SandBlocks: Integration visueller und textueller Elemente in Live-Programmiersysteme

Leon Bein, Tom Braun, Björn Daase, Elina Emsbach,
Leon Matthes, Maximilian Stiede, Marcel Taeumel,
Toni Mattis, Stefan Ramson, Patrick Rein,
Robert Hirschfeld, Jens Mönig

Technische Berichte Nr. 132

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Universität Potsdam

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 132

Leon Bein | Tom Braun | Björn Daase | Maximilian Stiede | Marcel Taeumel |
Toni Mattis | Stefan Ramson | Patrick Rein | Robert Hirschfeld | Jens Mönig

SandBlocks

Integration visueller und textueller Elemente in Live-Programmiersysteme

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2020
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
https://doi.org/10.25932/publishup-43926
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-439263

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-482-1

Vorwort

Visuelle Programmiersprachen werden heutzutage zugunsten textueller Programmiersprachen nahezu nicht verwendet, obwohl visuelle Programmiersprachen einige Vorteile bieten. Diese reichen von der Vermeidung von Syntaxfehlern, über die Nutzung konkreter domänenspezifischer Notation bis hin zu besserer Lesbarkeit und Wartbarkeit des Programms. Trotzdem greifen professionelle Softwareentwickler nahezu ausschließlich auf textuelle Programmiersprachen zurück.

Damit Entwickler diese Vorteile visueller Programmiersprachen nutzen können, aber trotzdem nicht auf die ihnen bekannten textuellen Programmiersprachen verzichten müssen, gibt es die Idee, textuelle und visuelle Programmelemente gemeinsam in einer Programmiersprache nutzbar zu machen. Damit ist dem Entwickler überlassen wann und wie er visuelle Elemente in seinem Programmcode verwendet.

Diese Arbeit stellt das SandBlocks-Framework vor, das diese gemeinsame Nutzung visueller und textueller Programmelemente ermöglicht. Neben einer Auswertung visueller Programmiersprachen, zeigt es die technische Integration visueller Programmelemente in das Squeak/Smalltalk-System auf, gibt Einblicke in die Umsetzung und Verwendung in Live-Programmiersystemen und diskutiert ihre Verwendung in unterschiedlichen Domänen.

August 2019

Leon Bein, Tom Braun, Björn Daase,
Elina Emsbach, Leon Matthes, Maximilian Stiede,
Marcel Taeumel, Toni Mattis, Stefan Ramson, Patrick Rein,
Robert Hirschfeld, Jens Mönig

Inhaltsverzeichnis

1 Einführung

Computerprogramme werden in der Regel mittels textueller Programmiersprachen erstellt. In der Industrie werden dafür heutzutage vorrangig Sprachen wie Java, C oder Python verwendet. Sie existieren schon seit vielen Jahrzehnten wodurch für sie bereits eine große Anzahl an Entwicklerwerkzeugen vorhanden ist. Zudem konnten Entwickler langjährige Erfahrungen mit diesen Sprachen sammeln.

Schon seit den 1950er Jahren gibt es die Idee, dass statt Text grafische Elemente die Syntax der Programmiersprache bilden. Diese Ansätze werden als visuelle Programmiersprachen bezeichnet. Prominente Vertreter kommen dabei aus dem Lern- und Bildungsbereich. Beispielsweise werden Sprachen wie Snap! [54], Scratch [74] oder Etoys [42] eingesetzt, um Kindern das Programmieren beizubringen. Weiterhin gibt es visuelle Programmiersprachen für Domänenexperten, in denen die Symbolik der Domänen genutzt wird ohne tiefgreifende Programmierkenntnisse vorauszusetzen. Beispiele dafür sind die Unreal Engine Blueprints oder Physiksimulationen.

Visuelle Programmiersprachen erfreuen sich bei professionellen Softwareentwicklern bisher nicht sonderlich großer Beliebtheit und werden von ihnen oft als ‚für Kinder' oder ‚nicht für richtige Entwickler' abgetan. Dabei bieten sie jedoch einige Vorteile: So haben Domänen oft eine bereits vorgeprägte Symbolik, die sich nicht direkt in Text abbilden lässt. Ein Beispiel dafür ist die Schalttechnik mit ihren Symbolen in Schaltplänen. Weiterhin lassen sich diese visuellen Sprachen besser explorieren, wodurch sie für den Menschen besser erfassbar werden. Zudem werden Syntaxfehler vermindert, da für visuelle Elemente ihre Kombination eingeschränkt werden kann. Somit können syntaktisch falsche Kombinationen vermieden werden, während sich mit Text auch irreguläre Kombinationen bilden lassen.

Um Entwicklern ihre textuelle Programmiersprache zu erhalten und gleichzeitig die Möglichkeit zu geben, die Vorteile visueller Programmiersprachen zu nutzen, gibt es Ideen, die textuelle und visuelle Ansätze verbinden. Einen solchen Ansatz stellt das SandBlocks-Framework dar. Es ermöglicht den Entwicklern innerhalb des Squeak/Smalltalk-Systems visuelle und textuelle Ansätze gemeinsam zu nutzen. Dem Entwickler ist dann selbst überlassen an welchen Stellen und in welchem Maße er visuelle Elemente als angemessen betrachtet und wie er sie verwendet.

Dabei handelt es sich bei Squeak um ein Live-Programmiersystem mit der objektorientierten Programmiersprache Smalltalk. Morphic ist das genutzte Grafik-Framework zur Integration visueller Elemente in Text.

Somit wird es einem Entwickler mit SandBlocks frei gestellt wo und wie viele visuelle Elemente er in seinem Programmcode nutzt. Dies ermöglicht nicht nur eine leichtere und weniger fehleranfälliger Erstellung, insbesondere wird so eine bessere Lesbarkeit und Wartbarkeit von Programmen gefördert. Weiterhin ist dieses

interaktiv und direkt durch den Entwickler manipulierbar. Die genutzten visuellen Elemente können zu einem beliebigen Zeitpunkt verändert werden. Somit wird die Liveness von Squeak/Smalltalk einen Schritt weitergeführt: Neben dem nicht interaktiven Text können die visuellen Live-Objekte als Programmelemente eingefügt werden, die während ihres gesamten Lebenszyklus visuell, live und manipulierbar bleiben.

Diese Arbeit unterteilt sich in sechs Kapitel. Zunächst stellen wir in Kapitel 2 ein Konzept vor, wie man textuelle und visuelle Programmiersprachen vereinigen kann. Darauf aufbauend wird die konkrete Integration in Squeak/Smalltalk-System sowie die Integration in das bestehende Tooling in Kapitel 3 gezeigt. In Kapitel 4 wird die Einordnung in die konkrete Syntax vorgenommen, sowie Details zum Kompilieren dieses heterogenen Programmtexts beschrieben. Weiterhin zeigt Kapitel 5 wie dieser, mit visuellen Elementen angereicherte, Programmtext mit anderen Entwicklern für die Zusammenarbeit geteilt werden kann. Abschließend werden mit Software-Entwurfsmustern in Kapitel 6 und nichtlinearem Programmfluss in Kapitel 7 noch zwei konkrete Anwendungsbeispiele vorgestellt, die mithilfe des SandBlocks-Frameworks innerhalb des Squeak/Smalltalk-Systems umgesetzt wurden.

2 Erweiterung einer textuellen Programmiersprache um visuelle Elemente – Vorstellung eines Konzepts heterogener Programmierung

Computerprogramme beschreiben Strukturen und Verhalten. Sie sind zugleich Anleitung und Modell. Im Rahmen dieses Kapitels soll ein Ansatz untersucht werden, wie die Verbindung zwischen dem zu beschreibendem Sachverhalt und dem diesen beschreibenden Programmtext verbessert werden kann.

Wir werden dazu ein Konzept für die Einbindung semantisch bedeutungsvoller visueller Elemente in Programmtext vorstellen. Dazu gehen wir auf die entsprechende Terminologie ein, skizzieren die Entwicklung visueller Programmierung und erarbeiten ein Konzept zur Vereinigung von textueller und visueller Programmierung. Anhand eines im Programmiersystem Squeak/Smalltalk implementierten Framework stellen wir eine prototypische Umsetzung dieses Konzepts vor. Anschließend werden wir dieses evaluieren. Dabei gehen wir auf typische Vor- und Nachteile visueller Programmierung ein und überprüfen die These, ob durch die Verbindung visueller und textueller Programmierung die Vorteile beider Ansätze genutzt und die Nachteile ausgeglichen werden können.

2.1 Einleitung

Programmieren bedeutet modellieren. Bei jedem Computerprogramm handelt es sich letztendlich um ein Modell. Stachowiak [105] beschreibt die Modellbildung als dreistellige Relation zwischen einem Original, einem Modell und einem modellbildenden Subjekt. Demnach sind Modelle vereinfachende Abbildungen von Originalen, die eine Ersetzungsfunktion einnehmen [64, 117]. Zu betonen ist hier, dass das Modell ein *durch* das Subjekt, also den Menschen wahrgenommenes Original ersetzt und dass es eine Funktion *für* den Menschen hat. Somit erscheint es naheliegend, dass ein Modell ein Original in einer das menschliche Wahrnehmungssystem ansprechenden Art und Weise repräsentierten sollte.

Welche Auswirkungen hat dies nun auf die Tätigkeit des Programmierens? Die Geschichte der Informatik ist geprägt von einer Orientierung hin zum Menschen. Während zu Beginn die technischen Voraussetzungen von Maschinen im Mittelpunkt der Konzeption von Programmen und Programmiersprachen standen, richteten sich spätere Ansätze immer mehr nach den Anforderungen der menschlichen Konstitution. Die Ablösung von Programmen in Form von Zahlencodes durch die ersten mit Assemblersprachen geschriebenen textuellen Programme oder auch die Einführung grafischer Benutzeroberflächen stellen die Anfänge dieser Entwicklung dar.

Eine Vielzahl von Entwicklern hat den Ansatz der visuellen Programmierung als weiteren entscheidenden Schritt in der Annäherung an die menschliche Vorstellungswelt identifiziert. Doch trotz zahlreicher Arbeiten zu diesem Thema, stellen visuelle Programmiersprachen in der Praxis nach wie vor eine Ausnahme dar. Woher rührt diese zurückhaltende Einstellung?

Zum einen herrscht innerhalb der ingenieurwissenschaftlichen Disziplinen die Tendenz, an etablierten Paradigmen festzuhalten. Dies hängt teilweise damit zusammen, dass es viele bestehende Systeme gibt, welche mit alten Techniken entwickelt wurden und sich nun – außer durch eine vollständige Neukonzeption – nicht auf die neuen Methoden umstellen lassen. Auch ist die Aneignung einer vollständig neuen Herangehensweise mit einigen Anstrengungen und der Aufgabe des Expertenstatus für den einzelnen Programmierer verbunden.

Andere Gründe stehen in unmittelbarem Zusammenhang mit Entwurfsschwierigkeiten visueller Programmiersprachen. Es hat sich gezeigt, dass visuelle Notationen dann am wirkungsvollsten sind, wenn sie unmittelbar einer Anwendungsdomäne entspringen. Soll das visuelle Potential also optimal ausgenutzt werden, entstehen Sprachen, welche sich nur für eingeschränkte Einsatzgebiete eignen. Dominiert hingegen der Anspruch an die Sprache, allgemein einsetzbar zu sein, nimmt diese weniger intuitive visuelle Formen an.

Eine weitere Schwierigkeit bei der Konzeption einer visuellen Programmiersprache besteht darin, auch für abstrakte Programmkonstrukte wie Kontrollstrukturen, Variablen, Rekursionen oder funktionale Abstraktionen Visualisierungen zu finden. Während es zwar teilweise zugängliche visuelle Gegenstücke zu den üblichen textuellen Repräsentationen gibt, etwa in Form von Datenfluss-Graphen [17], fällt es zumeist eher schwer adäquate visuelle Äquivalente zu finden. Interessanterweise lassen sich gerade komplexe und abstrakte Konzepte gut textuell beschreiben und erklären, während konkrete Strukturen für eine visuelle Darstellung prädestiniert sind.

2.1.1 Forschungsfrage

Anknüpfend an die im vorangegangenen Abschnitt beschriebene Situation erscheint uns die Integration visueller Elemente in eine etablierte textuelle Programmiersprache eher den momentanen Herausforderungen praktischer Software-Entwicklung zu entsprechen als die Entwicklung einer weiteren rein visuellen Programmiersprache. Daher möchten wir uns im Rahmen dieses Kapitels mit der folgenden Forschungsfrage beschäftigen:

> Lassen sich semantisch bedeutungsvolle visuelle Elemente so in eine etablierte textuelle Programmiersprache einbinden, dass jedes Paradigma das andere unterstützt, wo es überlegen ist?

Hierbei gilt es insbesondere zu klären, ob durch diesen heterogenen Programmieransatz die Vorteile beider Darstellungsarten erhalten und gleichzeitig die Nachteile reduziert werden können.

2.1.2 Gliederung

In Abschnitt 2.2 klären wir wichtige terminologische Aspekte der heterogenen Programmierung. In Abschnitt 2.3 möchten wir dann theoretisch unser Konzept heterogener Programmierung entwickeln. Um dieses in den Kontext bestehender Arbeiten einordnen zu können, gehen wir auf die historische Entwicklung visueller Programmierung ein. Dabei legen wir einen Schwerpunkt auf heterogene Tendenzen.

In Abschnitt 2.4 stellen wir mit SandBlocks ein von uns entwickeltes Framework zur Integration semantisch bedeutungsvoller visueller Elemente in die Programmierumgebung Squeak vor. In Abschnitt 2.5 steht nach der Architektur dann die Nutzung unseres Frameworks im Vordergrund.

Nachdem in Abschnitt 2.2 und Abschnitt 2.3 die theoretischen aber auch in Abschnitt 2.4 und Abschnitt 2.5 die praktischen Grundlagen zur Beantwortung der Forschungsfrage gelegt wurden, folgt in Abschnitt 2.6 die Evaluation und Diskussion des vorgestellten heterogenen Ansatzes. Hierzu benennen wir Vor- und Nachteile visueller Elemente und untersuchen, inwiefern diese auf unser Konzept heterogener Programmierung zu übertragen sind. In Abschnitt 2.7 weisen wir abschließend auf offene Fragestellungen hin und geben eine Zusammenfassung.

2.2 Terminologie

Um zu einem einheitlichen Begriffsverständnis zu gelangen, werden wir in diesem Abschnitt wesentliche Termini im Feld der visuellen Programmierung erläutern. Darüber hinaus soll durch die hier vorgenommene Analyse auf zentrale Charakteristika und somit Anknüpfungspunkte für weitere Untersuchungen aufmerksam gemacht werden. Besonders wichtig erscheint uns in diesem Zusammenhang die Gegenüberstellung der beiden Adjektive *visuell* und *textuell*. Auf dieser Unterscheidung aufbauend werden wir die Ausdrücke *visuelle Programmiersprache*, sowie *visuelle Programmierung* spezifizieren.

2.2.1 *Visuell* in Abgrenzung zu *textuell*

In dem Bemühen die Bedeutung von *visuell* nicht künstlich einzuschränken und dennoch den Unterschied zu *nicht-visueller* Programmierung herauszuarbeiten, greifen wir auf die in der Kognitionspsychologie verwendete Differenzierung zwischen textueller und visueller Information zurück [5].

Programme, welche mit *nicht-visuellen* Programmiersprachen erstellt werden, sind zwar sichtbar, aber nicht visuell. Der Quelltext wird hier durch das verbale Wahrnehmungssystem gedeutet, welches unabhängig davon ist, ob die Aufnahme optisch, akustisch oder haptisch erfolgt.

Visuelle Programmiersprachen hingegen bedienen sich Farben, Formen, Überlappungen, Verbindungen etc., deren Verarbeitung über das visuelle Wahrnehmungs-

system erfolgt. So erstellte Programme sind also nicht nur als sichtbar, sondern auch als visuell zu bezeichnen.

Für die der visuellen Programmierung gegenüberstehende nicht-visuelle Programmierung finden sich in der Literatur zahlreiche beschreibende Adjektive wie *eindimensional, herkömmlich, linear, textuell* oder *traditionell* ([101, S. 12], [21, S. 22], [85, S. 8], [80, S. 877]). Zwar würde sich aufgrund der obigen Ausführungen vor allem das Beiwort *verbal* zur Beschreibung eignen, doch halten wir uns aus Gründen der Einheitlichkeit an das in der Literatur vorherrschende Adjektiv *textuell*.

2.2.2 *Visuelle Programmiersprache* in Abgrenzung zu *textueller Programmiersprache*

Visuelle Programmiersprachen werden in der Literatur als formale Sprachen mit visueller Syntax oder visueller Semantik definiert ([2, S. 77], [22, S. 45], [23, S. 14], [96, S. 44]). Als Beispiel für ein visuelles syntaktisches Konstrukt führt Schiffer [96] einen Pfeil an, welcher zwei Elemente einer Grafik miteinander verbindet. Die Färbung des Pfeils, die den Zustand der Beziehung dieser beiden Elemente ausdrückt, etwa Grün für Aktiv und Rot für inaktiv, ist hingegen ein Beispiel für ein visuelles semantisches Konstrukt.

Als abgrenzendes Merkmal visueller Programmiersprachen gilt zudem deren Mehrdimensionalität ([50, S. 7], [80, S. 877], [100, S. 138]). Während textuelle Programmiersprachen ausschließlich auf eindimensionalen Zeichenketten beruhen, beziehen Syntax und Semantik visueller Programmiersprachen meist den zwei- oder dreidimensionalen Raum mit ein.

Somit ergibt sich die folgende Gegenüberstellung visueller und textueller Programmiersprachen. Innerhalb textueller Programmiersprachen sind visuelle Elemente syntaktisch und semantisch bedeutungslos. Die syntaktische Struktur des Programms ergibt sich aus der linearen Anordnung der lexikalischen Elemente. Visuelle Anreicherungen haben semantisch keine Bedeutung. Formatierungen, die den Lesefluss des Menschen erleichtern, werden bei der Übersetzung in Maschinencode ignoriert.

Visuelle Programmiersprachen hingegen enthalten syntaktisch oder semantisch bedeutungstragende visuelle Elemente. Die Syntax ergibt sich aus den verwendeten Grafiken und Texten, deren räumlicher Anordnung sowie deren Verbindungen. Bei der semantischen Deutung dieser werden visuelle Attribute berücksichtigt.

2.2.3 *Visuelle Programmierung* in Abgrenzung zu *Programmierung innerhalb einer visuellen Programmierumgebung*

Unter *visueller Programmierung* versteht man die Erstellung von Software mit visuellen Programmiersprachen ([23, S. 14], [27, S. vi], [100, S. 9], [96, S. 46]). Hiervon zu unterscheiden ist die Programmierung innerhalb einer *visuellen Programmierumgebung*.

Eine visuelle Programmierumgebung ermöglicht visuelle Arten des Umgangs mit einer Programmiersprache, unabhängig davon, ob die verwendete Programmiersprache selbst visueller oder textueller Natur ist ([23, S. 14], [96, S. 46]).

2.3 Geschichtliche Betrachtungen

Um die Forschungsfrage in einen Gesamtkontext einordnen zu können, skizzieren wir die Entwicklung visueller Programmierung. Dabei unterscheiden wir zwei Phasen. Die erste Phase umfasst die Ursprünge und ist geprägt von einer enthusiastischen und experimentellen Herangehensweise. Die zweite Phase ist als Antwort auf auftretende Skalierungsschwierigkeiten zu sehen und lässt sich durch eine zunehmende Spezifizierung charakterisieren. Anschließend an die Betrachtung dieser beiden Phasen, stellen wir die aktuelle Situation dar. Hierbei liegt der Fokus auf verwandten Arbeiten im Bereich der heterogenen Programmierung.

Für weitere geschichtliche Ausführungen sei auf die Arbeiten von Burnett [20], Chang [26], Schiffer [96] sowie Wexelblat [114] verwiesen.

2.3.1 Anfänge und erste Experimente

Bereits Ende der 50er Jahre entwickelte die Wissenschaftlerin Lois Haibt bei IBM ein System zur automatischen Generierung von Flussdiagrammen aus Assembler- und Fortran-Programmen [86].

Im Jahr 1962 entstand *Sketchpad* als Teil der Doktorarbeit von Ivan Sutherland am MIT. Bei Sketchpad handelte es sich um einen interaktiven Grafikeditor mit objektorientiertem Ansatz. Kay [63] bezeichnet Sketchpad als erstes brauchbares System mit grafischer Interaktion.

Im Jahr 1975 entwickelte David Canfield Smith im Rahmen seiner Dissertation an der Stanford University eine visuelle Programmierumgebung mit dem Namen *Pygmalion* [103]. Smith beschreibt sein System als

> „[...] computational extension of the brain's short term memory [that provides an] alternative storage for mental images during thought [...]" [103, S. ii].

Smith stellte in seiner Dissertation die These auf, dass die Kluft zwischen der menschlichen Herangehensweise an ein Problem und dessen Bewältigung durch den Computer bei traditionellen Programmieransätzen zu groß sei. Daraufhin verfolgte er jedoch nicht den Ansatz, die Tätigkeit des Programmierens dem menschlichen Denken anzunähern. Vielmehr sollten sich die mentalen Strategien des Programmierenden durch stetiges Aufzeigen des aktuellen Programmzustandes an der Verarbeitung durch den Computer orientieren [103].

1982 veröffentlichte MacDonald [72] in der Zeitschrift Datamation einen der ersten Artikel mit dem Titel „Visual Programming", in dem er die Meinung vertritt, dass visuelle Programmierung die Softwareentwicklung beschleunigen und auch Nicht-Programmierer in die Lage versetzen würde, eigene Anwendungen zu

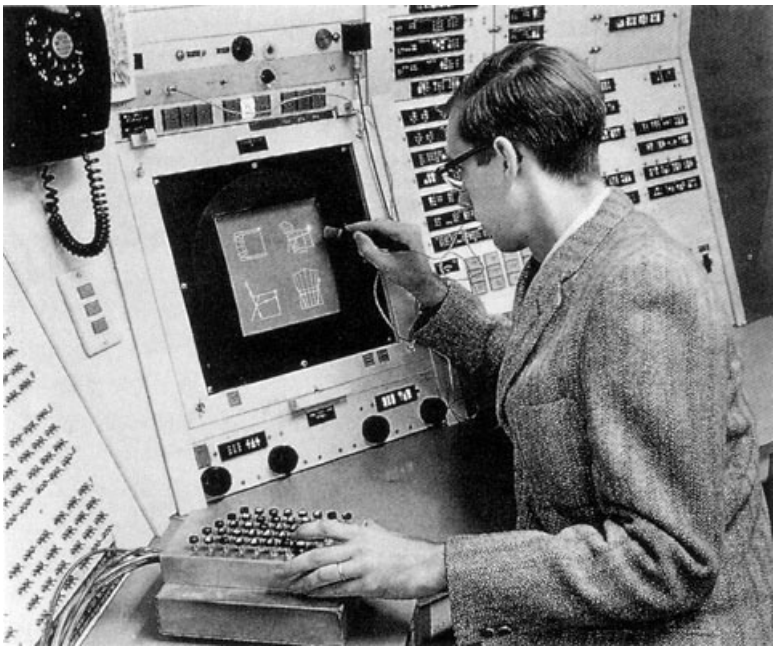


Abbildung 2.1: Ivan Sutherland bei der Arbeit mit Sketchpad im Jahr 1962. (`https://history-computer.com/ModernComputer/Software/Sketchpad.html`, letzter Zugriff am 18. Juni 2019)

implementieren. Zu dieser Zeit fand sich zudem eine internationale Gruppe von Wissenschaftlern unter dem Namen *Visual Language Community* zusammen, die es sich zum Ziel gesetzt hatten, den Schwerpunkt ihrer Forschungsaktivitäten ganz auf visuelle Programmiersprachen zu verlegen.

2.3.2 Skalierungsschwierigkeiten und Spezifizierung

Begünstigt durch Fortschritte im Hardwarebereich, entstanden immer komplexere Software-Systeme. Dies führte zu einer Art Rollenverschiebung zwischen visuellen und textuellen Programmieransätzen. Visuelle Systeme zeichneten sich bis dahin durch ihre intuitive Zugänglichkeit und Direktheit aus. Gegenüber den als stark formal empfundenen textuellen Sprachen bestachen sie durch ihre Nähe zur menschlichen Vorstellungswelt und der Ausnutzung sekundärer Notation (s. Abschnitt 2.6.1). Die komplexeren Softwareanwendungen ließen sich jedoch immer schwerer mit den vorhandenen visuellen Möglichkeiten modellieren. Textuelle Sprachen wirkten im Vergleich klar und übersichtlich.

Eine Strategie, diese Probleme zu überwinden, bestand darin, visuelle Elemente nur noch als unterstützende, nicht mehr aber als alleinstehende Ausdrücke einzusetzen. So entstanden kommerziell erfolgreiche visuelle Programmierumgebungen wie Microsofts *Visual Basic* oder das von ParcPlace Systems entwickelte *VisualWorks*. Wie in Abschnitt 2.2.3 erläutert, kann bei solchen Umgebungen im strengeren Sinne jedoch nicht von visueller Programmierung gesprochen werden.

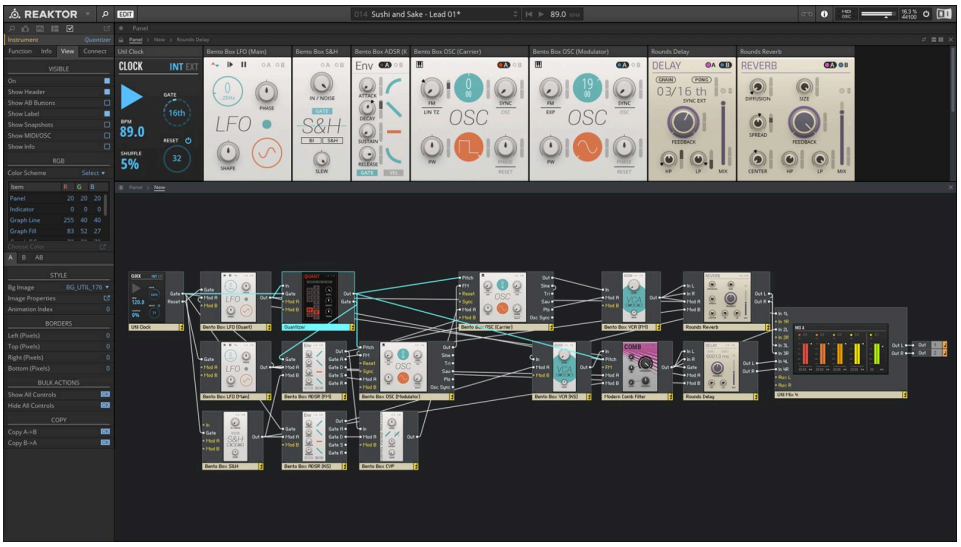


Abbildung 2.2: Benutzeroberfläche von Reaktor 6. (`https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/`, letzter Zugriff am 18. Juni 2019)

Ein davon abweichender Ansatz bestand darin, sich auf einzelne Domänen zu konzentrieren. Dies erlaubte einen reduzierten Befehlssatz sowie auf den Anwendungsbereich zugeschnittene Notationen. Die so entwickelten visuellen Systeme erlaubten es, das Vokabular der spezifischen Domäne beizubehalten, und unterstützten den Arbeitsprozess noch durch weitere direkt auf die Probleme ausgerichtete visuelle Elemente wie Icons, Buttons und Drop-Down-Menüs.

Der zweite Ansatz erfreute sich innerhalb der 1980er und 1990er Jahre großer Beliebtheit bei der Entwicklung von Spielen und in der Musikindustrie.

Frühe Systeme wie *Pinball Construction Set* (PCS) [71] oder *Adventure Construction Set* [49] fokussierten sich auf einen einzigen Spieletypus und bedurften so nur eines minimalen Befehlssatzes. Während im Bereich der visuellen Programmierung bis dahin vor allem auf Flussdiagramme zurückgegriffen worden war, wurden hier die Spielelemente selbst als Primitive genutzt.

Mitte der 90er Jahre entwickelten Blue Ribbon SoundWorks mit *Bars and Pipes* und Native Instruments mit *Reaktor* (s. Abbildung 2.2) visuelle Systeme zur Erstellung und Bearbeitung von Musik [92].

Neben Musik- und Spieleindustrie ist als weiteres wichtiges Anwendungsgebiet der Bildungssektor zu erwähnen. Dieser erscheint geradezu prädestiniert für den Einsatz visueller Programmiersprachen [12, 82, 112]. Die Komplexität der von Anfängern entworfenen Programme ist vergleichsweise gering und es genügt ein eingeschränkter Befehlssatz. Auf diesem Abstraktionslevel wirken die visuellen Elemente unterstützend und ermöglichen einen intuitiven Zugang zur Tätigkeit

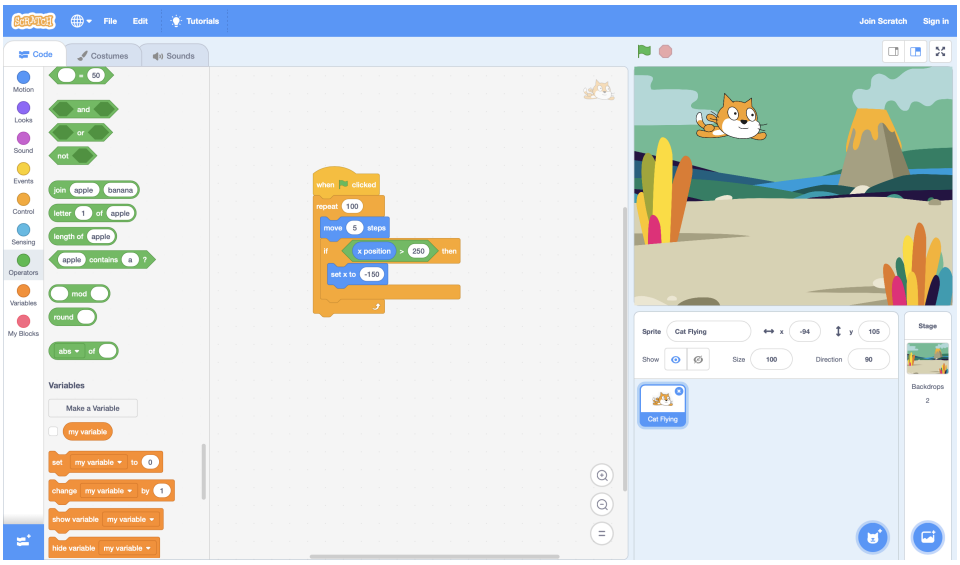


Abbildung 2.3: Benutzeroberfläche von Scratch 3.0.

des Programmierens. Zu den bekanntesten der so entstandenen Systeme zählen *Alice*[1], *Blockly* [2], *Etoys* [42], *Scratch* [74] (s. Abbildung 2.3) und *Snap!* [54].

2.3.3 Heterogene Ansätze

Studien wie die von Bau u. a. [12], Boshernitsan und Downes [14], Green und Petre [53], Noone und Mooney [82], Weintrop und Wilensky [112] und Whitley [115] scheinen zu belegen, dass visuelle Programmierung ein starkes Konzept für Anfänger und Nicht-Programmierer darstellt, für die Entwicklung komplexer Software und die Beschreibung abstrakter Strukturen jedoch weniger adäquat ist.

Für diese Sichtweise spricht auch, dass visuelle Programmiersprachen in der Praxis vor allem im Bildungsbereich eingesetzt werden. Dort werden sie als Einstieg betrachtet und zur Hinführung zu textuellen Programmiersprachen genutzt.

Die Frage, wie dieser Übergang vom visuellen zum textuellen Programmieren fließender gestaltet werden kann, ist in den letzten 15 Jahren in den Fokus der Forschung getreten [29, 33, 46, 61, 66, 113]. Es entstanden sogenannte *hybride* oder *heterogene* Ansätze, welche sowohl Merkmale visueller als auch textueller Programmierung aufweisen.

Weintrop und Wilensky [112] unterscheiden bei diesen heterogenen Systemen zwei Varianten: Entweder werden ein textueller und ein visueller Modus unterstützt oder textuelle Elemente werden in eine visuelle Umgebung integriert.

Im ersten Fall kann der Nutzer beliebig zwischen einer visuellen, meist blockbasierten und einer textuellen Ansicht wechseln. Als Beispiele sind hier *Pencil Code*

[1] Alice: `http://www.alice.org`, letzter Zugriff am 16. Juni 2019.
[2] Blockly: `https://developers.google.com/blockly/`, letzter Zugriff am 16. Juni 2019.

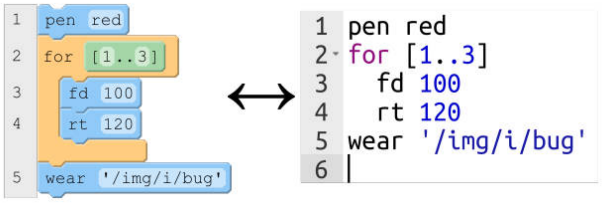


Abbildung 2.4: Pencil Code erlaubt das bidirektionale Wechseln zwischen einem block- und einem textbasierten Modus. [12]

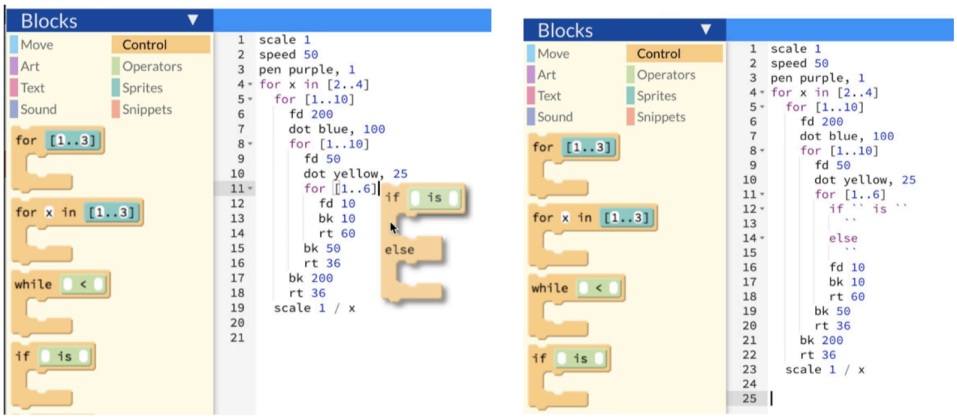


Abbildung 2.5: Benutzeroberfläche von Pencil.cc. Links ist zu sehen, wie Blöcke per Drag-and-drop in den Text-Editor bewegt werden können. Rechts ist das Ergebnis dieser Aktion dargestellt. [112]

[11] (s. Abbildung 2.4), *App Lab* [12], *BlockEditor* [76] und *Tiled Grace* [57] zu nennen, wobei es sich bei den jeweils unterstützten textuellen Sprachen um Coffee Script, JavaScript, Java und Grace handelt. *Alice* und *Blockly* erlauben ebenfalls eine textuelle Ansicht, jedoch ist diese nicht editierbar.

Die zweite Variante heterogener Systeme vermischt visuelle und textuelle Elemente innerhalb einer einzigen Umgebung [112, S. 44]. So behält *Pencil.cc* [112] die in visuellen Systemen verbreitete Seitenleiste zur Auswahl der verfügbaren Blocktypen sowie die Möglichkeit der Interaktion mittels Drag-and-drop bei, kombiniert dies aber mit einem Text-Editor. Sobald ein Nutzer einen Block mittels Drag-and-drop auf den Text-Editor bewegt, verwandelt sich der Block in sein textuelles Äquivalent, welches in das Programm eingefügt wird (s. Abbildung 2.5).

2011 integrierte Federici [38] Elemente der Sprache C in ein an Scratch erinnerndes Block-System und schuf so *blockC*. Ein ähnliches Projekt stellten Kyfonidis u. a. [68] mit *Block-C* vor. Matsuzawa, Ohata, Sugiura und Sakai Matsuzawa u. a. entwickelten eine auf Java basierende visuelle Programmiersprache. Robinson [94] präsentierte mit *Patch* eine Kombination von Python und Scratch.

All diesen Ansätzen gemeinsam ist ihre Fokussierung auf Programmiereinsteiger. Cheung u. a. [28] glauben, dass jeder Typus von Programmiersprache für eine

andere Altersgruppe geeignet sei. Für Kinder und Jugendliche bis zum Alter von 14 Jahren böten sich visuelle Programmiersprachen an. Von 15 bis 17 Jahren seien heterogene Umgebungen zu bevorzugen. Ab einem Alter von 18 Jahren sollten textuelle Sprachen verwendet werden. Anjudar u. a. [7] kommen zu ähnlichen Ergebnissen wie Cheung u. a. [28].

2.3.4 Konzept heterogener Programmierung

Um die Stoßrichtung verwandter Arbeiten im Bereich heterogener Programmierung der letzten Jahre zu verdeutlichen, möchten wir zwei Berichte zitieren. Die uns besonders relevant erscheinenden Stellen haben wir dabei kursiv hervorgehoben.

> „While deciding [text] symbols to be used in, it was kept in mind that *symbols must be as view as possible* and most understandable shapes have been selected *to represent corresponding constructs* of JavaScript.“ [3, S. 14]
>
> „The aim of this research is to design Visual Programming extension of JavaScript language to make programming in JavaScript more understandable *to fresh or new programmer*.“ [3, S. 19]
>
> „We found that on average a hybrid-based approach increases the *students* understanding of programming foundations, memorization, and *ease of transition* by more than 30% when compared to a *block-based to text-based learning approach*.“ [4, S. 1]
>
> „We build the hybrid-based PencilCode *to reduce the learning gap between the block-based and text-based approaches*. [4, S. 3]

Wie bereits in Abschnitt 2.3.3 beschrieben, richten sich heterogene Ansätze primär an Programmiereinsteiger. Zielgruppe sind Kinder und Jugendliche [7, 28].

Visuelle Konzepte sollen textuelle nicht erweitern, sondern nur illustrieren („to represent corresponding constructs“ [3, S. 14]). Das heißt, dass visuelle Elemente nicht über die Ausdrucksmächtigkeit von Text hinausgehen.

Das Verhältnis visueller und textueller Programmierung wird zeitlich begriffen. Visuelle Programmiersprachen stellen den Einstieg, den Beginn dar. Textuelle Programmiersprachen sind das Ziel, dass es zu erreichen gilt. Heterogene Programmiersysteme werden als Weg gesehen, dieses Ziel zu erreichen.

–

Wir möchten mit diesem Kapitel ein alternatives Konzept heterogener Programmierung aufzeigen. Im Folgenden beschreiben wir dieses und grenzen es von vorherigen Strategien ab.

Die Grundlage bildet eine textuelle Sprache. In unserem Fall greifen wir auf das Programmier-System Squeak/Smalltalk (s. Abschnitt 2.4.1) zurück. Diese textuelle Sprache wird um visuelle Ausdrucksmöglichkeiten erweitert. Visuelle und textuelle Programmierung stehen nicht nur als getrennte Modi zur Verfügung (s. Abschnitt 2.3.3). Stattdessen kann prinzipiell an jeder Stelle im Code sowohl ein

textuelles als auch ein visuelles Element verwendet werden. Somit gehört unser Ansatz nicht zu der ersten der von Weintrop und Wilensky [112] unterschiedenen Kategorien heterogener Systeme (s. Abschnitt 2.3.3). Dadurch dass wir eine textuelle Programmiersprache als Basis wählen und diese visuell anreichern, lässt sich unsere Herangehensweise auch nicht der zweiten Kategorie Weintrops zuordnen.

Zielgruppe unseres Konzepts sind nicht primär Programmiereinsteiger, sondern ganz allgemein Programmierende. Es werden keine Annahmen über deren Kenntnisstand getroffen.

Visuelle Notationen dienen nicht ausschließlich der Illustration, sondern stellen ausführbaren Code dar. Dies kann etwa die Form der Beschreibung eines Teils eines Algorithmus oder einer Datenstruktur annehmen.

Zudem ist es unser Ziel, dass die visuellen Elemente, welche wir einsetzen, im Vergleich zu ihren textuellen Entsprechungen potentiell mehr Funktionalitäten anbieten [3].

Es geht uns nicht darum, möglichst viel Text zu eliminieren [3, S. 14], sondern Konzepte zu identifizieren, welche sich besser visuell als textuell beschreiben lassen. Dabei streben wir keine Spezifizierung der gesamten Sprache an, wie dies seit 1980 in zahlreichen Fällen geschehen ist (s. Abschnitt 2.3.2), sondern bewegen uns auf einer Ebene innerhalb der Sprache. Visualisierungen sollen etwa für Datentypen, Datenstrukturen oder zur Unterstützung von Entwurfsmustern [43] gefunden werden. Es soll jedoch keine Fokussierung der gesamten Sprache auf eine einzelne Domäne stattfinden.

Trotz der Modifikation der zugrunde liegenden textbasierten Programmierumgebung, soll Kompatibilität gewährleistet werden. Das heißt, dass alte, rein textuelle Programme nach wie vor ausführbar sein müssen und rein textuelle Programmierung weiterhin unterstützt wird.

2.4 SandBlocks: Integration visueller Elemente in Smalltalk

In diesem Abschnitt stellen wir mit SandBlocks ein Framework zur Integration semantisch bedeutungsvoller visueller Elemente in die textbasierte Programmierumgebung Squeak vor. Mit SandBlocks lassen sich visuelle Elemente, sogenannte Blöcke, erstellen und in Programmcode einsetzen.

In Abschnitt 2.4.1 gehen wir zunächst kurz auf Squeak, sowie das Morphic-Framework ein und nennen die Anforderungen, an denen wir uns bei der Konzeption von SandBlocks orientiert haben. Daran anschließend beschäftigen wir uns in Abschnitt 2.4.2 mit der zentralen Abstraktion von SandBlocks, dem Block. Wir beschreiben Anknüpfungspunkte für das Konzept des Blocks, führen dieses weiter aus und skizzieren dessen Umsetzung.

2.4.1 Die Umgebung Squeak: Smalltalk und Morphic

Squeak ist eine objektorientierte, plattformunabhängige und quelloffene Programmierumgebung, welche auf der Programmiersprache Smalltalk basiert [15, 60].

Das gesamte System, d. h. Compiler, Debugger, Programmierwerkzeuge etc., beruht ausschließlich auf Programmtext der Sprache Smalltalk und kann durch den Nutzer nach Belieben verändert werden.

Das *Morphic*-Framework bildet die Grundlage für die grafische Benutzeroberfläche von Squeak [73] und löste das vormals verwendete Model View Controller (MVC) Konzept von Smalltalk-80 ab. Die zentrale Abstraktion von Morphic ist ein *Morph*. Bei einem Morph handelt es sich um ein Squeak-Objekt, welches eine visuelle Repräsentation besitzt. Alle auf dem Bildschirm, der sogenannten *Welt*, sichtbaren Formen erben von der Klasse `Morph`. Die beiden Kernideen von Morphic sind *Direktheit* und *Lebendigkeit* [13, S. 231]. In der Welt angezeigte Morphs lassen sich *direkt* untersuchen und modifizieren, etwa durch Interaktion mit der Maus. Mit *Lebendigkeit* ist gemeint, dass diese Interaktion zu jedem Zeitpunkt stattfinden kann: Die Sicht auf die Welt repräsentiert stets den aktuellen Zustand des Systems.

Squeak bietet sich insofern für den Entwurf einer heterogenen Programmiersprache an, als es zum einen mit Smalltalk auf einer etablierten textuellen Programmiersprache beruht. Zum anderen ist die Programmiersprache direkt in eine Programmierumgebung integriert und Compiler, Parser und Programmierwerkzeuge lassen sich unmittelbar modifizieren. Schließlich weist Squeak mit dem Morphic-Framework einen direkten Anknüpfungspunkt für die Integration visueller Elemente auf.

2.4.2 Unsere Anforderungen an SandBlocks

Mit SandBlocks stellen wir ein Framework vor, welches visuelle und textuelle Programmierung in Squeak miteinander verbinden soll. Dabei orientieren wir uns an dem in Abschnitt 2.3.4 entwickelten Konzept heterogener Programmierung. Für die Umsetzung ergeben sich die folgenden Anforderungen:

1. *Verwendbarkeit visueller Elemente im Programmcode.* Neben Text lassen sich auch visuelle Elemente im Programmcode verwenden. Sie können prinzipiell an beliebiger Stelle stehen, sind editierbar und folgen wie auch Text gewissen Formatierungsregeln. Textuelle und visuelle Elemente sind direkt miteinander kombinierbar und erfordern keine getrennten Darstellungsumgebungen (vgl. Abschnitt 2.3.3).

2. *Ausführbarkeit visueller Elemente.* Visuelle Elemente dienen nicht ausschließlich der Illustration, sondern stellen semantisch bedeutungsvolle, ausführbare Einheiten dar. Sie können sowohl Objekte als auch Methoden und Nachrichten repräsentieren.

3. *Lebendigkeit visueller Elemente.* Visuelle Elemente sind ähnlich wie Morphs lebendige Instanzen (vgl. Abschnitt 2.4.1), welche sich in den bislang ausschließlich statischen Programmcode integrieren lassen.

4. *Erstellbarkeit und Modifizierbarkeit visueller Elemente.* Der Nutzer verfügt über die Möglichkeit eigene visuelle Elemente zu erstellen, sowie bereits vorhan-

dene anzupassen. Das Erscheinungsbild visueller Elemente ist zu jedem Zeitpunkt modifizierbar. Für ein semantisches Konzept können zudem mehrere visuelle Repräsentationen konstruiert werden, unter denen sich dann auswählen lässt.

5. *Kompatibilität mit vorhandenen Werkzeugen.* Squeak stellt eine Reihe mächtiger Entwicklungswerkzeuge zur Verfügung. Zu diesen gehören u. a. *Browser, Workspace, Test Runner, Inspector* und *Debugger*. Diese sind auch für visuelle Elemente nutzbar und stellen gegebenenfalls erweiterte Funktionalitäten im Umgang mit ihnen zur Verfügung.

6. *Serialisierbarkeit und Versionierbarkeit.* Squeak bietet mit *Change Sets*, *Monticello Browser* oder *Git Browser* Möglichkeiten zur Speicherung von Code und zur Verwaltung von Projekten. Diese Mechanismen werden so erweitert bzw. angepasst, dass sie auch die Verwendung visueller Elemente in Programmen gestatten.

7. *Kompatibilität mit der aktuellen Squeak Version 5.2b.* Trotz der am System vorgenommenen Änderungen, sind zuvor erstellte Programme weiterhin ausführbar.

8. *Skalierbarkeit des visuellen Anteils innerhalb eines Programms.* Rein textuelle Programmierung wird weiterhin unterstützt und ist uneingeschränkt möglich. Der Anteil an sowie die Art von visuellen Elementen unterliegt der Kontrolle des Nutzers.

So wie das Morphic-Framework auf Morph-Objekten basiert, so stellen Blöcke die zentrale Abstraktionskomponente von SandBlocks dar. Die Grundidee von SandBlocks besteht darin, Konzepte von Morphic und MVC miteinander zu kombinieren.

Morphic besticht durch seine direkte Manipulierbarkeit (s. Abschnitt 2.4.1). Es stehen visuelle Möglichkeiten zur Verfügung, Morphs in der Welt zu inspizieren, zu modifizieren und zu komponieren. So lassen sich bspw. sehr schnell Prototypen zusammenstellen.

Obwohl sich die visuelle Repräsentation eines Morphs in der Welt leicht verändern lässt, fehlt die Möglichkeit der Wiederverwendbarkeit. Die Tätigkeit des Programmierens in Squeak ist auf das Klassen-System konzentriert. Anwendungen entstehen dadurch, dass die entsprechenden Klassen implementiert werden. Bei sichtbaren Morphs handelt es sich jedoch um individuelle Objekte. Zwar spiegeln sich Änderungen an der Klasse eines Morphs unmittelbar in den in der Welt sichtbaren Instanzen dieser wider (s. Abschnitt 2.4.1). Es besteht jedoch andererseits nicht die Möglichkeit, die an einer sichtbaren Instanz vorgenommenen Änderungen im Code des Klassen-Systems zu referenzieren. In unmittelbarem Zusammenhang hiermit steht, dass Morphs keine klare Trennung zwischen dem Objekt selbst und seiner visuellen Repräsentation aufweisen.

Dem Morphic-Framework steht nun der Ansatz von MVC gegenüber. Dieser unterscheidet mit *Model*, *View* und *Controller* drei Aspekte eines Objekts. Model

und View sind voneinander entkoppelt. Dies ermöglicht einen höheren Grad an Flexibilität und Wiederverwendbarkeit [43]. Doch fehlt hier die Idee der Direktheit.

SandBlocks vereinigt nun im Konzept des Blocks die Direktheit von Morphic mit der Trennung in Model und View von MVC.

2.4.3 Das Konzept des *Blocks*

Essenziell für SandBlocks ist das Konzept des Blocks. Wie bei Morphs handelt es sich auch bei Blöcken um Objekte, die eine visuelle Repräsentation besitzen (vgl. Abschnitt 2.4.1). Doch lässt sich bei einem Block ähnlich wie bei MVC klar zwischen der Rolle des Models und der Rolle der View unterscheiden (s. Abschnitt 2.4.2). Zu beachten ist hier, dass es sich bei Model und View um verschiedene Rollen, jedoch nicht zwangsläufig auch um separate Objekte handelt.

Model und View sind durch die Etablierung eines subscribe-notify Protokolls voneinander entkoppelt und folgen in ihrer Interaktion dem Entwurfsmuster *Observer* [43]. Sobald sich die Daten des Models ändern, informiert es die von ihm abhängige View darüber. Diese erhält daraufhin die Möglichkeit, die Änderungen in ihrer Ansicht widerzuspiegeln. Im Gegenzug benachrichtigt die View das Model über Nutzer-Interaktionen.

Während eine View sich stets nur auf ein Model bezieht, kann ein Model durchaus über mehrere Views verfügen. Die aktuelle View eines Models lässt sich dann durch den Nutzer interaktiv austauschen. Dieses Prinzip findet sich bspw. auch bei Scratch [74] in Form der verschiedenen Kostüme (engl. *Costume*) eines Objekts (hier engl. *Sprite*).

2.4.4 SandBlocks in Squeak

Zentral für das Konzept des Blocks sind die beiden Komponenten Model und View wie in Abbildung 2.6 dargestellt. Da es sich hierbei nur um Rollen handelt, sieht das SandBlocks-Framework verschiedene Möglichkeiten vor, Blöcke zu konzipieren.

Für den Aspekt der View ist vor allem die bereits existierende Klasse `Morph` entscheidend. Zusätzlich wurde die Klasse `BTRBlockMorph` implementiert. Zur Umsetzung des Models wurde die Klasse `BTRAbstractBlockModel` und deren Spezialisierung `BTRConstantModel` sowie der Trait[3] `TBTRBlockModel` eingeführt.

Soll ein einziges Objekt sowohl die Rolle des Block-Models als auch die der Block-View realisieren, so muss dieses den `TBTRBlockModel`-Trait verwenden. Gleiches gilt auch, wenn es sich bei Model und View zwar um verschiedene Objekte handeln, das Model aber in eine spezifische Klassenhierarchie eingebunden werden soll.

Sollen die Rolle des Models und die der View von verschiedenen Objekten übernommen werden und ist es zudem nicht erforderlich, das Model in eine bestehende Klassenhierarchie einzupflegen, so kann das Model von `BTRAbstractBlockModel` er-

[3] Traits sind modulare Erweiterungen für Klassen, um das Konzept der Einfachvererbung zu ergänzen.

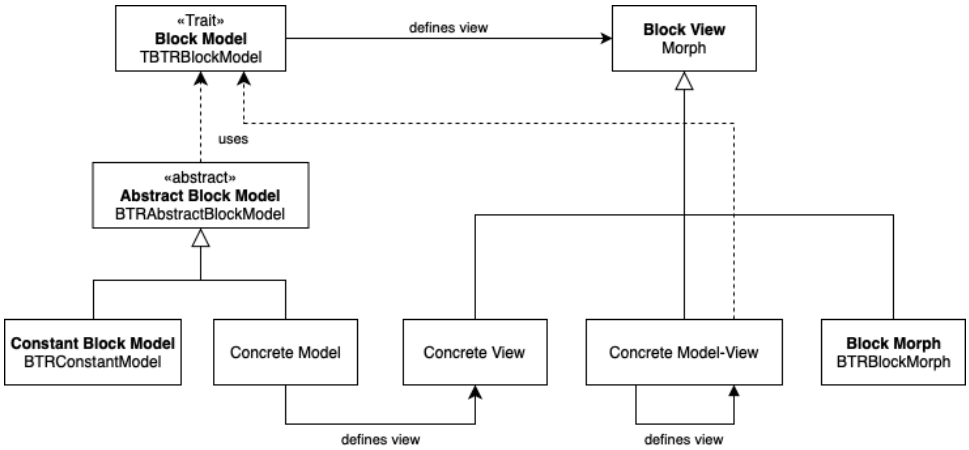


Abbildung 2.6: Komponenten eines Blocks im SandBlocks-Framework

ben. Der Nutzer muss den Trait in diesem Fall nicht selbst einbinden. Handelt es sich bei dem Objekt, welches durch den Block repräsentiert werden soll, zudem um eine Konstante, so kann das Model von der Klasse `BTRConstantModel` erben.

Falls die View in keine spezifische Klassenhierarchie eingefügt werden soll, kann als Superklasse `BTRBlockMorph` genutzt werden. Die Klasse `BTRBlockMorph` verfügt im Unterschied zu der Klasse `Morph` mit der Instanzvariable „blockModel“, sowie den entsprechenden Zugriffsmethoden `blockModel` und `blockModel:` bereits über einen Referenzmechanismus auf ein zugehöriges Block-Model (s. Listing 2.3).

2.5 Anwendung des SandBlocks-Frameworks

In diesem Abschnitt besprechen wir die Nutzung des Frameworks. Wir erläutern die Erstellung von eigenen Blöcken und illustrieren dies anhand eines Beispiels. Zudem schildern wir die Interaktion mit visuellen Elementen und zeigen mögliche Anwendungsfälle für den Einsatz von Blöcken auf.

Für weitere technische Ausführungen zur Integration des SandBlocks-Frameworks in das Programmiersystem Squeak/Smalltalk sei auf Kapitel 4 und Kapitel 5 verwiesen. Diese beschäftigen sich vertieft mit Anpassungen des Kompilierungsprozesses sowie der Serialisierung und Versionierung, welche für die Einbindung visueller Elemente erforderlich sind.

2.5.1 Erstellung von Blöcken

Um die Nutzung des Frameworks zur Erstellung eines Blocks zu illustrieren, geben wir ein erstes kleines Beispiel. Unser Ziel ist es, Farbobjekte, also Instanzen der Klasse `Color` visuell darzustellen. Zudem soll auch die Interaktion mit einem Farbobjekt visuell in Form eines Farbwählers unterstützt werden. Das gewünschte Ergebnis ist in Abbildung 2.7 zu sehen.

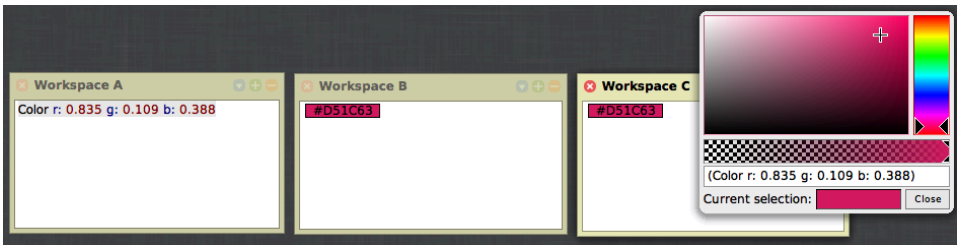


Abbildung 2.7: Der Color-Block in SandBlocks ersetzt die textuelle Beschreibung (links) von Farben. Ein interaktiver Editor (rechts) ermöglicht die Bearbeitung der Farben.

Workspace A zeigt die textuelle Darstellung einer `Color`. In Workspace B ist dieselbe `Color` zu sehen, jedoch wird sie hier durch einen Block repräsentiert. Durch einen Klick mit der Maus auf diesen Color-Block öffnet sich ein `NewColorPickerMorph`, mit welchem sich die `Color` bearbeiten lässt. Dieser Vorgang ist in Workspace C dargestellt.

Bei der Implementierung des Color-Blocks entscheiden wir uns, Color-Model und Color-View in Form von zwei separaten Objekten zu realisieren. Wir betrachten nun zunächst die Umsetzung des Color-Models. Dazu legen wir die Klasse `BTRColorModel` an, welche von `BTRConstantModel` (s. Abschnitt 2.4.4) erbt. Die entsprechende Implementierung ist in Listing 2.1 zu sehen.

Listing 2.1: BTRColorModel

```
1  BTRConstantModel subclass: #BTRColorModel
2     instanceVariableNames: ''
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'BlocksToTheRescue-Blocks'
6
7  BTRColorModel class >> #worksForObject: anObject
8     ^ anObject isColor
9
10 BTRColorModel >> #defaultValue
11    ^ Color seaFoam
12
13 BTRColorModel >> #inlineView
14    ^ BTRColorMorph for: self
```

Damit ein Objekt als Block-Model agieren kann, müssen die beiden folgenden Methoden für es definiert werden:

1. Die Methode `value` wird aufgerufen, wenn der Block ausgewertet wird. Ihr Rückgabewert spezifiziert den Wert, zu welchem der Block evaluiert.

2. Die Methode `inlineView` gibt stets einen Morph zurück. Dieser Morph übernimmt die Rolle der View. Handelt es sich bei Model und View um dasselbe Objekt, kann hier `self` zurückgegeben werden.

Erbt ein Block-Model von der Klasse `BTRConstantModel`, so stehen diese beiden Methoden direkt zur Verfügung. Dafür muss jedoch `worksForObject:` klassenseitig definiert werden. Als Argument wird ein Objekt übergeben, für das es zu entscheiden gilt, ob es für diese Art von Block-Repräsentation geeignet ist. Rückgabewert ist ein `Boolean`. In unserem Beispiel werden alle Objekte der Klasse `Color` akzeptiert (s. Listing 2.1, Z. 7 f.).

Erbt ein Block-Model von der Klasse `BTRConstantModel`, so wird empfohlen die Methode `defaultValue` zu überschreiben. Hiermit lässt sich der initiale Wert (engl. *value*) spezifizieren. In unserem Beispiel legen wir dafür „Color seaFoam“ fest (s. Listing 2.1, Z. 10 f.).

Würden wir die Methode `inlineView` nicht überschreiben, so würde standardmäßig ein Inspector die Rolle der View übernehmen. Wir entscheiden uns jedoch zu diesem Zweck die neue Klasse `BTRColorMorph` zu implementieren und diese von `BTRBlockMorph` erben zu lassen (s. Abschnitt 2.4.4). Listing 2.2 zeigt die entsprechende Umsetzung.

Listing 2.2: BTRColorMorph

```
BTRBlockMorph subclass: #BTRColorMorph
   instanceVariableNames: 'colorStringMorph'
   classVariableNames: '' poolDictionaries: ''
   category: 'BlocksToTheRescue-Blocks'

BTRColorMorph >> #initialize
   super initialize.
   colorStringMorph := #'#00000000' asMorph.
   self extent: colorStringMorph extent.
   self borderWidth: 1.
   self borderColor: Color black.
   self color: self blockModel value.
   self addMorphCentered: colorStringMorph.
   self updateText.

BTRColorMorph >> #updateText
   colorStringMorph contents: self color asHTMLColor.
   colorStringMorph center: self center.
   colorStringMorph color: (self color luminance < 0.35
      ifTrue: [Color white]
      ifFalse: [Color black]).

BTRColorMorph >> #color: aColor
   super color: aColor.
   self updateText.
   self blockModel ifNotNil: [self blockModel value: aColor].

BTRColorMorph >> #handlesMouseDown: anEvent
   ^ anEvent redButtonPressed

BTRColorMorph >> #mayActOnClick
   ^ true

BTRColorMorph >> #mouseDown: anEvent
   (NewColorPickerMorph on: self) openInWorld.
```

In der Methode `inlineView` (s. Listing 2.1, Z. 13 f.) wird der `BTRColorMorph` als View des Color-Models festgesetzt. Die an die Klasse `BTRColorMorph` gesendete Nachricht `for:` ist Teil der API von SandBlocks. Sie ist auf `BTRBlockMorph` klassenseitig implementiert (s. Listing 2.3, Z. 7–12) und registriert die View als `dependent` des Models.

Listing 2.3: BTRBlockMorph

```
1  Morph subclass: #BTRBlockMorph
2     instanceVariableNames: 'blockModel'
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'BlocksToTheRescue-Core'
6
7  BTRBlockMorph class >> #for: aBlockModel
8     ^ self basicNew
9        blockModel: aBlockModel;
10       in: [:instance | aBlockModel addDependent: instance];
11       initialize;
12       yourself
13
14 BTRBlockMorph >> #blockModel
15    ^ blockModel
16
17 BTRBlockMorph >> #blockModel: aBlockModel
18    blockModel := aBlockModel.
```

Die Erscheinung des `BTRColorMorph` (s. Abbildung 2.8) wird bei dessen Initialisierung festgelegt (s. Listing 2.2, Z. 7–15). Dabei fügen wir dem `BTRColorMorph` noch einen weiteren Morph der Klasse `StringMorph` als Komponente (hier engl. *submorph*) hinzu, welcher das repräsentierte Farbobjekt in textueller Form beschreibt (s. Listing 2.2, Z. 9, Z. 14). Die Methode `updateText` (s. Listing 2.2, Z. 17–22) stellt sicher, dass dieser durch die Instanzvariable `colorStringMorph` referenzierte `StringMorph` stets den aktuellen value des Color-Blocks wiedergibt.

Mit den Methoden `handlesMouseDown:`, `mayActOnClick` und `mouseDown:` (s. Listing 2.2, Z. 29–36) stellt ein `BTRColorMorph` Interaktionsmöglichkeiten für den Nutzer zur Verfügung.

Der so erstellte Color-Block lässt sich nun in jedem Kontext einsetzen, in welchem auch auf eine textuelle Beschreibung einer `Color` zurückgegriffen werden könnte. So kann bspw. bei der Erstellung von Benutzeroberflächen die textuelle Spezifikation der zu verwendenden Farben (s. Abbildung 2.9) durch eine visuelle unter Verwendung des von uns entwickelten Color-Blocks ersetzt werden (s. Abbildung 2.10).

Abbildung 2.8: Der Color-View ermöglicht es Farben darzustellen und in Text einzubetten.

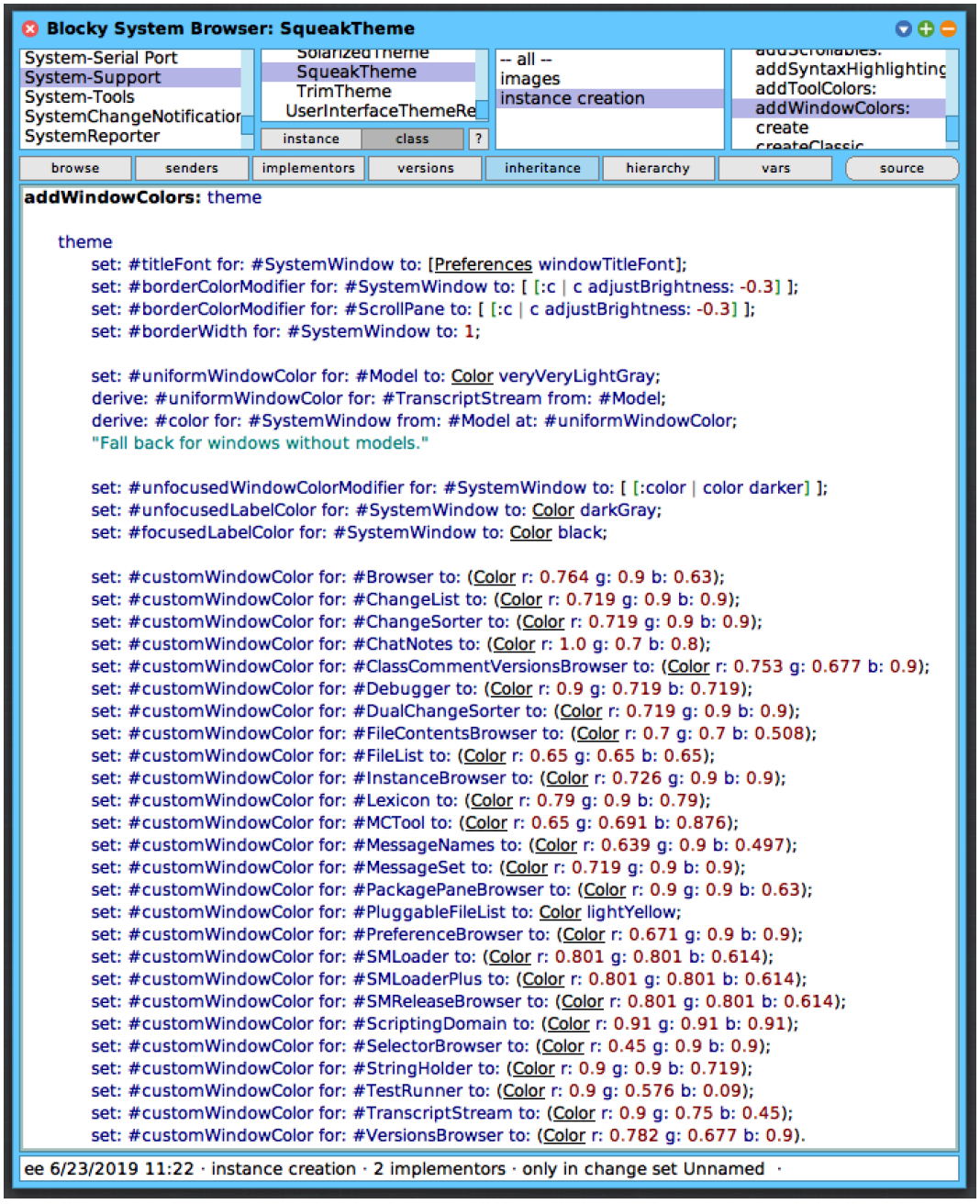


Abbildung 2.9: Klassenseitige Methode `#addWindowColors:` des `SqueakTheme`

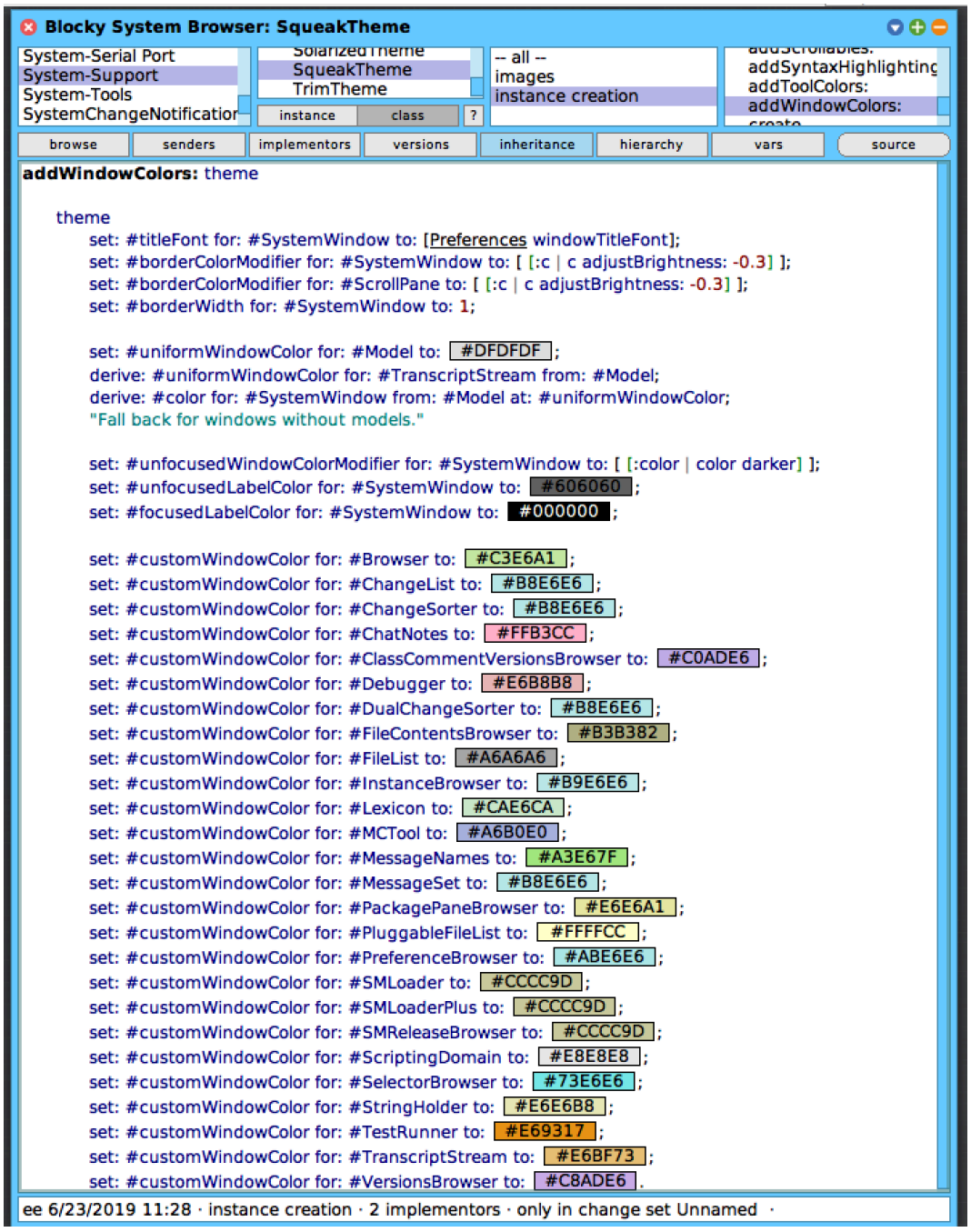


Abbildung 2.10: Klassenseitige Methode `#addWindowColors:` des `SqueakTheme` unter Verwendung von Color-Blöcken

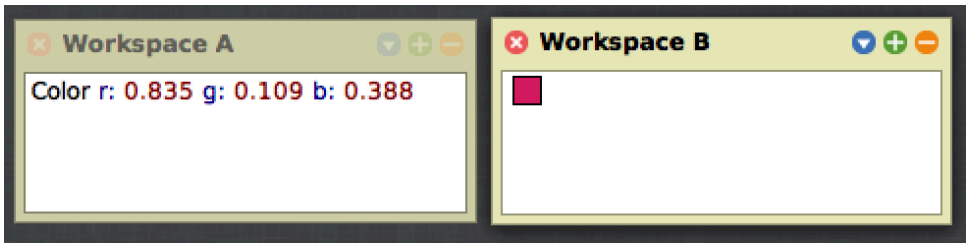


Abbildung 2.11: Links: Quelltext für eine Instanz von Color; rechts: alternative View des Color-Blocks.

2.5.2 Erstellung verschiedener Views

In Abschnitt 2.4.3 haben wir erklärt, dass ein Model durchaus verschiedene Views zur Verfügung stellen kann (vgl. Abschnitt 2.4.2). Bis jetzt sind wir jedoch nur auf die Verbindung eines Models mit einer einzigen View eingegangen. Diese View wurde als Rückgabewert der Nachricht `inlineView` spezifiziert (s. Abschnitt 2.5.1). Im Folgenden möchten wir erläutern, wie sich verschiedene Views mit einem Model verbinden lassen. Dies illustrieren wir anhand des in Abschnitt 2.5.1 eingeführten Color-Blocks.

Abbildung 2.11 zeigt eine alternative View eines Color-Blocks (s. Workspace B). Es handelt sich hierbei um eine schlichtere View als die in Abschnitt 2.5.1 vorgestellte. Die zu repräsentierende Farbe wird durch ein Quadrat dargestellt, welches entsprechend dieser Farbe gefärbt ist. Die Implementierung dieser alternativen Ansicht ist in Listing 2.4 zu sehen.

Listing 2.4: BTRSimpleColorMorph

```
1  BTRBlockMorph subclass: #BTRSimpleColorMorph
2     instanceVariableNames: ''
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'BlocksToTheRescue-Blocks'
6
7  BTRSimpleColorMorph >> #initialize
8     super initialize.
9     self extent: 15@15.
10    self color: self blockModel value.
11    self borderWidth: 1.
12    self borderColor: Color black.
13
14 BTRSimpleColorMorph >> #color
15    ^ self blockModel value
```

Der `TBTRBlockModel`-Trait bietet mit den beiden Methoden `inlineViewTypes` und `inlineViewType` einen Anknüpfungspunkt für die Verbindung mehrerer Views mit einem Model. Um diese Möglichkeit zu nutzen, ist es jedoch erforderlich, dass das Model über die Instanzvariable „viewData“, sowie den entsprechenden Akzessor `viewData` verfügt.

1. Die Methode `viewData` bietet Zugriff auf ein `Dictionary`. Wird der Trait eingebunden, so muss sowohl diese Nachricht überschrieben als auch die entsprechende Instanzvariable „viewData“ angelegt werden. Erbt das Model hingegen von `BTRAbstractBlockModel`, so ist dies obsolet (s. Listing 2.5).

2. Die Methode `inlineViewTypes` gibt eine `Collection` von Symbolen zurück. Die Symbole sind wiederum die Bezeichner von Methoden, welche Morphs zurückgeben müssen. Bei den Morphs handelt es sich dann um Instanzen der verschiedenen Views. Listing 2.6 zeigt die an der Klasse `BTRColorModel` vorgenommenen Änderungen.

   Das Color-Model verfügt nun über zwei Views (s. Listing 2.6, Z. 1 f., Z. 7–11). Mittels der Nachricht `inlineViewType` kann die aktuell ausgewählte View abgefragt werden. Standardmäßig handelt es sich bei dieser um die durch das erste Symbol der von `inlineViewTypes` zurückgegebenen Collection bezeichnete View. In unserem Beispiel übernimmt also automatisch eine Instanz der Klasse `BTRColorMorph` die Rolle der View.

Listing 2.5: BTRAbstractBlockModel

```
1 Object subclass: #BTRAbstractBlockModel
2    uses: TBTRBlockModel
3    instanceVariableNames: 'cachedStoreString originalText viewData'
4    classVariableNames: ''
5    poolDictionaries: ''
6    category: 'BlocksToTheRescue-Blocks'
7
8 BTRAbstractBlockMode >> #viewData
9    ^ viewData ifNil: [viewData := Dictionary new]
```

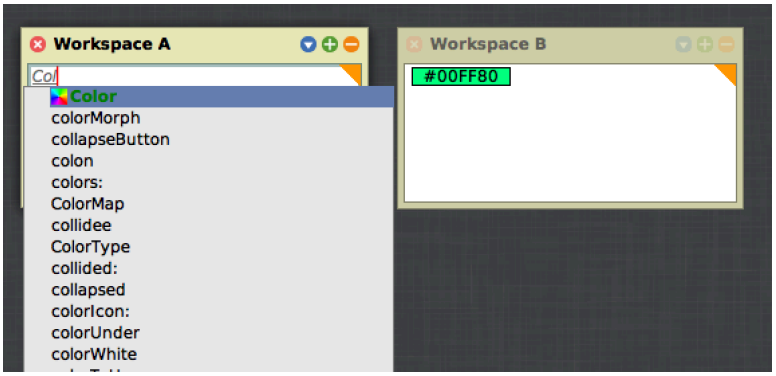
Listing 2.6: Angepasstes BTRColorModel

```
1 BTRColorModel >> #inlineViewTypes
2    ^ { #colorMorph . #simpleColorMorph }
3 BTRColorModel >> #colorMorph
4    ^ BTRColorMorph for: self
5 BTRColorModel >> #simpleColorMorph
6    ^ BTRSimpleColorMorph for: self
7 BTRColorModel >> #inlineView
8    ^ self perform: self inlineViewType
```

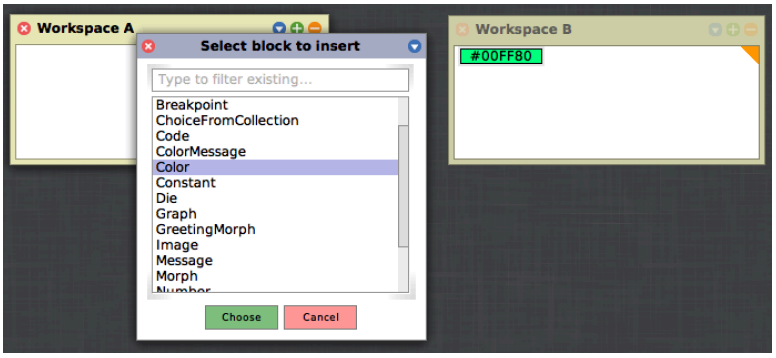
2.5.3 Interaktionsmöglichkeiten

In Textfelder (`StringHolder`) eingefügte Blöcke lassen sich mit den gleichen Tastenkombinationen wie Text kopieren, ausschneiden und löschen. Für das Einfügen selbst stehen sowohl visuelle als auch textuelle Optionen zur Verfügung:

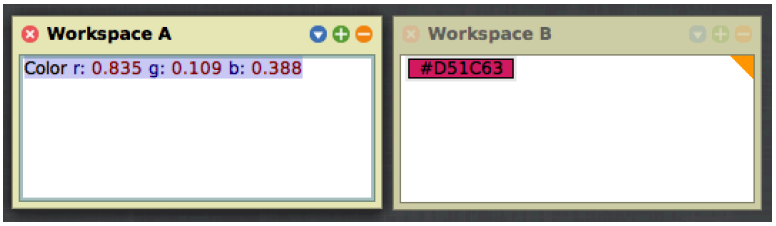
1. *Autovervollständigung.* Die Autovervollständigung von Squeak arbeitet mit einer Dropdown-Liste. Kommt für die Ergänzung einer Benutzereingabe ein Block in Frage, so wird dieser in der Liste von Vorschlägen aufgeführt. Um Block- von Textvorschlägen abzugrenzen, sind die ersteren stets mit einem Icon annotiert:



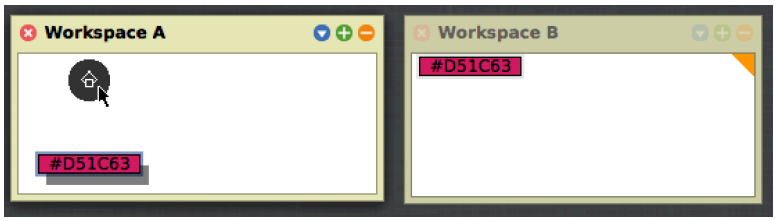
2. *Auswahlmenü.* Mit der Tastenkombination Ctrl + h lässt sich ein Auswahlmenü öffnen. Dieses listet alle momentan verfügbaren Blocktypen auf:



3. *Textersetzung.* Die textuelle Beschreibung eines Programmelements kann durch Markieren und anschließende Betätigung der Tastenkombination Ctrl + h in einen Block umgewandelt werden. Steht kein spezifischer Block zur Verfügung, so wird der Text durch einen Inspector ersetzt:



4. *Drag-and-drop.* Sämtliche Morphs in der Welt lassen sich per Drag-and-drop bei gleichzeitigem Drücken der Shift -Taste in ein Textfeld einfügen. Da Morphs ohnehin mittels Drag-and-drop in der Welt bewegt werden können, haben wir uns entschieden, zum Einfügen in Textfelder zusätzlich die Betätigung der Shift -Taste zu fordern. So soll vermieden werden, dass Morphs versehentlich in Textfelder eingefügt werden. Die Drag-and-drop-Technik kann neben dem Einfügen von Blöcken auch für das Verschieben in und das Entfernen aus Textfeldern verwendet werden:



Um einen Block weiter untersuchen und bearbeiten zu können, steht ein Block-Menü zur Verfügung. Dieses lässt sich mittels eines Klicks mit der rechten Maustaste auf den entsprechenden Block öffnen. Das Block-Menü ermöglicht unter anderem, View und Model zu inspizieren sowie zwischen verschiedenen Views eines Models zu wechseln (s. Abbildung 2.12).

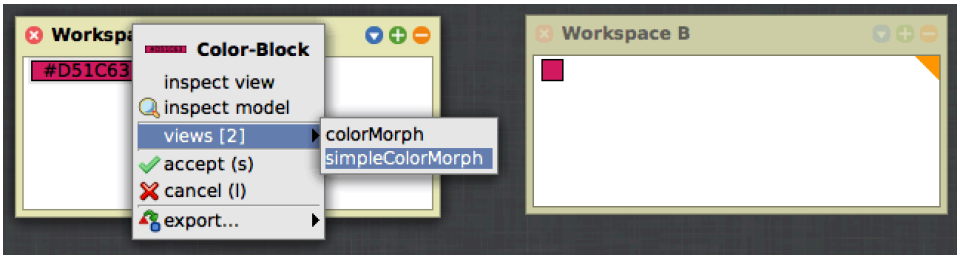


Abbildung 2.12: Das Block-Menü ermöglicht die Auswahl von Views und weitere Bearbeitung.

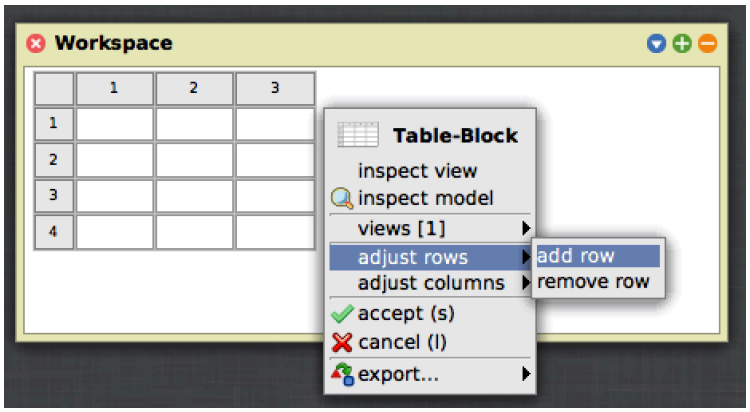


Abbildung 2.13: Das Block-Menü im Table-Block ermöglicht das Verändern von Zeilen und Spalten der Tabelle.

Auch ist es möglich, das Block-Menü an den jeweiligen Block anzupassen und mit weiteren Funktionen auszustatten. Dazu ist es lediglich nötig, dass die Block-View zwei bestimmte Methoden überschreibt:

1. Die Methode `wantsYellowButtonMenu` muss immer `true` zurückgeben.
2. Die Methode `addNestedYellowButtonItemsTo:event:` bekommt zwei Parameter. Der erste Parameter ist ein Menü, der zweite ein `Event`. Das übergebene Menü kann nun nach Belieben angepasst werden. Es ist kein expliziter Rückgabewert erforderlich.

So kann zum Beispiel ein Table-Block das Menü so anpassen, dass über dieses die Zeilen und Spalten der entsprechenden Tabelle bearbeitet werden können (s. Abbildung 2.13).

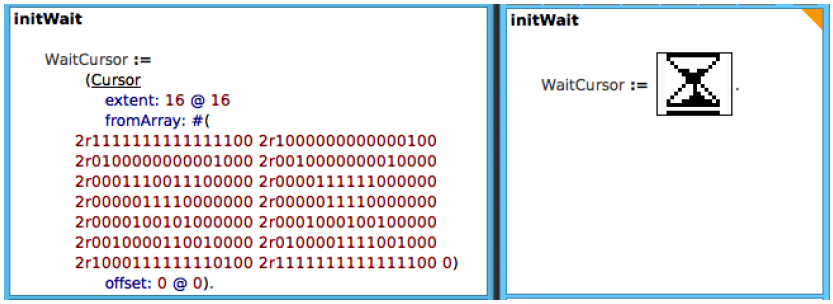
2.5.4 Anwendungsbeispiele

Neben den bereits diskutierten Objekten für Farben (`Color`), gibt es weitere Beispiele für allgemein nützliche Blocktypen. Innerhalb des Morphic-Frameworks kann jeder Morph einen Blocktyp darstellen, da unsere Views als Morphs umgesetzt wurden. Ein Beispiel ohne Selbstdarstellung ist der Blocktyp für Wahrheitswerte (`Boolean`), welcher hier in der Methode `Morph >> #wantsSteps` verwendet wird:

```
wantsSteps
    "Return true if the receiver overrides the default Morph step method."
    "Details: Find first class in superclass chain that implements #step and
     return true if it isn't class Morph."

    | c |
    self isPartsDonor ifTrue: [^ false ].
    (self == self topRendererOrSelf) ifTrue: [self player wantsSteps ifTrue: [^ true ]].
    c := self class.
    [c includesSelector: #step] whileFalse: [c := c superclass].
    ^ c ~= Morph
```

Ein Beispiel mit Selbstdarstellung – ähnlich zu Morphs – ist der Blocktyp für Bilder (`Form`), welcher hier statt der textuellen Darstellung einer Ressource in der Methode `Cursor class >> #initWait` verwendet wird:



Datenstrukturen wie Listen, Tabellen und Matrizen können durch entsprechende Blöcke repräsentiert werden:

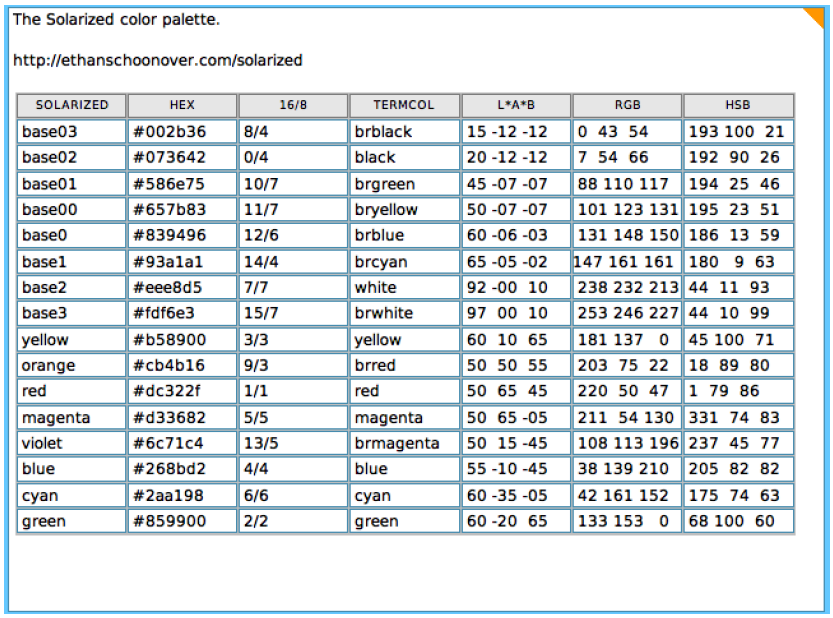
Dabei lassen sich Blöcke nicht nur im Programmcode selbst, sondern auch in Klassenkommentaren einsetzen. Folgender Kommentar der Klasse `SolarizedTheme` dient als Beispiel:

```
The Solarized color palette.

http://ethanschoonover.com/solarized

SOLARIZED HEX 16/8 TERMCOL XTERM/HEX L*A*B RGB HSB
--------- ------- ---- ------- ----------- ---------- ----------- -----------
base03 #002b36 8/4 brblack 234 #1c1c1c 15 -12 -12 0 43 54 193 100 21
base02 #073642 0/4 black 235 #262626 20 -12 -12 7 54 66 192 90 26
base01 #586e75 10/7 brgreen 240 #585858 45 -07 -07 88 110 117 194 25 46
base00 #657b83 11/7 bryellow 241 #626262 50 -07 -07 101 123 131 195 23 51
base0 #839496 12/6 brblue 244 #808080 60 -06 -03 131 148 150 186 13 59
base1 #93a1a1 14/4 brcyan 245 #8a8a8a 65 -05 -02 147 161 161 180 9 63
base2 #eee8d5 7/7 white 254 #e4e4e4 92 -00 10 238 232 213 44 11 93
base3 #fdf6e3 15/7 brwhite 230 #ffffd7 97 00 10 253 246 227 44 10 99
yellow #b58900 3/3 yellow 136 #af8700 60 10 65 181 137 0 45 100 71
orange #cb4b16 9/3 brred 166 #d75f00 50 50 55 203 75 22 18 89 80
red #dc322f 1/1 red 160 #d70000 50 65 45 220 50 47 1 79 86
magenta #d33682 5/5 magenta 125 #af005f 50 65 -05 211 54 130 331 74 83
violet #6c71c4 13/5 brmagenta 61 #5f5faf 50 15 -45 108 113 196 237 45 77
blue #268bd2 4/4 blue 33 #0087ff 55 -10 -45 38 139 210 205 82 82
cyan #2aa198 6/6 cyan 37 #00afaf 60 -35 -05 42 161 152 175 74 63
green #859900 2/2 green 64 #5f8700 60 -20 65 133 153 0 68 100 60
```

Daraus lässt sich übersichtlich eine tabellarische Repräsentation erstellen:



The Solarized color palette.

http://ethanschoonover.com/solarized

SOLARIZED	HEX	16/8	TERMCOL	L*A*B	RGB	HSB
base03	#002b36	8/4	brblack	15 -12 -12	0 43 54	193 100 21
base02	#073642	0/4	black	20 -12 -12	7 54 66	192 90 26
base01	#586e75	10/7	brgreen	45 -07 -07	88 110 117	194 25 46
base00	#657b83	11/7	bryellow	50 -07 -07	101 123 131	195 23 51
base0	#839496	12/6	brblue	60 -06 -03	131 148 150	186 13 59
base1	#93a1a1	14/4	brcyan	65 -05 -02	147 161 161	180 9 63
base2	#eee8d5	7/7	white	92 -00 10	238 232 213	44 11 93
base3	#fdf6e3	15/7	brwhite	97 00 10	253 246 227	44 10 99
yellow	#b58900	3/3	yellow	60 10 65	181 137 0	45 100 71
orange	#cb4b16	9/3	brred	50 50 55	203 75 22	18 89 80
red	#dc322f	1/1	red	50 65 45	220 50 47	1 79 86
magenta	#d33682	5/5	magenta	50 65 -05	211 54 130	331 74 83
violet	#6c71c4	13/5	brmagenta	50 15 -45	108 113 196	237 45 77
blue	#268bd2	4/4	blue	55 -10 -45	38 139 210	205 82 82
cyan	#2aa198	6/6	cyan	60 -35 -05	42 161 152	175 74 63
green	#859900	2/2	green	60 -20 65	133 153 0	68 100 60

Für komplexere Anwendungsbeispiele mit Blöcken sei auf Kapitel 7 und Kapitel 6 verwiesen. Auch die Ideen von Erwig und Meyer [36] zur Erweiterung der Programmiersprache Prolog um eine visuelle Subsprache von Zustandsautomaten [36, S. 319] sowie der Sprache C um grafische Pointer-Elemente [36, S. 323 f.] können der Illustration dienen.

2.6 Evaluation

Im Laufe dieses Kapitels haben wir in Abschnitt 2.2 zunächst theoretisch unser Konzept heterogener Programmierung entwickelt (s. insbes. Abschnitt 2.3.4). Dabei sind wir auf historische und gegenwärtige Entwicklungen eingegangen, um Anknüpfungspunkte zu verdeutlichen. Daran anschließend haben wir in Abschnitt 2.4 eine beispielhafte praktische Umsetzung des zuvor theoretisch besprochenen Konzepts heterogener Programmierung in der Programmierumgebung Squeak vorgestellt.

In diesem letzten Abschnitt möchten wir nun die von uns dargestellte Art der Einbindung semantisch bedeutungsvoller visueller Elemente in eine etablierte textuelle Programmiersprache evaluieren sowie auf weitere Forschungsfragen hinweisen.

Die beiden Fragen, die sich uns bei der Evaluation stellen, sind die folgenden:

1. Treten Vorteile visueller Programmierung trotz der Beibehaltung von Text ein?

2. Können Nachteile visueller Programmierung aufgrund der Beibehaltung von Text vermieden werden?

Für eine ausführliche Beschreibung der Vor- und Nachteile visueller Programmierung an sich sei auf die Arbeiten von Green und Petre [53], Schiffer [96] und Whitley [115] verwiesen.

Studien der letzten Jahre, wie die von Bau u. a. [12], Noone und Mooney [82], Weintrop und Wilensky [112] oder Alrubaye u. a. [4], in denen unter anderem auch heterogene Ansätze besprochen werden, konzentrieren sich primär auf die Untersuchung der Nützlichkeit dieser für den Bildungsbereich. Unser Konzept heterogener Programmierung orientiert sich jedoch nicht an dem Einsatz in der Lehre, weshalb wir die dort gewonnen Erkenntnisse anders gewichten müssen.

Im Folgenden wollen wir einige uns wichtig erscheinende Vor- und Nachteile visueller Programmierung aufgreifen und diese mit unserer Vorstellung heterogener Programmierung in Zusammenhang bringen.

2.6.1 Vorteile visueller Elemente

2.6.1.1 Strukturierung

Whitley [115] bemerkt, dass Menschen der Umgang mit Informationen leichter falle, wenn diese in einer konsistenten und strukturierten Form vorliegen. Zudem würden gerade solche Repräsentationen als effizient empfunden, welche Informationen explizit darstellen. Diese beiden Feststellungen treffen sowohl auf textuelle als auch auf visuelle Notationen zu. Studien wie die von Day [31], Schwartz [98] oder Carroll u. a. [24] legen jedoch nahe, dass visuelle Repräsentationen eine bessere Organisation erlauben und Informationen expliziter dargestellt werden können. Wir gehen davon aus, dass dies in direktem Zusammenhang mit dem Einsatz sekundärer Notation sowie dem Gestalt-Effekt [83] steht.

2.6.1.2 Sekundäre Notation

Vergleichen wir in Abschnitt 2.5.4 die Abbildung der Tabelle mit ihrer textuellen Variante, so müssen wir eingestehen, dass wir für die heterogene Repräsentation mehr Zeichen verwenden als für die rein textuelle, um den gleichen Sachverhalt darzustellen. Bei diesen zusätzlichen Zeichen handelt es sich um das Gitter, sowie die farbliche Hervorhebung der ersten Zeile des Table-Blocks, also um Elemente, welche die Daten selbst nicht beeinflussen, dem Leser jedoch visuelle Hinweise bezüglich ihrer Bedeutung geben. Solche das Layout oder auch die Typografie betreffenden Zeichen sind Teil einer sekundären Notation.

Auch im Zusammenhang mit textuellen Programmiersprachen spielt sekundäre Notation eine Rolle. Doch sind hier die Möglichkeiten aufgrund der symbolischen Natur von Buchstaben sowie dem linearen Charakter von Text (s. Abschnitt 2.2.2) deutlich geringer als in visuellen Programmiersprachen.

Green und Petre [52] weisen darauf hin, dass durch eine Beschränkung zugleich auch Fehler vermieden werden können und Einsteiger nicht in der Lage seien, sekundäre Notation optimal einzusetzen. Dem halten wir zum einen entgegen, dass wir uns nicht auf Einsteiger konzentrieren und in visuellen Elementen folglich nicht nur ein Mittel sehen, den Einstieg zu erleichtern, sondern eines, welches dauerhaft Vorteile schaffen soll. Zum anderen verhält es sich ganz im Allgemeinen so, dass

mit der Anzahl der Chancen einer Technik auch die ihrer Risiken wächst. Wir sehen in der Sorge um Gefahren jedoch nicht die Rechtfertigung für die komplette Einschränkung aller Möglichkeiten.

2.6.1.3 Gestalt-Effekt

Petre [83] führt zur Beschreibung der Stärken visueller Notationen den Begriff der *Gestalt* bzw. des *Gestalt-Effekts* ein. Vergleichen wir die visuelle Darstellung einer Farbe durch einen Color-Block (s. Abbildung 2.8) mit der textuellen durch einen RGB-Wert (s. Abbildung 2.7) oder auch einen Bezeichner (z. B. `Color red`), so fällt auf, dass sich die textuelle Repräsentation auf eine spezifische Eigenschaft der Farbe bezieht, die visuelle jedoch einen Gesamteindruck der Farbe vermittelt.

Petre [83] ist der Überzeugung, dass diese Vermittlung der Gestalt, der Gestalt-Effekt, daher rührt, dass sich visuelle Elemente auf einem höheren Abstraktionsniveau als textuelle befinden. Bezieht man das Abstraktionslevel auf die Entfernung von den Eigenheiten der Programmiersprache, also des Darstellungsmittels, so hat Petre sicherlich recht. Bringt man den Grad der Abstraktion jedoch mit dem darzustellenden Objekt in Verbindung, so erscheint es, dass textuelle Repräsentationen stärker abstrahieren, da sie das Darzustellende in größerem Maße reduzieren und sich von ihm entfernen. Visuelle Notationen hingegen bleiben näher am Subjekt und sind somit auch eher in der Lage dessen Gesamteindruck, dessen Gestalt wiederzugeben. Der Gestalt-Effekt bewirkt nun, dass visuelle Notationen in der Regel eine größere Fülle an Informationen liefern, wobei sie jedoch nicht im gleichen Verhältnis ein Mehr an Platz beanspruchen als Text. Die Kluft zwischen Abbildung und Original ist bei den visuellen Elementen geringer und somit erscheinen sie uns intuitiver, verständlicher und leichter zugänglich.

2.6.1.4 Zugänglichkeit

Baroth und Hartsough [10] sehen einen Vorteil visueller Programmiersprachen in ihrer Zugänglichkeit für Nicht-Programmierer. Sie beschreiben eine kollaborative Art des Programmierens, bei der Endnutzer und Programmierer eng zusammenarbeiten. Interessant ist hier die Rolle der Nicht-Programmierer und die sich daraus ergebende Bedeutung visueller Elemente.

Im Allgemeinen wird das Argument der Zugänglichkeit visueller Programmiersprachen in Verbindung mit deren Eignung für die Lehre angeführt. Visuelle Programmierung wird als starkes Konzept für die Einführung in die Programmierung betrachtet [82] und soll Programmier-Einsteiger sowie nicht-professionelle Programmierer dazu befähigen, eigene Anwendungen zu entwickeln [4, 12, 112].

Dieser erleichterte Einstieg in die Programmierung muss jedoch nicht automatisch als Einstieg in die *Tätigkeit* des Programmierens gedeutet werden. Er kann auch als Zugang zum *Verständnis* von Programmen angesehen werden. Dann handelt es sich bei den von Baroth und Hartsough [10] identifizierten Nicht-Programmierern nicht um Menschen, welche an dem Programmierprozess interessiert sind, sondern um solche, welche die dargestellten Inhalte verstehen wollen. Zu dieser zweiten Art von Nicht-Programmierern gehören bspw. Kunden und Auftraggeber von Softwareprojekten. Die Zugänglichkeit visueller Elemente erlaubt dann,

dass der Austausch zwischen Entwicklern und Kunden über das zu erstellende Produkt, den Programmcode, näher an diesem Produkt stattfinden kann.

Diese Sichtweise auf die Rolle von Nicht-Programmieren als Leser im Unterschied zu Verfassern von Programmcode, hat auch Auswirkungen auf die Konzeption visueller Elemente. Im Bildungsbereich werden visuelle Programmiersprachen oft so gestaltet, dass sie die Vermeidung von Syntaxfehlern unterstützen [82] oder die semantische Verwandtschaft zwischen Programmstrukturen hervorheben (Scratch [74], Snap! [54], Blockly[4]). Das heißt, viele Entwurfsentscheidungen orientieren sich an Programmkonzepten statt an Problemdomänen. Betrachtet man nun jedoch Programmierer, welche syntaktische und semantische Konzepte beherrschen, und Nicht-Programmierer, welche am Verstehen und nicht Entwickeln von Programmcode interessiert sind, als die Zielgruppe visueller Notationen, so kann man den Fokus bei der Gestaltung visueller Programmiersprachen verändern: weg vom Programm und hin zu dem darzustellenden Sachverhalt.

2.6.2 Nachteile visueller Elemente

2.6.2.1 Deutsch Limit

L. Peter Deutsch erklärte im Rahmen einer von Scott Kim und Warren Robinett organisierten Diskussionsrunde, dass das Problem an visuellen Programmiersprachen das sei, dass sich nicht mehr als 50 visuelle Primitive auf einmal auf einem Bildschirm anzeigen ließen [78]. Fred Lakin prägte daraufhin den Begriff des *Deutsch Limit*, welcher sich auf die relativ geringe Bildschirmdichte visueller Notationen im Vergleich zu textuellen bezieht [115]. Trotz Studien zur Untersuchung platzsparender visueller Notationen wie der von Ramsey u. a. [87] beanspruchen textuelle Elemente nach wie vor deutlich weniger Raum als visuelle.

Wir wollen keineswegs abstreiten, dass ein einzelnes Wort oft weniger Platz einnimmt als ein visuelles Element. Doch gilt es zu hinterfragen, ob ein Wort ausreicht, um den durch ein einziges visuelles Element ausgedrückten Sachverhalt zu beschreiben. Betrachten wir Abbildung 2.11 oder die Beispiele in Abschnitt 2.5.4, so scheint die These des Deutsch Limit nicht auf alle Anwendungsfälle zuzutreffen. Durch unseren heterogenen Ansatz kann nun aber stets auf die kompaktere Darstellungsform zurückgegriffen werden und es lässt sich somit sogar eine effizientere Bildschirmausnutzung als mit einer rein textuellen Notation erreichen.

2.6.2.2 Match-Mismatch-Hypothese

Gilmore und Green [45] stellten die sogenannte *Match-Mismatch-Hypothese* auf, welche besagt, dass jede Notation gewisse Aspekte von Informationen hervorhebt, wohingegen sie andere verschleiert. Wird bspw. der Datenfluss hervorgehoben, so kann darunter die Darstellung des Kontrollflusses leiden; werden die Bedingungen für die Ausführung von Aktionen unterstrichen, tritt deren sequenzielle Reihenfolge womöglich weniger stark hervor. Gilmore und Green folgerten, dass keine

[4] Blockly: `https://developers.google.com/blockly/`, letzter Zugriff am 16. Juni 2019.

Darstellungsart allgemein als *gut* oder *schlecht* zu bezeichnen sei, sondern deren Güte sich stets danach richte, wofür und von wem sie eingesetzt würde [53].

Die Studien von Hendry und Green [55], Moher u. a. [79] und Wright und Reid [116] werfen nun die Frage auf, ob visuelle Programmiersprachen ein ausreichend großes Spektrum an unterschiedlichen Informationen betonen können, um von praktischem Interesse zu sein. Die in Abschnitt 2.3.2 geschilderten Skalierungsschwierigkeiten und Spezifizierungsmaßnahmen scheinen die in der Frage mitschwingenden Zweifel zu rechtfertigen. Als Stärke visueller Programmiersprachen wird deren Nähe zum Darzustellenden gesehen. Um diese Stärke jedoch nutzen zu können, entstehen zumeist domänenspezifische Ansätze. Daraus erwächst die Überzeugung, dass visuelle Programmiersprachen weniger flexibel und beschränkter in ihrem Anwendungsbereich sind als textuelle Programmiersprachen. Nickerson [81] macht zudem darauf aufmerksam, dass es für gewisse Konstrukte wie Rekursion oder Selbstbezüglichkeit keine befriedigenden visuellen Notationen gebe.

Um diesen Problemen zu begegnen, haben wir uns für ein heterogenes Konzept entschieden, bei welchem der Entwurf und Einsatz visueller Elemente dem Nutzer selbst obliegt. Dieser wird somit nicht durch im Voraus getroffene Gestaltungsentscheidungen in seinen Ausdrucksmöglichkeiten eingeschränkt. Es ist so möglich, die Erscheinung visueller Elemente stark an eine Domäne, einen Anwendungsfall oder auch die persönlichen Präferenzen des Nutzers anzupassen, ohne dadurch die ganze Sprache einzuschränken.

2.6.2.3 Unschärfe

Aufgrund des Gestalt-Effekts (s. Abschnitt 2.6.1) gewähren visuelle Elemente einen Gesamteindruck des Darzustellenden. Petre [83] weist jedoch darauf hin, dass der Erfolg einer Repräsentation nicht davon abhängt, möglichst viele Informationen zur Verfügung zu stellen, sondern davon, die notwendigen in einer klaren und übersichtlichen Art zu präsentieren.

Er charakterisiert weiterhin visuelle Programmiersprachen als analoge Repräsentationen und textuelle Programmiersprachen als deskriptive.

Deskriptive Repräsentationen sind dem Wortsinn gemäß *beschreibender* Natur. Sie beruhen auf einem – vergleichsweise kleinen – abgeschlossenen Zeichensatz. Ausdrücke lassen sich durch eine geregelte Kombination dieser Zeichen bilden. Ihre Interpretation folgt festgelegten Regeln.

Analoge Repräsentationen *spiegeln wider*. Sie bieten ein eher offenes Vokabular und haben somit eine weniger feste Grundlage. Analoge Notationen können von dem Gestalt-Effekt profitieren, welcher einen Eindruck von dem Ganzen sowie Einblicke in die Gesamtstruktur erlaubt. Der Gestalt-Effekt führt jedoch auch zu einem Verlust an Präzision, da ein Großteil der Informationen implizit in der analogen Abbildung enthalten ist. Die Interpretationsregeln sind weniger klar als bei deskriptiven Notationen und leiden somit unter einer größeren Unschärfe.

Das von uns vorgestellte heterogene Konzept eröffnet mit den Konstruktionsmöglichkeiten eigener visueller Elemente eine theoretisch unbeschränkte Erweiterung des Alphabets der zugrunde liegenden textuellen Sprache. Dies bewirkt nun aber die von Petre [83] beschriebenen Ungewissheiten und Uneindeutigkeiten. Betrach-

ten wir etwa die beiden vorgestellten Views eines Color-Blocks (s. Abschnitt 2.6.1 und Abschnitt 2.5.2), so unterscheiden sich diese nicht nur in ihrem Äußeren, sondern auch in Bezug auf ihren Funktionsumfang.

2.6.2.4 Aufwand

Visuelle Programmiersprachen wie Scratch [74], Snap! [54] oder Blockly[5] sind oft in einem deutlich größeren Maße an eine spezifische Entwicklungsumgebung gebunden als dies bei textuellen Programmiersprachen wie Java, C oder Python der Fall ist [61].

Dies hängt unter anderem damit zusammen, dass bei textuellen Sprachen in der Regel eine übliche Tastatur als Eingabemedium genügt, visuelle Sprachen jedoch zur Erzeugung visueller Elemente auf andere Mittel zurückgreifen müssen. Entwicklungsumgebungen visueller Sprachen weisen zu diesem Zweck Seitenleisten oder spezielle Menüs auf, welche die verfügbaren visuellen Elemente anzeigen. Diese können dann par Drag-and-drop-Technik in Programme eingefügt werden (s. Abbildung 2.3).

Die erforderlichen Auswahlmenüs verschärfen zum einen das Problem des Deutsch Limit. Zum anderen sind sie aber auch umständlicher und langsamer zu bedienen als eine Tastatur [59].

In Abschnitt 2.5.3 haben wir verschiedene Eingabe- und Bearbeitungsmöglichkeiten für Blöcke vorgestellt. Diese reichten von rein textuellen bis zu rein visuellen Techniken. Bei der Konzeption des SandBlocks-Frameworks war es uns wichtig, dass alle textuellen Bearbeitungsmechanismen auch für Blöcke zur Verfügung stehen. Die Interaktionsmöglichkeiten im Umgang mit Blöcken sollten eine Obermenge der textuellen darstellen.

Obwohl es uns scheint, das Problem des erhöhten Aufwands bei der Interaktion überwunden zu haben, treffen wir doch auf einen gewissen Mehraufwand bei der Erstellung von Blöcken. Wir wollten kein fest beschränktes Repertoire an Blöcken zur Verfügung stellen, sondern dem Nutzer die eigenständige Erstellung und freie Gestaltung von Blöcken ermöglichen. Dies hat zur Folge, dass dem Nutzer jedoch auch der Aufwand für die Erstellung obliegt. Einmal entwickelt, können Blöcke jedoch auch anderen Anwendern zur Verfügung gestellt werden (s. Kapitel 5), so dass mit der Zeit Block-Packages bzw. -Bibliotheken entstehen, die stets erweiterbar sind.

[5] Blockly: `https://developers.google.com/blockly/`, letzter Zugriff am 16. Juni 2019.

2.7 Zusammenfassung und Ausblick

> *„Data is not information. Data must be presented in a usable form before it becomes information, and the choice of representation affects the usability. But usability is not simply ‚better' or ‚worse'; how good a representation is depends on what you want to use it for."*
>
> – Green und Petre [53]

Wir sind der Überzeugung, dass die Einführung semantisch bedeutungsvoller visueller Elemente in eine textuelle Programmiersprache diese bereichert. Durch den heterogenen Ansatz erübrigt sich die Frage, ob visueller oder textueller Programmierung der Vorzug zu geben sei, da durch ihn die Ausschließlichkeit dieser beiden Konzepte aufgehoben wird.

Den Vorteil heterogener Programmierung sehen wir nicht so sehr in der Unterstützung der Tätigkeit des Programmierens, sondern vielmehr in der Förderung des Verstehens von und der Kommunikation über Programme. Tatsächlich werden in der Praxis gerade visuelle Artefakte für die Dokumentation von und den Austausch über Software-Systeme eingesetzt. Doch handelt es sich bei diesen visuellen Mitteln nicht um ausführbaren Programmcode, sondern um Diagramme, welche von diesem Code getrennt sind. Wir glauben nun, dass durch den Ansatz heterogener Programmierung diese Kluft zwischen Dokumentation und Beschreibung auf der einen und einsetzbarem Produkt auf der anderen Seite überwunden werden kann.

Durch die Vereinigung von veranschaulichendem Material und Veranschaulichtem wird es zudem möglich, das Problem der Konsistenz zu umgehen. So besteht bei von dem Programmcode getrennten visuellen Diagrammen stets die Gefahr, dass diese nicht an Änderungen der Software angepasst werden und somit einen veralteten Zustand widerspiegeln. Wenn aber die beschreibenden Dokumente stets aktualisiert werden, hat jeder Änderungswunsch einen doppelten Änderungsaufwand zur Folge, da neben dem Produkt als solchem, auch alle Diagramme modifiziert werden müssen.

Wir denken nun nicht, dass durch heterogene Programmiersprachen jegliche Notwendigkeit zur Dokumentation entfällt. Doch sind wir der Meinung, dass die Kombination von visuellen und textuellen Elementen übersichtlichere und strukturiertere Programme erlaubt und somit die Menge an zusätzlichen Beschreibungen reduziert werden.

Der Mangel an Einheitlichkeit visueller Elemente scheint deren größte Schwäche darzustellen. Wie bereits in Abschnitt 2.6.2 erläutert, kann der sehr große Spielraum bei der Gestaltung visueller Elemente zu irregeführten Erwartungen und voneinander abweichenden Interpretationen führen.

Zum einen könnte diesem Problem entgegengetreten werden, indem sich, wie in der Modellierung mit der *Unified Modelling Language* (*UML*), eine standardisierte Sprache für visuelle Notationen herausbildet.

Zum anderen sehen wir jedoch gerade in dieser Offenheit der Gestaltung eine große Stärke visueller Elemente. Durch diese ist es möglich, sehr spezifische

Notationen zu schaffen, welche auf den jeweiligen Kontext der zu entwickelnden Anwendung abgestimmt sind, und gleichzeitig durch den textuellen Rahmen die Vorteile einer allgemein einsetzbaren Sprache zu wahren.

So bewerten wir die gelöstere Form visueller Elemente nicht als Schwäche, sondern sehen in ihr eher ein Risiko, welches zugleich eine Chance in sich birgt.

Laut Green und Petre [53] muss bei der Bewertung der Güte einer Repräsentation ihr Einsatzgebiet betrachtet werden. Wir glauben nun, dass durch die Integration visueller Elemente in eine textuelle Sprache, das Einsatzgebiet der Sprache erweitert werden kann, indem ihre Fähigkeit zur Spezifizierung erhöht und gleichzeit ihr allgemein einsetzbarer Charakter erhalten wird.

Ausblick

Um den hier vorgestellten Ansatz heterogener Programmierung weiter ausbauen und evaluieren zu können, bedarf es gewisser Studien und Erfahrungsberichte. So haben wir uns bei der Diskussion der Vor- und Nachteile visueller Elemente primär auf Arbeiten zu rein visuellen Programmiersprachen bzw. dem in Abschnitt 2.3.3 und Abschnitt 2.3.4 beschriebenen Konzept heterogener Programmierung im Bildungsbereich gestützt. Wir wollen diesen keineswegs ihre Gültigkeit absprechen, doch denken wir, dass spezifischere Erkenntnisse nur durch eine gezielte Untersuchung des von uns angestrebten alternativen Ansatzes heterogener Programmierung gewonnen werden können.

Es wäre im nächsten Schritt erforderlich, weitere textuelle Programmiersprachen, um semantisch bedeutungsvolle visuelle Elemente zu erweitern. Ansonsten besteht die Gefahr, die Besonderheiten des jeweilig zugrunde liegenden Programmiersystems, wie in unserem Fall Squeak/Smalltalk, mit den Eigenheiten heterogener Programmierung zu verwechseln.

Auch wären Studien zum Vergleich visueller, heterogener und textueller Programmierung interessant. Solche wurden bspw. von Lewis [70] oder Smith u. a. [104] bereits im Bildungsbereich angestellt. Es wäre aufschlussreich diese im Feld der professionellen Softwareentwicklung vorzunehmen.

Die Untersuchung der kognitiven Verarbeitungsunterschiede zwischen visuell und textuell präsentierten Informationen im Feld der Programmierung, wie sie in Abschnitt 2.2.1 angedeutet wurden, könnten weitere Erkenntnisse liefern. Hier würde sich bspw. das von Green und Petre [53] vorgestellte System zur Bewertung kognitiver Dimensionen als Anknüpfungspunkt anbieten.

3 SandBlocks: Integration grafischer, ausführbarer Objekte in die Live-Programmierumgebung Squeak/Smalltalk

Als Kombination von textuellen und grafischen Programmierkonzepten setzt das SandBlocks-Projekt eine Kombination aus visuellen und textuellen Programmiersprachen in grafischen, ausführbaren Objekten um. Die Herausforderung besteht darin, diese sogenannten Blöcke in die Entwicklungsumgebung Squeak der objektorientierten Sprache Smalltalk sowie deren Werkzeuge zu integrieren.

In diesem Kapitel werden die Voraussetzungen dafür geschaffen, solche Blöcke als alternative Abstraktion zu Methoden zu verwenden, als auch anstelle von Smalltalk-Syntax in Text einzubauen. Dazu wurde der Texteditor, als auch die Textdarstellung an die gewählte Block-Implementierung angepasst und dokumentiert.

Als Ergebnis dieses Projektes können einem domänenspezifischen Entwickler Blöcke seiner Domäne zur Verfügung gestellt werden, um die Arbeit an seinen Programmen zu vereinfachen, als auch für jeden Squeak-Entwickler das SandBlocks-Programmgerüst, in welchem neue Blöcke entwickelt werden können.

3.1 Einleitung

Schon seit Beginn der Programmierung an Heimrechnern gab es Überlegungen, wie man Programmtext vereinfachen und für den Programmierer anschaulicher machen kann. Diese Abstraktion hatte aber lange Zeit nur die Folge, dass Abstraktion der Sprache die textuelle Syntax veränderte. Die Nutzung von konsolenbasierten Texteditoren[1] ermöglichte das Schreiben von beliebigem Programmcode; grafische Strukturen mussten mit Hilfe des gegebenen Zeichensatzes dargestellt werden. Später, mit Aufkommen von grafischen Benutzerschnittstellen und auch integrierten Entwicklungsumgebungen, wurden dann vermehrt Werkzeuge entwickelt, die direkt auf bestimmte Programmiersprachen zugeschnitten wurden, um den Programmcode zu strukturieren und zu modifizieren.

Dabei gab es schon mehrere Überlegungen, dass man Programme auch grafisch, also visuell darstellen könnte. Parallelen wurden gezogen, wie zum Beispiel bei Architekturentscheidungen in der Programmierung Modelle[2] erstellt wurden, welche

[1] Einer der ersten Texteditoren war beispielsweise O26 für den CDC-6000-Großrechner von 1967 [102]; andere Editoren wären der POSIX-standardisierte Zeileneditor ed (1973) bzw. später ex oder der daraus entstandene konsolenbasierte Editor vi für Unix aus dem Jahr 1976.

[2] Heutzutage würden hier beispielsweise UML-Diagramme [95] angewendet werden.

zu Programmcode konvertiert werden konnten. Diese formale Programmierung ist auch unter dem Begriff der modellgetriebenen Entwicklung (engl. *model-driven development*, MDE) bekannt. Ungefähr seit den 1950er Jahren konnten darauf aufbauend viele visuelle Programmiersprachen (engl. *visual programming languages*, VPL) entstehen, welche den Algorithmus oder das Systemverhalten durch grafische Elemente und deren Anordnung definierten. Bekannt geworden sind damit die bildungsorientierten Sprachen wie *Scratch* [74] oder *Snap!* [54], aber auch andere Anwendungsbereiche in vielen Domänen wurden abgedeckt.

Visuelle Programmiersprachen vermieden es jedoch, dem Programmierer den Programmcode zu zeigen und die Menge an grafischen Elementen nahm mehr Sichtbereich ein als deren entsprechende textuelle Repräsentation. Die Idee, die Grundlage des Programmtextes mit grafischen Elementen zu erweitern oder beide Ansätze zu verbinden, wurde seltener verfolgt.

Einer der wenigen Ansätze stammt beispielsweise von Erwig und Meyer 1995 [36], welche in ihrer Umsetzung die Ansätze aus textueller und visueller Welt in Prolog vereinten. Sie prägten damit den Begriff der heterogenen visuellen Programmiersprachen (engl. *heterogeneous visual programming language*, HVPL).

Leider blieb es dabei bei visuellen Elementen, die keine nativen Sprachelemente in der Syntax der Programmiersprache wurden. Mit Hilfe eines Präprozessors wurden sie vor Laufzeit des Programmes zu einer textuellen Repräsentation runtergebrochen. Der Vorteil eines Objektes, sowie dessen Manipulierbarkeit und Interaktivität, geht dabei verloren. In speziellen Live-Programmiersystemen wäre es jedoch möglich, das Konzept der heterogenen visuellen Programmiersprachen im gesamten Lebenszyklus des Programmcodes beizubehalten, indem grafische und ausführbare Elemente mit Text kombiniert werden.

Forschungsfrage und Gliederung

In diesem Kapitel wird nun vorgestellt, wie es im Projekt SandBlocks möglich ist, grafische, interaktive und ausführbare Objekte, bei uns Blöcke genannt, in die Werkzeuge und Editoren der textbasierten Live-Programmierumgebung Squeak/Smalltalk zu integrieren. Dabei werden die folgenden Punkte abgedeckt:

- Wie ist unsere gewählte Abstraktion als visuelle Methoden für das hybride Konzept in Squeak/Smalltalk geeignet? (↦ Abschnitt 3.3.2)
- Wie integrieren sich diese Objekte in die gegebenen Werkzeuge, wie den textuellen Programmeditor? (↦ Abschnitt 3.4.2)
- Spezifisch, wie integrieren sich die Objekte in die Darstellung von Fließtext? (↦ Abschnitt 3.4.3)

Im Hintergrund in Abschnitt 3.2 wird auf bedeutende, bestehende Aspekte von Squeak/Smalltalk eingegangen, sowie das SandBlocks-Projekt vorstellt und dessen Begriffe benannt. Im folgenden Abschnitt 3.3 wird Vorwissen betreffend Anforderungen und gewählten Abstraktionen des SandBlocks-Systems geklärt. Den Hauptteil stellt Abschnitt 3.4 dar. Zunächst schließt sich in Abschnitt 3.4.1 die Konzeption

der Schnittstellen an, bevor diese in Abschnitt 3.4.2 umgesetzt und diskutiert werden. Danach wird in Abschnitt 3.4.3 genauer auf die Textintegration der Blöcke eingegangen. Darauf folgt eine Auswertung der Umsetzung im Abschnitt 3.5, sowie ein Einblick in verwandte Arbeiten in Abschnitt 3.6, bevor wir mit einem Fazit und Ausblick in Abschnitt 3.7 abschließen.

3.2 Hintergrund

Wenn man moderne (integrierte) Entwicklungsumgebungen (engl. *integrated development environment*, IDE), wie zum Beispiel Visual Studio Code[3], mit älteren textbasierten vergleicht, fallen oft die erweiterten grafischen Werkzeuge auf, die im und um den Programmtext eingefügt wurden und das Programmieren vereinfachen sollen. Wir betrachten hier nur diese, welche für Universalsprachen (engl. *general purpose language*, GPL) gedacht sind.

Bekannte Funktionalitäten, um dem Programmierer mehr Übersicht zu bieten kann beispielsweise neben einer Textzeile stehen. Hier sind oft Zeilennummer mit Haltepunkt beim Debuggen, Zustand der Versionsverwaltung, Code-Faltung (engl. *code folding*), IntelliSense, Linter- oder Compiler-Warnung gegeben.

Jedoch gibt es auch Werkzeuge, welche direkt im Programmtext zu finden sind. Kaum noch fehlend ist mittlerweile die Syntaxhervorhebung, sowie Syntaxfehlerhervorhebung. Auch Verlinkung von Methodenaufrufen oder Variablen zu ihren Definitionen sind als Aktion auf Textteilen implementiert – ähnlich URIs oder Dateien, welche dann gegebenenfalls außerhalb der IDE geöffnet werden könnten.

Die bisher genannten Aktionen sind jedoch alle nur eine Anpassung des Textes – es gibt keine andere, nicht-textuelle Darstellung, welche hier auch nicht benötigt werden würde. Eine Besonderheit ist eine Farbvorschau, die bei manchen IDEs vor Farbangaben eingefügt wird und einen Farbwähler öffnen kann. Dies ist Interaktivität, welche den Code direkt modifizieren kann – ein wichtiger Aspekt unseres Projektes. Ein wichtiger Unterschied dabei ist jedoch, dass dieses Element rein grafisch im Editor vorhanden ist und die Sprache nicht erweitert, also keine Auswirkung zur Laufzeit des Programms haben kann.

3.2.1 System Squeak/Smalltalk und Morphic

System Squeak/Smalltalk Das Projekt SandBlocks wurde in Squeak/Smalltalk umgesetzt. Smalltalk ist dabei eine dynamisch typisierte, objektorientierte[4] Programmiersprache und Squeak sowohl ein Dialekt dieser als auch die Implementierung einer Entwicklungsumgebung. Eine wichtige Eigenschaft des Systems ist,

[3] Visual Studio Code: `https://code.visualstudio.com/docs/getstarted/userinterface`, letzter Zugriff am 30. Juli 2019.

[4] In objektorientierten Sprachen ist alles ein Objekt – diese sind Instanzen einer Klasse und kommunizieren untereinander durch Nachrichten (engl. *messages*), die normalerweise Methodenaufrufe auslösen.

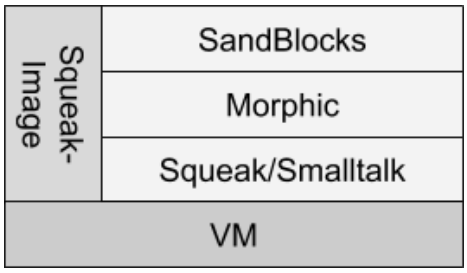


Abbildung 3.1: Hierarchischer Aufbau des Systems. Untere Schichten liegen näher am Betriebssystem, oben befindet sich die höchste Abstraktion.

dass Squeak ein „lebendiges“ (engl. *live* oder *liveness*)[5] System ist, was bedeutet, dass das System ständig läuft. Interaktion mit dessen Zustand erzwingt kein Warten auf neue Programmbefehle. Die Motivation hinter Liveness ist, von der typischen Start-und-Stop-Interaktion (bspw. zum Debuggen) wegzukommen und die Datenströme in kurzen Feedbackschleifen im Programm sichtbarer zu machen. Squeak wird dazu in einem Image gespeichert, welches ein Arbeitsspeicherabbild (engl. *image*) der VM repräsentiert (s. Abbildung 3.1), in dem alle Objekte in einem Objektbaum abgelegt sind. Dazu gehören auch alle Methoden und Klassen der Standardbibliothek, unter anderem auch Parser, Compiler, wichtige Datentypen oder die sichtbare Entwicklungsumgebung. Änderungen von Code und Daten haben so direkte Auswirkungen auf das laufende System und sind nicht durch einen externen Kompilierungsschritt oder einen Neustart der Anwendung getrennt. Dass in einem Squeak-Image alle zur Ausführung benötigten und genutzten Klassen und Objekte mitgeliefert sind und somit Open-Source vorliegen, ist für die Erweiterbarkeit des Systems hilfreich. Dies, gepaart mit der Liveness ist für ein Projekt wie unseres ein wertvoller Vorteil, wenn wir uns mit interaktiven Strukturen in die somit offene Werkzeugkette integrieren wollen.

Morphic-System Innerhalb von Squeak wird für die grafische Benutzerschnittstelle das Morphic-System genutzt. Dieses enthält Logik zur Darstellung von grafischen Objekten, Morphs genannt, in der „Welt“ – dem Bereich, der sichtbar im Squeak-Fenster dargestellt werden kann [73]. Um grafische Nutzerschnittstellen (engl. *graphical user interface*, GUI) zu bauen, können Morphs zusammengesetzt (engl. *composing/composition*) und in beliebiger Tiefe verschachtelt werden. Weiterhin sind in Morphic Eventsensoren, welche Interaktionen mit diesen Elementen ermöglichen, sowie viele Implementierungen für verschiedenste Morphs enthalten. Die entscheidende Grundidee für Morphic liegt in dessen Direktheit (engl. *directness* und Lebendigkeit. Direktheit beschreibt dabei die Möglichkeit, dass jedes dargestellte Gebilde auf dem Bildschirm, also jeder Morph, untersucht und direkt bearbeitet werden kann – dies wird durch einen mittleren Mausklick erreicht,

[5] Es existieren verschiedene Klassifikationen von Liveness. Steve Tanimoto definiert bis zu 6 Level der Liveness [107], von denen das vierte Level dem von Squeak entspricht [19].

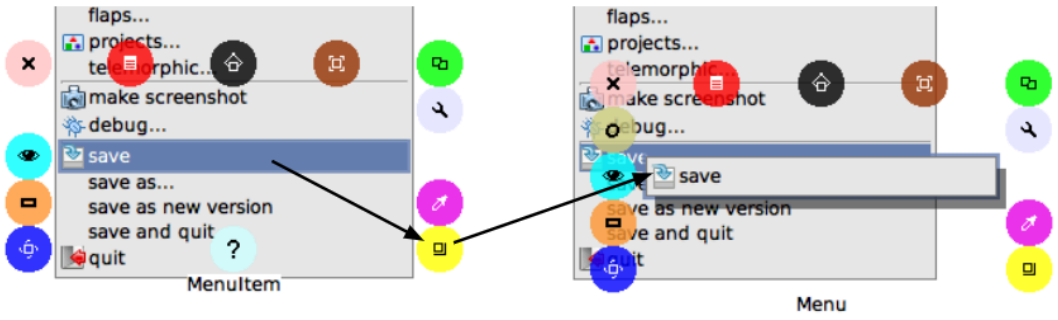


Abbildung 3.2: Auswählen der *Save*-Menüaktion und kopieren dieser in einen neuen Button mit Hilfe des Halos

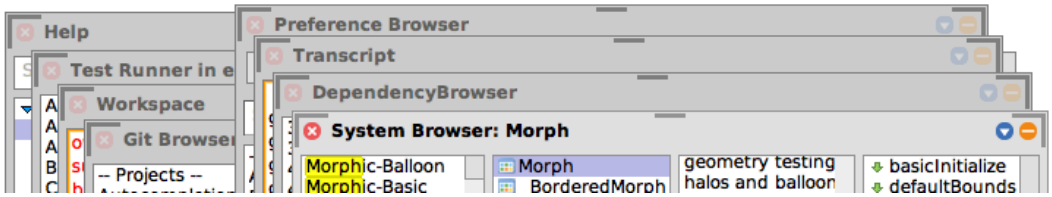


Abbildung 3.3: Auswahl verschiedener Werkzeuge in Squeak

welche ein Kontextmenü um den Morph öffnet, welches auch „Halo“ genannt wird (s. Abbildung 3.2). Liveness bezieht sich auf die bereits bekannte Eigenschaft aus Smalltalk, dass die angezeigten Informationen immer den aktuellen Zustand beschreiben, während sich das System ändert [75]. Ein Beispiel für beides wäre, dass man einen Menüeintrag aus seinem Menü herausziehen kann, um ihn als Schaltfläche an anderer Stelle zu benutzen.

Werkzeuge Zum Arbeiten innerhalb von Squeak wird eine Menge von Werkzeugen bereitgestellt, wie in Abbildung 3.3 gezeigt. Diese öffnen sich typischerweise in einem Morphic-eigenem Fenstermanager in der bereits erwähnten Welt. Dazu gehören neben dem Code- und Klassen-Browser/Editor unter anderem ein Debugger, Objekt-Inspektor/Explorer, Arbeitsbereich, Transcript/Ausgabe, Test-Umgebung oder Git-Versionsverwaltung.

Für die Implementierung unseres Projektes nutzen wir das Morphic-Framework auf dem Stand von Squeak 5.2[6] aus dem Jahr 2018/19. Die in diesem Kapitel beschriebenen Werkzeuge sind jedoch bereits in älteren Versionen zu finden oder nachinstallierbar.

[6] Wir nutzen dabei auch Vorabversionen aus Squeak-Trunk, um Verhalten der Nachfolgeversion zu testen.

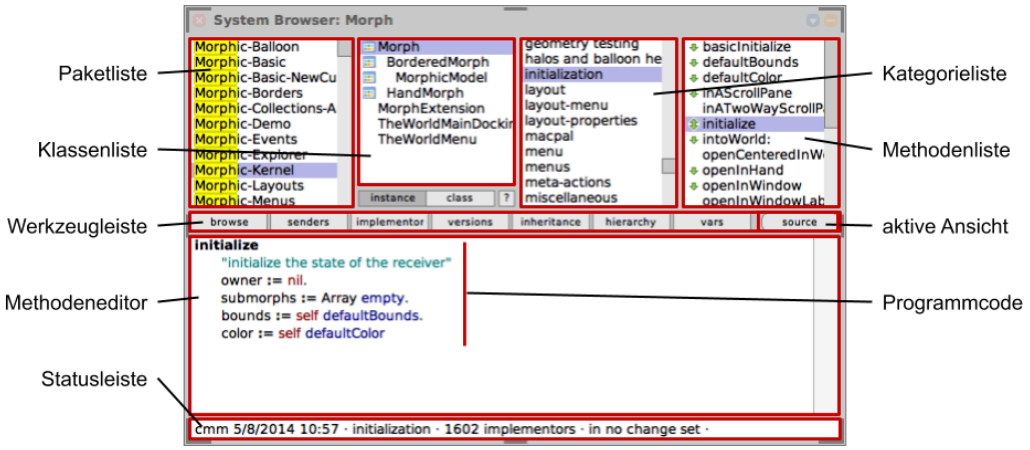


Abbildung 3.4: Übersicht des Code-Browsers

3.2.2 Squeak als Entwicklungsumgebung

Für den Entwickler ist typischerweise der Code-Browser der interessanteste Einstiegspunkt (s. Abbildung 3.4. Dieser ist vertikal in einen Navigationsbereich, eine Werkzeugleiste, sowie den Editor-Bereich aufgeteilt. Im Navigationsbereich lässt sich in einer hierarchischen Baumstruktur aus Paket-, Klassen-, Kategorie- und schließlich Methodennamen[7] der Methoden-Programmcode bearbeiten. Dabei wird im Code-Browser die Navigation nicht in einem aufklappbaren Baummenü – wie zum Beispiel aus einem Dateiexplorer in modernen Betriebssystemen bekannt – umgesetzt, sondern in Listenansichten, die nebeneinander die vier genannten Kategorien auflisten. Der ausgewählte Eintrag pro Liste wird jeweils hervorgehoben und aktualisiert die folgende Liste. Unter der Klassenliste kann zusätzlich zwischen Instanz- und Klassenmethoden gewechselt werden. Zwischen den Listen und dem Code-Editor erstreckt sich auf voller Breite eine Werkzeugleiste, welche kontextspezifisch weitere Werkzeuge öffnen kann.

Der Code-Editor kann verschiedenste textuelle Informationen anzeigen (s. Abbildung 3.5): mit der Auswahl des Methodennamens wird im Editor der Quellcode einschließlich dem Messagepattern angezeigt – Syntaxhervorhebung ist standardmäßig aktiviert. Alternativ können im Editor Methoden- und Klassentemplates (Smalltalk-Code), Klassendefinitionen (auch Code), Dokumentationen (Text), Methoden-Dekompilat (Code) oder -Bytecode (Text) dargestellt werden.

Der Arbeitsbereich (engl. *Workspace*) (Abbildung 3.6) ermöglicht skript-artiges Schreiben von Smalltalk-Code ohne Methoden für dessen Ausführung von Hand anlegen zu müssen – ein Kontext für die verwendeten Variablen wird bei der Ausführung automatisch angelegt.

[7] Der Begriff *Methodenname* entspricht hier dem Smalltalk-Begriff *Selektor* (generell Methodensignatur genannt, also ohne Parameternamen) – *Messagepattern* ist das Äquivalent mit Parameternamen.

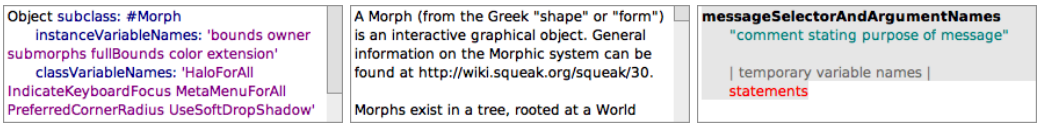


Abbildung 3.5: Editoransicht für Klassendefinition, Klassendokumentation und Methodentemplate

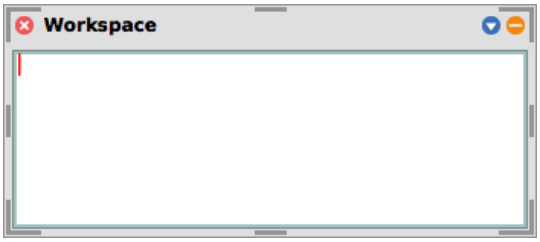


Abbildung 3.6: Workspace-Fenster

Da diese beiden Werkzeugfenster, Code-Browser und Arbeitsbereich, die wichtigsten Programmcode-Eingaben sind, konzentrieren wir uns in diesem Kapitel auf die Integration unserer grafischen, ausführbaren Objekte in diese Werkzeuge. Eine große Rolle wird der Editor im Code-Browser spielen, welcher bisher immer ein Textfeld enthielt – identisch zum Arbeitsbereich. Andere Werkzeuge sollten, wenn sie Code darstellen, aber auch die eingebetteten Objekte darstellen können.

3.2.3 Terminologie des Projektes SandBlocks

Im Projekt SandBlocks wurde das Konzept semantischer, grafischer Elemente in textbasierten Programmierumgebungen umgesetzt – ein hybrider Ansatz aus textueller und visueller Programmcode-Darstellung[8]. SandBlocks ermöglicht als Programmbibliothek und Rahmenstruktur die Erstellung von sogenannten Blöcken, welche unsere visuellen Elemente abstrahieren. Diese sind, wie alles in Smalltalk, Objekte. Blöcke zeichnen sich dadurch aus, dass sie in den Programmtext integrierbar und sowohl in der grafischen Darstellung interaktiv als auch im Programmfluss ausführbar sind. Dazu kann ein Block im Programmcode einsetzbar sein oder diesen in einer Methode vollständig ersetzen[9]. Er behält in beiden Fällen seine Ausführbarkeit im aktuellen Kontext bei.

Model/View Blöcke sind getrennt in Block-Modell (engl. *model*) für die Daten und Logik, sowie Block-Ansicht (engl. *View*), welche Darstellung und Behandlung von Interaktionen definiert[10]. Die Darstellungs- und Interaktionslogik stammt aus dem Morphic-Framework und wird per Vererbung von einer `Morph`-Klasse übernom-

[8] Eine detaillierte Einführung in das Projekt SandBlocks ist in Kapitel 2 nachzulesen.
[9] Diese Abstraktion wird in Abschnitt 3.3.2 nochmal genauer aufgegriffen.
[10] Eine Beschreibung dieser Trennung, sowie Gründe dafür sind in Abschnitt 2.4.2 zu finden.

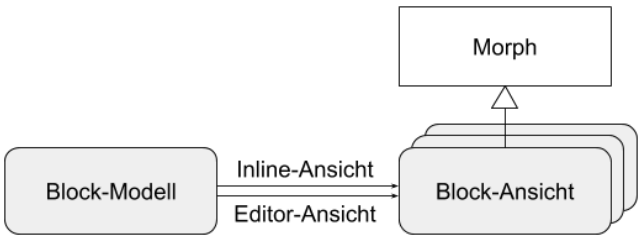


Abbildung 3.7: Model/View-Trennung. Ein einzelnes Modell bietet mehrere Ansichten an, welche von der `Morph`-Klasse erben.

Abbildung 3.8: Darstellungen eines Nummernblocks. Oben: Slider von 0 bis 100; unten: numerisches Eingabefeld.

men (s. Abbildung 3.7). Ein Block kann dabei dem Entwickler im Programmcode verschiedenste Interaktionen anbieten – darunter Mausinteraktion wie Klicken, Kontextmenü oder Verschieben, sowie Texteingaben oder Interaktionen durch andere Morphs, Daten und Logik sind in eine zweite Klasse auslagerbar und werden zur Speicherung und Ausführung genutzt.

Ansichten Ein Block kann jedoch nicht nur eine Darstellungsmöglichkeit für sein bestimmtes Programmverhalten nutzen, wie in Abbildung 3.8 angedeutet. Je nachdem, in welchem Kontext der Block verwendet wird, kann er entweder alleinstehend als benannte Methode oder als Teil von Quelltext zwischen anderem (textuellen) Code stehen. Die alleinstehende Ansicht nennen wir ab sofort nach ihrer Position Editor-Ansicht, die im Text vorkommende Ansicht Inline-Ansicht (*inline*, engl. für *inzeilig* bzw. in der Zeile stehend) – mehr dazu ist in Abschnitt 3.4.2. Weiterhin kann der Nutzer bei einigen Blöcken im jeweiligen Zustand auch zu einer alternativen Darstellung wechseln – beispielsweise kann ein numerisches Eingabefeld auch durch einen Schieberegler (engl. *Slider*) dargestellt werden.

3.2.4 Motivation

Um die Idee der Blöcke aus SandBlocks in Squeak/Smalltalk umzusetzen, muss nun eine passende Abstraktion für hybriden Programmcode gefunden werden – bei uns als Inline-Block und Block-Methode, welche von uns per Implementierung umgesetzt werden. Dabei ist insbesondere auf die Eigenschaften von Morphic und Squeak Rücksicht zu nehmen, welche sich mit ihrer Direktheit (engl. *directness*) und Lebendigkeit (engl. *liveness*) positiv auf die Umsetzung auswirkten. Als Erweiterung des bestehenden Squeak-Systems werden weiterhin verschiedene An-

satzpunkte an Benutzerschnittstellen, an denen das theoretische Block-Konzept erfolgreich in das System angepasst werden konnte, gezeigt, darunter die Einbettung von Blöcken in Text oder als Block-Methode in Klassen.

Dabei wird in diesem Kapitel vor allem auf unsere Integration in bestehende Werkzeuge in der Welt von Squeak eingegangen:

- Welche Anpassungsmöglichkeiten bietet der bisherige textuelle Programmcode-Editor?
- Wie kann ein grafisches Objekt im Text stehend genutzt werden und wie unterstützt das Morphic-System Interaktionen mit diesem?
- Welche der bekannten Nutzerinteraktionen des Textes können weiterhin zur Verfügung gestellt werden?

Schlussendlich werden durch diese Ausführungen manche verwendeten Konzepte des Projektes SandBlocks, welche bereits durch Kapitel 2 aufgestellt wurden, praktisch dargelegt und die erfolgreiche Implementierung derer in ausgewählten Details verdeutlicht. Weiterhin gehen Kapitel 4 und Kapitel 5 auf andere Teilaspekte des Projektes ein, sowie Kapitel 7 und Kapitel 6 auf eine fortführende Nutzung desselbigen.

3.3 Entwurf der Blöcke des SandBlocks-Systems

Vor unserem Projekt, sowie während dessen Entwicklung, mussten wir uns immer wieder Gedanken zu den Anforderungen an unser SandBlocks-System, sowie dessen Block-Elementen machen. Mit der Wahl von Squeak, bzw. dessen Morphic-System hatten wir weiterhin Anforderungen und Einschränkungen, in denen wir uns bewegen mussten.

3.3.1 Anforderungen an Blöcke

Für das SandBlocks-System definieren wir Anforderungen an unser Framework visueller Elemente in Abschnitt 2.4.2. Weiterhin spezialisieren wir diese in Abschnitt 4.1.1 für Blöcke die LIB-Prinzipien, welche für **l**iveness, **I**dentität und **b**eliebigen Programmcode stehen. Liveness bezieht sich auf die bekannte Eigenschaft, welche unsere Block-Instanzen im statischen Programmtext behalten sollten, dass immer der aktuelle Zustand im System bearbeitet wird. Identität bezieht sich auf die Referenz zum Zustand des Blocks, die über den Lebenszyklus im Programmcode, von Editierzeit bis Ausführungszeit erhalten bleiben soll. Mit beliebigem Programmcode ist schließlich die freie Nutzung eines Blocks als Abstraktion des textuellen Codes zur Ausführungszeit gemeint (s. Abbildung 3.9).

Abbildung 3.9: Ablauf der Phasen für Programmcode

```
handIcon

    ^ Form fromBase64:
'iVBORw0KGgoAAAANSUhEUgAAAAwAAAAMCAYAAAB
WdVznAAAAV0lEQVQoz5WRQRKAMAgDE8b/fzme0FiR
Yk7MNNtZWqKObKYfxFdZUgW/AFkRBirBwDAJjoHMUXl
2oXuTxLrDo0xeT6au6EDcM0dKv5fm7h9cB0Dpoe7yE
71jJAU1coySAAAAAElFTkSuQmCC'
```

```
handIcon

    ^
```

Abbildung 3.10: Beispielimplementierung für den Getter eines Icons. Links: textuell; rechts: Version mit Block.

3.3.2 Code-Abstraktion

In der Programmierung kann es schnell dazu kommen, dass komplexe Berechnungsvorschriften (Algorithmen) oder Datensätze viel Platz im Programmcode einnehmen. Diese Komplexität kann in der textuellen Darstellung den Ausführungsfluss schnell unübersichtlich machen (s. Kapitel 2). Typischerweise stößt man bei diesem Problem in der objektorientierten Programmierung auf Methoden, in denen man die Komplexität kapselt. Sie kann jedoch bei uns auch hinter Blöcken verborgen werden, ohne dass es für das Resultat einen Unterschied macht.

Beispielsweise geschieht es in Squeak öfter, dass auf eine Konstante mit einer Grafik zugegriffen werden soll, welche gegebenenfalls noch nicht gesetzt ist – eine Form der verzögerten Initialisierung (engl. *lazy initialization*). Das Bild wird in diesem Fall im *Getter*[11] aus einem Bytearray dekodiert, zugewiesen und die Variable zurückgegeben – hier kann der Getter nun, anstatt ein Bild aus einem Bytearray zu dekodieren und zuzuweisen, einfacher einen Block enthalten, welcher bereits das Bild-Objekt enthält – dieses also nicht dekodieren oder extra abspeichern muss. Nun ist es aber auch nicht notwendig, dass die Funktionalität dieser Methode (den Inhalt und das Zurückgeben des Bildes) textuell dargestellt wird. Stattdessen stellen wir im Block eine Benutzerschnittstelle zur Verfügung, welche das Bild der Methode anzeigt, zugreifbar und austauschbar macht (s. Abbildung 3.10).

3.3.2.1 Abstraktion als Methode

Als gemeinsame Schnittstelle aus Text und Block biete sich nun eine Methode an, da bei einem Methodenaufruf die Implementierung dieser versteckt bleibt und nur das Resultat von Bedeutung ist (s. Abbildung 3.11). Der gleichartige Zugriff per Nachricht an das betreffende Objekt unterscheidet also nicht, ob hinter der

[11] Ein Getter ist eine Methode, welche direkt ein Hilfsobjekt zurückgibt, ohne etwas zu berechnen.

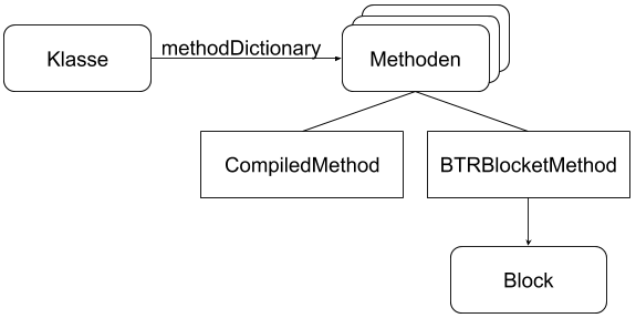


Abbildung 3.11: Struktur, in der Block-Methoden eingebunden werden. Eine Klasse speichert ihre Methoden im `methodDictionary`. Die Methoden können eine `CompiledMethod` oder jetzt auch unsere `BTRBlocketMethod` sein, welche ihren Block enthält.

Methode Text liegt, oder ein Block, der die Berechnung durchführt oder Daten liefert. Damit ein Block überhaupt an dieser Stelle stehen kann, muss er jedoch eine gemeinsame Schnittstelle zur textuellen Methode bieten.

In Squeak sind Methoden einer Klasse in deren `methodDictionary` abgelegt – einer Zuordnung aus Methodenname zu einem Methoden-Objekt (normalerweise `CompiledMethod`). Wir haben uns dazu entschieden, anstatt einer gekapselten Überklasse für bestehende (textuelle) Methoden und unsere Block-Methoden, eine `BTRBlocketMethod` als Subklasse der `CompiledMethod` anzulegen. Dies erspart uns ein Neuschreiben vieler Systemkomponenten, wie beispielsweise des Compilers[12].

Da das bestehende System verlangt, dass der Quelltext der Methode ermittelt werden kann, implementieren wir dessen textuelle Repräsentation wie eine normale Methode, welche als Rückgabewert den Inline-Block zurückgibt. Da dieser String unter anderem serialisiert in der Versionsverwaltung genutzt wird, müssen wir noch ein Wiedererkennungsmerkmal einbauen, damit Block-Methoden von den normalen unterschieden werden können: das Pragma[13] `<BTRBlockModel>` wird vor der Rückgabe in den Quelltext geschrieben und gilt als Markierung, dass bei der Anzeige im Code-Browser die Block-Methode wiederhergestellt werden soll (s. Abbildung 3.12).

3.3.2.2 Verwendung im textuellen Code

Ein Block muss jedoch nicht zwingend in eine Methode ausgelagert werden – er liegt in einer Methode gekapselt vor und kann dadurch wiederverwendet werden, indem er von verschiedenen Stellen im Programm über seinen Selektor referenziert

[12] Der Compiler musste aber zumindest erweitert werden um Objekte im Programmtext zu verarbeiten oder Block-Methoden erstellen zu können (s. Kapitel 4).

[13] Pragmas sind Meta-Annotationen (vergleiche mit Java Annotations und Reflection), welche in Quelltext eingefügt werden können und auf Methoden abgefragt werden können. Sie können gegebenenfalls parametrisiert sein.

Abbildung 3.12: Icon-Getter von Abbildung 3.10. Links: Block als Block-Methode; rechts: Block in der Inline-Darstellung einschließlich enthaltenem Pragma.

wird. Falls dies nicht notwendig ist, kann ein Block auch anstelle eines Programmausdrucks (engl. *expression*) verwendet werden (s. Abschnitt 3.4.2.1).

Unabhängig davon, ob ein Block in eine eigene Methode ausgelagert wurde oder nicht, wird verborgen vom Nutzer die `valueInContext:`-Schnittstelle des Blocks aktiviert und der Rückgabewert entweder als Methodenrückgabe oder als Ergebnis des Ausdrucks verwendet.

3.4 Umsetzung der Blöcke in Squeak und Morphic

Dieser Abschnitt setzt wichtige Aspekte der Umsetzung unseres SandBlocks-Systems in Zusammenhang zum Morphic-System. Das Konzept eines Blocks, wie in Abschnitt 2.4.2 erläutert, bildet hierbei die Grundlage.

Es wird zuerst auf die Voraussetzungen des Gesamtsystems eingegangen, welche Strukturen sich im Morphic-System ergeben und welche bereits bekannten Vorlagen als Ideen verwendet wurden. Die geschieht sowohl aus Sicht des Programmierers als auch aus interaktiver Sicht eines Block-Nutzers.

Darauffolgend teilt sich der Inhalt in zwei Teile, welche erst die Abstraktion der Blöcke im Code-Browser untersucht und die Implementierung in einer Retroperspektive bewertet. Weiterhin wird im zweiten Teil spezifisch die Integration von Blöcken in Morphics Textdarstellung angesprochen und dessen Auswirkungen und Einschränkungen auf das SandBlocks-System analysiert.

3.4.1 Schnittstellen-Design der Blöcke

Unsere Blöcke müssen verschiedene Anforderungen erfüllen, um ausführbar[14] zu sein (s. Kapitel 4). Weiterhin mussten wir uns Gedanken zum Lebenszyklus des Block-Objektes machen (s. Abschnitt 3.4.3.5), dessen Trennung in Daten (bzw. Modell) und Darstellung (s. Kapitel 2), sowie der Serialisierung außerhalb des VM-Images (s. Kapitel 5). Als Entwickler eines Blockes möchte ich außerdem mit

[14] Ausführbarkeit beschreibt aus technischer Sicht die Eigenschaft, in den Bytecode des kompilierten Programms eingefügt werden zu können.

so wenigen zu implementierenden Methoden wie möglich auskommen, um einen Block anzulegen und nutzbar zu machen.

Wir haben uns dazu entschieden, die benötigten Schnittstelle zur Datenseite (bzw. Ausführungsseite) per Trait in `TBTRBlockModel` anzugeben, wie in Abbildung 3.13 veranschaulicht. Manche Methoden müssen zwingend implementiert werden, andere sind optional, beziehungsweise entsprechend der Funktionalität des Blocks anpassbar. Es werden beispielsweise Methoden angeboten, um die zu verwendenden Klassen zur Darstellung anzugeben – `inlineView` für die inzeilige Darstellung, `editorView` für die Editoransicht (welche als Ersatz die erstere Ansicht nutzen würde). Mit `valueInContext:` wird die eigentliche Funktionalität des Blocks implementiert: zur Laufzeit wird der Programm-Kontext übergeben und als Rückgabe der Wert erwartet, dem der Block entsprechen soll. Wird kein Kontext benötigt, so kann auch `value` genutzt werden, auf welches `valueInContext:` zurückfallen würde. Weitere Methoden, wie der Gleichheitsoperator enthalten Standardimplementierungen und können bei Bedarf überschrieben werden.

Da eine Schnittstelle keine benötigten Variablen definieren kann, haben wir eine abstrakte Klasse definiert, welche dem Entwickler bereits die meiste Arbeit abnimmt: `BTRAbstractBlockModel`. Falls man jedoch keine zwei Klassen für die Model-View-Trennung verwenden möchte, muss man den Trait nutzen und dessen Methoden entsprechend implementieren.

Die Darstellungsseite ließ sich ohne neue Klassen lösen. Hier ist die einzige Anforderung, dass von der Klasse oder einer Unterklasse von `Morph` geerbt wird, damit der Block im Morphic-System angezeigt werden kann. Es hat sich hier als sinnvoll erwiesen, dass die Darstellungsseite ihre Datenseite kennt, um deren Daten aktualisieren zu können. Dabei müssen beliebig viele Instanzen der Darstellung auf ein Datenmodell zugreifen können. Dies ermöglicht es, zu alternativen Darstellungen zu wechseln und mehrere Ansichten gleichzeitig zu nutzen – zum Beispiel in mehreren Editoren. Falls die Darstellung weitere Daten enthält, die über ihren Lebenszyklus hinaus benötigt werden, können diese als `viewData` im Modell abgelegt werden. Damit können beispielsweise weitere, identische Ansichten angelegt werden.

3.4.2 Schnittstellen und Werkzeuge

In der Terminologie des Projektes ging ich bereits auf Darstellungsmöglichkeiten eines Blockes ein. Dabei unterscheiden wir nach Kontext zwischen der Editor-Ansicht und der Inline-Ansicht, doch wie ergibt sich dieses Konzept? Welche Nutzerschnittstellen werden erwartet und welche Werkzeuge sind betroffen?

Der Nutzer unserer Blöcke ist in unseren Annahmen in den meisten Fällen ein Entwickler oder zumindest jemand mit Erfahrung im Schreiben von Programmtext. Wir erwarten daher, dass die Erwartungen sich vor allem an bestehender Textbearbeitung orientieren. Dazu kommt die Anforderung der visuellen Interaktion – unserer Blöcke sollten Bedienkomfort von grafischen Elementen ausreizen, welche über Text allein hinausgeht.

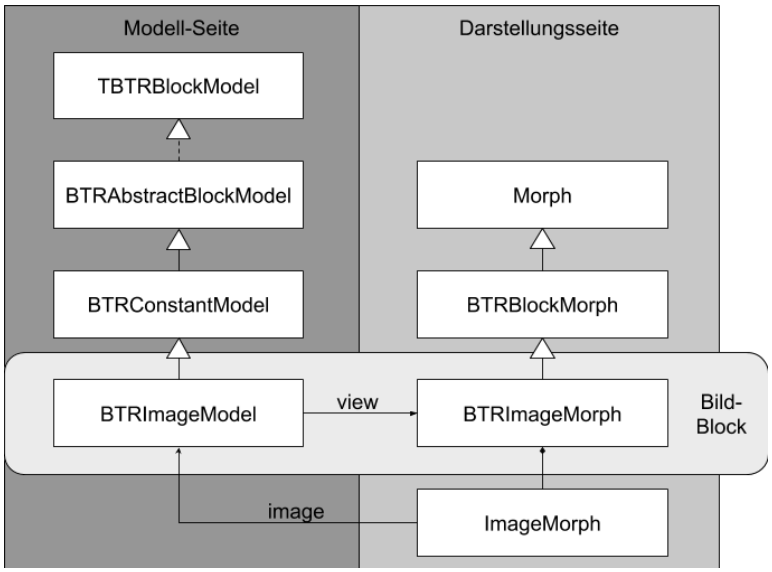


Abbildung 3.13: Vererbungsbaum für den aus Abbildung 3.10 bekannten Bild-Block. Das Bild ist ein statischer Wert, der gesetzt werden kann und der Rückgabewert des Blocks ist. Das `BTRImageModel` erbt von der Klasse `BTRConstantModel`, welche für solche Konstanten eine fertige Schnittstelle zur Verfügung stellt.

Je nach Zweck in dem der Block erzeugt wird, kann er verschiedene Darstellungsformen annehmen. Wir erinnern uns an die Begriffe Inline- und die Editor-Darstellung: bei ersterer umfließt Text einen Block in einer Textzeile; bei letzterer räumen wir dem Block den Platz des gesamten Textbereichs ein, indem wir in ersetzen. Wie deren Implementierung in Squeak aussieht, wird im kommenden Abschnitt 3.4.3 erläutert, im aktuellen widmen wir uns den Konzepten hinter den Darstellungen.

3.4.2.1 Inline-Ansicht

Solange wir die Syntax von Programmtext erweitern (s. Abbildung 3.14), haben wir mit dem Bedienkonzept von Texteingaben zu tun. Es werden ein oder mehrere Textcursor verwendet, um im Text zu navigieren und positionsbasiert Schriftzeichen (engl. *Character*) hinzuzufügen oder zu löschen. Mit der Selektion eines oder mehrerer Zeichen können gemeinschaftliche Operationen auf diesen ausgeführt werden – Ersetzung, Löschen, Verschieben (engl. *drag and drop*), oder Operationen auf der Zwischenablage, wie Kopieren, Ausschneiden oder Einfügen. Die Maus hilft dabei den Textcursor zu positionieren oder die angesprochenen Aktionen durch ein Kontextmenü auswählbar zu machen.

Wenn jetzt Teile des Textes durch grafische Objekte ersetzt werden, beschränken wir uns wegen unseres Anwendungsfalls der Programmierung auf einzelne Syntaxelemente (s. Abbildung 3.15), welche in sich abgeschlossen sind, die sogenannte

```
^ Morph new
    color: #0013FF;
    yourself
```

Abbildung 3.14: Beispiel einer Textselektion in hellem Blau. Der Block ist in der Selektion eingeschlossen.

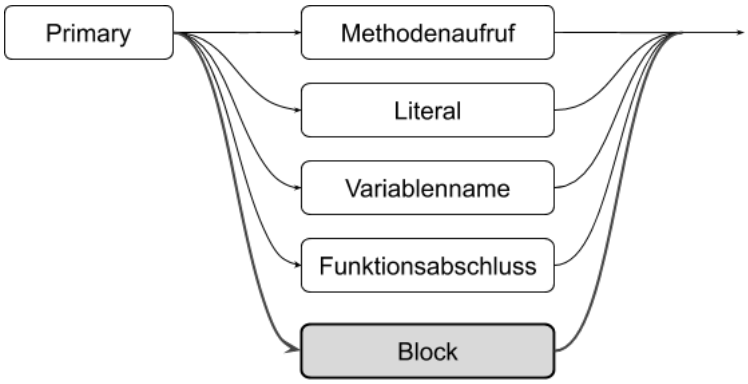


Abbildung 3.15: Syntaxdiagramm der *Primary*-Notation. Der Block stellt die Erweiterung dar.

Kategorie *Primary* – dies können Methodenaufrufe, Literale[15], Variablenname oder Funktionsabschlüsse (engl. *function closure*, in Smalltalk BlockClosure) sein (s. Kapitel 4).

Da der Block einen Bereich im Text einnimmt und somit an einer bestimmten Position im Text liegt, kann der Textcursor mit dem Block interagieren, als wäre dieser ein einzelnes Zeichen – jede genannte Tastaturoperation, einschließlich Selektion wird weiterhin Anwendung finden. Dabei wird der Maus nun ein neues Bedienkonzept[16] zugewiesen: bekannte Aktionen von Elementen, wie Schaltflächen, Schaltern, Schieberegler oder eines eigenen Texteingabefeldes, können in die Inline-Darstellung integriert werden. Klicken und Ziehen sind dabei eine neue Möglichkeit, um die Daten, bzw. den Zustand zu editieren – dies verhindert aber im Bereich des Blocks dessen bekannte Nutzung als Textwerkzeug. Das Kontextmenü kann wiederum kombiniert aus dem Menü des Textes, sowie dem des Blocks gebildet werden und wird nur um Funktionalität erweitert.

Weiterhin erzwingt das Textbearbeitungsmodell auch die Anforderung, dass Blöcke immer kopierbar sein müssen und dabei ihren Zustand übernehmen sollen – der Inhalt des Blocks muss also einen Kopiervorgang vollständig unterstützen und danach in einem identischen Zustand vorliegen.

[15] Literale sind eine Notation für einen festen Wert, welche in der Syntax der Sprache definiert ist. Beispielsweise ein String, eine Zahl oder ein Zeichen.

[16] Es ist auch möglich die neuen Mausinteraktionen auf Tastaturaktionen abzubilden, was aber hier nicht beschrieben werden soll.

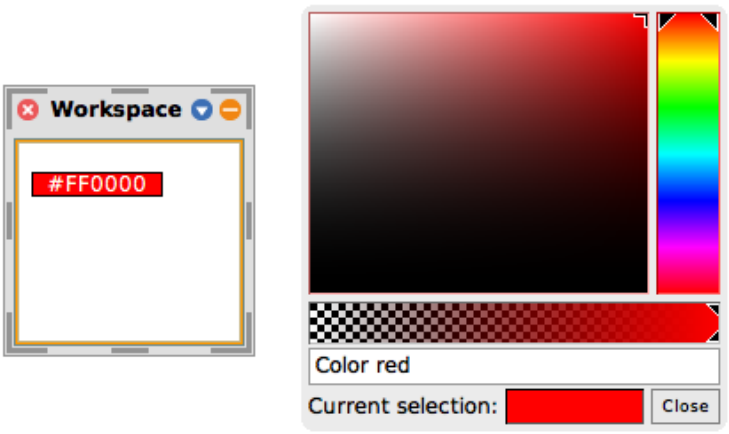


Abbildung 3.16: Links: Farbwahl-Block im Textbereich des *Workspace*; rechts: Farbwähler der Klasse `NewColorPicker`. Der Wähler öffnet sich nach einem Klick auf den Block. Mit dem Betätigen der *Close*-Schaltfläche würde die aktuell gewählte Farbe übernommen werden.

Abschließende Anforderungen an die Inline-Darstellung sind, dass wir unsere rechteckige Darstellung nicht wie eine Textzeile umbrechen lassen, um sie am Anfang der nächsten Zeile fortzusetzen. Dies resultiert darin, dass wir unsere Darstellung auf einen einzelnen Teil einer Zeile beschränken und diesen in der Höhe nicht zu sehr über die Zeilenhöhe hinaus vergrößern wollen, da ansonsten der Abstand des restlichen Textes zur darüber- und darunterliegenden Zeile wachsen würde. Um die Übersicht im Text (und damit die Bedienbarkeit) zu erhalten, sollte die Darstellung bereits besonders auf wesentliche Aspekte des Blocks beschränkt werden. Beispielsweise kann ein Farbwahl-Block sich nach seiner eingestellten Farbe einfärben und als einzige Aktion den Mausklick anbieten, auf welchen hin er ein eigenes Fenster mit Farbwahlrad und RGB-Wert-Eingaben öffnet (s. Abbildung 3.16).

Wie genau wir dabei die Blöcke in den bestehenden Texteditor von Morphic integrieren konnten, wird in Abschnitt 3.4.3 genauer beschrieben.

3.4.2.2 Editor-Ansicht

Entfernen wir uns vorerst von der reinen Texteditierung: Wenn wir die Methoden-Abstraktion auf einen Block anwenden wollen, bietet es sich an, die Darstellung vollständig zu spezialisieren. In einer Ansicht außerhalb eines Texteditors fällt die Platzbeschränkung einer Zeile weg[17] und es ist möglich, ein eigenes Interface mit eigenen Eingabemethoden neben Maus und Tastatur, wie zum Beispiel Kamera, Mikrofon oder anderen Sensoren, zu entwerfen. Auch weitere Werkzeuge zum Analysieren und Editieren der Block-Daten können ergänzt werden. Als Beispiel bietet sich hier eine Definition eines Zustandsautomaten an, welcher beispielsweise

[17] Scrollbalken bieten beliebig viel horizontalen und vertikalen Platz, im Text sind diese jedoch unserer Meinung nach nicht zu empfehlen.

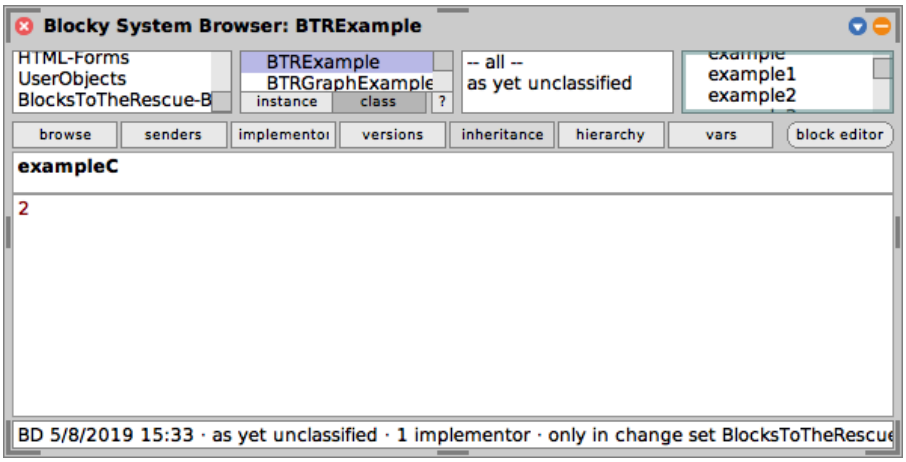


Abbildung 3.17: Editoransicht mit unserem modularen Browser. Der Editor der Methode `exampleC` besteht aus einem ein Code-Block, der den Wert 2 beinhaltet und durch ein Textfeld dargestellt wird.

seine Verbindungen zwischen Zuständen über Lasso-Selektion auswählen oder Ablaufpfade hervorheben könnte. Es ist aber weiterhin angebracht, sich an die Benutzerschnittstellen von Squeak anzupassen, um einen Stilbruch im System zu vermeiden.

Da wir hier den gesamten textuellen Teil der Methode durch ein grafisches Objekt ersetzt haben, erfüllt der Texteditor (s. Abschnitt 3.4.3.2) des Code-Browsers (s. Abschnitt 3.2.2) keinen Zweck mehr und wir ersetzen ihn durch einen selbstgebauten Block-Editor (s. Abbildung 3.17). Dieser soll sich nahtlos in den Code-Browser einfügen. Dafür benötigen wir jedoch noch die Möglichkeit, der Block-Methode einen Methodennamen, sowie Parameter anzugeben, wie bei der textuellen Methode in der ersten Zeile des Methodencodes per Messagepattern der Fall ist. Dazu haben wir in unserem eigenen Block-Editor eine Signatur-Zeile oberhalb der eigentlichen Editor-Ansicht hinzugefügt, in welcher dieser wie gewohnt bearbeitbar ist. Auch eine Behandlung des Änderungszustandes, um versehentliches Verwerfen der Änderungen zu verhindern, sowie eine Anzeige dessen musste ergänzt werden, da diese Textfeld-spezifisch war.

Innerhalb der nun vom Block-Editor zur Verfügung gestellten Fläche kann ein Block jetzt schließlich seine Editor-Ansicht komplett selbst definieren und seine Bedienschnittstellen selbst verwalten. Er erhält dazu über bestehende Eventsysteme von seinen Eltern-Morphs alle Interaktionen übermittelt, auf welche er reagieren kann.

Wir haben beispielsweise dem bekannten Farbwahl-Block (s. Abbildung 3.18) hier die Möglichkeit gegeben, direkt seine Schnittstelle mit Farbwahlrad und RGB-Wert-Eingaben anzuzeigen, welche inline erst nach einer Interaktion mit dem Block geöffnet werden würde. Alternativ kann hier natürlich auch ein Block für einen Funktionsabschluss erstellt werden, der Parameter entgegennehmen kann und einen Wert zurückgibt. Da dieser in Smalltalk geschrieben wird, nutzen wir als

Abbildung 3.18: Farbwahl-Block als Editor der `blueExample`-Methode. Der `BTRColorPickerMorph` nimmt die gesamte Fläche des Editors ein. Dieser erbt vom bekannten `NewColorPicker`, entfernt aber die nicht benötigte *Close*-Schaltfläche.

Darstellung einen Texteditor, womit wir in der Editor-Ansicht beinahe den bestehenden Programmcode-Editor nachgebaut haben – dieser ist nun durch die Abtrennung seines Messagepatterns etwas modularer und speichert seinen Block als Block-Methode ab.

Da der Text-Editor des Code-Browsers ersetzt werden soll, musste letzterer entsprechend angepasst werden, um dies zu unterstützen. Unser neuer BTRBrowser erbt vom bestehenden Browser und ermöglicht den wiederholten Wechsel zwischen verschiedenen Editoren. Wir haben dafür ein bestehendes Auswahlfeld genutzt, mit welchem man unter anderem zwischen Quellcode, Dokumentation, Dekompilat oder Bytecode wechseln konnte. Die Quellcode-Ansicht zeigt bei Block-Methoden die textuelle Quellcode-Repräsentation mit dem `BTRBlockModel`-Pragma und dem Block inline als Rückgabewert. Weiterhin ist in diese Liste eine Aktion für die Konvertierung der Methode zu einer Block-Methode eines gewählten Block-Modells, ein Schalter für einen Wechsel zum Block-Editor selbst, sowie zu einer serialisierten Textansicht hinzugekommen. Der Block-Editor ist bei Block-Methoden standardmäßig ausgewählt.

3.4.2.3 Diskussion

Mit der Wahl des BTRBrowser als Standard-Code-Browser sind Block-Methoden aus Nutzersicht kaum noch von normalen Methoden zu unterscheiden. Wir konnten die Editierung der Blöcke wie gewünscht in das bestehende Squeak-System integrieren, ohne einen Stilbruch zu erzeugen. Mit dem eigenen Block-Editor war es weiterhin möglich, die vollen Möglichkeiten von Interaktionen in Morphic an die Darstellung des Blocks weiter zu reichen. Auch die Nutzung der bestehenden Code-Ansichten wird weiterhin unterstützt und um Wechsel zu den neuen Ansichten erweitert.

Betreffend der vom Code-Browser angebotenen Werkzeuge für Methoden mussten wir jedoch Nichtfunktionalität in Kauf nehmen: bestehende Navigationsmöglichkeiten im Quelltext, wie Methoden-Aufrufe oder -Implementierung anzusehen,

wird auf Block-Methoden keine Resultate liefern, außer ihren eigenen Namen und die valueInContext:-Schnittstelle. Dies ist an sich jedoch nicht verkehrt, da der Block seine Inhalte von dem System abkapselt und gegebenenfalls seine eigenen Werkzeuge dafür bereitstellen müsste. Andere Werkzeuge, wie der Versionsverlauf, welcher in Kapitel 5 beschrieben wird, funktionieren wie erwartet.

Auch die gewohnte Möglichkeit, beim Editieren der Methode die Änderung dieser sichtbar zu machen und somit beim Speichern den Zustand in die Klasse zu kompilieren funktioniert wie bekannt mit dem Tastenkombination Strg + S – sowohl beim Ändern eines Block-Zustandes, als auch bei der Signatur der Block-Methode.

3.4.3 Textintegration von Blöcken

Nachdem wir die Editor-Umgebung des Code-Browsers kennengelernt haben, wird hier wieder auf die textuelle Programmcode-Darstellung in Squeak 5.2 eingegangen – spezifisch die Darstellung von Text, sowie der Inline-Ansicht von Blöcken in diesem.

3.4.3.1 Text versus String

In vielen textuellen Programmiersprachen liegt der Programmcode in Textdateien vor und wird mit einem externen Editor bearbeitet. In Squeak befindet sich der Programmcode jedoch direkt im Speicher des Images und Änderungen geschehen durch das System direkt auf diesem.

Normalerweise geht man bei den Zeichenketten, die für SandBlocks relevant sind, davon aus, dass diese aus einzelnen Zeichen bestehen. Beispielsweise ist der Smalltalk-Programmcode eine in ASCII oder UTF-8 codierter String – ein Array aus ASCII-Werten oder Unicode Code Points, welches in Squeak als ein Objekt der abstrakten Klasse `String` repräsentiert wird[18]. Genauer betrachtet wird die Subklasse `ByteString` als Implementierung verwendet, welche intern nichts anderes als ein Byte-Array ist und somit für jedes Zeichen nur einen Byte Speicher zur Verfügung stellt.

Nun ist dies ein Problem für unsere Blöcke, da ein Objekt nicht in einem Byte-Array referenziert werden kann. Damit Programmcode die notwendigen Informationen enthalten kann, könnte man diese serialisiert im String ablegen, was aber dem Liveness-Prinzip im laufenden Squeak-System widersprechen würde. Wie können nun alternativ interaktive Elemente Objekte in einer Menge Zeichen kombiniert sowohl gespeichert, dargestellt als auch mit Events versorgt werden?

Wir verwenden statt Strings die `Text`-Klasse, wie in Abbildung 3.19 skizziert. Ein `Text` besteht aus einem `String`, sowie einer Menge an Metadaten. Diese erben von der Klasse `TextAttribute` und gelten für Teil-Textabschnitte (sogenannte Text-Runs) des Textes. So kann eine `TextAction` beispielsweise ein Klick-Event für Verlinkun-

[18] „Source methods written by programmers are represented in the Smalltalk-80 system as instances of `String`." – [48, Kapitel 26]

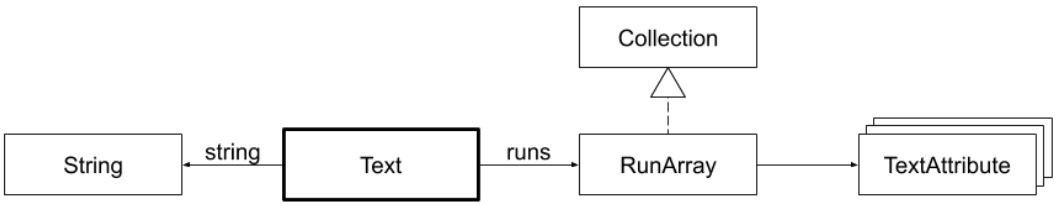


Abbildung 3.19: Klassendiagramm für die `Text`-Klasse. Text enthält je eine Instanz von `String` und `RunArray`. Dieses beinhaltet alle Text-Attribute des Textes.

Abbildung 3.20: Verschiedene Formatierungen für Programmtext. Von links nach rechts: String ohne Formattierung, Differenz zwischen zwei Versionen, Text mit Syntax-Hervorhebung.

gen verarbeiten; andere Attribute können abweichende Formatierung[19] definieren, beispielsweise Hervorhebung, Unterstreichung oder Farbe. Sowohl Text, als auch String erben von `ArrayedCollection` und unterstützen dementsprechend ähnliche Schnittstellen. Indem wir Text nutzen, können die Blöcke als Text-Attribute in den angezeigten Programmcode eingefügt werden. Die meisten Werkzeuge in Squeak nutzen bereits Text oder mussten nur daran angepasst werden, Metainformationen nicht zu verwerfen – entsprechend angepasst kann auch der Compiler die Objekte der Blöcke verarbeiten (s. Kapitel 4).

3.4.3.2 Das Textfeld „`TextMorph`“

Wenn in Squeak ein String ohne Formatierung angezeigt werden soll, kann der simple `StringMorph` verwendet werden. Öfter ist es jedoch der Fall, dass der `TextMorph`, beziehungsweise dessen Subklasse `TextMorphForEditView`, genutzt wird. Der `TextMorph` unterstützt im Gegensatz zum `StringMorph` direkte Bearbeitung seines Inhalts, ohne ein Bearbeitungswerkzeug anlegen zu müssen. Ein weiterer Vorteil der Textdarstellung ist deren Möglichkeit, die auf Text angewendeten Attribute, wie beispielsweise Textabschnitte mit Formatierung darzustellen oder Eventbehandlung auf diesen zu definieren. Im Code-Browser werden die Text-Attribute zur Syntax-Hervorhebung verwendet (s. Abbildung 3.20), in der Versionsverwaltung für die Darstellung der Differenzen zwischen zwei Versionen. Da alle für uns relevanten Werkzeuge in Squeak bereits diese Textdarstellung nutzen, unterstützen sie direkt auch unsere Blöcke.

[19] Die Darstellung des Textes nutzt einen (Standard-) `TextStyle` mit Schriftfamilie und wendet das definierte Aussehen auf diese an.

Mit der Umsetzung vieler Anforderungen – spezifisch Eventbehandlung, Textanordnung, sowie -Darstellung – im `TextMorph` sieht seine Implementierung komplexer aus als die des `StringMorph`s. Für eine bessere Struktur und Anpassbarkeit sind unter anderem bekannte Textwerkzeuge in der ausgelagerten Klasse `TextEditor` implementiert. Hier finden sich Textbearbeitung (Zeichen einfügen oder löschen), Cursor- und Selektionsaktionen, Manipulation der Zwischenablage, Bearbeitungsverlauf (Rückgängig/Wiederholen) oder Text-Hervorhebung (beispielsweise Smalltalk-Syntax). Der Texteditor erbt von Editor für simple Zeichenketten und hat die Subklasse `SmalltalkEditor` speziell für Smalltalk-Programmcode.

Andere Schnittstellen für Eventverarbeitung (s. Abschnitt 3.4.3.3) auf Textteilen oder rudimentäres Einfügen von Morphs in Text implementiert der `TextMorph` selber. Auch die Speichern-Routine[20] findet hier ihren Ursprung, wobei diese Aktion an den `editView` des `TextMorphForEditView` weitergereicht wird, welche Änderungen des Ausgangstextes überwacht und warnen kann, falls man den aktuellen Stand verwerfen würde.

3.4.3.3 Textrendering im TextMorph

Der folgende Abschnitt beschreibt das Textrendering in der von uns genutzten Squeak-Version 5.2. Die gegebenen Informationen dokumentieren den Stand aus dem Image, den Klassenkommentaren und den Methoden. Die Abbildung 3.21 veranschaulicht unsere Ausführungen. Da viele Klasse bereits viel länger existieren, konnten Teile des Morphic-Rendering-Systems aus dem vorherigen MVC-System übernommen werden, weshalb dessen Dokumentation auf dem Stand von Squeak Version 3.7 (Jahr 2007) [21], sowie die Dokumentation der Smalltalk-80-Implementierung von 1983 [48], auch hier einfließen konnte.

Der `TextMorph` enthält eine Instanz der Klasse `NewParagraph` oder dessen Subklasse. Ein Paragraf definiert seinen enthaltenen Text, sowie einen Bereich, in welchem der Text angeordnet werden kann. Der Bereich kann ein Rechteck oder `TextContainer` für komplexere Ausmaße sein. Weiterhin ordnet ein Paragraf seinem Text einen standardmäßigen Textstil zu, welcher unter anderem die zu verwendende Text-Ausrichtung, -Abstände, -Größe oder Schriftfamilie enthält.

Zusätzlich gehören zur Text-Darstellung in Squeak Schriften, welche als Bitmaps vorliegen. In der Klasse `StrikeFont`, welche typischerweise verwendet wird, liegen alle 256 Zeichen der ASCII-Kodierung des lateinischen Alphabets als sogenannte (typografische) Glyphen einer bestimmten Schriftgröße (in der Einheit `pt` = Punkt) in einem Bild vor, wessen Breite der summierten Breite jedes Glyphen entspricht. Eine `xTable` gibt die linke x-Koordinate jedes zur Glyphe zugehörigen Teilbildes an. Eine zweite Zuordnungstabelle bildet die ASCII-Werte auf die entsprechenden

[20] Diese Routine wird durch die Tastenkombination Strg + S oder bei einzeiligen Textfeldern oft per Enter ausgeführt.

[21] Das Muster Model-View-Controller: `http://web.archive.org/web/20180221083104/http://www.bildungsgueter.de/Smalltalk/Pages/MVCTutorial/Contents.htm`, letzter Zugriff am 29. Juli 2019.

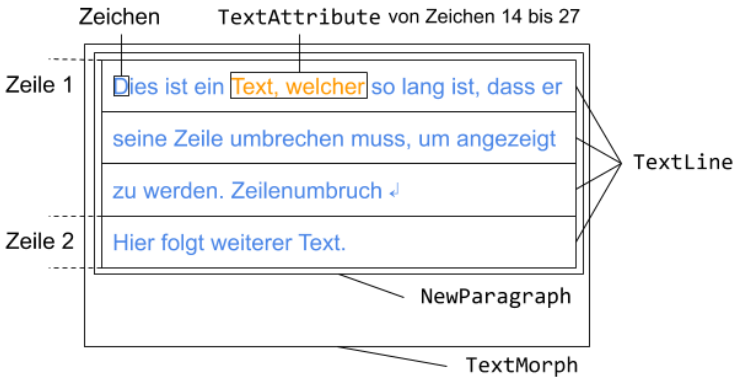


Abbildung 3.21: Beispielhafter Aufbau einer Textdarstellung. Der `TextMorph` enthält `Text` mit einem Umbruch. Der Text nutzt einen Paragrafen, welcher einen blauen Textstil aufweist. Der `NewParagraph` wird entsprechend seiner Ausmaße in `TextLine`s aufgeteilt (dazu gleich mehr). Abweichungen im Textstil und andere Metainformationen werden durch `TextAttribute` angehängt, beispielsweise die orange Farbe des Textabschnittes in der Grafik.

Glyphen ab. Weitere Metainformationen zu jeder Glyphe geben verschiedenste typografische Abstände an, welche für die Zeichendarstellung relevant sind.

Um nun die Position der einzelnen Zeichen des Textes im möglichen Bereich des Paragrafen zu bestimmen, wird ein auf Scannern basierender Schriftsetz-Algorithmus (Schriftsetzer, engl. *typesetter* auch *layouter*) verwendet. Dabei existieren verschiedene Scanner-Subklassen, welche von der gemeinsamen Klasse `CharacterScanner` erben. Ein Scanner implementiert einen Algorithmus, um alle Zeichen eines Paragrafen zu durchlaufen und Layout-Eigenschaften für jedes Zeichen zu bestimmen. Lokale Instanzvariablen enthalten dabei den aktuellen Zustand während des Durchlaufs, darunter die Koordinaten für die Positionierung des nächsten Zeichens, dessen Index, der aktuelle Textstil, sowie die verwendete Schrift. Er enthält das Array `stopConditions`, bestehend aus sogenannten Haltebedingung, welches typischerweise Platz für 258 Werte bietet – für jeden Wert aus der ASCII-Kodierung (1 bis 256) plus `endOfRun` (257 – Bedingung, die angibt, dass der Textabschnitt des aktuellen Text-Attributes abgeschlossen ist) und `crossedX` (258 – Bedingung für das Übertreten des rechten Randes, was einen Zeilenumbruch notwendig macht)[22]. Für jeden Index kann entweder `nil` oder ein Symbol eingetragen sein. Während der Iteration über die Zeichen der Zeichenkette wird in den `stopConditions` geprüft, ob ein nicht-`nil`-Wert vorliegt, in welchem Fall eine Sonderbehandlung notwendig wird und die Iteration mit einem Aufruf des Halte-Symbols auf dem Scanner unterbrochen wird – daher der Name.

Jeder Subklasse des Scanners definiert verschiedene Haltebedingungen, um seine Aufgabe zu erfüllen. Gemeinsam haben sie jedoch die vier folgenden:

[22] Weiterhin existiert bei Blocksatz-Formatierung die Haltebedingung `paddedSpace`, welche das bestehende Leerzeichen ersetzt, um die Textzeile auf die Blocksatz-Breite zu erweitern.

cr zeigt an, dass die aktuelle Zeile abgeschlossen ist und der Schriftsetzer eine neue Zeile anfangen soll (typischerweise für ASCII-Wert $0D_{16}$[23] = CR, carriage return definiert).

tab zeigt an, dass ein Tabulatoreinzug erwartet wird – die Position soll auf den nächsten möglichen gesetzt werden (ASCII-Wert 09_{16} = HT, horizontal tab).

space zeigt an, dass das Ende des aktuellen Wortes erreicht wurde und es möglich wäre auf eine neue Zeile zu wechseln, falls das nächste Wort nicht in die aktuelle Zeile passen würde – genannt Textumbruch (engl. *text wrapping*) (ASCII-Wert 20_{16} = SP, space).

embeddedObject zeigt an, dass hier Platz für ein einzufügendes Objekt geschaffen werden soll (ASCII-Wert 01_{16} = SOH, start of heading).

Wir gehen nun auf die drei wichtigsten `CharacterScanner` ein, welche für das Text-Layouting relevant sind: `CompositionScanner` für die Aufteilung der Zeichen, `DisplayScanner` für die Darstellung von Zeichen, sowie `CharacterBlockScanner` für die Nutzerinteraktion mit Zeichen.

CompositionScanner Der Text eines Paragrafen wird mit Hilfe dieses Scanners in visuelle Textzeilen eingeteilt. Textzeilen bezeichnen hier die Abschnitte des Textes, welche eine gemeinsame Zeilenhöhe, sowie entweder die maximale Breite oder weniger einnehmen. Der Scanner bestimmt den Platz eines jeden Zeichens und bildet neue Zeilen, wenn die vorherige voll ist. Die Positionsberechnung bestimmt sich durch die Glyph-Größe bei normalen Zeichens mit Hilfe von `StrikeFont` und `TextAttribute` oder durch Berechnung in der Haltebedingung. Die dabei entstehenden Textzeilen werden mit der Klasse `TextLine` samt ihren Metadaten (Abmessungen der Zeilen, Start-/Stop-Index, Grundlinie, Abstände, usw.) im Paragrafen des Textes zwischengespeichert.

DisplayScanner Der `DisplayScanner` verarbeitet die resultierenden Textzeilen des `CompositionScanner`. Die typischerweise verwendete Subklasse dessen ist der `BitBltDisplayScanner`, welcher Glyphen auf gegebene Bildbereiche kopieren kann. Auf jeder Textzeile wird für die einzelnen Zeichen deren Rahmen (engl. *bounding box*) berechnet und die Glyphe aus der `StrikeFont` darauf abgebildet. Nach diesem Schritt ist der `Paragraph` im Bildbereich sichtbar.

CharacterBlockScanner Als dritte Klasse kommt dem `CharacterBlockScanner` die Aufgabe zu, Rückschlüsse aus Koordinaten auf das dahinterliegende Zeichen eines Textes zu ziehen. Er nutzt auch die Textzeilen des Paragrafen und bestimmt die Rahmen der Zeichen. Für einen gegebenen Zeichen-Index kann der diesen nun zurückgeben oder für eine Koordinate in den Abmaßen des Paragrafen das entsprechende Zeichen bestimmen. Dies wird beispielsweise für die Eventverarbeitung

[23] Die tiefgestellte Sechzehn steht für die hier verwendete Hexadezimal-Notation.

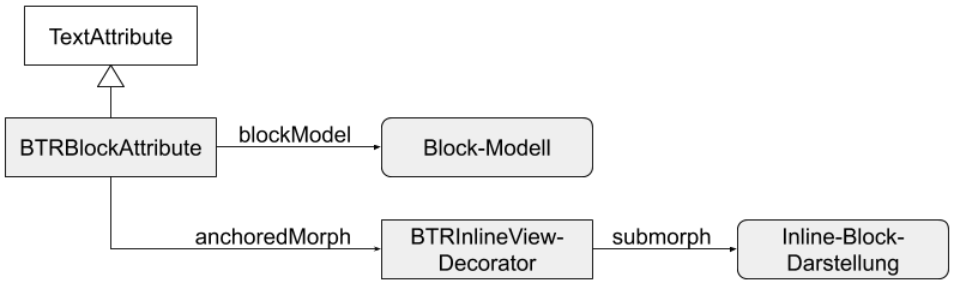


Abbildung 3.22: Klassendiagramm für die neue Klasse `BTRBlockAttribute`. Diese referenziert ihr Block-Modell und speichert den dazugehörenden Morph der Inline-Darstellung zwischen.

benötigt, wenn auf ein Zeichen geklickt wird oder der Textcursor gesetzt werden soll.

3.4.3.4 Inline-Block-Darstellung per `SOH`-Zeichen

Um nun einen Block in Text zu integrieren, erstellen wir ein eigenes Text-Attribut, das `BTRBlockAttribute` (s. Abbildung 3.22). Dieses speichert eine Referenz auf das Block-Modell und auch auf den anzuzeigenden Morph, sobald dieser erzeugt wurde. Dieser Morph ist dabei ein `BTRInlineViewDecorator`, welcher die Block-Darstellung abkapselt, um das Wechseln zwischen verschiedenen Inline-Ansichten per Kontextmenü zu ermöglichen.

Nun muss ein Text-Attribut immer an einem Textbereich definiert sein, wozu minimal ein Zeichen gehört. Ein Block benötigt diese unterliegende Zeichenkette nicht, da er eine eigene Darstellung mitbringt. Für solche Fälle besteht in Squeak bereits ein Konzept, mit welchem bereits Bilder oder reine Morphs in Text eingefügt werden konnten: das `SOH`-Zeichen.

Der ASCII-Standard definiert als Wert 1 das Zeichen „Beginn der Kopfzeile“ (engl. *Start of Heading*, `SOH`), welches nun als Platzhalter-Symbol für einen Block definiert wird[24]. Dieses Zeichen wird solange es ein angehängtes Text-Attribut hat nicht dargestellt und bekommt während des Renderings von uns die Ausmaße des Blocks zugewiesen, womit sich die Zeilenhöhe automatisch anpasst. Die grafische Darstellung des Blocks, also der `BTRInlineViewDecorator` einschließlich der Block-Darstellung, nimmt dann diesen leeren Platz ein, da er gleichzeitig ein Kind-Morph des `TextMorph` s ist. Er nutzt dabei seine Größe, welche er während des von uns angepassten Layouting-Prozesses in den bekannten Variationen von `CharacterScanner` gesetzt bekommen hat. Sollte sich die Morph-Abmessungen der Block-Darstellung

[24] Der Unicode-Block Specials (U+FFF0 bis U+FFFD) würde ein wirkliches Platzhalter-Kontrollzeichen für Objekte anbieten [109] – das Objektersetzungszeichen (engl. *object replacement character*, U+FFFC), welcher in Verbunddokumenten (engl. *compound document*) verwendet wird, um bspw. in der Textverarbeitung Tabellenkalkulationsobjekte, also grafische Objekte, einzufügen. Squeak Version 5.2 unterstützt dies jedoch nicht.

verändern, muss dementsprechend der gesamte `Paragraph` neu gelayoutet werden, da diese Änderung veränderte Textzeilen zur Folge haben kann.

Die Auswirkungen auf die bekannten Scanner sind wie folgt: da SOH als Haltebedingung registriert ist, kann in der Haltebehandlung das Block-Text-Attribut des Zeichens ausgelesen werden, in welchem der Morph der Inline-Ansicht referenziert ist – sollte diese noch nicht existieren, wird eine neue angelegt. Der `CompositionScanner` nutzt nun die Morph-Größe anstatt der Glyph-Größe für seine Layoutberechnung; im `DisplayScanner` wird im Rahmen des SOH-Zeichens nichts gerendert, sondern stattdessen die Position des Morphs gesetzt. Dieser wird anschließend vom Submorph-Rendering des `TextMorph` gezeichnet. Die Abmaße und Position des Morphs werden vom `CharacterBlockScanner` genutzt, um seine Berechnungen entsprechend des SOH-Zeichens zu bestimmen.

Da nun ein zusätzliches Zeichen im Text vorkommt, an welchem der Block als Metainformation hängt, kann der Compiler diesen an richtiger Stelle aus dem Text extrahieren und verarbeiten. Sollte der Text beispielsweise zur Serialisierung als String verarbeitet werden, bleibt das SOH-Zeichen bestehen und der Block wird serialisiert dahinter eingefügt (s. Kapitel 5). In umgekehrter Richtung muss beim Einlesen des serialisierten Strings und Finden des SOH-Zeichens der dahinterliegende Block deserialisiert werden und als Text-Attribut an das Zeichen gehängt werden. Sollte die Deserialisierung fehlschlagen, da beispielsweise die entsprechende Block-Klasse fehlt oder kein deserialisierter Block hinter dem Zeichen liegt, fehlt auch das entsprechende Text-Attribut am Zeichen und es wird als Platzhalter eine SOH-Grafik gerendert. Der Compiler würde in diesem Fall einen Fehler werfen.

3.4.3.5 Lebenszyklus und Zustand von Blöcken in Text

Der Lebenszyklus eines Blockes, spezifisch seines Block-Modells kann in 3 Zustände eingeteilt werden (s. Abbildung 3.23). Wenn im Editor ein neuer Block angelegt wird, besteht er erstmal nur als Arbeitskopie (engl. *working copy*) des Editors und hat eine Darstellung, über welche er editiert wird. Mit dem Abspeichern innerhalb des Editors startet dieser das Kompilieren der (Block-)Methode. Als Teil dieser Methode wird der Block in das Klassensystem übernommen – sein Modell ist jetzt der Methode „verpflichtet“ (engl. *committed*). Beim Laden einer Methode wird wieder eine neue Arbeitskopie erstellt und für den Editor eine Darstellung angelegt. Für eine Nutzung des Programmcodes außerhalb des Squeak-live-Systems kann dieser samt Block weiterhin serialisiert werden – ein textuell serialisiertes Modell entsteht, welches auch wieder deserialisiert werden kann[25].

Eine Methode wird normalerweise als Quellcode geladen – eine Momentaufnahme aus dem aktuellen System, welche editiert werden kann. Mit Abschluss der Bearbeitung durch Speichern wird die bestehende Methode durch den Compiler mit der neuen innerhalb einer atomaren Operation ersetzt – die Identität der Methode hat sich somit geändert. Dies verhält sich ähnlich zum bestehenden Bearbeitungskonzept, in dem einzelne Zeichen des Textes verändert wurden. Um

[25]Genutzt wird die Serialisierung für Versionsverwaltung und kollaboratives Arbeiten (s. Kapitel 5).

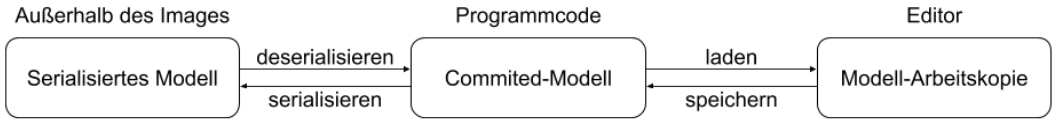


Abbildung 3.23: Lebenszyklus eines Block-Modells in seinen verschiedenen Zuständen

Inkonsistenzen oder (defekte) Zwischenzustände im System zu vermeiden, muss auf ein definierten Zeitpunkt (Speichern durch den Nutzer) gewartet werden, bevor der Compiler das System anpasst.

Bei der Nutzung unserer Blöcke kann dieses Verhalten mitunter unintuitiv sein, da manche Blöcke keine inkorrekten Zustände zulassen. Man erwartet beispielsweise bei einem Boolean-Block, welcher nur zwischen wahr und falsch wechseln kann, dass ein Klick auf diesen seinen Wert im System ändert. Würde sich diese Änderung nach einer Interaktion direkt auf das laufende System auswirken, wäre es nach dem Konzept der Liveness-Level-4 ideal. Da wir jedoch die textuelle Bearbeitungssemantik als Vorbild genommen haben, wird die Änderung auf der Arbeitskopie durchgeführt[26] und deren Auswirkung erst durch den Commit in das System übernommen. Die Liveness ist noch immer vorhanden, jedoch nur mit einem Zwischenschritt nach der direkten Interaktion, womit unser Bearbeitungssystem Liveness Level 3 entspricht[27]. Dies ist unabhängig davon, ob der Block inline vorliegt oder in seiner Block-Methode gekapselt ist – es wird in beiden Fällen für eine Editor-Bearbeitung eine Arbeitskopie erstellt, entweder durch den Editor oder durch den Text während des Ladens des Quellcodes. Somit können wir nur den Zustand und die Gleichheit eines Blockes, nicht dessen Identität als Objekt im Squeak-System während der Editor-Bearbeitung erhalten.

3.4.3.6 Diskussion

Die bestehende Funktionalität, mittels eines Text-Attributes Morphs in Text einzufügen, hat uns überrascht, da dieser Anwendungsfall in unserem alltäglichen Squeak-Arbeitsablauf nie aufgetreten ist. Später wurde uns dann bewusst, dass dies vor allem dem Problem geschuldet war, dass die bestehende Funktionalität einzig für reinen Text implementiert war und viele Werkzeuge der Entwicklungsumgebung noch nicht mit Text(-Attributen) arbeiten konnten. Mit der Implementierung von Textunterstützung in allen Ebenen der Programmverarbeitung wurde die Erstellung unseres hybriden Ansatzes durch Integration der grafischen Elemente in den Text stark vereinfacht, wie in Abbildung 3.24 beispielhaft veranschaulicht.

Im nächsten Schritt konnte die Information des Blocks auch durch Ausbau des Compilers durch den Programmcode in die Methoden abgelegt werden. Die gerin-

[26] Die Interaktion mit dem Editor wurde dabei erweitert, um den Nutzer wie gewohnt über ungespeicherte Änderungen zu informieren.

[27] Das direkte Bearbeiten der Eigenschaften des Blocks als Objekt ist davon nicht betroffen und kann noch immer live nach Level 4 stattfinden.

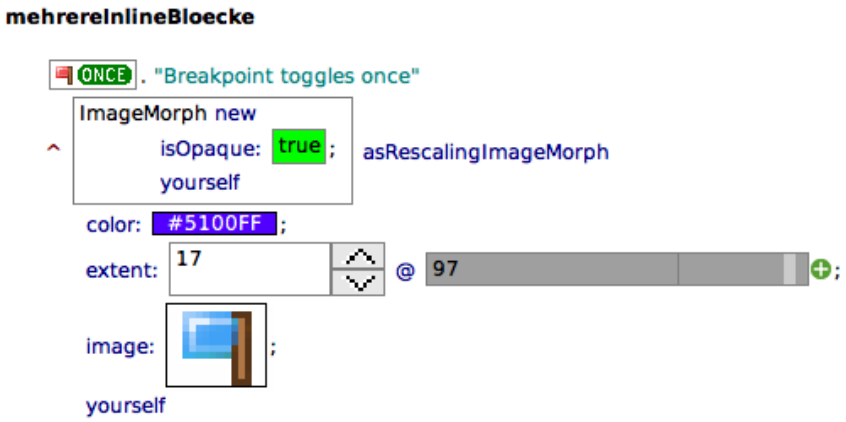


Abbildung 3.24: Beispielprogrammcode, welcher mehrere Inline-Blöcke verwendet. Darunter befinden sich ein Haltepunkt-, Codefeld-, Boolean-, Farbwahl-, Nummerneingabe- und Slider-, sowie Bild-Block. Anzumerken ist, dass im Codefeld-Block der Boolean-Block geschachtelt liegt.

gen Unterschiede zwischen Strings und Text vereinfachten uns dabei den Umstieg sehr und ermöglichten uns nur wenige Teile des Compiler-Systems anfassen zu müssen.

Bisher konnten wir weiterhin keine Probleme mit der Wahl des SOH-Zeichens feststellen. Die gewählte Einbindung des Blocks per Text-Attribut und die Kapselung der Inline-Ansicht in ein Container-Objekt ermöglichen gute Erweiterbarkeit an verschiedenen Punkten der Darstellung, beispielsweise durch neue Aktionen auf dem Kontextmenü des Blocks. Auch das bestehende Morphic-Eventsystem bot eine Anbindung an die Inline-Darstellung im `TextMorph`, sodass keine manuelle Umrechnung aus Zeichenposition und Mauszeiger eingesetzt werden musste – selbst drag and drop funktionierte wie erwartet und wurde eine Nutzerinteraktion zum Einfügen von Blöcken wie auch in Abschnitt 2.5.3 beschrieben wird.

Noch nicht ideal umgesetzt ist die Aktualisierung der Inline-Block-Ansicht. Mit jeder Änderung an dessen Morph wird ein erneutes Text-Rendering ausgelöst, welches eine relativ teure Operation ist. Es sollte ausreichen, nur im Fall einer Größenänderung ein neues Text-Layout zu bestimmen – andere Veränderungen brauchen nur den Morph neu zu zeichnen. Die Editor-Ansicht ist von dem Problem nicht betroffen, da sie keinen Text verwendet und durch normales Morph-Rendering aktualisiert wird.

3.5 Evaluation

Im Folgenden wird auf die verschiedenen Punkte dieses Kapitels eingegangen und deren Auswirkungen auf das bestehende System, sowie die daraus folgenden Vor- und Nachteile diskutiert. Dafür wird zuerst auf die Idee des Blocks eingegangen, danach die Umsetzung eines Blocks als eigenständige Methode und dessen Integra-

tion in den Code-Browser. Anschließend wird die umgesetzte Inline-Funktionalität evaluiert, sowie ein Abschluss mit dem Blick auf das Liveness-Konzept gebildet.

Die Erweiterbarkeit des Squeak-Smalltalk-Systems war entscheidend für viele der in diesem Kapitel besprochenen Resultate. Dank der verfügbaren Quellen und direkten Eingriffsmöglichkeiten in das laufende Live-System, konnte der gewünschte hybride Ansatz aus ausführbarem Text und interaktiver Grafik eingebaut werden. Geschlossene Programmiersprachen, sowie solche, welche einen externen Editor benötigen, würden eine Trennung zwischen ihrer eigentlichen Syntax und dessen Erweiterung durch Blöcke benötigen, was bei unserer Umsetzung nicht der Fall ist.

3.5.1 Blöcke als visuelle, interaktive und ausführbare Elemente

Als Grundlage für unser SandBlocks-System konzeptionierten wir Blöcke (s. Kapitel 2), welche in zwei Darstellungen umgesetzt wurden: Inline in Programmcode, was unsere Hauptidee war und Primaries ersetzen kann (s. Kapitel 4), sowie als Methode, um Wiederverwendung und Kapselung zu unterstützen. Beide Konzepte machen für verschiedene Anwendungsfälle Sinn und Blöcke, wie der Farbwahlblock oder ein Zustandsautomat, konnten ihre Werte besser visualisieren oder die Komplexität des dahinterliegenden Programmcodes verbergen. Ohne diese Trennung in beide Darstellungen würden der Benutzer weniger Freiheiten bei der Umsetzung seines Anwendungsfalls haben – es wird kein Text mehr benötigt, wenn man keinen nutzen möchte.

Für die in Squeak vorhandene Programmierumgebung und ihre Werkzeuge ist mit dem Projekt eine Integration der Blöcke in diese entstanden, welche auf dem Bedienkonzept der textuellen Programmiersprachen aufbaut, um den Einstieg in unser System zu vereinfachen. Die Aufteilung der Block-Abstraktion schlägt sich hier in seiner Darstellung nieder: Um einen Block als bekannte Methode zu bearbeiten, wurde der bestehende Methoden-Text-Editor aus dem Code-Browser ausgetrennt und parallel dazu unser Block-Editor hinzugefügt, sowie Rückumwandlungen der Darstellung zum bestehenden Text-Editor eingebaut. Um das System diesem Eingriff nicht direkt auf dem bestehenden System einzuführen legten wir die Änderungen in einer Subklasse des Browsers ab, welche aber als Standard für den Code-Browser eingestellt werden kann.

3.5.2 Block-Methoden im Code-Browser

Der umgesetzt Block-Editor bietet einem Block-Entwickler den vollständigen Freiraum des Morphic-Systems, um beliebige Benutzerschnittstellen umzusetzen. Es können problemspezifische Interaktionen auf dem gesamten vom Code-Browser dafür zur Verfügung gestellten Platz aufgebaut werden – man ist nicht an einen möglichst kompakten Überblick auf den Block gebunden, da er alleinstehend angezeigt wird. Rein visuelle Sprachen, wie Snap!, welche den Zusammenhang mit allen anderen Blöcken zeigen, müssten hier Einschränkungen vornehmen, um nicht zu viele Informationen in engem Raum darzustellen (s. Kapitel 2).

Auch wenn man eine Block-Methode im Code-Browser ähnlich wie eine normale Methode öffnen kann, so sind diese konzeptuell nicht identisch zueinander. In den Code-Browser integrierte Werkzeuge, wie bestehende Navigationsmöglichkeiten im Quelltext (Methoden-Aufrufe oder -Implementierung aufrufen), liefern auf einem Block keine Resultate; andere Werkzeuge – wie der Versionsverlauf – sind jedoch übertragbar und funktionieren weiterhin.

3.5.3 Inline-Blöcke im Programmtext

Auch bei der Verwendung von Blöcken innerhalb des bestehenden Textes erwies sich Squeak als hilfreich: Frühere Ideen, grafische Elemente per SOH-Zeichen in Text einzufügen mussten von uns nur fertiggestellt und auf das restliche bestehende Werkzeug-System übernommen werden. Ein Block entspricht dabei im Text dem Konzept eines einzelnen Zeichens, welches er mit seiner Darstellung ersetzt. Diese Darstellung kann weiterhin ein Morph sein und unterstützt die Interaktionen, welche auch anderen Morphs geboten wird. Der umgebende Text kann weiterhin wie gewohnt bearbeitet werden und die Integration des Blocks verhindert keine Funktionalitäten, wie Ausschneiden oder Kopieren. Jedoch werden an der Stelle der Blockdarstellung die Maus-Interaktionen vom Block behandelt, womit vor allem Verschiebe-Aktionen (engl. *drag and drop*) oder Selektion dort nicht stattfinden können. Es ist für uns jedoch keine Anforderung, das Bedienkonzept explizit für Tastatur oder Maus zu optimieren – hier richteten wir uns nach dem bestehenden System.

Es resultierten auch neue Anforderungen aus der Verwendung des Textkonzepts, wie die zwingende Kopierbarkeit oder der Austausch der Darstellung im Text, welche in die Umsetzung unseres Systems eingeflossen sind – zum einen in den Lebenszyklus des Modells, in welchem Arbeitskopien Kopien bereits erlauben, sowie in einer beliebigen Anzahl an Darstellungen auf ein und demselben Modell.

Dank der Nähe von Text und String zueinander konnten im bestehenden System, die Teile, welche nur Strings verarbeiteten, um die Verarbeitung von Text erweitert werden. Die Blöcke wurden als Metainformationen an textuellen Programmcode angehängt und wurden im erweiterten Compiler in die ausführbare Methode eingebaut, in welcher sie durch eine gegebene Schnittstelle aufgerufen werden konnten.

3.5.4 Liveness des SandBlocks-Systems

Schlussendlich mussten wir auch Abstriche im Level des live-Systems machen. Eine volle Level-4-Liveness ist bei Fokus auf Visualität von Objekten ideal und ist mit Morphic kein Problem, jedoch verlangt textueller Programmcode im Squeak den Schritt des Kompilierens, welcher erst kohärente Zustände ermöglicht. Die Auswirkungen der Änderung unserer Blöcke innerhalb des Editors sind somit erst nach einem atomaren Commit im System vorhanden, was noch immer Level 3-live ist. Ein Block, welcher bereits in einer Methode liegt und ausgeführt werden kann, kann aber außerhalb eines Editors bearbeitet werden – seine Darstellung

würde vollständige Level-4-Liveness unterstützen. Ein Vorteil dessen ist auch, dass Werkzeuge die lebendigen Block-Objekte direkt aus dem System erhalten können, ohne diese erst aus einem serialisierten Wert aufbauen zu müssen, was bei nicht-live-Programmiersprachen der Fall ist. Dort wäre das Einbauen von Blöcken auch möglich, jedoch nur mit dem eben beschriebenen Zwischenschritt.

3.6 Verwandte Arbeiten

3.6.1 Heterogeneous Visual Languages [36]

Unser System ist nicht das erste, welches einen Hybrid aus visuellen und textuellen Programmiersprachen erdacht hat. Wie bereits in Abschnitt 3.1 erwähnt, bildet das Konzept von M. Erwig and B. Meyer die Grundlage für die Idee der grafischen und ausführbaren Blöcke. Anders als im Projekt SandBlockssetzen sie ihre visuellen Programmierelemente jedoch nicht in einem Live-Programmiersystem um, sondern übersetzen die Elemente in einem Zwischenschritt in ausführbaren Code. Ohne diesen fehleranfälligen Schritt ist unser System eine Fortführung dieses Ansatzes.

3.6.2 Das System *Snap!* [54]

ist eines von vielen grafischen Programmiersystemen, welches hier beispielhaft genannt werden soll, um die von uns integrierten Konzepte der visuellen Programmierung hervorzuheben. *Snap!* selbst ist eine einfache Programmiersprache, welche auf Programmbausteinesetzt, auch Blöcke genannt. Diese werden aneinandergehängt und bilden so den abstrakten Syntaxbaum (engl. *abstract syntax tree*, AST) des Programms – somit sind sie ausführbar. Da die Blöcke jedoch rein grafisch zusammenhängend bereits die gesamte Syntax bilden, können sie nicht mit Text gemischt werden – auch das Bedienkonzept ist rein visuell und fokussiert sich vor allem auf Mausbedienung. *Snap!* wird, wie auch sein Vorgänger *Scratch* [74], wegen seiner Einfachheit gerne als Einstieg in die Programmierung gelehrt – bei größeren Systemen fallen die Nachteile visueller Programmierung zu stark ins Gewicht (s. Kapitel 2). Wir konnten dieses Problem in SandBlocks durch die fortgesetzte Nutzung von textuellen Konzepten des objektorientierten Smalltalks verhindern.

3.6.3 Die IDE *Visual Studio Code* (auch *VSCode*)

ist eine mächtige moderne Entwicklungsumgebung von Microsoft. Geschrieben in HTML und JavaScript und ausgeführt mit Electron, ist sie explizit darauf ausgelegt, erweitert zu werden. Typischerweise kann jede textuelle Programmiersprache in der Editor-Ansicht von *VSCode* geschrieben werden – dies bearbeitet direkt die hinter dem Text stehende Datei. Wie in vielen anderen IDEs ist es auch in *VSCode* möglich auf dem Programmtext Widgets anzuzeigen, welche jedoch nur interaktive, grafische Elemente sind, die auf Werkzeuge abbilden, und keine zur Ausführungs-

zeit vorhandenen Objekte. Ein Weg, die lebendigen Objekte aus Squeak auf die Widgets dieser Entwicklungsumgebung zu überführen, würde somit das Schreiben eine HVPL in *Visual Studio Code* ermöglichen.

3.7 Zusammenfassung und Ausblick

Wir haben in diesem Kapitel dargelegt, wie das Projekt SandBlocks grafische, interaktive Elemente, sogenannte Blöcke in die Werkzeuge der Live-Programmierumgebung Squeak/Smalltalk und das Morphic-System eingebaut hat. Blöcke setzen dabei einen Hybriden aus grafischen und textbasierten Programmiersprachen um, indem sie entweder Teile von Programmtext oder ganze Methoden ersetzen. Blöcke sind somit ein ausführbarer Teil des laufenden Systems geworden.

Entsprechende bestehende Werkzeuge des Systems wurden daran angepasst, dass sie Text mit integrierten Blöcken verarbeiten konnten – dazu zählen unter anderem der Code-Browser und der Workspace, der Compiler (s. Kapitel 4) oder die Versionsverwaltung (s. Kapitel 5). Dabei wurde im Detail der Ausbau des Editors zum Block-Editor beschrieben, sowie die Integration von Objekten in Text und die Darstellung desselbigen im Morphic-System.

Ausblick

Squeak und Morphic waren mit Unterstützung von Level-4-Liveness und einer offenen Architektur eine gute Wahl für das SandBlocks-Projekt, jedoch sind nur wenige größere Systeme mit dieser Eigenschaft ausgestattet. Fortführend zu diesem Projekt könnte untersucht werden, wie sich das Konzept von hybriden Elementen in Programmcode anderen Programmiersprachen und Systemen verhalten würde.

Welche Zwischenschritte müssten sowohl in der Programmiersprache (Syntax, Compiler), als auch in deren Entwicklungsumgebung vorgenommen werden, bevor ein Objekt in textuellen Programmcode eingefügt und ausgeführt werden kann und wie kann dieses in dateibasierten Sprachen im Editor interaktiv angezeigt werden?

Die Ansätze sind vielfältig und würden die Vorteile von grafischen Programmiersprachen in die bekannten textuellen Sprachen einbringen. Mit SqueakJS und Morphic.JS bestehen bereits Implementierungen, um Liveness in Umgebungen außerhalb von Squeak oder Smalltalk zu bringen. Von JavaScript ist es aber nicht mehr weit zu Entwicklungsumgebungen, wie Visual Studio Code (angesprochen in Abschnitt 3.6.3), welche auf dieser nicht-live-Sprache basieren und beliebig erweiterbar sind.

Dieses Kapitel hat nur einen kleinen Teil des SandBlocks-Projektes abgedeckt. Dieses kann jedoch eine Grundlage für die Entwicklung weiterer visueller Metaphern bieten. Für die Integration von Blöcken in Squeak/Smalltalk mussten neben der Darstellung in Text oder die Erweiterung der Werkzeuge viele weitere Probleme gelöst werden. Die anderen Kapitel enthalten weitere Konzepte (s. Kapitel 2),

Umsetzungen (s. Kapitel 4 und Kapitel 5) oder Nutzungsmöglichkeiten für den hybriden Ansatz von Blöcken (s. Kapitel 7 und Kapitel 6).

Schlussendlich ist jedoch ebendiese Integration besonders wichtig, um Blöcke überhaupt nutzbar zu machen. Wo vorher Text stand, befindet sich jetzt ein grafisches Objekt – interaktiv und ausführbar. Kann dies eventuell in weiteren Programmiersprachen und -system ermöglicht werden?

4 Erweiterung eines text-orientierten Compilers für die Verarbeitung visueller Programmelemente

In objektorientierten Live-Programmiersystemen sollte alles ein, durch den Entwickler interaktiv veränderbares, Objekt sein. Bisher ist es jedoch nicht möglich, diese Live-Objekte wieder in den Programmcode, der auch nur ein Objekt ist, zu integrieren und auszuführen. Solche Live-Objekte können z.B. visuelle Elemente sein, die damit als vollwertige Programmelemente in den Programmcode aufgenommen werden können. So entsteht ein Programm, welches sowohl Text als auch Grafik enthält.

In diesem Kapitel wird dafür eine Erweiterung einer objektorientierten, textuellen Syntax gegeben, sodass sie zusätzlich visuelle Programmelemente enthält. Vor diesem Hintergrund wird der Squeak/Smalltalk-Compiler angepasst, dass er visuelle Elemente kompilieren kann, welche dann ganz normal ausgeführt werden. Diese Integration visueller Elemente in den Programmcode ermöglicht so die freie Kombination interaktiver visueller Elemente und Text sowie eine erhöhte Performance bei der Ausführung von Programmen mit visuellen Elementen.

4.1 Einleitung

Neben textuellen Programmiersprachen sind seit den 1950er Jahren immer mehr visuelle Programmiersprachen (VPLs) entstanden. Prominente Vertreter kommen dabei insbesondere aus dem Lern- und Bildungsbereich, bekannt sind so zum Beispiel Snap! [54] und Scratch [74]. Allein der englischsprachige Wikipediaartikel zu VPLs[1] umfasst rund 150 verschiedene Sprachen, deren Anwendungsbereiche sich über jegliche Domänen erstrecken. Bei all diesen Ansätzen handelt es sich um rein visuelle Lösungen, die versuchen textuelle Repräsentationen komplett zu vermeiden.

Um Entwicklern ihre textuelle Grundlage zu erhalten und gleichzeitig die Möglichkeit zu geben, visuelle Programmelemente zu nutzen, gibt es Ideen, die textuelle und visuelle Ansätze verbinden. Eine der ganz wenigen Umsetzungen findet sich von Erwig & Meyer in [36], in dem die Autoren visuelle Programmelemente in die textuelle Basissprache Prolog integrieren. Die entstandene Integration wird als heterogene visuelle Programmiersprache (HVPL) bezeichnet.

[1] Visuelle Programmiersprache: `https://en.wikipedia.org/wiki/Visual_programming_language#List_of_visual_languages`, letzter Zugriff am 06. Juni 2019.

4.1.1 Konzeptionelle Einordnung von *SandBlocks*

In der Umsetzung von Erwig & Meyer [36] haben visuelle Elemente, ebenso wie der textuelle Programmcode, keinen durch den Entwickler veränderbaren Zustand. Stattdessen ermöglichen sie nur eine weitere Form der Darstellung. An dieser Stelle setzt das SandBlocks-Projekt (s. Kapitel 2) mit dem Bau einer solchen HVPL in Squeak/Smalltalk an. Das Projektverfolgt die Integration visueller Elemente, genannt ein Block[2] bezeichnet, als gleichwertige Programmelemente in das zugrundeliegende Programmiersystem. Dabei sollen die visuellen Elemente ihre Vorteile neben der Visualität, insbesondere die Interaktivität und Liveness, beibehalten. Ein visuelles Element wird dabei in ein Modell und eine Präsentation aufgespalten, kann somit für verschiedene Kontexte unterschiedliche Ansichten zur Verfügung stellen Kapitel 2. Auch das Modell kann jedoch rein visuell sein, benötigt also keine textuelle Repräsentation.

Um diese Integration zu ermöglichen und dabei den visuellen Elementen sowohl während des Programmierens als auch zur Ausführung einen sinnvollen Nutzen zu geben, wurden in Kapitel 2 Anforderungen an SandBlocks gestellt. Die sich daraus ergebenden Anforderungen an visuelle Elemente werden als **LIB**-Prinzipien zusammengefasst:

L = Live Visuelle Elemente werden als Live-Objekte in den bis dato ausschließlich statischen Programmcode integriert.

I = Identitätserhaltend Die Identität visueller Elemente bleibt während des gesamten Lebenszyklus erhalten. Wird also zur Editierzeit ein visuelles Element in den Programmcode eingefügt, liegt zur Ausführungszeit dasselbe visuelle Element vor.

B = Beliebiger Programmcode Visuelle Elemente können zur Ausführungszeit beliebigen Programmcode ausführen.

4.1.2 Forschungsfrage

Anknüpfend an den Stand der Forschung und der konzeptuellen Idee hinter SandBlocks soll in diesem Kapitel folgende Frage behandelt werden:

> Wie kann man Compiler für textuelle Sprachen so erweitern, dass diese auch visuelle Programmelemente nativ verarbeiten können und damit heterogene, visuelle Programmiersprachen kompilieren?

Dabei sollen nicht, wie in der Umsetzung von Erwig & Meyer, visuelle Elemente mittels eines Präprozessors auf eine textuelle Repräsentation abgebildet werden, da es sich dabei um keine Integration der visuellen Elemente als native Sprachelemente handelt. Ziel ist eine Integration der visuellen Elemente in die Syntax

[2] Im Folgenden wird ein Block weiter als visuelles Element bezeichnet, um mögliche Verwechslungen mit dem Syntaxelement *BlockClosure*, umgangssprachlich nur als *Block* bezeichnet, zu vermeiden.

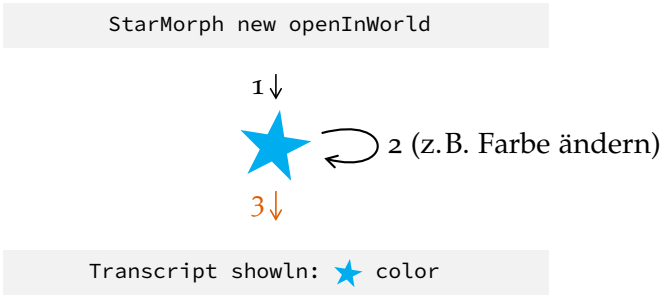


Abbildung 4.1: Reintegration visueller Elemente in den Programmcode. Nachdem ein visuelles Element erstellt wird (1), kann dieses beliebig oft manipuliert werden (2). Danach folgt die Reintegration dieses Elements in den Text (3).

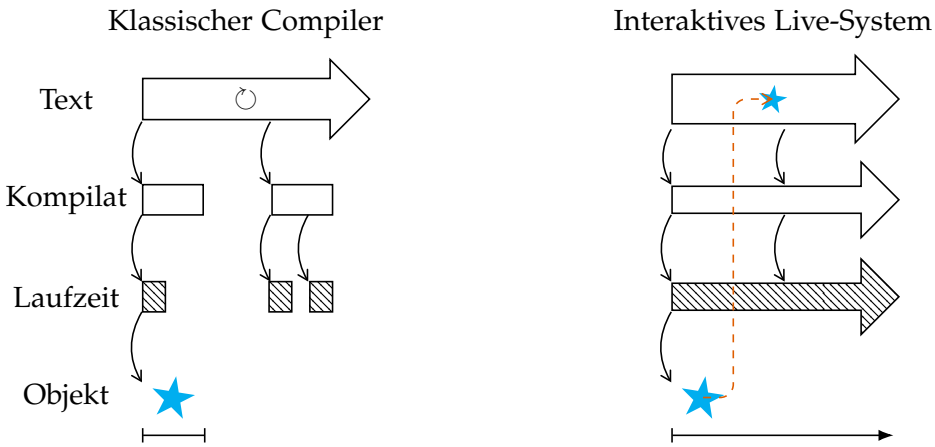


Abbildung 4.2: Lebenszyklus von Objekten bei klassischen Compilern im Gegensatz zu den Möglichkeiten interaktiver Live-Programmiersysteme

der Basissprache. Dies hat den Vorteil, dass die visuellen Elemente während ihres gesamten Lebenszyklus als vollwertige Live-Objekte erhalten bleiben und so ihre Interaktivität und Liveness zu jedem Zeitpunkt bestehen bleibt.

In interaktiven Live-Programmiersystemen wie Squeak/Smalltalk können visuelle Elemente erstellt und dann beliebig vom Nutzer verändert werden. Nicht gegeben ist jedoch bisher die Reintegration dieser Elemente als Programmelemente in den Programmcode, während sie weiter live und manipulierbar bleiben (s. Abbildung 4.1).

In klassischen Compilern ergibt sich diese Möglichkeit der Reintegration auch nicht, da dort ein Objekt nach dem nach dem Ende seiner Laufzeit nicht mehr existiert (s. Abbildung 4.2).

Zudem besteht das konzeptionelle Problem, dass eine Beschreibung für einen Compiler fehlt, der sowohl textuelle als auch visuelle Elemente verarbeiten kann. Es liegt dem Autor aktuell auch keine Syntaxbeschreibung vor, die sowohl visuelle als auch textuelle Elemente erlaubt.

4.1.3 Beiträge und Gliederung

In diesem Kapitel werden die folgenden Beiträge geleistet:

- Eine Erweiterung objektorientierter Syntax, sodass visuelle Elemente als vollwertige Programmelemente aufgenommen werden.
- Eine konkrete Anpassung des Squeak/Smalltalk-Compilers, um visuelle Elemente zu verarbeiten.
- Ein Konzept zur Ausführung visueller Elemente als gleichwertige Sprachelemente in Live-Programmiersystemen.
- Eine deutliche erhöhte Performance von Programmen, wenn ihr Programmcode visuelle Elemente enthält.

Im Folgenden wird in Abschnitt 4.2 eine Beschreibung des zugrunde liegenden Squeak/Smalltalk-Programmiersystems gegeben. Auf dieser Grundlage analysiert Abschnitt 4.3 wie visuelle Elemente in die Syntax einer textuellen, objektorientierten Basissprache integriert werden können. Zur Umsetzung dieser konzeptionellen Überlegungen werden in Abschnitt 4.4 die notwendigen Anpassungen des Compilers implementiert sowie die sich damit ergebenden Veränderungen zur Ausführung visueller Elemente beschrieben. Abschnitt 4.5 analysiert die Performanceeinflüsse auf die Kompilierungs- und Ausführungszeit sowie die Integration in vorhandene Tools. In Abschnitt 4.6 wird ein Überblick über verwandte Arbeiten gegeben. Zuletzt fasst Abschnitt 4.7 dieses Kapitel zusammen und gibt einen Ausblick auf Anknüpfungspunkte für zukünftige Forschung.

4.2 Der Compiler für Smalltalk-Code in Squeak

Squeak/Smalltalk bietet für den Bau einer HVPL mit den beschriebenen Eigenschaften die Grundlage eines objektorientierten Live-Programmiersystems. Es bringt mit Smalltalk eine objektorientierte Sprache, deren Sprachdesign für Liveness ausgelegt ist, und bietet mit Squeak die Programmierumgebung. Dessen Grafik ist durch das Morphic-Framework umgesetzt, demzufolge werden für die visuellen Elemente Morphs genutzt. So erlaubt Squeak als Live-Programmiersystem die Erstellung visueller Objekte mit *Level-4-Liveness* [108], welche für den Nutzer interaktiv mit veränderbaren Zustand sind. Mit SandBlocks lassen sich diese visuellen Elemente mit veränderbarem Zustand auch wieder in den Programmcode einbinden.

Das Konzept von HVPLs, deren visuelle Elemente diesen **LIB**-Prinzipien folgen, ließe sich auf verschiedene Weisen umsetzten. Eine Möglichkeit wäre, eine eigene Programmiersprache mit visuellen und textuellen Elementen von Grund auf selbst zu definieren und implementieren. Damit einhergehend müsste jedoch die zugehörige Programmierumgebung, welche Liveness und Interaktivität ermöglicht, ebenso separat erstellt werden. Eine Neuentwicklung eines Systems mit knapp einer Million Zeilen Programmcode ist utopisch.

Die naheliegende Variante ist daher die Erweiterung einer bestehenden Programmiersprache. Da es sich bei Squeak/Smalltalk um ein quelloffenes, komplett in sich selbst, also Smalltalk, geschriebenes System handelt, ermöglicht es die vollständige Veränderbarkeit des Systems [60]. Somit ist es möglich auch Systemteile, die die Sprache direkt beschreiben, insbesondere den Compiler, anzupassen. Von daher wird diese naheliegende Variante der Erweiterung gewählt.

Um zu verstehen an welchen Stellen visuelle Elemente in Programmcode eingefügt werden können, muss zunächst die Smalltalk-Syntax analysiert werden. Damit können in Abschnitt 4.3 passende Syntaxelemente identifiziert werden, die sich durch visuelle Elemente erweitern lassen.

4.2.1 Smalltalk-Syntax

Smalltalk setzt primär das objektorientierte Paradigma[3] um. Hinzu kommt, dass in Smalltalk alles, angefangen von Zahlen bis zu hoch komplexen Dingen, z.B. Prozessen, Objekte sind [48, S. 7]. Diese kommunizieren ausschließlich über Nachrichten miteinander. Dementsprechend schlank und einfach ist die Syntax, somit ist es relativ einfach, diese zu erweitern und anzupassen.

Eine Syntaxbeschreibung des aktuellen Squeak/Smalltalk-Systems wird z.B. in Ohm/S [90], einer Smalltalk-Implementierung auf Basis des Ohm-Frameworks [110], durch die genutzte Grammatik gegeben. Da sich diese auf das aktuelle Squeak/Smalltalk System bezieht, umfasst sie Sprachelemente, die in Smalltalk-80 (noch) nicht vorgesehen waren, z.B. *Pragmas* oder *ObjectArrayLiterals*.

[3] Eine Ausnahme bildet die Zuweisung `:=`. Da das objektorientierte Paradigma aber nicht gebrochen werden soll, wird diese hier bewusst ignoriert.

Die Syntaxbeschreibung des originalen Smalltalk-80 Systems wird durch [48, Anhang] in Form von Syntaxdiagrammen gegeben[4]. Da sie keine erweiterten Sprachelemente umfasst, ist sie deutlich kürzer als die Ohm/S Grammatik. Des Weiteren arbeitet sie, im Gegensatz zu der aus Ohm/S, auf Tokens, was sie ebenfalls verkürzt. Demzufolge wird für die nachfolgenden Betrachtungen diese Syntax verwendet, da sie absolut ausreichend ist. Entsprechend wird auch die Terminologie von [48, S. 37] übernommen.

Einzelne Betrachtungen dieser 33 Syntaxelemente wären aufgrund der unterschiedlichen Abstraktionsebenen wenig ertragreich. Zusätzlich ginge die Allgemeingültigkeit verloren. Stattdessen wird zunächst die Kernsyntax identifiziert, anhand derer eine Betrachtung erfolgen soll. Die Kernsyntax besteht aus den folgenden Elementen: *Identifier*, *Literal*, *Primary*, *Message Expression*, *Method*, *Comment*.

Unter einem *Identifier* versteht man jeglichen Bezeichner, der sich aus Ziffern und Buchstaben zusammensetzt und bildet damit die Grundlage der gesamten Syntax. Zusammen mit *Symbol* und *Array* ergibt sich ein *Literal*. Da Literale jedoch nie exklusiv vorkommen können, ergeben sich *Primarys*, die zudem noch Variablennamen, Zuweisungen und Blöcke umfassen. Daneben findet sich als weiteres wesentliches Element die *Message Expression*, die Nachrichten ermöglicht. Eine *Method* gibt die Möglichkeit, mehrere *Expressions* zu kapseln. Weiterhin findet sich noch das Syntaxelement *Comment*, welches insofern spannend ist, da es explizit keinen Einfluss auf die Ausführung hat.

Eine Methode, die sämtliche ausführungsrelevante oben beschriebene Syntaxelemente enthält, ist `Example>>redStar`. In einer textuellen Repräsentation ist sie in Listing 4.1 gegeben, sie gibt einen roten, sternförmigen Morph zurück. Zunächst wird eine neue Instanz der Klasse `StarMorph` erzeugt indem ein *Message Send* mit Empfänger `StarMorph` und unärer Nachricht `new` erfolgt. An dieses Resultat wird wiederum die n-äre Nachricht `color:` mit Parameter `Color red`, welcher wiederum ein unärer *Message Send* ist, gesendet[5]. Das Ergebnis wird, gekennzeichnet durch den ↑, an den Aufrufer zurückgegeben.

Listing 4.1: Beispielmethode, die einen roten, sternförmigen Morph erzeugt

```
1 Example >> #redStar
2    ^ StarMorph new
3       color: Color red
```

Im Folgenden wird die Beispielmethode genutzt, um die Änderungen direkt im Programmcode zu zeigen. Insgesamt eignet sie sich für die weiteren Betrachtungen, da sie (1) sehr kurz und einfach ist, (2) viele verschiedene Syntaxelemente enthält, (3) verschieden komplexe Objekte enthält und (4) unäre und n-äre Nachrichten abdeckt.

[4] Die Syntaxdiagramme sind in Abschnitt A beigefügt.

[5] Formal korrekt müsste auf das `Color red` eine Kaskade, welche durch ein `yourself` beendet wird, folgen. Diese wird jedoch bewusst weggelassen, um das Beispiel zu vereinfachen. Der Code bleibt auch so absolut valide.

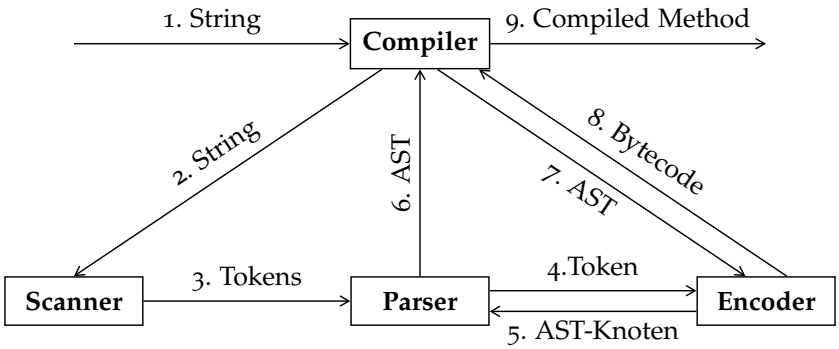


Abbildung 4.3: Abhängigkeiten und Ablauf einer Kompilierung in Squeak

4.2.2 Aufbau des Squeak/Smalltalk-Compilers

Die Smalltalk-Syntax benötigt jedoch auch einen Compiler, der sie umsetzen kann. Dafür findet sich in *Squeak5.2* die in Abbildung 4.3 beschriebene Infrastruktur des Compilers. Die zugrundeliegende Architektur folgt dem Entwurfsmuster „Facade“ (dt. Fassade) [43, S. 185 ff.]. Der Compiler bildet die Fassade, während Scanner, Parser und Encoder das Subsystem bilden.

Bei dem Squeak/Smalltalk-Compiler handelt es sich um einen klassischen *Single-pass-Compiler*. Er folgt dem typischen Aufbau, bestehend aus einer Analysephase gefolgt von einer Übersetzungsphase [65, S. 65, Bild 2.3.1]. Die Analysephase teilt sich in eine lexikalische, syntaktische und semantische Analyse, die Übersetzungsphase besteht aus der Codegenerierung gefolgt von einer ggf. vorhandenen Codeoptimierung.

Es findet sich folgende Zuteilung von Phasen zu Klassen: Die lexikalische Analyse wird vom Scanner übernommen, syntaktische und semantische Analyse werden vom Parser realisiert. Der Encoder übernimmt die eigentliche Codegenerierung der einzelnen Knoten, der Compiler setzt diese dann zu einer `CompiledMethod` zusammen. Eine Optimierungsphase existiert nicht.

Bei dem Parser handelt es sich um einen *Recursive descent parser* [62, S. 19], dessen Grammatik nicht explizit vorliegt. Damit ist dieser besonders schnell, das Einpflegen veränderter Grammatik aber nicht sonderlich komfortabel.

Es ergibt sich folgender Durchlauf durch den Compiler:

1. Der Compiler wird angestoßen. Dies funktioniert in der einfachsten Form mittels `aClass compile: aString`, was im Endeffekt den angegeben `String` an den Compiler übergibt.
2. Der Compiler reicht den String an den Scanner durch.
3. Dieser zerlegt den String in Tokens und übergibt diese Tokens sowie den zugehörigen Tokentyp an den Parser.
4. Der Parser übergibt die Tokens einzeln an den Encoder. Dieser speichert sich einige Informationen, um z. B. später Namen von Variablen auflösen zu können.

5. Im Gegenzug bekommt der Parser einen entsprechenden AST-Knoten zurück und kann diesen in den AST einbauen.
6. Der Parser gibt den fertigen AST an den Compiler zurück.
7. Dieser wiederum gibt den AST zum Encoder. Hier erfolgt die Umsetzung des AST zu Bytecode.
8. Der erzeugte Bytecode wird an den Compiler gegeben, der daraus eine `CompiledMethod` formt.
9. Die `CompiledMethod` wird als Endergebnis vom Compiler zurückgegeben.

Es gilt zu beachten, dass Squeak lediglich eine konzeptionelle Trennung zwischen Scanner und Parser vornimmt. Für eine bessere Performance existieren zwar zwei Klassen, die die darunter liegenden Konzepte allerdings vermischen.

Für die Beispielmethode aus Listing 4.1 ergeben sich folgende Artefakte: In Abbildung 4.4 findet sich in der ersten Zeile der Programmcode, also Listing 4.1. Darauf folgen die vom Scanner erzeugten Tokens.

Der generierte AST findet sich in Abbildung 4.5. Die AST-Knoten teilen die gemeinsame Wurzelklasse `ParseNode`.

Der vom Encoder erzeugte Bytecode findet sich als Teil der vom Compiler generierten `CompiledMethod` in Listing 4.2. Die genauen Bytecodes sind in spitzen Klammern im Hexadezimalsystem angegeben. *Method Header* und *Method Tally*

redStar	^	*StarMorph*	*new*	*color:*	*Color*	*red*	
#word	#upArrow	#word	#word	#keyword	#word	#word	#doIt

Abbildung 4.4: Die Token der Beispielmethode `redStar`. Der Parser liest am Ende beliebig viele #doIt Zeichen, die das Ende des Programmcodes symbolisieren.

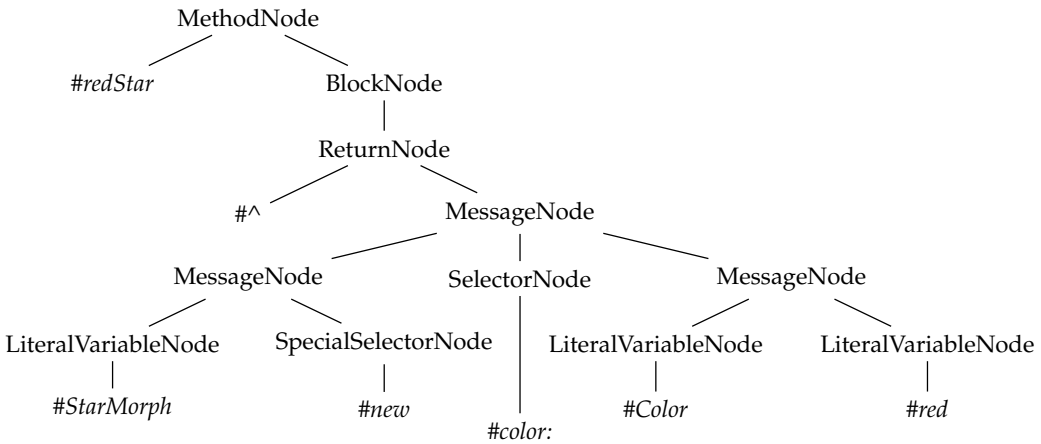


Abbildung 4.5: Der generierte AST der Beispielmethode. Die Blätter werden von konkreten Symbolen gebildet und sind kursiv dargestellt. (Bei `#StarMorph` und `#Color` handelt es sich um ein spezielles Binding von Symbol zu Klasse.)

wurden weggelassen. Eine genaue Erklärung zu den konkreten Bytecodes findet sich auf [48, S. 548ff].

Listing 4.2: Generierter Bytecode der Beispielmethode inkl. der Literale

```
literal1 #color:
literal2 #StarMorph=>StarMorph
literal3 #red
literal4 #Color=>Color
literal5 #redStar
literal6 #Example=>Example
57  '<41> pushLitVar: #StarMorph=>StarMorph'
58  '<CC> send: new'
59  '<43> pushLitVar: #Color=>Color'
60  '<D2> send: red'
61  '<E0> send: color:'
62  '<7C> returnTop'
```

4.2.3 String versus Text in Squeak/Smalltalk

Squeak/Smalltalk unterscheidet explizit zwischen Text und String. Während Strings eine Aneinanderreihung von Zeichen sind, die aus alphanumerischen Zeichen und Sonderzeichen bestehen, reichert Text diese Zeichen noch mit Attributen an. Diese Attribute sind den einzelnen Zeichen zugeordnet und für deren Aussehen zuständig.

Text wird insbesondere in Werkzeugen eingesetzt, um z.B. Sprachkonstrukte hervorzuheben (engl. *(syntax) highlighting*), Fachworte kursiv darzustellen oder Hyperlinks zu kennzeichnen. Strings hingegen werden primär für eine technische, implementierungsnahe Darstellung genutzt. Hinzu kommt das Problem, dass die Klassen `String` und `Text` zwar von `ArrayedCollection` erben, jedoch keine gemeinsame Oberklasse haben, welche ein Text/String-Protokoll implementiert [48, S. 146].

So war insbesondere der Compiler nie darauf ausgelegt, Text zu verarbeiten. Dieser Umstand findet sich schon in [48] im ersten Satz des Compiler-Kapitels:

> „*Source methods written by programmers are represented in the Smalltalk-80 system as instances of* `String`“

Die Integration visueller Elemente in das textbasierte System funktioniert nun wie folgt: Es wird Text statt Strings genutzt. An der Stelle, an der das visuelle Element dargestellt werden soll, wird ein spezielles Zeichen in den String eingefügt. Da dieses Zeichen nicht sichtbar sein soll und auch nicht anderweitig genutzt werden darf, wird das Steuerzeichen *Start of Heading* (ASCII-Wert:1, [25]) genutzt (die letztendliche Auswahl des Zeichens ist arbiträr). Im weiteren wird dieses Zeichen entsprechend des RFCs als `SOH` abgekürzt. Diesem `SOH` wird dann als Textattribut das entsprechende visuelles Element angehängt. Weitere Details zur Integration visueller Elemente finden sich in Kapitel 3.

Der sich ergebende Text kann dann, in der Theorie, an den Compiler gegeben werden. Dieser behandelt ihn als sei er ein String und nutzt aber bei Bedarf das Textattribut, an welchem das darzustellende visuelle Element hängt.

4.3 Visuelle Elemente als Erweiterung der Syntax

In diesem Abschnitt wird die Erweiterung der Smalltalk-Syntax für visuelle Element vorgestellt. Dafür werden in Abschnitt 4.3.1 die in Abschnitt 4.2.1 identifizierten, für eine Erweiterung geeigneten, Syntaxelemente beschrieben sowie die Erweiterungen an der Syntax vorgenommen. Danach folgen in Abschnitt 4.3.2 visuell nicht unterstützte Syntaxelemente.

Als laufendes Beispiel dient die Methode `redStar` aus Listing 4.1, welche Stück für Stück um visuelle Elemente ergänzt wird. Es folgen zunächst alle daraus resultierenden Listings zur Referenz.

Listing 4.3: Objekterzeugung der Farbe Rot ersetzt durch ein visuelles Element

```
Example >> #redStar
   ^ StarMorph new
      color: #FF0000
```

Listing 4.4: Objekterzeugung von `StarMorph` ersetzt durch ein visuelles Element

```
Example >> #redStar
   ^ ★
      color: Color red
```

Listing 4.5: Unäre Nachricht `new` ersetzt durch ein visuelles Element

```
Example >> #redStar
   ^ StarMorph ■
      color: Color red
```

Listing 4.6: N-äre Nachricht `color:` ersetzt durch ein visuelles Element

```
Example >> #redStar
   ^ StarMorph new
      Color red
```

Listing 4.7: N-äre Nachricht `r:g:b:` ersetzt durch ein visuelles Element

```
Example >> #redStar
   ^ StarMorph new
      color: Color 1 0 0  "statt 'Color r: 1 g: 0 b: 0' "
```

Listing 4.8: Vollständige Methode ersetzt durch ein visuelles Element

```
Example >> #redStar
   ★
```

4.3.1 Visuell unterstütze Syntaxelemente

Die wesentlichen Elemente der Objektorientierung sind Objekte, Nachrichten und Methoden, beschreiben durch Stefik & Bobrow [106] als:

> *„All of the action in object-oriented programming comes from sending messages between objects. [...] Objects respond to messages using their own procedures (called „methods“) for performing operations.“*

Zunächst sollen demzufolge Objekte, Nachrichten und Methoden den Syntaxelementen zugeordnet und ersetzt werden.

4.3.1.1 *Primary* als visuelles Element

Ein Objekt kann jegliches Element, insbesondere aus der nahezu vollständig visuellen Realwelt, sein (Beispiele finden sich auf [48, S. 6]). Sie umfassen ein kontinuierliches Spektrum von einfachen bis zu hochkomplexen, konfigurierten Objekten. Bei all diesen Objekten handelt es sich um domänenspezifische Objekte, die sich besonders für eine visuelle Darstellung anbieten [36, S. 1]. Verbindet man diese mit direkter Manipulation und Liveness handelt sich nun um Programmcode, in dem sich interaktive Live-Objekten befinden.

Ein einfaches Element findet sich in der Beispielmethode in Form der Farbe, die ausschließlich aus drei Zahlen für Rot-, Grün- und Blauanteil besteht. Ersetzt man die gegebene Farbe ergibt sich Listing 4.3. Entsprechend findet sich die Ersetzung eines komplex konfigurierten Objekts, des `StarMorph`, in Listing 4.4[6]. Beide Methoden evaluieren weiterhin zu einem roten, sternförmigen Morph.

Bei all diesen Ersetzungen, egal ob für einfache oder komplexe Objekte, findet sich am Ende pro Ersetzung ein visuelles Element im Programmcode. Somit kann es syntaktisch analog zu einem *Literal* betrachtet werden und in die Kategorie *Primary* eingeordnet werden. Es ergibt sich die in Abbildung 4.6 dargestellte Syntaxerweiterung.

4.3.1.2 *Message Expression* als visuelles Element

Nachrichten werden an Objekte geschickt, um diesen die Möglichkeit zu geben auf sie zu reagieren. Die Zuordnung von Nachricht zu Methode funktioniert dabei i. d. R. über Selektoren, bei diesen handelt es sich um *Identifier*. Visuelle Elemente hingegen bringen zunächst nur große Datenmengen mit sich und erschweren somit die Erstellung einer eindeutigen Repräsentation für den Entwickler. Eine Ersetzung von Nachrichten durch visuelle Elemente scheint somit zunächst wenig natürlich.

Eine solche Ersetzung hat jedoch durchaus Vorteile, auch wenn eine andere Motivation als bei *Primarys* vorliegt: Nachrichten lassen sich so als vollwertige Objekte umsetzen. Statt einer Zuordnung über den Selektor, wird dem Empfänger das gesamte Nachrichtenobjekt zur Verfügung gestellt. Dem Empfänger wird dann

[6] Der äquivalenter Smalltalk-String eines `StarMorph` findet sich in Listing 4.25.

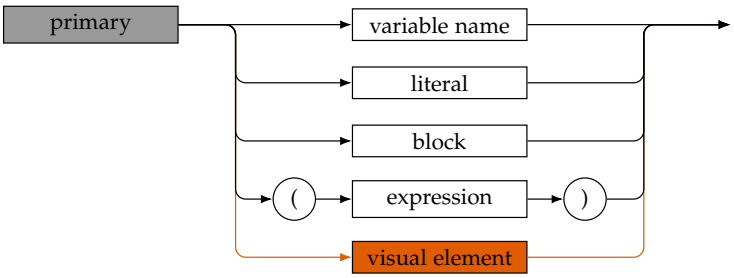


Abbildung 4.6: Erweitertes Syntaxdiagramm des *Primary*. Alle Syntaxdiagramme bauen auf [48, Anhang] auf. Hier und in den folgenden Syntaxdiagrammen sind weiße und graue Elemente gegenüber der gegebenen Syntax unverändert, orange wurden hinzugefügt.

zur Laufzeit überlassen, wie er die Nachricht behandelt. Diese Behandlung muss separat, z.B. mittels des *Message Channel Pattern* [56, S. 60 ff.], umgesetzt werden.

Zusätzlich bieten Nachrichtenobjekte die Möglichkeit Kontextinformationen zur Verfügung zu stellen. So ließen sich z.B. `this`-Bindings[7] (bzw. in Smalltalk `self`-Bindings) umsetzen.

In der Smalltalk-Syntax werden Nachrichten auf unäre, binäre oder n-äre Selektoren abgebildet. Selektoren sind wiederum durch *Identifier* und ggf. zusätzliche Parameter, *Primarys* oder weitere unäre Ausdrücke repräsentiert. So finden sich in der Beispielmethode die unäre Nachricht `new` als auch die n-äre Nachricht `color:`. Beide lassen sich durch ein visuelles Element ersetzen, eine visuelle Repräsentation ist jedoch nicht eindeutig[8]. Die Ersetzung des unären `new` ist in Listing 4.5 dargestellt.

Analog findet sich die Ersetzung der n-ären Nachricht `color:` in Listing 4.6 bzw. in Listing 4.7 bei einer Nachricht mit mehreren Parametern wie `r:g:b:`.

Auf syntaktischer Ebene handelt es sich dabei um die Ersetzung des Selektors und, sofern vorhanden, der Parameter. Die Änderungen an der Smalltalk-Syntax finden sich damit in Abbildung 4.7.

4.3.1.3 *Method* als visuelles Element

Ebenso wie ein Objekt oder eine Nachricht kann auch eine ganze Methode ausgetauscht werden. Dies hat primär den Hintergrund der Übersichtlichkeit [41, S. 109]: Betrachtet man die Darstellung eines größeren visuellen Elements, ist dieses schnell so komplex, dass es sich nicht mehr komfortabel in den normalen Programmfluss einordnen lässt Kapitel 7. Die Kapselung visueller Elemente als Methoden ermöglicht zudem ihre Abstraktion (*Means of Abstraction*, [1, S. 6]). Ebenso ermöglicht

[7] Das `this`-Binding in JavaScript: `https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/this#Die_bind_Methode`, letzter Zugriff am 12. Juni 2019.

[8] Es ist keine einheitliche Zuordnung von Darstellung zu Verhalten möglich, sodass dem Entwickler einer konkreten visuellen Nachricht je nach Domäne die Darstellung frei überlassen wird. Demzufolge werden alle folgenden visuellen Nachrichten als graue Rechtecke dargestellt.

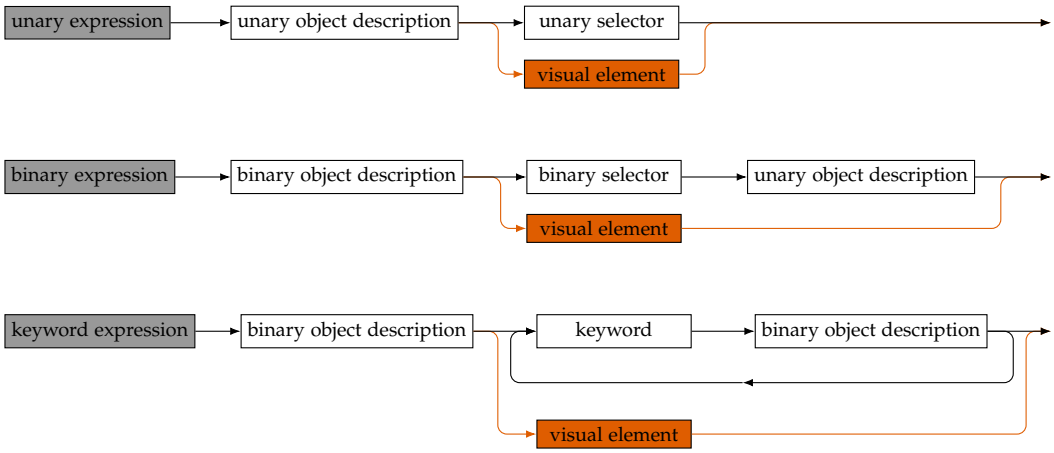


Abbildung 4.7: Erweitertes Syntaxdiagramm der *Message Expression*

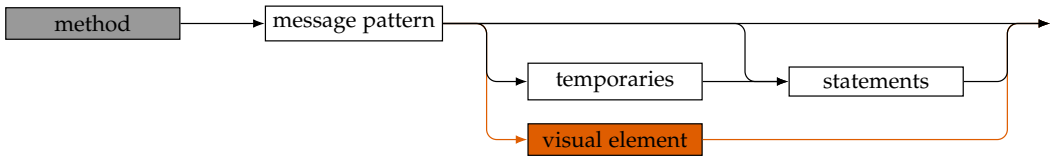


Abbildung 4.8: Erweitertes Syntaxdiagramm der *Method*

der sich ergebende Gestaltungsraum, welcher u. a. über die eindimensionale Darstellung von linearem Programmcode hinaus geht, die Kombination (*Means of Combination*, [1, S. 6]) visueller Elemente.

Um dieses Problem komplexer Darstellungen visueller Elemente zu lösen, lassen sich für dasselbe visuelle Element unterschiedliche Präsentation (gemäß Abschnitt 4.1.1) mit eigenen Interaktionen definieren und anzeigen. So hat eine Darstellung als eigene Methode (hohe Komplexität, hohe Komplexität, maximale Interaktivität) mehr Platz zur Verfügung als Darstellungen direkt im Programmfluss (minimale Darstellung, niedrige Komplexität, minimale Interaktivität) und muss keine Rücksicht auf umfließenden Text nehmen.

Die Ersetzung der Beispielmethode findet sich in Listing 4.8. Auch sie evaluiert weiterhin zu einem roten, sternförmigen Morph.

Methoden existieren als eigenes Syntaxelement *Method*. Die Veränderung an der Smalltalk-Syntax ist dabei in Abbildung 4.8 darstellt. Einzig das *Message Pattern*, also der Selektor, bleibt erhalten.

Eine solche vollständig visuelle Methode wird im nachfolgenden als *VisualElementMethod* bezeichnet. Sie lässt sich ganz normal in Programme einbetten und gibt immer ihr visuelles Element zurück.

4.3.2 Nicht visuell unterstütze Syntaxelemente

Rein technisch können visuelle Elemente an beliebiger Stelle im Programmcode eingefügt werden. Dabei ist weder der Abstraktionsgrad relevant noch ob es sich

um kontroll- oder datenflussbezogene Elemente handelt. So könnte jeder *Identifier* ersetzt werden. Dies bringt jedoch nahezu keinen Vorteil: Wie in Abschnitt 4.3.1.2 beschrieben, sind *Identifier* geschaffen, um Eindeutigkeit herzustellen. Hierfür ist eine textuelle Repräsentation völlig ausreichend.

Trotzdem lassen sich einige mögliche Anwendungsfälle visuell unterstützer *Identifier* finden. Diese wären z.B. *Babylonian Programming* [88] oder Internationalisierung (einfache und flexible Umstellung zwischen verschiedenen Darstellungen der Selektoren).

Ebenso nicht umgesetzt wurden *Characters*, *Symbols* und *Strings*, also direkte textuelle Repräsentationen. Auch hoch spezialisierte Programmierkonstrukte wie *Arrays*, bei welchen Vorteile visueller Repräsentationen nicht erkennbar sind, wurden nicht umgesetzt.

4.4 Anpassungen des Compilers

Dieser Abschnitt setzt die konzeptionellen Überlegungen aus Abschnitt 4.3 um. Dafür werden in Abschnitt 4.4.1 zunächst zusätzliche Anforderungen an die Erweiterung des Compilers gestellt bevor in Abschnitt 4.4.2 die Umsetzung der Syntaxelement diskutiert und beschrieben wird. Abschnitt 4.4.3 betrachtet noch einige Besonderheiten bei der Ausführung visueller Elemente.

4.4.1 Anforderungen an den Compiler

In Abschnitt 4.1.1 wurden die **LIB**-Prinzipien beschrieben. Aus diesen lassen sich folgende technische Anforderungen ableiten, die für alle umzusetzenden Syntaxelemente aus Abschnitt 4.3.1 gelten soll:

L = Live Visuelle Elemente müssen auch nach dem Kompilieren Smalltalk-Objekte bleiben und weiterhin Nachrichten empfangen und auf diese reagieren können, sowie durch den Nutzer manipulierbar bleiben.

I = Identitätserhaltend Visuelle Elemente dürfen nicht kopiert, serialisiert oder auf ihnen Programmcodetransformationen durchgeführt werden. Dies ist notwendig, um eine native Integration mit den vorhandenen Tools zu ermöglichen. So soll man z.B. bei weiterem Zugriff im Debugger den geschriebenen Programmcode, insbesondere das visuelle Element, erhalten.

B = Beliebiger Programmcode Auf visuellen Elementen wird eine Methode, die das Verhalten zur Ausführung definiert, aufgerufen. Visuelle Elemente werden dabei generisch als *Primary*, *Message Expression* und *Method* verwendet. Dies ermöglicht visuelle Elemente bequem als beliebige Syntaxelemente zu nutzen und legt nebenbei eine technische Grundlage um *Higher Order Messaging* [111] mit visuellen Elementen in Squeak/Smalltalk umzusetzen.

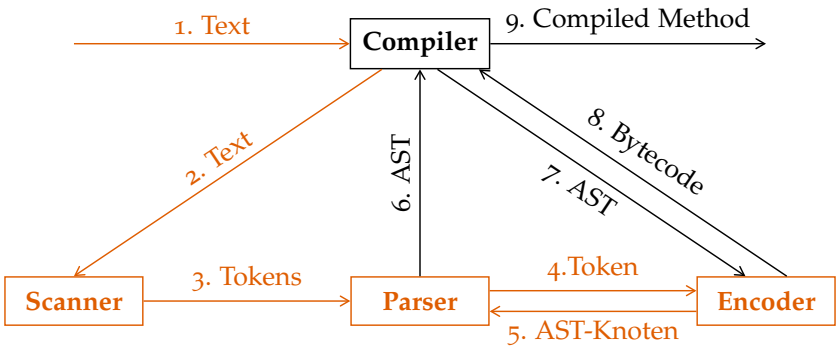


Abbildung 4.9: Anzupassende Komponenten des Compilers zur Verarbeitung von Text. Schwarze Elemente bleiben unverändert, orange wurden angepasst.

4.4.2 Kompilieren visueller Elemente

Der Compiler erhält nun, wie in Abschnitt 4.2.3 beschrieben, nicht mehr ausschließlich Strings, sondern Text. Somit reichen die Mittel für die Behandlung stringbasierter Objekte nicht mehr aus. Er muss so angepasst werden, dass er visuelle Elemente nativ verarbeiten kann, die unterschiedlichste syntaktische Rollen einnehmen können. Es werden folgende Änderungen notwendig:

1. Der Compiler und sein Subsystem müssen `Text` verarbeiten.
2. Der Scanner muss für das `SOH`-Zeichen, welches ein visuelles Element repräsentiert, das passende Token erzeugen.
3. Der Parser muss je nach Kontext das ggf. neue Token behandeln.
4. Der Encoder muss zum einen aus dem ggf. neuen Token den passenden AST-Knoten erzeugen und später aus dem ggf. neuen AST-Knoten den passenden Bytecode generieren.

Entsprechend werden Änderungen an Klassen und Artefakten gemäß Abbildung 4.9 notwendig.

Um visuellen Elementen die Möglichkeit zu geben über das Kompilieren informiert zu werden und ggf. sogar eingreifen zu können, werden in Tabelle 4.1 verschiedene Hooks definiert.

Für die Umstellung auf Text ist es ausreichend, an allen Stellen im Compiler die Konvertierung von Text zu String zu entfernen (`Text>>asString`). Da es sich dabei weder um eine konzeptionelle noch eine technische Limitierung handelt, wird sie hier nicht weiter erläutert.

4.4.2.1 *Primary* als visuelles Element

In Abbildung 4.9 sowie der zugehörigen Aufzählung wurden die anzupassenden Komponenten des Compilers identifiziert. Anhand dieser werden die notwendigen Anpassungen für die Nutzung visueller Elemente als *Primarys* beschrieben.

Tabelle 4.1: Hooks für unterschiedliche Zeitpunkte und ihr Zweck

Zeitpunkt	Hook	Zweck
Editierzeit	`inlineView:`, `editorView:`	Tooling, Interaktivität
Scanzeit	`scannedWith:`	Lese-Makros
Parsezeit	`parsedWith:`	Kontextinformationen, AST-Transformationen
Ausführungszeit	`value`, `valueInContext:`, `recieveVisualMessage:`	Effekt des visuellen Elements

Anpassungen am Scanner Zunächst wird der Typentabelle des Scanners das `SOH` hinzugefügt. Liest der Scanner nun ein `SOH`, wird das visuelle Element aus dem Textattribut gelesen und als Token mit dem Typ `#visualElement` an den Parser übergeben. Der veränderte Programmcode des Scanners findet sich in Listing 4.15.

Anpassungen am Parser Anschließend muss die Behandlung von *Primarys* im Parser erweitert werden: Neben Variablennamen, Literalen und Blöcken ergibt sich der neue Fall *VisualElement*. Dabei wird das visuelle Element analog zu Literalen enkodiert. Die Änderungen am Parser sind in Listing 4.16 angehängt.

Anpassungen am Encoder Für das enkodierte visuelle Element muss weiterhin der passende Teil-AST erstellt werden. Dafür lassen sich zwei Möglichkeiten implementieren: Entweder bestimmt der Parser den erstellten Teil-AST oder das visuelle Element entscheidet selbst, wie dieser aussieht. Beide Varianten unterscheiden sich in der Dynamik, die das visuelle Element im Bytecode schaffen kann. Sie werden im Folgenden erläutert.

Der Parser bestimmt die Erstellung des Teil-AST Dieses Vorgehen ist analog zu der Behandlung von Literalen in Squeak/Smalltalk. Somit ist die Integration nativ und auch für Entwickler zukünftiger visueller Elemente mit weniger Aufwand verbunden, da dieser sich nicht explizit um die Erstellung eines AST kümmern muss. Stattdessen wird während des Parsens eines visuellen Elements eine dedizierte `ParseNode`, eine `VisualElementNode`, erstellt und an der entsprechenden Stelle in den AST eingehängt. Ihre Erzeugung findet sich in Listing 4.17. Die `VisualElementNode` ist eng an eine einfache `LiteralNode` angelehnt, die sich ergebende Vererbungshierarchie ist in Abbildung 4.10 dargestellt.

Betrachtet man wieder die Ersetzung der Farbe aus Listing 4.3 ergibt sich der AST aus Abbildung 4.11. Der Fokus liegt auf dem orange markierten Teil-AST, da es sich bei `MethodNode`, `BlockNode` und `ReturnNode` zwar um relevante Teile für die Erstellung ausführbaren Programmcodes handelt, diese sich jedoch bei der Anpassung einer Nachricht der Methode nicht verändern.

Allerdings muss ein visuelles Element mächtiger sein als ein einfaches Literal. Es muss festgelegt werden, wie man den Bytecode so erzeugt, dass der **B**-Teil der **LIB**-

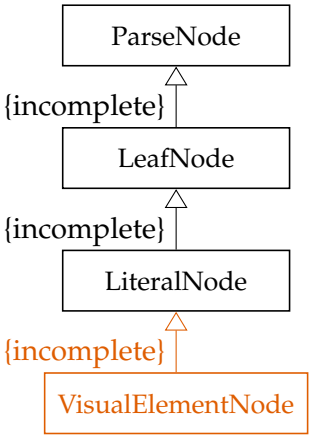


Abbildung 4.10: Einordnung einer `VisualElementNode` als Spezialisierung von `ParseNode` in der Vererbungshierarchie (UML-Klassendiagramm)

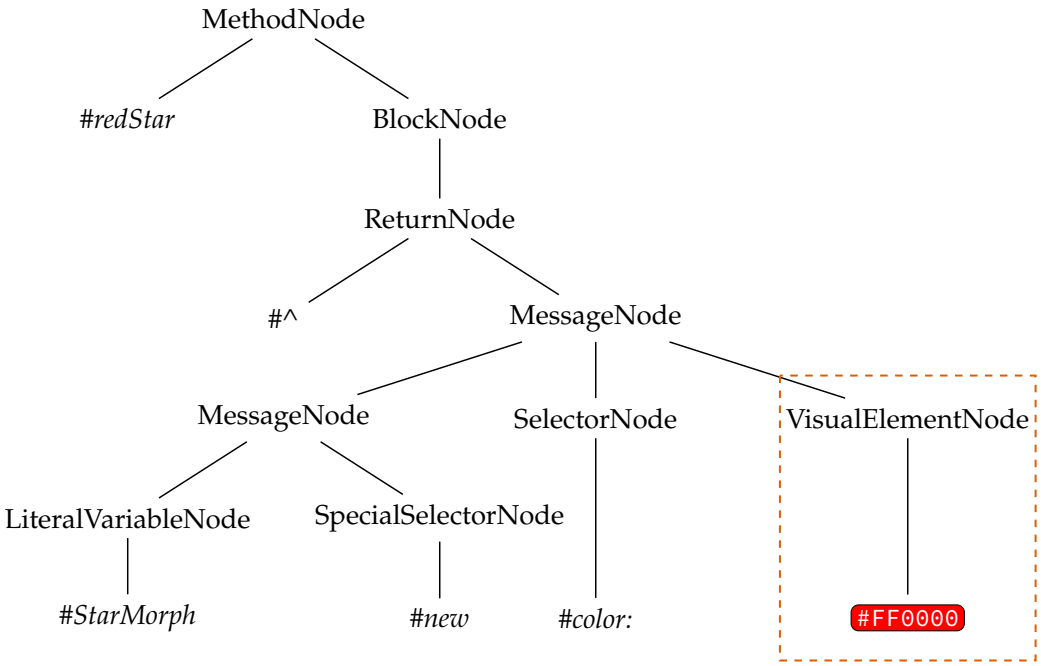


Abbildung 4.11: AST mit einer Farbe als visuelles Element

Prinzipien erfüllt ist: Dafür wird auf visuellen Elementen eine `valueInContext:`-Methode, die das Verhalten zur Ausführung definiert, aufgerufen (gemäß Tabelle 4.1). Entsprechend wird für ein visuelles Element Bytecode erzeugt, der dem Programmcode `valueInContext: thisContext` entspricht. Dies findet sich in Listing 4.18.

Der sich ergebende veränderte Bytecode ist in Listing 4.9 aufgeführt, der genau so ohne weitere Anpssungen ausgeführt werden kann. Das visuelle Element wird dabei zunächst als weiteres Literal (`literal3`) aufgenommen, außerdem findet sich bei den Literalen noch das `#valueInContext:`-Symbol (`literal4`). Dann wird zunächst das visuelle Element (`59`) und darauf folgend der aktuelle Kontext auf den Stack gepusht (`60`), sodass dann der `valueInContext:`-*Message Send* (`61`) ausgeführt werden kann. Die Implementierung ähnelt der einer `MessageNode`.

Listing 4.9: Bytecode mit einer Farbe als visuellem Element. Hier und im folgenden Bytecode sind schwarze Zeilen unverändert, orange wurden angepasst.

```
literal1 #color:
literal2 #StarMorph=>StarMorph
literal3 #FF0000
literal4 #valueInContext:
literal5 #redStar
literal6 #Example=>Example
57  '<41> pushLitVar: #StarMorph=>StarMorph'
58  '<CC> send: new'
59  '<22> pushConstant: #FF0000'
60  '<89> pushThisContext: '
61  '<E3> send: valueInContext:'
62  '<E0> send: color:'
63  '<7C> returnTop'
```

Zusätzlich muss gesichert werden, dass die `VisualElementNode` nicht im Zuge von Optimierungen nicht in den Bytecode geschrieben wird, wie es bei normalen `LiteralNodes` passieren kann. Dies wird durch Überschreiben der Methode `VisualElementNode >> #emitCodeForEffect:encoder` zugesichert, angehängt in Listing 4.19.

Das visuelle Element bestimmt die Erstellung des Teil-AST Diese Variante hingegen ermöglicht einem visuellen Element einen beliebigen Teil-AST einzufügen. Dabei nutzt das visuelle Element gemäß Tabelle 4.1 statt eines Hooks zur Ausführungszeit (`value` bzw. `valueInContext:`) einen Hook zur Zeit des Parsens (`parsedWith:`), der eine Erweiterung des ASTs ermöglicht. Dieser Ansatz ist mächtiger, da er das Einfügen eigener Kompilierinstruktionen ermöglicht, statt nur die Ausführung zu verändern. Zur Parsezeit muss somit keine spezielle `ParseNode` erstellt werden, es werden schon vorhandene `ParseNodes` genutzt.

Mit der Erstellung eines beliebigen Teil-AST durch das visuelle Element ergeben sich die folgenden Auswirkungen:

1. Die Erstellung des Teil-AST durch den Parser, ist nur ein Spezialfall dieser Umsetzung. So können z. B. einfache visuelle Elemente, die nur Literale darstellen, z. B. Zahlen, auch nur eine `LiteralNode` zurückgeben.

2. Visuelle Elemente können nicht nur als R-Werte, sondern auch als L-Werte umgesetzt werden. D. h. ein visuelles Element kann auch auf der linken Seite einer Zuweisung stehen.

3. Visuelle Elemente müssen die Erstellung des Teil-AST an Subelemente propagieren. Finden sich also geschachtelte visuelle Elemente, müssen die inneren aktiv von den äußeren behandelt werden.

4. Die Hauptaufgabe eines visuellen Elements besteht somit darin, seinen Teil-AST zu produzieren. Diese Aufgabe muss zwangsläufig von dem Entwickler eines neuen visuellen Elements übernommen werden.

Ermöglicht man also einem visuellen Element die Erstellung des AST, wird dieser flexibler. Er kann klarerer und domänenspezifischer designed werden, ohne eine generische `VisualElementNode` nutzen zu müssen. Dies könnte partiell zu einem Performancegewinn führen, da die konkreten `ParseNodes` für ihren speziellen Anwendungsfall optimiert sind.

Der Nachteil ist jedoch, dass Ersteller eigener, domänenspezifischer Elemente sich nicht nur um Semantik und visuelles Design, sondern auch um die konkrete Erstellung des Teil-AST kümmern müssen. Es wurde daher zunächst die Variante, dass der Parser die Erstellung des Teil-AST bestimmt, umgesetzt.

4.4.2.2 *Message Expression* als visuelles Element

Gemäß Abschnitt 4.2.3 wurden visuelle Elemente jeglicher Art als genau ein Zeichen dargestellt. Handelt es sich bei diesen nun aber um eine *Message Expressions* mit Parametern, finden sich die Parameter nicht explizit im Programmcode. Stattdessen hält ein visuelles Element seine Parameter selber.

Betrachtet man wieder die Beispielmethode mit Farbrepräsentation via n-ärer Nachricht aus Listing 4.7, würde der Scanner nur ein `SOH` lesen und ausschließlich das `#visualElement` Token erzeugen. Dies ist in Abbildung 4.12 dargestellt, es finden sich in der ersten Zeile die Sicht des Entwicklers und damit Eingabe des Scanners, in der zweiten Zeile der String auf dessen Grundlage dieser die Tokens erzeugt und in der letzten Zeile die Tokentypen der letztendlich erzeugten Tokens.

Damit sind jedoch die Parameter einer visuellen Nachricht verloren gegangen, der Inhalt der in Orange gestrichelten Box ist also unvollständig. Zwar werden auch die Tokens, also die visuelle Nachricht und damit die Parameter, an den Parser weitergereicht, sie wurden jedoch nie gescannt. Die Instanziierung eines neuen Parsers für die Parameter löst dieses Problem nicht, da dies zwar für Parameter die Literale sind funktioniert, jedoch nicht, wenn noch Bindings, i. d. R. für Variablen,

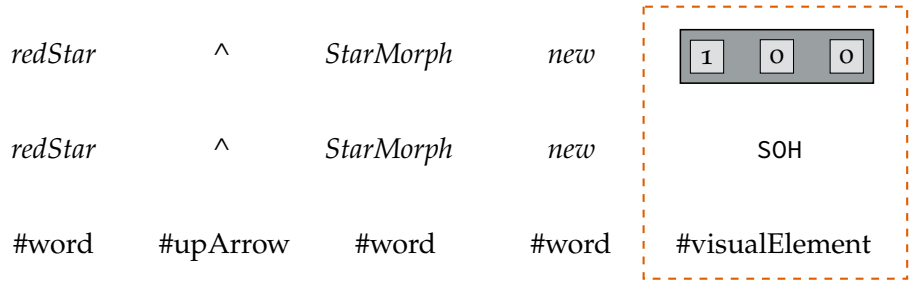


Abbildung 4.12: Die Tokens der Beispielmethode mit visuellem Element als n-äre Nachricht mit dem Scannen aus Abschnitt 4.4.2.1

aufgelöst werden müssen. Da bekannt sein muss, welchen konkreten Wert ein Binding beim Auftreten hat, ist der Kontext des Auftretens relevant.

Um den Parser nicht grundlegend umzubauen, muss er demzufolge ein Format erhalten in welchem er alle notwendigen Informationen vorfindet. Es muss also zunächst das Scannen angepasst werden, ehe man sich um das Parsen der visuellen Nachrichten kümmern kann.

Anpassungen am Scanner Das einfache Scannen eines visuellen Elements, wie in Abschnitt 4.4.2.1 beschrieben, wird so erweitert, dass eine visuelle Nachricht die Möglichkeit hat, ihr Format an die bestehende Nachrichteninfrastruktur anzugleichen. Dafür werden die Lese-Makros aus Tabelle 4.1 benötigt, die in [51, Kapitel 17, Read-Macros] beschrieben werden als

> *„Macros get hold of the program when it has already been parsed into Lisp objects by the reader, and read-macros operate on a program while it is still text. However, by invoking read on this text, a read-macro can, if it chooses, get parsed Lisp objects as well.“*

Wird während des Scannens ein visuelles Element, also ein `SOH`, gelesen, wird diesem zusätzlich die Möglichkeit gegeben, den aktuellen Stream wie folgt zu manipulieren: Die Parameter werden, getrennt durch das Paragraphenzeichen `$` [9], hinter das `SOH` auf den Stream geschrieben und erhalten den Tokentyp `#vESeparator` [10]. Der zugehörige Programmcode findet sich in Listing 4.20.

Damit bleiben die mit der visuellen Nachricht mitgegebenen Parameter erhalten und der Inhalt der in Orange gestrichelten Box aus Abbildung 4.13 verändert sich zu Abbildung 4.12.

Dieses Vorgehen trägt auch den Fall einer visuellen Nachricht, die nicht nur Literale als Parameter erhält (die mit der Instanziierung eines neuen Parsers nicht behandelbar ist). Die orange gestrichelte Box hat in diesem Fall den vollständigen Inhalt aus Abbildung 4.14.

[9] Das Paragraphenzeichen ist wiederum ein Sonderzeichen, welches keine spezifische Verwendung im System findet und auch in Bezeichnern nicht verwendet werden darf [90, Grammatik].

[10] Die Kurzform `vESeparator` steht dabei für *visualElementSeparator*.

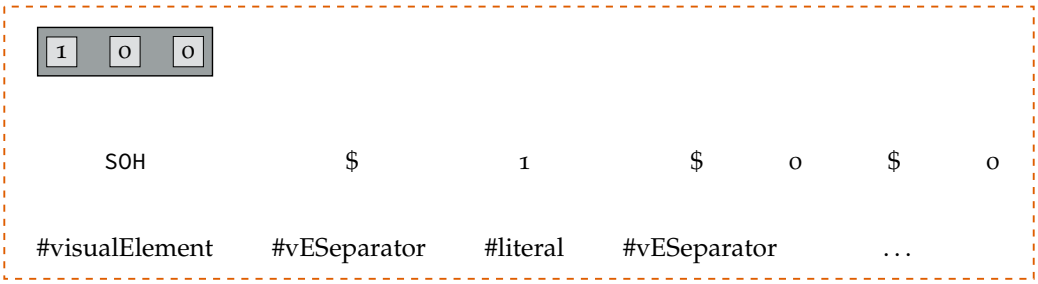


Abbildung 4.13: Die Tokens der Beispielmethode einer n-ären Nachricht mit Parametern mit dem erweiterten Scannen aus Abschnitt 4.4.2.2

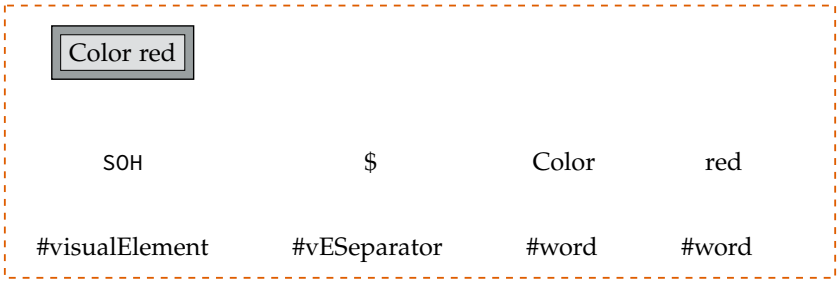
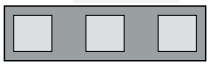


Abbildung 4.14: Die Tokens der Beispielmethode einer n-ären Nachricht mit Parametern, die keine Literale sind, mit dem erweiterten Scannen aus Abschnitt 4.4.2.2

Die Lese-Makros werden ausschließlich während des Kompilierens genutzt, nicht jedoch bei Zugriff auf den Programmcode zu einem späteren Zeitpunkt mit anderen Tools.

Anpassungen am Parser Der Parser kann diesen veränderten Stream jetzt nahezu wie gewohnt, angelehnt an n-äre Nachrichten, verarbeiten. Es wird ein *Message Pattern* gelesen, nur dass die Parametertrenner nicht Teil des Selektors, also einzelne *Keywords*, sind, sondern explizite `$`. Zwischen diesen finden sich die, bereits geparsten, Parameter. Die Veränderungen am Parser sind in Listing 4.21 angehängt.

Dabei wird eine `Message` erstellt, welche ganz normal Empfängerklasse und Argumente gesetzt hat, nur der Selektor unterscheidet sich. Statt eines Symbols (z.B. `#r:g:b:` oder `#color:`) bildet nun das visuelle Element den Selektor (z.B.  oder ). Damit werden Anpassungen am Encoder vermieden.

Anpassungen der Ausführung Es ergeben sich jedoch Anpassungen bei der Ausführung, es verbleibt die Zuordnung dieser neuen Nachricht zu einer Methode. Dies könnte in der VM passieren indem man die Zuordnung über Symbole hinaus erweitert, ist aus den folgenden Gründen aber wenig sinnvoll:

1. Änderungen sollen so lokal und begrenzt wie möglich erfolgen.
2. Die VM ist ein zweites sehr komplexes System.

3. Veränderungen am Image[11] lassen sich leicht an andere ausliefern und über Versionskontrollsysteme verteilen, eine Veränderung an der VM benötigt eigene Patches und ein damit verbundenes Kompilieren sämtlicher VM-Quellen.

Die Zuordnung wird somit über `Object >> #doesNotUnderstand:` im Image gelöst. Dem Empfänger wird mittels des Hooks `recieveVisualMessage:` zur Ausführungszeit gemäß Tabelle 4.1 die vollständige Nachricht zugestellt[12]. Das veränderte `doesNotUnderstand:` findet sich in Listing 4.22. Die Auflösung der Nachricht folgt dann dem *Actor*-Modell, wie es schon in Smalltalk-72 funktionierte [40, S. 328], der Empfänger ist also selbst für den Umgang damit verantwortlich. Die Eigenschaften, nach denen eine Nachricht aufgelöst wird, sind theoretisch beliebig. Es bieten sich aber visuelle Eigenschaften, z.B. Farbe oder Größe, besonders an.

4.4.2.3 *Method* als visuelles Element

In Squeak/Smalltalk finden sich `CompiledMethods`[13] als einziges Konstrukt, welches die VM verarbeiten kann. Eine Einführung einer grundsätzlich neuen Methodenklasse für visuelle Methoden würde auch hier ggf. zu Änderungen an der VM führen.

Wie in Abschnitt 4.3.1.3 beschrieben, handelt sich bei visuellen Elementen als Methoden jedoch nur um eine andere Darstellung des visuellen Elements. Daher wird eine `VisualElementMethod` eingeführt, die von `CompiledMethod` erbt, ihre Einordnung in das Gesamtsystem findet sich in Abbildung 4.15. Zusätzlich erhält eine `VisualElementMethod` gegenüber der normalen `CompiledMethod` noch einige Bequemlichkeitsmethoden, um z.B. direkt das visuelle Element abzufragen.

Für jede `VisualElementMethod` existiert somit eine Darstellung als `CompiledMethod`. Nur der `CodeBrowser` erzeugt für den Entwickler eine andere Repräsentation. So wird die Methode aus Listing 4.10 intern zu Listing 4.11 umgewandelt. Der Programmcode zur Umwandlung ist in Listing 4.23 angehängt.

Listing 4.10: `VisualElementMethod` der Beispielmethode

```
1 Example >> #redStar
2     ★
```

[11] Image bezeichnet den Bereich des Squeak/Smalltalk-Systems, welcher nicht von der VM dargestellt wird. Diese Notation folgt den Bezeichnungen auf `https://squeak.org/downloads/`, letzter Zugriff am 14. Juni 2019.

[12] Zusätzlich wird das visuelle Element, also die visuelle Nachricht, in dem `doesNotUnderstand:` darüber informiert, dass es gleich aufgerufen wird. Dieser Hook ist z.B. einsetzbar für Programmanalysen oder die Erstellung von Aufrufbäumen.

[13] Es existiert noch die Klasse `CompiledBlock`, die ebenso wie `CompiledMethod` von `CompiledCode` erbt, jedoch vom modernen `EncoderForV3PlusClosures` nicht mehr unterstützt wird.

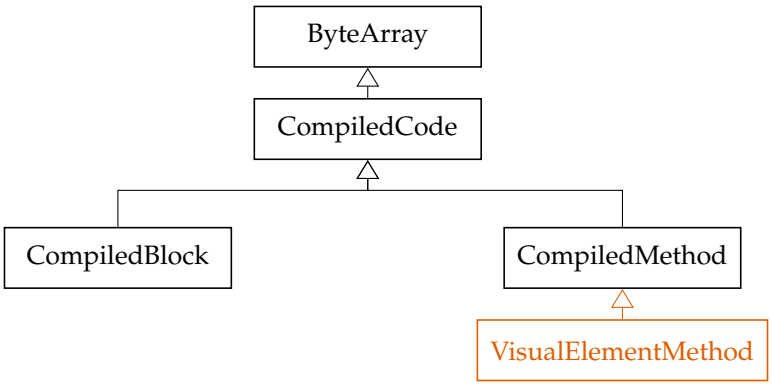


Abbildung 4.15: Einordnung einer `VisualElementMethod` als Spezialisierung von `ByteArray` in der Vererbungshierarchie (UML-Klassendiagramm)

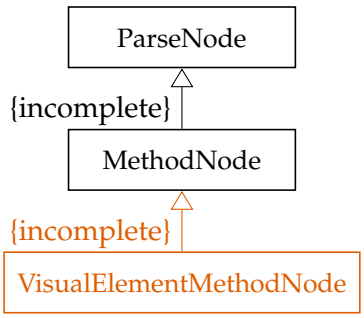


Abbildung 4.16: Einordnung einer `VisualElementMethodNode` als Spezialisierung von `ParseNode` in der Vererbungshierarchie (UML-Klassendiagramm)

Listing 4.11: `CompiledMethod`-Äquivalent der Beispielmethode

```
1 Example >> #redStar
2    <VisualElementMethod>
3    ^ ★
```

Anpassungen am Scanner Am Beginn der Methode befindet sich das Pragma `<VisualElementMethod>`, welches diese als spezielle `VisualElementMethod` kennzeichnet. Beim Scannen wird dieses Pragma wie alle anderen Pragmas behandelt, was keine Anpassungen am bestehenden Scanner notwendig macht.

Anpassungen am Parser Die Unterscheidung erfolgt wieder beim Parsen: Statt einer regulären `MethodNode` wird eine `VisualElementMethodNode` erzeugt, sofern das entsprechende Pragma nach der AST-Konstruktion gefunden wurde. Die dafür notwendigen Änderungen am Encoder finden sich in Listing 4.24. Ihre Einordnung in die Klassenhierarchie ist in Abbildung 4.16 dargestellt.

Anpassungen am Encoder Somit ergibt sich die weitere Behandlung im Encoder und der Bytecode gemäß der Behandlung von *Primarys* aus Abschnitt 4.4.2.1. Hier sind also keine Anpassungen notwendig. Der Bytecode findet sich in Listing 4.12.

Listing 4.12: Bytecode der Beispielmethode, in dem die komplette `CompiledMethod` durch eine `VisualElementMethod` ersetzt wurde

```
literal1 ★
literal2 #valueInContext:
literal3 an AdditionMethodState (8450)
literal4 #Example=>Example
41  '<22> pushConstant: ★'
42  '<89> pushThisContext: '
43  '<E3> send: valueInContext:'
44  '<E0> send: color:'
45  '<7C> returnTop'
```

Das einzige variable Element ist hierbei immer das Literal an erster Stelle, also das visuelle Element, der Rest des Bytecodes ergibt sich immer gleich aus dem generierten Programmcode der `CompiledMethod`. Damit ist eine volle Interoperabilität der `VisualElementMethod` mit der `CompiledMethod` gegeben. Ihre Ausführung ist somit ebenfalls äquivalent zu der von *Primarys*.

4.4.3 Besonderheiten bei der Ausführung visueller Elemente

Mit den neuen visuellen Elementen und dem dabei erzeugten Bytecode wurde die Ausführung visueller Elemente jeweils in den entsprechenden Abschnitten behandelt. Sie hat jedoch einige allgemeine Besonderheiten und Herausforderungen, die in diesem Abschnitt beleuchtet werden.

4.4.3.1 Kontextsensitive Ausführung

Für visuelle Elemente wird Bytecode generiert, welcher im Effekt die Nachricht `valueInContext:` mit dem Argument `thisContext` an das visuelle Element schickt. Dieser Programm- bzw. Bytecode setzt sich also aus zwei Bestandteilen zusammen:

1. Dem Aufruf von `valueInContext:` auf dem visuellen Element.
2. Die Mitgabe des `thisContext` an das visuelle Element.

Dem Aufruf von `valueInContext:` ermöglicht es den **B**-Teil der **LIB**-Prinzipien umzusetzen. Ein visuelles Element implementiert eine `valueInContext:` Methode, in welcher es beliebigen Programmcode ausführen kann. Hier können je nach Belieben die passenden Informationen zurückgegeben oder Seiteneffekte ausgeführt werden. `valueInContext:` ruft dabei `value` auf, sofern keine Implementierung vor-

handen ist. In der Implementierung von `value` auf `Object`[14] gibt diese einfach den Empfänger selber zurück.

Die Mitgabe des `thisContext` ist notwendig, damit sich visuelle Elemente für den Entwickler wie bekannter textueller Programmcode verhalten. Dafür müssen sie auf Kontextinformationen zugreifen können. Das umfasst den Zugriff auf temporäre Variablen bis hin zu Zugriff auf Informationen über die Methode, in der sich das visuelle Element befindet.

Eine ähnliche Herausforderung findet man bei `BlockClosures`, welche auch aus ihrem Kontext Informationen für sich erfassen müssen. Schon bei diesen hat sich die Frage ergeben, ob man dynamisches oder statisches (lexikalisches) Scoping [30, S. 1] nutzt. Man hat sich für statisches Scoping entschieden.

Visuellen Elemente hingegen wird einfach der aktuelle `thisContext` zur Verfügung gestellt, dies entspricht dynamischem Scoping. Ein explizites Erfassen von Kontext wie bei einer `BlockClosure` würde aufgrund minimaler visueller Repräsentation direkt im Programmcode einen deutlichen visuellen Overhead erzeugen. Dynamisches Scoping sorgt zudem nicht für zusätzlichen Verlust von Programmverständnis, da ein visuelles Element ohnehin beliebigen Programmcode ausführen kann. Außerdem ist es wesentlich einfacher umzusetzen als lexikalisches Scoping.

4.4.3.2 Kopie und Referenz bei der Ausführung

In Listing 4.4 wurde `StarMorph new` durch ein visuelles Element ersetzt. Im zugehörigen Bytecode findet sich dann genau der selbe `StarMorph`, der vom Entwickler in der Methode eingefügt wurde (analog zu der Farbe aus Listing 4.2). Wird die Methode mittels `Example new redStar openInWorld` ausgeführt, ergeben sich zwei Möglichkeiten:

1. Der Aufrufer erhält eine Kopie des Elements, das sich im Bytecode befindet. Der `identityHash` beider Objekte ist ungleich.

2. Der Aufrufer erhält das visuelle Element, das sich auch im Bytecode befindet. Der `identityHash` beider Objekte ist gleich.

An dieser Stelle wird dasselbe visuelle Element zurückzugeben. Dabei handelt es sich um den **I**-Teil der **LIB**-Prinzipien: Es wird nicht nur dasselbe visuelle Element ausgeführt, sondern es kann auch dasselbe Element manipuliert werden. Dasselbe visuelle Element kann zudem leicht durch den Entwickler mit einem zusätzlichen `copy` in das gleiche visuelle Element überführt werden kann. Dies ist in Listing 4.13 dargestellt.

Die Nutzung desselben visuellen Elements bietet zudem noch die Möglichkeit selbstmodifizierender visueller Elemente sowie ein implizites Caching.

[14] `Object` ist die Wurzelklasse aller anderen Klassen im Squeak/Smalltalk-System. Einzige Ausnahme bildet dabei die Klasse `ProtoObject`, welche jedoch in der Regel nicht von Anwendern genutzt wird.

Listing 4.13: Beispielmethode mit zusätzlichem `copy`

```
Example >> #redStar
   ^ ★ copy
      color: Color red
```

Selbstmodifizierende visuelle Elemente Selbstmodifizierende visuelle Elemente ermöglichen selbstmodifizierenden Programmcode, wie er schon von Konrad Zuse beschrieben wurde [118, S. 8]. So kann ein visuelles Element z. B. in Abhängigkeit seiner Farbe in `valueInContext:` bestimmten Programmcode ausführen, diesen aber gleichzeitig über den weiteren Programmcode verändern.

Implizites Caching Wird eine Methode nicht nur einmal, sondern mehrfach ausgeführt, entsteht ein implizites Caching. So ist z. B. das Ändern der Farbe eines Morphs immer auch mit dem Propagieren der Änderungen (`self changed`) verbunden, sofern sich alte und neue Farbe unterscheiden. Ist diese einmal beim ersten Aufruf auf die neue Farbe gesetzt, wird mit den folgenden Aufrufen kein `self changed` mehr gesendet.

4.5 Evaluation

In diesem Abschnitt wird der Ansatz zur Erweiterung des Squeak/Smalltalk-Compilers evaluiert. Zunächst wird in Abschnitt 4.5.1 der Performanceeinfluss visueller Elemente betrachtet. Danach wird in Abschnitt 4.5.2 die Integration in bestehende Werkzeuge analysiert. Zuletzt werden in Abschnitt 4.5.3 die Limitierungen des Ansatzes und der aktuellen Umsetzung betrachtet.

4.5.1 Performanceanalyse

Zielgruppe Für eine sinnvolle Nutzung visueller Elemente ist erforderlich, dass diese sowohl performant genug ist für

1. Entwickler, die sie in ihrem Programmcode verwenden, als auch
2. Nutzer, die den Programmcode mit visuellen Elementen ausführen.

Messgrößen Dazu werden im Folgenden die durchschnittliche Kompilierungs- und Ausführungszeit von Methoden mit visuellen Elementen mit rein textuellen Darstellungen verglichen.

Benchmarking Für das Benchmarking werden folgende fünf Methoden aus Abschnitt C betrachtet. Die Ergebnisse wurden mittels `[ "..." ] benchFor: 5 seconds` ermittelt.

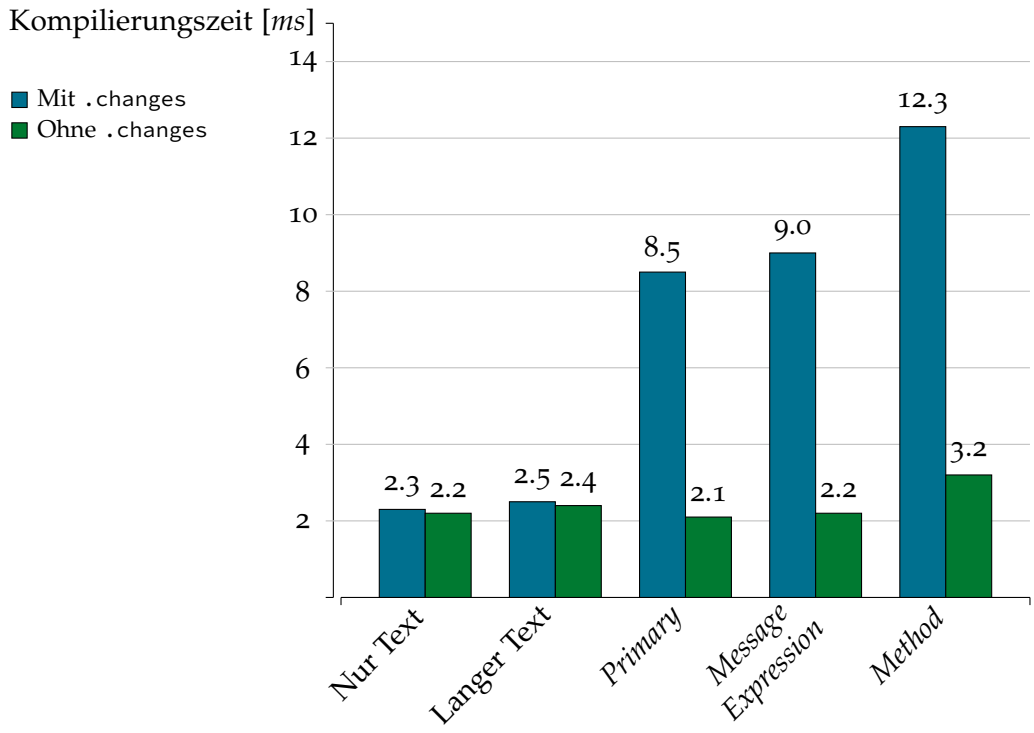


Abbildung 4.17: Kompilierungszeiten einer Methode, in der jeweils unterschiedliche Syntaxelemente durch ein visuelles Element ersetzt wurden

1. Methode ohne visuelle Elemente (Listing 4.26)
2. Methode in Langform ohne visuelle Elemente (Listing 4.27)
3. Methode mit visuellem Element als *Primary* (Listing 4.28)
4. Methode mit visuellem Element als *Message Expression* (Listing 4.29)
5. Methode mit visuellem Element als *Method* (Listing 4.30)

System Alle Messungen wurden auf einem Dell XPS 9370 (CPU: Intel® Core™ i7-8550U CPU @ 1.80GHz; 16GB RAM) mit einem 64-Bit Ubuntu 19.10 und Linux-Kernel Version 5.0.0-17-generic durchgeführt. Es wurde Squeak-Version *5.2* und VM-Version *5.0-201810071412* verwendet.

4.5.1.1 Performance beim Kompilieren

Für den Entwickler ist relevant, dass die Performance während des Kompilierens einigermaßen konstant bleibt. Die blauen Balken aus Abbildung 4.17 beschreiben die Kompilierungszeiten der fünf Benchmarkingmethoden.

Es fällt auf, dass die Kompilierungszeit deutlich zunimmt, wenn sich ein visuelles Element im Programmcode befindet. Das lässt sich wie folgt erklären: Beim Erzeugen einer neuen Version einer Methode wird in Squeak immer die `.changes`-Datei

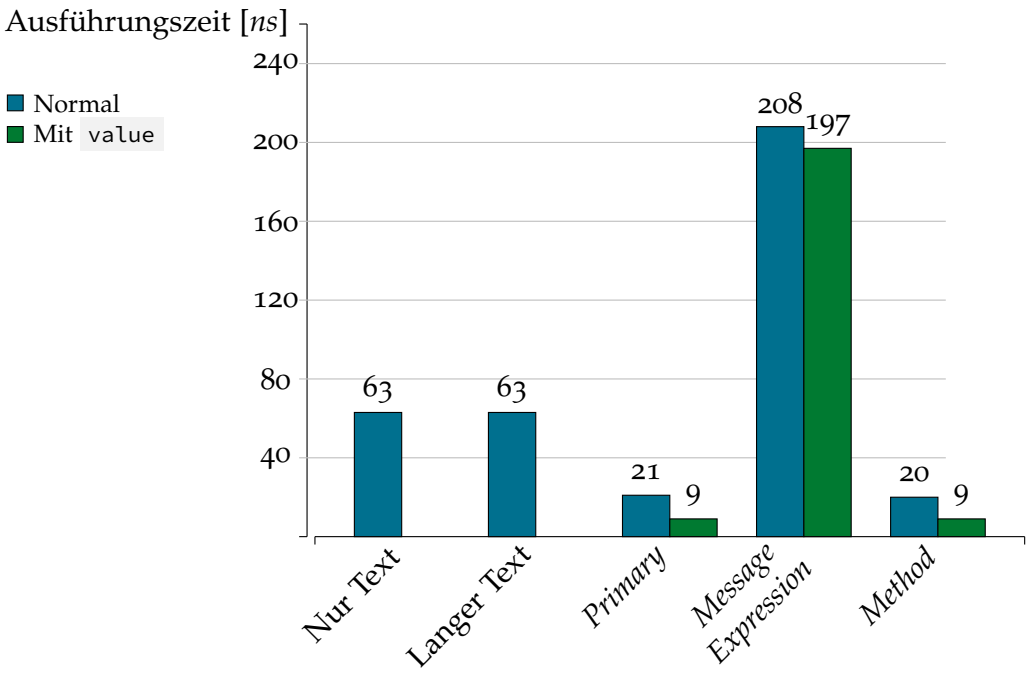


Abbildung 4.18: Ausführungszeiten einer Methode, in der jeweils unterschiedliche Syntaxelemente durch ein visuelles Element ersetzt wurden

geschrieben. Sie speichert alle bisherigen Versionen einer Methode und ermöglicht auf alte Versionen von Methoden zurückzugehen. Da diese jedoch als explizite Datei im Dateisystem vorliegt und der Speicherstring eines visuellen Elements wesentlich länger als seine textuelle Repräsentation ist, liegt dort ein Flaschenhals vor. Deaktiviert man das Schreiben in die `.changes`-Datei, ergeben sich die grünen Balken aus Abbildung 4.17.

Damit gleichen sich die Kompilierungszeiten der ersten vier Methoden nahezu vollständig an, Methoden mit visuellem Element als *Primary* oder *Method* sind sogar minimal schneller. Das Kompilieren einer `VisualElementMethod` ist etwas langsamer, was sich wahrscheinlich auf die Konvertierung zu einer `CompiledMethod` zurückführen lässt.

Zusammenfassend lässt sich sagen, dass die Performance beim Kompilieren eines visuellen Elements als *Method* leicht abnimmt, in den anderen beiden Fällen sorgen die, durch die visuellen Elemente erzeugten, kürzeren Methoden jedoch für einen minimalen Performancegewinn.

4.5.1.2 Performance beim Ausführen

Wesentlich kritischer als das meist einmalige Kompilieren ist die Ausführungszeit. Die Performancedaten werden analog zum vorherigen Abschnitt erhoben.

Die Ausführung einer Methode mit visuellem Element als *Primary* bzw. als *Method* ist deutlich schneller als die Ausführung einer Methode in textueller Form, dargestellt durch die blauen Balken in Abbildung 4.18. Dies lässt sich dadurch

erklären, dass bei der textuellen Form das Live- Objekte noch nicht vorliegt, bei den visuellen Varianten aber schon. Zudem handelt es sich bei der Erstellung von Morphs um einen relativ aufwendigen Prozess, so dauert schon alleine die Erstellung eines neuen Morphs (`Morph new`) rund 62 *ns*. Auch hier wird durch die Live-Objekte ein implizites Caching eingebaut.

Trotzdem dauert der Aufruf einer Methode mit visuellem Element als *Primary* noch 21 *ns*. Dies lässt sich auf die Materialisierung des `thisContext` zurückführen: Nutzt man zur Ausführung statt `valueInContext:` nur `value`, wie in Tabelle 4.1 als Alternative gegeben, ergeben sich die grünen Balken aus Abbildung 4.18 (für die rein textuellen Methoden ist dies selbstverständlich nicht möglich), die Zeiten verringern sich um rund 11 *ns*. Dies scheint genau die Zeit zu sein, die das Materialisieren des Kontextes benötigt. Bei der Nutzung visueller Elemente als *Primary* macht dies über 50 % der Performance aus.

Wesentlich langsamer ist jedoch das Ausführen visueller Elemente als *Message Expression* im Programmcode. Dies lässt sich über die Auflösung der visuellen Nachricht über das `doesNotUnderstand:` erklären: Normalerweise wird eine Nachricht durch die VM zugeordnet, nun muss ein zusätzlicher Wechsel aus C nach Smalltalk geschehen, dort die Zuordnung erfolgen, ehe diese neue Nachricht wieder über die VM zugeordnet werden kann. Dieser zusätzliche Wechsel ist schon ohne visuelle Nachricht sehr teuer. Vergleicht man die Zeit, die für die Ausführung einer rein textuellen Nachricht benötigt wird, welcher der Empfänger interpretieren kann (rund 6 *ns*), und die Zeit für eine Nachricht, die der Empfänger nicht interpretieren kann und erst in seiner `doesNotUnderstand:` auf eine passende Nachricht transformiert (126 *ns*), ist letzteres rund einen Faktor 21 langsamer. Genau diesen Faktor kann man bei den grünen Balken in Abbildung 4.18 zwischen der Ausführungszeit einer Methode mit visuellem Element als *Primary* und einem visuellen Element als *Message Expression* beobachten.

Es lässt sich festhalten, dass die Performance beim Ausführen eines visuellen Elements als *Message Expression* abnimmt, dafür in den anderen beiden Fällen mit visuellen Elementen jedoch deutlich zunimmt. Benötigt man den Kontext nicht, wird sie sogar noch wesentlich schneller.

4.5.2 Integration in Debugger und Decompiler

So wie andere Entwicklungsumgebungen, lebt auch Squeak/Smalltalk in großem Maße von den Werkzeugen, die dem Nutzer zur Verfügung gestellt werden. Eine besondere Stellung nimmt der Debugger in Squeak/Smalltalk ein (verglichen mit den Möglichkeiten für C wird die Debuggingerfahrung als hervorragend bezeichnet [60]), der maßgeblich den Decompiler nutzt.

Programmcode ist in Squeak/Smalltalk dabei entweder

1. in einer Methode gekapselt und wird über diese aufgerufen, oder
2. wird im `Workspace` mittels eines `doIt` ausgeführt.

Dabei liegt bei einer Methode der Programmcode vor, während dieser bei einem `doIt` aktiv aus der vorliegenden `CompiledMethod` rekonstruiert werden muss.

Debugging mit Methoden Es müssen keine Anpassungen vorgenommen werden, dass visuelle Elemente als beliebiges Syntaxelement im Debugger vorliegen und die bekannte Behandlung (*stepInto*, *stepOver* etc.) erhalten bleibt. In diesem Fall kann der Debugger seine volle Funktionalität beibehalten.

Debugging ohne Methoden Bei einem `doIt` muss zum einen während der Erstellung der `VisualElementNode` das Dekompilieren vorgesehen werden (u. a. korrektes Weitersetzen des Intstruktionszeigers). Zum anderen muss die Behandlung im Decompiler, falls dieser ein visuelles Element liest, angepasst werden. Dies lässt sich für *Primarys* und *Methods* relativ einfach umsetzen, sodass auch für sie in diesem Fall das Debugging weiterhin funktioniert.

Anders verhält es sich bei visuellen Elementen als *Message Expresssion*. Aufgrund der Inkompatibilität von Parametern visueller Elemente und der eigentlichen Smalltalk-Syntax wurde beim Kompilieren eine Transformation mittels Lese-Makros durchgeführt. Im Decompiler fehlt jedoch noch eine passende Rekonstruktion damit das Debugging wie gewohnt funktioniert.

4.5.3 Bekannte Limitierungen

Die Erweiterung der Syntax und die Anpassungen des Compilers funktionieren und die **LIB**-Prinzipien werden erfüllt. Trotzdem zeigt die aktuelle Umsetzung noch Limitierungen, die in diesem Abschnitt betrachtet werden.

Bis dato nicht umgesetzte visuelle Elemente Abschnitt 4.2.1 nennt noch weitere Syntaxelemente, die man mit visuellen Elementen umsetzen könnte. Einige spezielle Einsatzmöglichkeiten visueller Elemente sind daher bisher noch nicht möglich.

***Message Expression* als visuelles Element** In Abschnitt 4.5.1 und Abschnitt 4.5.2 haben sich Performancelimitierungen und Schwierigkeiten beim Debuggen visueller Elemente als *Message Expression* gezeigt. In diesem Punkt muss die gewählte Implementierung noch angepasst und verbessert werden, insbesondere die Lösung über das `doesNotUnderstand:` sollte für zukünftige Versionen überdacht und ggf. doch durch eine Behandlung in der VM ersetzt werden.

Integration in das Gesamtsystem In der aktuellen Implementierung existieren tiefe Eingriffe in verschiedene Klassen des Gesamtsystems. Dies erschwert eine Integration in die Hauptversion von Squeak/Smalltalk. Damit müssen die Anpassungen an bestehenden Klassen und Strukturen parallel zur Hauptversion gepflegt werden, womit sie gegenüber zukünftigen Eingriffen in die zugrundeliegende Infrastruktur nicht sonderlich robust sind.

4.6 Verwandte Arbeiten

Der gewählte Ansatz beginnt eine Erweiterung von Squeak/Smalltalk zu einer HVPL. Dafür musste an verschiedenen Stellen der Compiler angepasst werden, wobei die Manipulation der AST-Knoten eine zentrale Rolle spielte.

4.6.1 Heterogene visuelle Programmiersprachen

Die erste Idee von HVPLs wurde 1995 durch Erwig & Meyer [36] beschrieben. Sie ermöglicht die Nutzung konkreter visueller Elemente in textuellem Programmcode. Die Autoren trennen in ihrem entwickelten Framework strikt zwischen visuellen und textuellen Konstrukten. Dies zeigt sich insbesondere in dem gezeigten Compiler, welcher der Phase für Text eine Phase für visuelle Elemente vorstellt.

Der in diesem Kapitel gewählte Ansatz ist die logische Fortführung. Durch visuelle Elemente als integraler Bestandteil der Syntax und der damit verbundenen Verarbeitung im bestehenden Compiler, benötigen sie keine eigenen Grammatiken, die zudem explizit gewartet werden müssen. Der Wegfall der Übersetzungsphase zu Programmtext vermeidet dabei einen fehleranfälligen Punkt bei der Erstellung neuer visueller Elemente.

4.6.2 Anpassungen des Smalltalk-Compilers

In einer Analyse von Smalltalk als reflektierende Programmiersprache zeigen die Autoren Variationspunkte für die Erweiterung des Smalltalk-Compilers auf [93, S. 8 ff.]. Hintergrund ist die Erweiterung der *Message Send* Semantik.

In diesem Kapitel werden alle beschriebenen Variationspunkte des Compilers genutzt, zusätzlich werden für die Integration weitere Syntax mit visueller Repräsentation nur Änderungen am Scanner notwendig. Darauffolgend können die bereits identifizierten Variationspunkte für textuelle Erweiterungen genutzt werden.

4.6.3 Live-Elemente im AST

Auch in Moonchild [35] werden live veränderbare visuelle Elemente in textuellen Programmcode integriert. Hier ist die Manipulation des AST von Interesse. Informationen zur Darstellung visueller Elemente werden einem AST-Knoten als Metainformation angehängt.

Der in diesem Kapitel gewählte Ansatz verändert jedoch den AST statt diesen nur um Darstellungsdaten zu erweitern. So hat der Zustand eines visuellen Elements Einfluss auf die Ausführung und nicht seine textuelle Repräsentation. Dafür wird das gesamte visuelle Element in den AST-Knoten übernommen.

4.7 Zusammenfassung und Ausblick

Die vorgenommenen Anpassungen am Squeak/Smalltalk-Compiler implementieren erfolgreich die notwendigen Erweiterungen der Smalltalk-Syntax für visuelle Elemente. Sowohl Objekte als auch Nachrichten, also die Kernelemente objektorientierter Programmierung, lassen sich durch visuelle Elemente ersetzen und diese kompilieren. Auch Wege zur Abstraktion und Kombination wurden aufgezeigt und implementiert. Besonders erfreulich sind die teils deutlichen Performancegewinne bei der Ausführung visueller Elemente.

Einige Veränderungen, insbesondere die Umsetzung visueller Elemente als *Message Expression*, benötigten weitere zukünftige Arbeit. Außerdem lassen sich in Zukunft noch weitere Syntaxelemente visuell unterstützen, hierfür steht jeweils eine konzeptionelle Einordnung sowie die Umsetzung aus. Portierungen in andere Live-Programmiersysteme können ebenfalls noch erfolgen.

Die Erweiterung eines textorientierten Compilers für die Verarbeitung visueller Elemente in einem Live-Programmiersystem ist ein entscheidender Schritt. Sie ermöglicht die Nutzung interaktiver, visueller Live-Objekte in textuellem Programmcode und ermöglicht damit dem Entwickler Text und Grafik integriert und frei austauschbar zu nutzen.

4.7.1 Weitere visuelle Elemente

Wir sehen mehrere Möglichkeiten, die bekannten Limitierungen zu verringern und die Nützlichkeit von SandBlocks zu verbessern.

Neben visuellen Elementen, die beliebige Domänen unterstützen könnten, sehen wir in der Sprache Smalltalk zwei besonders interessante Angriffspunkte:

- ***Comment* als visuelles Element** Potentiell interessant ist die Nutzung visueller Elemente als Kommentare. Hierbei ergibt sich die Möglichkeit, diese ggf. automatisiert aus Programmcode zu erzeugen. Auch prototypenbasierte Systeme könnten davon profitieren.
- ***Message Send* als visuelles Element** Ebenso noch möglich ist die Ersetzung impliziter Syntax, in diesem Fall des Leerzeichens, welches in Smalltalk den *Message Send* repräsentiert. Die Darstellung als Leerzeichen stellt in Smalltalk eine Besonderheit dar, aus anderen Programmiersprachen kennt man diesen als Punkt zwischen Empfänger und Nachricht.

Dabei bringt der *Message Send* das größere Potential mit sich. Anstatt nur entweder den Empfänger der Nachricht oder die Nachricht selbst auszutauschen, wird der Versand als solcher verändert. Es wird also nicht angepasst *Was* an einen Empfänger gesendet wird, sondern *Wie* dies passiert. Würde man z.B. den *Message Send* der Nachricht `new` an den Empfänger `StarMorph` austauschen, könnte sich Listing 4.14 ergeben.

Listing 4.14: Beispielmethode, in welcher eine *Message Send* durch ein visuelles Element ersetzt wurde

```
1 Example >> #redStar
2    ^ StarMorph new "visuelles Element statt Leerzeichen"
3       color: Color red
```

Die sich dabei ergebenden Änderungen würden jedoch tiefe Eingriffe in der VM erfordern, da es zwar die Klasse `MessageSend` im *Image* gibt, das eigentliche Senden der Nachricht jedoch von der VM übernommen wird. Hinzu kommt, dass die aktuelle Betrachtung auf Tokens nicht mehr ausreichend wäre, man also endgültig statt der Syntaxbeschreibung aus [48, Anhang], die aus [90, Grammatik] verwenden müsste.

4.7.2 Bindings für visuelle Elemente

Weitere Limitierung aus Abschnitt 4.5.3 ist die schwierige Wartbarkeit durch Eingriffe in bestehende Infrastruktur. Daher wird eine weitere Möglichkeit skizziert visuelle Elemente nativ auszuführen, die zudem eine leichte Portierung auf andere Systeme wie z. B. Pharo[15] ermöglicht.

Dafür kann für jedes visuelle Element ein Binding, bestehend aus einem eindeutigen Bezeichner und dem visuellen Element, erzeugt und in einem globalen `Dictionary` gespeichert werden. Dann ersetzt man im Programmcode das visuelle Element durch das Binding, durchläuft damit den Compiler und löst das Binding zur Ausführung auf, wie es bei allen anderen Bezeichnern, z. B. Variablen, der Fall ist. Damit entsteht eine Lösung, die unabhängig von einem konkreten Compiler ist, es muss ausschließlich ein Vorschritt hinzugefügt und die Ausführung minimal angepasst werden. Ein Compiler sowie das Konzept von Bindings liegt in den meisten Systemen ohnehin vor.

Somit würde ein wesentlich geringerer Eingriff in das Gesamtsystem entstehen, der konkrete Compiler kann unverändert bleiben.

Dabei ist zu jedoch beachten, dass man einige architektonische Unschönheiten vermeidet. So benötigt die Auflösung der Bindings visueller Elemente globalen Zustand (i. d. R. ein `Dictionary`). Außerdem öffnet man den gesamten Problemraum eindeutiger Bezeichner und deren Synchronisation (z. B. über verschiedene Images) und durchbricht ggf. grundlegende Modularitätsprinzipien objektorientierten Designs (s. z. B. SPOT-Regel [89, S. 91]).

4.7.3 Visuelle Elemente *„all the way down“*

Auch wenn das SandBlocks-Projekt explizit eine gleichzeitige integrierte Nutzung visueller und textueller Elemente vorsieht, ist es interessant visuelle Elemente *all the way down* zu betrachten. In erster Form würde dies eine Ersetzung aller Literale im Image durch visuelle Elemente umfassen, in extremster Form die Ersetzung

[15] Das Projekt Pharo/Smalltalk: `https://pharo.org/`, letzter Zugriff am 12. Juli 2019.

aller Methoden durch Instanzen von `VisualElementMethod`. Offen ist, ob wirklich alle textuellen Repräsentationen ersetzt werden können.

Eine solche Ersetzung würde zunächst sehr deutlich zeigen, wie sich die Performance verändert, wenn die Nutzung visueller Elemente über eine vereinzelte Verwendung in Methoden hinausgeht. Damit einhergehend wäre auch die Stabilität visueller Elemente weiter zu untersuchen, insbesondere bei der Ersetzung kritischer Infrastruktur.

Interessant zu beobachten wäre außerdem, ob sich bei Entwicklern eine andere Interaktion mit einem nun rein visuellen System ergeben würde. Somit könnte dies auch in einer ersten größeren Nutzerstudie resultieren.

A Syntax von Smalltalk-80

In den Betrachtungen in Abschnitt 4.2.1 wird auf die originale Smalltalk-80-Syntax aus [48, Anhang] verwiesen. Diese ist für das Verständnis der Erweiterungen aus Abschnitt 4.3.1 hilfreich. Auf den folgenden Seiten sind diese Syntaxdiagramme als zur vereinfachten Referenz aus dem Original übernommen.

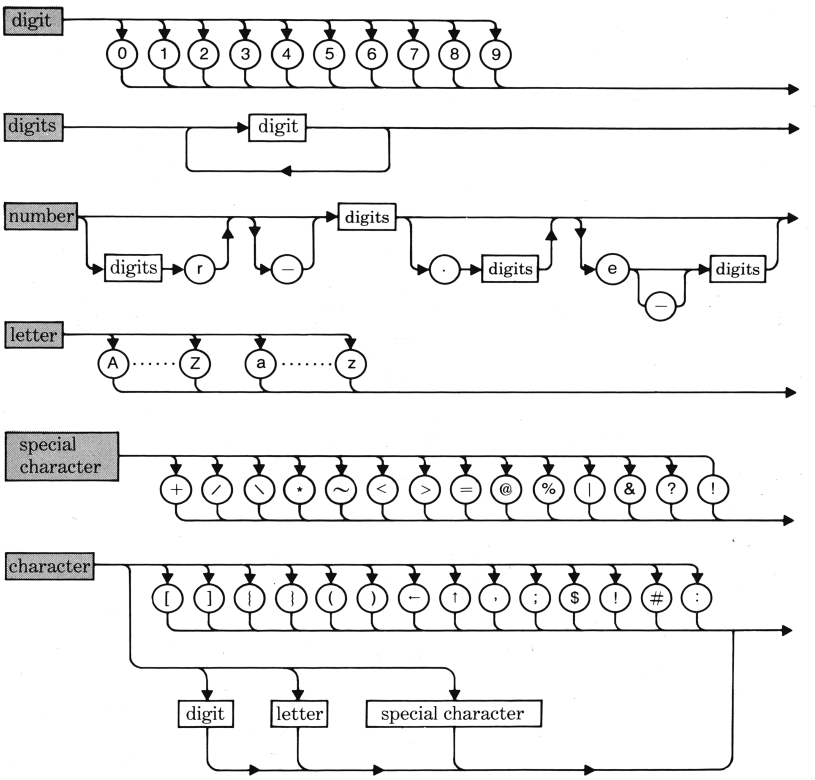


Abbildung 4.19: Syntax von Smalltalk-80 [48]

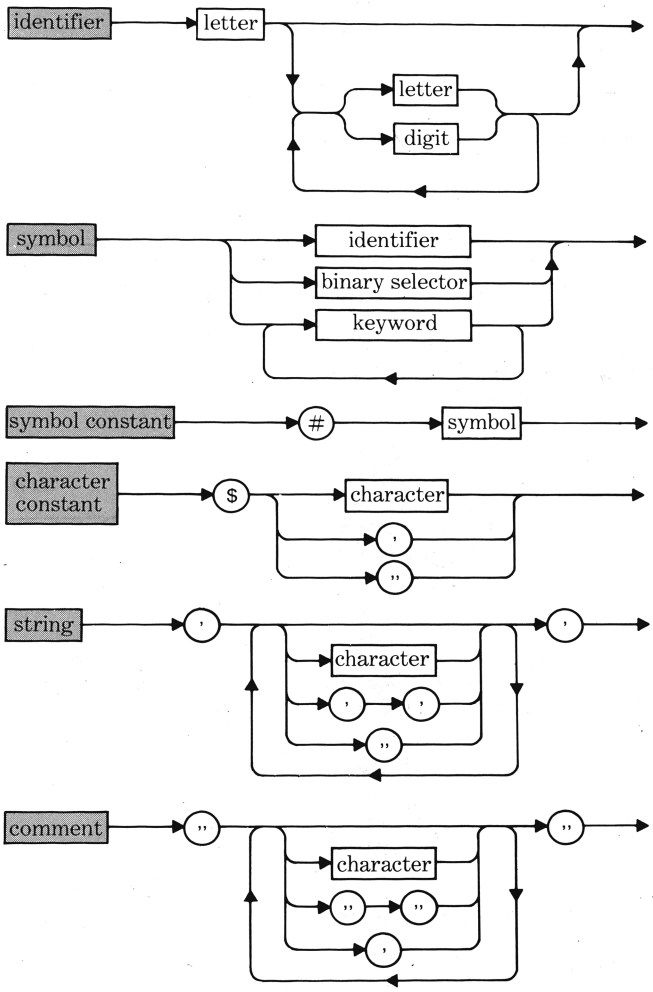


Abbildung 4.20: Fortsetzung der Syntax von Smalltalk-80 [48]

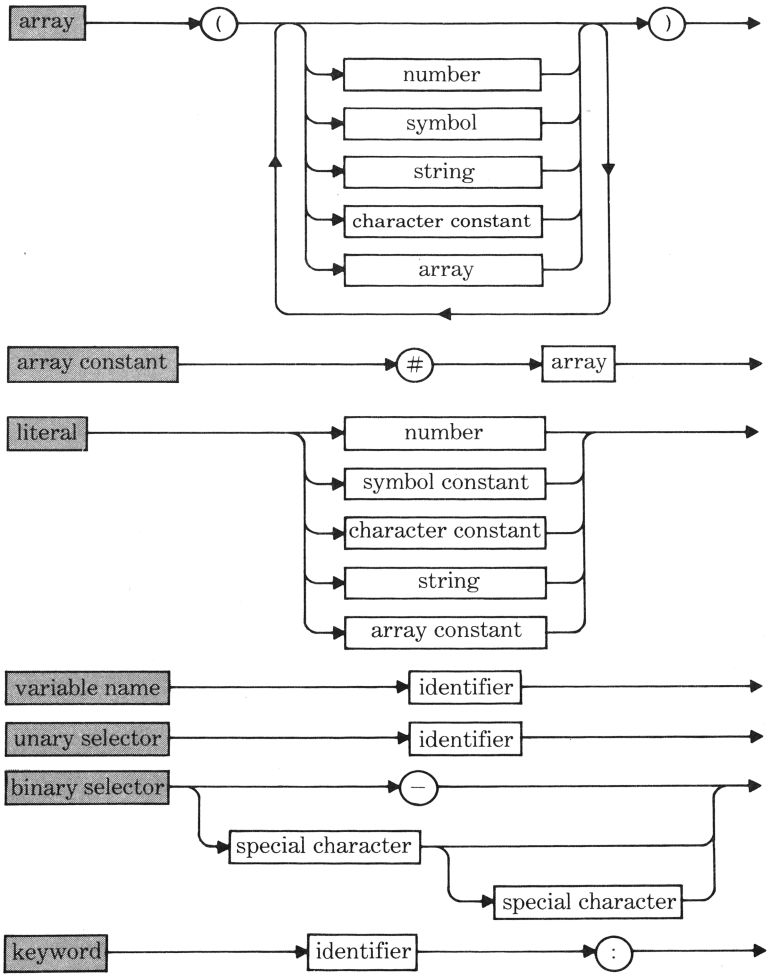


Abbildung 4.21: Fortsetzung der Syntax von Smalltalk-80 [48]

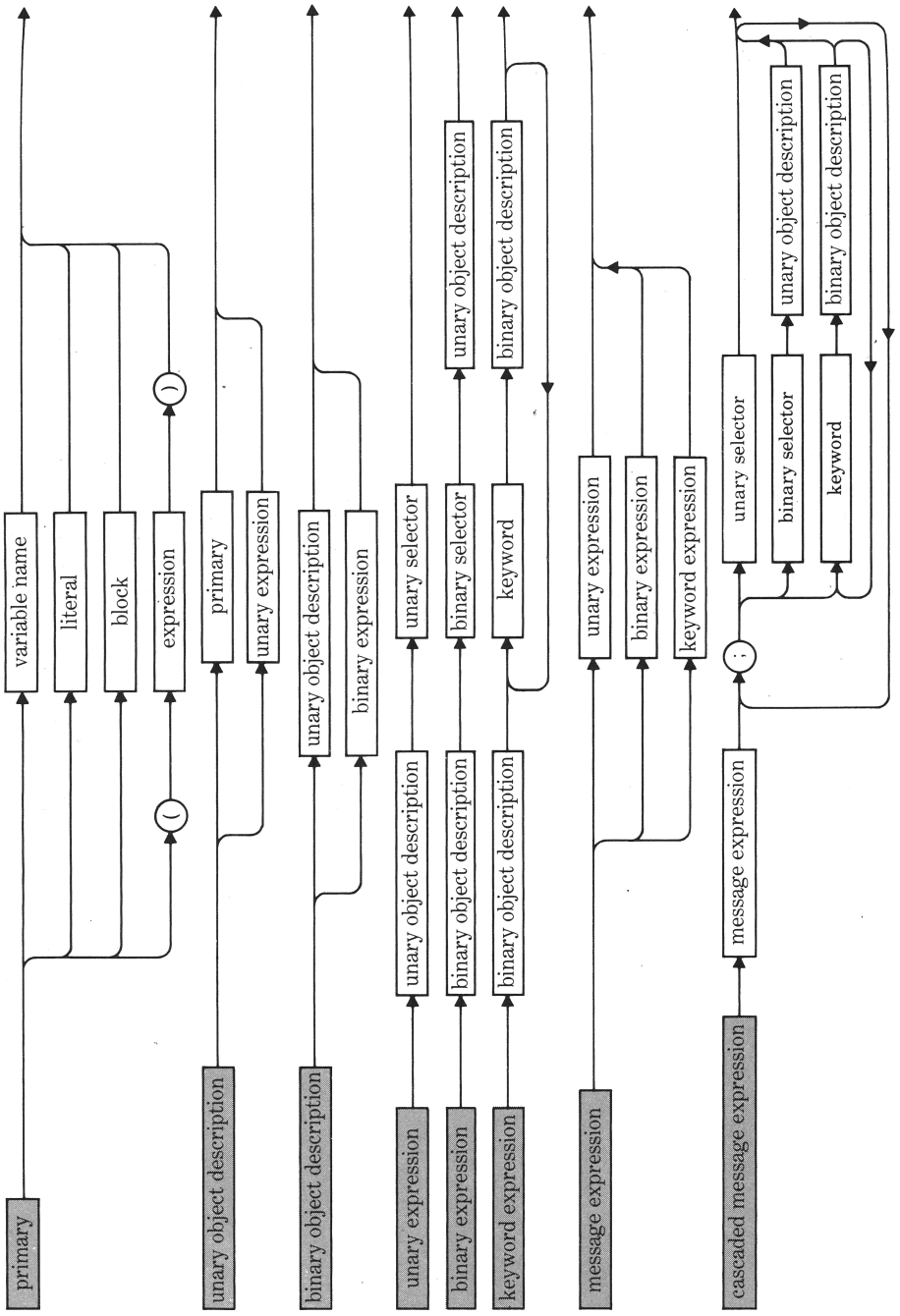


Abbildung 4.22: Fortsetzung der Syntax von Smalltalk-80 [48]

B Quelltext für Compiler-Erweiterung

In diesem Abschnitt finden sich primär die verschiedenen wichtigsten Erweiterungen rund um den Compiler, die für die Verarbeitung visueller Elemente notwendig sind. Sie erweitern entweder Methoden aus *Squeak5.2* oder sind damit kompatibel.

In Abschnitt 4.4.2 wird eine allgemeine Erweiterung des Scanners für visuelle Elemente gegeben. Dafür werden insbesondere die Methoden zur Erkennung und zum Auslesen visueller Elemente benötigt.

Listing 4.15: Allgemeine Erweiterungen des Scanners

```
Scanner >> #initializeTypeTable
   | newTable |
   "default"
   newTable := Array new: 256 withAll: #xBinary.
   "tab lf ff cr space"
   newTable atAll: #(9 10 12 13 32 16rA0 ) put: #xDelimiter.
   newTable atAll: ($0 asciiValue to: $9 asciiValue) put: #xDigit.

   1 to: 255
      do: [:index |
         (Character value: index) isLetter
            ifTrue: [newTable at: index put: #xLetter]].

   newTable at: VisualElementAttribute
      blockCharacter asciiValue put: #xVisualElement.
   newTable at: $$ asciiValue put: #xVESeparator.
   newTable at: $" asciiValue put: #xDoubleQuote.
   newTable at: $# asciiValue put: #xLitQuote.
   newTable at: $$ asciiValue put: #xDollar.
   newTable at: $' asciiValue put: #xSingleQuote.
   newTable at: $: asciiValue put: #xColon.
   newTable at: $( asciiValue put: #leftParenthesis.
   newTable at: $) asciiValue put: #rightParenthesis.
   newTable at: $. asciiValue put: #period.
   newTable at: $; asciiValue put: #semicolon.
   newTable at: $[ asciiValue put: #leftBracket.
   newTable at: $] asciiValue put: #rightBracket.
   newTable at: ${ asciiValue put: #leftBrace.
   newTable at: $} asciiValue put: #rightBrace.
   newTable at: $^ asciiValue put: #upArrow.
   newTable at: $_ asciiValue put: #xUnderscore.
   newTable at: $| asciiValue put: #verticalBar.
   TypeTable := newTable.
```

```
Scanner >> #xVisualElement
   token := self readVisualElement.
   tokenType := #visualElement.
   token ifNotNil: [token scannedWith: self].
   self step.
```

```
Scanner >> #readVisualElement
   | startOfToken initialPosition visualElement |
   startOfToken := mark.
   initialPosition := source instVarNamed: 'initialPositionOrNil'.
   (initialPosition isNil or: [initialPosition = 1])
      ifFalse: [startOfToken := startOfToken - initialPosition + 1].
   visualElementAttribute :=
      (source contents asText attributesAt: startOfToken)
         detect: [:each | each visualElementModel notNil]
         ifNone: [^ nil "Signal a Parser error"].

    ^ visualElementAttribute visualElementModel
```

In Abschnitt 4.4.2.1 wird eine Erweiterung des Parsers für visuelle Elemente als *Primarys* benötigt.

Listing 4.16: Erweiterung des Parsers für *Primarys*

```
Parser >> #primaryExpression
   "Handle keywords, brackets, braces and parenthesis here"

   (hereType == #string
      or: [hereType == #number
         or: [hereType == #literal
            or: [hereType == #character]]])
      ifTrue:
         [parseNode := encoder encodeLiteral: self advance.
         ^true].
   hereType == #visualElement
      ifTrue:
         [parseNode := self encodeVisualElement.
         ^true].

   "Handle numbers here"

   ^false
```

```
Parser >> #encodeVisualElement
   | visualElementModel visualElementNode |
   here ifNil: [^ self expected: 'VisualElementModel'].
   visualElementModel := self advance.
   visualElementModel compiledIn: cue getClass.
   visualElementNode := encoder
      encodeVisualElement: visualElementModel.
   visualElementNode generateSelectorWith: encoder.
   encoder
      noteSourceRange:
         (prevMark + requestorOffset to: self endOfLastToken)
      forNode: visualElementNode.
   ^ visualElementNode
```

In Abschnitt 4.4.2.1 wird außerdem eine weitere `ParseNode`, eine `VisualElementNode`, für visuelle Elemente eingeführt. Sie wird vom Encoder wie folgt erzeugt:

Listing 4.17: Erstellung einer `VisualElementNode`

```
Encoder >> #encodeVisualElement: aVisualElement
   ^ self
      name: aVisualElement
      key: aVisualElement
      class: VisualElementNode
      type: LdLitType
      set: litSet
```

Ihre wichtigsten Methoden zur Generierung von Bytecode finden sich im Folgenden.

Listing 4.18: Bytecodegenerierung der `VisualElementNode`

```
VisualElementNode >> #emitCodeForValue: stack encoder: encoder
   self pushBlockModel: stack encoder: encoder.
   self pushThisContext: stack encoder: encoder.
   pc := encoder nextPC.
   self pushSelector: stack encoder: encoder.
```

```
VisualElementNode >> #pushBlockModel: stack encoder: encoder
   stack push: 1.
   encoder genPushLiteral: index.
```

```
VisualElementNode >> #pushThisContext: stack encoder: encoder
   stack push: 1.
   encoder genPushThisContext.
```

```
VisualElementNode >> #pushSelector: stack encoder: encoder
   selectorNode
      emitCode: stack
      args: 1 "thisContext"
      encoder: encoder.
```

Die folgende Methode stellt das Aufnehmen visueller Elemente in den Bytecode sicher.

Listing 4.19: Sicheres Aufnehmen der `VisualElementNode` in Bytecode

```
VisualElementNode >> #emitCodeForEffect: stack encoder: encoder
   self emitCodeForValue: stack encoder: encoder.
   encoder genPop.
   stack pop: 1.
```

In Abschnitt 4.4.2.2 wird die Anpassung am Parser für die Verarbeitung visueller Elemente als *Message Expression*, sowie die Lese-Makros und die Auflösung über das `doesNotUnderstand:`, beschrieben.

Listing 4.20: Streammanipulation durch visuelle Elemente

```
VisualElementMessage >> #scannedWith: aScanner
   | oldSource newSource position |
   self isUnary ifTrue: [^self].
   oldSource := aScanner source.
   position := oldSource position.

   "If the stream is at the end, we called next
   one time less and appended a DoItCharacter manually."
   oldSource atEnd ifTrue: [position := position + 1].

   newSource := ReadStream on:
      ((oldSource collection copyFrom: 1 to: position - 1),
      self selectorWithArgs,
      (oldSource collection copyFrom: position to: oldSource size)).
   newSource position: position.

   "We already overstepped the aheadChar, so set it back."
   aScanner aheadChar: VisualElementMessage vESeparatorCharacter.

   aScanner source: newSource.
```

Listing 4.21: Parsen des manipulierten Streams

```
Parser >> #messagePart: level repeat: repeat
   | start receiver selector args precedence words keywordStart |
   [receiver := parseNode.

   "Complex handling of keywords here"

   hereType == #block
      ifTrue: [here ifNil: [^self expected: 'VisualElementModel'].
            here isUnary
               ifTrue:
                  [start := self startOfNextToken.
                  selector := self advance.
                  args := #().
                  precedence := 1]
               ifFalse:
                  [start := self startOfNextToken.
                  selector := self advance.
                  args := OrderedCollection new.
                  precedence := 1.
                  [hereType == #vESeparator]
                     whileTrue: [self advance.
                              self primaryExpression
                                 ifFalse: [^self expected: 'Argument'].
                              self messagePart: 3 repeat: true.
                              args addLast: parseNode]]]
      ifFalse: [^ args notNil]].

```

```
   parseNode := MessageNode new
            receiver: receiver
            selector: selector
            arguments: args
            precedence: precedence
            from: encoder
            sourceRange: (start to: self endOfLastToken).
   repeat]
      whileTrue: [].
   ^true
```

Listing 4.22: Auflösung der visuellen Nachricht

```
Object >> #doesNotUnderstand: aMessage
   | exception resumeValue |
   aMessage selector isBTRBlockModel
      ifTrue: [
         aMessage selector aboutToBeCalled.
         ^ aMessage selector calledWith:
            (self receiveVisualElementMessage: aMessage)].

   (exception := MessageNotUnderstood new)
      message: aMessage;
      receiver: self.
   resumeValue := exception signal.
   ^exception reachedDefaultHandler
      ifTrue: [aMessage sentTo: self]
      ifFalse: [resumeValue]
```

In Abschnitt 4.4.2.3 wird eine `VisualElementMethod` in eine `CompiledMethod` konvertiert.

Listing 4.23: Konvertierung einer `VisualElementMethod`

```
VisualElementMethod class >>
#sourceOn: aVisualElementModel signature: aString
   ^ aString asText, '
   <', VisualElementMethod pragma, '>
   ^ ', (VisualElementAttribute asTextFrom: aVisualElementModel)
```

```
VisualElementMethod class >> #pragma
   ^ #VisualElementModel
```

Weiterhin wird in Abschnitt 4.4.2.3 die Erstellung einer `VisualElementMethodNode` beschrieben, sofern diese das Pragma `VisualElementModel` enthält.

Listing 4.24: Erstellung einer `VisualElementMethodNode`

```
Encoder >> #methodNodeClass
   ^ (requestor properties
      at: VisualElementMethod pragma
      ifAbsent: [nil])
         ifNil: [MethodNode]
         ifNotNil: [VisualElementMethodNode]
```

In Abschnitt 4.3.1.1 wird auf die Beispielmethode als äquivalenter Smalltalk-String verwiesen. Sie wurde mittels der Methode `Object >> #storeString` erzeugt.

Listing 4.25: Methode mit visuellem Element serialisiert

```
Example >> #redStar
   ^ (StarMorph basicNew
         instVarNamed: #bounds put: (-15@ -15 corner: 14@14);
         instVarNamed: #owner put: nil;
         instVarNamed: #submorphs put: Array empty;
         instVarNamed: #fullBounds put: nil;
         instVarNamed: #color put: (Color r: 0.8 g: 1 b: 1);
         instVarNamed: #extension put:
            (MorphExtension basicNew instVarNamed: #locked put: false;
         instVarNamed: #visible put: true;
         instVarNamed: #sticky put: false;
         instVarNamed: #balloonText put: nil;
         instVarNamed: #balloonTextSelector put: nil;
         instVarNamed: #externalName put: nil;
         instVarNamed: #isPartsDonor put: false;
         instVarNamed: #actorState put: nil;
         instVarNamed: #player put: nil;
         instVarNamed: #eventHandler put: nil;
         instVarNamed: #otherProperties put: ((IdentityDictionary new)
            add: (#borderStyle ->
               (SimpleBorder basicNew
                  instVarNamed: #baseColor put: (Color r: 0.0 g: 0.0 b: 0.0);
                  instVarNamed: #color put: (Color r: 0.0 g: 0.0 b: 0.0);
                  instVarNamed: #width put: 1;
         yourself)); yourself);
         yourself);
         instVarNamed: #borderWidth put: 1;
         instVarNamed: #borderColor put: (Color r: 0.0 g: 0.0 b: 0.0);
         instVarNamed: #vertices put:
            ((Array new: 10)
               at: 1 put: (10.000000000000012@9.999999999999998);
               at: 2 put: (0.9217989253436502@5.820009361114252);
               at: 3 put: (-6.420395219202068@12.600735106701006);
               at: 4 put: (-5.2503062944587 56@2.675164674667523);
               at: 5 put: (-13.968022466674205@ -2.2123174208247502);
               at: 6 put: (-4.166666666666665@ -4.166666666666669);
               at: 7 put: (-2.2123174208247374@ -13.968022466674206);
               at: 8 put: (2.6751646746675277@ -5.2503062944587535);
               at: 9 put: (12.600735106701013@ -6.420395219202057);
               at: 10 put: (5.820009361114252@0.9217989253436453);
               yourself);
         instVarNamed: #closed put: true;
         instVarNamed: #filledForm put: nil;
         instVarNamed: #arrows put: #none;
         instVarNamed: #arrowForms put: #();
         instVarNamed: #smoothCurve put: false;
         instVarNamed: #curveState put: nil;
         instVarNamed: #borderDashSpec put: nil;
         instVarNamed: #handles put: nil;
         instVarNamed: #borderForm put: nil;
         yourself)
      color: Color red
```

C Quelltext für Benchmarks

In diesem Abschnitt finden sich die in Abschnitt 4.5.1.1 zum Benchmarking verwendeten Methoden. Listing 4.26 stellt die einfachste Möglichkeit zur Erstellung eines Morphs dar.

Listing 4.26: Vollständig textuelle Methode zur Erstellung eines Morphs

```
Benchmarking >> #newMorph
   ^ Morph new
```

Auch Listing 4.27 ist eine ausschließlich textuelle Repräsentation. Statt `Morph new` wird der äquivalente String verwendet, der ihn wiederherstellen kann. Somit ist der Text, der vom Compiler verarbeitet werden muss, deutlich länger. Diese Variante ist partiell realistischer, da für ein komplexes, konfiguriertes visuelles Element oft keine kurze textuelle Darstellung vorliegt.

Listing 4.27: Vollständig textuelle Methode, die den `StoreString` nutzt

```
Benchmarking >> #newMorphAsLongText
   ^ (Morph basicNew instVarNamed: #bounds put: (0@0 corner: 50@40);
         instVarNamed: #owner put: nil;
         instVarNamed: #submorphs put: Array empty;
         instVarNamed: #fullBounds put: nil;
         instVarNamed: #color put: (Color r: 0.0 g: 0.0 b: 1);
         instVarNamed: #extension put: nil;
         yourself)'.
```

In Listing 4.28 wurde ein *Primary*, in Listing 4.29 eine *Message Epression* und in Listing 4.30 eine ganze *Method* durch ein visuelles Element ersetzt.

Listing 4.28: Visuelles Element als *Primary*

```
Benchmarking >> #newMorphAsPrimary
   ^ ■
```

Listing 4.29: Visuelles Element als *Message Expression*

```
Benchmarking >> #newMorphAsMessage
   ^ Morph ■
```

Listing 4.30: Visuelles Element als *Method*

```
Benchmarking >> #newMorphAsMethod
   ■
```

5 Objekte Serialisieren und Austauschen: Heterogener, visueller Programmcode in Squeak/Smalltalk

Heterogene, visuelle Programmiersprachen stellen ein Konzept vor, um visuelle Programmierelemente in eine existierende textuelle Programmiersprache einzuführen. In einem objektorientierten Programmiersystem wie Squeak/Smalltalk ist es vorteilhaft, wenn visuelle Elemente direkt auf Objekte der Sprache abbilden. Durch diese Umsetzung ist es nicht mehr möglich Programmcode, über die Systemgrenzen hinaus, zwischen Entwicklern auszutauschen.

Diesen Austausch wieder zu ermöglichen ist das Ziel dieses Kapitels. Dafür wird die bestehende Infrastruktur angepasst, um visuelle Elemente als Programmcode zu akzeptieren. Wenn visuelle Elemente über die Systemgrenze hinaus transportiert werden, müssen diese serialisiert werden. Die verschiedenen existierenden Serialisierungsformate werden miteinander verglichen und das geeignetste Verfahren gewählt. Nachdem der heterogene, visuelle Programmcode in ein anderes System übertragen wurde, werden die Systeme zum Vergleichen und Einpflegen der Änderungen angepasst, um diesen zu unterstützen. Diese Änderungen erlauben es Smalltalk-Programmierern heterogenen, visuellen Programmcode genau wie Smalltalk-80 Programmcode zu teilen, ohne einen neuen Arbeitsfluss zu lernen.

5.1 Einleitung

Moderne Softwareentwicklung bedeutet Teamarbeit. Ein aus vielen Browsern bekanntes Beispiel, ist die v8 virtuelle Machine, die JavaScript-Code ausführt. Die Software verzeichnet bereits 330 Mitwirkende[1].

Um Teamarbeit in der Softwareentwicklung zu ermöglichen existieren bereits mehrere Lösungen. Softwaresysteme wie Git, Subversion oder Mercurial können Programmcode unabhängig von der verwendeten Programmiersprache zwischen Entwicklern austauschen. Diese Systeme können mit beliebigen Dateien umgehen, bieten aber verbesserte Unterstützung für Textdateien. Textuelle Programmiersprachen wie C++, Java oder Javascript erfreuen sich dementsprechend hervorragender Unterstützung zum Teilen des Programmcodes.

Neben textuellen Programmiersprachen existieren auch visuelle Programmiersprachen. Rein visuelle Sprachen wie Scratch [74] oder Snap! [54] werden häufig im Bereich der Lehre eingesetzt. In der professionellen Anwendung sind diese allerdings weiterhin die Ausnahme. Um den Übergang zwischen visuellen und textuellen Programmiersprachen zu erleichtern stellen Erwig & Meyer in „*Heterogenous Visual Languages-integrating Visual and Textual Programming*“[36] eine Lösung

[1] V8 auf GitHub: `https://github.com/v8/v8`, letzter Zugriff am 17. Juli 2019.

vor, um visuelle und textuelle Programmiersprachen in ein gemeinsames System zu integrieren. Die technische Umsetzung dieser Integration übersetzt die visuellen Elemente in Programmtext, um diese ausführen zu können.

In objektorientierten Systemen wie Squeak/Smalltalk[2] wäre es allerdings möglich visuelle Elemente überhaupt nicht mehr auf Text abzubilden, sondern diese direkt als Objekte in den Programmcode einzufügen. Ein solches System und die daraus entstehenden Vor- und Nachteile beschreibt Kapitel 2 mit der allgemeinen Vorstellung und Einordnung des Projektes „*SandBlocks*“.

5.1.1 Das Projekt SandBlocks

Im Gegensatz zu den heterogenen visuellen Programmiersprachen die 1995 von Erwig & Meyer [36] vorgestellt wurden, zeichnet sich das Projekt SandBlocks dadurch aus, dass für visuelle Programmelemente keine textuelle Repräsentation existieren muss. Ein visuelles Element bezeichnet SandBlocks als einen Block. In diesem Kapitel werden Blöcke weiterhin als visuelle Elemente bezeichnet, um möglichen Verwechslungen mit dem `BlockClosure`[3] Syntaxelement in Squeak vorzubeugen. Die Bezeichnung Block wird weiterhin im aufgelisteten Programmcode verwendet.

Diese visuellen Elemente unterteilen sich in zwei konzeptionelle Rollen: Modell und Ansicht. Das Modell beinhaltet die für die Ausführung notwendigen Daten und Logik. Die Ansicht stellt das Element dar und kann es bearbeiten.

Grundlage des Systems bietet das Squeak-Programmiersystem, welches die Programmiersprache Smalltalk verwendet. Die visuellen Elemente in SandBlocks sind Smalltalk-Objekte. Da der Smalltalk-Code von Squeak direkt in Squeak bearbeitet werden kann, müssen diese nicht in Smalltalk-Programmtext übersetzt werden, um ausgeführt zu werden. Die Details der Einbindung der visuellen Elemente in die Ausführung sind in Kapitel 4 ausführlich beschrieben. Durch Einführung von Smalltalk-Objekten als Syntaxelemente der Sprache ist es allerdings nicht mehr möglich die bestehenden Teilungssysteme in Squeak zu verwenden. Diese erwarten Smalltalk-80 Programmcode, welcher vollständig durch einfache Zeichenketten beschrieben werden kann.

5.1.2 Forschungsfrage und Gliederung

Da keine textuelle Repräsentation existiert, ist es nicht mehr möglich diesen Programmcode mit den bestehenden Werkzeugen in Squeak/Smalltalk zu teilen. In diesem Kapitel wird der Ausdruck „*Teilen von Programmcode*“ als „*Austauschen von Programmcode mit dem Ziel der Zusammenarbeit*“ verstanden. Dieses Kapitel stellt vor, welche Schritte vorgenommen werden müssen, um das Squeak/Smalltalk-System zu erweitern damit die bestehenden Möglichkeiten zum Teilen von Programmcode mit visuellen Elementen genutzt werden können. Ziel ist es, dass der Arbeitsfluss

[2] Das Projekt Squeak/Smalltalk: `https://squeak.org/`, letzter Zugriff am 19. Juli 2019.

[3] Instanzen von `BlockClosure` werden umgangssprachlich auch Block genannt.

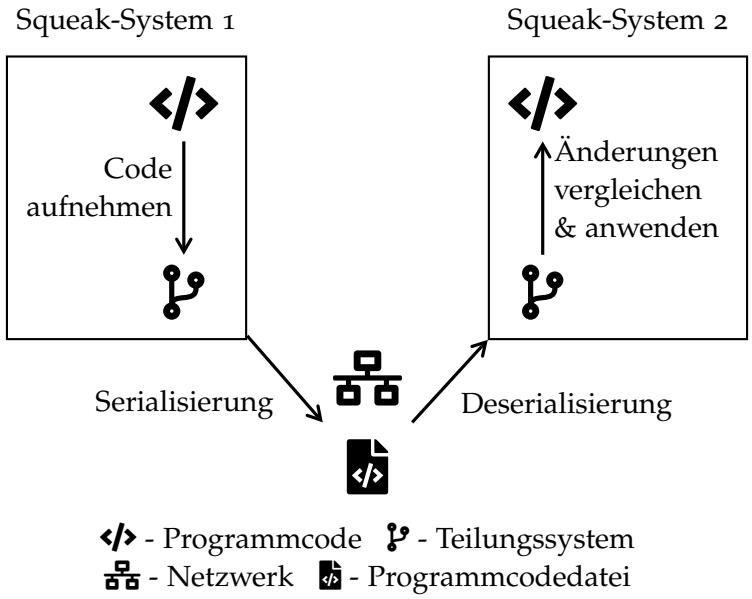


Abbildung 5.1: Ablauf des Teilens von Programmcodes zwischen zwei Squeak-Systemen. Hier ist zu beachten, dass die Teilungssysteme selbst in Squeak implementiert sind. Der Code verlässt die Systemgrenze erst nachdem er vom Teilungssystem verarbeitet wurde.

beim Austauschen visueller Elemente sich nicht vom Arbeitsfluss beim Austauschen von Programmtext unterscheidet. Dies unterstützt die Grundidee heterogener, visueller Programmsprachen, da diese das Ziel haben, den Übergang von der textuellen zur visuellen Programmierung zu erleichtern.

Abbildung 5.1 skizziert den Ablauf des Teilens von Programmcode in Squeak. Zuerst wird der Programmcode in die Infrastruktur zum Teilen von Code aufgenommen. Wie visuelle Elemente aufgenommen werden können, wird in Abschnitt 5.2 und Abschnitt 5.3 besprochen. Aufgenommener Programmcode verlässt die Squeak-Systemgrenze in serieller Form, normalerweise über das Dateisystem oder eine Internetverbindung. Beides setzt eine serielle Repräsentation des Codes vorraus, welche in Abschnitt 5.4 besprochen wird. Der Code kann dann von einem anderen Squeak-System eingelesen werden und auf das andere Squeak-System angewendet werden. Einige der Systeme erlauben es auch die Änderungen vorher einzusehen und nur bestimmte Änderungen zu übernehmen. Das Erkennen und die Anzeige von Änderungen wird in Abschnitt 5.5 angepasst, um visuelle Elemente zu unterstützen.

In Abschnitt 5.6 wird unser Ansatz evaluiert und diskutiert. Die verwandten Arbeiten, welche unseren Ansatz im Kontext der aktuellen Forschung einordnet, werden in Abschnitt 5.7 beschrieben. Zuletzt zieht Abschnitt 5.8 ein Fazit und gibt einen Ausblick auf mögliche nächste Schritte.

5.2 Teilungssysteme in Squeak

Squeak/Smalltalk bietet mehrere Werkzeuge an, um Programmcode zu teilen. Solche Werkzeuge werden im weiteren als Teilungssysteme bezeichnet. In diesem Abschnitt wird ein Überblick über die Teilungssysteme gegeben, um zu identifizieren welche Systeme existieren und dementsprechend angepasst werden müssen.

5.2.1 Smalltalk-Dateien

Zu Beginn wurden in Squeak zum Austauchen von Programmcode Smalltalk-Dateien (Dateienung: .st) verwendet. Diese wurden mit externen Lösungen, wie beispielsweise per E-Mail, zwischen Entwicklern ausgetauscht.

Smalltalk-Dateien beinhalten sowohl Klassen- als auch Methodendefinitionen in einer Datei. Diese bilden dabei immer nur einen Zustand der Definitionen ab. Wie in Listing 5.1 zu sehen können Smalltalk-Dateien Zusatzinformationen wie die Kategorie und den Zeitstempel der Methode speichern[4].

Listing 5.1: Eine beispielhafte Smalltalk-Datei die eine Klasse mit einer Instanzmethode und einer Klassenmethode beschreibt

```
'From Squeak5.2 of 22 January 2019 [latest update: #18231] on 14 June 2019 at
    ↪ 2:19:17 pm'!
Object subclass: #MyClass
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Playground'!

!MyClass methodsFor: 'my category' stamp: 'LM 6/14/2019 14:18'!
myInstanceMethod
   "This is an example method, it does nothing"
   ^ self! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

MyClass class
   instanceVariableNames: ''!

!MyClass class methodsFor: 'instance creation' stamp: 'LM 6/14/2019 14:19'!
myClassMethod

   ^ self new! !
```

[4]Smalltalk-Dateien werden beim einlesen stückweise vom Compiler evaluiert. Dadurch ist es prinzipiell möglich beliebigen Programmcode in eine Smalltalk-Datei zu schreiben, welcher dann ausgeführt wird. In diesem Kapitel wird nur auf die Nutzung von Smalltalk-Dateien zum Austauschen von Programmcode eingegangen.

5.2.2 Changesets

Das Teilen von Programmcode wurde später durch die Einführung von *Changesets* verbessert. Ein *Changeset* (Dateiendung: .cs) verwendet dasselbe Dateiformat wie eine Smalltalk-Datei und kann dieselben Informationen speichern.

Der Unterschied liegt in den bereitgestellten Werkzeugen in Squeak. Diese erlauben es, nur neue Änderungen, in ein Changeset aufzunehmen. Methoden, die nicht verändert wurden, werden auch nicht aufgenommen. Zusätzlich ist es möglich, verschiedene *Changesets* anzulegen, Definitionen zu einem *Changeset* hinzuzufügen, diese zu entfernen, Definitionen zwischen *Changesets* zu verschieben und *Changesets* zu benennen. *Changesets* sind ein einfaches Mittel, um Codeänderungen in Squeak zu teilen und werden auch auf den Squeak-Mailinglisten verwendet.

5.2.3 Monticello

Das Squeak Wiki beschreibt Monticello als ein „distributed, optimistic, concurrent, versioning system for Squeak and Pharo code“[5], also ein verteiltes, optimistisches, nebenläufiges Versionierungssystem für Squeak.

Versionen in Monticello bestehen immer aus dem gesamten Inhalt der gewünschten Pakete zum Zeitpunkt der Versionserstellung[6]. Zusätzlich speichert jede Version einen eindeutigen Versionsidentifikator und alle Versionsidentifikatoren der Vorgängerversionen. Dadurch ist es möglich gemeinsame Ursprünge verschiedener Versionen zu erkennen, den Unterschied zwischen verschiedenen Versionen zu ermitteln und diese zusammenzuführen.

Monticello ist unter Squeak-Nutzern weit verbreitet und wird zur Weiterentwicklung des Squeak-Systems selbst verwendet. Der sogenannte Squeak-*Trunk*[7] in dem sich die neuste Entwickler-Version von Squeak befindet ist ein Monticello-Repository. Genauso wie die *Inbox*[8], in die jeder Nutzer neue Änderungen als Vorschlag in Form einer neuen Monticello-Version hochladen kann.

5.2.4 FileTree/Cypress

Da in Monticello Versionen als Binärdateien erzeugt werden (Dateiendung: .mcz), sind diese nur in geringem Maße dazu geeignet mit externen Werkzeugen wie Git oder Subversion übertragen zu werden. Dieses Problem löst FileTree[9], indem es ein Monticello-*Repository* anbietet in dem die Monticello-Pakete im Cypress Austauschformat[10] in das Dateisystem geschrieben werden. FileTree verwendet

[5] Monticello: `https://wiki.squeak.org/squeak/1287`, letzter Zugriff am 14. Juni 2019.
[6] Monticellos Elemente: `https://wiki.squeak.org/squeak/3636`, letzter Zugriff am 6. Juni 2019.
[7] Squeak Trunk: `https://wiki.squeak.org/squeak/6538`, letzter Zugriff am 13. Juni 2019.
[8] Squeak Inbox: `https://wiki.squeak.org/squeak/6545`, letzter Zugriff am 13. Juni 2019.
[9] FileTree auf GitHub: `https://github.com/dalehenrich/filetree`, letzter Zugriff am 14. Juni 2019.
[10] Cypress auf GitHub: `https://github.com/CampSmalltalk/Cypress/wiki`, letzter Zugriff am 14. Juni 2019.

```
Playground.package
├─.filetree
├─properties.json
├─monticello.meta...Ordner der die für die Monticello-
│                   Version notwendigen Informationen spei-
│                   chert.
└─MyClass.class
  ├─README
  ├─methodProperties.json
  ├─properties.json
  ├─class
  │ └─myClassMethod.st
  └─instance
    └─myInstanceMethod.st
```

Abbildung 5.2: Aufbau des Pakets (engl. *package*) der Beispielklasse `MyClass` im Cypress-Format

die Monticello-Klassen um Versionen in Squeak zu repräsentieren und ist mit der Monticello-Infrastruktur in Squeak kompatibel.

Wie in Abbildung 5.2 zu sehen ist organisiert Cypress die Methoden einer Klasse in zwei Ordnern: *class* und *instance*. Im *class*-Ordner befinden sich alle Klassenmethoden und im *instance*-Ordner alle Instanzmethoden. Jede einzelne Methode ist wiederum in einer einzelnen .st-Datei gespeichert. Wie in Listing 5.2 zu sehen beinhalten diese .st-Dateien den Smalltalk-Code allerdings in einem anderen Format als die in Abschnitt 5.2.1 angesprochenen Smalltalk-Dateien.

Um Verwechslungen vorzubeugen, werden die von Cypress generierten .st-Dateien im Weiteren als *„Cypress-Dateien“* bezeichnet.

Listing 5.2: Die myInstanceMethod.st-Datei aus dem in Abbildung 5.2 dargestellten FileTree-*Repository*

```
1 my category
2 myInstanceMethod
3  "This is an example method, it does nothing"
4  ^ self
```

Die erste Zeile einer solchen Cypress-Datei gibt die Kategorie der Methode an. Der gesamte Rest der Datei ist der Programmtext in UTF-8-kodiert. Weitere Eigenschaften wie der Zeitstempel der Methode speichert Cypress in der Datei `methodProperties.json`, in der die Zusatzinformationen aller Methoden gesammelt sind.

Da Cypress den Programmcode als Reintext speichert, können mit FileTree angelegte *Repositories* mit bestehenden Werkzeugen wie Git, Subversion, etc. übertragen werden und profitieren von der verbesserten Unterstützung für Textdateien.

5.2.5 Squot

Der Name Squot steht für „Squeak's Object Tracker" [91]. Ziel des Squot-Projekts ist es Versionskontrolle beliebiger Smalltalk-Objekte zu ermöglichen.

Squot bietet Möglichkeiten, beliebige Objekte zu teilen, Versionen der Objekte anzulegen, Unterschiede zu ermitteln, Objektgraphen zusammenzuführen und spezifische Versionen des Objekts wieder in das System zu laden.

Squot legt dabei für jede Version eines Objekts ein *Snapshot*-Objekt an, welches eine Version des tatsächlichen Objekts repräsentiert. Diese *Snapshot*-Objekte werden dann serialisiert und zwischen Systemen geteilt. Squot erlaubt es jedem Objekt über die Methode `captureWithSquot` selbst festzulegen wie das *Snapshot*-Objekt erzeugt werden soll [91, Kapitel 4.1.3 *Capturing*]. Die *Snapshot*-Objekte legen fest, wie sie serialisiert, Unterschiede an ihnen ermittelt oder diese ins System geladen werden.

Als Snapshot für Programmcodeobjekte verwendet Squot Instanzen der Klasse `SquotPackageShadow`. Diese wiederum verwenden die von Monticello bereitgestellten Klassen um Programmcode zu versionieren. Zum Serialisieren der Objekte von `SquotPackageShadow` verwendet Squot das Cypress-Format.

5.3 Integration in Squeak's Teilungssysteme

In Abschnitt 5.2 wurden 5 Systeme zum Teilen von Programmcode in Squeak vorgestellt. Diese Verfahren teilen viele Klassen in Squeak, wodurch es möglich ist die vorgestellten Verfahren in 2 Gruppen aufzuteilen:

1. Smalltalk-Dateien-basiert
   - Smalltalk-Dateien
   - Changesets
2. Monticello-basiert
   - Monticello
   - FileTree
   - Squot

Um visuelle Programmierelemente in diese Versionierungssysteme aufzunehmen muss also nur eine Integration in die zwei zugrundeliegenden Systeme vorgenommen werden.

5.3.1 Text versus String

Zuerst ist es wichtig zu wissen, in welcher Form Programmcode im Squeak-System vorliegt. In vielen populären Programmiersprachen, wie beispielsweise C++, Java und JavaScript, liegt der Programmcode in Textdateien vor. Um den Programmtext zu bearbeiten wird ein externer Editor verwendet.

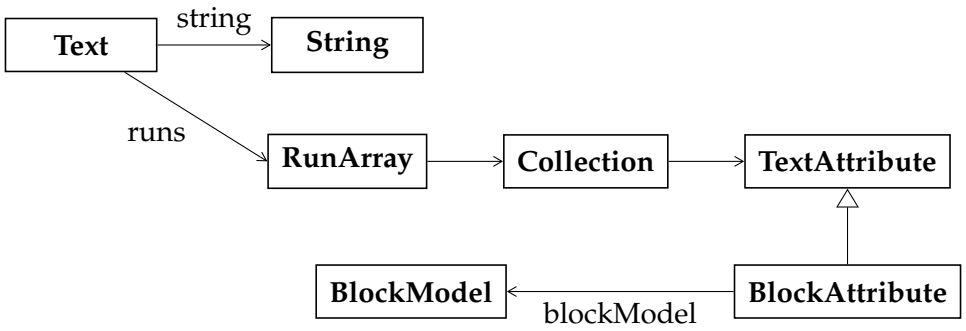


Abbildung 5.3: Die Verbindung der Klasse `Text` zum `BlockModel`

In Squeak wird die Bearbeitung des Programmcodes durch das Squeak-System selbst vorgenommen. Da Squeak ein objektorientiertes System ist, verwendet es Objekte, um den eigenen Programmcode zu repräsentieren.

Programmcode liegt in den meisten Fällen als reine Zeichenfolge vor. Eine solche Zeichenfolge bezeichnet Squeak als `String`. Für Smalltalk-Programmtext werden beinahe ausschließlich Instanzen der Klasse `ByteString` verwendet. Diese beinhalten für jeden Charakter ein Byte.

Um Zusatzinformationen zum reinen String speichern zu können, unterstützt Squeak die Klasse `Text`. Diese erweitert einen String um Textattribute, welche die Formatierung des Strings bestimmen. Wie in Abbildung 5.3 zu sehen werden in SandBlocks, visuelle Elemente in den Programmcode integriert, indem ein spezielles Textattribut eingefügt wird. Zusätzlich wird an diese Stelle im Text noch ein spezieller Charakter in den String eingefügt der signalisiert, dass an dieser Stelle im String ein Textattribut mit `BlockModel` zu finden sein muss. Bei dem Charakter handelt es sich um den sogenannten „Start of Heading"-Charakter (ASCII-Wert: 1, [25]). Im Weiteren wird dieser Charakter auch als SOH bzw. SOH-Charakter abgekürzt und als `SOH` visualisiert.

Wie in *„Smalltalk-80: The Language and its implementation"*[48, Kapitel *Compiler*] nachzulesen, werden in Smalltalk-80 Instanzen der Klasse `String` verwendet, um Programmcode zu repräsentieren.

> *„Source methods written by programmers are represented in the Smalltalk-80 system as instances of String."*

Instanzen der Klasse `Text` werden vor allem von Werkzeugen wie dem `Browser` verwendet, um Formatierung des Programmcodes zu ermöglichen. Da jetzt für den Programmcode notwendige Informationen im `Text` enthalten sind muss das bestehende System daran angepasst werden. Die notwendige Anpassung am Compiler werden in Kapitel 4 besprochen.

5.3.2 Integration in Smalltalk-Dateien-basierte Verfahren

In Squeak ist es möglich Programmcode eigene Formatierungen hinzuzufügen. Deshalb unterstützen die Smalltalk-Dateien-basierten Dateiformate bereits das Aufnehmen von Textattributen. Dabei kann jedes Textattribut selbst bestimmen wie es in die Smalltalk-Datei geschrieben bzw. gelesen wird.

Damit das `BlockAttribute` aufgenommen werden kann müssen drei Methoden implementiert werden:

1. `BlockAttribute class >> #scanCharacter`
2. `BlockAttribute >> #writeScanOn: aWriteStream`
3. `BlockAttribute class >> #scanFrom: aReadStream`

Die Methode `BlockAttribute class >> #scanCharacter` gibt einen Charakter aus, der beim Auslesen verwendet wird, um die zu verwendende Spezialisierung von `TextAttribute` zu ermitteln. Für `BlockAttribute` wird der SOH-Charakter als `scanCharacter` verwendet, da dieser noch von keinem anderen Textattribut verwendet wird und bereits als Signalcharakter im String benutzt wird.

In der Methode `writeScanOn:` wird das visuelle Programmierelement in serialisierter Form auf den `WriteStream` geschrieben. In der Methode `scanFrom:` wird das Objekt aus dem serialisierten String wiederhergestellt. Das Squeak-System speichert dann in der Smalltalk-Datei bzw. dem *Changeset* automatisch den gesamten Text und stellt diesen beim auslesen wieder her. Das verwendete Serialisierungsformat wird in Abschnitt 5.4 beschrieben.

Nachdem diese drei Methoden angepasst sind können nun sowohl Smalltalk-Dateien, als auch *Changesets*, wie gewohnt verwendet werden. Die visuellen Programmierelemente werden korrekt in den Dateien gespeichert und ausgelesen.

5.3.3 Integration in Monticello-basierte Verfahren

Da bisher in Smalltalk der `String` zur vollständigen Beschreibung des Programmcodes ausgereicht hat verwendet Monticello nur den `String` und nicht den `Text` um Programmcode zu teilen. Dies ist beispielsweise in Listing 5.3 zu sehen. Das Listing zeigt die Methode die Monticello, verwendet um den den Programmcode einer Methodenreferenz zu ermitteln. Hier ist klar zu sehen, dass der Programmcode zuerst zu einem `String` konvertiert wird und somit alle Zusatzinformationen des `Text`-Objektes verlorengehen. Da die visuellen Elemente aber nur als Teil des Textes gespeichert werden hat dies zur Folge, dass nun Teile des Programmcodes nicht mehr mitgespeichert werden.

Listing 5.3: Die Methode `source` der Klasse `MethodReference` konvertiert den Programmcode automatisch in einen `String`

```
1 MethodReference >> #source
2    ^ (self actualClass sourceCodeAt: methodSymbol) asString withSqueakLineEndings
```

`String` und `Text` bieten in Squeak zwei verschiedene, wenn auch ähnliche Schnittstellen an. Monticello erwartet dementsprechend, dass die Schnittstelle der Klasse `String` erfüllt ist.

Selbst wenn Monticello soweit angepasst wird, dass es beim Erstellen einer neuen Version mit FileTree keine Fehler erzeugt, speichert FileTree nur den Inhalt des Strings ab. Dementsprechend bleibt von einem visuellen Element wie in Listing 5.5 zu sehen, nur noch der SOH-Charakter übrig.

Listing 5.4: Eine beispielhaft visuell angereicherte Methode. Sie gibt die Farbe Grün zurück.

```
1 MyClass >> #foo
2    ^   #00FF00
```

Listing 5.5: Die Datei „foo.st“ die FileTree für die in Listing 5.4 beschriebene Methode anlegt

```
1 my category
2 foo
3    ^ SOH
```

Das bestehende Monticello-System schreibt also selbst wenn es einen Text erhält nur den String heraus, wobei die visuellen Programmierelemente verloren gehen. Das liegt unter anderem daran, dass das `Stream`-Interface zum Schreiben von Dateien `Text >> #at:` zum Zugriff verwendet. Diese Methode simuliert das Verhalten des Strings und gibt nur den Charakter, ohne die zugehörigen Textattribute, zurück. Das zugrundeliegende Problem ist aber, dass der Inhalt einer Datei nichts anderes als eine Abfolge von Bytes ist. Nach der Definition in Abschnitt 5.3.1 somit ein `ByteString`. Die Informationen, die in einem Text gespeichert sind, müssen also in jedem Fall in einen String übertragen werden.

In diesem Fall bietet es sich an, den Text direkt in Monticello zu einem passenden String zu konvertieren, da dann die bestehenden Systeme weiter mit dem String-Interface arbeiten können.

5.3.4 SandBlocks Textattribute speichern und lesen

Wie in Abschnitt 5.3.2 besprochen, gibt es bereits eine programmatische Schnittstelle, um beliebige Textattribute seriell zu speichern und zu lesen. Diese Schnittstelle wird verwendet, um eine Methode zu definieren, die einen beliebigen `Text` in einen `String` konvertiert, der die Informationen über die Textattribute beinhaltet.

Dazu werden 2 neue Methoden eingeführt:

1. `asBlockString` (Benennung angelehnt an `asString`)
2. `asBlockText` (Benennung angelehnt an `asText`)

Die Methode `asBlockString` konvertiert einen `Text` in einen `String` der die visuellen Elemente kodiert beinhaltet. Die Methode `asBlockText` konvertiert den `String` zurück in einen `Text`.

Es werden nur Textattribute in den `String` übernommen die an der Position eines SOH-Charakters sind. Dies ist ausreichend da das Textattribut eines visuellen Elements immer nur an der Stelle eines SOH vorkommen kann[11].

Das Verfahren übersetzt jeden SOH-Charakter im String durch einen SOH-Charakter, gefolgt von den serialisierten Textattributen an der Stelle des Charakters im Text. Um zu signalisieren, dass der normale Text weitergeht wird der „*Start of text*"-ASCII-Kontrollcharakter verwendet.

Der SOH-Charakter wurde als Signalcharakter gewählt, da er laut der in „*Smalltalk 80: The Language and its Implementation*" [48, Anhang] vorgestellten Grammatik kein valider Charakter in Smalltalk-Code ist. Somit kann er nicht mit bestehendem Programmcode interferieren. Auch wird der Charakter bereits in einem Text verwendet, um ein visuelles Element zu signalisieren. Der Charakter „*Start of text*" (Abkürzung: STX, Visualisierung: STX) wird auch verwendet, da er kein valider Charakter in der Smalltalk-Grammatik ist und laut RFC 20 verwendet werden kann um eine durch SOH gestartete Sequenz zu beenden [25].

> *„STX (Start of Text): A communication control character which precedes a sequence of characters that is to be treated as an entity and entirely transmitted through to the ultimate destination. Such a sequence is referred to as ‚text.' STX may be used to terminate a sequence of characters started by SOH."*

Eine beispielhafte Konvertierung der in Listing 5.4 vorgestellten Methode ist in Listing 5.6 zu sehen. Das hier zwei SOH-Charaktere nacheinander folgen ist dem geschuldet, dass dasselbe Verfahren zur Serialisierung der Textattribute gewählt wird das auch genutzt wird um die Textattribute in Smalltalk-Dateien zu serialisieren. Das zweite SOH ist der `scanCharacter` des Textattributs, der in Abschnitt 5.3.2 als SOH definiert wurde.

Listing 5.6: Der als blockString kodierte Text der in Listing 5.4 gezeigten Methode. „*Color green*" wird hier als Serialisierung des visuellen Elements verwendet.

```
1 MyClass >> #foo
2     ^ SOH SOH Color green STX
```

Die vollständigen Implementierungen der Methoden `asBlockString` und deren Variante für formatierten Text `asBlockText` sind im Abschnitt B unter Listing 5.15 und Listing 5.17 finden.

Unter Zuhilfenahme dieses Konvertierungsverfahrens kann Monticello nun erweitert werden, um visuelle Programmierelemente zu übertragen. Dazu wird die Klasse `MCMethodDefinition` angepasst. Die Konstruktoren wurden verändert, um den Text der Methode zu speichern, statt nur den String. Da das restliche System aber wie bereits besprochen das `String`-Protokoll erwartet, wurde die `source`-Methode der `MCMethodDefinition` angepasst (s. Listing 5.7). Die Methodendefinition

[11] Dieses Verfahren kann erweitert werden, um alle Textattribute in den String zu kodieren und über Monticello zu übertragen. Dieses Kapitel beschränkt sich auf die Übertragung visueller Elemente.

beinhaltet zwar den Text, gibt nach außen aber eine serialisierte Version als String aus. Das restliche System kann damit umgehen und schreibt die notwendigen Informationen heraus. Das gilt für FileTree, das Cypress-Format, und Squot.

Listing 5.7: Die veränderte `source`-Methode der `MCMethodDefinition` konvertiert den Text automatisch in einen String.

```
MCMethodDefinition >> #source
   ^ source asBlockString
```

Zwar werden jetzt die notwendigen Informationen herausgeschrieben, allerdings wird der kodierte String noch nicht wieder in einen Text übersetzt und kann somit nicht kompiliert werden.

Dies wird dadurch gelöst, dass beim Erstellen der Methodendefinition der String zurück in einen Text konvertiert wird. Dies geschieht wie in Listing 5.8 zu sehen beim initialisieren einer `MCMethodDefinition`. Die Methodenendefinitionen verwenden dann direkt den Text zum kompilieren. Somit können die visuellen Programmierelemente über Monticello übertragen werden.

Listing 5.8: Die Initialisierung einer `MCMethodDefinition` wurde angepasst, um den String zurück in einen Text zu konvertieren.

```
MCMethodDefinition >> #initializeWithClassName: classString
  classIsMeta: metaBoolean
  selector: selectorString
  category: catString
  timeStamp: timeString
  source: sourceString
    className := classString asSymbol.
    selector := selectorString asSymbol.
    category := catString
                    ifNil: [Categorizer default]
                    ifNotNil: [catString asSymbol].
    timeStamp := timeString.
    classIsMeta := metaBoolean.
    source := sourceString withSqueakLineEndings asBlockText
```

5.4 Serialisieren: Objekt-Graphen umwandeln

Um Programmcode über die Systemgrenze von Squeak hinaus zu transportieren, muss es möglich sein diesen in serieller Form darzustellen. Dies ermöglicht das schreiben ins Dateisystem oder das übertragen über eine Netzwerkverbindung.

Squeak bietet bereits mehrere Möglichkeiten, um Objekte zu serialisieren. Die verschiedenen Vor- und Nachteile der existierenden Verfahren werden in diesem Kapitel diskutiert, um zu entscheiden, welches Format am besten geeignet ist um die visuellen Elemente zu serialisieren.

5.4.1 Übersicht Problemraum

Wie in der Arbeit „*Toward Version Control in Object-based Systems*“ [91, Kapitel *Generic snapshot format for objects*] nachzulesen, lassen sich Objekte generell als eine Menge von *Slots* betrachten. Jeder *Slot* kann ein anderes Objekt referenzieren.

Um ein Objekt zu serialisieren ist es notwendig auch die Objekte zu serialisieren, auf die durch die *Slots* verwiesen wird. Wenn das serialisierte Objekt deserialisiert wird, müssen die Verweise auf diese Objekte wiederhergestellt werden.

Weiterhin zeichnet sich ein Objekt dadurch aus, welches Verhalten es auslöst wenn es eine Nachricht von einem anderen Objekt erhält. Laut „*Smalltalk-80: The Language*“ [47] wird das Verhalten eines Objekts in Smalltalk durch seine Klasse bestimmt.

> „*A class describes the form of its instances' private memories and it describes how they carry out their operations.*“

Wenn ein Objekt serialisiert wird, muss auch sichergestellt werden, dass das Verhalten des Objekts beim deserialisieren wiederhergestellt werden kann. Dabei setzt sich das Verhalten bei Erhalt einer Nachricht aus der Klasse und den beinhalteten Verweisen zusammen.

Das Problem der Serialisierung kann als Graphenproblem betrachtet werden, bei dem jedes Objekt einen Knoten und jeder Verweis eine Kante darstellt. Dabei ist wichtig, dass alle Verweise, sowie die strukturellen Informationen des Objekts beim deserialisieren wiederhergestellt werden können. Strukturelle Informationen eines Objekts sind die Klasse des Objekts, die Anzahl und Benennung der Instanzvariablen, sowie die Anzahl der indizierbaren Variablen. Da sich diese strukturellen Informationen durch Bearbeitung der Klasse ändern können ist es notwendig die seriellen Objekte zu neuen Versionen der Klasse migrieren zu können.

SandBlocks ermöglicht es dem Programmierer eigene visuelle Elemente zu entwickeln. Dementsprechend muss das genutzte Serialisierungsverfahren mit beliebigen Objekten umgehen können, da abgesehen von einem kleinen Interface, keine Annahmen darüber getroffen werden können, wie die visuellen Elemente aufgebaut sind. Auch sollte das Verfahren mit Änderungen an den Strukturinformationen umgehen können, da es dem Programmierer erlaubt sein soll die visuellen Elemente weiterzuentwickeln und somit die Strukturinformationen zu verändern.

5.4.2 Bestehende Serialisierungsverfahren in Squeak

Im Squeak-Programmiersystem existieren bereits mehrere Lösungen zur Serialisierung von Objekten. Übliche Serialisierungsverfahren im Squeak-System sind mittels `storeOn:` nach Smalltalk, JSON/STON[12] oder binär via `SmartRefStream`.

[12] STON auf GitHub: `https://github.com/svenvc/ston`, letzter Zugriff am 27. Juli 2019.

5.4.2.1 storeOn:

Da in Squeak jederzeit Smalltalk-Code ausgeführt werden kann, ist es naheliegend ein Serialisierungsverfahren zu verwenden, welches für ein Objekt Code produziert, der das Objekt programmatisch wiederherstellt. Dies ist die Aufgabe der Methode `storeOn:`. Diese fügt einem `WriteStream` den Code hinzu, der notwendig ist um ein äquivalentes Objekt zu erzeugen.

Das Objekt kann dann wiederhergestellt werden, indem der Code evaluiert wird. Problematisch ist bei dieser Methode allerdings, dass rekursiv alle Referenzen des Objekts auf den `WriteStream` geschrieben werden. Dabei wird allerdings für jede Referenz nicht nur die Referenz, sondern das gesamte Objekt gespeichert. Ein zyklischer Objektgraph erzeugt somit eine Endlosrekursion.

Außerdem ist dieses System nicht gegen Änderungen an den strukturellen Informationen abgesichert. Allein das Umbenennen einer Klasse führt zu Fehlern, da der generierte Code die Instanz über den Klassennamen erzeugt. Die Verwendung von `storeOn:` ist dementsprechend ungeeignet, da es Nutzern der visuellen Elemente möglich sein soll, komplexe Objekte zu konstruieren und deren Verhalten im Nachhinein zu bearbeiten.

5.4.2.2 JSON/STON

JSON (kurz für: JavaScript Object Notation) ist aufgrund seiner Verbreitung in der Webentwicklung weit unterstützt. Laut RFC8259 ist ein Objekt in JSON ein *„pair of curly brackets surrounding zero or more name/value pairs (or members)“*, also eine Liste von Namens-Wert-Paaren [16]. *„Smalltalk-80: The Language“* [47, Kapitel 1] beschriebt ein Smalltalk-Objekt allerdings auch als Instanz einer Klasse (*„Every object in the Smalltalk-80 system is an instance of a class“*). Diese Information wird von JSON nicht übernommen, da Objekte in Javascript keine Klasse besitzen. Außerdem ist es in JSON nicht möglich mehrere Referenzen auf dasselbe Objekt zu erzeugen.

Beide Probleme werden von der Smalltalk Object Notation gelöst. Wie in *„Enterprise Pharo“* im Kapitel *STON: a Smalltalk Object Notation*, beschrieben, basiert die Notation auf JSON und behebt beide Probleme [9]. STON erweitert JSON-Objekte um die Information der zugehörigen Klasse. Dabei behält STON die Grundstruktur des JSON-Formats weitgehend bei. Es können außerdem mehrere Referenzen auf dasselbe Objekt erzeugt werden.

Somit erlaubt es STON zyklische Referenzen zu serialisieren und auch wieder auszulesen. Allerdings ist auch STON nicht robust gegenüber Änderungen an den Metainformationen. Da der Klassenname zur Identifizierung verwendet wird und Instanzvariablen auch per Namen identifiziert werden ist es in der derzeitigen STON-Implementierung nicht möglich Klassen oder Variablen umzubenennen und das Objekt danach noch zu laden.

5.4.2.3 SmartRefStream

Um mit Veränderungen an den strukturellen Informationen umgehen zu können, wurde der `SmartRefStream` entwickelt. Der `SmartRefStream` ist eine Subklasse der Klasse `ReferenceStream`, welche wiederum eine Subklasse der `DataStream`-Klasse ist. Diese Streams schreiben Objekte in einem binären Format, in dem für jedes

serialisiertes Objekt erst der Typ des Objekts (Literal, Klasse, Instanz, etc.) und danach der Inhalt des Objekts geschrieben wird.

Ein `DataStream` kann genau wie JSON oder `storeOn:` keine Referenzen speichern und kann somit nur azyklische Objektgraphen serialisieren. Um Referenzen zu speichern wird der `ReferenceStream` verwendet, der diese Funktionalität anbietet.

Die Klasse `SmartRefStream` erweitert die ReferenceStream-Klasse und speichert zusätzlich zu jedem Objekt die Strukturinformationen die notwendig sind, um das Objekt wiederherstellen zu können, auch nachdem sich die Klasse geändert hat. Die gespeicherten Strukturinformationen sind:

1. Name der Klasse
2. Anzahl und Benennung der Instanzvariablen
3. Versionsnummer der Klasse[13]

Anhand dieser strukturellen Informationen kann dann das gespeicherte Objekt wiederhergestellt werden, auch wenn sich die Klasse im System geändert hat.

Der `SmartRefStream` bietet insgesamt Möglichkeiten Änderungen der folgenden Strukturinformationen zu migrieren:

- Umbenennung der Klasse
- Umbenennung der Instanzvariablen
- Hinzufügen/Entfernen von Instanzvariablen

Diese Migrationen werden durch zwei Systeme ermöglicht:

1. Migration einer Klassenumbenennung: Wenn eine Instanz einer unbekannten Klasse deserialisiert wird, fragt der SmartRefStream den Nutzer nach dem neuen Namen der Klasse. Instanzen, die die gleiche Klassenstruktur aufweisen werden in Zukunft automatisch migriert.
2. Migration von Änderungen der Klassenstruktur: Jede Klasse kann die Methode `convertToCurrentVersion:refStream:` überschreiben. Diese Methode nimmt als Argumente eine Menge an Assoziationen und den `SmartRefStream` selbst entgegen. Die Assoziationen assoziieren die alten Variablennamen mit den Werten. Anhand dieser Informationen kann das Objekt in die neue Klassenversion konvertiert werden.

Der `SmartRefStream` ist das einzige der vorgestellten Verfahren, dass die zu Beginn gestellten Anforderungen erfüllt. Der Stream kann mit Änderungen an den Strukturinformationen umgehen und ermöglicht es beliebige Objektgraphen serialisieren. Deshalb wurde der `SmartRefStream` als Verfahren zum Serialisieren der visuellen Programmierelemente gewählt.

[13] Die Versionsnummer wird manuell gesetzt, um zu signalisieren, dass sich die Interpretation der Instanzvariablen geändert hat, ohne die eigentliche Klassenstruktur zu ändern.

Da jeder `SmartRefStream` in einem binären Format serialisiert, muss die Ausgabe des Streams noch einmal in einen Text konvertiert werden. Dies ist notwendig, da wie in Abschnitt 5.3.4 angesprochen, Programmcode in einen `String` konvertiert wird. Diese sind in Squeak ASCII-kodiert und dürfen keinen beliebigen Binärcode beinhalten, da spezielle Binärcode-Sequenzen mit der ASCII-Kodierung interferieren können. So beendet zum Beispiel ein Byte mit Inhalt `0x00` den gesamten String.

Um dies zu umgehen wird die ASCII85-Implementierung von Squeak verwendet, um die binäre Ausgabe des `SmartRefStream` in eine Abfolge von ASCII-Charakteren zu konvertieren.

5.4.3 Lösen von Serialisierungsproblemen auf Objektebene

Ein Nachteil davon, dass der `SmartRefStream` gewählt wurde ist, dass dessen Serialisierungsformat nur geringfügige Anpassungen durch das Objekt erlaubt. In diesem Format können nur direkte Verweise zwischen Objekten gespeichert werden. Wenn dennoch zusätzliche oder andere Informationen gespeichert werden sollen, müssen diese Informationen direkt im Objektgraphen ersetzt werden.

5.4.3.1 Speichern zusätzlicher Informationen

Unter Umständen können für ein Objekt wichtige Informationen nicht direkt im Objekt gespeichert sein. Diese müssen für die Serialisierung in den Objektgraphen des Objekts eingefügt werden, um nicht verloren zu gehen. In diesem Abschnitt wird das Konzept des *Wrappers* vorgestellt, das für die Serialisierung diesen Zweck erfüllt. Das Konzept wird am Beispiel des *Observer*-Entwurfsmusters vorgestellt.

Das *Observer*-Entwurfsmuster wird in „*Design patterns: elements of reusable object-oriented software*“[43, Kapitel *Behavioral Patterns*] vorgestellt. Es beschreibt eine Möglichkeit eine eins-zu-viele-Beziehung zwischen Objekten zu modellieren indem dem einen Objekt (genannt Subjekt bzw. engl. *subject*) eine Liste von Beobachtern (engl. *observers* oder *dependents*) zugewiesen wird. Wann immer sich das Subjekt ändert benachrichtigt es alle Beobachter über eine fest definierte Schnittstelle, sodass Beobachter verschiedenster Typen auf Änderungen des Objekts reagieren können.

In Squeak wird eine Standardimplementierung dieses Entwurfsmusters angeboten, welche es erlaubt einem beliebigen Objekt eine beliebige Anzahl Beobachter zuzuweisen. In einem typischen Squeak-System benötigt nur eine geringe Anzahl der Objekte Beobachter (in einem aktuellen[14] Squeak-System haben 7 von 501.297 Objekten einen oder mehrer Beobachter). Deshalb beinhaltet nicht jedes Objekt eine Referenz auf seine Beobachter, da diese Referenz im Großteil der Objekte (≈99,999 % der Objekte im vorherigen Beispiel) überflüssig wäre. Aus diesem Grund werden alle Subjekte und deren Beobachter in einer globalen Variable, den `DependentsFields` gespeichert.

[14] Es wurde Squeak in der Version 5.2 (Update 18229) verwendet.

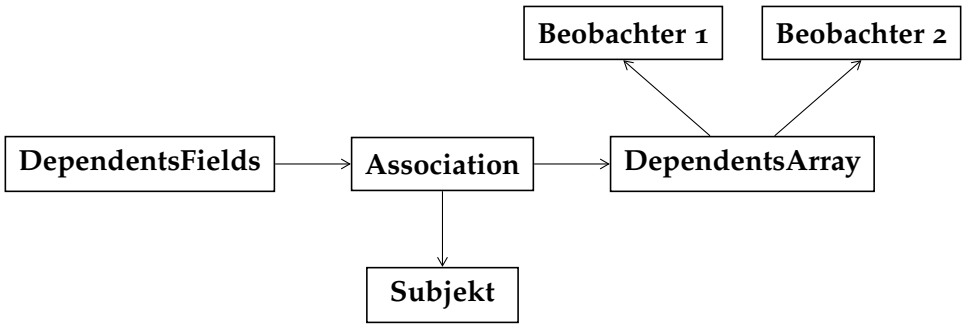


Abbildung 5.4: Die Beziehung zwischen dem Subjekt und seinen Beobachtern in der Squeak-Standardimplementierung. Das Subjekt beinhaltet keine Referenz, um die Subjekt-Beobachter-Beziehung zu speichern.

Wie in Abbildung 5.4 zu sehen, besitzt das Subjekt keine direkte Referenz, die die Subjekt-Beobachter-Beziehung abbildet. Wird nun das Subjekt mithilfe des `SmartRefStream` serialisiert geht diese Subjekt-Beobachter-Beziehung verloren, da sie in dem vom Subjekt aufgespannten Objektgraphen nicht enthalten ist. Da diese Beziehung aber für das Verhalten des Programms wichtig ist, muss diese Beziehung im dem zu serialisierenden Objektgraphen anders dargestellt werden.

Um solche Änderungen an dem zu serialisierenden Objektgraphen vornehmen zu können ist es möglich die Methode `objectForDataStream:` im Subjekt zu überschreiben. Diese Methode kann ein Objekt zurückgeben, welches stellvertretend für das eigentliche Objekt serialisiert wird.

Um Beobachter speichern zu können wird ein *Wrapper* erzeugt, der stellvertretend für ein beliebiges Subjekt steht und sowohl das Subjekt als auch dessen Beobachter referenziert, wie in Abbildung 5.5 zu sehen. Um einen solchen `DependentsWrapper` zu verwenden wird die `objectForDataStream:`-Methode auf dem Subjekt wie in Listing 5.9 zu sehen so implementiert, dass ein `DependentsWrapper` für das Subjekt zurückgegeben wird. Da dann sowohl Subjekt als auch Beobachter im Objektgraphen als solche referenziert werden geht diese Beziehung beim serialisieren nicht mehr verloren.

Anzumerken ist hier, dass das Subjekt bei Erstellung des `DependentsWrapper` durch eine flache Kopie[15] ersetzt wird. Dies ist notwendig, da das Objekt im serialisierten Objektgraphen automatisch durch den *Wrapper* ersetzt wird. Der `SmartRefStream` übernimmt das auch für den *Wrapper* selbst. Dieser kann also keine Referenz zu dem Subjekt speichern, da diese durch eine Referenz auf den *Wrapper* selbst ersetzt würde. Durch Erstellung der flachen Kopie kann das Problem umgangen werden, da technisch gesehen ein anderes Objekt referenziert wird.

Beim Deserialisieren eines solchen `DependentsWrapper` wird das Subjekt wieder an die Stelle des *Wrappers* eingesetzt. Außerdem werden dem Subjekt die Beob-

[15] Eine Kopie ist flach, wenn die vom Original referenzierten Objekte weiterhin geteilt werden.

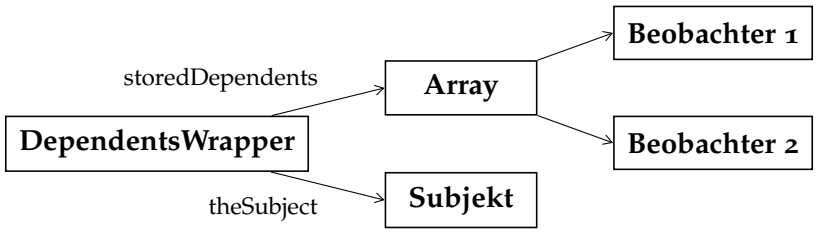


Abbildung 5.5: Der Objektgraph der serialisiert wird bildet die Subjekt-Beobachter-Beziehung direkt ab.

Listing 5.9: Die Methode `objectForDataStream:` des Subjekts gibt einen Dependents-Wrapper zurück, um Beobachter speichern zu können.

```
1 MyClass >> #objectForDataStream: aDataStream
2    ^ self as: DependentsWrapper
```

achter zugewiesen. Um einen solchen Konvertierungsvorgang nach dem Laden eines Objekts zu unterstützen wird nach dem Laden eines Objekts die Methode `comeFullyUpOnReload:` aufgerufen, in der das Objekt die Möglichkeit hat nötige Konvertierungen vorzunehmen und ein anderes Objekt an seiner Stelle zurückzugeben. In Listing 5.10 ist die Implementierung dieser Methode zu sehen.

Listing 5.10: Die Methode `comeFullyUpOnReload:` von `DependentsWrapper` fügt die Beobachter zum Subjekt hinzu und gibt das Subjekt zurück.

```
1 DependentsWrapper >> #comeFullyUpOnReload: aDataStream
2    self storedDependents do: [:each|
3       self theSubject addDependent: each].
4    ^ self theSubject
```

Durch Verwendung des `DependentsWrappers` kann also ein beliebiges Subjekt mitsamt den Beobachtern serialisiert werden, ohne die Subjekt-Beobachter-Beziehung zu verlieren.

Da das Serialisierungsformat nicht angepasst wurde bleiben die Fähigkeiten des `SmartRefStream` zur Migration der Objekte in neue Versionen der Klassen weiterhin erhalten. Verschiedene *Wrapper* können kombiniert werden, um beliebige zusätzliche Informationen zu einem Objekt zu speichern.

5.4.3.2 Komprimierung

Neben dem Abspeichern von Daten, die nicht direkt über den Objektgraphen erreichbar sind, kann es von Vorteil sein bestimmte Daten in komprimierter Form zu speichern. Dies ist insbesondere für Rastergrafiken sinnvoll. Diese beinhalten große Datenmengen, die durch Verfahren wie die PNG-Komprimierung verlustfrei komprimiert werden können, um die Datenmenge zu reduzieren. Da das gewählte

Serialisierungsverfahren es nicht erlaubt die Daten beim Schreiben zu komprimieren muss die Komprimierung bereits auf Objektebene vorgenommen werden.

Eine Rastergrafik in Squeak wird von Instanzen der Klasse `Form` repräsentiert. Diese beinhalten die Daten unkomprimiert in einer `BitMap`. Um die Daten für die Serialisierung zu ändern ist es erneut möglich die Methode `objectForDataStream:` zu verwenden, um ein Platzhalter-Objekt zurückzugeben, welches die Daten nur noch in komprimierter Form beinhaltet.

In diesem Fall handelt es sich nicht um einen *Wrapper*, sondern um ein vollständiges Ersetzen des Objekts für die Serialisierung. Für den konkreten Fall der Bildkomprimierung wurde die Klasse `CompressedForm` entwickelt. Wie in Listing 5.11 zu sehen, beinhaltet eine CompressedForm die als PNG komprimierten Daten der Form als `ByteArray`.

Listing 5.11: Eine `CompressedForm` konvertiert eine andere Form automatisch in ein `ByteArray`. In diesem sind die Daten im PNG-Bildformat komprimiert.

```
CompressedForm >> #setForm: aForm
   | imageStream |
   imageStream := WriteStream on: ByteArray new.
   (PNGReadWriter on: imageStream)
      nextPutImage: aForm;
      close.
   self bytes: imageStream contents.
```

Durch diese Komprimierung konnte im Fall einer 1020x520 Pixel umfassenden Rastergrafik eine Reduzierung von 1.471.505 Bytes auf 452.426 Bytes erreicht werden. Eine Reduzierung der Ausgabegröße auf ≈30,74% der ursprünglichen Größe. Dementsprechend reduziert sich auch die Zeit zum Übertragen der Daten auf ein Drittel.

Um die Rastergrafik beim Deserialisieren wiederherzustellen, konvertiert die Methode `comeFullyUpOnReload:`, wie in Listing 5.12 zu sehen, die Daten zurück in eine `Form`.

Listing 5.12: Während des Ladevorgangs wird die `CompressedForm` wieder in eine Form zurückkonvertiert.

```
CompressedForm >> #comeFullyUpOnReload: aSmartRefStream
   ^ self asForm

CompressedForm >> #asForm
   | reader image |
   reader := (PNGReadWriter on: (ReadStream on: self bytes)).
   image := reader nextImage.
   reader close.
   ^ image
```

Neben der Komprimierung von Objekten kann das Verfahren, ein Objekt durch ein `ByteArray` zu ersetzen, genutzt werden, um ein beliebiges Objekt in ein beliebiges Serialisierungsformat zu übersetzen. Somit kann mithilfe des `SmartRefStream` jede Struktur, die sich in serieller Form darstellen lässt, auch serialisiert werden.

Diese Möglichkeit sollte nur verwendet werden, wenn das andere Serialisierungsformat substantielle Vorteile gegenüber dem Standardformat des `SmartRefStream` bietet. Bei der Vermischung verschiedener Serialisierungsverfahren kann es zu Problemen kommen, wie beispielsweise, dass Objekte im einen Serialisierungsformat nicht vom anderen Format referenziert werden können. Dies führt dazu, dass dieselben Objekte in unterschiedlichen Serialisierungsformaten einzeln serialisiert wird und somit Kopien des Objekts entstehen.

5.4.4 Auswertung

Durch Nutzung des `SmartRefStream` ist es prinzipiell möglich beliebige Objektgraphen zu serialisieren.

Dies bietet den großen Vorteil, dass neue Typen visueller Elemente eingeführt werden können, ohne dass Anstrengungen unternommen werden müssen, um diese serialisieren zu können. Allerdings ergibt sich dadurch das Problem, dass nur ein Objektgraph serialisiert werden kann. Die dadurch entstehenden Probleme, wie das Speichern von indirekten Referenzen, lassen sich durch erzeugen eines passenden Objektgraphen lösen.

Somit ist es möglich mit dem gewählten Verfahren beliebige visuelle Elemente zu serialisieren, wobei im Normalfall keine Arbeit vom Blockentwickler notwendig ist.

5.5 Austauschen: Ermittlung von Änderungen

Nun ist es technisch möglich, visuelle Programmierelemente zu übertragen. Allerdings behandeln die Monticello-basierten Versionsverwaltungsverfahren den Programmcode immer noch als String. Änderungen an den visuellen Elementen werden nicht richtig erkannt, da nur der serialisierte String der Elemente verglichen wird. Außerdem wird dem Nutzer auch nur der serialisierte Programmtext angezeigt, was es schwer macht zu ermitteln, ob die visuellen Elemente im Programmcode den richtigen Zustand haben.

Diese Probleme werden gelöst, indem sowohl das Ermitteln als auch das Anzeigen der Änderungen angepasst werden. Beide Systeme sollen visuelle Elemente als solche erkennen und diese speziell behandeln.

5.5.1 Anzupassende Systeme

Bevor damit angefangen werden kann, die Ermittlung von Änderungen anzupassen, müssen alle Systeme identifiziert werden, die angepasst werden müssen.

Wie in Abschnitt 5.3 bereits besprochen lassen sich die Squeak-Systeme zum Teilen von Programmcode in zwei Kategorien einteilen. Die Smalltalk-Dateien-basierten Systeme ermitteln Änderungen nicht durch Vergleich des Programmcodes. Smalltalk-Dateien beschreiben immer den gesamten aktuellen Zustand, erkennen also Änderungen überhaupt nicht. *Changesets* hingegen basieren, wie der Name impliziert, auf dem Erkennen von Änderungen. Eine Änderung im Sinne eines *Changesets* ist das Speichern einer Methode. Dieses Ereignis signalisiert auch bei Programmcode, der visuelle Elemente beinhaltet, eine Änderung und kann somit von den *Changesets* weiter als Änderungserkennung verwendet werden. Dadurch ist es nicht nötig die Änderungserkennung von Smalltalk-Dateien-basierten Verfahren zu verändern.

Monticello-basierte Systeme hingegen, vergleichen zum Erkennen von Änderungen die alte Version mit der neuen Version. Diese Erkennung muss angepasst werden, um das Vergleichen visueller Elemente zu unterstützen. Von Monticello versionierte Elemente sind Instanzen der Subklassen von `MCDefinition`. Es wird der Operator `=` verwendet um zwei solcher Instanzen auf Änderungen zu überprüfen. Die Implementierung dieses Operators muss angepasst werden.

Wenn eine Änderung erkannt wird, verwenden alle genannten Systeme die Klasse `TextDiffBuilder` um die Änderungen anschaulich aufzubereiten. Die Klasse analysiert einen String[16] zeilenweise auf Änderungen und markiert neu hinzugefügte Zeilen, sowie entfernte Zeilen. Veränderte Zeilen zählen als das Entfernen der alten Zeile und Einfügen der neuen Zeile. Da `TextDiffBuilder`-Instanzen ein unabhängiges Vergleichsverfahren verwenden, muss dieses angepasst werden, um die Änderungen an den visuellen Elementen zu erkennen und zu markieren.

5.5.2 Vergleichen visueller Elemente

Wie in Abschnitt 5.3.1 beschrieben, verhält sich ein visuelles Element aus Sicht des Textes wie ein einzelner Charakter. Somit sollten sich visuelle Elemente auch beim Ermitteln von Unterschieden wie ein einzelner Charakter verhalten. Es ist allerdings nicht möglich nur den im String enthaltenen Charakter zum Ermitteln von Änderungen zu verwenden. Anstelle des visuellen Elements steht nur der SOH-Charakter im String. Visuelle Elemente würden immer als gleich gelten, da diese immer vom selben Charakter repräsentiert werden.

Es ist auch nicht möglich einfach den serialisierten String zu vergleichen. Dieser kann Daten beinhalten, die für die Semantik des Programmcodes nicht relevant sind. Beispielsweise beinhaltet jedes visuelle Element die absolute Position in der Welt. Diese kann sich, durch verschieben des Codebrowsers oder durch plattformabhängige Informationen wie der Bildschirmauflösung ändern, ohne dass die Funktionalität des visuellen Elements beeinflusst wird. Da das gewählte Serialisie-

[16] Die Benennung der Klasse `TextDiffBuilder` ist irreführend. Die Standardimplementierung des `TextDiffBuilder` unterstützt nur das Vergleichen von Strings und nicht von Text wie in Abschnitt 5.3.1 differenziert.

rungsverfahren diese Informationen allerdings dennoch serialisiert, werden diese im serialisierten String für eine Änderung sorgen, auch wenn sich der eigentliche Programmcode nicht geändert hat.

Es müssen somit jedes Mal, wenn beim Vergleichen des Programmstrings SOH-Charakter angetroffen werden, die visuellen Elemente an diesen Stellen ermittelt und direkt miteinander verglichen werden.

–

In Squeak wird der Gleichheitsoperator „`=`“ verwendet, um zu prüfen ob zwei beliebige Objekte „*gleich*“ sind. Laut „*Smalltalk-80: The Language*“ [47] hat der Gleichheitsoperator den Zweck zu prüfen ob zwei Objekte dieselbe Komponente repräsentieren.

> *„Equality (=) is the test of whether two objects represent the same component. The decision as to what it means to be 'represent the same component' is made by the receiver of the message; [..]“*

Im Squeak-System wird der Operator verwendet um zu ermitteln, ob zwei Objekte komplett austauschbar behandelt werden können.

Dies ist allerdings eine andere Aufgabe als zu überprüfen, ob zwei visuelle Programmierlemente die gleiche Semantik aufweisen. SandBlocks erlaubt es durch die Nutzung von *Traits*, in einer beliebigen Vererbungshierarchie ein visuelles Element zu erstellen, wie bereits in Abschnitt 2.4.4 beschrieben wurde. Dadurch kann allerdings die Definition davon, ob ein Objekt austauschbar behandelt werden kann, damit konfligieren, ob die visuellen Elemente austauschbar sind. Dies ist insbesondere bei visuellen Objekten der Fall, da diese nie austauschbar zu behandeln sind. Wenn diese allerdings als visuelles Element agieren, können diese als solche dennoch „*gleich*“ sein.

Um diese Änderungen zu reflektieren wird ein weiterer Gleichheitsoperator eingeführt. Dieser Operator vergleicht zwei Objekte darauf, ob sie bei der Ausführung dieselbe Semantik haben. Umgesetzt ist dieser Operator als die Methode `blockEquals:`.

Die Standardimplementierung von `blockEquals:` verwendet den normalen Gleichheitsoperator, da dieser für die meisten Fälle eine ausreichende Implementierung bietet. Dies hat allerdings zur Folge, dass sobald ein Objekt nicht `blockEquals:` selbst implementiert, die gesamte restliche Gleichheitsüberprüfung den `=`-Operator verwendet. Die visuellen Elemente müssen selbst sicherstellen, dass die referenzierten Objekte eine ausreichende Implementierung von `blockEquals:` besitzen oder der `=`-Operator für diese ausreicht.

Mehrere Gleichheitsoperatoren anzubieten, ist in anderen Programmiersprachen bereits ein bekanntes Konzept. Die Programmiersprache Ruby besitzt beispielsweise vier verschiedene Gleichheitsoperatoren: `==`, `===`, `eql?` und `equal?` [39, Abschnitt 3.8.5 *Object Equality*]. Ruby verwendet diese Operatoren für die Interaktion von Objekten mit Sprachfeatures, wie der `case/when`-Kontrollstruktur.

`blockEquals:` erfüllt einen ähnlichen Zweck, indem der Operator festlegt, wie das Objekt sich als Teil des Programmcodes verhält.

Der neue Gleichheitsoperator wird im Folgenden in das System eingebunden, um visuelle Elemente miteinander zu vergleichen. Wenn im Programmcode zwei visuelle Elemente als ungleich gelten, dann wird dies behandelt, als ob die Charaktere, an denen sie sich befinden, ungleich sind.

5.5.3 Implementierung

Zuerst wird die `MCDefinition` angepasst, um Änderungen richtig ermitteln zu können. Für Programmcode wird der `=`-Operator der Klasse `MCMethodDefinition` angepasst.

Im bestehenden System vergleicht der `=`-Operator nur die `source` der Definition. Dabei handelt es sich um den serialisierten Programmcode. Die `source` muss entsprechend mit der `asBlockText`-Methode zurück in einen Text konvertiert werden, um diesen auf Gleichheit überprüfen zu können.

Wie im Abschnitt B unter Listing 5.16 nachzulesen verwendet die Klasse `Text` allerdings zur Gleichheitsprüfung nur den String und ignoriert die Textattribute.

Da nun die Textattribute Einfluss auf die Semantik besitzen, passt der Gleichheitsoperator des Texts nicht mehr zu der durch `blockEquals:` getroffenen Aussage: *„Haben diese Objekte dieselbe Auswirkung auf die Ausführung?“*. Die Klasse `Text` wurde dementsprechend um eine eigene Implementierung von `blockEquals:` erweitert, welche auch die Textattribute via `blockEquals:` auf Gleichheit überprüft.

Listing 5.13: Der `=`-Operator von `MCMethodDefinition` wurde angepasst um `blockEquals:` als Vergleichsoperator für den Programmcode der Methode zu verwenden.

```
1 MCMethodDefinition >> #= aDefinition
2    ^ (super = aDefinition)
3          and: [aDefinition sourceText blockEquals: self sourceText]
4          and: [aDefinition category = self category]
5          and: [aDefinition timeStamp = self timeStamp]
6
7 MCMethodDefinition >> #sourceText
8    ^ source asBlockText
```

`Text`, `RunArray` und `BlockAttribute` stellen in ihren Implementierungen von `blockEquals:` sicher, dass die visuellen Elemente auch mit `blockEquals:` verglichen werden. Diese Implementierungen können im Abschnitt B unter Abschnitt B.1 nachgelesen werden.

Der Gleichheitsoperator der `MCMethodDefinition` wurde angepasst um den Text mit `blockEquals:` zu vergleichen, wie in Listing 5.13 zu sehen. Dies ist auch die richtige Aussage für den Gleichheitsoperator, da zwei Methodendefinitionen nur dann austauschbar behandelt werden können, wenn sie tatsächlich Programmcode beschreiben, der dieselbe Semantik besitzt.

Durch Nutzung von `blockEquals:` können visuelle Programmierelemente selbst festlegen, wann eine Änderung vorliegt. Die Erkennung der Änderungen ist dementsprechend unabhängig vom gewählten Serialisierungsverfahren. Dies erlaubt es auch das Serialisierungsverfahren, ohne Auswirkung auf die Erkennung von Änderungen, auszutauschen.

Erkannte Änderungen werden allerdings noch nicht in der UI wiedergespiegelt. Dort wird weiterhin der serialisierte String angezeigt und Änderungen in diesem zeilenweise markiert. Diese markierten Änderungen sind aber unabhängig von den tatsächlich erkannten Änderungen. Um dieses Problem zu beheben wurde der `TextDiffBuilder` erweitert um Textattribute im Text beizubehalten und zum Vergleichen der einzelnen Zeilen `blockEquals:` zu verwenden. Der `TextDiffBuilder` verwendet zum Ermitteln der Änderungen die Klasse `DiffElement`. Der Gleichheitsoperator dieser Klasse wurde überarbeitet, sodass nun auch `blockEquals:` zum Vergleichen verwendet wird. Somit wird dasselbe Vergleichsverfahren in der Anzeige der Änderungen als auch in deren Ermittlung verwendet.

Für eine beispielhafte Änderung der Farbe der in Listing 5.4 eingeführten Methode von grün zu blau, ist die Anzeige der Änderungen in Listing 5.14 zu sehen. Die vorgenommene Änderung ist klar in den visuellen Elementen zu erkennen.

Listing 5.14: Die Anzeige der Änderungen behandelt visuelle Elemente wie einzelne Charaktere und markiert Änderungen an diese entsprechend.

```
1 MyClass >> #foo
2     ^    #0000FF
3     ^    #00FF00
```

Problematisch ist hier nur, dass die Markierung geänderter Zeilen durch Textattribute vorgenommen wird. Diese werden allerdings bei grafischen Textelementen nicht angezeigt, wodurch Änderungen an Zeilen, die nur visuelle Elemente beinhalten, keine sichtbare Markierung erhalten. Mögliche Lösung für diese Probleme werden in Abschnitt 5.8 diskutiert.

5.5.4 Auswertung

Durch die vorgenommenen Änderungen werden in Monticello-basierten Systemen also Änderungen nur dann erkannt, wenn tatsächlich Änderungen vorliegen. Die Entscheidung, wann eine Änderung vorliegt, kann dabei von den visuellen Elementen selbst getroffen werden. Dadurch können diese eine beliebige Struktur besitzen, da keine Annahmen darüber getroffen werden, wie diese aufgebaut sind.

Erkannte Änderungen werden bei der Markierung wie Änderungen an einzelnen Charakteren behandelt. Visuelle Elemente werden im markierten Text angezeigt. Änderungen sind somit direkt sichtbar.

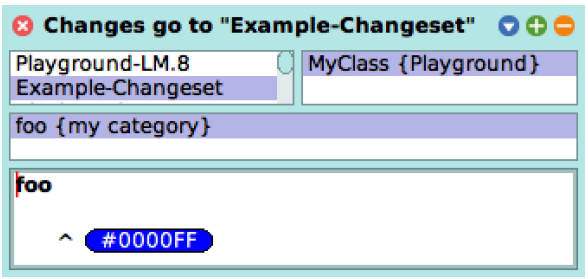


Abbildung 5.6: Ein *Changeset* zeigt die visuellen Elemente direkt im Programmcode.

5.6 Evaluation

Nachdem die vorgenommenen Änderungen erläutert wurden, gibt dieser Abschnitt einen Überblick über den resultierenden Arbeitsfluss. Es werden offenen Probleme aufgezeigt und potenzielle Lösungen für diese Probleme sowie weitere mögliche Verbesserungen diskutiert.

5.6.1 Übersicht Arbeitsfluss

Ziel der vorgenommenen Änderungen war es, dass der Programmierer seinen Arbeitsfluss nicht verändern muss, um visuelle Elemente teilen zu können. Dieses Ziel wurde erreicht, da Smalltalk-Dateien weiterhin aus Klassen, Paketen oder Methoden erstellt werden können. *Changesets* nehmen visuelle Elemente auf, diese werden, wie in Abbildung 5.6 zu sehen, visuell angezeigt und können herausgeschrieben werden.

Bei der Verwendung von Monticello-basierten Repositorien (engl. *repositories*) werden Änderungen richtig erkannt. Die erkannten Änderungen werden wie zuvor markiert, wie in Abbildung 5.7 zu sehen. Dies funktioniert auch mit der von Squot bereitgestellten Benutzerschnittstelle, welche eine eigene grafische Benutzerschnittstelle zum Anzeigen von Änderungen bereitstellt. Eine Abbildung davon ist unter Abschnitt A in Abschnitt A zu finden.

An den gezeigten visuellen Farbelementen wurden keine Änderungen vorgenommen, um diese serialisieren zu können. Das gewählte Serialisierungsverfahren kann diese automatisch serialisieren und deserialisieren.

5.6.2 Offene Probleme

5.6.2.1 Fehlerbehandlung bei automatischer Deserialisierung

Um die visuellen Elemente anzeigen zu können, müssen diese deserialisiert werden. Dies geschieht automatisch, um das Erkennen von Änderungen durch Monticello und die Anzeige des visuellen Elements zu ermöglichen. Die Behandlung von Fehlern bei dieser automatischen Deserialisierung ist allerdings noch nicht ausgereift.

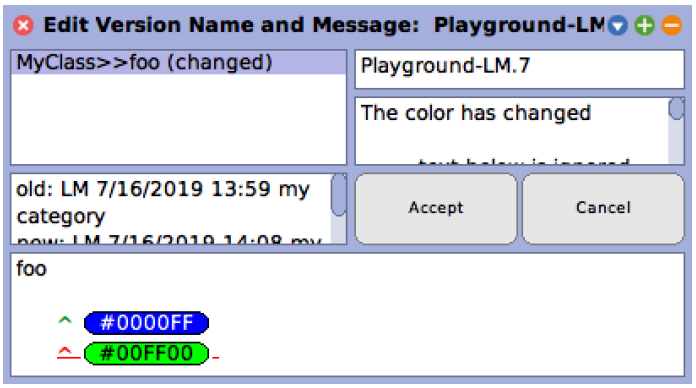


Abbildung 5.7: Monticello-basierte Verfahren markieren Änderungen an den visuellen Elementen.

In der momentanen Implementierung (Listing 5.17) werden alle Fehler abgefangen, der SOH-Charakter wird dem Text hinzugefügt, es fehlt nur das Textattribut. Dieser Fall wird vom Text-Zeichnungssystem speziell behandelt und es wird ein spezielles visuelles Element (SOH) gezeichnet. Dieses signalisiert dem Programmierer, dass an dieser Stelle das eigentliche visuelle Element fehlt.

Dieser Ansatz verhindert, dass Fehler in der Deserialisierung dazu führen, dass die Versionsverwaltungssysteme nicht mehr genutzt werden können. Dadurch wird der Programmierer aber nicht darüber benachrichtigt wird, welcher Fehler genau aufgetreten ist.

Ein weiteres Problem in der Fehlerbehandlung stellen modale Dialoge durch das Serialisierungsverfahren dar. Wie in Abschnitt 5.4.2.3 erläutert, wird der Nutzer über ein Dialogfeld nach dem Namen einer neuen Klasse gefragt, wenn die geforderte Klasse nicht gefunden werden konnte. Diese Eingabefelder sind notwendig um die Fähigkeit des `SmartRefStream`, mit der Umbenennung einer Klasse umgehen zu können, beizubehalten. Wenn die geforderte Klasse allerdings im aktuellen Squeak-System fehlt führt dies dazu, dass der Nutzer wiederholt nach der Klasse gefragt wird, bevor er die Möglichkeit hat das Versionsverwaltungssystem zu verwenden. Dies kann bei aktiver Entwicklung geschehen, wenn eine neue Klasse erst über das Versionsverwaltungssystem in das Squeak-System geladen wird. Visuelle Elemente die Objekte dieser neuen Klasse bereits verwenden sorgen dann für das besprochene Verhalten.

Beide Probleme treten allerdings nur auf, wenn die visuellen Elemente keine stabile Objektstruktur aufweisen. Programmierer, die visuelle Elemente nur benutzen, werden diese Probleme also nicht haben. Nur die Programmierer stoßen auf genannte Probleme, wenn sie visuelle Elemente selbst entwickeln, also deren Objektstruktur bearbeiten, oder eigene Objekte in ein visuelles Element einfügen.

5.6.2.2 Markierung grafischer Elemente ohne Text

Systeme, die den `TextDiffBuilder` verwenden zeigen das Hinzufügen und Entfernen von Zeilen durch Markierungen im Text an. Es handelt sich bei den visuellen

Elementen aber nicht um Text, weshalb diese selbst nicht markiert werden. Wenn ein visuelles Element nun allein in einer Zeile steht, dann ist bei einer Änderung nicht klar, welche Zeile hinzugefügt und welche entfernt wurde. In der Realität tritt dieser Fall selten auf, da die visuellen Elemente meist zurückgegeben, Variablen zugewiesen oder als Parameter für eine Methode verwendet werden. Dadurch sind die visuellen Elemente meist von Smalltalk-Code umgeben. Allerdings ist der Fall nicht unmöglich, dementsprechend sollten Änderungen in solchen Zeilen trotzdem markiert werden.

5.7 Verwandte Arbeiten

Im Bereich der Objektserialisierung und Versionierung heterogener visueller Programmiersprachen existieren noch weitere Arbeiten. Dieser Abschnitt soll einige dieser Arbeiten beleuchten und deren Verhältnis unserem Ansatz erklären.

5.7.1 Heterogeneous Visual Languages [36]

Die Konzepte dieses Artikels liefern die Grundlage des durch SandBlocks umgesetzten Programmiersystems. Allerdings schlagen Erwig & Meyer eine andere technische Umsetzung vor, indem Sie die visuellen Programmierelemente in Programmtext übersetzen. In einer solchen Umsetzung sind die in dieser Arbeit vorgenommenen Änderungen an der Integration und Serialisierung nicht zwingend notwendig.

5.7.2 Towards Version Control in Object-based Systems [91]

Diese Arbeit beschreibt das Squot Versionskontrollsystem und dessen Umsetzung. Das Ziel ist es Versionskontrolle für beliebige Smalltalk-Objekte zu ermöglichen. Es wäre möglich visuelle Elemente direkt mit Squot unter Versionskontrolle zu stellen. Dies hätte den Vorteil, dass Squot dafür gebaut wurde beliebige Objekte zu teilen, anstatt nur Programmtext. Es bietet somit beispielsweise die Möglichkeit die Objektidentität der visuellen Elemente zu bewahren, anstatt diese für die Übertragung zu kopieren.

5.7.3 Visual Comparison of Graphical Models [97]

In diesem Artikel beschreiben Schipper, Fuhrmann und von Hanxleden ein System, um Änderungen an beliebigen grafischen Modellen zu visualisieren. Obwohl dieser Artikel konkret nur auf das Vergleichen von Diagrammen eingeht, ist das Konzept auch auf anderen visuellen Elemente bei der Versionsverwaltung anwendbar.

5.7.4 Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing [58]

Das Zusammenführen von Änderungen an visuellen Elementen ist in dieser Arbeit, wie in Abschnitt 5.8 erwähnt, vernachlässigt worden. Eine Umsetzung, um dies zu ermöglichen, beschreiben Ignat und Norrie in ihrem 2004 veröffentlichten Artikel. Sie vergleichen dabei zustandsbasierte, als auch operationenbasierte Ansätze, um Änderungen an visuellen Elemente zusammenzuführen.

5.8 Zusammenfassung und Ausblick

Objekte direkt in den Programmcode einzufügen, bringt den schwerwiegenden Nachteil, dass Programmcode nicht mehr zwischen Systemen übertragen werden kann.

Durch die vorgestellten Änderungen ist es möglich, beliebige visuelle Objekte in die Teilungssysteme aufzunehmen. Visuelle Elemente können über externe Werkzeuge übertragen, die Änderungen visuell verglichen und in das Squeak-System eingepflegt werden. Das Vergleichen und Zusammenführen erfolgt dabei jedoch oft händisch und ist somit noch ausbaufähig.

Die Werkzeugunterstützung ist allerdings weiterhin stark an das Squeak-System gebunden. In externen Editoren können die visuellen Elemente nur als serialisierter String betrachtet werden. Dabei ist es kaum möglich sinnvolle Änderungen durchzuführen, da ein binäres Serialisierungsformat verwendet wird. Durch Verwendung eines anderen, textuell verständlichen und bearbeitbaren Serialisierungsformat kann dieses Problem gemindert werden.

Doch selbst mit dem binären Serialisierungsformat ist es möglich, visuelle Elemente zwischen Systemgrenzen zu übertragen, ohne den existierenden Arbeitsfluss zu ändern. Heterogener, visueller Programmcode kann durch die Squeak-Teilungssysteme genauso leicht geteilt werden wie normaler Programmcode.

Ausblick

In diesem Abschnitt wird auf die Arbeit eingegangen, die noch offenbleibt. Dies umfasst zum einen mögliche Lösungen für die in Abschnitt 5.6 angesprochenen Probleme, als auch weitere Änderungen, die das Teilen von visuellem Programmcode verbessern.

Die Fehlerbehandlung bei automatischer Deserialisierung sollte es dem Nutzer erlauben, den Fehler zu untersuchen, um die Ursache zu ermitteln. Dennoch sollte nicht für jeden Fehler automatisch ein Fenster geöffnet werden. Dies könnte dadurch gelöst werden, dass die bestehenden Benutzerschnittstellen erweitert werden, um das Untersuchen von Fehlern bei der Deserialisierung zu erlauben. Beispielsweise durch Hinzufügen eines Buttons der einen Debugger öffnet. Alternativ ist

es auch möglich alle Fehler in eine systemweite Ausgabe zu schreiben, die vom Nutzer eingesehen werden kann (In Squeak als *Transcript* bezeichnet).

Für die Markierung grafischer Elemente zur Änderungserkennung wäre es möglich, die visuellen Elemente selbst passend zum Text zu färben (z. B. rot für entfernt, grün für hinzugefügt). Dies ist allerdings problematisch, da bei visuellen Elementen, die Farbe selbst Bedeutung haben kann. Die in diesem Kapitel beispielhaft verwendeten visuellen Elemente, die eine Farbe repräsentieren, sind ein offensichtliches Beispiel für einen solchen Fall. Alternativ wäre es möglich den Hintergrund der Zeile, statt den Text zu färben. Dies funktioniert, solange das visuelle Element nicht die ganze Zeile einnimmt und den Hintergrund vollständig überdeckt.

Um das Markieren von Änderungen weiter zu verbessern, ist es möglich visuellen Elementen zu erlauben, Änderungen selbst feingranular anzuzeigen. In der derzeitigen Umsetzung werden visuelle Elemente als Einheit betrachtet. Es können somit immer nur ganze visuelle Elemente miteinander verglichen werden. Bei einfachen Elementen wie den Farbelementen ist das ausreichend. Komplexere visuelle Elemente wie Tabellen würden davon profitieren, wenn einzelne Teile miteinander verglichen werden könnten (z. B. die einzelnen Zellen). Dafür müsste es den visuellen Elementen möglich sein, Änderungen selbstständig anzuzeigen, wofür ein weiterer Ausbau der Programmierschnittstelle notwendig ist.

Es wurden bisher keine Anstrengungen unternommen das Zusammenführen von Änderungen (*Merging*) an visuellen Elementen zu ermöglichen. Das Squeak-System selbst bietet auch keine Möglichkeit, Änderungen an Smalltalk-Programmtext zusammenzuführen. Dies ist wahrscheinlich dem Fakt geschuldet, dass in Smalltalk der Programmcode einer Methode durchschnittlich nur etwa 9 Zeilen[17] umfasst. Monticello behandelt eine Smalltalk-Methode als kleinstmögliche Einheit. Solange nicht zwei verschiedene Personen derselben Methode arbeiten, ist es somit nicht notwendig die Änderungen zusammenzuführen. Sollte doch ein Zusammenführen notwendig sein ist dies bei so kurzen Methoden ohne großen Aufwand manuell möglich. Es ist allerdings noch nicht klar wie sich visuelle Elemente im Vergleich verhalten. Das Zusammenführen von Änderungen für diese könnte also ein wichtiges Feature sein.

[17] Ein aktuelles Squeak-System (Version 5.2, Update 18229) umfasst 549.542 Zeilen Smalltalk-Code, welche sich auf 60.564 Methoden aufteilen.

A Abbildungen

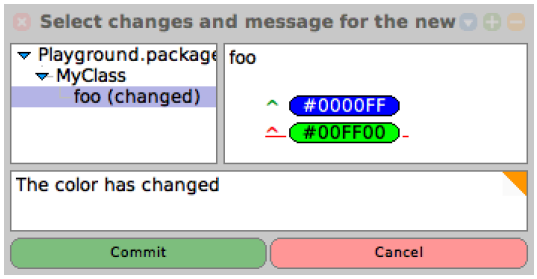


Abbildung 5.8: Die Anzeige einer Änderung eines visuellen Elements in der von Squot bereitgestellten Benutzeroberfläche.

B Quelltext

Listing 5.15: Die Implementierung der `asBlockString`-Methode der Klasse `Text`

```
Text >> asBlockString
   ^ String streamContents: [:writeStream | | readStream |
      readStream := string readStream.
      [readStream atEnd]
         whileFalse: [
            writeStream nextPutAll: (readStream upTo: Character startOfHeading).
            (readStream last = Character startOfHeading)
               ifTrue: [
                  "Only nextPut: the Character startOfHeading if the last
                      ↪ character was actually a startOfHeading, otherwise the
                      ↪ readStream hit the end."
                  writeStream nextPut: Character startOfHeading.
                  self attributesAt: readStream position do: [:textAttribute|
                     "Only write TextAttributes that have shoutShouldPreserve
                         ↪ enabled, otherwise we are also writing all the syntax
                         ↪  highlighting"
                     textAttribute shoutShouldPreserve ifTrue: [
                        textAttribute writeScanOn: writeStream]].
                  writeStream nextPut: Character startOfText]]].
```

Listing 5.16: Die Gleichheitsüberprüfung der Klasse `Text` prüft nur, dass der String gleich ist. Die Textattribute werden nicht auf Gleichheit überprüft.

```
Text >> #= other
   "Am I equal to the other Text or String?
   ***** Warning ***** Two Texts are considered equal if they have the same
       ↪ characters in them. They might have completely different emphasis,
       ↪ fonts, sizes, text actions, or embedded morphs. If you need to find out
       ↪  if one is a true copy of the other, you must do (text1 = text2 and: [
       ↪ text1 runs = text2 runs])."
   other isText ifTrue: [ ^ string = other string ].
   other isString ifTrue: [ ^ string = other ].
   ^ false
```

Listing 5.17: Die Implementierung der `asBlockText`-Methode der Klasse `String`, bestehend aus vier einzelnen Methoden

```
String >> #asBlockText
   ^ Text fromBlockString: self

Text class >> fromBlockString: aString
   ^ (self isBlockString: aString)
        ifTrue: [self parseBlockString: aString]
        ifFalse: [self string: aString attributes: #()].

Text class >> #isBlockString: aString
   ^ aString includesAllOf: {Character startOfHeading . Character startOfText}

Text class >> #parseBlockString: aString
   | readStream resultStream runs |
   readStream := ReadStream on: aString.
   resultStream := WriteStream on: ''.
   runs := RunArray new.
   [readStream atEnd]
      whileFalse: [ self
            parseBlockStringSequenceFrom: readStream
            into: resultStream
            withAttributes: runs].
   ^ self string: resultStream contents runs: runs

Text class >> #parseBlockStringSequenceFrom: aReadStream
 into: aWriteStream
 withAttributes: aRunArray
   | string |
   string := aReadStream upTo: Character startOfHeading.
   aWriteStream nextPutAll: string.
   aRunArray addLast: #() times: string size.
   aReadStream atEnd
      ifFalse: [
         | attributes |
         aWriteStream nextPut: Character startOfHeading.
         string := aReadStream upTo: Character startOfText.
         attributes := [TextAttribute readAttributesFrom: string]
            on: Error
            do: [ {} "When an error is encountered while reading the
                ↪ TextAttributes the SOH is still added but attributes will be
                ↪ empty."].
         aRunArray addLast: attributes].
```

B.1 blockEquals:

Dieser Abschnitt beinhaltet die Variationen der Methode `blockEquals:` in SandBlocks für `Object`, `BlockAttribute`, `Morph`, `Text` und `RunArray`.

Listing 5.18: In `Object` nutzt `blockEquals:` den einfachen Objektvergleich.

```
Object >> #blockEquals: anObject
   ^ self = anObject
```

Listing 5.19: In `BlockAttribute` nutzt `blockEquals:` den Vergleich von `blockModel`.

```
BlockAttribute >> #blockEquals: other
   ^ (other class = self class)
      and: [other blockModel = self blockModel]
```

Listing 5.20: In `Morph` versucht `blockEquals:` generisch zu ermitteln, ob sich an ihm etwas geändert hat. Die Position des Morphs wird ignoriert, da diese systemabhängig ist.

```
Morph >> #blockEquals: anObject
   "layouts and stuff break everything so do not check them"
   ^ (self class = anObject class
      and: [self color = anObject color]
      and: [self borderWidth = anObject borderWidth]
      and: [self extent = anObject extent])
      and: [self borderColor = anObject borderColor]
      and: [self submorphs size = anObject submorphs size]
      and: [
         self submorphs
            with: anObject submorphs
            do: [:first :second | (first blockEquals: second) ifFalse: [^ false]].
                ↪
         true]
```

Listing 5.21: In `Text` überprüft `blockEquals:` sowohl Zeichenkette (`string`) als auch Formatierung (`runs`).

```
Text >> #blockEquals: anotherObject
   ^ (anotherObject isText)
      ifTrue: [(runs blockEquals: anotherObject runs)
         and: [string blockEquals: anotherObject string]]
      ifFalse: [false]
```

Listing 5.22: In `RunArray` überprüft `blockEquals:` alle Attribute unter Zuhilfenahme von `blockRunsEqual:`.

```
RunArray >> #blockEquals: anotherObject
   ^ anotherObject class = self class
      and: [self blockRunsEqual: anotherObject]

RunArray >> #blockRunsEqual: aRunArray
   self size = aRunArray size and:
      [ self with: aRunArray do:
         [ :firstRun :secondRun |
            firstRun size = secondRun size and:
               [ firstRun with: secondRun do:
                  [ :firstAtt :secondAtt | (firstAtt blockEquals: secondAtt)
                                                ifFalse: [^ false]]]]].
   ^ true
```

6 Konzepte zur Entwicklung einer visuellen Sprache zur Beschreibung und Anwendung von Softwareentwurfsmustern am Beispiel des Observers in Squeak/Smalltalk

Entwurfsmuster (engl. „design patterns“) werden oft beim Schreiben von Software verwendet, um bereits bekannte Probleme, die immer wieder vorkommen, zu lösen. Allerdings leiden sie unter sich wiederholenden Implementierung und einer Verteilung über den Quelltext, was die Wartbarkeit der Software beeinträchtigt.

In diesem Kapitel wird konzeptionell eine visuelle Programmiersprache, am Beispiel des Observers (dt. Beobachter*), ausgearbeitet, die es ermöglicht auf Klassenebene zu programmieren und somit Muster beschreiben kann. Die Ideen werden anhand einer Referenzimplementierung in Squeak/Smalltalk mittels SandBlocksevaluiert und zeigen wie Entwickler beim Implementieren von Entwurfsmustern mit Hilfe dieser Sprache profitieren.*

6.1 Einleitung

Softwarearchitektur ist überall. Wenn Entwickler neue Softwaresysteme entwerfen müssen sie sich fundamentalen Entscheidungen stellen wie diese gestaltet werden sollen. Um diesen Prozess einfacher zu gestalten gibt es Entwurfsmuster (engl. „design patterns“): Lösungen zu Problemen, die oft eintreten und in der Praxis erprobt sind.

Das wohl bekannteste Werk über objektorientierte Entwurfsmuster ist wohl: „Design Patterns - Elements of Reusable Object-Oriented Software“ [43], in dem 23 Muster niedergeschrieben wurden. Diese Muster werden von vielen als gegeben hingenommen stellen aber in gewisser Weise ein Paradoxon dar, denn ein Muster ist etwas was immer wieder umgesetzt wird und das obwohl hier Potential bestände dieses einmal zu implementieren und danach wieder zu verwenden.

Der Versuch dies zu bewerkstelligen äußert sich darin, dass in vielen Programmiersprachen einige dieser Muster mitgeliefert werden, aber noch längst nicht alle. Ein Beispiel für ein solches Muster ist das Observer-Pattern (dt. *Beobachtermuster*), welches in Squeak/Smalltalk vorimplementiert ist und in diesem Kapitel genauer betrachtet werden wird.

6.1.1 Forschungsfrage

Die Frage, die im Verlauf dieses Kapitels behandelt werden soll, ist, wie besser mit Entwurfsmustern und insbesondere dem Observer-Muster in Squeak/Smalltalk gearbeitet werden kann.

Als Antwort darauf wird ein Entwurf für eine visuelle Programmiersprache innerhalb von Squeak/Smalltalk mithilfe des SandBlocksFrameworks ausgearbeitet. Mit dieser lässt sich insbesondere das Observer-Pattern ausdrücken und dessen betroffene Klassen werden besser erfassbar gemacht. Die Sprache ist absichtlich generischer gehalten, um im Ausblick deren Potenzial für das Beschreiben von weiteren Mustern zu betrachten.

Eine visuelle Darstellung dieser Zusammenhänge erscheint geeignet, da es üblich ist Softwarearchitektur über Diagramme zu kommunizieren und Abhängigkeiten der Klassen untereinander so besser zu erfassen sind. Ein Vorteil ist auf jeden Fall, dass das Muster dann nicht mehr erst beim Lesen des Quelltextes erschlossen werden muss und es direkt eine Ansicht gibt, welche Klassen wie an welchem Muster beteiligt sind.

Wichtig zu betonen ist, dass die vorgeschlagene Sprache nur bei der Implementierung der Muster und dem Verständnis welche Klasse wie agiert hilft. Es ist kein Ziel Entwurfsmuster an sich verständlicher zu machen oder zu dokumentieren warum ein Muster verwendet wurde. Auch richtet sich die Sprache an Entwickler, die bereits mit solchen Mustern vertraut sind.

Der Mehrwert dieses Kapitels ist, dass sie Entwurfsmuster auf eine visuelle Ebene, getrennt vom normalen Programmtext, hebt. Das Ziel eine Programmiersprache, die das bewältigt zu erstellen, ist sehr groß gesteckt und erfolgt daher nicht vollständig.

6.1.2 Gliederung

Im Verlauf dieses Kapitels werden zunächst in Abschnitt 6.2 Entwurfsmuster und im Besonderen das Observer-Muster genauer betrachtet, um daraufhin in Abschnitt 6.3 Anforderungen für eine visuelle Programmiersprache abzuleiten, die danach in Abschnitt 6.4 betrachtet wird. Die Ergebnisse davon werden in Abschnitt 6.5 ausgewertet, in Abschnitt 6.6 verwandte Arbeiten vorgestellt, die als weitere Inspiration für die Sprache dienen können, und zu guter letzt in Abschnitt 6.7 mit einer kurzen Zusammenfassung des Kapitels geschlossen.

6.2 Das Entwurfsmuster Observer in Squeak

In diesem Abschnitt werden zunächst Entwurfsmuster an sich sowie eine Kritik an diesen erläutert. Danach wird das generelle Observer-Muster und dessen Implementierung in Squeak/Smalltalk beschrieben.

6.2.1 Entwurfsmuster

Eine bekannte Möglichkeit Struktur in objektorientierte Software zu bringen ist es sich an Entwurfsmustern zu bedienen. Die wohl bekannteste Sammlung dieser Muster lässt sich in „Design Patterns: Elemtens of Reuseable Object-oriented Software“ [43] finden. Darin beschrieben sind bekannte Muster, die jeweils definierte Problemstellungen im Entwurf von Softwareteilen besonders gut lösen. Wenn sie dem Programmierer bekannt sind können sie direkt angewendet werden, ohne das Problem noch einmal von neuem lösen zu müssen. Außerdem sind sie auch eine Möglichkeit der Dokumentation, da sie, wenn richtig verwendet, das Vorkommen eines bekannten Problems kommunizieren, zumal sie auch durch ein einheitliches Vokabular den Austausch von Entwicklern untereinander vereinfachen.

In Squeak/Smalltalk steht nicht für jedes Muster eine Implementierung bereit, beziehungsweise werden die Muster nicht durch Features von der Sprache Smalltalk unterstützt. Daher muss das gewünschte Muster oft vom Entwickler selbst realisiert werden. Dies hat zur Folge, dass sich erst einmal auf die Implementierung des Musters konzentriert werden muss, statt an dem eigentlichen, neuen Problem arbeiten zu können.

Zusätzlich dazu wird auch das DRY-Prinzip [6] verletzt: Don’t repeat yourself - Wiederhole dich nicht und neben zusätzlichem Implementierungsaufwand müssen, wenn eine Änderung vorgenommen wird, auch alle Kopien geändert werden. Dies ist gerade auf Architekturebene problematisch, da mitunter an mehreren Stellen Quelltext verändert werden muss. Ist alle Architektur an einer Stelle muss nur diese angefasst werden, wodurch Entwickler nicht in die Gefahr laufen Details zu vergessen.

6.2.2 Das Observer-Pattern

Das Observer-Pattern stellt eine Verbindung von einem zu mehreren anderen Objekten her, die alle benachrichtigt werden, wenn sich das Objekt verändert. Zu dem Muster kennt das Buch folgende Rollen:

Subjekt ist das Objekt, von dem die Verbindungen aus gehen und das andere benachrichtigt. Es kennt seine Beobachter, von denen es eine beliebige Anzahl haben kann.

Beobachter (engl. *observer*) ist dem Subjekt bekannt und wird bei dessen Änderungen von ihm benachrichtigt.

Konkretes Subjekt kennt seine Observer und benachrichtigt diese falls sich sein Zustand verändern sollte.

Konkreter Beobachter ist die Klasse, die implementiert, wie auf eine Änderung des Subjekts reagiert werden soll.

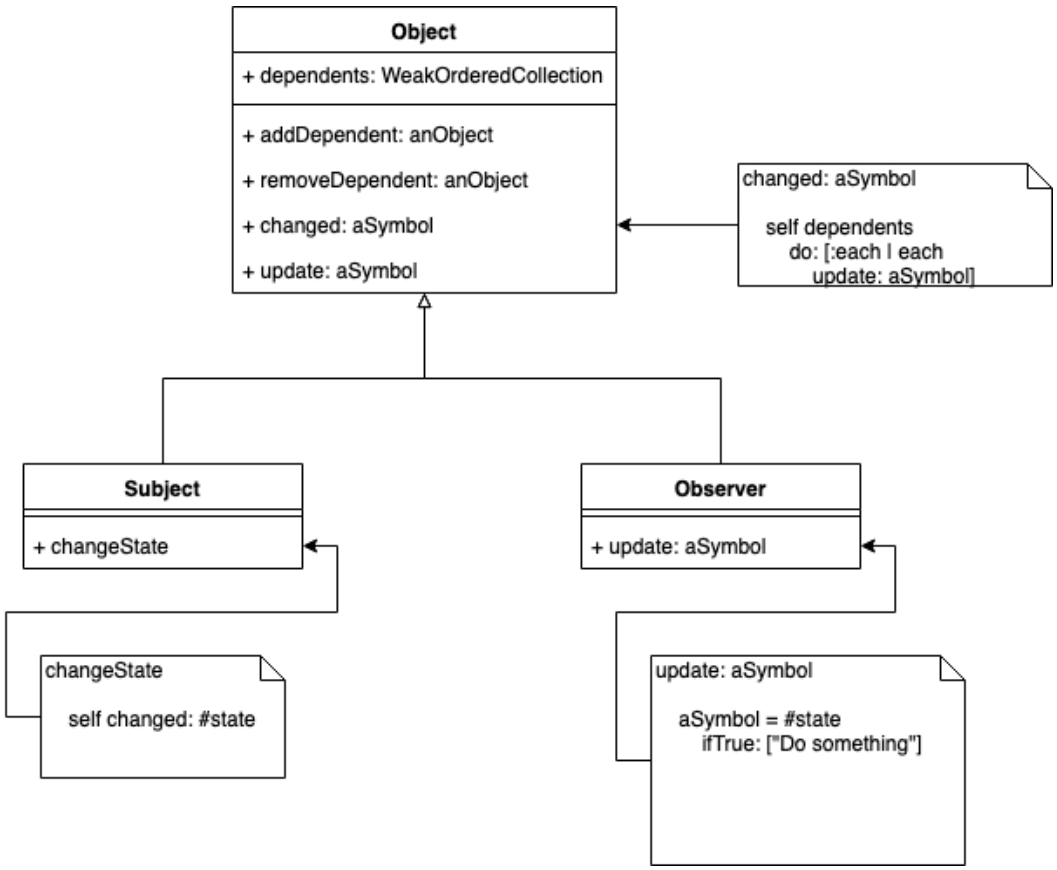


Abbildung 6.1: Das Observer-Pattern wird in Squeak über `dependents`, `changed:` und `update:` umgesetzt. (UML-Klassendiagramm)

6.2.3 Das Observer-Pattern in Squeak

In Abbildung 6.1 ist zu sehen, wie das Observer-Pattern gewöhnlicherweise in Squeak/Smalltalk umgesetzt wird. Auf der Klasse `Object`[1] existieren bereits alle nötigen Schnittstellenmethoden des Subjekts und des Beobachters, sowie die Instanzvariable `dependents`, und da alle Klassen von dieser erben, beinhalten sie auch diese Funktionalität bereits. Instanzen von `Subject` müssen nur noch über `addDependent:` die Instanzen von `Observer` hinzugefügt werden und wenn `changeState` aufgerufen wird, wird in deren `update:` Methode aSymbol gleich #state sein und damit das Objekt eine Aktion ausführen.

Die einzelnen Methoden erfüllen dabei folgenden Zweck:

- `changed:` ruft auf allen angemeldeten Observern die `update:` Methode mit dem spezifizierten Symbol auf.
- `update:` bestimmt je nach Symbol die Reaktion auf Änderungen.
- `addDependent:` fügt dem `dependents`-Array den übergebenen Parameter hinzu.
- `removeDependent:` entfernt den übergebenen Parameter aus dem `dependents`-Array.

Ein Problem, was auftreten kann ist, wenn Entwickler eine Klasse, die sich dieses Musters bedient, zu verstehen versuchen. Liest man solchen Quelltext, stößt auf ein `changed:` und weiß nicht welche Klassen die zugehörigen Observer haben kann das Muster sehr schwer nachzuvollziehen sein.

Eine gute Möglichkeit dies zu verstehen ist, die besagte Klasse korrekt zu instanziieren und deren Observer zu inspizieren. Da es aber auch nicht immer einfach ist das Objekt für sich alleinstehend zu erstellen ist auch dies mitunter sehr müßig und eine andere Art die Beziehung zu definieren sollte zur Verfügung gestellt werden.

Ansonsten bleibe es nur übrig den gesamten Quelltext zu lesen.

6.3 Anforderungen an die visuelle Sprache

In diesem Abschnitt werden Anforderungen an eine visuelle Programmiersprache aufgestellt. Diese leiten sich aus den in Abschnitt 6.2 vorgestellten Herausforderungen und eigenen Überlegungen zur Nutzbarkeit ab.

[1] Alle Klassen in Squeak/Smalltalk haben `Object` als obersten Teil ihrer Vererbungshierarchie.

6.3.1 Visuelle Repräsentation

Das System sollte zu großen Teilen auf visueller Repräsentation beruhen. Damit kann die Architektur ähnlich zu Diagrammen, die vielen Entwicklern schon bekannt sind[2], gelesen und verstanden werden.

6.3.2 Zentrale Platzierung der Beschreibung

Das Sprache soll an zentralen Stellen eine Darstellung einführen, die eine Übersicht über das Zusammenspiel von Klassen und deren Interaktion untereinander einführen. Die Positionierung erfolgt in dieser Art, um generell Übersicht über den Quelltext zu gewähren, die ansonsten nicht in so gegeben ist. Dies hilft zum Beispiel jemandem, der die Codebasis noch nicht kennt und sich einen ersten Einblick von der Struktur machen möchte, da er immer an einer zentralen Stelle nachlesen kann und nicht einzelne Stellen im Programmcode lesen muss und das System somit wie eine Dokumentation verwendet werden kann.

6.3.3 Einfluss auf beschriebene Teile des Squeak/Smalltalk Systems

Die Sprache sollte ausführbar sein, bzw. einen direkten Einfluss auf die Klassen im Squeak/Smalltalk System, die es beschreibt, haben. Dadurch gibt es keine Redundanz durch zusätzliche Dokumentation, die nicht mit dem tatsächlichen System übereinstimmt, was häufig der Fall ist [69]. Stattdessen ist die Dokumentation gleichzeitig ausführbar und kann, da sie damit das System mitbestimmt, nicht von der Implementierung abweichen.

6.3.4 Wiederverwendbarkeit

Das System sollte eine Form von Wiederverwendbarkeit besitzen. Dies beginnt damit, dass eine Beschreibung für zum Beispiel ein Muster nicht nur einmal, sondern mehrmals verwendet werden kann, aber auch bereits bestehende Beschreibungen weiter spezifiziert werden können. Dadurch wird das DRY-Prinzip nicht mehr verletzt und die Software somit besser wartbar, da Änderungen nur an einer Stelle vorgenommen werden müssen statt an vielen, die vergessen werden können. Außerdem kann Software dadurch, dass auf bereits bestehende Lösungen zurückgegriffen wird, schneller Software entwickelt werden [67].

6.3.5 Freie Platzierung der visuellen Elemente

Eine Anforderung an die Benutzeroberfläche ist, dass die verwendeten grafischen Elemente beliebig angeordnet werden können. Dadurch bleibt die sekundäre Notation [83] erhalten, also die Möglichkeit durch das Layout dem Leser die Zusam-

[2] Die Struktur von Mustern wird in [43] anhand von Klassendiagrammen erklärt.

menhänge zu suggerieren. So können Entwickler zum Beispiel wichtige Elemente weiter oben platzieren oder Elemente, die orthogonal zueinanderstehen, nebeneinander. Dies kann den Einstieg in das Verstehen der Struktur vereinfachen, wenn die Entwickler eine gewisse Disziplin an den Tag legen und die Elemente sinnvoll anordnen.

6.3.6 Direkte Manipulation

Für ein besseres Arbeiten mit dem System sollte es direkte Manipulation [99] unterstützen, das heißt es sollte im Besonderen die folgenden Eigenschaften besitzen:

1. Durchgehend die relevanten Objekte anzeigen. Dadurch kann ein Entwickler die Beschreibung besser erfassen und sie kann somit besser als Dokumentation dienen.
2. Direkte Interaktionsmöglichkeiten, zum Beispiel über die Maus. Dadurch wird die Lernkurve für die Benutzung des Systems weniger steil.
3. Änderungen auf Objekten, die sofort Wirkung zeigen und reversibel sind. Dadurch kann ein Entwickler kleinere Änderungen ausprobieren und rückgängig machen, falls sie nicht seinen Vorstellungen entsprachen.
4. Nutzer benötigen nur eine kleine Menge an Wissen über die Kommandos des Systems, um sinnvoll damit arbeiten zu können und, nachdem diese verinnerlicht wurden, weitere zu lernen. Dies ermöglicht es direkt, ohne lange Einarbeitungszeit, mit dem System zu arbeiten.

6.4 Definition der visuellen Sprache

In diesem Abschnitt wird zunächst in Abschnitt 6.4.1 eine Programmiersprache konzeptionell beschrieben und danach in Abschnitt 6.4.2 mithilfe einer Implementierung, die sich daran orientiert, das Observer-Pattern implementiert. Zu guter Letzt werden in Abschnitt 6.4.3 die Komponenten des Systems und dessen wichtigste Designentscheidungen vorgestellt.

6.4.1 Sprache

In diesem Abschnitt werden Konzepte einer visuellen Sprache beschrieben, die Entwurfsmuster und von diesen vornehmlich das Observer-Muster ausdrücken kann. Dafür müssen mindestens Klassen samt Rollen aufgeführt, Klassen Instanzvariablen gegeben und Methoden geschrieben werden können.

6.4.1.1 Elemente der Sprache

Die Elemente dieser Sprache sollen ähnlich zu Struktur und Interpretation von Computerprogrammen [1] beschrieben werden. Die darin identifizierten Elemente

sind *primitive Ausdrücke*, welche die einfachsten Einhaten der Sprache darstellen, *Mittel zur Kombination*, welche primitive Ausdrücke zu mächtigeren verbinden, und *Mittel zur Abstraktion*, welche meist diese Kombinationen benennen und damit von den Teilen abstrahieren.

Primitive Ausdrücke Primitiven in dieser Sprache sind Daten. Beispiele dafür sind Klassen und Methoden mit ihren jeweiligen Informationen. Das sind zum Beispiel für ein Klassenelement welche Klasse aus Squeak/Smalltalk referenziert wird und welche Rolle sie einnimmt.

Mittel zur Kombination Kombinationsmöglichkeiten sind gerichtete Verbindungen zwischen Primitiven. Sie sagen aus, dass Primitive X eine Aktion mit Primitive Y ausführt, wobei die Verbindung die Aktion ist.

Die Aussagen, die bisher getroffen werden konnten, waren, es gibt ein Primitive X und eine Primitive Y. Mit den Verbindungen kann ausgedrückt werden, dass X etwas mit Y macht, zum Beispiel ist X eine Klasse und hat Y als Methode.

Mittel zur Abstraktion Die Abstraktion wird ermöglicht durch die Unterscheidung zwischen den so genannten abstrakten und konkreten Beschreibungen. Eine abstrakte Beschreibung ähnelt einer Klasse. Sie definiert die Struktur und wie die Elemente dieser untereinander zusammenhängen, aber nicht für welche konkreten Werte dies gilt und wird von einer konkreten Beschreibung aus referenziert.

Eine konkrete Beschreibung dagegen verlangt, dass in der entsprechend vorher definierten, abstrakten Beschreibung definierte Werte ausgefüllt werden. So wird hier zum Beispiel eingetragen, welche Klasse dort auf bestimmte Art mit einer anderen interagieren soll.

Der Vorteil dieses Mechanismus ist die Wiederverwendbarkeit. Jetzt kann eine Klassenstruktur definiert werden und in einzelnen Projekten referenziert werden, ohne auf Ebene des Programmcodes nochmal neu definiert werden zu müssen. Dies zieht allerdings einen fundamentalen Unterschied zwischen abstrakten und konkreten Beschreibungen nach sich: eine abstrakte Beschreibung definiert wie die Struktur aussieht und die konkrete Sicht beschreibt wer welche Komponente dieser Struktur mit welchem Verhalten ist.

6.4.1.2 Spezialisierung

Dadurch, dass eine abstrakte Beschreibung mehrmals instanziiert werden kann, ist es nicht erforderlich jedes Mal, wenn ein bestimmtes Muster verwendet werden soll, dieses von Grund auf zu bauen. Man nimmt die Schablone und füllt sie aus. Allerdings existiert weder die Möglichkeit bereits definierte Muster genauer zu spezifizieren noch abstrakte Beschreibungen aus anderen zu definieren. Da diese Arten der Wiederverwendbarkeit das Entwickeln von Software erleichtern, sind sie wünschenswert [67].

Fraglich ist nur noch welche Art von Wiederwerwendbarkeit eingeführt werden soll. Dafür ergeben sich folgende Möglichkeiten:

1. Man nutzt Vererbung wie aus Squeak/Smalltalk bekannt und beschreibt die zusätzlichen Elemente und hat dann, ähnlich zu einem super-Aufruf[3] in Smalltalk, ein Element, das die vorherige Struktur repräsentiert.

2. Man nutzt einen Prototyp-basierten Ansatz und kopiert immer die gesamte Beschreibung. Diese wird aktualisiert, wenn sich die ursprüngliche Beschreibung verändert.

3. Es gibt keine Vererbung und Wiederverwendbarkeit wird durch Kopieren erreicht.

Sollte eine dieser Möglichkeiten bevorzugt werden?

Hat man Vererbung wie in Squeak/Smalltalk mit einem super-Aufruf ist es nicht möglich Verbindungen zu den einzelnen Komponenten der Superbeschreibung genauer zu spezifizieren. Allerdings ist es dann möglich Elemente, die innerhalb der Beschreibung genauer beschrieben wurden zu überschreiben. Dies wäre also nützlich, wenn nichts an der visuellen Beschreibung geändert werden soll, sondern nur an Daten.

Der prototypenbasierte Ansatz begünstigt die Flexibilität, gerade im Hinblick auf sekundäre Notation, die noch weitere Informationen suggerieren kann. Hier können neue Verbindungen an vorhandene Komponenten angebunden werden, verlieren allerdings die Möglichkeit die ursprüngliche Beschreibung zu verändern, wie es bei der Vererbung in Squeak/Smalltalk der Fall ist. Dies hat allerdings den Vorteil, dass bei Änderungen der Super-Beschreibung Veränderungen aktualisiert werden. Dies wirft aber noch die Frage auf, wie dies synchronisiert werden soll, da es Verbindungen zu Elementen geben kann, die gelöscht werden können.

Wenn nur kopiert wird liegt den Mehraufwand des Kopierens und das Verletzen des DRY-Prinzips vor. Dafür wird aber die völlige Freiheit bei den Änderungen gewonnen.

Die drei Varianten widersprechen sich nicht und es sollten dem entsprechend alle drei Varianten angeboten werden, wobei die letzte Möglichkeit immer besteht und keine besonderen Maßnahmen ergriffen werden müssen. Für die erste Variante muss eine Möglichkeit der Vererbung und des super-Aufrufs zur Verfügung gestellt werden und für die zweite eine Möglichkeit die visuelle Darstellung einer Beschreibung einzubinden und diese mit dem Original synchron zu halten. Welche der Möglichkeiten bevorzugt werden sollte hängt vom jeweiligen Anwendungsfall ab und liegt im Ermessen der Entwickler.

6.4.2 Das Observer-Pattern als Referenzimplementierung

Das Ziel dieses Abschnittes ist mithilfe einer Referenzimplementierung der konzeptionell in Abschnitt 6.4.1 beschriebenen visuellen Programmiersprache das Observer-Pattern aus Squeak/Smalltalk erneut zu implementieren.

[3] Ein Aufruf mittels `super` verwendet die Methode, die in der Vererbungshierarchie am nächsten über der Klasse mit dem Aufruf steht.

Dafür wird zunächst beschrieben, was das Beispiel leisten können soll und dann wie es mithilfe des Referenzsystems umgesetzt wird.

6.4.2.1 Zielsetzung

Es soll ein Beispiel des Squeak/Smalltalk Observer-Patterns von Grund auf programmiert werden. In diesem soll es eine Klasse `Subject`, die die Rolle des Subjekts einnimmt, sowie die Klassen `ConcreteObserver1` und `ConcreteObserver2`, die jeweils die Rolle des konkreten Beobachters einnehmen, geben, die jeweils unterschiedlich auf ein Symbol reagieren sollen.

6.4.2.2 Umsetzung

Um das gesetzte Ziel umzusetzen wird zunächst eine abstrakte Beschreibung des Observer-Patterns angelegt werden und anschließend für die einzelnen Observer je eine konkrete Beschreibung, wie diese auf den Parameter der `changed:` Methode reagieren soll.

Zunächst muss dafür eine Subklasse von `AbstractDescription`, die `ObserverPattern` heißt, angelegt werden. Diese wird in die Kategorie Architektur eingeordnet. In dieser Klasse wird dann eine Methode `description` angelegt, die als Blockmethode vom Typ `Abstract` agiert.

Nun kann sich auf das Beschreiben des Observer-Patterns konzentriert werden. Dafür werden zwei Klassenelemente angelegt, die die Rollen des Subjekts und des Beobachters übernehmen. Diese sind durch eine Verbindung des Typen `hasAsInstanceVariable` verbunden. Dies führt dazu, dass in `Subject` später eine Instanzvariable mit dem Namen Observer ergänzt wird. Dann legt man Methodenelemente für die Methoden `attach:` [4], `detach:` [5], `changed:` und `update:` an und verbindet diese über eine `hasMethod` Verbindung mit dem jeweiligen Klassenelement. Das Ergebnis ist in Abbildung 6.2 zu sehen.

Die bisher erstellte abstrakte Beschreibung verlangt es vom Entwickler in jeder konkreten Beschreibung dazu jedes Mal die einzelnen Methoden neu zu definieren, obwohl zumindest `attach:`, `detach:` und `changed:` sich wohl kaum jemals verändern dürften. Um dem zu entgehen können diese Methoden in der abstrakten Beschreibung über das Kontextmenu mit 'make concrete' zu konkreten Elementen gemacht werden, wodurch einmaliges Definieren ausreichend ist. Danach sieht die Beschreibung wie in Abbildung Abbildung 6.3 aus.

Nachdem die abstrakte Struktur steht können die benötigten Klassen angelegt werden. In diesem Fall sind das die Klassen `Subject`, `ConcreteObserver1`, `ConcreteObserver2`, die von `Object` erben. Weiter wird bis jetzt nichts auf diesen Klassen implementiert.

Auf diese Klassen soll nun das Observer-Pattern angewendet werden. Dafür müssen wir die vorher definierte abstrakte Beschreibung auf diese Klassen anwenden. Dies geschieht, indem eine konkrete Beschreibung erstellt wird, in der die

[4] Dies entspricht der Methode `addDependent:` in Squeak.

[5] Dies entspricht der Methode `removeDependent:` in Squeak.

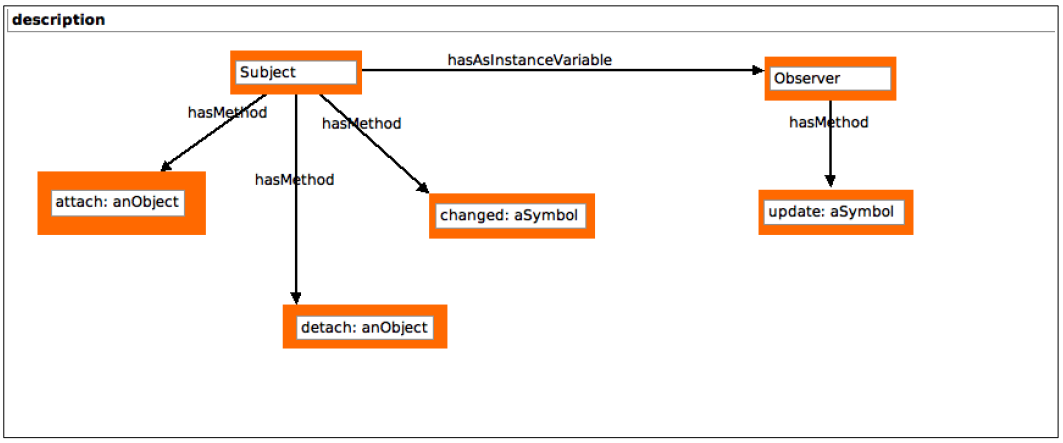


Abbildung 6.2: In der abstrakten Darstellung des Observer-Patterns werden keine konkreten Methodenlemente gezeigt.

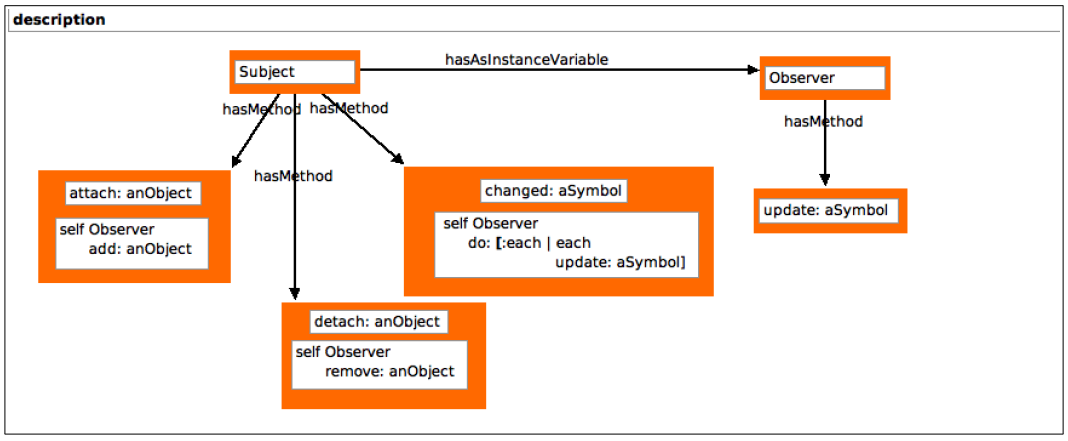


Abbildung 6.3: Wenn einige abstrakte Elemente in konkrete Methoden umgewandelt wurden, wird die Flexibilität dieses Modells deutlich.

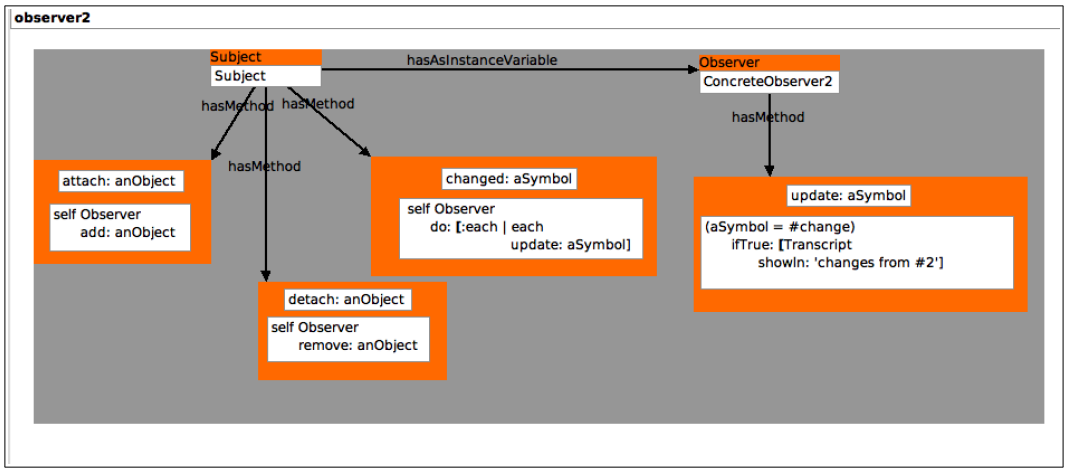


Abbildung 6.4: Die konkrete Beschreibung für `ConcreteObserver2` zeigt alle Methoden des Entwurfsmusters.

`ObserverPattern` Beschreibung referenziert und die Werte der Elemente eingetragen werden, welche bisher nur abstrakt waren.

Dafür wird eine neue Klasse, die von `ConcreteDescription` erbt, erstellt. Diese erhält die beiden konkreten Blockmethoden `observer1` und `observer2`. In diesen wird jeweils zu Subject Subject als Klasse eingetragen, Observer respektiv `ConcreteObserver1` und `ConcreteObserver2` und die `update:` Methoden reagieren beide auf das Symbol `#change` mit dem Schreiben der Nachricht ‚changes from #NUMMER', wobei NUMMER für `ConcreteObserver1` eins ist und für den anderen zwei.

Das Ergebnis ist in Abbildung 6.4 zu sehen. Der graue Kasten signalisiert, dass dies eine referenzierte Beschreibung ist. In diesem Fall die abstrakte Beschreibung Observer-Pattern. Die Folge daraus ist, dass in die Klassen die beschriebenen Methoden hineinkompiliert werden. Allerdings ist die Implementierung damit noch nicht komplett, da die Klasse `Subject` zwar eine Variable mit dem Namen Observer samt Zugriffsmethoden bekam, diese aber noch nicht initialisiert ist. Es muss also noch manuell nachgeholfen werden und dieser Wert zum Beispiel in der `initialize`[6] Methode auf eine Instanz von `OrderedCollection`[7] gesetzt werden. Ab jetzt kann der Observer benutzt werden (s. Listing 6.1).

[6]Beim Erzeugen einer Instanz einer Klasse wird in Squeak automatisch die Methode `initialize` aufgerufen.

[7]Eine dynamisch wachsende Liste mit einfacher Ordnung mittels `OrderedCollection` wird in Squeak üblicherweise bei Multiplizitäten > 1 gewählt.

Listing 6.1: Beispielhafte Verbindung von zwei Observern mit einem Subjekt

```
a := Subject new.
b := ConcreteObserver1 new.
c := ConcreteObserver2 new.

a
   attach: b;
   attach: c.

a changed: #change.
```

Den Variablen a, b, c wird jeweils eine neue Instanz des Objektes, das in dem Ausdruck steht, zugewiesen. Danach werden a die Beobachter b und c hinzugefügt. Wird anschließend `changed:` mit `#change` als Parameter aufgerufen, woraufhin der spezifizierte Text ausgegeben wird. Daran ist abzulesen, dass die Implementierung des Observer-Patterns mithilfe der Beschreibung erfolgreich war.

Listing 6.2: Textuelle Ausgabe des Beispiels

```
changes from #1
changes from #2
```

6.4.3 Implementierungsskizze

In diesem Abschnitt wird der Aufbau und wichtige Implementierungsentscheidungen des in Abschnitt 6.4.2 genutzten Systems vorgestellt. Die Klassenstruktur ist in der Abbildung 6.5 visualisiert.

Zur Implementierung wurde das SandBlocksFramework verwendet. Dieses ermöglicht es beliebige Morphs[8] in Text zu integrieren, wobei diese interaktiv bleiben. Diese Morphs werden Blöcke genannt.

Die Integration kann auf zwei Arten geschehen. Einmal kann ein Block direkt im Text platziert werden. Dieser hat dann zur Ausführungszeit ein vordefiniertes Verhalten. Die andere Variante ist es eine gesamte Methode als Block zu behandeln. Diese kann dann auf grafische Art und Weise programmiert werden, wie auch immer derjenige, der den Block geschrieben hat, das definiert hat.

Um einen Morph zu einem Block zu machen, muss dieser lediglich den Trait `TBTRBlockModel` umsetzen. Daraufhin kümmert sich das SandBlocks-Framework vom Ermöglichen des Zugriffs auf den Block bis hin zum Ausführen um alles weitere, wie in Kapitel 2 ausführlich beschrieben.

6.4.3.1 `StructureDescription` und Subklassen

Das Herzstück der Implementierung ist die `StructureDescription`. Sie ist der Editor, der am Ende im System Browser angezeigt wird und mit der der Nutzer interagiert. Da er ein Block ist muss er den `TBTRBlockModel` Trait umsetzen und von einer

[8] Morphs sind visuelle Elemente des Morphic Frameworks, das in Squeak/Smalltalk standardmäßig die Benutzeroberfläche stellt. Genaueres zu Morphic findet sich unter [75].

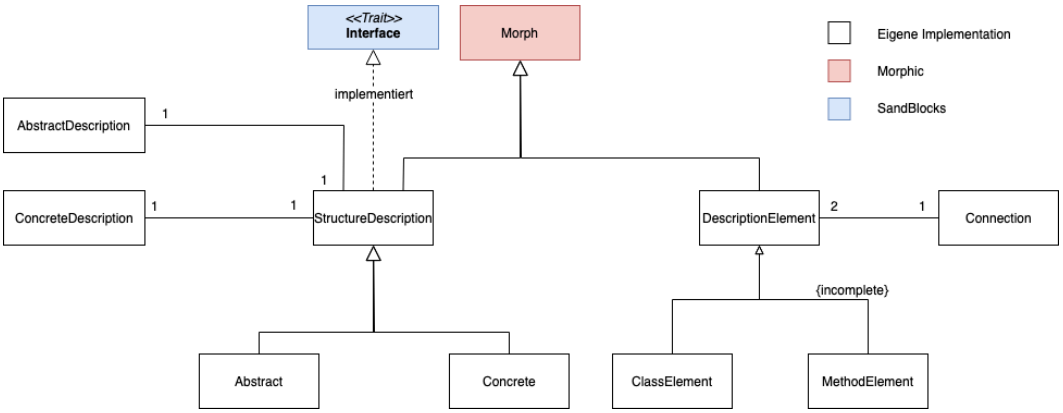


Abbildung 6.5: Architektur des Systems (UML-Klassendiagramm)

Subklasse von Morph erben. In diesem konkreten Fall erbt sie sogar direkt von Morph. Die Klasse `StructureDescription` an sich ist eine abstrakte Klasse deren konkretes Verhalten von `Abstract` und `Concrete` definiert wird. Diese sind jeweils der Editor für abstrakte oder konkrete Beschreibungen.

Sie unterscheiden sich jeweils in den Interaktionsmöglichkeiten. Die Haupmöglichkeit dafür ist das Kontextmenü, welches sich auf Rechtsklick öffnet. Zum einen können die beiden Klassen nach einem gemeinsamen Teil bestimmen welche spezifischen Menüpunkte angezeigt werden sollen, aber auch wie auf die einzelnen Punkte reagiert werden soll, da die unterschiedlichen Sichten unterschiedliche Elemente erfordern, wie in Abschnitt 6.4.3.2 erklärt wird.

Da eine `StructureDescription` als Methode agiert, stellt sich die Frage zu welchem Wert sie evaluiert. Hat man es mit einem `Abstract` zu tun, so evaluiert es zu einem `Concrete`, welches statt abstrakter konkrete Elemente beinhaltet. Dies wird benötigt, wenn die abstrakte Beschreibung von einer konkreten aus referenziert wird.

Ein `Concrete` dagegen evaluiert aktuell zu keinem Wert, beziehungsweise führt keine Aktionen durch. Im Abschnitt 6.4.3.5 wird erklärt, wie die Informationen auf die beschriebenen Klassen angewendet werden.

6.4.3.2 Elemente

`DescriptionElement` erbt von Morph und ist die Oberklasse für alle visuellen Elemente, die in einer Beschreibung verwendet werden können. Diese definiert vor allem das Drag-and-Drop Verhalten der Elemente, aber auch das Kontextmenü.

Die Subklassen haben eine interessante Eigenschaft: es muss sie sowohl in einer konkreten als auch abstrakten Variante geben, da in den unterschiedlichen Modi unterschiedliche Sachverhalte definiert werden. Während in der abstrakten Sicht deklariert wird, dass genauere Spezifikation benötigt wird, werden in der konkreten Sicht genau diese gegeben.

Dies wurde nicht über Vererbung gelöst. Wäre dieser Weg gewählt worden, würde für fast jede Art neues Element, das neu erstellt werden soll, zwei Klassen angelegt werden müssen, die sich kaum voneinander unterscheiden würden. Aber

gerade da sie sich so ähnlich sind, kann auch das Verhalten beider in eine Klasse gekapselt werden. Bei Bedarf muss der aktuelle Typ in den jeweils anderen Typ änderbar sein, da dies spätestens nötig ist, wenn eine konkrete Beschreibung zu einer abstrakten Beschreibung erstellt werden soll.

6.4.3.3 `Connection`

`Connection` ist zum einen für das Zeichnen der Linie zwischen zwei Elementen und die Positionierung der Annotation zuständig. Sie ist aber auch für die Ausführung der mit der Verbindung assoziierten Aktion zuständig, da sie beide Elemente kennt und somit über alle Daten verfügt, die dafür nötig sind. Dafür besitzt sie eine Instanzvariable, welche den Methodennamen der dafür auszuführenden Methoden beinhaltet, die schließlich mit `perform:` [9] ausgeführt wird.

6.4.3.4 `AbstractDescription` und `ConcreteDescription`

Die beiden Klassen `AbstractDescription` und `ConcreteDescription` sind leere Klassen, das heißt sie implementieren weder Methode noch besitzen sie Instanzvariablen. Sie sind jedoch die Klassen, von denen alle Klassen erben müssen, wenn sie eine Methode zur Beschreibung bereitstellen möchten. Dies hängt zum einen mit Konventionen zusammen: wenn zu sehen ist, dass von einer dieser beiden Klassen geerbt wurde, wird sofort vermittelt was diese macht. Aber auch wird dies zur Implementierung genutzt. In konkreten Beschreibungen können abstrakte Beschreibungen instanziiert werden. Dafür müssen alle möglichen abstrakten Beschreibungen gefunden werden. Dies kann ganz einfach erfolgen, indem sich die Reflektionsmöglichkeiten von Squeak/Smalltalk zunutze gemacht werden und alle Subklassen von `AbstractDescription` durchsucht.

Die zweite Konvention, die an dieser Stelle greift ist, dass die Methode, die die abstrakte Beschreibung darstellt, `description` heißt, damit das System weiß welche Methode die Beschreibung darstellt. Alternativ hätten auch alle Methoden der Klasse darauf untersuchen werden können ob sie von `AbstractDescription` erben, aber das könnte zu Problemen führen, wenn es mehrere gäbe. Ein weiterer Vorteil der Konvention ist auch, dass, gegeben dem Fall es gibt mehr als nur eine Methode in der Klasse, jemand, der die Beschreibung lesen will, diese nicht erst suchen muss und direkt auf `description` zugreifen kann.

Das Erben von konkreten Beschreibungen dient bisher nur der Konvention. Keine Komponente der Implementierung verlässt sich darauf. Das heißt es könnte prinzipiell auch von anderen Klassen geerbt werden, aber im Sinne der Konvention wird davon abgeraten.

6.4.3.5 Brücke Squeak/Smalltalk

Damit die Beschreibung auch Einfluss auf die Klassenstruktur haben kann, muss sie mit diesen manipuliert werden können. Eine Möglichkeit ist das Nutzen der

[9] Die Methode `perform:` nimmt ein Symbol entgegen, woraufhin der normale Nachrichtenversand von Smalltalk angestoßen wird. Sie ist Teil von Squeak's Meta-Objekt-Protokoll.

Zeit des Parsens als Variationspunkt[10]. Wenn der Block kompiliert wird, was jedes Mal passiert wenn die Blockmethode gespeichert wird, kann die Beschreibung versuchen ihre Daten anzuwenden. Die einfachste Lösung ist das Hineinkompilieren von Methoden in andere Klassen. Dies erfordert, dass immer eine Klasse dazu angegeben ist.

Eine Alternative wäre es immer umschließende Klassen für alle Klassen, auf die Einfluss genommen wird, anzulegen, und dann im Code nur über visuelle Elemente auf die Klassen aus der Beschreibung zuzugreifen, die dann die umschließenden Klassen zurückgeben. Dies hätte den Vorteil, dass die eigentliche Klasse nicht verändert wird und dort auf einmal Änderungen aus der Beschreibung auftauchen, die man eventuell überschreibt oder ähnliche Problem damit hat. Allerdings könnte es subtile Fehler nach sich ziehen immer nur mit Ersatzklassen zu arbeiten, wenn man zum Beispiel an die Überprüfung der Klasse denkt, weswegen die erste Variante gewählt wurde.

6.4.3.6 Erweiterung um neue Elemente und Verbindungen

In der bestehenden Architektur wird das Erstellen neuer Elemente durch Erben von `DescriptionElement` erreicht. Dann muss für dieses neu erstellte Element definiert werden, wie dessen konkrete, abstrakte Ansicht aussieht und wie die Konvertierung von einer Variante in die andere vonstattengeht. Anschließend muss in der `StructureDescription` ein Menüeintrag zum Einfügen dieses Elements erzeugt werden, damit dieses auch verwendet werden kann.

Bei Verbindungen müssen neue Methoden in der Kategorie `types` angelegt und diese dann entsprechend der gewünschten Funktionalität implementiert werden. Dies genügt schon, da alle Methoden der Kategorie `types` als Verbindungstyp automatisch zur Verfügung gestellt werden.

Beiden Erweiterungsvarianten ist gemein, dass sie einen Umweg über Squeak/Smalltalk erfordern, was daraus folgt, dass neue Primitiven erstellt werden und diese direkt mit Squeak/Smalltalk interagieren müssen, wozu ansonsten in dieser Form keine Möglichkeit existiert.

6.4.3.7 Zentrale Stelle

Eine Grundidee der Beschreibungen ist, dass sie Überblick schaffen und ein erster Anlaufpunkt sind, von dem aus der Programmcode weiter exploriert werden kann. Dies verlangt allerdings, dass dem Entwickler klar ist an welcher Stelle im Code er diese finden kann. Deswegen ist ein Vorschlag Beschreibungen an wohl definierten Stellen, einer Konvention folgend, zu platzieren.

Eine Möglichkeit dafür ist es sich zunutze zu machen, wie Squeak/Smalltalk Klassen speichert. Diese sind in benannten Kategorien organisiert. Wenn mehrere dieser Kategorien eine Einheit bilden sind sie gewöhnlicherweise unter dem Schema: ‚Oberbegriff der Kategorie'-‚genauere Spezifikation' organisiert. So teilt sich zum Beispiel der Compiler mit all seinen Klassen in folgende Kategorien auf:

[10] Genauere Informationen zum Kompilieren und zu Variationspunkten während dieses Vorgangs sind unter Kapitel 4 zugänglich.

Compiler-Exceptions, Compiler-Kernel, Compiler-ParseNodes, Compiler-Support, Compiler-Syntax.

Diese Benennung kann sich zunutze gemacht werden, indem definiert wird, dass unter ‚Oberbegriff der Kategorie'-Architektur alle lokalen und unter Architektur alle globalen Beschreibungen liegen.

6.5 Evaluation

6.5.1 Anforderungen

Wie in Abschnitt 6.3 vorgestellt, werden im Folgenden die Anforderungen diskutiert.

6.5.1.1 Visuelle Repräsentation

Das System ist für die Beschreibung von Strukturen völlig visuell und muss nur für Eingaben, wie Methoden, auf die textuelle Ebene zugreifen. Die Beschreibungen weisen eine gewisse Ähnlichkeit mit Klassendiagrammen auf, was aber gut ist, da viele Entwickler damit vertraut sein dürften. Die genaue visuelle Repräsentation muss noch ausgearbeitet werden. Neben subjektiv schlechtem Aussehen weist sie bei manchen Farbkonfigurationen der Squeak/Smalltalk Umgebung schlechte Lesbarkeit von Text auf.

6.5.1.2 Zentrale Stelle der Beschreibung

Diese Anforderung wird durch eine Konvention gelöst. Dies hat zur Folge, dass diese für Entwickler, die das System noch nicht kennen, erst einmal erklärt werden muss, zumal die Beschreibungen vom Squeak/Smalltalk System her als Klassen vorliegen und dies nicht direkt ersichtlich ist. Auch sind Entwickler keineswegs gezwungen, diese Konvention einzuhalten wodurch es zu Inkonsistenzen zwischen Entwicklergruppen kommen kann. Wenn sich allerdings an die Konvention gehalten wird und diese bekannt ist, ist der Einstiegspunkt selbsterklärend.

6.5.1.3 Einfluss auf beschriebene Teile des Squeak/Smalltalk Systems

Das System hat die Möglichkeit durch Hineinkompilieren von Methoden und Anlegen von Instanzvariablen direkten Einfluss auf Squeak direkte Veränderungen vorzunehmen. Nachteilig ist, dass dabei die Änderungen der Beschreibungen nicht verwaltet werden und zum Beispiel nicht rückgängig gemacht werden können.

6.5.1.4 Wiederverwendbarkeit

Durch die Aufteilung in abstrakte und konkrete Beschreibungen in Kombination mit der Möglichkeit abstrakte von konkreten Beschreibungen aus zu referenzieren wurde eine Grundmöglichkeit für Wiederverwendbarkeit geschaffen, da es möglich ist die abstrakte Beschreibung beliebig oft anzuwenden. Es ist allerdings noch nicht möglich diese Beschreibungen weiter zu spezialisieren, wie es in Abschnitt 6.4.1.2

beschrieben wurde. Es ist aufgrund der aktuellen Implementierung nicht einmal möglich Beschreibungen zu kopieren.

6.5.1.5 Freie Platzierung der visuellen Elemente

Die freie Platzierung von Elementen ist möglich, allerdings stellt in der aktuellen Umsetzung die Größe des Beschreibungsfensters ein Problem dar. Dieses kann zwar manuell größer gemacht werden, füllt aber irgendwann den gesamten Bildschirm aus und da es keine interne Verwaltung für Bildschirmverschiebung gibt, ist damit der maximale Ausbreitungsgrad beschränkt, was gerade bei vielen Elementen ein Problem mit der Anordnung nach sich zieht. Deswegen sollte ein solches System eingeführt werden. Auch sollte eine Unterstützung zur Positionierung von Elementen zur Verfügung gestellt werden, wie zum Beispiel ein Raster, an dem die Elemente ausgerichtet werden, da sich oft darum bemüht wird Elemente ordentlich anzuordnen und dies damit leichter fiele.

6.5.1.6 Direkte Manipulation

Von den vier Teilaspekten sind bisher nur der erste, die kontinuierliche Darstellung mit der visuellen Repräsentation und direkte Interaktionmöglichkeiten, wie Eingabe mit der Maus, Drag-and-Drop und Texteingabe über die Tastatur, umgesetzt. Die Änderungen an den Objekten sind direkt sichtbar, aber noch sind die Änderungen nicht reversibel. Es gibt zwar zu einigen Operationen, wie dem Umwandeln eines abstrakten in ein konkretes Element, Umkehropererationen, dabei gehen aber mitunter Daten verloren und es ist nicht mit einem wirklichen Rückgängig machen zu vergleichen. Um nur einen kleinen Teil des Systems zu kennen und damit sinnvoll arbeiten zu können ist es noch zu unfertig. Es ist nötig das gesamte System bis hin zur Implementierung zu kennen, um viel mehr als das Observer-Pattern umzusetzen, da es nur wenige Teile hat und dem entsprechend immer erweitert werden muss.

6.5.2 Vorteile für Entwickler

Die Sprache, wie sie bisher ist, eignet sich gut dazu Klassen aufgrund einfacher Informationen zu erweitern. Dies umfasst Funktionalitäten, wie das Erstellen von Methoden oder das Anlegen von Instanzvariablen. Dadurch eignet es sich hervorragend Zusammenhänge von der Art des Observer-Patterns zu beschreiben.

Sie bringt die Vorteile, dass zentrale Dokumentation von Mustern unter deren bekannten Namen ermöglicht wird, indem eine Beschreibung mit der passenden Benennung angelegt wird. Aus dieser ist direkt ersichtlich auf welche Klassen welche Muster angewendet werden, was beim Verständnis der Software hilft, gerade wenn diese noch nicht bekannt ist.

Konkret, für das Observer-Pattern, kann das Ablesen der Reaktion einer Klasse auf ein `change:` vereinfacht werden, wodurch nicht mehr der Quelltext danach durchsucht werden muss, wo dem Subjekt welche Beobachter hinzugefügt wurden und wie diese reagieren. Stattdessen kann das an einer zentralen Stelle nachgelesen werden.

Wenn man weiterdenkt eignet sich die Sprache auch für die Beschreibung von Schnittstellen, da diese nichts anderes als eine Ansammlung von Funktionalität sind, die von einer Klasse umgesetzt werden müssen. Wenn Methoden in eine Klasse hineinkompiliert werden erfüllen sie genau diesen Zweck. Im besten Fall schreibt man in auf diese Art erzeugte Methoden ‚self shouldBeImplemented'[11], da die Methoden beim Anwenden durch die konkrete Ansicht alle erst einmal nur erzeugt werden und damit das Implementieren nicht vergessen wird. Alternativ können die Methoden natürlich auch direkt in der konkreten Sicht definiert werden.

Des Weiteren eignet sie sich als Dokumentation der Architektur. Entwurfsmuster an sich stellen schon eine Art Dokumentation dar, aber gerade mit der Sichtbarkeit der Beschreibungen bieten sie einen hervorragenden ersten Anlaufpunkt, um eine Softwaresystem zu verstehen. Sie kommunizieren aber nicht mehr, als es das Muster an sich würde.

Ein positiver Nebeneffekt der von Squeak/Smalltalk mitgebrachten Werkzeuge, beziehungsweise der Nutzung von Versionskontrollsystemen ist, dass damit die Historie einer Architektur sichtbar wird. Wenn Beschreibungen zusätzlich dazu besser verfolgen würden, was für Änderungen sie vornehmen, könnten Entwickler damit die gesamte Architektur verwalten und auf einen älteren Stand zurücksetzen.

6.5.3 Herausforderungen und Potential

In diesem Abschnitt werden einige der Schwächen des bisherigen Konzeptes und der Implementierung aufgegriffen und daraus Ideen abgeleitet, wie die Beschreibungen erweitern werden können, um damit besser umzugehen.

6.5.3.1 Daten

Auch wenn die Sprache in der Lage ist das Observer-Pattern darzustellen, ist dies noch nicht optimal. Die eigentliche Implementierung der Methode `update:` ist redundant. Eigentlich sollen nur zu dem Subjekt Beobachter Paar die geschickten Symbole samt deren Konsequenzen beschreiben werden.

Auch problematisch ist es, wenn ein Beobachter zu mehreren Subjekten gehört, da die `update:` Methode immer nur hineinkompiliert wird. Alle Subjekte könnten in einer Beschreibung haben und diese dann mit dem Beobachter verbinden, aber wenn dies sehr viele würden, würde man diese aufgrund der Übersichtlichkeit aufspalten. Dies hätte zur Folge, dass nur die Version der Methode, deren Beschreibung zuletzt ausgeführt wurde, in die Klasse geladen wird und damit nicht das komplette Verhalten beschrieben würde.

Eine Lösung für diese beiden Probleme wäre es neben der Ausführungsebene, also dem direkten Anwenden der Beschreibung auf die Klassen, eine Datenebene einzuführen, wo zunächst Daten verarbeitet werden und deren Ergebnisse dann der Ausführungsebene übergeben werden, um sie anzuwenden. Bereitgestellt würden

[11] Die Methode `shouldBeImplemented` wirft einen Fehler zum Zwecke der Dokumentation.

nur noch das Symbol sowie der Quelltext und die Beschreibung würde die Methode schreiben.

6.5.3.2 Redundante Informationen

Was bisher noch nicht behandelt wurde ist das Verstecken von Informationen innerhalb der Beschreibungen im Zusammenhang mit dem Referenzieren von Beschreibungen. Beim Betrachten des Observer-Pattern kann festgestellt werden, dass die Implementierung der Methoden des Subjekts sich oftmals nicht geändert werden[12] und zwei von `removeDependent:` und somit für die Anwendung in der konkreten Beschreibung nicht relevant wären.

In diesem Sinne wäre es von Wert die Entwickler spezifizieren zu lassen, was zwischen Sichten als relevant mit übernommen werden soll und was nicht. Wahrscheinlich sollte aber dennoch auch die Funktionalität angeboten werden die versteckten Daten anzuzeigen, falls sie für eine bestimmte Instanz geändert werden müssten. Als erster Versuch wäre daher eine Art Zusammenfalten der redundanten Daten ein guter Kompromiss. Diese könnte aufgeklappt werden, um genauer zu spezifizieren, lenkt aber nicht so sehr ab wenn sie nicht gebraucht werden und die Beschreibung ist weiterhin vollständig.

6.5.3.3 Zusammenarbeit Squeak/Smalltalk und Beschreibungen

Bisher werden nur Beschreibungen auf Squeak/Smalltalk angewandt. Da sich aber auch der Quelltext ändert könnte es interessant sein Änderungen von Squeak/Smalltalk aus in den Beschreibungen zu ergänzen. Dadurch könnten die Beschreibungen besser als eine Art Dokumentation für die Architektur fungieren und nicht inkorrekte Sachverhalte darstellen.

Offen bleibt wie dies geschickt mit den Beschreibungen zu verbinden geht. Es könnten zum Beispiel Subklassen ohne Probleme in Squeak/Smalltalk eingefügt werden, müsste diese aber automatisch in die Beschreibung einfügen und würde somit Einfluss auf das Layout nehmen, was einen Konflikt mit der Anforderung nach freier Anordnung der Elemente nach sich ziehen könnte.

Eine weitere Idee wäre die Synchronisation mit definierten Methoden. Nachdem diese in eine Klasse geschrieben wurden könnten Änderungen, die direkt in der Klasse an der Methode vorgenommen wurden, in die Beschreibung übertragen werden. Fraglich ist ob man diesen Mechanismus so handhaben möchte, da man eventuell nur eine spezielle Klasse und nicht alle verändern möchte, wie es das Beispiel nach sich ziehen könnte.

Auch bleibt die Fragestellung offen wann die Ansicht der Beschreibung aktualisiert wird. Im Sinne der direkten Manipulation sollte dies sofort nach der Änderung geschehen. Was aber passiert, wenn gleichzeitig Änderungen an der Beschreibung, die noch nicht gespeichert wurden, und an einer Methode direkt in einer der betroffenen Klasse vorgenommen wurden.

[12] In einem aktuellen Squeak (Version 5.2, Update 18232) gibt es 5 Implementierung von `changed:` wobei die Methode 602 Referenzen im System aufweist.

Eine weitere Idee in die Richtung Zusammenarbeit ist die beiden noch näher zusammenzubringen, indem Beschreibungen SandBlocksBlöcke erzeugen, die dann im Quelltext verwendet werden können. Diese stellen dann zum einen klar, dass an dieser Stelle eine Beschreibung beteiligt ist, kann aber auch als Werkzeug genutzt werden. Aufbauend auf der Idee aus Abschnitt 6.5.3.1 könnte man zum Beispiel die `changed:` Methode als Block zur Verfügung stellen, wobei dies für den Aufruf steht, die Symbole daraus sammeln und in der Beschreibung automatisch als Datenelement zur Verfügung stellen, zu dem die darauf folgende Funktionalität spezifiziert werden muss oder bereits in der Beschreibung spezifizierte, aber noch nicht verwendete Symbole, werden ausgegraut um deren Nichtverwendung zu kommunizieren.

6.5.3.4 Verbindungen mit weiteren Informationen anreichern

Eine Verbindung besteht bisher zwischen genau zwei Elementen. Dies legt ihr allerdings gewisse Beschränkungen auf. Sie kann keine zusätzlichen Parameter haben, die sie aber eventuell benötigt. Denkt man an das Beispiel des Observers, könnte direkt über die Verbindung die Instanzvariable mit einem `Collection`[13] Typ initialisiert werden, statt dies von Hand vorzunehmen.

Dies entspräche einer Parametrisierung der Verbindungen. Wie sich das geschickt in der visuellen Ebene umsetzen lässt ist noch offen. Es könnten Mehrfachverbindungen zulassen werden, wobei sich dort bei mehr ale seinem Parameter die Frage stellt, wie man diese zuordnen kann und ob das nicht sehr unübersichtlich wäre.

6.6 Verwandte Arbeiten

6.6.1 Acme

Um Architekturen zu beschreiben gibt es dafür angepasste Beschreibungssprachen, wie zum Beispiel Acme [44]. Sie beschreiben die Architektur auf sehr abstrakter Ebene, mit zum Beispiel Interaktionsprotokollen, Bandbreiten und Latenzzeiten. Oftmals bringen sie auch Werkzeuge zum Kompilieren, Analysieren oder Simulieren der Architektur mit. Acme an sich hat einige Recht ähnliche Konzepte zu der in diesem Kapitel beschriebenen Sprache. So hat diese Sprache Komponenten (engl. *components*), Verbinder (engl. *connectors*) und Systeme (engl. *systems*), die jeweils den Elementen, Verbindungen und Referenzen auf abstrakte Beschreibungen entsprechen. Allerdings ist Acme stärker auf Architektur als auf Muster ausgelegt und ermöglicht viel mehr.

Von den Architekturbeschreibungssprachen kann sich besonders die Werkzeugunterstützung abgeschaut werden. Die Beschreibungen an sich könnten zum Beispiel SandBlocks Blöcke anbieten, die Daten sammeln, wie in Abschnitt 6.5.3.3

[13] `Collection` ist in Smalltalk die Oberklasse von Datensammlungen, wie Arrays, Mengen oder Hash-Tabellen.

bereits vorgeschlagen wurde. Aber auch könnten Werkzeuge auf den Beschreibungen aufsetzen und hätten etwas Kontext, mit dem sie arbeiten könnten. Ein Beispiel wäre das Exkludieren von bestimmten Methoden bei einem Benchmarking, da man bereits weiß, dass dies in einem aufwändigen Muster liegt und somit uninteressant ist.

6.6.2 Lisp Makros

Nach weiteren Überlegungen können die Beschreibungen aber auch als eine Art Makro Sprache betrachtet werden. Lisp Makros, wie erklärt in „On Lisp“ [51], sind eine Art auf der Metaebene zu programmieren, interagieren auf Programmcode und schreiben ihrerseits wieder Programmcode, der dann ausgeführt werden kann. Dabei steht ihnen die komplette Funktionalität von Lisp zur Verfügung[14].

Die Beschreibungen sind auch eine Art der Metaprogrammierung und sie schreiben auch Programmcode. Statt auf Programmcode zu interagieren, arbeiten sie aber auf Klassen und sind somit Makros für Klassenstrukturen.

Woran sich die Beschreibungen noch orientieren können, ist eine stärkere Einbindung der Umgebungssprache. Dadurch erhalten Entwickler die gesamte Squeak/Smalltalk Standardbibliothek, die in der Sprache verwendet werden kann. Konkret wäre dies bei dem in Abschnitt 6.5.3.1 beschriebenen Feature hilfreich, wenn die Daten zunächst transformiert werden müssen, ehe sie Verwendung finden können. Diese könnten zunächst über eine Datenverbindung an ein Element gegeben werden, das besagte Transformation mithilfe von Squeak/Smalltalk Programmcode beschreibt.

6.7 Zusammenfassung

In diesem Kapitel wurden Schwächen der aktuellen Nutzung von Entwurfsmustern und insbesondere des Observer-Musters in Squeak/Smalltalk aufgewiesen. Um diese zu schmälern wurde ein erster Entwurf für eine visuelle Programmiersprache gemacht, die diese Probleme beheben soll. Dies gelang für den gegebenen Kontext des Observers, ist aber im Hinblick auf weitere Muster und bessere Nutzbarkeit ausbaufähig.

Die Sprache sollte aber auch nie vollständig sein, sondern zum Nachdenken darüber anregen, wie Programmieren über Klassen hinweg erfolgen kann und nicht alle Details über den Quelltext verteilt sind. Dennoch bringt die Sprache bereits jetzt einige Vorteile mit sich, so können Beschreibungen als Dokumentation der Architektur verwendet werden und somit auch als Mittel zur Kommunikation und Verständnis des Systems. Aber auch hilft es Architekturänderungen zentral zu handhaben und ermöglicht eine bessere Wiederverwendbarkeit von Mustern.

[14] Sie sind Lisp Programmcode, der Lisp Programmcode entgegennimmt, um veränderten Lisp Programmcode zu schreiben.

7 Visuelle Codierung nichtlinearen Programmflusses mithilfe des SandBlocks-Systems

Programme mit nichlinearem Programmfluss besitzen eine besondere, graphbasierte, Struktur. Sie lassen sich daher schlecht durch textuelle, eindimensionale Codierung abbilden.

In diesem Kapitel wird die Einführung visueller Programmierelemente in den Entwicklungsprozess als Lösungsansatz vorgestellt. Dazu wird das SandBlocks-Projekt genutzt, welches die schnelle Entwicklung und Einbindung solcher Elemente erlaubt. Diese Nutzung führt in den gewählten Beispieldomänen, Nebenläufigkeit und Ereignisverarbeitung, zur Entwicklung besser verständlicher, besser strukturierter und besser wartbarer Programme. Ausschlaggebend dafür sind insbesondere die Aufteilung der inhaltlichen und strukturellen Programmcodierung auf textuellen und visuellen Code sowie die direktere Abbildung auf die Problemdomänen, die durch visuelle Metaphern möglich ist.

7.1 Einleitung

> *„Wir sollten (als weise Programmierer, inne unserer Grenzen) unser Äußerstes tun, um die konzeptionelle Grenze zwischen dem statischen Programm und dem dynamischen Prozess zu kürzen, um die Analogie zwischen dem Programm (ausgebreitet in Textraum) und dem Prozess (ausgebreitet in der Zeit) so trivial wie möglich zu machen.“*[1]
> *– E. W. Dijkstra* [32]

Diese Aussage entstammt dem bekannten Artikel „Go to statement considered harmful“ [32], mit dem Dijkstra einen entscheidenden Faktor zur Verbreitung strukturierter Programmierung legte, welche eine der Grundlagen moderner Softwareentwicklung ist. Dijkstra begründet mit diesem Artikel, dass mit *goto*-Befehlen codierter Programmtext schwer zu verstehen und damit zu vermeiden sei.

Text ist dabei die heutzutage übliche Art und Weise, Programme zu codieren. Text lässt sich auf eine Zeichenkette, ein eindimensionales Array reduzieren. Folgt eine Programmstruktur den drei Mustern strukturierter Programmierung, Sequenz, Selektion und Iteration, so reicht Text aus, um sie auszudrücken. All diese Muster setzen jedoch eine gewisse Linearität des Programmflusses voraus. Diese ist indes aber nicht immer gegeben. Motiviert durch Domänenspezifikationen oder programmiertechnische Konzepte, kann Programmfluss, der sich aufspaltet und mehrsträngig verläuft, wünschenswert sein. Für diese Programme ist eine eindi-

[1] Übersetzung des Autors

mensionale Codierung nicht ausreichend, um ihre Struktur direkt abzubilden. Dies spannt einen Spalt zwischen dem Programmcode und dem durch ihn beschriebenen Programm auf, welcher zu schlechter Strukturierung, erschwerter Verständlichkeit und aufwendiger Wartung führt. Analog zu Dijkstras Erkenntnissen, ist, für bestimmte Programme, mit (reinem) Text codierter Programmcode schwer zu verstehen und für diese daher zu vermeiden.

7.1.1 Forschungsfrage

Daher sollen alternative Wege der Codierung von Programmfluss zu reinem Text gefunden werden. Im Allgemeinen werden unter dem Begriff heterogene visuelle Programmiersprachen (HVPLs) Sprachen zusammengefasst, die visuelle und textuelle Elemente zur Beschreibung von Programmen kombinieren und damit alternative Codierung von Programmfluss bieten können.

Das SandBlocks-Projekt Kapitel 2 implementiert diese Konzepte, indem es die Möglichkeit bietet, generische visuelle Konstrukte zur Beschreibung von Programmen, in Squeak/Smalltalk einzubinden.

> Dieses Kapitel soll daher klären, ob die Codierung nichtlinearen Programmflusses mithilfe visueller Metaphern den konzeptionellen Spalt zwischen Programmcode und beschriebenem Programm verringern kann.

Eine Verbesserung der Entwicklung nichtlinearer Programme würde die Verbreitung der Domänen und Technologien, die diese Programme benötigen, begünstigen. Mit der verbesserten Strukturierung und Verständlichkeit würden auch die Programmqualität und Wartbarkeit steigen und damit auf lange Sicht auch die Entwicklungskosten solcher Anwendungen sinken.

7.1.2 Gliederung

Dieses Kapitel stellt die Nutzung von visuellen Metaphern zur Codierung nichtlinearen Programmflusses anhand des SandBlocks-Systems und den zwei Beispielszenarien der Nebenläufigkeit und Ereignisverarbeitung vor. Abschnitt 7.2 erläutert zunächst die in diesem Kapitel verwendeten Konzepte und Begrifflichkeiten, sowie die Grundlagen des genutzten SandBlocks-Systems. Abschnitt 7.3 motiviert anschließend ausführlich die Nutzung visueller Metaphern für die Codierung nichtlinearen Programmflusses. Anhand von Nebenläufigkeit als Beispiel eines programmiertechnischen Konzeptes und Ereignisverarbeitung als Beispiel einer Systementwurfstechnik zeigt Abschnitt 7.4 spezifische Fälle nichtlinearen Programmflusses auf. Es führt ihre konkreten Probleme textueller Codierung sowie visuelle Umsetzungen mithilfe von Sandblocks und deren Konsequenzen auf. Diese Implementierungen und Beobachtungen werden in Abschnitt 7.5 ausgewertet und in einen allgemeinen Kontext gerückt. Abschnitt 7.6 stellt verwandte Arbeiten vor, wobei hier eine Einordnung des Projekts in existierende Umsetzungen visueller

Nichtlinearität vorgenommen wird. Zuletzt schließt Abschnitt 7.7 das Kapitel mit einem kurzen Rückblick ab und skizziert weiterführende Fragen.

7.2 Hintergrund

Dieser Abschnitt stellt die in diesem Kapitel genutzten Begriffe und das genutzte SandBlocks-System vor.

7.2.1 Terminologie

7.2.1.1 Programmatische Abstraktionsebene

In diesem Kapitel werden Programme auf einer höheren Abstraktionsebene als auf Ebene einzelner Ausdrücke betrachtet, namentlich auf Ebene der Aktivitäten. Um präzise Aussagen auf dieser Ebene treffen zu können, werden nun die Konzepte und Termini dieser Ebene vorgestellt.

Aktivitäten Eine *Aktivität* ist im Allgemeinen eine logische Einheit innerhalb der Ausführung eines Prozesses. Aktivitäten können wiederum aus feingranulareren Aktivitäten komponiert sein und somit selbst einen Prozess darstellen. Im Sinne dieses Kapitels handelt es sich bei Aktivitäten insbesondere um logische Abschnitte von Programmausführung. In diesem Sinne entsprechen Aktivitäten dann den resultierenden Komponenten der funktionalen Dekomposition eines Softwaresystems.

Programmfluss *Programmfluss* beschreibt auf logischer Ebene die Beziehungen der Aktivitäten eines Programms zueinander bzw. die Reihenfolgen, in denen diese ausgeführt werden können. Programmfluss bezieht sich dabei immer auf die Abstraktionsebene eines Prozesses, da die einzelnen Aktivitäten entsprechend der obigen Definition wiederum aus anderen Aktivitäten zusammengesetzt sein können. Die Beziehung der Aktivitäten ist in der Regel kausaler Natur, wie eine Datenabhängigkeit.

Linearität von Programmfluss Programmfluss ist *linear* in einem Prozess, genau dann, wenn alle Aktivitäten dieses Kontexts sequenziell durchlaufen werden. Es gibt daher exakt eine Reihenfolge, in der die einzelnen Aktivitäten ausgeführt werden können. Demgegenüber ist Programmfluss *nichtlinear*, genau dann, wenn er nicht linear ist. Das bedeutet insbesondere, dass die Abhängigkeiten zwischen Aktivitäten nicht mehr als Liste modelliert werden können, sondern nur noch als allgemeiner Graph.

Daraus abgeleitet wird in diesem Kapitel der Begriff *nichtlineares Programm* für Programme genutzt, deren Programmfluss nichtlinear ist[2].

[2] Abweichend wird dieser Begriff auch als Synonym für nichtlineare Optimierung genutzt.

7.2.1.2 Klassifikation von Programmcode

In diesem Abschnitt werden Betrachtungen zur Anpassung der genutzten Programmiersprache und damit verbunden des geschriebenen Programmcodes getroffen. Daher werden auch diese Begrifflichkeiten zum Verständnis der folgenden Abschnitte hier aufgeführt.

Programmcode *Programmcode* ist die Beschreibung eines Programms mit geeigneten Mitteln zur Ausführung oder Kompilation durch einen Rechner. Programmcode ist meist menschenerzeugt und sollte für Zwecke der Softwarewartung möglichst gut menschenlesbar sein. Trotzdem muss Programmcode nicht zwingend textuell sein, auch beispielsweise eine Codierung durch visuelle Metaphern ist möglich.

Programmtext *Programmtext* oder auch *Quelltext* ist Programmcode in textueller Form.

Heterogene visuelle Programmiersprache *Heterogene Programmiersprachen* kombinieren mehrere Arten von Programmcode in einer Sprache. Art ist hierbei sehr weit gefasst und beschreibt die Codierungsmetaphern, die in dem jeweiligen Code verwendet werden. Eine *Heterogene visuelle Programmiersprache* (kurz HVPL) kombiniert dabei visuellen und textuellen Code, d. h. Teile ihrer Programme werden durch Text, Teile durch visuelle Metaphern beschrieben. Der Nutzer kann die Wahl haben, welche Codierung er für welche Aspekte seines Programmes nutzt, diese Zuordnung kann aber auch vorgegeben sein. Der Begriff HVPL wurde geprägt durch eine Umsetzung von Erwig und Meyer [36], ist aber auch in einem größeren Kontext gültig.

7.2.2 Nichtlinearer Programmfluss in SandBlocks

SandBlocks, durch Kapitel 2 im Detail beschrieben, implementiert eine heterogene visuelle Programmiersprache und wird in den folgenden Abschnitten als Beispiel einer solchen benutzt. Dieser Abschnitt erläutert daher die Grundmöglichkeiten von SandBlocks, komplexe Programmstruktur mithilfe visueller Metaphern zu beschreiben.

7.2.2.1 Blöcke als visuelle Elemente

SandBlocks baut auf der textuellen Sprache Smalltalk und dem UI-Framework Morphic auf. Als visuelle Elemente seiner Sprache führt es dabei das Konzept der Blöcke ein. *Blöcke* sind visuelle, interaktive, ausführbare Objekte. SandBlocks erlaubt dem Nutzer die freie Wahl, wann er Text und wann Blöcke benutzen will und bietet mehrere Möglichkeiten, von einer in die andere Ebene zu wechseln.

Einbindung Zunächst können Blöcke *inline* in Smalltalkcode integriert werden. Das bedeutet, dass der Block über Textattribute Teil des Programmtextes wird und mit diesem ausgeführt wird.

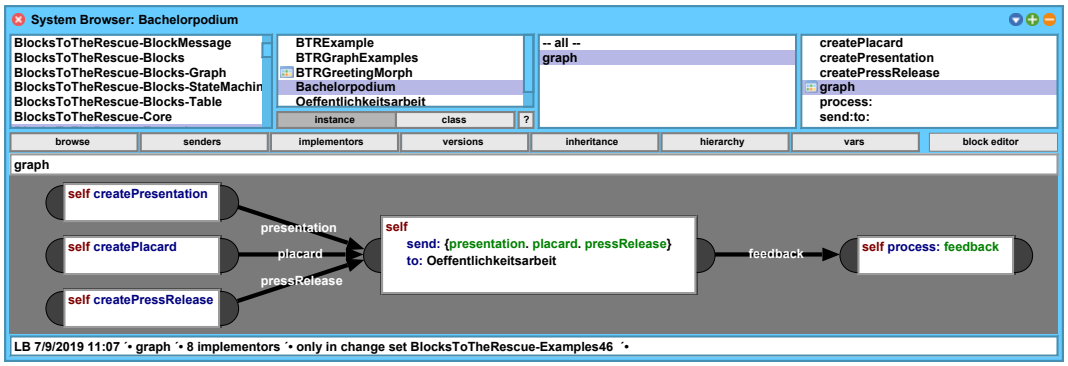


Abbildung 7.1: Beispiel einer Umsetzung des Graph-Blocks, eingebunden als Methode, angezeigt in einem Squeak Methodenbrowser

Blöcke können aber auch über eine Methodenabstraktion dem textuellen System verfügbar gemacht werden. Diese Block-Methoden verstecken das Implementierungsdetail, dass es sich bei ihrem Inhalt um visuellen Code handelt. Sie erlauben somit eine Umsetzung des Prinzips des gleichartigen Zugriffes[3] auf Methodenebene. Ein Block codiert dann eher eine Aktivität als einen einzelnen Befehl. Um aus der visuellen Ebene in die textuelle zurückzukehren gibt es aufgrund der Vielfalt der Blöcke viele Möglichkeiten. SandBlocks bietet aber bereits einen vorgefertigten Block, den *Code-Block*, welcher ein Textfeld enthält, dessen (textueller) Code bei Ausführung des Blockes interpretiert wird.

In der Regel erfolgt die Bearbeitung eines Blockes auf einer Arbeitskopie, welche beim Speichern auf das Original zurückgeschrieben wird. Dieses Original ist dann dasselbe (nicht nur das Gleiche), welches später ausgeführt wird. Es kann daher Zustand speichern. Einzelne Block, wie beispielsweise der *Breakpoint-Block*, welcher einen Schalter zum Anhalten des Programmflusses implementiert, nutzen diesen Zustand aus und manipulieren bei Bearbeitung direkt das Original. Dies wird durch einen möglichen Wahrheitswert im Block implementiert, welcher die Erstellung einer Arbeitskopie unterbindet.

Ausführung Blöcke werden immer in einem Kontext ausgeführt, welcher in seiner Abstraktion den Smalltalk-Kontexten entspricht. Diese enthalten, vergleichbar einer Materialisierung eines Stackframes, Informationen über ihren äußeren Kontext, ihren Empfänger (nur in Methoden), Parameter und lokale Variablen. Bei einer Inlineausführung ist der Kontext der Ausführungskontext der Methode, in die der Block eingebunden ist. Bei einer Ausführung als Methode ist der Kontext der Ausführungskontext derselben. Bei einer anderen Ausführung, beispielsweise aus einem anderen Block heraus, muss der Kontext manuell bestimmt werden.

7.2.2.2 Der Graph-Block

Der Graph-Block, der mit einigen Hilfsklassen in SandBlocks enthalten ist, ist ein Grundgerüst für graphbasierte Grammatiken mit Interpreterausführung. *Interpreter*, als Entwurfsmuster abgeleitet aus dem GOF-Buch [43], motiviert sich durch das hinreichend häufige Auftreten einer speziellen Problemklasse. Die Problemklasse in diesem Fall ist nichtlinearer Programmfluss, dessen Besonderheit seine Ausführungsstruktur als Graph ist.

Der Graph-Block selbst bietet zunächst Knoten, sowie Kanten zwischen diesen, als Datenstruktur an. Zudem erlaubt er grundlegende Operationen auf den ihm zugrundeliegenden Daten, wie die Ermittlung der Start- und Endknoten oder der ein- und ausgehenden Kanten eines Knotens. Zusätzlich zum Graph-Block enthält SandBlocks außerdem Grundklassen für eine visuelle Darstellung von Graphen. Der Block implementiert Methoden um als Modell für diese Darstellungen zu agieren.

Die Ausführung des Graph-Blocks startet standardmäßig einen neuen Interpreter der Interpreterklasse seiner Wahl auf ihm. Dieser übernimmt dann die Ausführung. Die angedachte Abstraktion der Knoten und Kanten in dieser Ausführung ist die von Aktivitäten und ihren Abhängigkeiten und sonstigen Beziehungen. Hier sind jedoch auch andere Interpretationen und Implementierungen möglich. Auch für den Interpreter ist eine Gerüstklasse gegeben. Diese bietet grundlegende Möglichkeiten für die Ausführung eines Knotens. Dabei werden zuerst alle Parameter des Knotens gezogen. Dann wird der Knoten mit diesen und dem allgemeinen Kontext der Ausführung des Graphen ausgeführt. Hierbei wird auf nötige Kantenaktivierungen und Knotenreaktivierungen gehört und diese verarbeitet. Kanten können in diesem Interpreter Werte tragen und gelten dann als aktiv, diese Werte können dann bei der Ausführung ihrer Zielknoten als Parameter gezogen werden.

7.3 Codierung nichtlinearen Programmflusses

Anforderungen an die Codierung nichtlinearen Programmflusses weichen aufgrund einiger Einzigartigkeiten der Nichtlinearität von denen linearen Programmflusses ab. In diesem Abschnitt werden diese Eigenheiten erläutert und damit motiviert, andere als rein textuelle Sprachen zur Codierung von Nichtlinearität zu nutzen.

7.3.1 Eigenschaften nichtlinearen Programmflusses

Nichtlinearer Programmfluss ist Form einer Programmstruktur und ergibt sich somit in erster Linie aus den Entscheidung des Entwicklers beim Programmentwurf. Relevant für diese Entscheidung ist jedoch, *was* programmiert werden soll. Nichtlinearität ergibt sich dann in der Regel aus der jeweiligen Domäne, da wenige

[3] Dieses Prinzip wird häufig seiner englischen Übersetzung folgend „Uniform Access“ genannt.

Vorgänge der reellen Welt einer strikten Sequenzialität folgen. Auch das gewählte Programmiermodell kann entscheidend sein, wie die in Abschnitt 7.4.1 vorgestellte Nebenläufigkeit zeigt.

Linearer Fluss besitzt genau einen Ausführungsstrang, dem ein Mensch gedanklich folgen kann, um ein System zu verstehen. Nichtlinearer Fluss hat mehrere mögliche Ausführungsfolgen, möglicherweise lässt sich die Reihenfolge zweier Aktivitäten überhaupt nicht bestimmen, da diese gleichzeitig stattfinden oder in keiner Abhängigkeitsbeziehung zueinanderstehen. Ein Abhängigkeitsgraph der Aktivitäten in linearem Programmfluss bildet immer eine zusammenhängende Liste, da alle Aktivitäten sequenziell abgearbeitet werden. Im Gegensatz dazu können, durch die Aufhebung der Beschränkung auf maximal eine Vorgänger- und Nachfolgeaktivität, in nichtlinearen Programmen allgemeine, unter Umständen sogar nicht zusammenhängende, Graphenstrukturen entstehen. Diese mögliche Unabhängigkeit zweier Aktivitäten stellt das Kernkonzept der Nichtlinearität dar. Sie lässt sich durch klassische, rein textuelle Codestrukturen, bei denen eine Zeile auf die andere, ein Wort auf das nächste folgt aber nicht oder nur schlecht darstellen.

7.3.2 Herausforderungen textueller Codierung

Disparität der Abbildung Text lässt sich auf eine Zeichenkette, eine eindimensionale Liste von Zeichen, reduzieren. Dies reicht aus, um eine Liste von Befehlen, also linearen Programmfluss, zu codieren. Die lineare Struktur textuellen Codes bildet aber nicht auf die nichtlineare Struktur nichtlinearen Programmflusses ab. Text erlaubt eine Positionierung von Programminhalt nur auf einer Dimension, der Position in der Zeichenkette. Nichtlineare Prozesse lassen sich jedoch, wie oben gezeigt, nicht auf eine „Kette“ reduzieren, sondern erlauben ein Nebeneinander von Aktivitäten.

Viele Programmierdomänen besitzen eine natürliche Nichtlinearität. Sie also in einem linearen System umzusetzen würde die Modularitätsregel der *direkten Abbildung* brechen. Das resultierende nichtlineare Programm in einem textuell-linearen Code zu beschreiben würde wiederum die direkte Abbildung zwischen Programm und Code brechen. Die implizite Linearität textuellen Codes widerspricht der tatsächlichen Programmstruktur. Dies verringert das Verständnis der Programmstruktur, aber auch der Domäne, die schlechter abgebildet wird.

Explizite Strukturbeschreibung Da in klassischen Programmiersystemen die Textstruktur immer linear ist, lässt sich mit ihr keine implizite Nichtlinearität ausdrücken. Es müssen daher explizite Sprachkonstrukte genutzt werden, um die Programmstruktur zu beschreiben. Insbesondere diese Konstrukte bilden häufig nicht gut auf die nichtlinearen Konzepte ab, da die Eindimensionalität des Textes beispielsweise nicht ausreicht, um gleichzeitige, konzeptionell „nebeneinanderliegende“ Programmabschnitte darzustellen.

Die Nutzung dieser Sprachkonstrukte ist häufig komplex, was Entwickler davon abhält sie zu nutzen. Bei einer Nutzung führt die Komplexität wiederum zu einer weiteren (menschlichen) Fehlerquelle.

Zudem erschwert die explizite Strukturbeschreibung sowohl die Dekomposition, als auch die Komposition nichtlinearer Systeme. Die Dekomposition, da derselbe Verwaltungsaufwand gegebenenfalls für jedes der Subprobleme vorgenommen werden muss und somit deren Komplexität weniger sinkt. Die Komposition, da zum Komponieren nichtlinearer Systeme gegebenenfalls erneuter Verwaltungsaufwand nötig wird.

Des Weiteren sinkt die Verständlichkeit und Lesbarkeit des betroffenen Programmes, da Code zur Beschreibung des Programminhaltes mit Code zur Beschreibung der Programmstruktur auf einer Codeebene vermischt wird. Damit steigt der nötige Text zum Beschreiben eines Programmes. Zudem ist es nun nicht mehr möglich, ein einzelnes Verständnis über Struktur oder Inhalt zu gewinnen, ohne den Code des jeweils anderen Aspekts mitlesen zu müssen.

Nichtlineare Systeme benutzen häufig eine geringe Menge an Strukturen, um nicht durch Reizüberflutung überladen zu werden. Durch den Zwang, diese aufwendig wieder und wieder textuell codieren zu müssen, entsteht unter Umständen eine hohe Codeduplikation, die weiteres Rauschen in die Beschreibung des Programminhalts bringt.

Konsequenzen Schlechtere Lesbarkeit und Verständlichkeit und hohe Komplexität erhöhen den Wartungsaufwand und sorgen damit dafür, dass die Programmqualität abnimmt. Textuelle Codierung nichtlinearen Programmflusses führt also zu schwer verständlichem, schlecht wartbarem, fehleranfälligem Programmcode.

Aufgrund dieser Probleme textueller Codierung nichtlinearen Programmflusses bietet sich eine Trennung der Strukturbeschreibung und Inhaltsbeschreibung in zwei Codierungsebenen an. Die Strukturebene soll die Linearität einer rein textuellen Beschreibung brechen, während die Inhaltsebene die Mächtigkeit rein textueller Beschreibung erhalten soll, damit dieselben Inhalte ausgedrückt werden können wie zuvor. Damit bietet sich eine heterogene visuelle Programmiersprache, welche sowohl eine Integration der visuellen, strukturbeschreibenden Metapher in das textuelle System, als auch die Reintegration des textuellen, inhaltsbeschreibenden in die visuelle Metapher erlaubt, an. Ein System, das alle diese Anforderungen erfüllt, ist das SandBlocks-System.

7.4 Visuelle Codierung nichtlinearer Domänen

Dieses Kapitel stellt heterogene visuelle Programmiersprachen als Alternative zu rein textuellen Programmiersprachen zur Codierung nichtlinearen Programmflusses vor. Anhand der Beispiele nebenläufiger Programmierung und Ereignisverarbeitung wird die Nutzung auf programmiertechnischer und Systementwurfsebene evaluiert. Dafür werden je eine rein textuelle und eine durch SandBlocks unterstützte Implementierung der Beispiele verglichen.

7.4.1 Nebenläufige Programmierung

> *„Multicore- und Manycore-Prozessoren sind heutzutage allgegenwärtig, paralleles Programmieren bleibt jedoch so schwierig wie vor 30-40 Jahren“*[4]
> aus „Parallel Programming in the Age of Ubiquitous Parallelism“ [84]

Computer, die echte Parallelität erlauben, weil sie Mehrkernprozessoren und GPUs enthalten, sind heutzutage der Standard. Die Parallelisierung entwickelter Anwendungen ist es jedoch nicht - auch weil der Prozess als mühselig gilt. Die folgende Sektion stellt nebenläufige Programme als Beispiel nichtlinearen Programmflusses vor, erläutert die Probleme bei der Entwicklung nebenläufiger Programme und evaluiert anhand eines Beispiels die Verwendung von HVPLs exemplarisch mithilfe des SandBlocks Graphsystems.

7.4.1.1 Hintergründe nebenläufiger Systeme

Nebenläufigkeit Zwei Aktivitäten A und B sind *nebenläufig*, wenn keine kausale Abhängigkeit zwischen ihnen besteht, das heißt A kann vor B, nach B oder parallel zu B ausgeführt werden. Parallelität sei hier Synonym für tatsächliche logische Gleichzeitigkeit und wird durch Nebenläufigkeit nicht notwendigerweise impliziert.

Parallelisierung *Parallelisierung* beschreibt die Tätigkeit von Entwicklern, Programme zum Zwecke der nebenläufigen Ausführung und Synchronisierung in Aktivitäten zu zerlegen oder anzupassen. Parallelisierung ist damit eine Tätigkeit im Rahmen der Entwicklung nebenläufiger Prozesse.

Gründe für die Nutzung von Nebenläufigkeit Es gibt mehrere Gründe für die Nutzung von Nebenläufigkeit. Häufig angebracht wird, dass Parallelisierung zur Optimierung benutzt werden kann und sogar muss, um das Wachstum an Rechengeschwindigkeit, dass durch das Mooresche Gesetz beschrieben wird, aufrechtzuerhalten. Das vorausgesehene exponentielle Wachstum flaute ab der Mitte der 00' Jahre aufgrund von technischen Limitierungen des Energieverbrauches und der Speichergeschwindigkeit stark ab (vgl. [77]). Vermehrte Parallelisierung gilt als eine der Kerntechnologien, um ein weiteres Wachstum der Rechengeschwindigkeit zu erwirken. Werden zwei Aktivitäten nebenläufig ausgeführt, kann ein intelligenter Scheduler beispielsweise Wartezeiten einer Aktivität nutzen, indem er die andere Aktivität in dieser Zeit weiter ausführt. Bei paralleler Ausführung zweier Aktivitäten ergibt sich gegenüber sequenzieller Ausführung sogar eine Zeitersparnis von bis zu der Dauer der kürzeren Aktivität.

Ein weiterer Grund für Parallelisierung findet weit weniger Beachtung, ist jedoch, insbesondere im Rahmen dieser Arbeit, trotzdem von großer Bedeutung: Für viele Anwendungsfälle ist Nebenläufigkeit natürlicher und vereinfacht somit das Programmdesign. Obwohl Nebenläufigkeit einige technische Herausforderungen

[4] Übersetzung des Autors

mit sich bringt, bildet die strenge sequenzielle Ordnung, wie sie für gewöhnlich mit imperativen Sprachen implizit erzeugt wird, nicht zwingend das Problem ab. Jeder moderne teamgetriebene Arbeitsprozess basiert darauf, dass mehrere Menschen parallel an unterschiedlichen Aufgaben arbeiten und sich zwischendrin synchronisieren. Computerprozesse sind hier nicht anders: Sollen für eine Simulation zum Beispiel mehrere unabhängige Eingabedateien vorbereitet werden, gibt es keinen Grund, eine „künstliche“ sequenzielle Ordnung zu erzeugen - obwohl exakt dies viele geläufige Sprachen tun, sofern nicht explizit eine Parallelisierung vorgenommen wird.

7.4.1.2 Motivation für die Nutzung von SandBlocks

Obwohl die Nutzung von Nebenläufigkeit aus technischer und Entwicklersicht immer wichtiger erscheint, ist sie doch stark beschränkt, da sich aus historischen Gründen stark sequenzielle Programme als „Standard“ etabliert haben. McCool et al. stellen fest, dass dieser starke Fokus, serielle Programme als Standard zu betrachten, sich in das Design moderner Programmiersprachen eingewoben hat [77]. Betrachtet man verbreitete Programmiersprachen (unter anderem Java, C(++), Python, JavaScript, PHP, Ruby [5]), so fällt auf, dass zwar alle betrachteten Sprachen Mittel zur Parallelisierung bereitstellen, jedoch bei allen Sprachen „standardmäßiger“ Programmcode, d. h. Code der nicht explizit parallelisiert wurde, eine implizite Sequenzialität anhand der textuellen Ordnung aufweist. Hervorzuheben ist hierbei zumindest JavaScript, das mit dem *async*-Keyword eine auf Promises basierte Nebenläufigkeit bereits tief verankert hat.[6]

> *„Parallele Programmierung ist schwieriger als sequenzielle Programmierung, da parallele Programme nicht nur die sequenziellen Berechnungen, sondern auch die Interaktionen [...] zwischen diesen Berechnungen, welche die Parallelität definieren, ausdrücken müssen. Um eine gute Leistung zu erzielen müssen Programmierer diese umfangreichen Strukturen verstehen.“*[7]
> aus „Visual programming and debugging for parallel computing“ [18]

Diese Notwendigkeit, Nebenläufigkeit explizit und häufig kompliziert angeben zu müssen, um sie zu nutzen, hemmt die Nutzung stark, insbesondere da sie einen Einstieg in nebenläufige Programmierung erschwert. Dies führt dazu, dass sich wenige Programmierer die Fähigkeit aneignen, nebenläufig zu Programmieren. Zudem erhöht sich die Komplexität des Programms durch den Aufwand zum Verwalten der Konstrukte, die zur Umsetzung der Nebenläufigkeit nötig sind. Programmtext muss geschrieben werden, der nicht der Beschreibung des Programminhalts, sprich der Aktivitäten, sondern der expliziten Verwaltung des Programmflusses dient. Dieser strukturelle Overhead wirkt sich negativ auf die Lesbarkeit des Programmes aus und verschlechtert damit die Wartbarkeit. Besser

[5] Ermittelt unter Anderem anhand des TIOBE-Indexes unter `https://www.tiobe.com/tiobe-index/`, letzter Zugriff am 11. Dezember 2019.

[6] Weitere Informationen beispielsweise unter `https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/async_function`, letzter Zugriff am 11. Dezember 2019.

[7] Übersetzung des Autors

wäre, wenn sich die Struktur des Programmes durch die Struktur seines Codes darstellen ließe.

Die Struktur nebenläufiger Programme unterscheidet sich von der sequenzieller. Sequenzielle Programme weisen per Definition eine natürliche Linearität auf, welche sich mit der Linearität von Programmtext deckt. Alle Aktivitäten werden in einer vorgegebenen und damit deterministischen Reihenfolge abgearbeitet. Nebenläufige Programme erlauben zusätzlich eine Spaltung (*Fork*) und Zusammenführung (*Join*) des Programmflusses. Dieser entwickelt sich daher von einer Liste zu einem allgemeineren Graphen. Somit liegt nichtlinearer Programmfluss im Sinne der vorangegangenen Abschnitte vor, welcher eine Disparität zur linearen Struktur üblichen Programmtextes aufweist.

Ein Ansatzpunkt für die Verbreitung nebenläufiger Programmierung ist daher, auf Ebene der Programmiersprachen Änderungen vorzunehmen. Das SandBlocks-System bietet hierfür die nötigen Werkzeuge.

7.4.1.3 Allgemeiner Lösungsansatz

Da der Programmfluss eine Graphstruktur bildet, bietet es sich an, ihn als solchen darzustellen. Dieser Graph versteckt die Details der Programmflussverwaltung hinter einer geeigneten visuellen Metapher, damit sich der Entwickler auf die Problemkomposition und Parallelisierung konzentrieren kann. Dies wird erreicht, indem die Spezifikation der Nebenläufigkeit und die Spezifikation des Programminhalts getrennt werden, auf die Ebene des Graphen und die Ebene der Knoteninhalte, sprich der Aktivitäten [34, S. 18, 20]. Nebenläufigkeit wird dabei implizit, aber ersichtlich erzeugt.

Die Knoten des Graphen entsprechen den Aktivitäten des Programmes. Um die generelle Nutzbarkeit des Graphen zu gewährleisten, sind die Aktivitäten wiederum durch den im Zielsystem verfügbaren Programmcode ausdrückbar. Dies hat den Vorteil, dass die Aktivitätsinhalte in einer bekannten, gut getesteten und gut durch Werkzeuge unterstützten Sprache definiert werden können, was beispielsweise das Debugging erleichtert.

Die Kanten des Graphen entsprechen den Abhängigkeiten der Aktivitäten zueinander. Diese Abhängigkeiten bilden dabei immer eine reine Kausalität oder eine Datenbeziehung ab. Die Kanten haben daher die Möglichkeit, Daten zu transportieren. Bei mehreren ein- oder ausgehenden Kanten eines Knotens müssen die Dateneingänge und -ausgänge über Bezeichner identifizierbar sein.

Eine Aktivität kann gestartet werden, wenn alle ihrer Abhängigkeiten erfüllt sind. Dies entspricht einer natürlichen Synchronisation, einem Warten auf die Ergebnisse (ob durch Seiteneffekte oder Datenfluss) aller nötigen Aktivitäten. Die Ausführung einer Aktivität erfüllt wiederum all ihre Abhängigkeiten sobald sie terminiert. Auf konzeptioneller Ebene entspricht dies dem Gedanken einer Aktivität, die in sich geschlossen ist, sodass nur auf das Beenden der gesamten Aktivität gewartet werden kann. Könnten einzelne Kanten aktiviert werden, würde diese Kapselung verletzt. Außerdem bleibt das Forkverhalten somit rein innerhalb der visuellen statt der textuellen Notation. Es ist damit allerdings nicht direkt möglich ist, dynamisch

zur Laufzeit Aktivitätsinstanzen zu erzeugen. Indirekt ist dies aber über Rekursion möglich.

Zu Beginn der Ausführung eines Graphen können alle Aktivitäten gestartet werden, die keine Abhängigkeiten besitzen. Die Termination des Graphen ist wiederum abhängig von einer (potenziell leeren) Menge von Aktivitäten, die er enthält.

Um Problemdekomposition und Rekursion zu ermöglichen, entspricht der Graph selbst der Abstraktionsstufe einer Aktivität und kann auch als solche behandelt werden. Beispielsweise kann der Graph als Methode abstrahierbar gemacht werden. Er schafft somit einen Kontextbereich, in dem Nebenläufigkeit der Standard ist. Dies hat insbesondere den Vorteil, dass bei jeder Methode abstrahiert wird, ob sie nebenläufig ausgeführt wird oder nicht.

7.4.1.4 Implementierung in SandBlocks

Diese Spezifikation wird nun mithilfe des SandBlocks-Systems in Squeak/Smalltalk umgesetzt. Die Implementierung baut dabei auf dem in Abschnitt 7.2.2.2 beschriebenen, in SandBlocks enthaltenen, Graph-Block auf. Durch die Implementierung als Block kann der Graph als Methode oder in einer Methode eingebunden werden.

Der Graph-Block bietet dabei bereits Grundstrukturen für die Umsetzung der Spezifikation: Aktivitäten können wie vorgesehen auf die Knoten, und Abhängigkeiten zwischen ihnen auf die Kanten abgebildet werden.

Die Kanten werden durch einen Namen identifiziert. Damit kann bei mehreren ausgehenden Kanten eines Knotens die Zielkante und bei mehreren eingehenden Kanten der Datenursprung bestimmt werden. Das bedeutet konkret, dass Knoten mithilfe von `activateEdgeNamed:` reinen Kontrollfluss, und mithilfe von `activateEdgeNamed:with:` Datenfluss gezielt an ihre Nachfolger weitergeben können. Außerdem werden die eingehenden Daten eines Knotens dann unter dem Namen der Kante in seinem Ausführungskontext bereitgestellt. Sind für einen Knoten alle eingehenden Kanten aktiv und damit alle nötigen Abhängigkeiten erfüllt, kann er mithilfe von `valuesWith:in:` ausgeführt werden. Als Argumente erhält ein Knoten dabei die gesammelten Kantenwerte als Parameter und den Gesamtkontext der Ausführung des Graph-Blocks als Methodenkontext in dem er ausgeführt wird, beispielsweise um Zugriff auf in diesem Kontext verfügbare temporäre Variablen zu gewähren.

Für die Umsetzung der Aktivitäten können eigene Knotenklassen benutzt werden. Denkbar wäre beispielsweise die Nutzung der Methodenabstraktion als Aktivitätsebene. Ein Knoten würde dann einen Methodenaufruf darstellen. Vielseitiger ist jedoch die Nutzung der Block-Abstraktion als Aktivitätsebene. Ein Knoten enthält dann genau einen Block, den er ausführt. Mithilfe des *Code-Blocks* ist es somit auch möglich, einen einzelnen Methodenaufruf als Aktivität zu konfigurieren. Zudem können andere Blöcke, insbesondere auch neue Graphen, als Aktivitäten eingesetzt werden. Hier ist jedoch darauf zu achten, die Abstraktionsebene einzuhalten. Dafür hilft wiederum die mögliche Abstraktion von Blöcken als Methoden. Ein großer Vorteil ist auch, dass die Ausführungsschnittstellen der Aktivitätsknoten und der Blöcke einander ähneln: Beide werden in einem Kontext mit verfügbaren temporären Variablen ausgeführt. Dadurch entsteht nur geringer Übersetzungs-

aufwand bei der Ausführung eines Knotens, namentlich die Kombination aus den Kontextinformationen des Graph-Blocks (Empfänger und Parameter der aktuellen Methode) und den speziellen Ausführungsinformation des Knotens (Parameter aus den Kanten) zu einem einzelnen Kontext, entsprechend der Spezifikation der Blöcke. Die Knoten in dieser Spezifikation sind so konfiguriert, dass sie automatisch den Rückgabewert des Blockes (bei Code-Blöcken der Wert des letzten Statements) als Array interpretiert zurückgeben. Dieses wird dann auf die ausgehenden Kanten des Knotens aufgeteilt.

Der Graph-Block startet bei seiner Ausführung zunächst alle Startknoten, d. h. Aktivitäten, die keine Abhängigkeiten außer dem Start des Graphen besitzen. Er terminiert dann, wenn er Ergebnisse aller Endknoten besitzt, d. h. Aktivitäten, von denen keine anderen Aktivitäten abhängen, und akkumuliert diese Ergebnisse zu einem Array. Er erlaubt dabei noch keine Endknoten, auf deren Termination nicht gewartet werden muss. Für diese wäre eine explizite Notation für die Rückgabe nötig, beispielsweise die Einführung eines dedizierten Rückgabeknotens. Eine Implementierung eines solchen steht noch aus.

Das spezielle Verhalten des Graph-Blocks für Nebenläufigkeit wurde mithilfe eines eigenen Graphinterpreters implementiert, welcher das Verhalten bei Knotenausführung und Kantenaktivierung, sowie den Rahmen um die Graphausführung definiert.

7.4.1.5 Beispielspezifikation und -umsetzungen

Anhand eines Beispiels wird nun die Nutzung visueller Systeme, insbesondere des Graph-Blocks, zur Umsetzung von Nebenläufigkeit evaluiert. Dazu wird das Beispiel sowohl textuell als auch visuell implementiert.

Beispielspezifikation Als Beispiel für Nebenläufigkeit gegeben sei der Vorgang ein Erdnussbutter-Gelee-Sandwich herzustellen[8]: Um das Sandwich herzustellen, müssen je eine erdnussbutter- und eine geleebestrichene Scheibe Brot aufeinandergelegt werden. Für eine bestrichene Scheibe werden eine blanke Scheibe sowie der gewünschte Belag benötigt. Vor dem Servieren müssen außerdem noch die übrigen Beläge weggeräumt werden. Damit ergeben sich folgende acht Aktivitäten: Erdnussbutter holen, Gelee holen, zwei Scheiben Brot holen, Ednussbutterscheibe bestreichen, Geleescheibe bestreichen, Scheiben zusammenlegen, Erdnussbutter weglegen, Gelee weglegen.

Textuelle Umsetzung In Listing 7.1 wurde das Szenario beispielhaft in Smalltalk umgesetzt. Die Methode ist äußerst komplex, messbar an den insgesamt zwölf temporären Variablen für die einzelnen Aktivitäten und ihre Ergebnisse, der Länge der Methode sowie der Länge und Komplexität der einzelnen Zeilen. Es ist nötig, die Aktivitäten als Variablen umzusetzen, da Referenzen auf sie für das

[8] Dieses Beispiel wurde gewählt, da es eine hohe Anzahl an Forks und Joins bei einer geringen Anzahl Aktivitäten besitzt.

Warten/Synchronisieren ihrer Nachfolgeaktivitäten benötigt werden. Es ist nötig die Aktivitätsergebnisse als Variablen zu implementieren, da keine Konstrukte zur Kommunikation zwischen den Aktivitäten gegeben sind.

Es ist schwer ersichtlich, welche Aktivitäten in welchen Abhängigkeiten stehen. Der Grund hierfür ist, dass die Aktivitäten durch ihre Bezeichner, also textuelle Symbole, identifiziert werden. Um zu verstehen, welche Verbindungen die Aktivität besitzt muss im Quelltext nach diesem Symbol gesucht werden. Alle Vorkommen zu finden ist dabei schwierig.

Des Weiteren zeichnet sich der Code durch seine hohe Duplikation, insbesondere strukturell, negativ aus. Obwohl die Benutzung einer dedizierten Aktivitätenklasse die Details der Prozessverwaltung bereits teilweise versteckt, weisen die Konstruktionen der Aktivitäten eine hohe Duplikation auf. Da Text zur Identifikation der Aktivitäten genutzt wird, lässt sich dieser Code kaum weiter vereinfachen. Angemerkt werden muss hierfür allerdings, dass dies teilweise den in der Sprache verfügbaren Konstrukten geschuldet ist. Gerade dies ist jedoch der Kritikpunkt an textuellem Code: Ihm fehlen die Mittel, Nichtlinearität adäquat darzustellen. Die Reihenfolge, in der die Aktivitäten erzeugt werden, suggeriert zusätzlich eine feste Ausführungsfolge. Dass das Wegräumen der Erdnussbutter, das Wegräumen des Gelees und das Zusammenlegen der beiden Scheiben in beliebiger Reihenfolge erfolgen kann, wird beispielsweise durch den textuellen Code nur nach mehrfachem Lesen ersichtlich.

Listing 7.1: Beispiel umgesetzt in Smalltalk

```
SandwichMaker >> #makeSandwichTextually

   | peanutButter jelly bread result
   takePeanutButter takeBread takeJelly spreadPeanutButter spreadJelly
   combineBread putAwayPeanutButter putAwayJelly |

   takePeanutButter := Activity do: [peanutButter := self takePeanutButter].
   takeBread := Activity do: [bread := {self takeBread. self takeBread}].
   takeJelly := Activity do: [jelly := self takeJelly].
   spreadPeanutButter := Activity
      after: {takePeanutButter. takeBread}
      do: [bread at: 1 put: (self spread: peanutButter on: bread first)].
   spreadJelly := Activity
      after: {takeJelly. takeBread}
      do: [bread at: 2 put: (self spread: jelly on: bread second)].
   combineBread := Activity
      after: {spreadJelly. spreadPeanutButter}
      do: [result := bread first, ' and ', bread second].
   putAwayPeanutButter := Activity
      after: spreadPeanutButter
      do: [self putAwayPeanutButter].
   putAwayJelly := Activity
      after: spreadJelly
      do: [self putAwayJelly].
   {combineBread. putAwayPeanutButter. putAwayJelly} do: #wait.

   ^ result
```

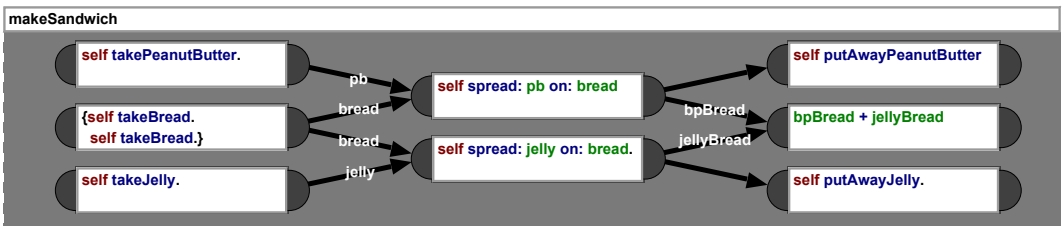


Abbildung 7.2: Beispiel umgesetzt in SandBlocks

Visuelle Umsetzung In Abbildung 7.2 wurde das obige Szenario nun im SandBlocks Graph-Block für Nebenläufigkeit umgesetzt. Sofort auffällig ist, dass sich der textuelle Code auf die auszuführenden Methoden beschränkt und die Nebenläufigkeit rein durch den visuellen Code ausgedrückt wird. Somit wird eine Trennung der Zuständigkeiten umgesetzt: Der Text beschreibt den Programminhalt, der Graph bzw. die Anordnung des Textes beschreibt die Struktur des Programms. Abhängigkeiten zwischen den Aktivitäten sind durch die Pfeilkanten direkt ersichtlich, insbesondere sind alle Informationen, die eine Aktivität betreffen (Abhängigkeiten und Inhalt) an einer räumlichen Stelle im Code. Die Codestruktur bildet also genauer auf die Programmstruktur ab.

7.4.2 Ereignisverarbeitung

Ereignisverarbeitung ist eine Herangehensweise an den Softwareentwurf, der auf dem Erkennen, Analysieren und Verarbeiten von Ereignissen basiert. Ein *Ereignis* im Sinne dieser Systeme ist das Auftreten eines relevanten Vorkommnisses oder einer relevanten Situation in einer bestimmten Domäne [37, Kapitel 1.1.1].

Anwendungen von Ereignisverarbeitung Ereignisverarbeitung ist eine häufig genutzte Technologie der Softwareentwicklung und befindet sich in stetigem Wachstum[9]. Klassische Anwendungsfälle sind das Geschäftsprozessmanagement, Observationen anhand von Sensortechnik, beispielsweise zur Wettervorhersage oder in der Medizin zur Überwachung von Patienten, oder Informationsorganisation, beispielsweise die Analyse von Mustern am Finanzmarkt. Im Allgemeinen eignet sich Ereignisverarbeitung für kontinuierlich eingehende Datenströme. Es ist daher eine Wertvolle Technologie für viele aktuelle Trends der Softwarebranche: Das Internet of Things erhält Datenereignisse von vielen verschiedenen Geräten, die verarbeitet werden müssen. Big Data zeichnet sich durch *Velocity* und *Volume* aus und benötigt eine Echtzeitverarbeitung seiner Daten. Vernetzte Fabriken der Industrie 4.0 produzieren Sensordaten, auf denen in Echtzeit Situationserkennungen durchgeführt werden müssen, um beispielsweise Fehlfunktionen vor ihrem Eintritt zu entdecken.

[9] Vgl. beispielsweise `http://www.complexevents.com/2019/06/09/eight-trends-in-event-stream-processing-may-2019/`, letzter Zugriff am 11. Dezember 2019.

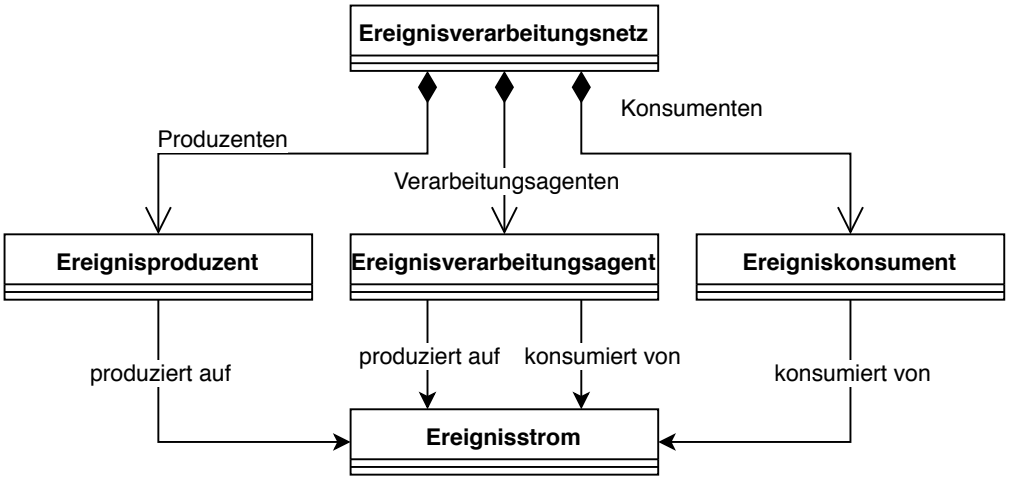


Abbildung 7.3: Abstrakte Struktur von Ereignisverarbeitungsnetzen (UML-Klassendiagramm)

Gründe für die Nutzung von Ereignisverarbeitung [37, Kapitel 1.2.3] Die Anwendungsfälle der Ereignisverarbeitung haben bereits einige der Vorteile dieser Herangehensweise aufgedeckt. Viele Domänen und Technologien besitzen von Natur aus Ereignischarakter. Quellen, die kontinuierlich Ereignisse produzieren, sind durch die Domäne gegeben. Um der Modularitätsregel der direkten Abbildung gerecht zu werden sollten Anwendungen so nah wie möglich an ihrer Problemdomäne angelehnt sein.

Ein ereignisgetriebener Programmieransatz führt, im Gegensatz beispielsweise zu einem Bündelverarbeitungsansatz, zu einer direkten Verarbeitung der Ereignisse. Dies ist insbesondere für die Situationserkennung relevant, deren Reaktion zeitkritisch ist und daher sofort erfolgen sollte. Dedizierte Ereignisverarbeitungssysteme führen eine hohe Abstraktion für den Umgang mit Ereignissen ein und können daher die Entwicklung und Verständlichkeit von Systemen stark vereinfachen. Zuletzt entkoppelt eine ereignisgetriebene Architektur die Ereigniserzeugung, -verarbeitung und -verwendung. Erzeugungs- und Verwendungskomponenten müssen sich nicht mehr kennen und können daher modular ausgetauscht werden.

7.4.2.1 Grundstrukturen von Ereignisverarbeitungsnetzen

Ereignisverarbeitungsanwendungen bestehen grundsätzlich aus vier Arten von Komponenten: *Ereignisproduzenten*, *Ereigniskonsumenten*, *Ereignsverarbeitungsagenten* und *Ereigniskanälen* (s. Abbildung 7.3). Diese sind durch Ereignisströme zu graphbasierten Netzstrukturen, sogenannten *Ereignisverarbeitungsnetzen*, verbunden. Formalisiert sind Ereignisverarbeitungsnetze also gerichtete Graphen, wobei die Kanten Ereignisströme sind und die Knoten Ereignisproduzenten, -konsumenten und -verarbeitungagenten darstellen.

Ereignisströme sind formal betrachtet eine temporal geordnete, zusammengehörige Menge von Ereignissen. Konzeptionell handelt es sich um das Kommuni-

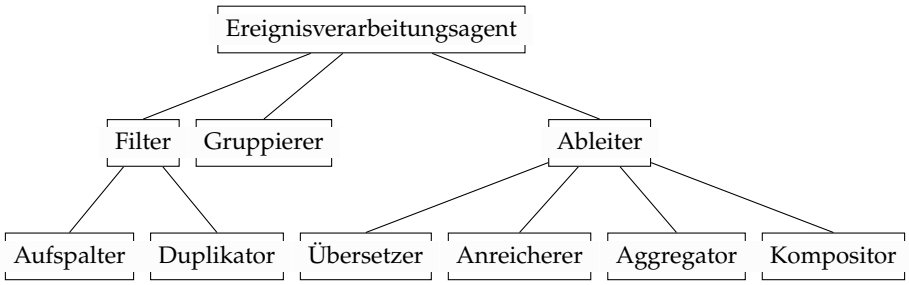


Abbildung 7.4: Vereinfachte Taxonomie von Ereignisverarbeitungsagenten

kationsmedium zwischen den Komponenten eines Ereignisverarbeitungsnetzes. Ereignisströme transportieren Ereignisse, indem einer ihrer Eingänge diese an sie weitergibt und ihre Ausgänge sie wieder abfragen können. Zur Vereinfachung sollen im Folgenden nur Ereignisströme mit genau einem Eingang und genau einem Ausgang betrachtet werden.

Ereignisproduzenten sind beispielsweise Sensoren, Geschäftsprozess-Engines (bzw. Steuerungssoftware) oder ein UI-Controller, der Nutzereingaben veröffentlicht. Im Allgemeinen sind also Hardware-, Software- und Nutzerinteraktionskomponenten möglich. Sie besitzen ausgehende Ereignisströme, was bedeutet, dass sie Ereignisse produzieren, d. h. in das System einspeisen, können.

Ereigniskonsumenten sind beispielsweise Lichtsysteme, Monitoringsysteme oder graphische Nutzerschnittstellen. Auch hier sind Hardware-, Software- und Nutzerfeedbackkomponenten möglich. Sie besitzen eingehende Ereignisströme, was bedeutet, dass sie Ereignisse konsumieren, d. h. aus dem System aufnehmen und verarbeiten, können.

Ereignisverarbeitungsagenten Das Herzstück der Ereignisverarbeitung sind *Ereignisverarbeitungsagenten*, kurz EPAs (Event Processing Agents), da sie die Verarbeitungslogik enthalten. EPAs besitzen dementsprechend sowohl ein- als auch ausgehende Ereignisströme und können daher je nach Definition als Ereigniskonsumenten *und* -produzenten angesehen werden. Im Gegensatz zu diesen, welche an den Grenzen oder außerhalb eines Ereignisverarbeitungssystems liegen, befinden sich EPAs jedoch innerhalb Systems [37, Kapitel 2.2.1]. EPAs können nach den Operationen kategorisiert werden, die sie durchführen. Ohne Anspruch auf Vollständigkeit werden hier einige aufgeführt:

Filter besitzen exakt einen Eingangsstrom und $n >= 1$ Ausgangsströme. Jeder Ausgangsstrom i besitzt eine Bedingung b_i und jedes eingehende Ereignis wird an jeden Ausgangsstrom weitergeleitet, dessen Bedingung es erfüllt. Aufspalter und Duplikatoren sind Sonderfälle des Filters. *Aufspalter* besitzen mehrere Ausgangsströme (d. h. $n > 1$) und alle Bedingungen sind disjunkt, aber vollständig. Als Resultat wird jedes eingehende Ereignis an exakt einen ausgehenden Strom weitergeleitet. *Duplikatoren* besitzen ebenfalls mehrere Ausgangsströme, alle Bedingungen sind jedoch Tautologien, was dazu führt, dass ein eingehendes Ereignis an alle ausgehenden Ströme weitergeleitet wird.

Gruppierer besitzen exakt einen ein- und exakt einen ausgehenden Ereignisstrom. Basierend auf Gruppierungscharakteristika wird aus mehreren eingehenden Ereignissen ein Gruppenereignis erzeugt. Mögliche Gruppierungscharakteristika sind vielfältig und können auf Ereignisquantität und -inhalt, aber auch auf Zeiträumen basieren. Dieser Vorgang enthält spezielle Komplexität, da er zustandsbehaftet agiert.

Ableiter erzeugen aus eingehenden Ereignissen neue Ereignisse. Sie besitzen exakt einen ausgehenden Ereignisstrom, können aber unter Umständen mehrere eingehende Ströme besitzen. Aufgrund ihrer Diversität werden sie weiter aufgegliedert: *Übersetzer* entfernen, verändern oder fügen Attribute hinzu. Übersetzer arbeiten dabei immer zustandslos.

Ein *Anreicherer* modifiziert oder ergänzt Attribute mithilfe externer Daten. Dies beinhaltet insbesondere die Möglichkeit, diesen Vorgang Zustandsbasiert auf Grundlage vorangegangener Ereignisse zu tun. Anreicherer können daher mehrere eingehende Ereignisströme besitzen. *Aggregatoren* führen Aggregationsoperationen auf Gruppenereignissen aus. Dies können beispielsweise übliche Aggregationen wie Maximum-, Durchschnitts- oder Anzahlberechnungen, aber auch die Erkennung von Mustern sein. *Kompositoren* besitzen mehrere Eingabeströme und erzeugen aus deren Ereignissen komponierte Ereignisse. Die Kompositionslogik ist dabei zustandsbehaftet und kann aufgrund der vielen zu beachtenden Faktoren äußerst komplex werden.

7.4.2.2 Motivation für die Nutzung von SandBlocks

In der Einleitung dieses Kapitels wurde bereits beschrieben, warum dedizierte Ereignisverarbeitungssysteme genutzt werden. Mit diesem Abschnitt wird nun erläutert, warum die Nutzung einer SandBlocks-artigen heterogenen visuellen Programmiersprache zur Beschreibung von Ereignisverarbeitungsnetzen sinnvoll ist.

Modelle von Ereignisverarbeitungsnetzen sind bereits der Standard zur Spezifikation, Dokumentation und Kommunikation solcher Systeme. Systeme wie IBM Streams[10] (s. Abschnitt 7.6) oder Oracle CEP[11] setzen visuelle Ereignisverarbeitung bereits um. Dies begründet sich zum einen darin, dass ein großer Teil derjenigen, die die Ereignisverarbeitungsnetze definieren, einen weniger programmiertechnischen Hintergrund besitzen und somit von der höheren Abstraktionseben profitieren [37, Kapitel 12].

Zum anderen besitzen ereignisgetriebene Programme eine einzigartige Eigenschaft: Der Programmfluss wird nicht, wie üblich, an einer Stelle, zum Beispiel durch den Start des Programms oder durch eine eingehende Anfrage angestoßen. Stattdessen gibt es viele (Ereignis-)Quellen, die Programmfluss und Verarbeitung anstoßen können und deren Verarbeitung dabei gegenseitig voneinander abhängt. Diese einzigartige Netzstruktur kann aufgrund ihrer Nichtlinearität (strukturell)

[10] IBM Streams: `https://ibmstreams.github.io/`, letzter Zugriff am 11. Dezember 2019.

[11] Oracle CEP: `https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/overview.htm#CEPGS106`, letzter Zugriff am 11. Dezember 2019.

besonders schlecht textuell beschrieben werden, da sie sowohl mehrere Eingabestellen als auch mehrere Ausgabestellen besitzt.

Eine Einbindung der Ereignisverarbeitung speziell in SandBlocks und ähnliche Systeme bietet jedoch noch weitere Vorteile: Da das dedizierte Ereignisverarbeitungssystem in eine generelle Programmiersprache eingebettet wird, können bereits im System komplexere Berechnungen getätigt werden. Die verfügbaren Operationen sind nicht auf die eines abgespaltenen Systems beschränkt.

Zudem entfällt der Aufwand, die Distanz zwischen dem Ereignisverarbeitungssystem und seinem Backend zu überwinden. Ereignisverarbeitung wird daher auch kleineren Anwendungen zugänglich gemacht, die diese nutzen können, ohne dass ein großes System aufgesetzt werden muss. Die Einbindung in eine Liveumgebung erlaubt es zudem, am laufenden System Änderungen vorzunehmen.

7.4.2.3 Grundgerüst für Ereignisverarbeitung

Um eine Implementierung eines visuellen Ereignisverarbeitungsprogramms zu ermöglichen, soll ein Grundgerüst für Ereignisverarbeitung gegeben sein. Eine mögliche Implementierung[12] sei hier nun beschrieben.

Für Ereignisströme sei eine Klasse `EventStream` gegeben, auf den sich Produzenten über `addAdvertiser:` als Eingabe und Konsumenten per `addSubscriber:` als Ausgabe eintragen können. Diese Ströme besitzen Namen als Bezeichner, um bei mehreren ein- oder ausgehenden Strömen eine Identifikation zu ermöglichen. Diese sind über `identifier` zugreifbar. Eingetragene Produzenten können über `put:` Ereignisse an den Strom übergeben. Dieser benachrichtigt dann eingetragene Konsumenten per `notify:from:` über das neue Ereignis und dessen Quellstrom.

Auch für Ereignisproduzenten, -konsumenten und -verarbeitungsagenten seien Grundklassen vorgegeben, respektive die Klassen der Schnittstellen (bzw. Traits) `TEventConsumer`, `TEventProducer`, `TEventProcessingAgent`. Um einheitliche Schnittstellen zu erstellen, erbt die Ereignisverarbeitungsagentenklasse von den Produzent- und Konsumentklassen.

Produzenten können sich mithilfe von `advertiseTo:` mit einem Ereignisstrom verbinden. Mit `produce:to:` kann eine Ereignis auf einen ausgehenden Strom, mit `produce:` auf alle ausgehenden Ströme produziert werden.

Konsumenten können sich mithilfe von `subscribeTo:` mit einem Ereignisstrom verbinden. Wie oben beschrieben werden sie mit `notify:from:` über eingehende Ereignisse informiert. Standardmäßig wird dann `consume:` auf dem nächsten Ereignis des Stroms ausgeführt. Sowohl das Benachrichtigungs- als auch das Konsumierverhalten sind jedoch anpassbar.

Entitäten haben global verfügbare Bezeichner. Diese bieten so Zugriff auf Produzenten (`AvailableEventProducers`) und Konsumenten (`AvailableEventConsumers`), welche dadurch Systemkomponenten darstellen.

[12] Es handelt sich hierbei um die Beschreibung einer Smalltalk-Implementierung, auf der sowohl die SandBlocks-Implementierung, als auch die textuelle Vergleichsimplementierung der folgenden Abschnitte aufbauen.

Ereignisverarbeitungsagenten kombinieren die Schnittstellen von Produzenten und Konsumenten, um eine einheitliche Implementierung der Ereignisströme zu ermöglichen.

Das Grundgerüst neben der Schnittstellenklasse bietet dabei eine Klasse, welche eine möglichst generische Implementierung der Verarbeitungsagentenschnittstelle umsetzt: `PluggableEventProcessingAgent`. Dazu enthält diese Klasse eine *optionale Aktion* für die Reaktion auf eine Benachrichtigung eines eingehenden Ereignisses und eine *zwingende Aktion* für das Verhalten beim Konsumieren eines Ereignisses.

7.4.2.4 Allgemeiner Lösungsansatz

Ereigniverarbeitungsnetze stellen Graphen mit Ereignisproduzenten, -konsumenten und -verarbeitungagenten als Knoten und Ereignisströmen als Kanten dar. Es bietet sich daher an, diese dementsprechend (als Graphen) darzustellen.

Ereignisproduzenten und Konsumenten können entweder Systemkomponenten oder auf den Kontext eines Netzes spezifische Komponenten sein. Beim Hinzufügen einer dieser Knoten kann dies ausgewählt werden und wird auch visuell markiert.

Ereigniskonsumenten können in mehreren Formen hinzugefügt werden. Auf Grundlage des Grundgerüsts aus Abschnitt 7.4.2.3 ist eine generelle Form, welche die Definition der zwei Aktionen in zwei getrennten Eingabefeldern erlaubt vorhanden. Die Freiheit der visuellen Strukturierung zahlt sich hierfür aus. Zudem sind für die in Abschnitt 7.4.2.1 beschriebenen EPA-Typen eigene, in ihren Eingaben angepasste Knoten vorhanden.

Die Einbindung der Netze ist der jeweiligen Implementierung überlassen. Wichtig ist jedoch, dass das Netz immer aktiv ist, das heißt, wann immer ein Ereignis in einem Produzenten auftritt, kann das Netz reagieren. Daher bietet sich eine Einbindung in ein Live-System wie Squeak an, in dem das Netz dauerhaft im Objektraum verbleibt. Zu beachten ist jedoch, dass das Netz zur Bearbeitung immer zugreifbar ist.

Zudem sollen die Netze schachtelbar sein. Dies ermöglicht die Nutzung von Ereignisbasiertheit und eine sauberere Problemkomposition auf mehreren Abstraktionsebenen. Dazu müssen die Netze nur die Schnittstelle des EPA implementieren und lokale Produzenten und Konsumenten zulassen. Erhält ein Netz in seiner Funktion als EPA eine Benachrichtigung für ein eingehendes Ereignis, so simuliert er dies durch eine Produktion seines passenden lokalen Produzenten. Passend wird hierbei über den Namen des eingehenden Stroms und des Produzenten bestimmt. Konsumiert ein lokaler Konsument ein Ereignis, so gibt er dieses wiederum nach außen an sein Netz weiter. Dieses produziert dann das jeweilige Ereignis in seiner Funktion als Agent auf seinen passenden ausgehenden Strom.

7.4.2.5 Implementierung in SandBlocks

Das SandBlocks-Ereignissystem setzt Ereignisverarbeitungsnetze als Graph-Block um. Zudem implementiert dieser Blocktyp die Schnittstelle der Ereignisverarbeitungsagenten. Wie spezifiziert, werden Ereignisströme auf die Kanten und Ereig-

nisproduzenten, -konsumenten und -verarbeitungsagenten auf drei Knotentypen des Blockes abgebildet.

Agenten sind dabei wiederum als Blöcke implementiert. Sie können daher als Methoden eingebunden und somit über einen Bezeichner identifiziert werden. In einem Netz kann dann aus den verfügbaren Agenten ausgewählt werden, wenn ein neuer Agent hinzugefügt wird. Dies hat den Vorteil, dass die Implementierungsdetails eines Agenten hinter einem Bezeichner versteckt werden können und somit die Darstellung des Netzes nicht überladen wird. Aktuell ist nur eine Nutzung des vorgegebenen `PluggableEventProcessingAgent` möglich. In Zukunft sind jedoch weitere Knotentypen mit verbesserter Nutzung, z. B. dedizierten Reaktionen auf einzelne Ströme, strukturierteren Eingabemöglichkeiten u. Ä., geplant.

EPAs können zwar als oder in Methoden in das System eingebunden werden, über `valueInContext:` aufgerufen führt das jedoch nicht zur Ausführung ihrer Verarbeitungslogik. Stattdessen geben sich die Agenten selbst zurück. Wie oben beschrieben besitzen Ereignisverarbeitungsnetze und -agenten im Gegensatz zu klassischen Methoden keinen einzelnen Startpunkt für ihre Ausführung. Vielmehr wird ihre Ausführung dezentral über ein eingehendes Ereignis durch einen Produzenten oder einen eingehenden Ereignisstrom angestoßen. Das Netz oder der Agent verarbeitet dann das eingehende Ereignis und kehrt in seinen „passiven" Zustand zurück. Dies ist in SandBlocks möglich, da in Squeak/Smalltalk als Livesystem Objekte keine Laufzeit besitzen, sondern, solange es Referenzen auf sie gibt, im Speicher verbleiben. Wichtig für einen Netz. B.ock ist dann jedoch, dass es von ihm nur eine Version gibt. Daraus resultiert, dass beim Bearbeiten keine Arbeitskopien angelegt werden dürfen oder sollen. Somit ist es möglich, bei der Bearbeitung des Netzes direkt Änderungen am System vornehmen zu können.

Da der Netz. B.ock das Interface eines Ereignisverarbeitungsagenten implementiert, ist es möglich, Netz. B.ockes ineinander zu verschachteln. Dazu wird das innere Netz als Methode eingebunden und ist dadurch über seinen Bezeichner für das äußere Netz verfügbar.

7.4.2.6 Beispielspezifikation und -umsetzungen

Mithilfe des Graph-Blocks wurde ein Ereignisverarbeitungssystem durch visuelle Metaphern implementiert. Um im Allgemeinen die Vorteile einer solchen Implementierung herauszuheben, wird diese nun anhand eines Beispiels mit rein textueller Ereignisverarbeitung verglichen.

Beispielspezifikation Als Beispielszenario für eine Ereignisverarbeitungsnetz sei hier die vereinfachte Steuerung einer programmierbaren Flugdrohne gegeben, ein klassischer Anwender von Sensortechnik. Der Nutzer soll die Möglichkeit haben, die Steuerungslogik der Drohne selbst zu implementieren. In diesem Beispiel soll vom Nutzer eine Position geschickt werden können, an der die Drohne windfest stehen bleibt, bis sie neue Koordinaten erhält. Gegeben seien drei Sensoren, für die Windgeschwindigkeit (gerichtet), die Höhe und die GPS-Position. Gemeinsam mit den Nutzereingaben ergeben sich hiermit vier Ereignisquellen. Die erhaltenen

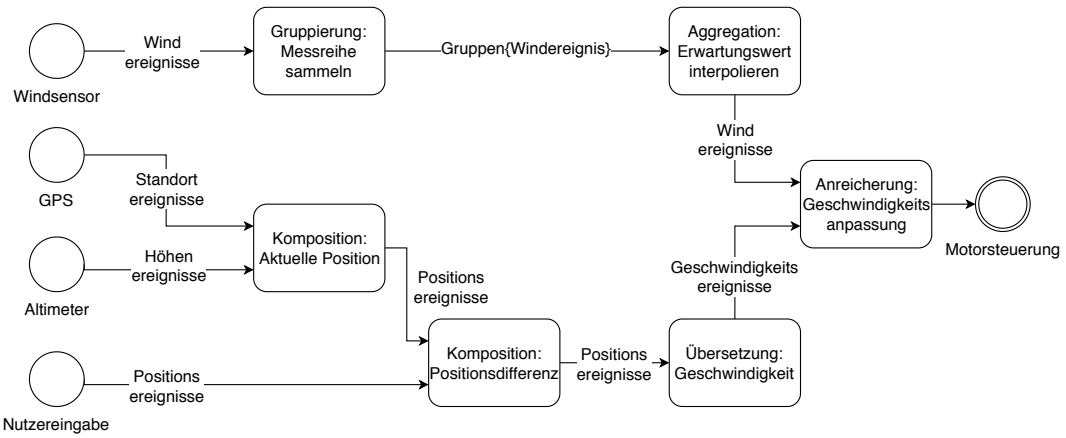


Abbildung 7.5: Skizzierung des Beispiels als Diagramm. Quellen und Senken sind als Kreise, EPAs als Rechtecke dargestellt.

Daten sollen für die Geschwindigkeitsanpassung der Motorsteuerung verarbeitet werden, welche somit als Ereigniskonsument agiert.

Um Stabilität im Wind zu gewährleisten, soll die zukünftige Windgeschwindigkeit anhand der vorhergegangenen fünf Messwerte prognostiziert werden, immer wenn ein Messpunkt eintrifft. Die Ergebnisse des Höhen- und Positionssensors müssen kombiniert werden, um dann eine Differenz zu der vom Nutzer spezifizierten Position berechnen zu können. Diese wird dann in eine gerichtete Zielgeschwindigkeit für die Drohne umgesetzt. Jedes Zielgeschwindigkeitsereignis wird um die aktuelle Windinformation angereichert und an die Motorsteuerung als Ereigniskonsument weitergegeben.

Für die Drohne ist eine Klasse `Drone` gegeben, in der mithilfe der Methode `eventProcessingNet` das entsprechende Netz konstruiert werden soll.

Probleme textueller Umsetzung Listing 7.2 zeigt die beispielhafte Umsetzung des Szenarios mit dem gegebenen System in Smalltalk. Der `-->` Operator ist zur besseren Lesbarkeit aufseiten der Ereignisproduzenten und -ströme implementiert und konstruiert einen Ereignisstrom vom Empfänger zum Parameter. Er gibt zudem den Ereigniskonsumenten zurück, sodass eine Verkettung möglich ist. Für die einzelnen Ereignisverarbeitungsagenten sind hier Methoden gegeben, die über den internen Zustand des Objekts Auskunft geben. Von der Implementierung dieser Methoden sei hier abstrahiert.

Vergleichbar zu Abschnitt 7.4.1.5 ist auch diese textuelle Umsetzung messbar komplex. Zwar müssen keine temporären Variablen für Aktivitätsergebnisse und durch die Verkettung des Pfeiloperators nur für Agenten mit mehreren eingehenden (oder ausgehenden) Ereignisströmen angelegt werden, eine Verbesserung der Lesbarkeit wurde jedoch kaum erreicht. Die Methode ist mit knapp 20 Zeilen äußerst lang, was an der großzügigen Nutzung von Zeilenumbrüchen liegt.

Listing 7.2: Beispiel umgesetzt in Smalltalk

```
AerialDrone >> #eventProcessingNet
   | composePosition composePositionDifference enrichSpeedAdaption |
   composePosition := self composePosition.
   composePositionDifference := self composePositionDifference.
   enrichSpeedAdaption := self enrichSpeedAdaption.
   AvailableEventProducers windSensor
      --> #windSpeed --> self groupTestSeries
      --> #groupedWind --> self aggregateInterpolatedSpeed
      --> #expectedWind --> enrichSpeedAdaption.
   AvailableEventProducers gpsSystem
      --> #location --> composePosition.
   AvailableEventProducers altimeter
      --> #altitude --> composePosition
      --> #position --> composePositionDifference.
   AvailableEventProducers droneUser
      --> #targetPosition --> composePositionDifference
      --> #positionDelta --> self translateSpeed
      --> #speed --> enrichSpeedAdaption
      --> #speed --> AvailableEventConsumers engineControl.
```

Reduzierte man diese jedoch, stiegen die Zeilenlängen auf einen inakzeptablen Wert.

Durch die Verkettung des Pfeiloperators lassen sich lineare Kausalketten, wie die vom Drohnennutzer zur Motorsteuerung, gut darstellen. Sobald jedoch Nichtlinearität dazu kommt, muss der Bezeichner eines Agenten an mehreren Stellen im Code auftauchen. Somit werden die Beziehungen dieses Agenten schwerer ersichtlich und leicht zu übersehen, komplexere Kausalitäten und Abhängigkeiten lassen sich also schlecht darstellen.

In einem komplexeren System das höhere Kantenzahlen besitzt, tauchen Knotenbezeichner häufig mehr als zweifach im Code auf. Dann sind die Beziehungen eines Knotens noch schwieriger ersichtlich.

Zur Duplikation der Bezeichner kommt zusätzlich noch eine strukturelle Duplikation zur Erschaffung der Ereignisströme. Auch wenn diese geringer ist als in Abschnitt 7.4.1.5, so vermischen sich hier doch Code, der zur Beschreibung der Programmstruktur, und Code, der zur Beschreibung des Programminhalts verantwortlich ist.

Neben der Komplexität des Programmcodes ergeben sich beim Beispiel der Ereignisverarbeitung konzeptionelle Limitierungen. Eine Anpassung des Netzes ist nur über eine Änderung des Quelltextes und eine erneute Ausführung der Methode möglich. Dabei wird ein komplett neues Netz konstruiert, sodass das alte Netz gelöscht und aus dem System entfernt werden muss. Bei einem Entfernen oder Hinzufügen eines Knotens muss aufwendig die Differenz zwischen altem und neuem Netz ermittelt werden. Agenten können zwar durch die Grundimplementierung angepasst werden, ohne neue Objekte zu erzeugen, ein Zugriff auf sie ist aber schwer, da dafür eine Referenz auf das Netz nötig wäre. All dies hemmt das Entwickeln und Debugging mit kurzer Feedbackschleife stark.

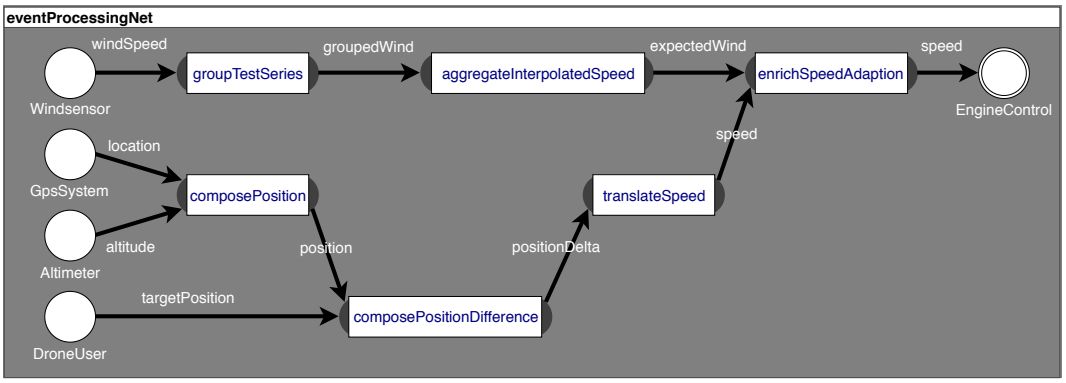


Abbildung 7.6: Eine beispielhafte Umsetzung in SandBlocks

Zudem muss Quelltext, um Änderungen an einer Stelle des Netzes (beispielsweise einem Agenten) vorzunehmen, eventuell an mehreren Stellen geändert werden. Dies verletzt das Prinzip der eindeutigen Verantwortlichkeit[13] auf Codeebene.

Visuelle Umsetzung Abbildung 7.6 zeigt eine Implementierung des Beispiels mithilfe des Netz. B.ockes. Im Vergleich zur textuellen Umsetzung ist der Text auf die Benennung der Akteure beschränkt. Kein Text ist für die Beschreibung der Strombeziehungen nötig. Der visuelle Code stellt also den Ereignisfluss, der textuelle Code die Ereignisverarbeitung dar. Somit liegt auch hier eine Trennung der Verantwortlichkeiten vor. Auf welche Ereignisströme wann produziert wird unterliegt jedoch noch den Knoteninhalten.

Alle Informationen zu einem Akteur, beispielsweise einem EPA sind an genau einer Position zu finden, namentlich seinem Knoten. Insbesondere mehrere ein- oder ausgehende Ströme eines Knotens sind nun ersichtlich.

Das erzeugte Layout des Graphen ähnelt stark dem der Problembeschreibung aus Abbildung 7.5, auch wenn es durch die Länge der Methodennamen verzerrt ist. Dies deutet auf eine starke Direktheit der Abbildung der Problemdomäne hin.

Da das Netz als Block-Methode in das System integriert wurde, handelt es sich bei der Darstellung um eine „lebendige“ Ansicht auf das tatsächliche Netz. Das bedeutet insbesondere, dass über die Werkzeuge direkte Änderungen am Netz gemacht werden können: Der Text der Knoten, und damit auch direkt die Verarbeitungslogik eines Agenten kann direkt geändert werden, beim Speichern des jeweiligen Textfeldes wird diese Änderung direkt in das Netz übernommen. Über Kontextmenüs der Knoten und des Netzes ist es möglich, direkt neue Knoten hinzuzufügen oder zu löschen.

[13] Dieses Prinzip wird häufig seiner englischen Übersetzung folgend „Single Responsibility“ genannt.

7.5 Evaluation

In diesem Abschnitt werden allgemeine Erkenntnisse aus den Beobachtungen der in Abschnitt 7.4 beschriebenen Anwendungen einer HVPL zur Beschreibung nichtlinearer Programmstruktur abgeleitet. Dies beinhaltet sowohl nebenläufige Programmierung als auch Ereignisverarbeitung.

7.5.1 Systemverantwortlichkeit

Mithilfe des Graphen für Nebenläufigkeit wird dem Programmierer der Aufwand abgenommen sich auf niedriger Ebene um die Prozessverwaltung seines nebenläufigen Programms kümmern zu müssen. Dies hat mehrere Vorteile: Zum einen fällt der Faktor Mensch als Fehlerquelle für inkorrekte Prozesssynchronisation weg. Alle Forks und Joins werden durch Konstrukte der visuellen Sprache erzeugt, wodurch sichergestellt werden kann, dass es sich um valide Operationen handelt. Zeit, die durch Fehlersuche und -behebung in diesem Bereich verbraucht würde, wird dadurch frei. Zum anderen kann sich der Entwickler durch den Wegfall des Prozessverwaltungsaufwandes auf die funktionale Komposition und Parallelisierung seines Problems fokussieren. Dies fördert die Entwicklung strukturell besserer Programme [34]. Für den Graphen für Ereignisverarbeitung ist der Aspekt der Fehlervermeidung durch Systemverantwortlichkeiten weniger stark ausgeprägt. Trotzdem verhindert dieser Graph die Konstruktion invalider Ereignisströme und vereinfacht die Erstellung dersolchen, sodass eine Konzentration auf Erstellung und Konfiguration der Ereignisverarbeitungsagenten möglich ist.

Im Allgemeinen lässt sich also beobachten, dass durch die visuelle Codierung nichtlinearen Programmflusses Verantwortlichkeit an das System abgegeben wird. Dadurch entsteht Sicherheit für den Programmierer, der in diesen Bereichen keine Fehler mehr erzeugen kann, was es ihm erlaubt, sich stärker auf inhaltliche Aspekte seines Programmes zu konzentrieren. Somit können besser durchdachte und damit strukturierte Programme erzeugt werden.

7.5.2 Trennung der Verantwortlichkeiten

Wie bereits beobachtet, führt die Implementierung des Graphen als reine Strukturebene zu einer Trennung der Verantwortlichkeiten zwischen Text und Graphen. Die textuelle Ebene ist hier für den Programminhalt, die visuelle Ebene für die Programmstruktur zuständig. Dies spiegelt sich auch in der bereits beschriebenen Trennung der Verantwortlichkeiten zwischen Entwickler und System wider, zwischen Problembeschreibung und Implementierung der Nichtlinearität wie Nebenläufigkeit oder Ereignisfluss. Diese Trennung erhöht zudem aber auch die Übersichtlichkeit und Lesbarkeit des Programmes. Durch Wegfall des Programmtextes zur strukturellen Verwaltung bleibt weniger Text übrig, der damit besser verständlich ist. Insbesondere, da der übrig gebliebene Text rein zur Inhaltsbeschreibung genutzt wird. Bei Betrachtung des Textes beispielsweise eines Nebenläufigkeitsgraphen wird ersichtlich, welche Routinen und Aktivitäten ausgeführt werden sollen,

unabhängig von der gewählten Parallelisierung. Bei einem Ereignisverarbeitungsgraphen wird die Art der Ereignisverarbeitung deutlich.

Auf der anderen Seite lässt sich die Programmstruktur besser überblicken, da der Graph aufgrund der direkteren Abbildung eine bessere Darstellung umsetzt. Abhängigkeiten, die sich auf einen einzelnen Akteur, zum Beispiel eine Aktivität oder einen Ereignisverarbeitungsagenten, beziehen, befinden sich durch die visuelle Metapher an einer einzelnen räumlichen Stelle des Codes und sind daher besser ersichtlich. Will also ein Leser den Programminhalt verstehen, muss er sich rein auf den Text konzentrieren, will er die Struktur verstehen, muss er sich rein auf die Graphenstrukturierung konzentrieren. Somit werden Lesbarkeit und Verständlichkeit verbessert. Dies erleichtert auch die Manipulation der beiden Programmaspekte: Soll der Inhalt oder die Struktur angepasst werden müssen nur der Text *oder* Graph angefasst werden. Insbesondere ist es nun möglich, die Programmstruktur direkt zu manipulieren, anstatt indirekt über textuellen Code.

7.5.3 Direktere Problemabbildung

Wie insbesondere das Beispiel der Ähnlichkeit von Beispielmodell und -umsetzung der Flugdrohnensteuerung aus Abschnitt 7.4.2.6 zeigt, kann visueller Code seine Problemdomäne direkter abbilden, wenn Nichtlinearität Teil der Problembeschreibung ist. Dies drückt sich beispielsweise aber auch durch die beschriebene Sammlung aller Informationen zu einem Akteur an exakt einer Stelle aus.

Die direktere Abbildung hat einige positive Konsequenzen auf die Entwicklung visuellen Codes: Zunächst sorgt die Nähe zu der Domäne für ein besseres Verständis des Codes. Hierbei sei auf Kapitel 2 verwiesen, in dem ausführlich die Vorzüge visueller Programmierelemente erläutert werden. Visuelle Elemente eignen sich insbesondere um Strukturen und Muster darzustellen. Im Gegensatz zu textueller Codierung, stehen visueller Codierung zwei geometrische Dimensionen zur Anordnung von Aktivitäten zur Verfügung. Dabei kann eine Dimension genutzt werden, um Sequenz darzustellen und eine, um Nebeneinander zu codieren. Die in Abschnitt 7.4 beschriebenen Graphen dokumentieren sich zu einem großen Teil selbst, insbesondere ihre Struktur. Selbstdokumentierender Code ist dabei eine generell äußerst wünschenswerte Eigenschaft von Software, da er das Verständnis und die Wartbarkeit enorm erhöht.

Direkte Strukturabbildung erlaubt auch direkte Strukturmanipulation und direkte Manipulationen auf Problemebene. Ereignisverarbeitungsnetzgraphen erlauben beispielsweise das direkte Hinzufügen von Produzenten, Konsumenten und EPAs. Dies verbessert die Entwicklung und Debugging enorm, da mit geringem Aufwand Änderungen am System vorgenommen und evaluiert werden können. Zusätzlich zur Manipulation wird auch eine strukturelle Analyse der Programme vereinfacht, da die den visuellen Elementen zugrundeliegenden Modelle eine automatische Formalisierung der Struktur bieten.

7.5.4 Eingeschränktheit visueller Sprachen

Visuelle Sprachen bieten, dem Erhalt ihrer Verständlichkeit geschuldet, häufig nur eine geringe Zahl visueller Metaphern und damit begrenzte Ausdrucksmöglichkeiten. Für rein visuelle Sprachen kann diese mangelnde Ausdruckskraft höchst problematisch werden. Ein hybrider Ansatz erlaubt jedoch die Nutzung der Mächtigkeit textuellen Codes wo nötig. Die Einschränkungen des visuellen Codes können dann konstruktiv genutzt werden: Innerhalb des Graphen für Nebenläufigkeit wird beispielsweise nur das Warten auf alle eingehenden Abhängigkeiten und das Signalisieren aller ausgehenden Abhängigkeiten unterstützt. Im positiven Sinne führt die Einschränkung visueller Metaphern dazu, dass das System nicht nur als syntaktischer Zucker für die Vereinfachung der Aktivitäts- und Prozessverwaltung agiert, sondern auch als syntaktisches Salz. Würde beispielsweise der Entwickler selbst für die Aktivierung der einzelnen Kanten zuständig sein, bräche dies die Trennung von Struktur und Inhalt. Die Komplexität der Entwicklung würde stark ansteigen. Durch die Implementierung exakt eines Fork- und Join-Verhaltens bleibt sowohl die Nutzung als auch das Verständnis der Graphen vorhersehbar und unkomplex. Diese Simplizität spielt auch eine positive Rolle für die Qualität der Programmstruktur, da für die Nebenläufigkeit eine höhere Ebene der Abstraktion genutzt wird.

Im Allgemeinen können die Einschränkungen des visuellen Anteils einer HVPL, insbesondere in Domänen, in denen ein besonders strukturierter Programmieransatz nötig ist, dazu genutzt werden, um die Nutzung ebenjener Strukturen zu ermutigen und zu forcieren. Resultierende Programme sind daher besser strukturiert und damit auch verständlicher.

–

Insgesamt führt die Nutzung visueller Elemente zur Codierung nichtlinearen Programmflusses somit zur Entwicklung besser strukturierter, besser lesbarer und besser wartbarer Programme.

7.6 Verwandte Arbeiten

Im folgenden Abschnitt wird die Umsetzung nichtlinearen Programmflusses mithilfe visueller Programmierelemente, wie sie in diesem Kapitel beschrieben wird, in den Kontext verwandter Arbeiten übertragen. Hierbei werden insbesondere Referenzen für ähnliche Ansätze und vertiefende Evaluation gegeben.

7.6.1 Klassifikation visueller nebenläufiger Sprachen

„Visual Programming to Support Parallel Design“ [34] befasst sich weiterführend mit visuellen Ansätzen zur nebenläufigen Programmierung im Allgemeinen. Im Rahmen dessen wird eine Möglichkeit zum Vergleich visueller Sprachen für nebenläufige Programmierung mithilfe von vier Aspekten vorgestellt. Anhand dieser

vier Aspekte, *Anwendbarkeit*, *Ausdruckskraft*, *Freundlichkeit* und *Portabilität*, wird nun auch der in Abschnitt 7.4.1.3 beschrieben Graph für Nebenläufigkeit in den allgemeinen Kontext visueller Sprachen zur Nebenläufigkeit eingeordnet.

Anwendbarkeit gibt an, in welchem Rahmen potenzieller Anwendungen mit einem System entwickelt werden kann. Der große Vorteil des Graphensystems ist, dass ein Aufruf aus und eine Rückführung zu der Ursprungssprache problemlos möglich sind. Die resultierende Flexibilität erlaubt die Entwicklung einer großen Zahl an Anwendungen.

Ausdruckskraft gibt an, welche Menge algorithmischer Entwurfskonzepte ein System enthält. Aufgrund des Fehlens komplexerer Strukturen als Fork und Join muss für viele Algorithmen auf Rekursion zurückgegriffen werden, was die Ausdruckskraft senkt. Insbesondere Grobstrukturen von Programmen und Routinen lassen sich jedoch sehr gut ausdrücken.

Freundlichkeit misst den Aufwand, den der Nutzer betreiben muss, um das nebenläufige System aufzubauen. Dies ist der Aspekt, auf den der Graphen ausgelegt ist. Durch die Integration in die existierende Sprache ist das Erschaffen eines Kontexts für Nebenläufigkeit kein großer Aufwand. Der Graph ist aufgrund seiner Simplizität zudem gut verständlich.

Portabilität bestimmt, wie gut sich mit einem System entwickelte Anwendungen auf andere Rechner übertragen lassen. Dies hängt beim Graphen stark von der tatsächlichen Implementierung ab. Da die SandBlocks Implementierung auf Smalltalk aufsetzt, besitzt sie bereits eine natürliche Betriebssystemportabilität. Um ein Programm zu transferieren müssen jedoch im Zielsystem sowohl SandBlocks als auch seine Graphimplementierung vorhanden sein.

7.6.2 Verwandte Programmiersprachen und -systeme

Luna *Luna*[14] ist eine graphbasierte Datenverarbeitungs- und Datenvisualisierungsumgebung. Luna stellt seine Komponenten als Datenflussgraphen dar, wobei die Knoten Datenquellen, Senken oder Verarbeitungsschritte darstellen. Dies ähnelt stark der Ereignisverarbeitungsnetzstruktur von Ereignisproduzenten, -konsumenten und -verarbeitungsagenten. Zudem besitzt Lunas Datenverarbeitungsgraph eine natürliche Nebenläufigkeit. Im Gegensatz zu SandBlocks, dass keine textuelle Repräsentation seiner Blöcke führt, bezeichnet sich Luna als Duale Sprache, d. h. für die visuellen Elemente existiert eine textuelle Repräsentation und umgekehrt.

Luna stellt eine professionelle Umsetzung des Konzeptes, visuelle Elemente zur Codierung von Programmstruktur zu nutzen, dar. Dabei handelt es sich um ein recht junges Projekt, das sich noch in einer frühen Phase befindet. Daher liegen noch keine Daten zur Nutzung vor. Die Entwicklung dieser Sprache wird jedoch im Kontext dieses Kapitels mit Spannung verfolgt.

[14] Luna: `https://www.luna-lang.org`, letzter Zugriff am 11. Dezember 2019.

IBM Streams *IBM Streams*[15], ist ein Entwicklungssystem für Datenstromverarbeitung. Insbesondere eignet sich Streams auch für Ereignisstromverarbeitung. Streams stellt dabei eine Vielzahl an Möglichkeiten zur Verfügung, Datenstromgraphen zu definieren. Dazu gehören neben programmiersprachlicher Erstellung z. B. über Java und Python und einer eigenen *Stream Processing Language* auch eine visuelle graphbasierte Sprache. Diese wird jedoch eher zur Konfiguration der Netze genutzt, die aus der textuellen Beschreibung resultieren.

Stream ist jedoch insbesondere interessant, weil es sich sowohl um eine Nutzung visueller Codierung für Programmstruktur auf sehr hoher Ebene handelt als auch um, nach eigener Aussage von IBM, *„einen Führer des Marktes für Datenstromanalytik."*[16]

7.7 Zusammenfassung und Ausblick

In diesem Kapitel wurde die Nutzung visueller Metaphern zur Codierung nichtlinearen Programmflusses vorgestellt. Nachdem das SandBlocks-Projekt zur Integration visueller Elemente in eine textuelle Sprache vorgestellt wurde, wurde gezeigt, dass textueller Programmcode nur schwer auf nichtlinearen Programmfluss abbildet. Vor diesem Hintergrund wurden die Domänen der Nebenläufigkeit und Ereignisverarbeitung vorgestellt, die nichtlinearen Programmfluss benötigen. Mithilfe des vorgestellten Projekts wurden dann Beispiele dieser Domänen umgesetzt und mit textuellen Umsetzungen verglichen. Beobachtet wurden insbesondere bessere Lesbarkeit und eine Trennung der Codierungen von Programmstruktur und Programminhalt auf die Ebenen der visuellen und textuellen Codierung. Aus diesen Beobachtungen wurden folgender Schluss gezogen:

Da nichtlineare Programme sich nur schlecht durch klassischen textuellen Code darstellen lassen, führt eine visuell-textuell hybride Codierung in den gewählten Beispieldomänen zur Entwicklung besser verständlicher, besser strukturierter und besser wartbarer Programme.

Damit zeigt sich, dass die Codierung nichtlinearen Programmflusses mithilfe visueller Metaphern den konzeptionellen Spalt zwischen Programmcode und beschriebenem Programm verringern kann.

Insgesamt bieten visuelle Elemente eine praktikable Alternative zur textuellen Codierung von Programmfluss. Trotzdem ist ihre Nutzung kaum verbreitet. Das SandBlocks Projekt stellt eine gute Grundlage dar, dies durch schnelle und einfache Entwicklung neuer visueller Metaphern zu ändern.

[15] IBM Streams: `https://ibmstreams.github.io/`, letzter Zugriff am 11. Dezember 2019.

[16] Übersetzung des Autors. Beschreibung von IBM Streams: `https://www.ibm.com/cloud/blog/deepen-and-expand-business-insights-through-ibm-streams-v4-3`, letzter Zugriff am 11. Dezember 2019.

Ausblick

Einige Aspekte sind in der Implementierung des Graphen für Nebenläufigkeit aktuell noch nicht betrachtet. Ressourcenzugriff in nebenläufigen Systemen kann zu einigen Problemen der Dateninkonsistenz führen. SandBlocks-Graph bietet keine Schutzmechanismen dagegen. Dieses Problem geht Hand in Hand damit, dass nebenläufige Programme häufig aufgrund des nichtdeterministischen Scheduling-Verhaltens selbst nicht deterministisch sind. Determnismus ist aber in der Softwareentwicklung beispielsweise wichtig für Debugging, Tests und Korrektheitsprüfungen von Programmen [77, S. 3]. Um das Parallelisierungsziel der Leistungsoptimierung zu erfüllen, müssten die Implementierungsdetails optimiert werden, dies war nicht Fokus dieser Arbeit. Als Erkenntnis aus der Evaluation wäre es zudem sinnvoll, in vorsichtigem Maße neue Konzepte in die Graphensprache zu integrieren. Inbesondere neue Knoten zur Selektion und Iteration würden die Ausdruckskraft der Sprache stark verbessern, da viele Programmstrukturen dynamisch erzeugt werden. Auch die Einführung eines dedizierten Rückgabeknotens, welcher Endknoten, auf deren Termination nicht gewartet werden müsste, umsetzbar macht, wäre denkbar. Bei allen Erweiterungen der visuellen Sprache muss jedoch darauf geachtet werden, dass die gewählten zusätzlichen visuellen Metaphern leicht verständlich sind, die Trennung der Verantwortlichkeiten nicht verletzen und die Sprache ihre Leichtgewichtigkeit und Übersichtlichkeit erhält.

Auch der Graph für Ereignisverarbeitung kann um neue Knotentypen erweitert werden: Um die Konfiguration der EPAs leichter zu gestalten, können dedizierte Knotentypen für identifizierte Klassen von EPAs eingeführt werden. Zudem können übliche Konzepte wie die Nutzung von Anfragesprachen wie CQL [8] zur Beschreibung von EPAs testweise in das System integriert werden. Auch die Nutzung eines eigenen Knotentypen für Ereignisströme, die mehrere Ein- oder Ausgänge besitzen (Ereigniskanälen) kann untersucht werden.

Des Weiteren birgt der allgemeine Graph-Block noch Potenzial: Die bereits existierende visuelle Darstellung kann beispielsweise durch Laufzeitinformation angereichert werden. Informationen wie die Anzahl aktiver Instanzen einer Aktivität, der Mittelwert einer Kante, oder die Frequenz eines Ereignisproduzenten könnten so leicht abgefragt werden.

–

Zusätzlich zur Verbesserung der Implementierungen ergeben sich weiterführende Fragestellungen zur Codierung von Programmstruktur mithilfe visueller Metaphern.

Als direktes Resultat der Implementierungen sollte überprüft werden, welche Möglichkeiten die Nutzung visueller Elemente für Werkzeuge eröffnet und welche Konsequenzen sich dadurch für die Programmentwicklung ergeben. Zudem soll untersucht werden, wie sich die Erkenntnisse dieses Kapitels verhalten, wenn die Implementierungen in einem größeren Rahmen, d. h. zur Entwicklung größerer Softwaresysteme, genutzt werden.

Neben den beiden gewählten Beispielen wäre es nötig, zu evaluieren, in welchen weiteren Programmierbereichen sich visuelle Metaphern zur Beschreibung von Nichtlinearität nutzen lassen, um die Allgemeinheit der Erkenntnisse dieses Kapitels aufzuzeigen. Denkbar wäre eine Nutzung für (Datenbank-)Anfragen, für Simulationen oder für verteilte Systeme. Dafür wäre insbesondere relevant, welche weiteren visuellen Metaphern außer einem Graphen genutzt werden können.

Es erscheint zudem sinnvoll, die geringe Verbreitung visueller Metaphern zur Programmierung zu untersuchen, um weitere Forschung und Entwicklung in diesem Bereich anzuregen. Dazu wäre beispielsweise die Durchführung einer Nutzerstudie über Systeme wie SandBlocks möglich.

8 Schlussbetrachtungen

In dieser Arbeit haben wir mit dem SandBlocks-Projekt einen Prototypen beschrieben, der die gemeinsame Nutzung visueller und textueller Programmelemente in einer Programmiersprache ermöglicht. Damit vereint es die Vorteile textueller und visueller Programmiersprachen ohne dem Entwickler vorzuschrieben, wann und wie er visuelle Elemente nutzen muss.

Dafür wurden zunächst visuelle Programmiersprachen analysiert, darauf folgend eine technische Integration in das Squeak/Smalltalk-System beschrieben, Einblicke in die Umsetzung und Verwendung in Live-Programmiersystemen und zuletzt ihre Verwendung in unterschiedlichen Domänen diskutiert.

Die beschriebenen Konzepte und Techniken für die Erstellung von SandBlocks helfen das Programmieren zu erleichtern. Dabei stellen die Vorteile objektorientierter, interaktiver, visueller Live-Programmiersysteme einen relevanten Startpunkt für zukünftige Forschung dar.

SandBlocks wurde unter der MIT Lizenz veröffentlicht.[1]

[1] SandBlock auf GitHub: `https://github.com/hpi-swa-lab/SandBlocks`, letzter Zugriff am 16. Dezember 2019.

Literaturverzeichnis

[1] H. Abelson und G. J. Sussman. *Structure and Interpretation of Computer Programs*. 2. Auflage. MIT Press, 1996. ISBN: 0-262-01153-0.

[2] R. Ahmad. „Visual Languages: A New Way of Programming“. In: *Malaysian Journal of Computer Science* 12.1 (1999), Seiten 76–81.

[3] W. Ali, K. Sultana und S. Pervez. „A Study on Visual Programming Extension of JavaScript“. In: *International Journal of Computer Applications* 17.1 (2011), Seiten 13–19.

[4] H. Alrubaye, S. Ludi und M. W. Mkaouer. „Comparison of Block-based and Hybrid-based Programming Environments in Transferring Programming Skills to Text-based Environment“. Magisterarbeit. Rochester Institute of Technology, University of North Texas, 2019.

[5] J. R. Anderson. *Kognitive Psychologie: Eine Einführung*. Spektrum der Wissenschaft Verlagsgesellschaft, 1989. ISBN: 978-3-893-30703-6.

[6] D. T. Andrew Hunt. *The Pragmatic Programmer - From Journeyman to Master*. Addison Wesley Longman Publishing Co., Inc., 1999. ISBN: 978-0-201-61622-4.

[7] M. Andujar, L. Jimenez, J. Shah und P. Morreale. „Evaluating Visual Programming Environments to Teach Computing to Minority High School Students“. In: *Journal of Computing Sciences in Colleges* 29.2 (2013), Seiten 140–148.

[8] A. Arasu, S. Babu und J. Widom. „The CQL Continuous Query Language: Semantic Foundations and Query Execution“. In: *The VLDB Journal* 15.2 (2006), Seiten 121–142. ISSN: 0949-877X. DOI: 10.1007/s00778-004-0147-z.

[9] O. Auverlot, S. V. Caekenberghe, D. Cassou, G. O. Cotelli, C. Demarey, M. Dias, L. Dolia, S. Ducasse, L. Fabresse, J. Fabry, C. F. Delbecque, N. Hartl, G. Larchevêque, M. Leske, E. Lorenzano, A. Magyar, M. Martinez-Peck und D. Pollet. *Enterprise Pharo*. Square Bracket Associates, 2016. ISBN: 978-1-326-65097-1.

[10] E. Baroth und C. Hartsough. „Visual Programming in the Real World“. In: *Visual Object-oriented Programming: Concepts and Environments*. Herausgegeben von M. M. Burnett, A. Goldberg und T. G. Lewis. Manning Publications Co., 1995. Kapitel 2, Seiten 21–42. ISBN: 0-13-172397-9.

[11] D. Bau, D. Bau, M. Dawson und C. Pickens. „Pencil Code: Block Code for a Text World“. In: *Proceedings of the 14th International Conference on Interaction Design and Children*. Boston, Massachusetts: ACM, 2015, Seiten 445–448. ISBN: 978-1-4503-3590-4.

[12] D. Bau, J. Gray, C. Kelleher, J. Sheldon und F. Turbak. „Learnable Programming: Blocks and Beyond“. In: *Communications of the ACM* 60.6 (2017), Seiten 72–80. DOI: 10.1145/3015455.

[13] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou und M. Denker. *Squeak by Example*. Square Bracket Associate, 2007. ISBN: 978-3-9523341-0-2.

[14] M. Boshernitsan und M. S. Downes. *Visual Programming Languages: a Survey*. Technischer Bericht UCB/CSD-04-1368. EECS Department, University of California, Berkeley, 2004.

[15] N. Bouraqadi und S. Stinckwich. „Bridging the Gap Getween Morphic Visual Programming and Smalltalk Code“. In: *ICDL*. 2007, Seiten 101–120. DOI: 10.1145/1352678.1352685.

[16] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Technischer Bericht. 2017, Seiten 1–16. DOI: 10.17487/RFC8259.

[17] T. Brown und T. Kimura. „Completeness of a Visual Computation Model“. In: *Software Concepts and Tools* 15 (1994), Seiten 34–48.

[18] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore und P. Newton. „Visual Programming and Debugging for Parallel Computing“. In: *IEEE Parallel & Distributed Technology: Systems & Applications* 3.1 (1995), Seiten 75–83. DOI: 10.1109/88.384586.

[19] M. M. Burnett, J. W. Atwood Jr und Z. T. Welch. „Implementing Level 4 Liveness in Declarative Visual Programming Languages“. In: *Proceedings of the IEEE Symposium on Visual Languages*. VL '98. Washington, DC, USA: IEEE Computer Society, 1998, Seiten 126–133. ISBN: 0-8186-8712-6.

[20] M. M. Burnett. „Visual Programming“. In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. Herausgegeben von J. G. Webster. New York: John Wiley & Sons Inc., 1999. ISBN: 978-0-471-13951-5.

[21] M. M. Burnett und A. L. Ambler. „Influence of Visual Technology on the Evolution of Language Environments“. In: *IEEE Computer* 22 (1989), Seiten 9–22. DOI: 10.1109/2.42011.

[22] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang und P. Zee. „Scaling Up Visual Programming Languages“. In: *IEEE Computer* 28 (1995), Seiten 45–54. DOI: 10.1109/2.366157.

[23] M. M. Burnett und D. W. McIntyre. „Visual Programming“. In: *IEEE Computer* 28.3 (1995), Seiten 14–16. DOI: 10.1109/MC.1995.10027.

[24] J. M. Carroll, J. C. Thomas und A. Malhotra. „Presentation and Representation in Design Problem-Solving“. In: *British Journal of Psychology* 71.1 (1980), Seiten 143–153.

[25] V. G. Cerf. „ASCII Format for Network Interchange“. In: *RFC* 20 (1969), Seiten 1–9. DOI: 10.17487/RFC0020.

[26] S.-K. K. Chang. „Ten Years of Visual Languages Research“. In: *Proceedings of 1994 IEEE Symposium on Visual Languages*. St. Louis, MO, USA, 1994, Seiten 196–205. DOI: 10.1109/VL.1994.363617.

[27] S.-K. K. Chang, E. Glinert, J. G. Bonar, M. Graf und A. T. Berztiss. *Principles of Visual Programming Systems*. Prentice Hall Professional Technical Reference, 1990. ISBN: 0-13-710765-X.

[28] J. Cheung, G. Ngai, S. Chan und W. Lau. „Filling the Gap in Programming Instruction: A Text-enhanced Graphical Programming Environment for Junior High Students“. In: *ACM SIGCSE Bulletin* 41 (2009), Seiten 276–280. DOI: 10.1145/1508865.1508968.

[29] C. Cilliers, A. Calitz und J. Greyling. „The Effect of Integrating an Iconic Programming Notation into CS1“. In: *SIGCSE Bulletin* 37.3 (Juni 2005), Seiten 108–112. ISSN: 0097-8418. DOI: 10.1145/1151954.1067478.

[30] P. Costanza. „Dynamically Scoped Functions as the Essence of AOP“. In: *SIGPLAN Notices* 38.8 (2003), Seiten 29–36. DOI: 10.1145/944579.944587.

[31] R. S. Day. „Alternative Representations“. In: Herausgegeben von R. S. Day. Band 22. Psychology of Learning and Motivation. Academic Press, 1988, Seiten 261–305. DOI: 10.1016/S0079-7421(08)60043-2.

[32] E. W. Dijkstra. „Letters to the Editor: Go To Statement Considered Harmful“. In: *Communications of the ACM* 11.3 (1968), Seiten 147–148. DOI: 10.1145/362929.362947.

[33] M. Dorling und D. White. „Scratch: A Way to Logo and Python“. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, Seiten 191–196. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677256.

[34] G. Dózsa. „Parallel Program Development for Cluster Computing“. In: Nova Science Publishers, 2001. Kapitel Visual Programming to Support Parallel Design, Seiten 17–44. ISBN: 978-1-56072-865-8.

[35] P. Dubroy. *Moonchild (2014)*. https://github.com/harc/moonchild, letzter Zugriff 16. Dezember 2019.

[36] M. Erwig und B. Meyer. „Heterogeneous Visual Languages – Integrating Visual and Textual Programming“. In: *Proceedings 11th International IEEE Symposium on Visual Languages, Darmstadt, Germany, September 5-9*. IEEE Computer Society, 1995, Seiten 318–325. ISBN: 0-8186-7045-2. DOI: 10.1109/VL.1995.520825.

[37] O. Etzion, P. Niblett und D. C. Luckham. *Event Processing in Action*. Manning Greenwich, 2011. ISBN: 1-935182-21-8.

[38] S. Federici. „A Minimal, Extensible, Drag-and-Drop Implementation of the C Programming Language“. In: *Proceedings of the 2011 Conference on Information Technology Education*. SIGITE '11. West Point, New York, USA: ACM, 2011, Seiten 191–196. ISBN: 978-1-4503-1017-8. DOI: 10.1145/2047594.2047646.

[39] D. Flanagan und Y. Matsumoto. *The Ruby Programming Language – Everything You Need to Know: Covers Ruby 1.8 and 1.9*. O'Reilly, 2008. ISBN: 978-0-596-51617-8.

[40] B. Foote und R. E. Johnson. „Reflective Facilities in Smalltalk-80“. In: *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), New Orleans, Louisiana, USA, October 1-6, Proceedings*. Herausgegeben von G. Bosworth. ACM, 1989, Seiten 327–335. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74911.

[41] M. Fowler. *Refactoring – Improving the Design of Existing Code*. 1. Auflage. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7.

[42] R. Freudenberg, A. Hyde und A. Gentle. *Squeak/Etoys Reference Manual (2017)*. http://www.gosargon.com/EtoysReferenceManualV0.8.pdf, letzter Zugriff am 16. Dezember 2019.

[43] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[44] D. Garlan, R. Monroe und D. Wile. „Acme: An Architecture Description Interchange Language“. In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. IBM press. 1997, Seite 7.

[45] D. J. Gilmore und T. R. G. Green. „Comprehension and Recall of Miniature Programs“. In: *International Journal of Man-Machine Studies* 21.1 (1984), Seiten 31–48. DOI: 10.1016/S0020-7373(84)80037-1.

[46] D. Giordano und F. Maiorana. „Use of Cutting Edge Educational Tools for an Initial Programming Course“. In: *2014 IEEE Global Engineering Education Conference (EDUCON)* (2014), Seiten 556–563. DOI: 10.1109/EDUCON.2014.6826147.

[47] A. Goldberg und D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989. ISBN: 0-201-13688-0.

[48] A. Goldberg und D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983. ISBN: 0-201-11371-6.

[49] D. Grady. *Stuart Smith's Adventure Construction Set - The Manual*. Electronic Arts, 1984.

[50] R. B. Grafton und T. Ichikawa. „Visual Programming“. In: *IEEE Computer* 18.8 (Aug. 1985), Seiten 6–9. DOI: 10.1109/MC.1985.1662970.

[51] P. Graham. *On LISP. Advanced Techniques for Common LISP*. Prentice-Hall, 1993. ISBN: 0-130-30552-9.

[52] T. R. G. Green und M. Petre. „When Visual Programs are Harder to Read than Textual Programs“. In: *Human-Computer Interaction: Tasks and Organisation*. Herausgegeben von G. C. van der Veer, M. J. Tauber, S. Bagnarola und M. Antavolits. ECCE-6 (6th European Conference on Cognitive Ergonomics). Rom, 1992, Seiten 167–180.

[53] T. Green und M. Petre. „Usability Analysis of Visual Programming Environments: A “Cognitive Dimensions' Framework“. In: *Journal of Visual Languages & Computing* 7 (Juni 1996), Seiten 131–174. DOI: 10.1006/jvlc.1996.0009.

[54] B. Harvey und J. Mönig. *Snap! Reference Manual (2017)*. http://snap.berkeley.edu/SnapManual.pdf, letzter Zugriff am 16. Dezember 2019.

[55] D. Hendry und T. Green. „Creating, Comprehending and Explaining Spreadsheets: A Cognitive Interpretation of What Discretionary Users Think of the Spreadsheet Model“. In: *International Journal of Human-Computer Studies* 40.6 (Juni 1994), Seiten 1033–1065.

[56] G. Hohpe und B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 8. Auflage. Addison-Wesley, 2004. ISBN: 0-321-20068-3.

[57] M. Homer und J. Noble. „Combining Tiled and Textual Views of Code“. In: *2014 Second IEEE Working Conference on Software Visualization*. VISSOFT '14. Victoria, BC, Kanada, Sep. 2014, Seiten 1–10. DOI: 10.1109/VISSOFT.2014.11.

[58] C.-L. Ignat und M. C. Norrie. „Operation-based Versus State-based Merging in Asynchronous Graphical Collaborative Editing“. In: *Sixth International Workshop on Collaborative Editing Systems, CSCW'04, IEEE Distributed Systems online* (2004). ISSN: 1541-4922.

[59] Y. Inayama und H. Hosobe. „Toward an Efficient User Interface for Block-Based Visual Programming“. In: 2018, Seiten 293–294. DOI: 10.1109/VLHCC.2018.8506530.

[60] D. Ingalls, T. Kaehler, J. H. Maloney, S. Wallace und A. C. Kay. „Back to the Future: The Story of Squeak – A Usable Smalltalk Written in Itself“. In: *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'97), Atlanta, Georgia, October 5–9*. Herausgegeben von M. E. S. Loomis, T. Bloom und A. M. Berman. ACM, 1997, Seiten 318–326. ISBN: 0-89791-908-4. DOI: 10.1145/263698.263754.

[61] C. Johnson und A. Abundez-Arce. „Toward Blocks-Text Parity“. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Band 1. Juli 2017, Seiten 413–419. DOI: 10.1109/COMPSAC.2017.244.

[62] R. E. Johnson, J. O. Graver und L. W. Zurawski. „TS: An Optimizing Compiler for Smalltalk“. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), San Diego, California, USA, September 25-30, Proceedings*. Herausgegeben von N. K. Meyrowitz. ACM, 1988, Seiten 18–26. ISBN: 0-89791-284-5. DOI: 10.1145/62083.62086.

[63] A. C. Kay. „The Early History of Smalltalk“. In: *ACMSP* 28.2 (1993), Seiten 69–95. DOI: 10.1145/154766.155364.

[64] G. Klaus, Herausgeber. *Wörterbuch der Kybernetik*. 2. Auflage. Berlin: Dietz Verlag, 1968.

[65] B. Klöppel. *Compilerbau. Am Beispiel der Programmiersprache SIMPL*. 1. Auflage. Vogel, 1991. ISBN: 3-8023-0363-6.

[66] R. Koitz und W. Slany. „Empirical Comparison of Visual to Hybrid Formula Manipulation in Educational Programming Languages for Teenagers“. In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '14. Portland, Oregon, USA: ACM, 2014, Seiten 21–30. ISBN: 978-1-4503-2277-5. DOI: 10.1145/2688204.2688209.

[67] C. W. Krueger. „Software Reuse“. In: *ACM Computing Surveys (CSUR)* 24.2 (1992), Seiten 131–183. DOI: 10.1145/130844.130856.

[68] C. Kyfonidis, N. Moumoutzis und S. Christodoulakis. „Block-C: A Block-based Visual Environment for Supporting the Teaching of C Programming Language to Novices“. In: *9th International Conference "New Horizons in Industry, Business and Education" (NHIBE 2015)*. 2015, Seiten 160–166. ISBN: 978-960-99889-9-5.

[69] T. C. Lethbridge, J. Singer und A. Forward. „How Software Engineers Use Documentation: The State of the Practice“. In: *IEEE software* 20.6 (2003), Seiten 35–39. DOI: 10.1109/MS.2003.1241364.

[70] C. M. Lewis. „How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch“. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 2010, Seiten 346–350.

[71] D. Lloyd. „Pinball Construction Set“. In: *Ahoy!* (1984), Seite 49. ISSN: 8750-4383.

[72] A. MacDonald. „Visual Programming“. In: *Datamation* 28.11 (Okt. 1982), Seiten 132–140.

[73] J. H. Maloney. „An Introduction to Morphic: The Squeak User Interface Framework“. In: *Squeak: Open Personal Computing and Multimedia*. Herausgegeben von M. Guzdial und K. Rose. Prentice Hall, 2002. Kapitel 2, Seiten 39–67. ISBN: 978-0-13-028091-6.

[74] J. H. Maloney, M. Resnick, N. Rusk, B. Silverman und E. Eastmond. „The Scratch Programming Language and Environment“. In: *TOCE* 10.4 (2010), 16:1–16:15. DOI: 10.1145/1868358.1868363.

[75] J. H. Maloney und R. B. Smith. „Directness and Liveness in the Morphic User Interface Construction Environment“. In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, Seiten 21–28. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215636.

[76] Y. Matsuzawa, T. Ohata, M. Sugiura und S. Sakai. „Language Migration in non-CS Introductory Programming Through Mutual Language Translation Environment“. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, Seiten 185–190. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677230.

[77] M. McCool, J. Reinders und A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012. ISBN: 0-12-415993-1.

[78] D. W. McIntyre. *Comp.Lang.Visual - Frequently-Asked Questions List (1998)*. https://web.archive.org/save/http://www.faqs.org/faqs/visual-lang/faq/, letzter Zugriff 16. Dezember 2019.

[79] T. G. Moher, D. C. Mak, B. Blumenthal und L. M. Leventhal. „Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets“. In: *Empirical Studies of Programmers – Fifth Workshop*. Herausgegeben von C. Cook, J. Scholtz und J. C. Spohrer. ESP - Empirical Studies of Programmers. Ablex Publishing, Dez. 1993, Seiten 137–161. ISBN: 1-56750-088-9.

[80] B. A. Myers. „Program Visualization“. In: *Encyclopedia of Software Engineering*. 1994, Seiten 877–892. ISBN: 0-471-54004-8.

[81] J. V. Nickerson. „Visual Programming“. Dissertation. New York University, 1994.

[82] M. Noone und A. Mooney. „Visual and Textual Programming Languages: A Systematic Review of the Literature“. In: *Journal of Computers in Education* (Okt. 2017). DOI: 10.1007/s40692-018-0101-5.

[83] M. Petre. „Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming“. In: *Communications of the ACM* 38.6 (Juni 1995), Seiten 33–44. ISSN: 0001-0782. DOI: 10.1145/203241.203251.

[84] K. Pingali. *Parallel Programming in the Age of Ubiquitous Parallelism (2014)*. http://web.archive.org/web/20190904131643/https://www.sics.se/sites/default/files/pub/keshavpingalisweden.pdf, letzter Zugriff am 4. September 2019.

[85] J. Poswig. „Visuelle Programmiersprachen – Die Realisierung und konzeptionelle Weiterentwicklung eines Prototypen“. Dissertation. Universität Dortmund, 1994.

[86] B. A. Price, R. M. Backer und I. S. Small. „A Principled Taxonomy of Software Visualization“. In: *Journal of Visual Languages & Computing* 4.3 (Sep. 1993), Seiten 211–266. DOI: 10.1006/jvlc.1993.1015.

[87] H. R. Ramsey, M. E. Atwood und J. R. V. Doren. „Flowcharts Versus Program Design Languages: An Experimental Comparison“. In: *Communincations of the ACM* 26.6 (Juni 1983), Seiten 445–449. ISSN: 0001-0782. DOI: 10.1145/358141.358149.

[88] D. Rauch, P. Rein, S. Ramson, J. Lincke und R. Hirschfeld. „Babylonian-style Programming – Design and Implementation of an Integration of Live Examples Into General-purpose Source Code“. In: *Programming Journal* 3.3 (2019), Seite 9. DOI: 10.22152/programming-journal.org/2019/3/9.

[89] E. S. Raymond. *The Art of UNIX Programming*. 1. Auflage. Addison-Wesley, 2003. ISBN: 0-131-42901-9.

[90] P. Rein, R. Hirschfeld und M. Taeumel. „Gramada: Immediacy in Programming Language Development“. In: *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Herausgegeben von E. Visser, E. R. Murphy-Hill und C. Lopes. Amsterdam, The Netherlands: ACM, Nov. 2016, Seiten 165–179. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986022.

[91] J. Reschke, M. Taeumel, T. Pape, F. Niephaus und R. Hirschfeld. *Towards Version Control in Object-based Systems*. Band 121. Universitätsverlag Potsdam, 2018. ISBN: 978-3-86956-430-2.

[92] M. Revell. *What Is Visual Programming? (2017)*. http://web.archive.org/save/https://www.outsystems.com/blog/what-is-visual-programming.html, letzter Zugriff am 16. Dezember 2019.

[93] F. Rivard. „Smalltalk: a reflective language“. In: *Reflection'96*. Herausgegeben von G. Kiczales. 1996, Seiten 21–38.

[94] W. Robinson. „From Scratch to Patch: Easing the Blocks-Text Transition“. In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. WiPSCE '16. ACM, 2016, Seiten 96–99. ISBN: 978-1-4503-4223-0. DOI: 10.1145/2978249.2978265.

[95] B. Rumpe. „Executable Modeling with UML. A Vision or a Nightmare?“ In: *CoRR* (2014). arXiv: 1409.6597.

[96] S. Schiffer. *Visuelle Programmierung – Grundlagen und Einsatzmöglichkeiten*. Addison Wesley, 1998. ISBN: 978-3-8273-1271-6.

[97] A. Schipper, H. Fuhrmann und R. v. Hanxleden. „Visual Comparison of Graphical Models“. In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*. 2009, Seiten 335–340. DOI: 10.1109/ICECCS.2009.15.

[98] S. H. Schwartz. „Modes of Representation and Problem Solving: Well Evolved is Half Solved“. In: *Journal of Experimental Psychology* 91.2 (1971), Seiten 347–350. DOI: 10.1037/h0031856.

[99] B. Shneiderman. „Direct Manipulation: A Step Beyond Programming Languages“. In: *IEEE Computer* 16.8 (1983), Seiten 57–69. DOI: 10.1109/MC.1983.1654471.

[100] N. C. Shu. *Visual Programming*. Van Nostrand Reinhold Co., 1988. ISBN: 0-442-28014-9.

[101] N. C. Shu. „Visual Programming Languages: A Perspective and a Dimensional Analysis“. In: *Visual Languages*. Herausgegeben von S.-K. Chang, T. Ichikawa und P. A. Ligomenides. Springer US, 1986, Seiten 11–34. ISBN: 978-1-4613-1805-7. DOI: 10.1007/978-1-4613-1805-7_2.

[102] D. Singmaster. *Chronology of Computing (2000)*. http://www2.fbi.fh-darmstadt.de/~vmi/chronologie/main.htm, letzter Zugriff am 16. Dezember 2019.

[103] D. C. Smith. „Pygmalion: A Creative Programming Environment“. Dissertation. Stanford University, 1975.

[104] N. Smith, C. Sutcliffe und L. Sandvik. „Code Club: Bringing Programming to UK Primary Schools Through Scratch“. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. ACM, 2014, Seiten 517–522. ISBN: 978-1-4503-2605-6. DOI: 10.1145/2538862.2538919.

[105] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 978-3-211-81106-1.

[106] M. Stefik und D. G. Bobrow. „Object-Oriented Programming: Themes and Variations“. In: *AI Magazine* 6.4 (1986), Seiten 40–62. DOI: 10.1609/aimag.v6i4.508.

[107] S. L. Tanimoto. „A Perspective on the Evolution of Live Programming“. In: *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 2013, Seiten 31–34. ISBN: 978-1-4673-6265-8. DOI: 10.1109/LIVE.2013.6617346.

[108] S. L. Tanimoto. „VIVA: A Visual Language for Image Processing“. In: *Journal of Visual Languages & Computing* 1.2 (1990), Seiten 127–139. DOI: 10.1016/S1045-926X(05)80012-6.

[109] Unicode, Inc. *The Unicode Standard, Version 12.1 – Section Specials*. https://unicode.org/charts/PDF/UFFF0.pdf, letzter Zugriff am 16. Dezember 2019.

[110] A. Warth, P. Dubroy und T. Garnock-Jones. „Modular Semantic Actions“. In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1*. Herausgegeben von R. Ierusalimschy. ACM, 2016, Seiten 108–119. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989231.

[111] M. Weiher und S. Ducasse. „Higher Order Messaging“. In: *Proceedings of the 2005 Symposium on Dynamic Languages, DLS 2005, October 18, San Diego, California, USA*. Herausgegeben von R. Wuyts. ACM, 2005, Seiten 23–34. DOI: 10.1145/1146841.1146844.

[112] D. Weintrop und U. Wilensky. „How Block-based, Text-based, and Hybrid Block/Text Modalities Shape Novice Programming Practices“. In: *International Journal of Child-Computer Interaction* 17 (2018), Seiten 83–92. DOI: 10.1016/j.ijcci.2018.04.005.

[113] D. Weintrop und U. Wilensky. „To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming“. In: *Proceedings of the 14th International Conference on Interaction Design and Children*. IDC '15. Boston, Massachusetts: ACM, 2015, Seiten 199–208. ISBN: 978-1-4503-3590-4. DOI: 10.1145/2771839.2771860.

[114] R. Wexelblat. *History of Programming Languages*. 1. Auflage. Academic Press Inc, 1981. ISBN: 978-0-12-745040-7.

[115] K. N. Whitley. „Visual Programming Languages and the Empirical Evidence For and Against“. In: *Journal of Visual Languages & Computing* 8.1 (1997), Seiten 109–142. DOI: 10.1006/jvlc.1996.0030.

[116] P. Wright und F. Reid. „Written Information: Some Alternatives to Prose for Expressing the Outcomes of Complex Contingencies“. In: *Journal of Applied Psychology* 57.2 (1973). Herausgegeben von G. Chen, Seiten 160–166. DOI: 10.1006/jvlc.2000.0198.

[117] K. D. Wüstneck. „Einige Gesetzmäßigkeiten und Kategorien der wissenschaftlichen Modellmethode“. In: *Deutsche Zeitschrift für Philosophie* 14.12 (1966), Seiten 1452–1467. DOI: 10.1524/dzph.1966.14.14.1452.

[118] K. Zuse. „Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen“. Unveröffentliches Manuskript. 1943.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
131	978-3-86956-481-4	**Was macht das Hasso-Plattner-Institut für Digital Engineering zu einer Besonderheit?**	August-Wilhelm Scheer
130	978-3-86956-475-3	**HPI Future SOC Lab : Proceedings 2017**	Christoph Meinel, Andreas Polze, Karsten Beins, Rolf Strotmann, Ulrich Seibold, Kurt Rödszus, Jürgen Müller
129	978-3-86956-465-4	**Technical report : Fall Retreat 2018**	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Erwin Böttinger, Christoph Lippert
128	978-3-86956-464-7	**The font engineering platform : collaborative font creation in a self-supporting programming environment**	Tom Beckmann, Justus Hildebrand, Corinna Jaschek, Eva Krebs, Alexander Löser, Marcel Taeumel, Tobias Pape, Lasse Fister, Robert Hirschfeld
127	978-3-86956-463-0	**Metric temporal graph logic over typed attributed graphs : extended version**	Holger Giese, Maria Maximova, Lucas Sakizloglou, Sven Schneider
126	978-3-86956-462-3	**A logic-based incremental approach to graph repair**	Sven Schneider, Leen Lambers, Fernando Orejas
125	978-3-86956-453-1	**Die HPI Schul-Cloud : Roll-Out einer Cloud-Architektur für Schulen in Deutschland**	Christoph Meinel, Jan Renz, Matthias Luderich, Vivien Malyska, Konstantin Kaiser, Arne Oberländer
124	978-3-86956-441-8	**Blockchain : hype or innovation**	Christoph Meinel, Tatiana Gayvoronskaya, Maxim Schnjakin
123	978-3-86956-433-3	**Metric Temporal Graph Logic over Typed Attributed Graphs**	Holger Giese, Maria Maximova, Lucas Sakizloglou, Sven Schneider
122	978-3-86956-432-6	**Proceedings of the Fifth HPI Cloud Symposium "Operating the Cloud“ 2017**	Estee van der Walt, Isaac Odun-Ayo, Matthias Bastian, Mohamed Esam Eldin Elsaid
121	978-3-86956-430-2	**Towards version control in object-based systems**	Jakob Reschke, Marcel Taeumel, Tobias Pape, Fabio Niephaus, Robert Hirschfeld
120	978-3-86956-422-7	**Squimera : a live, Smalltalk-based IDE for dynamic programming languages**	Fabio Niephaus, Tim Felgentreff, Robert Hirschfeld

ISBN 978-3-86956-482-1
ISSN 1613-5652