aus Parametern vom Typ `ConstValue` und werden erst zur Laufzeit in konkrete Zahlen umgewandelt.

Mathematische Funktionen: Für diese Funktionen existiert eine weitere Klasse – `MathFunction`. Solche Funktionen können selbst ein Produkt verschiedenener Konstanten und weiterer Funktionen als Argument enthalten.

Variablen: Dabei handelt es sich um Parameter, die *nicht* vom Typ `ConstValue` sind, sowie um lokale Variablen. Ihnen wird zur Laufzeit ein für das gesamte Constraint-System eindeutiger Name gegeben, der sich auf eine Primitive bezieht.

Es darf nicht immer jeder dieser vier Bestandteile enthalten sein. Welche konkret vorhanden sind, hängt letztlich von der Art des Formelteils ab. Diese wiederum wird bestimmt durch die Art der Gesamtformel und durch die linke bzw. rechte Formelseite. Über die Methode `getKind()` kann die Art ermittelt werden. Zurückgegeben wird eine von mehreren statischen Konstanten, die in `EquationPart` definiert sind. Die wichtigste Methode dieser Klasse ist jedoch `getPart()`. Diese Methode bringt den entsprechenden Formelteil in die korrekte Formelschreibweise von PARCON , indem es Konstanten in konkrete Zahlen umformt, mathematische Funktionen mit den konkreten Argumenten berechnet und alle Bestandteile zusammenfaßt.

Die Abbildung 7.2 versucht, die Zusammenhänge aller hier erwähnten Klassen aufzuzeigen. Die Hauptklasse bildet dabei die Klasse `Constraint`.

7.2.5 Verwaltung von Typen und Constraints

Die Klasse `DatabaseHLCL` verwaltet Instanzen der Klassen `Type` und `Constraint`. Sie initiiert das Einlesen und Parsieren von Typen und Constraints der Definitionsdateien. Neben den bereits im Abschnitt 7.2 erläuterten Methoden existieren jedoch noch weitere, um auf bestimmte Typen und Constraints zuzugreifen.

Die Methode `getType()` gibt bei der Angabe eines Typbezeichners den dazugehörigen Typ zurück. Weitere Methoden zur Typ-Verwaltung sind nicht notwenig, da jeder Constraint, der Typen verwendet, eigene Referenzen auf diese Typen besitzt. Die Klasse `Type` wird von anderen Klassen kaum verwendet.

Für die Verwaltung von Constraints existieren dagegen vielfältige Methoden. Eine Liste von allen verfügbaren Constraints wird durch die Methode `getConstraints()` zurückgegeben. Wird als Parameter ein bestimmter Constraint-Name angegeben, werden nur die entspechenden Constraints gesucht. Um einen ganz bestimmten Constraint zu erhalten, kann die Methode `getConstraint()` benutzt werden. Hier genügt es jedoch nicht, ausschließlich den Namen anzugeben, da dieser mehrdeutig sein kann. Deshalb wird eine Kombination aus dem Namen und den Parametertypen verwendet, die jeweils durch einen Punkt getrennt sind (`name.type1.Set of type2.type3.typeN`). Groß- und Kleinschreibung müssen dabei unbedingt beachtet werden. Wird der entsprechende Constraint nicht gefunden, gibt die Methode `null` zurück. Mit der Methode `getConstraintSmart()` wird auf intelligentere Weise nach einem Constraint gesucht. Als Parameter wird wieder eine Kombination aus Name und Parametertypen übergeben. Wird der Constraint nicht gefunden, sucht diese Methode jedoch weiter, indem

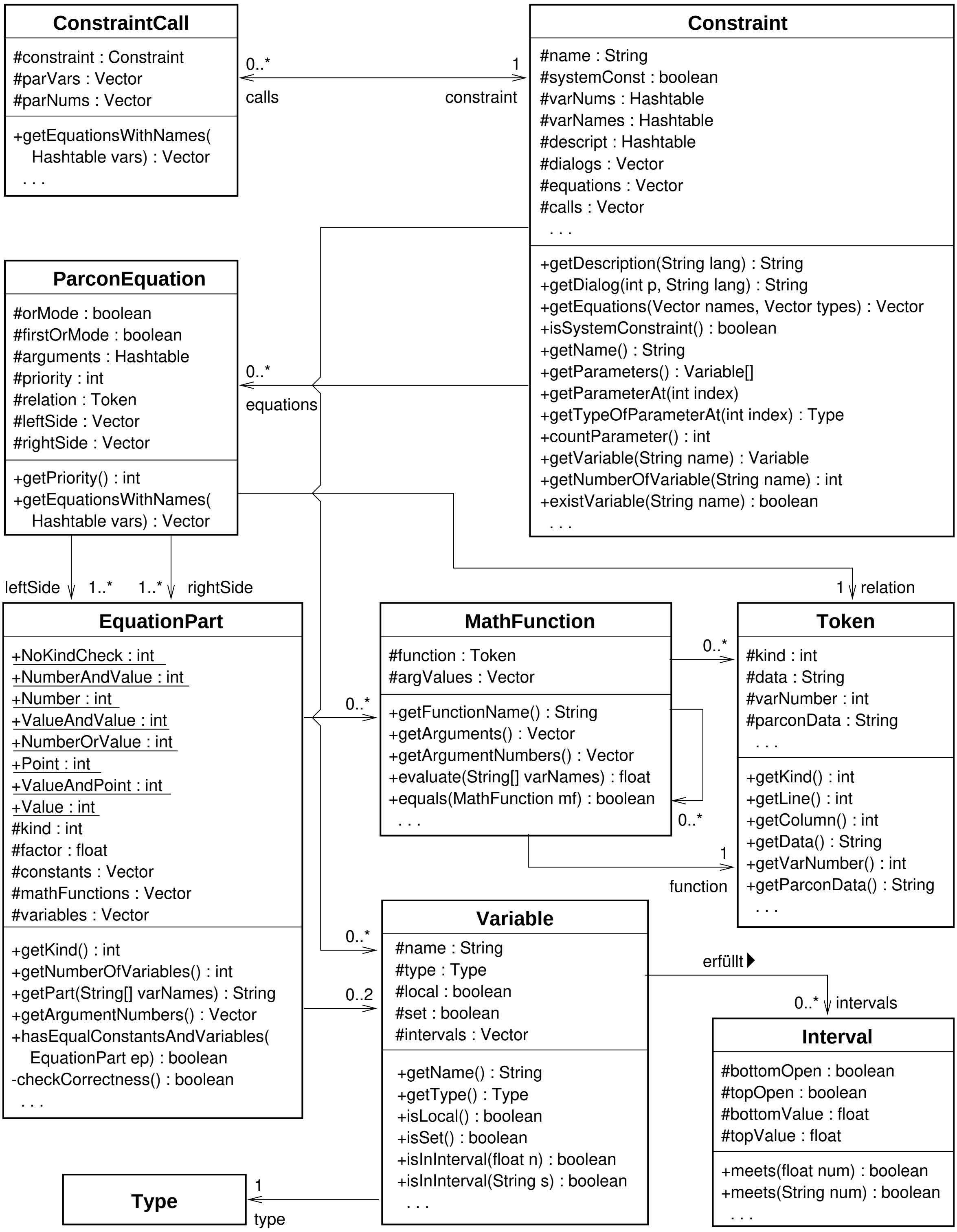


Abbildung 7.2: Klassendiagramm: Typ- und Constraint-definierende Klassen

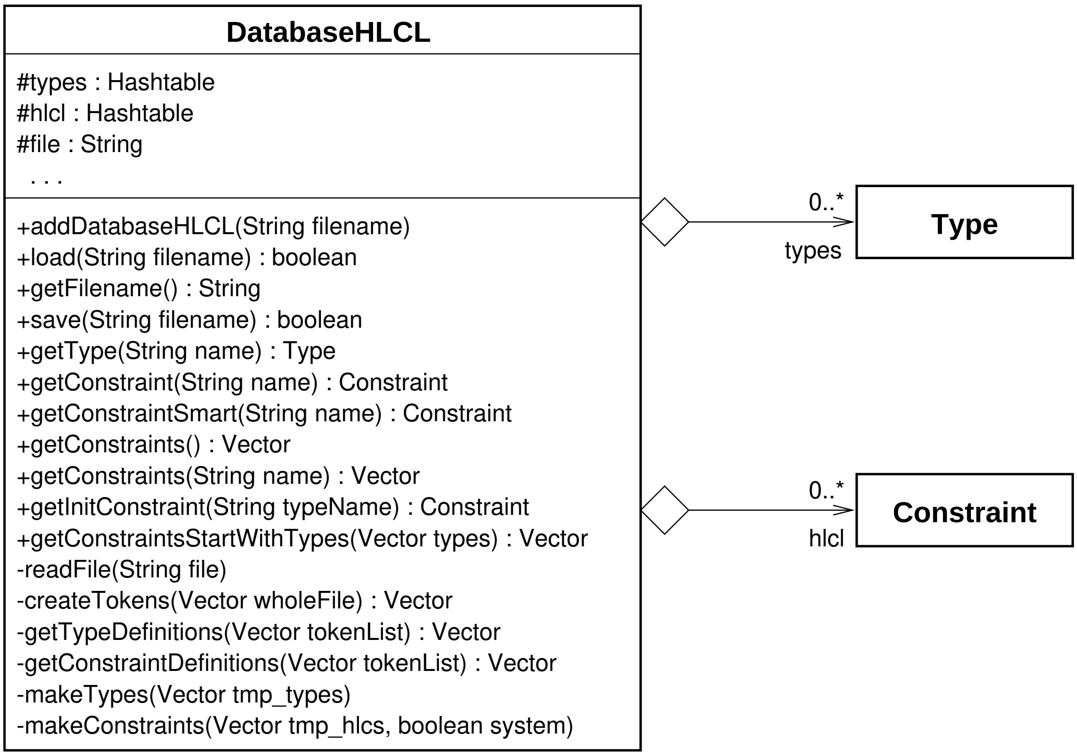


Abbildung 7.3: Klassendiagramm: `DatabaseHLCL`

nacheinander die verschiedenen Basistypen der Parameter eingesetzt werden. Die Methode `getConstraintsStartWithTypes()` sucht aus der Constraint-Datenbank jeden Constraint heraus, dessen erste Parameter mit bestimmten Typen übereinstimmen. Die Bezeichner der gewünschten Typen werden in Form eines Vectors als Parameter übergeben. Die Methode berücksichtigt bei der Suche auch Basistypen der übergebenen Typen.

Jede Primitive gehört zu einem bestimmten Typ. Um sicherzustellen, daß sie immer verschiedene Eigenschaften erfüllt, gibt es für die meisten Typen sogenannte Initialisierungs-Constraints. Die Methode `getInitConstraint()` liefert bei Angabe eines Typbezeichners den dazugehörigen Initialisierung-Constraint, sofern er existiert.

Die Abbildung 7.3 gibt einen Überblick über die Klasse `DatabaseHLCL` mit ihren Attributen, Methoden und Beziehungen zu anderen Klassen.

7.3 Anbindung an PARCON

Ein Constraint-System besteht aus verschiedenen variablen Werten sowie aus den Beziehungen zwischen diesen Werten. Viele der Variablen beziehen sich dabei auf Eigenschaften von Objekten, den Primitiven. Constraints verknüpfen dabei in der Regel verschiedene Primitiven miteinander. Mitunter werden auch auf einer einzelnen primitiven

Grafik spezielle Eigenschaften definiert.

Die Verbindung zum externen Constraint-Solver PARCON erfolgt unter anderem mit der Klasse `ConstSolver`. In den folgenden Ausführungen werden wir daher mitunter auch im Zusammenhang mit dieser Klasse von einem „Constraint-Solver“ reden. Damit das Lösen von Constraints unabhängig von konkreten Grafikobjekten bleibt, haben wir ein Interface `ConstSolverObject` für Objekte eingeführt, die an Constraints beteiligt sein können. Zur Anwendung eines High-Level-Constraints, wie wir ihn bisher in diesem Kapitel kennenlernten, wird die Klasse `ConstraintObject` benutzt, die konkrete Parameter für einen Constraint verwaltet.

7.3.1 Objekte im Constraint-Solver

Das Interface `ConstSolverObject` wird von allen Primitiven, Symbolen und Datentypen implementiert, die sich im Package `genged.language` befinden. Dieses Interface bietet hauptsächlich Methoden an, die im Zusammenhang mit der Klasse `ConstSolver` genutzt werden, die die Verbindung mit dem Constraint-Solver PARCON verwaltet.

Typen und Eigenschaften: Die Methode `getTypeName()` in dem Interface `ConstSolverObject` gibt den Typbezeichner eines Typs zurück, dessen Eigenschaften das Objekt besitzt. Dieser Typbezeichner entspricht einem Typ, der in der systeminternen Constraint-Datenbank definiert wurde. Dadurch erfolgt die Abbildung von Primitiven auf Typen. Die einzelnen Eigenschaften des Typs werden dabei auf die Attribute eines Objektes abgebildet. Dazu werden verschiedene Attribut-Schlüssel benutzt, die für jedes Objekt durch die Methode `getAttrKeys()` abgefragt werden können. Diese Schlüssel müssen genau einer Eigenschaft des Typs entsprechen. Jede Eigenschaft wird dabei bis auf einen numerischen Wert – dem Typ `Value` – dereferenziert. Diesem Typ ist der Java-Typ `int` zugeordnet. Über die Methoden `setValue()` und `getValue()` kommuniziert die Klasse `ConstSolver` mit dem Objekt. Sie setzt verschiedene Attribute, die von dem Constraint-Solver PARCON errechnet werden. Außerdem übermittelt sie an ihn die aktuellen Attributwerte. Die Abbildung 7.4 veranschaulicht diesen Sachverhalt an dem Beispiel eines Kreises. In der Constraint-Datenbank wird der Typ `Circle` durch Erweiterung anderer Typen definiert. In der Implementierung der Klasse `CircleObject` werden die Attribut-Schlüssel ebenfalls von übergeordneten Methoden bezogen. Dies ist allerdings eine reine Design-Entscheidung. Tatsache ist, daß *alle* Eigenschaften, die der Typ `Circle` besitzt, von der Methode `getAttrKeys()` zurückgegeben werden müssen. Auch die Methoden `setValue()` und `getValue()` müssen mit allen Eigenschaften arbeiten können. Der Kreis selbst besitzt implementierungs-technisch nur die Attribute `p1` und `p2`. Das sind die gegenüberliegenden Punkte der Bounding-Box. Die Methoden, die mit der Klasse `ConstSolver` kommunizieren, müssen daher den Wert für jeden Attribut-Schlüssel auf die beiden Punkte `p1` und `p2` umrechnen.

An- und Abmeldung beim Constraint-Solver: Damit die Attribute der Objekte dem Constraint-Solver bekanntgemacht werden können, muß ein Objekt bei der Klasse `ConstSolver` registriert werden. Beim Aufruf der Interface-Methode

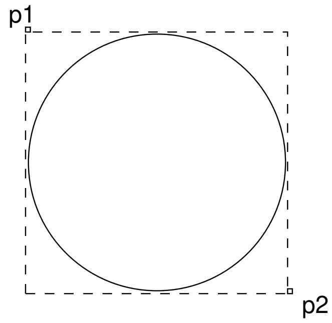
Constraint-Datenbank

```
type Base {
  Point lt;
  Value w, h;
}

type Ellipse extends Base {
  Point m;
  Value rx, ry;
}

type Circle extends Ellipse {
  Value r;
}
```

Grafik mit Bounding-Box



Klasse CircleObject

```
public String getTypeName() {
  return "Circle";
}

public Vector getAttrKeys() {
  Vector keys = super.getAttrKeys();
  keys.addElement("r");
  return keys;
}

 Rückgabewert ist ...
 ["lt.x", "lt.y", "w", "h",
  "m.x", "m.y", "rx", "ry", "r"]
```

Abbildung 7.4: Beziehung zwischen Constraint-Datenbank und Implementierung

`activate()` meldet sich ein Objekt selbständig bei dem übergebenen Constraint-Solver an. Was passiert bei dem Restrierungsprozeß? Es wird automatisch ein systemweit eindeutiger Name generiert, der ein Kürzel enthält, um die Art des Objekts zu identifizieren. Der Name wird in allen Constraints verwendet, die sich auf dieses Objekt beziehen. Das Kürzel im Namen wird übrigens vom Constraint-Solver über die Interface-Methode `getShort()` abgerufen. Weiterhin wird für das Objekt ein Initialisierungs-Constraint erstellt, der objektspezifische Eigenschaften definiert und dafür sorgt, daß ein grafisches Objekt nicht aus der Zeichenfläche hinausbewegt werden kann. Die Methode `deactivate()` meldet ein Objekt wieder ab. Über die Methode `getConstraintSolver()` wird eine Referenz auf den aktuell benutzten Constraint-Solver zurückgegeben. Außerdem gibt die Methode `isActive()` den Wert `true` zurück, wenn das Objekt bei der Klasse `ConstSolver` angemeldet ist.

Information des Constraint-Solvers über Änderungen: Ändern sich aufgrund von Benutzeraktivitäten die grafischen Attribute eines Objekts, muß der Constraint-Solver davon in Kenntnis gesetzt werden. Dazu wird die Methode `notifySolverFor()` benutzt, der ein Attribut-Schlüssel übergeben wird. Sie teilt die Änderung der aktuellen `ConstSolver`-Instanz mit. Sind alle Attribute ungültig, kann die Methode `notifySolverForAll()` aufgerufen werden. Jedes Objekt sollte außerdem ein Attribut besitzen, welches Aussagen darüber macht, ob ein Objekt bereits gelöste oder ungültige Werte enthält. Die Methoden `setSolved()` und `isSolved()` dienen dem Setzen und der Abfrage dieses Attributs. Über die Methode `getValue()` fragt der Constraint-Solver die aktuellen Werte der ungültigen Attribute ab.

Übermittlung neuer Werte durch den Constraint-Solver: Initiiert die Methode `solve()` der Klasse `ConstSolver` das Lösen des Constraint-Systems durch PARCON, wird zuvor für jedes registrierte Objekt, das ungültige Attribute besitzt, die Methode `startSolving()` aufgerufen. Neue Werte werden mit Hilfe der Methode `setValue()` gesetzt. Am Ende des Lösungs-Prozesses ruft der Constraint-Solver `finishSolving()` auf.

7.3.2 Constraints im Constraint-Solver

Einige Funktionalitäten der Klasse `ConstraintObject` wurden bereits im fünften Kapitel erläutert. Die wichtigsten Methoden in der Zusammenarbeit mit der Klasse `ConstSolver` sind `activate()`, `deactivate()` und `getEquations()`.

An- und Abmeldung beim Constraint-Solver: Beim Aufruf der Methode `activate()` versucht das Constraint-Objekt, sich selbständig beim Constraint-Solver anzumelden. Dies funktioniert jedoch nur, wenn alle Parameter des Constraints auf gültige Werte gesetzt wurden. Über die Methode `getEquations()` ruft der Constraint-Solver alle Formeln ab, die nach Einsetzen konkreter Variablennamen in korrekter PARCON-Schreibweise vorliegen. Die Variablennamen werden dazu als Parameter übergeben. Treten bei der Übertragung der Formeln zu dem externen Constraint-Solver PARCON Probleme auf, oder sind die Formeln widersprüchlich, wird die Anmeldung des Constraint-Objekts abgebrochen. Bereits registrierte Formeln werden wieder abgemeldet. Über die Methode `isActive()` kann festgestellt werden, ob die Registrierung erfolgreich verlief. Die Methode `getConstraintSolver()` gibt den aktuell benutzten Constraint-Solver zurück. Um ein Constraint-Objekt wieder abzumelden, wird die Methode `deactivate()` aufgerufen.

Initialisierungs-Constraints: Für Initialisierungs-Constraints wird eine abgeleitete Klasse namens `InitConstraintObject` benutzt, deren Funktionalitäten jedoch im wesentlichen gleich sind. Sie hält zusätzlich noch eine Referenz auf das Objekt, das mit diesem Constraint initialisiert wird.

Beteiligung von Operationen: Zur Vereinfachung wurden auch in der Klasse `Operation` des Packages `genged.language` die Methoden `activate()` und `deactivate()` eingeführt. Dadurch ist es möglich, alle Constraints einer Operation auf einmal beim Constraint-Solver an- bzw. abzumelden.

7.3.3 Verbindung zu PARCON

Mit der Klasse `ConstSolver` wurde ein Mechanismus implementiert, der Objekte und Constraints, die ein Constraint-System bilden, verwaltet und deren Werte und Formeln mit dem externen Constraint-Solver PARCON austauscht. Zum Austausch der Daten wird die Klasse `ParconInOut` verwendet, die eine Socketverbindung zu PARCON unter Nutzung des dazugehörigen Daemons herstellt.

Verwaltung der Objekte und Constraints: Zur Verwaltung der registrierten Objekte und Constraints existieren verschiedene Hashtables. Bei einigen Daten tritt Redundanz auf, die jedoch aus Optimierungsgründen beabsichtigt ist. In der Hashtable `nameToObject` wird der Name eines Objekts auf eine Objektreferenz abgebildet. `objectToName` bildet dagegen ein Objekt auf einen Namen ab. Alle Initialisierungs-Constraints der Objekte sind in dem Attribut `objectToInitConstraint` gespeichert. Die Hashtable `constraints` enthält schließlich alle angemeldeten Constraint-Objekte. Objekte, die ungültige Werte besitzen, werden in der speziellen Hashtable `invalidObjects` abgelegt.

An- und Abmeldung von Objekten: Mit der Methode `registerObject()` werden Objekte angemeldet, die das Interface `ConstSolverObject` implementieren. Es wird ein Name generiert, der folgendermaßen aufgebaut ist: {`t_`|`i_`}`XXXXccc`. Handelt es sich bei dem Objekt um ein Template, so wird der Prefix `t_` benutzt, bei Instanzen dagegen `i_`. Es folgt statt `XXXX` ein Kürzel zur Identifikation des Objekts. Dieses Kürzel wird über die objekteigene Methode `getShort()` abgerufen. Die Zeichen `ccc` werden durch eine Nummer ersetzt, die zur Laufzeit des Systems ständig inkrementiert wird. Auf diese Weise werden innerhalb eines Constraint-Solvers eindeutige Namen benutzt. Ein solcher Name repräsentiert ein Objekt, und damit auch einen möglichen Parameter für einen Constraint. Er wird in die verschiedenen Hashtables eingetragen. Außerdem wird ein Initialisierungs-Constraints der Klasse `InitConstraintObject` generiert, der über die Methode `registerConstraint()` angemeldet wird. Schlägt die Anmeldung des Constraints fehl, wird das gesamte Objekt wieder abgemeldet. Mit der Methode `unregisterObjekt` kann ein Objekt jederzeit wieder abgemeldet werden. Das hat allerdings zur Folge, daß auch alle Constraints, an denen das Objekt beteiligt ist, entfernt werden.

An- und Abmeldung von Constraints: Die Methode `registerConstraint()` dient dazu, Constraints anzumelden; das betrifft nicht nur Initialisierungs-Constraints. Dabei werden aus den Hashtables die Namen für übergebene Parameter herausgesucht. Enthält ein Constraint Objekte als Parameter, die noch nicht registriert worden sind, wird die Anmeldung automatisch vollzogen. Lokale Variablen erhalten einen speziellen Namen, der mit einem Unterstrich (_) beginnt. Er wird von der Klasse `ConstraintObject` automatisch vergeben und nicht in Hashtables gespeichert. Er ist für weitere Betrachtungen uninteressant, da er nur lokal in dem Teilsystem des Constraint-Objekts gültig ist. Nachdem alle Formeln an PARCON übermittelt wurden, wird überprüft, ob das Constraint-System widersprüchlich ist. In einem solchen Fall meldet PARCON einen Fehler; als Folge davon melden wir alle Gleichungen des Constraints wieder ab. Die Registrierung ist fehlgeschlagen. Das Constraint-Objekt wird mit der Methode `setInvalid()` als ungültig deklariert. Ist dagegen die Registrierung erfolgreich, werden die Werte aller Objekte, die dem Constraint als Parameter dienen, als ungültig gewertet, damit neue Werte berechnet werden. Mit der Methode `unregisterConstraint()` werden Constraints wieder abgemeldet.

Lösen des Constraint-Systems: Um das Constraint-System lösen zu lassen, wird die Methode `solve()` benutzt. Diese ruft zuerst für alle angemeldeten Objekte die Methode `startSolving()` auf. Danach überträgt sie die veränderten Werte der Objekte zu PARCON. Als Resultat gibt PARCON neue Werte aus, die an die einzelnen Objekte verteilt werden (Methode `setValue()`). Wenn alle Wer-

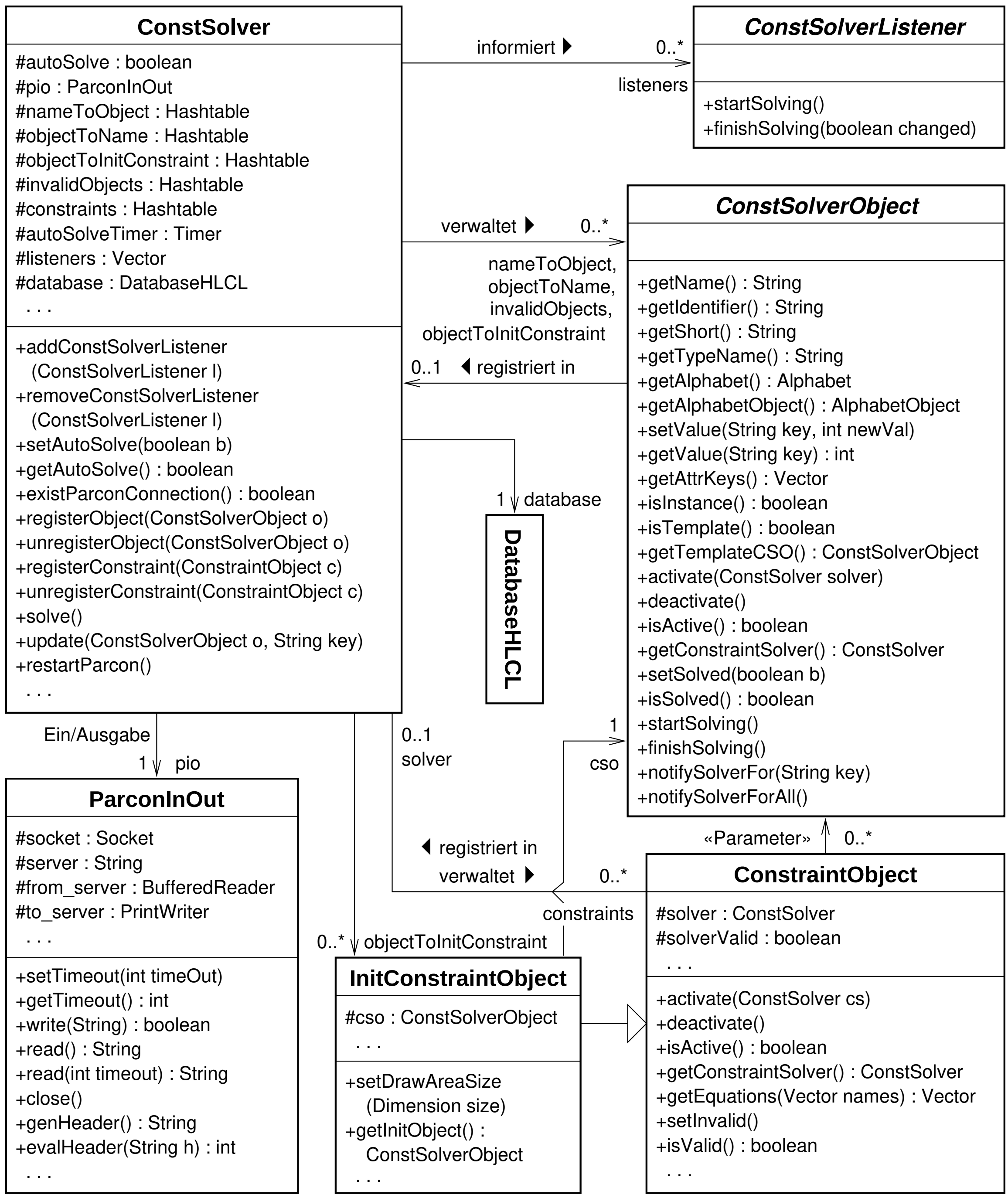


Abbildung 7.5: Klassendiagramm: `ConstSolver`

te übertragen wurden, informiert der Constraint-Solver alle Objekte durch Aufruf der Methode `finishSolving()`. Mit der Methode `setAutoSolve()` kann ein Modus aktiviert werden, in dem der Constraint-Solver registrierte Constraints automatisch löst. Nach jeder Meldung von ungültigen Objektwerten wird ein Timer initialisiert, der nach 500ms die Methode `solve()` aufruft. Diese Verzögerungszeit wurde bewußt eingebaut. Werden dem Constraint-Solver innerhalb dieser Zeit weitere ungültige Werte mitgeteilt, wird der Timer wieder zurückgesetzt, damit sich dieser Vorgang nicht mit dem Lösen der Constraints überschneidet. Über die Methoden `addConstSolverListener()` und `removeConstSolverListener()` können sich Objekte an- und abmelden, die über den Start und das Ende eines Lösungsprozesses informiert werden möchten. Beim Beenden des Lösungsprozesses wird beispielsweise in allen registrierten Listenern die Methode `finishSolving()` aufgerufen, die mitgeteilt bekommt, ob sich Werte der Objekte verändert haben. Verschiedene Editoren machen von dieser Information Gebrauch und aktualisieren beispielsweise ihre Zeichenfläche.

In der Abbildung 7.5 sind die Beziehungen des Constraint-Solvers zu den verschiedenen anderen Klassen dargestellt. Außerdem zeigt sie die wichtigsten Attribute und Methoden.

Kapitel 8

Symbol- und Verbindungs-Editor

In diesem Kapitel sollen die Ansichten und Kontrollstrukturen der beiden wichtigsten Teil-Editoren des Alphabet-Editors betrachtet werden. Beide Komponenten sind von der in Kapitel 4 bereits vorgestellten Klasse `AbstractEditor` abgeleitet.

8.1 Symbol-Editor

Der Symbol-Editor wird im Package `genged.alphabeteditor.symeditor` implementiert und dient der Erstellung von Symbolen und ihrer Primitiven, von Datentypen und internen Constraints. Die Klasse `SymbolEditor` enthält die Kernfunktionalität der Kontrollstruktur. In den Klassen `SymbolModel` und `SymbolSelectionModel` wird die Abbildung einiger Alphabet-Daten auf die Strukturansicht vorgenommen. Mit der Klasse `SymbolDrawArea`, die von der Klasse `AbstractDrawArea` im Package `genged.gui` abgeleitet ist, wird eine an den Symbol-Editor angepaßte Zeichenfläche bereitgestellt. Die Abbildung 8.1 zeigt die wichtigsten Attribute und Methoden sowie die Beziehungen der einzelnen Klassen zueinander.

Die Benutzeroberfläche enthält neben der Strukturansicht und der Zeichenfläche zwei weitere wichtige Komponenten: Eine Werkzeugleiste, aus der verschiedene Werkzeuge zur Erstellung und Bearbeitung der Grafiken ausgewählt werden, sowie ein Farbmenü, mit dessen Hilfe den verschiedenen Grafiken Linien- und Füllfarben zugeordnet werden.

8.1.1 Benutzeraktionen

Der Benutzer kann eine Vielzahl von Aktionen ausführen – entweder unter Nutzung der Menü- und Symbolleiste oder durch direkte Manipulation von Objekten in der Strukturansicht oder Zeichenfläche. Für die Aktionen der Menüs und Symbole wird konsequent der Mechanismus zur Verwaltung von Benutzeraktionen angewendet, der in Kapitel 6 vorgestellt wurde. Für jede der betreffenden Aktionen existiert eine sogenannte Eintrittsmethode in der Klasse `SymbolEditor`. Um sie von den „normalen" Methoden zu unterscheiden, beginnt der Name jeweils mit einem Großbuchstaben. Außerdem wer-

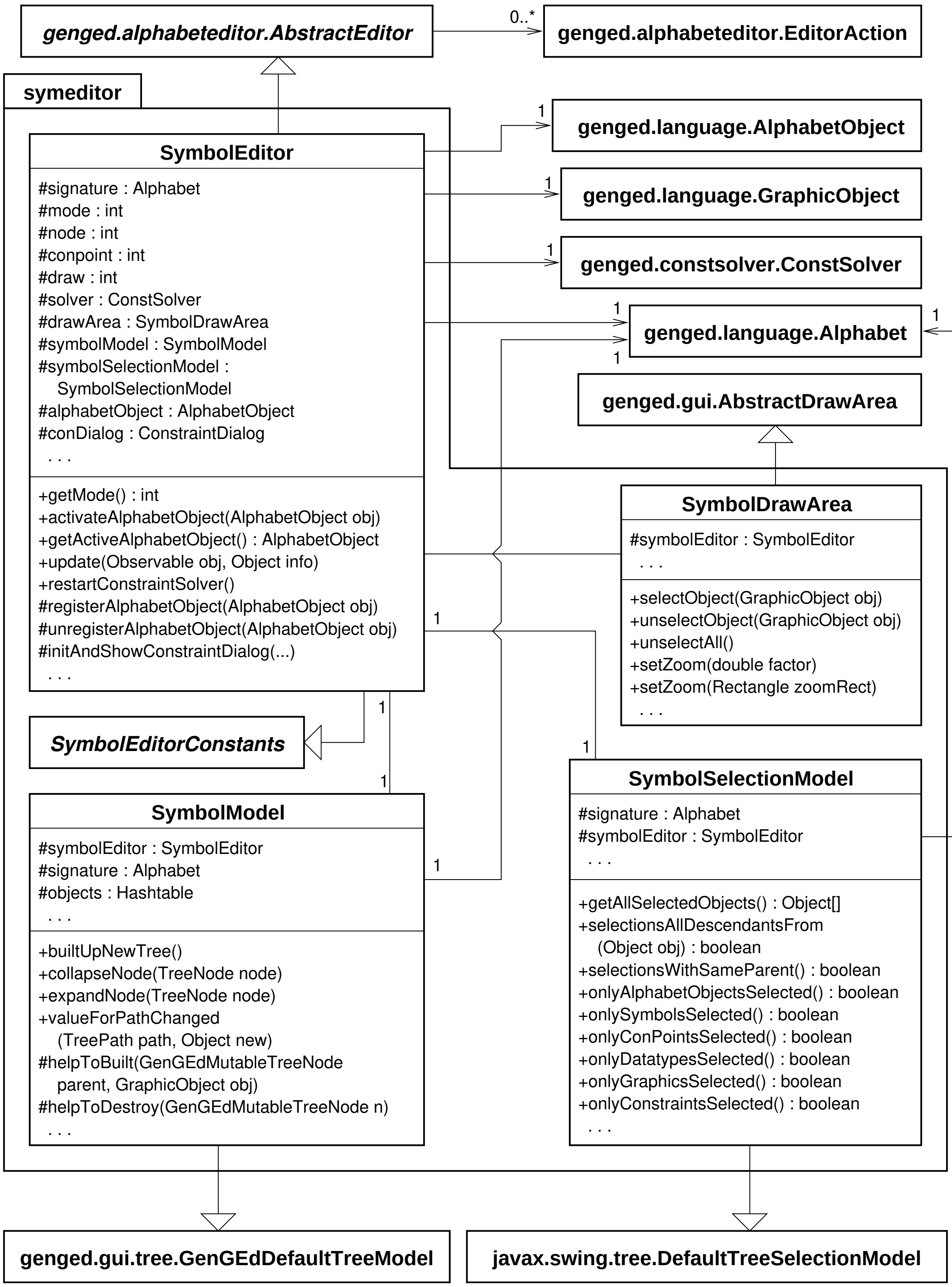


Abbildung 8.1: Klassendiagramm: Package `genged.alphabeteditor.symeditor`

den ihr immer genau zwei Parameter übergeben: `EditorAction` und `ActionEvent`. Die Tabellen 8.1 und 8.2 listen alle Eintrittsmethoden des Symbol-Editors mit einer kurzen Beschreibung zu ihrer Funktion auf.

Nicht alle Benutzeraktionen können zu jeder Zeit ausgeführt werden. Viele sind abhängig von dem aktuellen Zustand des Symbol-Editors. Andere wiederum hängen von der Anzahl und Art der markierten Objekte in der Strukturansicht und Zeichenfläche ab. Daher existieren verschiedene Methoden, die die gesamte Benutzeroberfläche – eingeschlossen der verschiedenen Elemente der Benutzeraktionen – aktualisieren. Die wichtigsten werden kurz vorgestellt:

Die Methode `updateActionAccess()` paßt die Checkboxen des Symbol-Editor an die eingestellten Zustände an. Hier wird beispielsweise festgelegt, ob Raster oder Fadenkreuz der Zeichenfläche eingeschaltet sind. Mit der Methode `updateToolActions()` werden alle Benutzeraktionen, die unter anderem über die Werkzeugleiste verfügbar sind, „enabled" oder „diabled" gesetzt. Das Farbmenü wird aufgrund der aktuellen Markierung von Primitiven in der Methode `updateColorPanel()` aktualisiert. Alle Aktionen, die nur ausgeführt werden können, wenn Objekte markiert wurden, werden aufgrund der aktuellen Markierung in der Methode `updateSelectionDependendActions()` „enabled" oder „diabled".

8.1.2 Strukturansicht

In der Strukturansicht werden die erstellten Symbole, Primitiven, Datentypen und Constraints durch eine Baumstruktur dargestellt. Dazu muß das Modell des Systems – das Alphabet – auf ein passendes Strukturmodell abgebildet werden. Die Klasse `SymbolModel` wurde implementiert, um diese Aufgabe zu erfüllen. Sie ist abgeleitet von der Klasse `DefaultTreeModel`, die bereits von den Swing-Klassen zur Verfügung gestellt wird. In der Klasse `SymbolSelectionModel`, die von `DefaultTreeSelectionModel` abgeleitet ist, wird durch verschiedene Methoden festgelegt, wann welche Objekte markiert werden dürfen.

Beide Strukturmodelle arbeiten eng mit der Klasse `SymbolEditor` zusammen. Außerdem implementieren beide Klassen die drei verschiedenen Alphabet-Listener, die über Änderungen des Alphabets informieren. Objekte können markiert und umbenannt werden. Daher dient die Strukturansicht nicht nur als Ansicht, sondern erfüllt auch eine gewisse Kontrollfunktion.

8.1.3 Zeichenfläche

Die Zeichenfläche erfüllt ebenfalls sowohl eine Ansichts- als auch eine Kontrollfunktion. Die sichtbaren Bestandteile eines Alphabets können durch die Zeichenfläche angezeigt werden. Wird in dem Symbol-Editor ein Symbol bearbeitet, werden alle Primitiven der ersten Hierarchie-Ebene des Symbols in die Zeichenfläche eingefügt. Ein Symbol stellt eine Gruppe von Primitiven dar. Es kann auch wieder Untergruppen besitzen, deren Objekte jedoch nicht separat in die Zeichenfläche eingefügt werden. Natürlich werden sie dennoch gezeichnet, da das Gruppenobjekt alle Mitglieder der Gruppe kapselt.

Eintrittsmethode	Beschreibung der Funktionalität
`AddCatchedConPoint()`	Aktiviert den Arbeitsmodus zum Hinzufügen eines Verbindungspunktes im Fangmodus
`AddNode()`	Aktiviert den Arbeitsmodus zum Hinzufügen eines Knotens
`AddNormalConPoint()`	Aktiviert den Arbeitsmodus zum Hinzufügen eines frei positionierten Verbindungspunktes
`AutoSolveOnOff()`	Schaltet den Modus zum automatischen Lösen von Constraints ein und aus
`ChangeObjectPosition()`	Ändert die Zeichenreihenfolge von Primitiven
`ChoiceMode()`	Aktiviert den Arbeitsmodus zum Auswählen, Verschieben, Skalieren, Rotieren und Verzerren von Objekten
`ColorPanelOnOff()`	Macht das Farbmenü sichtbar oder unsichtbar
`Copy()`	Kopiert markierte Objekte in die Zwischenablage (nicht implementiert)
`Cut()`	Kopiert markierte Objekte in die Zwischenablage und entfernt sie aus dem Editor (nicht implementiert)
`GridOnOff()`	Schaltet das Raster der Zeichenfläche ein und aus
`GroupObjects()`	Erstellt eine Gruppe aus den markierten Primitiven
`ImportBitmap()`	Importiert eine Bilddatei (GIF, JPG)
`JavaExpressionOnOff()`	Schaltet die Darstellung des Java-Ausdrucks bei Datentypen ein und aus
`MoveCatchedConPoint()`	Aktiviert den Arbeitsmodus zum Verschieben eines Verbindungspunktes im Fangmodus
`MoveConPoint()`	Aktiviert den Arbeitsmodus zum Verschieben eines Verbindungspunktes an eine beliebige Position
`MoveNode()`	Aktiviert den Arbeitsmodus zum Verschieben eines Knotens
`NewConstraint()`	Erstellt einen neuen internen Constraint
`NewDatatypeObject()`	Erstellt einen neuen Datentyp mit einem Default-Wert
`NewFile()`	Erstellt ein neues leeres Alphabet
`NewSymbolObject()`	Erstellt ein neues leeres Symbol
`NodeWorkGroup()`	Zeigt die Werkzeugauswahl zur Knotenbearbeitung
`OpenFile()`	Öffnet eine Alphabet-Datei von einem Datenträger
`Paste()`	Kopiert Objekte der Zwischenablage in den Symbol-Editor (nicht implementiert)
`Redo()`	Führt rückgängig gemachte Aktionen erneut aus
`Remove()`	Entfernt alle markierten Objekte aus dem Alphabet
`RemoveConPoint()`	Aktiviert den Arbeitsmodus zum Entfernen eines Verbindungspunktes
`RemoveNode()`	Aktiviert den Arbeitsmodus zum Entfernen eines Knotens
`RestartParCon()`	Neustarten des Constraint-Solvers PARCON
`ReticuleOnOff()`	Schaltet das Fadenkreuz der Zeichenfläche ein/aus

Tabelle 8.1: Eintrittsmethoden in der Klasse `SymbolEditor`, Teil 1

Eintrittsmethode	Beschreibung der Funktionalität
`SaveFile()`	Speichert das Alphabet in eine Datei
`SetDrawAction()`	Aktiviert ein Zeichenwerkzeug in einem Arbeitsmodus
`SetFill()`	Setzt eine Standard-Füllung für markierte Primitiven (teilweise implementiert)
`SetFillColor()`	Setzt die Füllfarbe für markierte Primitiven oder als Standard
`SetFillColorProgress()`	Setzt den Farbverlauf einer Füllung für markierte Primitiven oder als Standard (nicht implementiert)
`SetFillPattern()`	Setzt das Füllmuster für markierte Primitiven oder als Standard
`SetLineColor()`	Setzt die Linienfarbe für markierte Primitiven oder als Standard
`SetLineStyle()`	Setzt den Linienstil für markierte Primitiven oder als Standard (teilweise implementiert)
`SetLineWidth()`	Setzt eine Linienbreite für markierte Primitiven oder als Standard (nicht implementiert)
`ShowConPointGroup()`	Zeigt die Werkzeugauswahl zur Bearbeitung von Verbindungspunkten
`ShowFillStyleGroup()`	Zeigt die Werkzeugauswahl zur Bearbeitung von Fülleigenschaften
`ShowFontDialog()`	Setzt die Schriftart für markierte Text-Primitiven bzw. Datentypen oder als Standard
`ShowInvisibleOnOff()`	Schaltet die Darstellung von unsichtbaren Objekten in der Zeichenfläche ein und aus
`ShowLineGroup()`	Zeigt die Auswahl von Linienwerkzeugen
`ShowLineStyleGroup()`	Zeigt die Werkzeugauswahl zur Bearbeitung von Linieneigenschaften
`ShowOvalGroup()`	Zeigt die Auswahl von Ellipsenwerkzeugen
`ShowPolygonGroup()`	Zeigt die Auswahl von Polygonwerkzeugen
`ShowRectGroup()`	Zeigt die Auswahl von Rechteckwerkzeugen
`ShowZoomGroup()`	Zeigt die Werkzeugauswahl zum Ändern des Maßstabs der Zeichenfläche (Zoom)
`SmartNodes()`	Aktiviert den Arbeitsmodus zur intelligenten Knotenbearbeitung
`Solve()`	Löst alle internen Constraints des aktivierten Symbols oder Datentyps
`ToolPanelOnOff()`	Macht die Werkzeugleiste sichtbar oder unsichtbar
`Undo()`	Macht ausgeführte Aktionen rückgängig
`UngroupObjects()`	Löst eine markierte Gruppe von Primitiven auf
`VisibilityOnOff()`	Schaltet die Sichtbarkeit unsichtbarer Objekte in der Zeichenfläche ein und aus
`Zoom1To1()`	Setzt den Zoom-Faktor der Zeichenfläche auf 1
`ZoomAll()`	Setzt den Zoom-Faktor der Zeichenfläche auf einen optimalen Wert (alle Objekte sind sichtbar)
`ZoomMinus()`	Erniedrigt den Zoom-Faktor der Zeichenfläche
`ZoomPlus()`	Erhöht den Zoom-Faktor der Zeichenfläche

Tabelle 8.2: Eintrittsmethoden in der Klasse `SymbolEditor`, Teil 2

Über die verschiedenen Arbeitsmodi der Zeichenfläche, die bereits im Kapitel 6 erläutert wurden, beeinflußt der Symbol-Editor die Arbeitsweise der Kontrollstrukturen. Es wurde eine Klasse `SymbolDrawArea` von der Klasse `AbstractDrawArea` abgeleitet. Außerdem wurden alle drei Alphabet-Listener implementiert, um auf Veränderungen des Alphabets zu reagieren. Das betrifft hauptsächlich das Markieren von Objekten.

8.1.4 Arbeitsmodi

Durch sogenannte Arbeitsmodi wird ein Teil des internen Zustands von dem Symbol-Editor beschrieben. Dabei handelt es sich um Konstanten, die in dem Interface `SymbolEditorConstants` definiert werden. Der aktuelle Arbeitsmodus wird in dem Attribut `mode` gespeichert und kann mit der Methode `getMode()` abgefragt werden. In jedem Arbeitsmodus werden nur bestimmte Aktionen ausgeführt. Oft wird dabei die Zeichenfläche verwendet, die verschiedene Bearbeitungsmodi zur Verfügung stellt. Ändert sich der Arbeitsmodus des Symbol-Editors, dann wird veranlaßt, daß auch für die Zeichenfläche ein neuer Modus aktiviert wird. Die Tabelle 8.3 stellt die jeweiligen Modi des Symbol-Editors und der Zeichenfläche gegenüber. Einige der Arbeitsmodi enthalten noch verschiedene Unterzustände. Diese werden eingerückt dargestellt.

Arbeitsmodus Symbol-Editor	Bearbeitungsmodus Zeichenfläche
CHOICE_MODE	CHOICE_MODE
NODE_MODE	
ADD_NODE	ADD_NODE_MODE
MOVE_NODE	MOVE_NODE_MODE
REMOVE_NODE	REMOVE_NODE_MODE
SMART_NODES	SMART_NODES_MODE
ZOOM_MODE	ZOOM_MODE
DRAW_MODE	OTHER_MODE
CONPOINT_MODE	
ADD_NORMAL_CONPOINT	ADD_NORMAL_CONPOINT_MODE
ADD_CATCHED_CONPOINT	ADD_CATCHED_CONPOINT_MODE
MOVE_CONPOINT	MOVE_CONPOINT_MODE
MOVE_CATCHED_CONPOINT	MOVE_CATCHED_CONPOINT_MODE
REMOVE_CONPOINT	REMOVE_CONPOINT_MODE
LINESTYLE_MODE	CHOICE_MODE
FILLSTYLE_MODE	CHOICE_MODE
CHOICE_ALL_MODE	CHOICE_ALL_MODE
OTHER_MODE	OTHER_MODE

Tabelle 8.3: Arbeitsmodi des Symbol-Editors und der Zeichenfläche

In dem Modus `CHOICE_MODE` können Grafiken markiert, verschoben, skaliert, rotiert oder verzerrt werden. Der `NODE_MODE` dient verschiedenen Möglichkeiten der Knotenbearbeitung. Er besitzt einige Unterzustände, die in dem Attribut `node` gespeichert sind. Im Arbeitsmodus ZOOM_MODE kann der Maßstab der Zeichenfläche verändert werden. Der wichtigste Modus ist jeoch DRAW_MODE. Er besitzt sehr viele Unterzustände, die aufgrund ihrer Anzahl nicht in der Tabelle aufgeführt sind. Jeder dieser Unterzustände steht für eine Primitive, die in diesem Arbeitsmodus erstellt werden kann. Der aktuelle Zustand wird im Attribut draw abgelegt. Auch der Arbeitsmodus CONPOINT_MODE besitzt Unterzustände. In ihnen können Verbindungspunkte für Primitiven erstellt und bearbeitet werden. Diese Unterzustände werden im Attribut conpoint gespeichert. In den Arbeitsmodi LINESTYLE_MODE und FILLSTYLE_MODE werden Linien- und Füllfarbe festgelegt. Dabei arbeitet der Benutzer aufgrund des eingestellten Modus der Zeichenfläche trotzdem weiter mit den Primitiven. Im CHOICE_ALL_MODE können zusätzlich Verbindungspunkte angezeigt und ausgewählt werden. Dieser Modus wird bei der Constraint-Definition verwendet, um auch Verbindungspunkte als Constraint-Parameter festzulegen. Der Arbeitsmodus OTHER_MODE hat im Symbol-Editor keinerlei Bedeutung – er wird zur Zeit nicht verwendet.

8.1.5 Definition der Bestandteile des Alphabets

Symbole und Datentypen: Symbole und Datentypen können über die Menü- oder Symbolleiste erzeugt werden. Es kann allerdings nur genau ein Alphabet-Objekt in der Zeichenfläche angezeigt werden. Über die Strukturansicht kann zwischen Symbolen bzw. Datentypen umgeschaltet werden. Wir sprechen in diesem Zusammenhang von dem „aktuellen Symbol“ oder dem „aktuellen Datentyp“. Bei jeder Markierung eines Objekts ruft die Strukuransicht in der Klasse SymbolEditor die Methode valueChanged() auf, die durch das Interface ChangeListener spezifiziert wird. Handelt es sich bei dem markierten Objekt um ein Symbol oder einen Datentyp, wird die Methode activateAlphabetObject() benutzt, um das neue Alphabet-Objekt zu aktivieren. Dazu müssen Primitiven des „alten“ Objekts von der Zeichenfläche ab- und die Primitiven des „neuen“ Objekts angemeldet werden. Die Methode getActiveAlphabetObject() gibt das Alphabet-Objekt – Symbol oder Datentyp – zurück, das zur Zeit bearbeitet wird.

Primitiven: Primitiven bilden die Bestandteile eines Symbols. Die Abbildung 8.2 gibt einen Überblick über alle Primitiven, die innerhalb des Symbol-Editors verwendet werden können. Zur Erstellung einer Primitive muß der Arbeitsmodus DRAW_MODE eingeschaltet und ein konkretes Werkzeug ausgewählt sein. Dadurch wird in der Zeichenfläche der Modus OTHER_MODE eingeschaltet, der keine eigenständigen Mausaktionen der Zeichenfläche zuläßt. Stattdessen wertet der Symbol-Editor selbst verschiedene Maus- und Rubberband-Aktionen (Package gui.event) aus, die er von der Zeichenfläche über Maus- und Rubberband-Listener erhält. Es wird unterschieden zwischen Primitiven, die durch Drag-And-Drop, und zwischen solchen, die durch einen oder mehrere Mausklicks erstellt werden. Jede Klasse einer Primitive enthält deshalb ein Attribut createByDrag, welches diesen Sachverhalt klärt. Abhängig von diesem Attribut wird die statische Methode createTemplate() der Factory-Klasse TemplateFactory des Packages genged.language aufgerufen, wenn ein Mausklick erfolgt (Methode mouseClicked()) oder ein Rubberband aufgezogen wird (Methode rubberBandStarted()). Daraufhin wertet die Primitive selbst

Maus- und Rubberband-Aktionen aus. Sie ist selbst dafür verantwortlich ihre Form festzulegen. Ziel dieses Mechanismus ist, das System offen und flexibel zu gestalten. Dadurch können unabhängig von den einzelnen Kontrollstrukturen beliebige weitere Primitiven implementiert werden. Ist die Form der Primitive vollständig festgelegt oder bricht der Benutzer den Prozeß ab, ruft die Primitive die eigene Methode `finishCreation()` auf, die über einen Observer-Mechanismus den Symbol-Editor informiert.

Alle Primitiven sind skalierbar und können auf der Zeichenfläche frei positioniert werden. Primitiven, die auf Polygonen beruhen (z.B. Linien, Freihand-Zeichnungen oder ein geschlossenes Polygon), können außerdem rotiert und verzerrt werden. Besondere Eigenschaften einzelner Primitiven sind änderbar. Das betrifft beispielsweise den Start- und Endwinkel eines Ellipsenausschnitts oder den Rundungsradius eines abgerundeten Rechtecks. Für viele der Primitiven kann eine Linien- bzw. Umrißfarbe und eine Füllfarbe festgelegt werden. Zur Bearbeitung einer Primitive muß einer der verschiedenen Arbeitsmodi aktiviert werden. Die Zeichenfläche wird darüber informiert, so daß der Benutzer über Funktionalitäten der Zeichenfläche Objekte verändern kann.

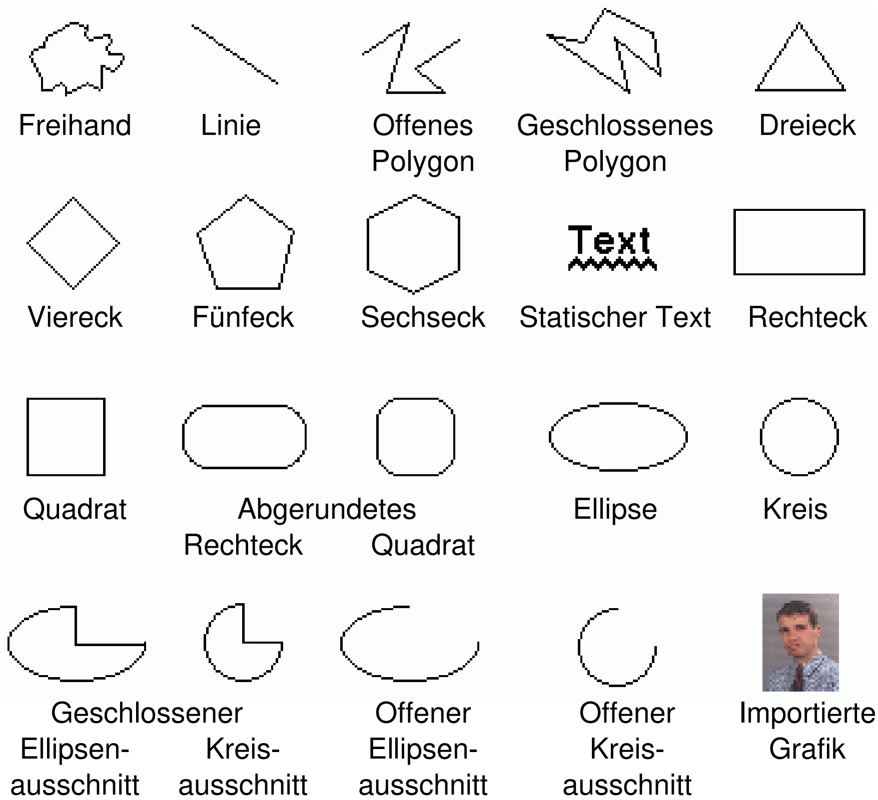


Abbildung 8.2: Zur Verfügung stehende Primitiven des Symbol-Editors

Interne Constraints: Es können mit Hilfe eines komplexen nicht-modalen Dialogfensters interne Constraints zwischen den Primitiven eines Symbols erstellt werden. Dazu wird die Klasse `ConstraintDialog` aus dem Package `genged.gui` verwendet. Das Dialogfenster existiert während der gesamten Arbeit mit dem Alphabet-Editor. Es wird zwischendurch nicht geschlossen, sondern nur unsichtbar gemacht. Um das Dialogfenster mit einem bestimmten Constraint zu initialisieren und sichtbar zu machen, wird die Methode `initAndShowConstraintDialog()` in der Klasse `SymbolEditor` aufgerufen. Wenn der nicht-modale Dialog aktiv und damit sichtbar ist, gibt die Methode `isConstraintDialogActive()` den Wert `true` zurück. Die Klasse `SymbolEditor` implementiert unter anderem das Interface `ConstraintDialogSupplier` aus dem Package `genged.gui`, welches Methoden zur Kommunikation zwischen Editor und Dialog bereitstellt. Beispielsweise kann der Constraint-Dialog mit Hilfe der Methode `setTypeFilter()` die Objekte einschränken, die markiert werden können. So ist es möglich, daß bei der Zuweisung von Objekten als Constraint-Parameter nur passende „Typen" ausgewählt werden. Wird das Dialogfenster geschlossen, ruft der Dialog im Symbol-Editor die Methode `finishConstraintDialog()` auf, die die Änderungen des Constraints auswertet und dem Alphabet mitteilt. Constraints können automatisch gelöst werden, indem der Methode `setAutoSolve()` im Symbol-Editor der Wert `true` übergeben wird.

Änderungsinformationen: Der Symbol-Editor implementiert die verschiedenen Alphabet-Listener und ist bei dem Alphabet angemeldet. So bekommt er Änderungen der Primitiven, der Symbole bzw. Datentypen und der internen Constraints mitgeteilt und kann darauf reagieren.

8.1.6 Verwendung des Constraint-Solvers

Der Symbol-Editor arbeitet direkt mit der `ConstSolver`-Klasse zusammen. Eine Referenz wird in dem Attribut `solver` gespeichert. Der Constraint-Solver kann im Fehlerfall mit der Methode `restartConstraintSolver()` neu gestartet werden.

Da innerhalb der Zeichenfläche genau ein Alphabet-Objekt aktiv ist, müssen mit dem Constraint-Solver nur die internen Constraints dieses Objekts gelöst werden. Um das aktuelle Objekt bei dem Solver anzumelden, wird von der Methode `activateAlphabetObject()` die Methode `registerAlphabetObject()` aufgerufen. In dieser Methode wird die `activate()`-Methode des Alphabet-Objekts benutzt, um das Objekt selbst und alle seine Primitiven anzumelden. Die internen Constraints zu dem Objekt befinden sich in der internen Operation dieses Alphabet-Objekts. Nachdem das Operations-Objekt aus dem Alphabet ermittelt wurde, wird auch dafür die Methode `activate()` aufgerufen.

Die Methode `unregisterAlphabetObject()` wird benutzt, um ein Objekt wieder abzumelden. Über die `deactivate()`-Methoden der internen Operation und des Objekts selbst werden alle Constraints, auch die Initialisierungs-Constraints der Primitiven aus dem Constraint-Solver entfernt.

8.1.7 Beispiel: Nassi-Shneiderman-Diagramme

Wir wollen an dieser Stelle das Beispiel aus Kapitel 2 wieder aufgreifen und einige Symbole und Datentypen für Nassi-Shneiderman-Diagramme erstellen. Dafür werden gleiche oder ähnliche Begriffe und Namen für die einzelnen Symbole, Datentypen, Primitiven und Constraints verwendet wie im Abschnitt 2.1.7.

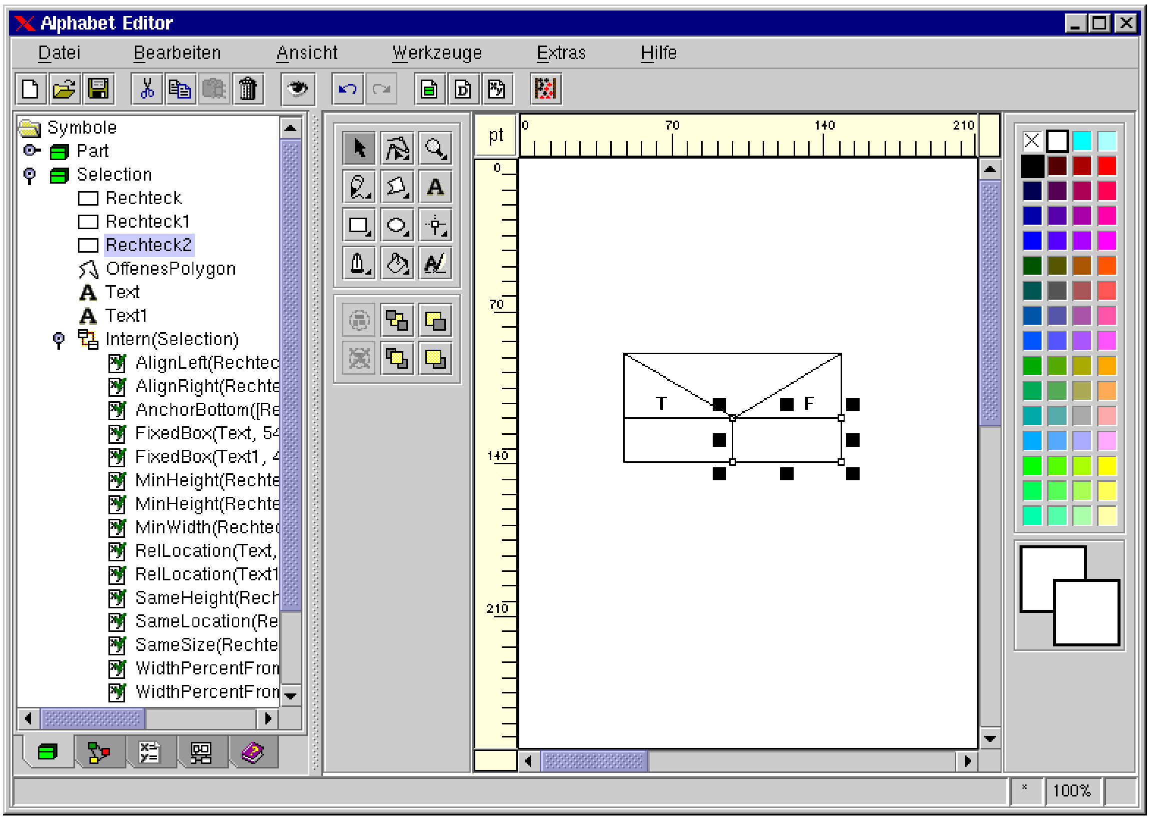


Abbildung 8.3: Screenshot: Arbeitsfenster mit aktiviertem Symbol-Editor

Die Abbildung 8.3 zeigt einen Screenshot von dem Arbeitsfenster des Alphabet-Editors. In ihm ist der Symbol-Editor aktiviert. In den Abbildungen 8.6 bis 8.5 sind jeweils ein Teil der Strukturansicht und die erstellte Symbolgrafik bzw. der Datentyp dargestellt. Wir haben folgende Elemente der Nassi-Shneiderman-Diagramme ins Alphabet aufgenommen:

- Datentypen:
  - Bedingung (`Condition`)
  - Ausdruck (`Expression`)
- Symbole:
  - Programmteil (`Part`)
  - Leere Abfolge als Platzhalter (`Sequence`)
  - Fallunterscheidung (IF-THEN) (`Selection`)

Eine wesentliche Abweichung ist bei dem Vergleich der Constraint-Parameter von Kapitel 2 und diesem Beispiel festzustellen. Numerische Parameter, die eine absolute Größe, einen Abstand oder eine Position angeben, sind mit dem Wert 100 multipliziert worden. Dieser Schritt war nötig, da das gesamte Alphabet intern mit der Einheit 1/100pt rechnet.

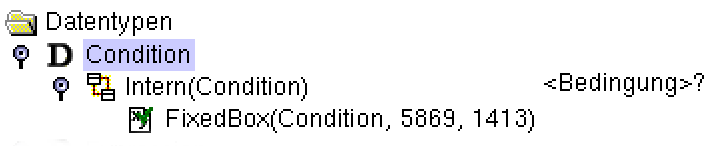


Abbildung 8.4: Nassi-Shneiderman-Diagramme: Datentyp `Condition`

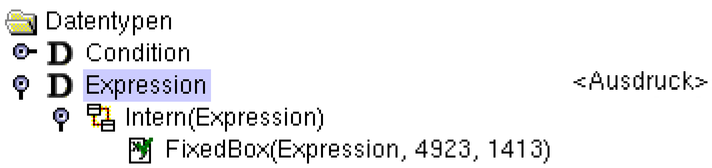


Abbildung 8.5: Nassi-Shneiderman-Diagramme: Datentyp `Expression`

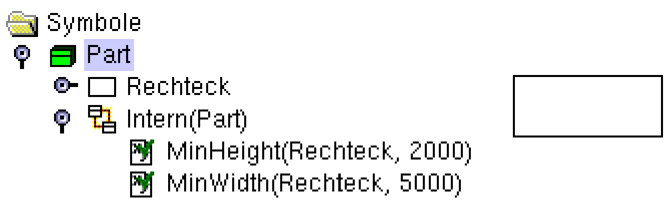


Abbildung 8.6: Nassi-Shneiderman-Diagramme: Symbol `Part`

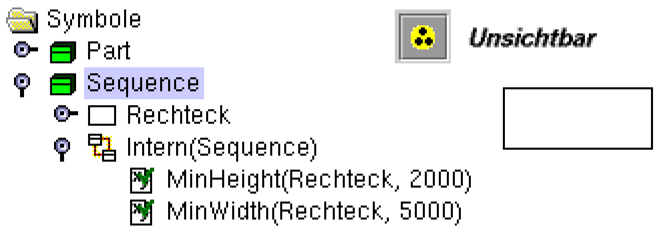


Abbildung 8.7: Nassi-Shneiderman-Diagramme: Unsichtbares Symbol `Sequence`

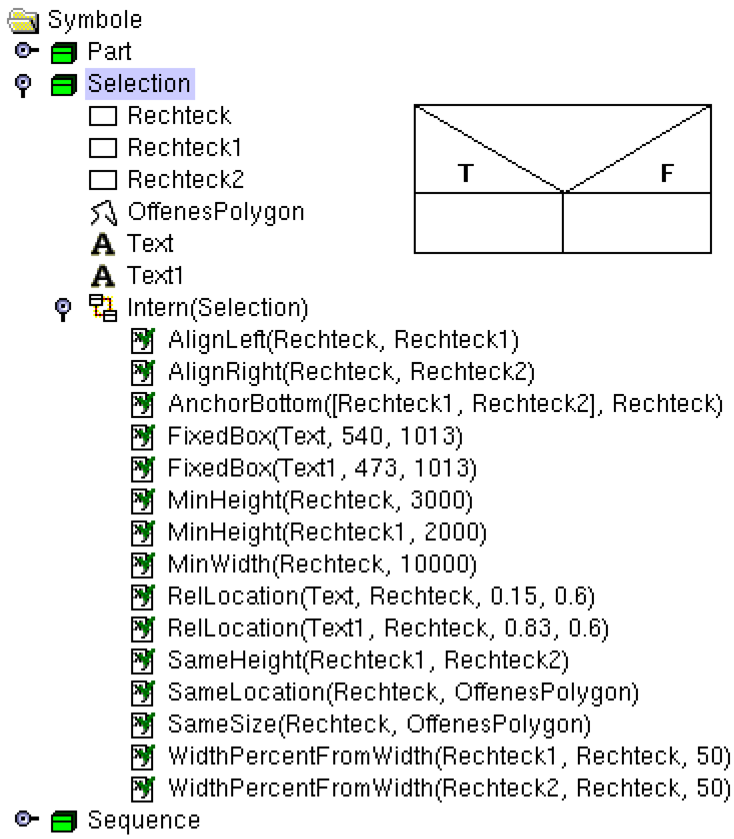


Abbildung 8.8: Nassi-Shneiderman-Diagramme: Symbol `Selection`

8.2 Verbindungs-Editor

Der Verbindungs-Editor wird im Package `genged.alphabeteditor.coneditor` implementiert. Mit ihm können grafische Verbindungen – externe Operationen – zwischen zwei Symbolen bzw. Datentypen definiert werden. Das Layout der einzelnen Primitiven innerhalb der beteiligten Objekte wird durch eine beliebige Anzahl externer Constraints beeinflußt. Die Klasse `ConnectionEditor` enthält die Kernfunktionalität der Kontrollstruktur. In den Klassen `ConnectionModel` und `ConnectionSelectionModel` wird die Abbildung verschiedener Alphabet-Daten auf das Modell der Strukturansicht vorgenommen. Mit der Klasse `ConnectionDrawArea`, die auch hier wieder von der Klasse `AbstractDrawArea` des Packages `genged.gui` abgeleitet ist, wird eine an den Verbindungs-Editor angepaßte Zeichenfläche bereitgestellt. Die Abbildung 8.9 stellt die wichtigsten Attribute, Methoden sowie die Beziehungen der einzelnen Klassen dar.

Die Benutzeroberfläche enthält neben der Strukturansicht und der Zeichenfläche noch eine kleine Werkzeugleiste, die Funktionalitäten zur Maßstabsänderung anbietet.

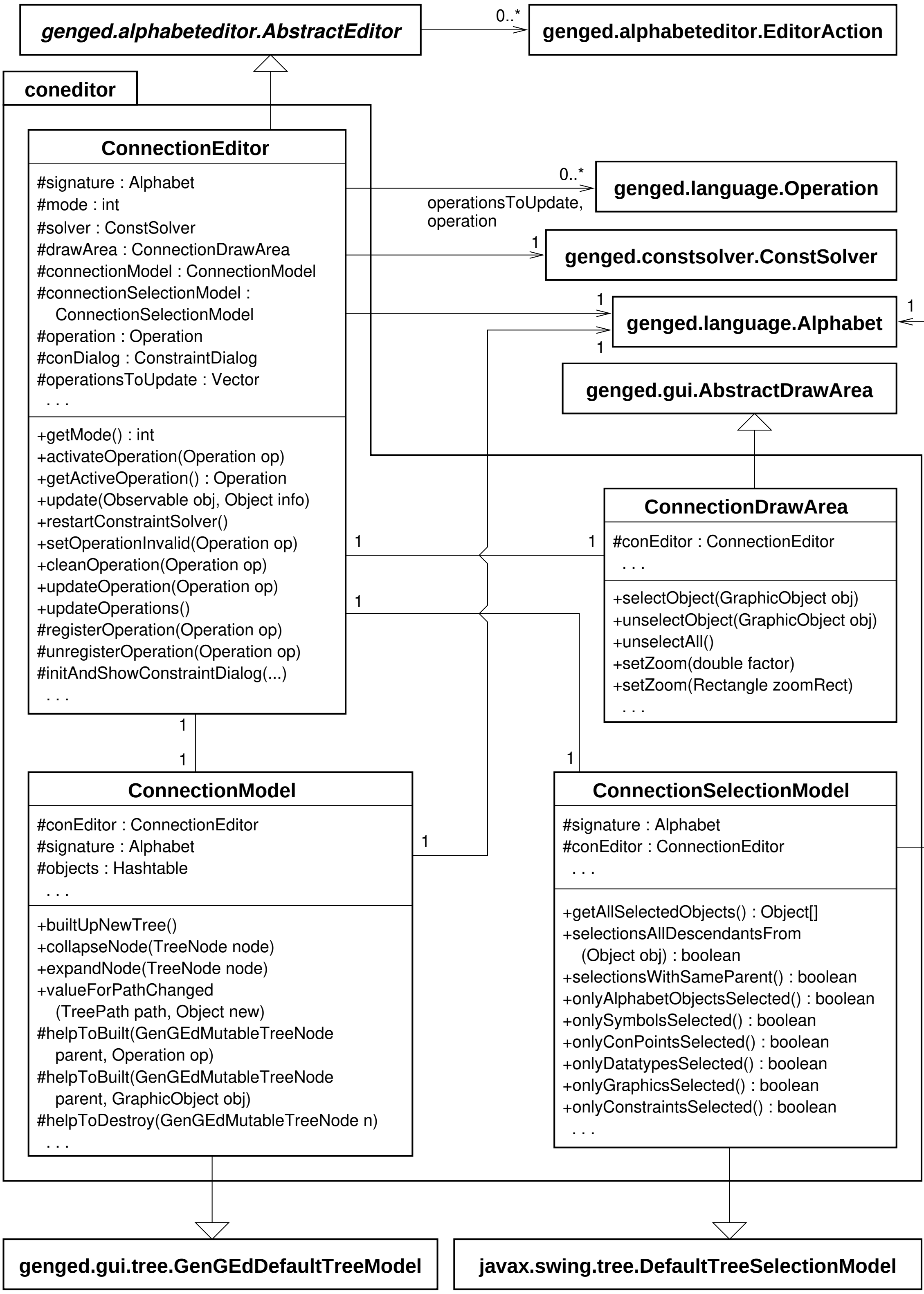


Abbildung 8.9: Klassendiagramm: Package `genged.alphabeteditor.coneditor`

8.2.1 Benutzeraktionen

Die Komplexität des Verbindungs-Editors ist wesentlich geringer als die des Symbol-Editors. Dennoch gibt es auch hier diverse Aktionen, die der Benutzer über die Menü- und Symbolleiste bzw. durch direkte Manipulation von Objekten in der Strukturansicht oder Zeichenfläche ausführen kann. In der Tabelle 8.4 werden daher alle Eintrittsmethoden des Verbindungs-Editors mit einer kurzen Beschreibung zu ihrer Funktion aufgelistet.

Um die Verfügbarkeit der verschiedenen Benutzeraktionen zu kontrollieren, existieren ähnliche Methoden wie im Symbol-Editor. Um beispielsweise dafür zu sorgen, daß die Checkboxen des Verbindungs-Editors mit internen Zuständen übereinstimmen und daß in Abhängigkeit der markierten Objekte Benutzeraktionen auf „enabled“ oder „disabled“ gesetzt werden, gibt es die Methoden `updateActionAccess()`, `updateToolActions()` und `updateSelectionDependendActions()`.

8.2.2 Strukturansicht

In der Strukturansicht werden die externen Operationen mit den daran beteiligten Objekten sowie den definierten Constraints dargestellt. Die Abbildung der Datenstrukturen des Alphabets auf die verwendeten Modelle der Baumstruktur erfolgt mit Hilfe der Adapterklassen `ConnectionModel` und `ConnectionSelectionModel`. Die Funktionalität entspricht im wesentlichen den entsprechenden Klassen des Symbol-Editors.

Beide Klassen sind direkt mit dem Verbindungs-Editor gekoppelt, um gegenseitige Funktionalitäten zu nutzen. Außerdem werden alle Alphabet-Listener implementiert. Änderungen der externen Operationen oder Constraints des Alphabets können so direkt auf die Strukturansicht übertragen werden. In der Strukturansicht können die meisten Einträge markiert und einige umbenannt werden. Somit dient diese Ansicht auch als Controller.

8.2.3 Zeichenfläche

In der Zeichenfläche werden generell zwei Instanzen von Alphabet-Objekten dargestellt – das Source- und das Target-Objekt. Dabei handelt es sich jedoch nur um temporäre Instanzen, die dem Festlegen externer Constraints dienen. Es ist möglich, daß beide Instanzen von demselben Template-Objekt abgeleitet wurden, um eine grafische Verbindung zu definieren. In die Zeichenfläche des Symbol-Editor wurden die einzelnen Primitiven der ersten Ebene eines Symbol eingefügt. Im Verbindungs-Editor enthält die Zeichenfläche dagegen genau zwei Alphabet-Objekte: Source und Target. Die Primitiven werden natürlich trotzdem gezeichnet, allerdings unter Benutzung der `paint()`-Methoden von Source- und Target-Objekt. Sofern die einzelnen Objekte es zulassen, können sie entsprechend der Funktionalität der Zeichenfläche verschoben, skaliert, rotiert oder verzerrt werden. Über die internen Constraints von Source und Target ist sichergestellt, daß das festgelegte Layout beibehalten wird.

Eintrittsmethode	Beschreibung der Funktionalität
AutoSolveOnOff()	Schaltet den Modus zum automatischen Lösen von Constraints ein und aus
ChoiceMode()	Aktiviert den Arbeitsmodus zum Auswählen, Verschieben, Skalieren, Rotieren und Verzerren von Objekten
Copy()	Kopiert markierte Objekte in die Zwischenablage (nicht implementiert)
Cut()	Kopiert markierte Objekte in die Zwischenablage und entfernt sie aus dem Editor (nicht implementiert)
GridOnOff()	Schaltet das Raster der Zeichenfläche ein und aus
NewConstraint()	Erstellt einen neuen externen Constraint
NewFile()	Erstellt ein neues leeres Alphabet
NewOperation()	Erstellt eine neue externe Operation zwischen zwei Alphabet-Objekten
OpenFile()	Öffnet eine Alphabet-Datei von einem Datenträger
Paste()	Kopiert Objekte der Zwischenablage in den Verbindungs-Editor (nicht implementiert)
Redo()	Führt rückgängig gemachte Aktionen erneut aus
Remove()	Entfernt alle markierten Objekte aus dem Alphabet
RestartParCon()	Neustarten des Constraint-Solvers PARCON
ReticuleOnOff()	Schaltet das Fadenkreuz der Zeichenfläche ein/aus
SaveFile()	Speichert das Alphabet in eine Datei
ShowInvisibleOnOff()	Schaltet die Darstellung von unsichtbaren Objekten in der Zeichenfläche ein und aus
ShowZoomGroup()	Zeigt die Werkzeugauswahl zum Ändern des Maßstabs der Zeichenfläche (Zoom)
Solve()	Löst alle internen und externen Constraints der Objekte in der Zeichenfläche
ToolPanelOnOff()	Macht die Werkzeugleiste sichtbar oder unsichtbar
Undo()	Macht ausgeführte Aktionen rückgängig
Zoom1To1()	Setzt den Zoom-Faktor der Zeichenfläche auf 1
ZoomAll()	Setzt den Zoom-Faktor der Zeichenfläche auf einen optimalen Wert (alle Objekte sind sichtbar)
ZoomMinus()	Erniedrigt den Zoom-Faktor der Zeichenfläche
ZoomPlus()	Erhöht den Zoom-Faktor der Zeichenfläche

Tabelle 8.4: Eintrittsmethoden in der Klasse ConnectionEditor

Über die verschiedenen Bearbeitungsmodi der Zeichenfläche beeinflußt der Verbindungs-Editor die Arbeitsweise ihrer Kontrollstrukturen. Es wurde eine Klasse ConnectionDrawArea von der Klasse AbstractDrawArea abgeleitet – hauptsächlich, um die drei Alphabet-Listener zu implementieren. Dadurch werden Änderungen des Alphabets direkt in der Zeichenfläche sichtbar gemacht.

8.2.4 Arbeitsmodi

Auch innerhalb der Klasse `ConnectionEditor` existieren Arbeitsmodi, die bereits aus der Klasse `SymbolEditor` bekannt sind. Aufgrund des geringeren Funktionsangebots gibt es jedoch nur die drei Modi `CHOICE_MODE`, `ZOOM_MODE` und `CHOICE_ALL_MODE`, die jeweils auf die Bearbeitungsmodi der Zeichenfläche abgebildet werden (vgl. Tabelle 8.3). Der aktuelle Arbeitsmodus wird in dem Attribut `mode` gespeichert und kann mit der Methode `getMode()` abgefragt werden.

8.2.5 Externe Operationen

Source- und Target-Objekt: Eine neue externe Operation wird über eine Funktion der Menü- bzw. Symbolleiste erzeugt. Über ein Dialogfenster, das durch die Klasse `SourceTargetDialog` des Packages `genged.gui` zur Verfügung gestellt wird, müssen jeweils ein Template für Source- und Target-Objekt der Operation ausgewählt werden. Nachdem die externe Operation mit den Template-Objekten erzeugt wurde, leitet die Methode `updateOperation()` zwei temporäre Instanzen für Source und Target ab und fügt sie dem Alphabet und der Operation hinzu. Für die Operation wird dabei der Temporary-Instance-Modus aktiviert, der das Template einer Operation zur Arbeit mit Instanzen befähigt. Über die Methode `optimizeArrangement()` wird die Position der Source- und Target-Instanzen verändert, so daß sie ordentlich in der Zeichenfläche angeordnet sind. Die Struktur des Source- und Target-Objekts wird unter dem Eintrag der Operation in der Strukturansicht dargestellt.

Innerhalb des Verbindungs-Editors kann immer nur *eine* Operation gleichzeitig bearbeitet werden. Wird in der Strukturansicht eine Operation markiert, dann wird im Verbindungs-Editor die Methode `valueChanged()` aufgerufen. Diese aktiviert die gewählte Operation unter Nutzung der Methode `activateOperation()`. Dazu werden die Alphabet-Objekte der „alten" Operation von der Zeichenfläche abgemeldet und die Objekte der „neuen" Operation angemeldet. Durch Aufruf `getActiveOperation()` wird die aktuelle Operation zurückgegeben, die in dem Attribut `operation` gespeichert wird.

Externe Constraints: Zur Definition externer Constraints zwischen den Primitiven von Source- und Target-Objekt wird das gleiche Dialogfenster benutzt wie im Symbol-Editor. Dazu existiert eine Instanz der Klasse `ConstraintDialog` aus dem Package `genged.gui`, die in dem Attribut `conDialog` gespeichert wird. Mit der Methode `initAndShowConstraintDialog()` der Klasse `ConnectionEditor` wird das nicht-modale Dialogfenster mit einem externen Constraint initialisiert. Die Methode `isConstraintDialogActive()` gibt den Wert `true` zurück, wenn es sichtbar ist. Auch die anderen Funktionalitäten betreffs der Constraints entsprechen im wesentlichen denen des Symbol-Editors. Der einzige Unterschied ist, daß die externen Constraints wie auch die externe Operation selbst im Temorary-Instance-Modus arbeiten. Alle nicht-numerischen Parameter der Constraints werden durch Instanzen belegt, die in dem Constraint auf ihre Templates zurückgeführt werden.

Änderungsinformationen: Eine sehr wichtige Rolle spielt im Verbindungs-Editor der Mechanismus, der über Änderungen des Alphabets informiert. Da Symbol- und Verbindungs-Editor auf dasselbe Alphabet zugreifen und es verändern, muß in allen Ansichten beider Editoren Konsistenz sichergestellt werden. Werden mit Hilfe des Verbindungs-Editors dem Alphabet Operationen hinzugefügt, wird der Symbol-Editor davon nicht tangiert. Für seine Ansichten sind externe Operationen uninteressant. Änderungen im Symbol-Editor wirken sich jedoch direkt auf den Verbindungs-Editor aus. Wenn Symbole oder Datentypen betroffen sind, die sich an externen Operationen beteiligen, müssen die im Verbindungs-Editor abgeleiteten Instanzen aktualisiert werden. Tritt eine Änderung eines Symbols oder Datentyps auf, wird im Verbindungs-Editor eine Methode der drei verschiedenen Alphabet-Listener aufgerufen. Diese sorgt dafür, daß mit Hilfe der Methode `setOperationInvalid()` alle betreffenden Operationen als ungültig erklärt werden. Mit der Methode `updateOperations()` werden für alle ungültigen Operation die Methoden `cleanOperation()` und `updateOperation()` aufgerufen, die von Änderungen betroffene temporäre Instanzen aus dem Alphabet löschen und neue Instanzen von den veränderten Alphabet-Objekten ableiten.

8.2.6 Verwendung des Constraint-Solvers

Der Verbindungs-Editor besitzt wie der Symbol-Editor einen „eigenen“ Constraint-Solver, dessen Referenz in dem Attribut `solver` gespeichert wird. Er kann im Fehlerfall mit der Methode `restartConstraintSolver()` neu gestartet werden.

An dem Constraint-Solver sind nur die Bestandteile der gerade aktiven externen Operation angemeldet. Dazu gehören alle Primitiven sowie die internen Constraints des Source- und Target-Objekts und alle externen Constraints, die innerhalb der Operation definiert wurden. Um alle diese Objekte anzumelden, wird von der Methode `activateOperation()` die Methode `registerOperation()` aufgerufen. Sie benutzt die `activate()`-Methoden der Alphabet-Objekte, der internen Operationen sowie der aktuellen externen Operation, um die Objekte bei dem Constraint-Solver anzumelden.

Die Methode `unregisterOperation()` wird benutzt, um eine Operation mit all ihren Objekten wieder abzumelden. Über die `deactivate()`-Methoden werden alle Constraints und Objekte aus dem Constraint-Solver entfernt.

8.2.7 Beispiel: Nassi-Shneiderman-Diagramme

Nachdem im Symbol-Editor einige der Symbole und Datentypen von Nassi-Shneiderman-Diagrammen erstellt wurden, sollen nun beispielhaft einige Operationen definiert werden.

Die Abbildung 8.10 zeigt einen Screenshot von dem Arbeitsfenster des Alphabet-Editors. In den Abbildungen 8.11 bis 8.15 auf den folgenden Seiten wird jeweils ein Teil der Strukturansicht und die erstellte Operation dargestellt. Wir haben folgende Elemente der Nassi-Shneiderman-Diagramme ins Alphabet aufgenommen:

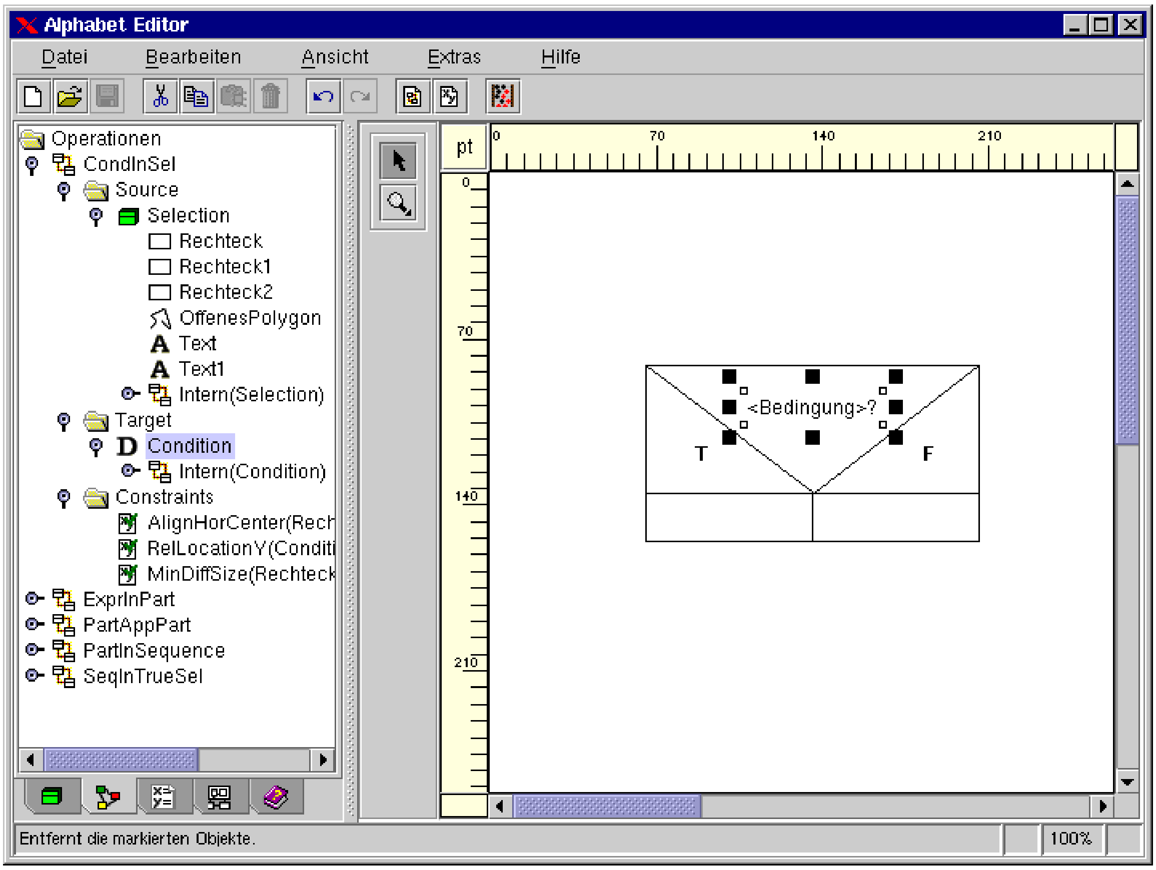


Abbildung 8.10: Screenshot: Arbeitsfenster mit aktiviertem Verbindungs-Editor

- Einfügen eines Ausdrucks in einen Programmteil (`ExprInPart`)
- Anhängen eines Programmteils an einen anderen (`PartAppPart`)
- Einfügen eines Programmteils in eine Abfolge (`PartInSeq`))
- Einfügen einer Bedingung in eine Fallunterscheidung (IF THEN) (`CondInSel`)
- Einfügen einer Abfolge in den True-Teil einer Fallunterscheidung (`SeqInTrueSel`)

Zu beachten ist, daß auch in den externen Constraint-Definitionen Positionen, Breiten, Höhen und Abstände mit dem Faktor 100 multipliziert wurden, da intern mit der Einheit 1/100pt gerechnet wird.

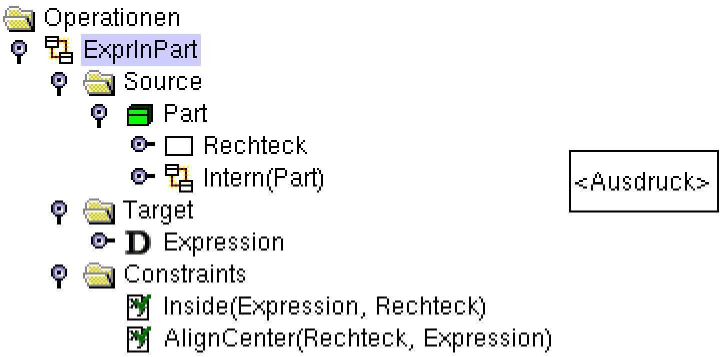


Abbildung 8.11: Nassi-Shneiderman-Diagramme: Operation `ExprInPart`

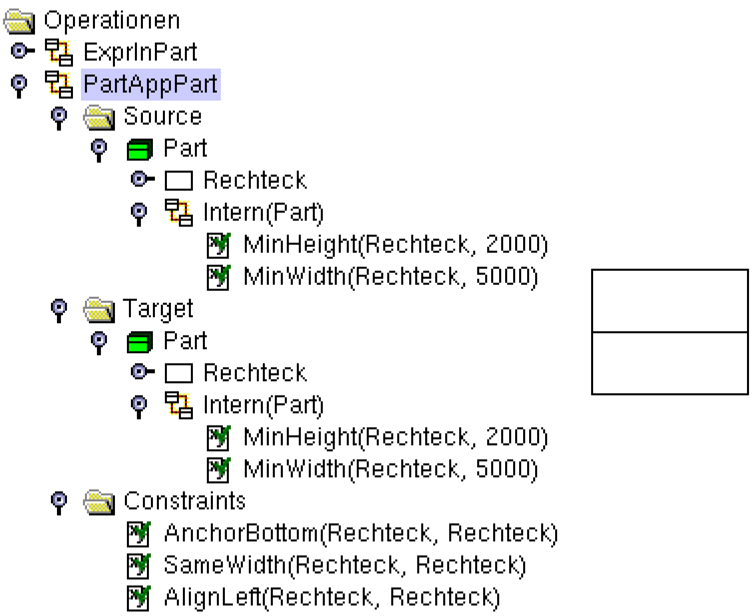


Abbildung 8.12: Nassi-Shneiderman-Diagramme: Operation `PartAppPart`

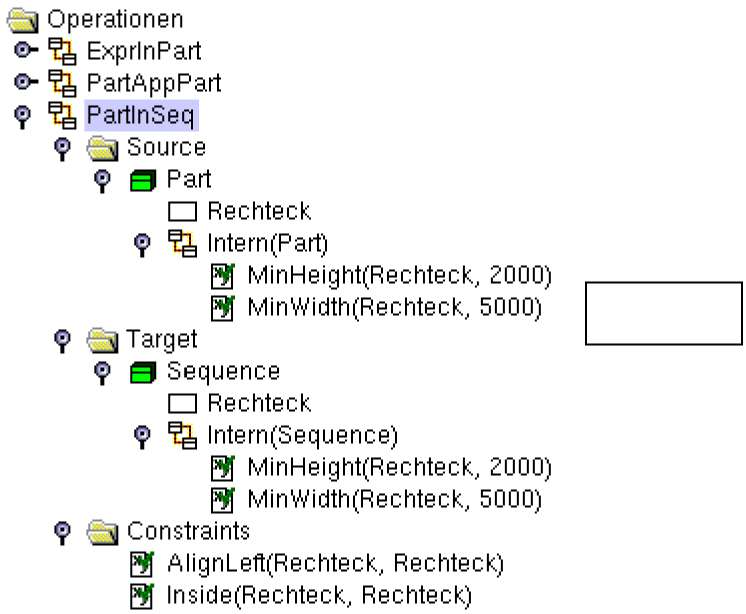


Abbildung 8.13: Nassi-Shneiderman-Diagramme: Operation `PartInSeq`

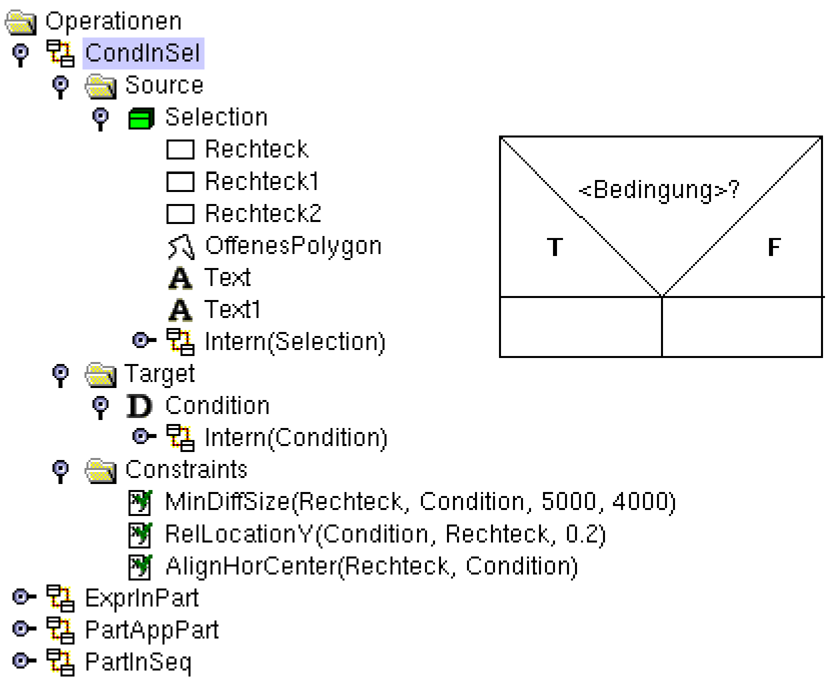


Abbildung 8.14: Nassi-Shneiderman-Diagramme: Operation `CondInSel`

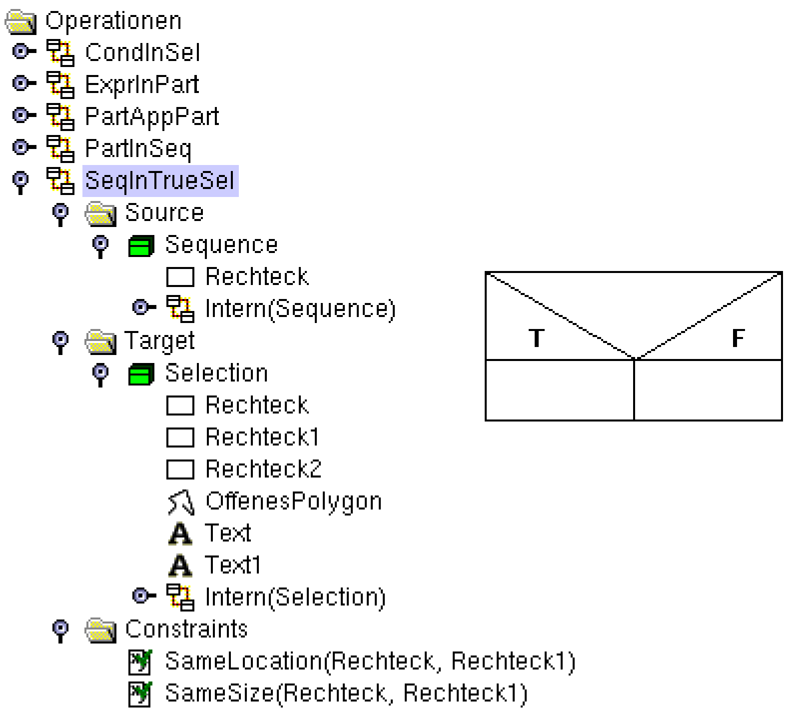


Abbildung 8.15: Nassi-Shneiderman-Diagramme: Operation `SeqInTrueSel`

Kapitel 9

Test-System

In diesem Kapitel wird ein Test-System zum Zeichnen von Diagrammen unter Benutzung eines visuellen Alphabets vorgestellt. Dabei handelt es sich um einen separaten Editor, der nur optisch in den Alphabet-Editor integriert wurde. Charakteristisch ist, daß er ein vom Alphabet-Editor unabhängiges Datenmodell besitzt. Das Test-System befindet sich in dem Package `genged.alphabeteditor.testsystem` und besteht aus drei grundlegenden Klassen: `TestSystem`, `TestSystemDrawArea` und `TestSystemModel`. Im folgenden werden wir kurz auf das Modell und auf die Ansichten und Kontrollstrukuren des Test-Systems eingehen.

9.1 Modell des Diagramms

Die Klasse `DiagramAlgebra`, die sich im Package `genged.language` befindet, erweitert die bereits bekannte Klasse `Alphabet`, um die Bestandteile eines Diagramms zu verwalten. Ein Alphabet dient dazu, Templates bereitzustellen, von denen beliebig viele Instanzen abgeleitet werden können. Innerhalb der Klasse `Alphabet` gibt es daher – abgesehen von temporären Instanzen zur Erstellung von externen Operationen – keine Instanzen. Der Sinn der Klasse `DiagramAlgebra` ist, Instanzen aller Bestandteile eines Alphabets zu verwalten, die zusammengenommen ein Diagramm bilden. Daher können alle Strukturen, die ein Alphabet für Templates zur Verfügung stellt, ebenso für Instanzen genutzt werden. Zusätzlich wird ein Attribut `alphabet` eingefügt, welches das Alphabet speichert, von dem die einzelnen Instanzen abgeleitet werden. Mit den Methoden `setNewAlphabetAlphabet()` und `getAlphabetAlphabet()` kann das Alphabet gesetzt und abgefragt werden.

Das Attribut `alphabet` wird nicht mit in einer Diagramm-Datei gespeichert, sondern nur über einen Dateinamen referenziert. Während des Öffnens einer Diagramm-Datei wird zuerst versucht, das Alphabet in den Speicher zu laden.

In der Abbildung 9.1 ist das Klassendiagramm der Klasse `DiagramAlgebra` dargestellt. Diese Klasse verwaltet die Daten eines Diagramms.

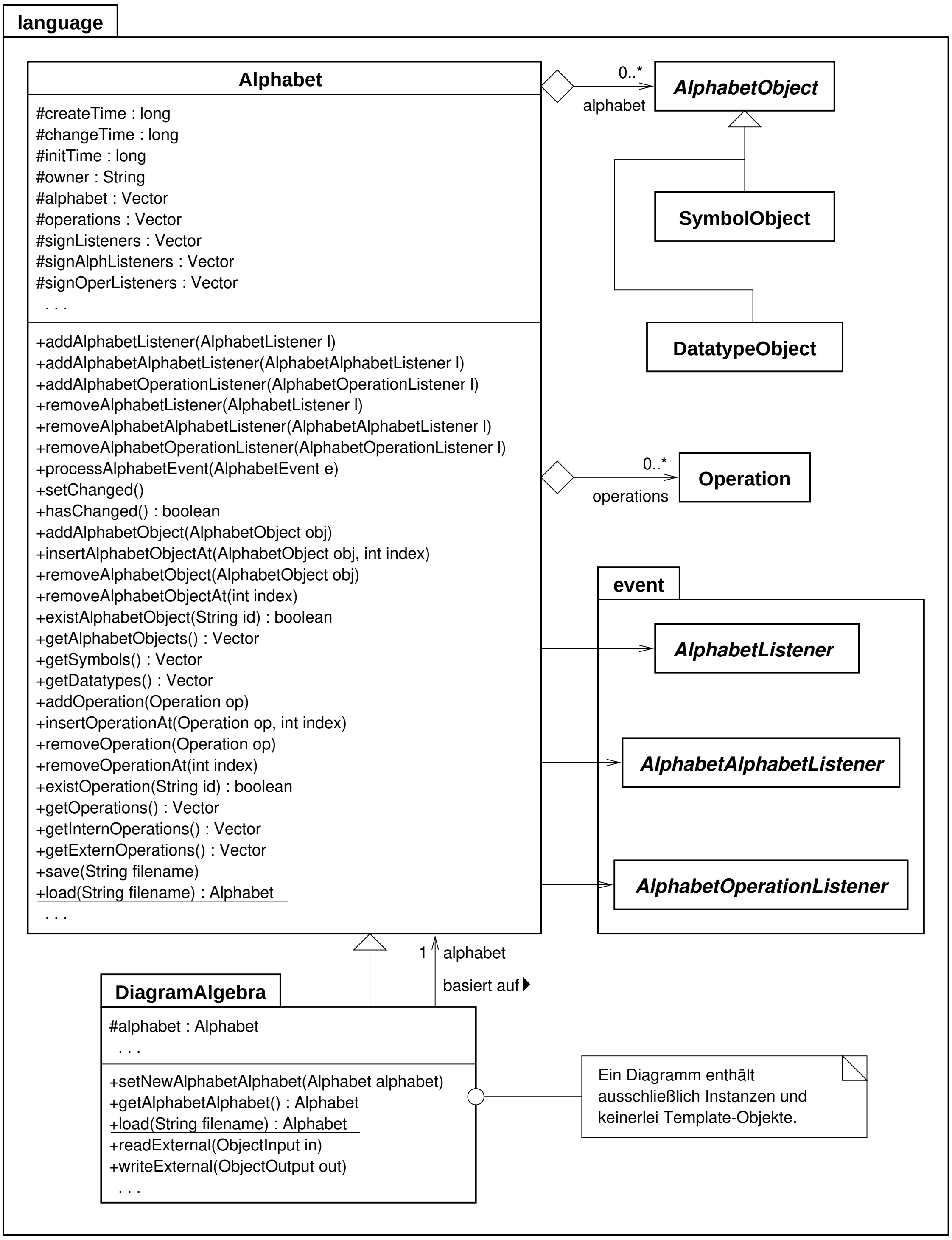


Abbildung 9.1: Klassendiagramm: `DiagramAlgebra`

9.2 Struktur des Test-Systems

Der Aufbau des Test-Systems entspricht im wesentlichen dem Aufbau der bereits vorgestellten Editoren. Die Klasse `TestSystem` enthält die Kernfunktionalität der Kontrollstruktur. In der Klasse `TestSystemModel` wird das Model der Strukturansicht implementiert. Als Modell für Selektierungen innerhalb der Strukturansicht benutzen wir die Klasse `DefaultTreeSelectionModel` aus dem Package `javax.swing`. Mit der Klasse `TestSystemDrawArea`, die von der Klasse `AbstractDrawArea` im Package `genged.gui` abgeleitet ist, wird eine Zeichenfläche zum Zeichnen von Diagrammen bereitgestellt. Die Abbildung 9.2 zeigt Attribute und Methoden der verschiedenen Klassen und ihr Zusammenwirken.

9.3 Ansichten zu Alphabet und Diagramm

Strukturansicht: Die Strukturansicht dient diesmal nicht als Überblick über die erstellten Bestandteile des Diagramms. Sie enthält stattdessen die Symbole, Datentypen und externe Operationen des Alphabets, auf dem das Diagramm basiert. Die Klasse `SystemTestModel`, die als Adapterklasse zwischen Alphabet und Baumstruktur fungiert, enthält nur wenige Methoden, da keine komplexen Strukturen darzustellen sind. Um eine Instanz von einem Symbol, einem Datentypen oder einer Operation zu erstellen, muß das Template in der Strukturansicht markiert werden und über das Menü bzw. die Symbolleiste eine entsprechende Benutzeraktion ausgelöst werden. Von Constraints abstrahieren wir; sie werden automatisch instanziiert, wenn von einem Alphabet-Objekt oder einer externen Operation eine Instanz gebildet wird.

Zeichenfläche: Die Zeichenfläche, die durch die Klasse `TestSystemDrawArea` repräsentiert wird, enthält alle Instanzen der Symbole und Datentypen, die zusammen ein Diagramm bilden. Die gewohnten Funktionalitäten stehen auch im Test-System zur Verfügung.

9.4 Kontrollstrukturen

9.4.1 Steuerung von Aktionen

Benutzeraktionen: Die Anzahl der Benutzeraktionen ist vergleichbar mit dem Verbindungs-Editor. In der Tabelle 9.1 sind alle Eintrittsmethoden aufgelistet, die in der Klasse `TestSystem` implementiert wurden.

Arbeitsmodi: Neben den Benutzeraktionen, die über das Menü bzw. die Symbolleiste angeboten werden, können mit Hilfe der Zeichenfläche oder der Strukturansicht weitere Aktionen ausgeführt werden. Dazu existieren wie im Symbol- und Verbindungs-Editor verschiedene Arbeitsmodi, die den internen Zustand des Test-Systems bestimmen. Der aktuelle Arbeitsmodus ist in dem Attribut `mode` gespei-

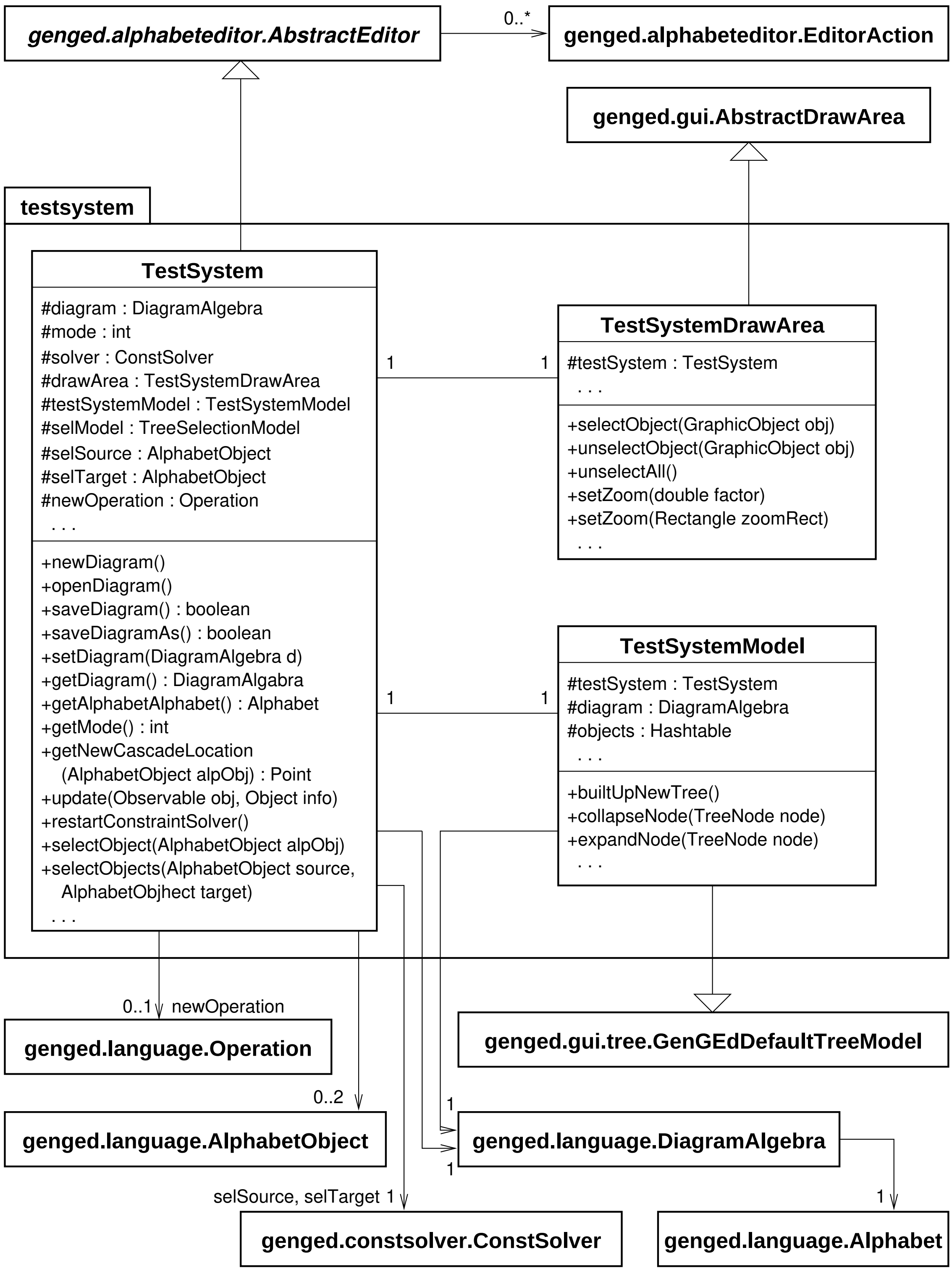


Abbildung 9.2: Klassendiagramm: Package `genged.alphabeteditor.testsystem`

Eintrittsmethode	Beschreibung der Funktionalität
`AutoSolveOnOff()`	Schaltet den Modus zum automatischen Lösen von Constraints ein und aus
`ChangeObjectPosition()`	Ändert die Zeichenreihenfolge von Diagramm-Objekten
`ChoiceMode()`	Aktiviert den Arbeitsmodus zum Auswählen, Verschieben, Skalieren, Rotieren und Verzerren von Objekten
`CloseFile()`	Speichert das Diagramm und schließt es
`Copy()`	Kopiert markierte Objekte in die Zwischenablage (nicht implementiert)
`Cut()`	Kopiert markierte Objekte in die Zwischenablage und entfernt sie aus dem Test-System (nicht implement.)
`GridOnOff()`	Schaltet das Raster der Zeichenfläche ein und aus
`NewDatatype()`	Erstellt eine neue Instanz eines Datentypen und fügt sie in das Diagramm ein
`NewFile()`	Erstellt ein neues leeres Diagramm auf der Grundlage eines Alphabets, das in einer Datei vorliegen muß
`NewOperation()`	Erstellt eine neue Instanz einer Operation und fügt sie in das Diagramm ein
`NewSymbol()`	Erstellt eine neue Instanz eines Symbols und fügt sie in das Diagramm ein
`OpenFile()`	Öffnet eine Diagramm-Datei von einem Datenträger
`Paste()`	Kopiert Objekte der Zwischenablage in das Test-System (nicht implementiert)
`Redo()`	Führt rückgängig gemachte Aktionen erneut aus
`Remove()`	Entfernt alle in der Zeichenfläche markierten Objekte aus dem Diagramm
`RemoveOperation()`	Entfernt eine Operation aus dem Diagramm
`RestartParCon()`	Neustarten des Constraint-Solvers PARCON
`ReticuleOnOff()`	Schaltet das Fadenkreuz der Zeichenfläche ein/aus
`SaveFile()`	Speichert das Diagramm in eine Datei
`ShowInvisibleOnOff()`	Schaltet die Darstellung von unsichtbaren Objekten in der Zeichenfläche ein und aus
`ShowZoomGroup()`	Zeigt die Werkzeugauswahl zum Ändern des Maßstabs der Zeichenfläche (Zoom)
`Solve()`	Löst alle internen und externen Constraints der Objekte in der Zeichenfläche
`ToolPanelOnOff()`	Macht die Werkzeugleiste sichtbar oder unsichtbar
`Undo()`	Macht ausgeführte Aktionen rückgängig
`Zoom1To1()`	Setzt den Zoom-Faktor der Zeichenfläche auf 1
`ZoomAll()`	Setzt den Zoom-Faktor der Zeichenfläche auf einen optimalen Wert (alle Objekte sind sichtbar)
`ZoomMinus()`	Erniedrigt den Zoom-Faktor der Zeichenfläche
`ZoomPlus()`	Erhöht den Zoom-Faktor der Zeichenfläche

Tabelle 9.1: Eintrittsmethoden in der Klasse `TestSystem`

chert und kann mit der Methode `getMode()` abgefragt werden. Es existieren diesmal nur die Arbeitsmodi `CHOICE_MODE` und `ZOOM_MODE`.

9.4.2 Erstellen der Bestandteile des Diagramms

Symbole und Datentypen: Nachdem die Instanz von dem Template abgeleitet wurde, kann sie dem Diagramm hinzugefügt werden. Der gesamte Prozeß wird durch ein Befehlsobjekt – der Klasse `InsertAlphabetObjectInstance` im Package `genged.language.commands` – gekapselt und ausgeführt. Dieses Befehlsobjekt sorgt auch dafür, daß die interne Operation des Alphabet-Objekts mit instanziiert und ebenfalls dem Diagramm hinzugefügt wird. Das Test-System selbst sorgt durch Aufrufen der Methode `getNewCascadeLocation()` dafür, daß eine neue Position in der Zeichenfläche bestimmt wird. Die Instanz des Alphabet-Objekts wird an diese Position verschoben.

Externe Operationen: Das Instanziieren einer externen Operation gestaltet sich etwas komplizierter. Nachdem die Operation ausgewählt wurde, müssen Source- und Target-Objekt durch einen Doppelklick in die Zeichenfläche ausgewählt werden. Während dieses Prozesses werden nacheinander die Attribute `newOperation`, `selSource` und `selTarget` mit Inhalt gefüllt. Wird zwischendurch eine andere Benutzeraktion ausgeführt, werden alle drei Attribute durch die Methode `cancelActivities()` auf `null` zurückgesetzt. Hat der Benutzer das Target-Objekt festgelegt, wird in der Maus-Listener-Methode `mouseClicked()` das Befehlsobjekt `InsertOperationInstance` erzeugt, das die Operation instanziiert und dem Diagramm hinzufügt. Danach werden die drei Attribute, die die neue Operation beschrieben haben, wieder auf `null` gesetzt.

9.4.3 Verwendung des Constraint-Solvers

Das Test-System enthält in dem Attribut `solver` eine Referenz auf einen Constraint-Solver. Objekte und Constraints werden im Test-System ausschließlich durch Methoden der Alphabet-Listener im Constraint-Solver an- und abgemeldet. Wird beispielsweise dem Diagramm eine neue Symbol-Instanz hinzugefügt, ruft das Diagramm im Test-System die Methode `symbolInserted()` des Interfaces `AlphabetAlphabetListener` auf. In dieser Methode wird das eingefügte Symbol mit seiner Methode `activate()` am Constraint-Solver angemeldet.

9.4.4 Laden und Speichern eines Diagramms

Zum Laden und Speichern eines Diagramms existieren im Test-System eigene Methoden, die infolge der Unabhängigkeit vom Alphabet-Editor implementiert werden mußten. Die Methode `openDiagram()` öffnet ein Diagramm von einem Datenträger. Der Name kann aus einem Dialogfenster gewählt werden. Über die Methoden `saveDiagram()` und `saveDiagramAs()` wird ein Diagramm gesichert. Bei der zuletzt genannten Methode wird zuvor ein Dialogfenster aufgerufen, um einen Dateinamen einzugeben.

9.5 Beispiel: Nassi-Shneiderman-Diagramme

Nachdem wir einige Bestandteile der Nassi-Shneiderman-Diagramme durch Symbole, Datentypen und Operationen definiert haben, können wir mit Hilfe des Test-Systems die Korrektheit und Vollständigkeit der grafischen Verbindungen testen. Die Abbildung 9.3 zeigt einen Screenshot des Alphabet-Editors mit aktiviertem Test-System. Es wurde eine Fallunterscheidung modelliert, die eine Bedingung und drei verschiedene Programmteile enthält.

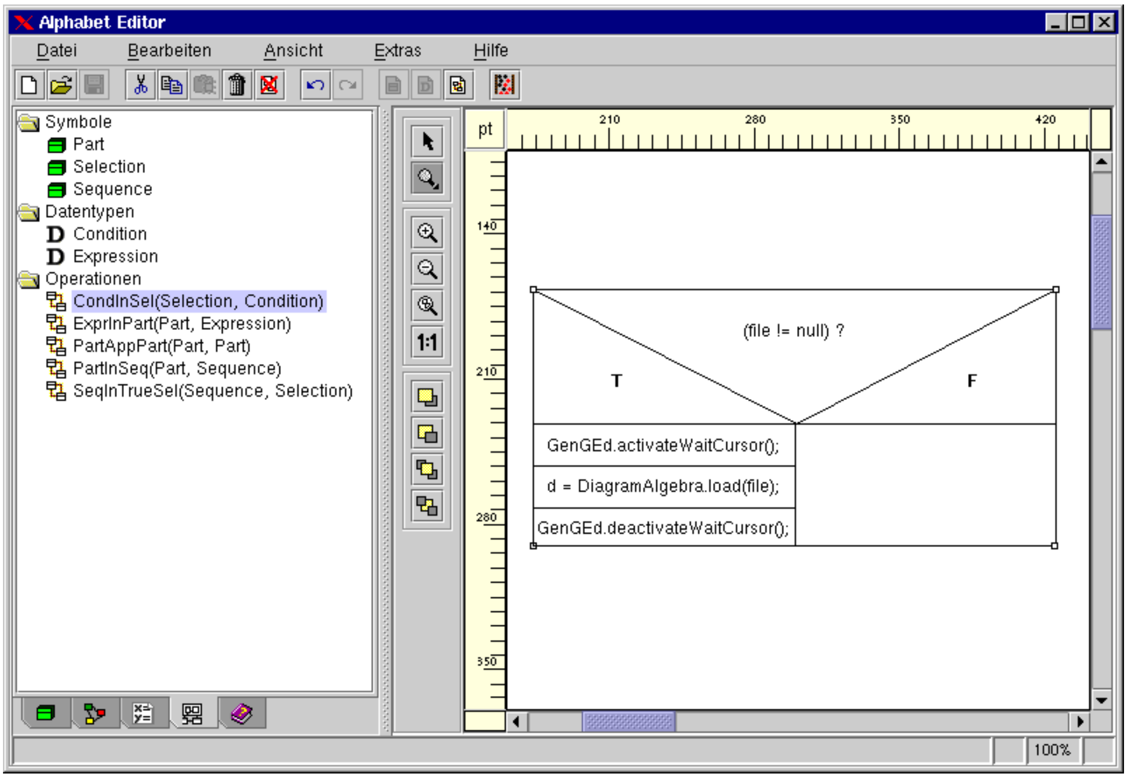


Abbildung 9.3: Screenshot: Arbeitsfenster mit aktiviertem Test-System

Kapitel 10

Zusammenfassung und Ausblick

Zum Abschluß dieser Diplomarbeit wollen wir die in dieser Arbeit behandelten Themen nochmals kurz aufgreifen und einen Blick aus dem Fenster des Alphabet-Editors hinaus werfen. Wie wird es weitergehen? Was kann verbessert werden? Wir unterscheiden hier zwischen Komponenten, die möglicherweise einmal den Alphabet-Editor erweitern werden, und Verbesserungen in eigener Sache.

10.1 Zusammenfassung

In dieser Arbeit wurde die Konzeption und Entwicklung eines Alphabet-Editors für visuelle Sprachen beschrieben. Nachdem der Begriff der „visuellen Sprache" erläutert wurde, setzten wir uns mit den Anforderungen eines Alphabets auseinander. Konzeptionell besteht ein visuelles Alphabet aus Symbolen und Verbindungen zwischen diesen Symbolen. Die Symbole repräsentieren Grafiken oder Datentypen. Verbindungen bestehen aus einstelligen Operationen zwischen Symbolnamen und Constraints zwischen den Symbolgrafiken. Constraints beeinflussen das Layout der Grafiken, indem Zusammenhänge zwischen Eigenschaften wie beispielsweise Position und Größe beschrieben werden. Die verschiedenen Details wurden durch die Betrachtung am Beispiel der Nassi-Shneiderman-Diagramme herausgearbeitet und demonstriert. Ausgehend von diesen Anforderungen untersuchten wir mögliche Funktionalitäten eines Alphabet-Editors. Mit ihm sollten auf benutzerfreundliche Art und Weise die einzelnen Bestandteile eines Alphabets definiert werden können.

Um die entwickelten Konzepte umzusetzen, bedienten wir uns verschiedener Hilfsmittel und Werkzeuge. Zur Modellierung des Software-Systems wurden Auszüge der Unified Modeling Language eingesetzt: Klassen- und Sequenzdiagramme. Für den Entwurf wurden verschiedene Design Patterns benutzt und an unsere Bedürfnisse angepaßt. Implementiert wurde der Alphabet-Editor schließlich in der Programmiersprache Java. Da unser System constraint-basiert ist, wurde zum Lösen der grafischen Constraints ein externer Constraint-Solver namens PARCON eingesetzt. Wir haben in kurzen Zügen umrissen, wie er funktioniert und wie wir mit ihm kommunizieren.

In dem restlichen größten Teil dieser Diplomarbeit wurde der Entwurf und die Implementierung des Alphabet-Editors erläutert. Wir stellten zu Beginn die Systemstruktur vor und stellten die Unterteilung des Gesamtsystems in verschiedene Editoren vor – einen Symbol-Editor, einen Verbindungs-Editor und ein Test-System. Danach beschäftigten wir uns ausführlich mit den Datenstrukturen des Alphabets. Die Anforderungen wurden in konkrete Klassen der Programmiersprache Java umgesetzt. Wir setzten uns in diesem Zusammenhang mit primitiven Grafiken, mit Gruppen von Grafiken, mit Symbolen und Datentypen sowie mit verschiedenen Arten von Operationen und Constraints auseinander. In dem Alphabet-Editor finden sich viele allgemeine Mechanismen wie beispielsweise zur Internationalisierung oder zur Benutzung von Zeichenflächen und Strukturansichten. Auch die Konzepte der Benutzeraktionen und des Undo-Mechanismus wurden vorgestellt.

Ein weiterer Schwerpunkt dieser Diplomarbeit lag auf der Entwicklung einer High-Level-Constraint-Sprache, um mehrere Constraints von PARCON zu einem typisierten und parametrisierten High-Level-Constraint zusammenzufassen. In diesem Zusammenhang wurde auch die Anbindung unseres Systems an den Constraint-Solver PARCON beschrieben.

Die letzten beiden Kapitel beschäftigten sich mit den konkreten Editoren, dem Symbol- und Verbindungs-Editor, sowie mit dem Test-System, das es ermöglicht, rein constraint-basierte Diagramme auf der Basis eines Alphabets zu zeichnen. Um zu zeigen, daß der Alphabet-Editor die Anforderungen aus den ersten Kapiteln dieser Arbeit erfüllt, benutzten wir wieder Nassi-Shneiderman-Diagramme. Verschiedene Symbole und Operationen wurden als Beispiele definiert, um daraus schließlich ein kleines konkretes Diagramm zu zeichnen.

Die Benutzung des Alphabet-Editors wird ausführlich in einem Benutzerhandbuch beschrieben, das sich im Anhang dieser Diplomarbeit befindet. Es beschreibt an verschiedenen Beispielen die Vorgehensweise zur erfolgreichen Definition eines visuellen Alphabets.

10.2 Erweiterungen

Das gesamte Projekt GENGED hatte zum Ziel, eine visuelle Sprache zu erstellen, mit der in einem Grafik-Editor Diagramme erstellt werden können. In dem Alphabet-Editor wird nur das Alphabet der visuellen Sprache generiert. Aufbauend auf den Datenstrukturen des Alphabets wurde zeitgleich zu dieser Arbeit der Grammatik- und Grafik-Editor fertiggestellt [Nie99]. So ist es bereits möglich, komplexe Grammatik-Regeln auf dem Alphabet zu definieren und dadurch eine konkrete Syntax für eine Modellierungs- bzw. Diagrammtechnik festzulegen.

Die nächsten Schritte werden zeigen, ob unser System den Anforderungen gewachsen ist. Es könnten beispielsweise verschiedene visuelle Sprachen wie UML, OMT, Petrinetze oder Statecharts vollständig definiert werden. Dabei wird man ziemlich schnell die Grenzen von GENGED zu spüren bekommen. So faßt die Unified Modeling Language verschiedenste Diagrammtechniken unter einem Dach zusammen. Es wird also überhaupt nicht möglich sein, die gesamte UML als *eine* visuelle Sprache einzugeben. Stattdessen müßten mehrere Alphabete erstellt werden.

Zwischen verschiedenen Alphabeten gibt es zur Zeit jedoch noch keine Verknüpfungsmöglichkeiten. Um erfolgreich mit UML zu modellieren, müßten Verbindungen beispielsweise zwischen Klassen-, Objekt- und Sequenzdiagrammen bestehen. Die verschiedenen Objekte, die in den Diagrammen dargestellt werden, sind oft Abbilder gleicher realer Objekte. Gelingt es irgendwann, dieses Problem zu lösen, könnten wir uns als Endziel diverser Weiterentwicklungen das Generieren von Programm-Code-Fragmenten vorstellen, wie es bereits Modellierungsprogramme tun, die auf einer konkreten Technik basieren. Als Beispiel wäre hier Rational Rose zu erwähnen.

10.3 Verbesserungen

Eine alte Informatiker-Weisheit besagt, daß ein Programm niemals fertig ist. Hier und da werden gerade bei größeren Systemen immer wieder kleine Fehler zu Tage treten, Verbesserungen gewünscht und neue Anforderungen gestellt werden. Auch der Alphabet-Editor ist noch lange nicht perfekt – es gibt noch zahlreiche Möglichkeiten, ihn benutzerfreundlicher zu gestalten und weitere Optimierungen vorzunehmen.

Fehlende Funktionen im Symbol-Editor: Im Symbol-Editor fehlen noch einige Funktionalitäten, die zum einen aus zeitlichen Gründen, zum anderen aber auch aus programmiertechnischen Gründen[1] nicht umgesetzt werden konnten. Folgendes sollte noch implementiert werden:

- Linienstile für Linien und Umrisse, z.B. gepunktet oder gestrichelt
- Liniendicke für Linien und Umrisse
- Füllmuster für geschlossene Primitiven
- Farbverlaufsfüllung für geschlossene Primitiven
- Gleichmäßige geschlossene Polygone wie Drei-, Vier-, Fünf- und Sechsecke
- Bezierkurven
- Dynamische Linien, die vom Constraint-Solver beeinflußte Start- und Endpunkte besitzen, zusätzlich jedoch beliebig viele weitere Punkte enthalten, die auch später noch durch den Benutzer verändert werden können (Hinzufügen, Verschieben, Entfernen)
- Funktionen der Zwischenablage zum Ausschneiden, Kopieren und Einfügen von Objekten (Primitiven, Symbole, Datentypen, Constraints, Operationen)
- Drucken des gesamten Alphabets oder einzelner Alphabet-Bestandteile
- Drucken eines Diagramms innerhalb des Test-Systems
- Speichern der grafischen Benutzeroberfläche, um sie beim Starten des Alphabet-Editors laden zu können und dadurch die sehr lange Anlaufzeit zu verkürzen

[1] Für einige grafische Funktionalitäten benötigen wir neue Klassen des JDK 2.0 von Sun Microsystems. Da der Alphabet-Editor auf einem Linux-System entwickelt wurde, stand bis Februar 1999 keine derartige Java-Version zur Verfügung.

Algorithmen zur Behandlung von Alphabet-Änderungen: Ein Benutzer erstellt unter Anwendung eines bestimmten Alphabets ein Diagramm. Danach wird das zugrundeliegende Alphabet verändert. Was passiert mit dem Diagramm? Hier werden Algorithmen benötigt, die beim Öffnen eines Diagramms überprüfen, ob die verwendete Alphabetversion noch aktuell ist. Wenn inzwischen eine neuere Version vorliegt, muß für alle abgespeicherten Instanzen des Diagramms überprüft werden, ob das dazugehörige Template noch existiert. Dabei sollten auch die systemweit eindeutigen Namen berücksichtigt werden. Existiert zu einer Instanz kein Template, muß sie entfernt werden. Wenn zwar ein Template den richtigen Namen, jedoch die „falschen" Primitiven besitzt, sollte versucht werden, die Instanz durch eine neue Instanz zu ersetzen, die von dem aktuellen Template abgeleitet wird. Dasselbe gilt auch für Operationen und Constraints. Da in den Diagramm-Dateien ausschließlich Instanzen und deren Template-Namen gespeichert werden, könnte bei der Verwendung absolut gleicher Namen sogar ein Alphabet ausgetauscht werden. Eine andere Möglichkeit wäre es, einen Mechanismus zu entwerfen, mit dem der Benutzer ein Diagramm eines Alphabets regelrecht in ein Diagramm eines anderen Alphabets umwandeln kann.

Constraint-Editor: Die Klassen für den Constraint-Editor existieren bereits im Package `genged.alphabeteditor.consteditor`. Sie sind jedoch noch leer. Geplant ist, einen integrierten Text-Editor zum Verändern der Constraint-Datenbank anzubieten, der zusätzlich über eine Strukturansicht verfügt. In dieser Ansicht könnten alle Constraints alphabetisch geordnet aufgeführt werden. Per Mausklick auf einen Eintrag wird der Constraint auf der Arbeitsfläche sichtbar. Außerdem sollte der Editor Text-Highlighting unterstützen, um Schlüsselworte, Bezeichner oder andere Bestandteile der High-Level-Constraint-Sprache hervorzuheben. Denkbar wäre außerdem, einen Assistenten anzubieten, der den Benutzer beim Erstellen von Constraints unterstützt, indem bereits vorgefertiger „Code" erzeugt wird, den der Assistent mit Hilfe verschiedener Abfragen über mehrere Dialogfenster vervollständigt. Die Constraint-Datenbank sollte zur Laufzeit eingelesen und auf Fehler überprüft werden können.

Online-Hilfe-System: Die unfertigen Klasse des Online-Hilfe-Systems befinden sich im Package `genged.alphabeteditor.helpsystem`. Das Benutzerhandbuch, das sich im Anhang dieser Diplomarbeit befindet, sollte als HTML-Variante in den Alphabet-Editor integriert und um Links erweitert werden. Außerdem könnten verschiedene Suchfunktionen, z.B. Stichwort- und Volltextsuche, sowie eine History-Funktion angeboten werden. Die dazugehörige Strukturansicht könnte das Inhaltsverzeichnis darstellen, über das man auf schnelle Art und Weise zu der gewünschten Seite gelangen kann. Die Swing-Klassen von Java unterstützen die Darstellung von HTML.

Import- und Exportmöglichkeiten: Es wäre schön, wenn einzelne oder auch mehrere Symbole und Grafiken unabhängig von anderen Komponenten des Alphabets importiert oder exportiert werden können. Dadurch könnten gleiche Grafiken in mehreren Alphabeten auftreten, ohne sie jedes Mal neu zu zeichnen.

Datei-Import- und Export-Filter: Um grafische Daten mit anderen vektororientierten Zeichenprogrammen wie CorelDraw oder XFig auszutauschen, könnten verschiedene Filter zum Import und Export von Dateien zur Verfügung gestellt werden. Seit 1987 gibt es einen Standard für ein Dateiformat zum Archivieren grafischer Informationen in einem Metafile. Es handelt sich hierbei um den Standard CGM (Computer Graphics Metafile) [ISO87], der in ISO8632 zertifiziert wurde. Die meisten auf dem Markt erhältlichen Zeichenprogramme unterstützen dieses Format.

Fehlererkennung in der High-Level-Constraint-Sprache: Beim Auswerten der Constraint-Datenbank, die die Typ- und Constraint-Definitionen enthält, werden alle Syntax-Fehler ausnahmslos erkannt. Allerdings wird sofort beim Finden des ersten Fehlers das Einlesen der Datenbank abgebrochen. Eine sinnvolle Erweiterung wäre die mehrfache Fehlererkennung. Dazu müßte der Constraint-Parser aufgrund seiner „Grammatik“ wieder auf einer Constraint-Definition aufsetzen, nachdem er einen Fehler gefunden hat. Dadurch könnten (fast) alle Fehler auf einmal erkannt werden.

Alternativen zur Verwendung der bisherigen High-Level-Constraints: Die Erfahrung der Arbeit mit High-Level-Constraints hat gezeigt, daß man schnell den Überblick verliert. Inzwischen gibt es ungefähr 60 verschiedene Constraints, die in einer Liste zur Auswahl angeboten werden. Günstiger wäre es, alle Constraints durch Verwendung von Kategorien zu organisieren. Außerdem wurde darüber nachgedacht, jeden definierten High-Level-Constraint mit einem Symbol auszustatten. Eine ganz andere Alternative wäre folgende: Wir lassen neben den bereits vorhandenen Constraints sogenannte „freie“ Constraints zu, die erst zur Laufzeit definiert werden. Ein solcher Constraint dient nicht dazu, mehrfach verwendet zu werden, sondern bezieht sich auf eine konkrete Situation. Dazu werden die Formeln des Constraints in ein geeignetes Dialogfenster eingegeben, das einem Formeleditor ähnelt.

Constraint-Solving: Der Constraint-Solver in der jetzigen Variante ist ziemlich langsam. Es sollte untersucht werden, ob die Geschwindigkeitseinbußen an der Klasse `ConstSolver` und ihrer Objektverwaltung, an der Socketverbindung zu PARCON oder an PARCON selbst liegen. Für den ersten Fall könnte man versuchen, verschiedene Dinge zu optimieren. Auf die beiden anderen Fälle haben wir keinen direkten Einfluß.

Relative und absolute Verbindungspunkte: Zur Zeit können im Symbol-Editor ausschließlich Verbindungspunkte festgelegt werden, deren Position relativ zu der Position und Größe der dazugehörigen Primitive ist. Wird die Primitive skaliert, ändert der Verbindungspunkt ebenfalls seine Position. Bei der Arbeit mit den Editoren hat sich gezeigt, daß es sinnvoll wäre, auch absolut positionierbare Verbindungspunkte zuzulassen. Sie wären dann nur von der linken oberen Ecke der Bounding-Box einer Primitive abhängig, nicht jedoch von deren Breite oder Höhe. In diesem Fall würde der Verbindungspunkt bei einer Skalierung seine Position beibehalten.

Optimieren der Matrizenrechnung in Primitiven: Die Punkte einer Primitiven ergeben sich durch Umrechnungen mit Hilfe einer Transformationsmatrix. Um einen Bildpunkt einer Primitive zu erhalten, wird der Vektor des Originalpunktes mit der Transformationsmatrix multipliziert. Die Rückrechnung erfolgt mit der Inversen. Speziell bei Linienobjekten ist folgendes Problem aufgetreten: Durch die Skalierung einer Primitive wird die Transformationsmatrix geändert. Werden dagegen einzelne Knoten verschoben, z.B. Start- oder Endpunkt einer Linie, ändern sich dadurch direkt die Originalpunkte. Es sind Situationen aufgetreten, in denen entweder die Werte der Matrix oder die Originalpunkte ungültige Werte angenommen haben. Sie sind „übergelaufen“, obwohl der Typ `int` einen sehr großen Definitionsbereich besitzt. Da wir intern mit 1/100pt rechnen, ergeben sich mitunter ziemlich große Zahlen. Um solche Situationen zu verhindern, müßten Matrix und Originalpunkte über einen Algorithmus angeglichen werden.

Literaturverzeichnis

[AER96] M. Andries, G. Engels, and J. Rekers. How to represent a Visual Program ? In *[TVL96]*, 1996.

[AU92] A.V. Aho and J.D. Ullman. *Foundations of Computer Science.* Computer Science Press, New York, 1992.

[Bar97] R. Bardohl. GENGED – A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars. In *[VL'97]*, pages 48–55. 1997.

[BMR+98] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orientierte Software-Architektur.* Addison-Wesley, Bonn, Paris, u.a, 1998.

[Boo86] G. Booch. Object-oriented development. *IEEE Trans. on Software Engineering, vol. SE-12, no. 2*, pages 211–221, 1986.

[Boo91] G. Booch. *Object-oriented design with applications.* Benjamin Cummings, Redwood City, 1991.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 1994.

[BSMM93] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, Frankfurt am Main, 1993.

[BT97] R. Bardohl and G. Taentzer. Defining Visual Languages by Algebraic Specification Techniques and Graph Grammars. In *[TVL97]*, pages 27–42. 1997.

[BTMS] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In *[Roz99]*.

[Bur97] R. Burkhardt. *UML – Unified Modeling Language – Objektorientierte Modellierung für die Praxis.* Addison-Wesley-Longman, Bonn, 1997.

[CODL95] G. Costagliola, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *IEEE Computer*, pages 56–66, 1995.

[DIN79] *DIN NORM 66241*, 1979. Informationsverarbeitung, Entscheidungstabelle, Beschreibungsmittel.

[Dud83] Dudenredaktion. *Der kleine Duden „Fremdwörterbuch“.* Bibliographisches Institut, Zürich, 1983.

[Dud89] Dudenredaktion. *Der Große Duden: Wörterbuch und Leitfaden der deutschen Rechtschreibung*. Bibliographisches Institut, Leipzig, 1989.

[Fel92] W.-D. Fellner. *Computer-Grafik, 2. Auflage*. BI-Wissenschafts-Verlag, Mannheim, Leipzig, Wien, 1992.

[Fla96] D. Flanagan. *Java in a Nutshell – Deutsche Ausgabe für Java 1.0*. O'Reilly, Bonn, 1996.

[Fla97] D. Flanagan. *Java in a Nutshell – A Desktop Quick Reference, 2. Auflage*. O'Reilly, Sebastopol, 1997.

[FS93] P. Formai and V. Schiavi. *Mit Unix arbeiten – System V Release 4 – Standard der 90er Jahre*. Verlag Heinz Heise, Hannover, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GO95] G.H. Golub and J.M. Ortega. *Wissenschaftliches Rechnen und Differentialgleichungen – Eine Einführung in die numerische Mathematik*. Heldermann Verlag, Berlin, 1995.

[Gri96] P. Griebel. *Parcon – Paralleles Lösen von grafischen Constraints*. PhD thesis, Universität-Gesamthochschule Paderborn, Februar 1996.

[ISO87] *Information Processing Systems - Computer Graphics - Metafile for the Storage and Transfer of Picture Description, IS 8632 Information (CGM)*, August 1987.

[JCJv92] I. Jacobsen, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Softare Engineering, A Use Case Driven Approach*. Addison-Wesley, Wokingham, 1992.

[Mel97] B. Melamed. Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem. Studienarbeit, Fachbereich 13, 1997.

[Min93] M. Minas. Spezifikation von Diagrammeditoren mit automatischer Layoutanpassung. In H.Reichel, editor, *Proc. 23. GI-Jahrestagung, Dresden, Reihe 'Informatik aktuell'*, pages 334–339. GI, September 1993.

[MV95] M. Minas and G. Viehstaedt. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In *[VL'95]*, 1995.

[Nie99] Magnus Niemann. Konzeption und Implementierung eines generischen Grammatikeditors für visuelle Sprachen. Master's thesis, TU Berlin, 1999.

[Oes97] B. Oesterreich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, München, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Roz97] G. Rozenberg, editor. *The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific Publishing, 1997.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications*. World Scientific Publishing, 1999. to appear.

[Rud97] Michael Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Master's thesis, TU Berlin, 1997.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, Wiesbaden, 1991.

[TER] G. Taentzer, C. Ermel, and M. Rudolf. The AGG Approach: Language and Tool Environment. In *[Roz99]*.

[TPF91] *Turbo Pascal für Windows – Programmierhandbuch*. Borland, Starnberg, 1991.

[TSF+91] W. Tietz, W. Schulz, B. Frank, D. Richter, and O. Mader. *Wissensspeicher Mathematik: Das Wichtigste bis zum Abitur in Stichworten und Übersichten*. Volk und Wissen Verlag, Berlin, 1991.

[TVL96] *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, 30. Mai 1996.

[TVL97] *Proc. of Theory of Visual Languagues*, Capri, Italy, 27. September 1997.

[UML99] UML – Unified Modeling Language. Technical report, Rational Software Corporation, http://www.rational.com, 1999.

[VL'95] *Proc. Symp. on Visual Languages*, Darmstadt, Germany, September, 5-9 1995. IEEE Computer Society Press.

[VL'97] *Proc. Symp. on Visual Languages*, Capri, Italy, September 1997. IEEE Computer Society Press.

[VLXX] *Proc. IEEE Symposium/ Workshop on Visual Languages (VL'XX)*, Los Alamitos, CA, 19XX. IEEE Computer Society Press.

Abbildungsverzeichnis

1.1 Einfache Diagrammstruktur – Ein Netzplan 3

2.1 Beispielhafter Aufbau eines Symbols 8
2.2 Beispiele für Constraints 10
2.3 Standard-Verbindungspunkte einer Primitive 11
2.4 Nassi-Shneiderman-Diagramme: Bestandteile 13
2.5 Nassi-Shneiderman-Diagramme: Programmteil (`Part`) 14
2.6 Nassi-Shneiderman-Diagramme: Leere Abfolge (`Sequence`) 15
2.7 Nassi-Shneiderman-Diagramme: Fallunterscheidung (`Selection`) 16
2.8 Nassi-Shneiderman-Diagramme: Fallunterscheidung (`Case3`) 17
2.9 Nassi-Shneiderman-Diagramme: Wiederholung (UNTIL) (`DoUntil`) . . . 18
2.10 Nassi-Shneiderman-Diagramme: Programmteil – Ausdruck 18
2.11 Nassi-Shneiderman-Diagramme: Programmteil – Programmteil 19
2.12 Nassi-Shneiderman-Diagramme: Programmteil – Abfolge 20
2.13 Nassi-Shneiderman-Diagramme: Bedingung – Fallunterscheidung 20
2.14 Nassi-Shneiderman-Diagramme: Abfolge – Fallunterscheidung 20
2.15 Rudimentärer Aufbau des Alphabet-Editors 23

3.1 Unified Modeling Language: Notation von Klassendiagrammen 25
3.2 Unified Modeling Language: Notation von Sequenzdiagrammen 28
3.3 Design Pattern: Observer 30
3.4 Design Pattern: Listener 31
3.5 Design Pattern: Whole-Part, Variante des Composite-Pattern 33
3.6 Design Pattern: Command-Processor 34
3.7 Design Pattern: Model-View-Controller 35
3.8 Anbindung des Alphabet-Editors an PARCON 42
3.9 Ausschnitt eines angepaßten OSI-Schichten-Modells 43

4.1 Struktur des Alphabet-Editors und seiner Komponenten 47
4.2 Struktureller Aufbau des Arbeitsfensters vom Alphabet-Editor 49
4.3 Klassendiagramm zur Organisation des Editoren-Managements 50
4.4 GUI-Komponenten der Benutzeroberfläche des Alphabet-Editors 51

5.1 Hiearchische Zusammensetzung eines Alphabets 55
5.2 Klassendiagramm: `GraphicObject`, Ableitungen 57

5.3 Klassendiagramm: `GraphicObject`, Allgemein 58
5.4 Klassendiagramm: `GraphicObject`, Bounding-Box 61
5.5 Klassendiagramm: `GraphicObject`, Transformationen 63
5.6 Beispiele für Knoten in verschiedenen Primitiven 64
5.7 Klassendiagramm: `GraphicObject`, Knotenpunkte 64
5.8 Klassendiagramm: `GraphicObject`, Verbindungspunkte 66
5.9 Sequenzdiagramm: Definition eines Verbindungspunktes 67
5.10 Klassendiagramm: `GraphicObject`, Bearbeitung 71
5.11 Klassendiagramm: `GraphicObject`, Darstellung 73
5.12 Klassendiagramm: `GroupObject` 75
5.13 Klassendiagramm: `AlphabetObject`, Ableitungen 77
5.14 Klassendiagramm: `DatatypeObject`, Beziehungen 80
5.15 Klassendiagramm: `Operation` 83
5.16 Objektorientierte Typisierungshierarchie der Primitiven 85
5.17 Klassendiagramm: `ConstraintObject` 87
5.18 Klassendiagramm: `Alphabet`, Allgemein 90
5.19 Klassendiagramm: `Alphabet`, Benachrichtigungsmechanismus 93
5.20 Sequenzdiagramm: Benachrichtigung von Alphabet-Listenern 94

6.1 Klassendiagramm: `LanguageText` und `GenGEd` 97
6.2 Klassendiagramm: `AbstractDrawArea` 99
6.3 Zeichenreihenfolge der Objekte einer Zeichenfläche 102
6.4 Klassendiagramm: Knotenklassen der Strukturansicht 105
6.5 Klassendiagramm: `EditorAction` 111
6.6 Verwaltung der Befehlsobjekte im Undo-Manager 114
6.7 Klassendiagramm: `GenGEdUndoManager` 115

7.1 Klassendiagramm: `Type` 124
7.2 Klassendiagramm: Typ- und Constraint-definierende Klassen 127
7.3 Klassendiagramm: `DatabaseHLCL` 128
7.4 Beziehung zwischen Constraint-Datenbank und Implementierung 130
7.5 Klassendiagramm: `ConstSolver` 133

8.1 Klassendiagramm: Package `genged.alphabeteditor.symeditor` . . . 136
8.2 Zur Verfügung stehende Primitiven des Symbol-Editors 142
8.3 Screenshot: Arbeitsfenster mit aktiviertem Symbol-Editor 144
8.4 Nassi-Shneiderman-Diagramme: Datentyp `Condition` 145
8.5 Nassi-Shneiderman-Diagramme: Datentyp `Expression` 145
8.6 Nassi-Shneiderman-Diagramme: Symbol `Part` 145
8.7 Nassi-Shneiderman-Diagramme: Unsichtbares Symbol `Sequence` 145
8.8 Nassi-Shneiderman-Diagramme: Symbol `Selection` 146
8.9 Klassendiagramm: Package `genged.alphabeteditor.coneditor` . . . 147
8.10 Screenshot: Arbeitsfenster mit aktiviertem Verbindungs-Editor 152
8.11 Nassi-Shneiderman-Diagramme: Operation `ExprInPart` 153

8.12 Nassi-Shneiderman-Diagramme: Operation `PartAppPart` 153
8.13 Nassi-Shneiderman-Diagramme: Operation `PartInSeq` 154
8.14 Nassi-Shneiderman-Diagramme: Operation `CondInSel` 154
8.15 Nassi-Shneiderman-Diagramme: Operation `SeqInTrueSel` 155

9.1 Klassendiagramm: `DiagramAlgebra` 157
9.2 Klassendiagramm: Package `genged.alphabeteditor.testsystem` . . 159
9.3 Screenshot: Arbeitsfenster mit aktiviertem Test-System 162

Tabellenverzeichnis

2.1 Nassi-Shneiderman-Diagramme: Verwendete Constraints 15
2.2 Nassi-Shneiderman-Diagramme: Datentypen (`Expression`, `Condition`) 17
2.3 Nassi-Shneiderman-Diagramme: Grafische Verbindungen 19

5.1 Darstellungsmodi der Klasse `GraphicObject` 68
5.2 Bearbeitungsmodi der Klasse `GraphicObject` 69
5.3 Konstanten zur näheren Beschreibung einer Alphabet-Änderung 92

6.1 Arbeitsmodi der Klasse `AbstractDrawArea` 103
6.2 Suffixes für Aktions-Eigenschaften innerhalb einer Definitionsdatei . . . 108

7.1 Methoden der Klasse `TypeParser` 122
7.2 Methoden der Klasse `ConstraintParser` 123

8.1 Eintrittsmethoden in der Klasse `SymbolEditor`, Teil 1 138
8.2 Eintrittsmethoden in der Klasse `SymbolEditor`, Teil 2 139
8.3 Arbeitsmodi des Symbol-Editors und der Zeichenfläche 140
8.4 Eintrittsmethoden in der Klasse `ConnectionEditor` 149

9.1 Eintrittsmethoden in der Klasse `TestSystem` 160

Benutzerhandbuch

Sonstige benutzte Literatur

[VLXX] [Rud97] [DIN79] [CODL95] [Boo86] [RBP^{+}91] [JCJv92] [Boo94] [UML99] [Boo91] [Nie99] [TER] [Roz99] [Roz97] [BTMS] [BT97] [Bar97] [MV95] [Min93] [AER96] [VL'95] [VL'97] [TVL96] [TVL97] [Fla97] [Fel92] [BMR^{+}98] [GHJV95] [ISO87] [Oes97] [GO95] [TSF^{+}91] [BSMM93] [Dud83] [Dud89] [TPF91] [Sch91] [Bur97] [FS93] [Fla96] [AU92] [Gri96] [Mel97]