Fehler und deren Vermeidung bei der Programmierung und Konfiguration von Automatiken im ATEO-Projekt

Studienarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Andreas Wickert
geboren am:	15. Februar 1980
in:	Berlin
Gutachter:	Professor Dr. K. Bothe
Betreuer:	Dipl. Inf. Nicolas Niestroj
eingereicht am:	20. Dezember 2012

Abstract

In the project ATEO, which is part of the exploration focus no. 8 whithin the graduate lecture prometei, an experimental environment named SAM has been implemented. It is a complex sociotechnical system, which supports the exploration of connections between resources of system-developers and the quality of their automations for such a system.

While planning and implementing these automation-concepts, errors of many different kinds occur. To sensibilize the developers, especially those working on ATEO, for those errors and their origins, I intensively examined the most common error-classes and -causes and present several alternatives to avoid the identified causations.

In addition I explain in detail the several coordinate systems, that one has to deal with using Squeak. Missunderstandings in the development-team-discussions were often the cause of error or at least maloperation.

Due to Squeak's limited performance resources it was necassary to optimize some of the less performant methods. Several experiments have shown, that the modifications are improvements to the system.

Zusammenfassung

Im Projekt ATEO, das im Rahmen des Forschungsschwerpunktes 8 des Graduiertenkollegs prometei durchgeführt wird, wurde die Versuchsumgebung SAM entwickelt. Ein komplexes soziotechnisches System, das bei der Erforschung der Abhängigkeiten zwischen Ressourcen von Entwicklern und der Qualität ihrer entwickelten Automationen für dieses System helfen soll.

Bei der Umsetzung dieser Automatik-Konzepte werden jedoch auch Fehler in vielfältiger Weise produziert. Um Entwickler, die speziell an ATEO beteiligt sind, für diese Fehler und ihre Ursachen zu sensibilisieren, habe ich mich intensiv mit den häufigsten Fehlerklassen und -quellen auseinander gesetzt. Als Ergebnis meiner Überlegungen präsentiere ich in dieser Arbeit Möglichkeiten zur Fehlervermeidung für die identifizierten Fehlerklassen und -quellen.

Zudem stelle ich die verschiedenen Koordinatensysteme in Squeak detailliert vor, da sich immer wieder gezeigt hat, dass diesbezüglich häufig Missverständnisse in der Team-Kommunikation unter den Entwicklern existierten und es dadurch zu Fehlern oder Fehlverhalten kam.

Weiterhin hat sich gezeigt, dass die Leistungsfähigkeit von Squeak stärker beschränkt ist, als zunächst angenommen. Daher wurden einige Maßnahmen durchgeführt, die den Messergebnissen nach Verbesserungen für das System darstellen.

Hinweis: Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für beide Geschlechter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Akronyme

AAF	**A***TEO* **A***utomation* **F***ramework* 3, 15, 22
AMD	**A***TEO* **M***aster* **D***isplay des Operateursarbeitsplatzes* 3
ATEO	**A***rbeits***t***eilung* **E***ntwickler* **O***perateur* 1, 4, 5, 11, 16, 18, 34, 37
BMP	*Windows* **B***it***M***a***p**-*Format* 26
CASE	**C***omputer-***A***ided* **S***oftware* **E***ngineering* 13, 14
CSV	**C***omma-***S***eparated* **V***alues* 33
GUI	*graphische Benutzungsoberfläche beziehungsweise* **G***raphical* **U***ser* **I***nterface* 3, 20
IDE	**I***ntegrated* oder **I***nteractive* **D***evelopment* **E***nvironment* 11, 14, 34
LOC	**L***ines* **O***f* **C***ode* 5
prometei	**Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I***nteraktion* 1
SAM	**S***ocially* **A***ugmented* **M***icroworld* 1–4, 9, 12, 15, 18, 25, 26, 28, 31

1 Einleitung

1.1 ATEO - ein Beitrag zum Graduiertenkolleg prometei

Das interdisziplinäre Graduiertenkolleg **Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I***nteraktion* (prometei) wird vom Zentrum für Mensch-Maschine-Systeme an der Technischen Universität Berlin betreut. Die Verantwortung für Forschungsschwerpunkt 8 mit dem Titel *Funktionsteilung Mensch-Maschine und Arbeitsteilung Entwickler-Operateur: Zwei Perspektiven auf Mensch-Maschine-Systeme* trägt Prof. Dr. Wandke vom *Lehrstuhl für Ingenieurpsychologie und Kognitive Ergonomie an der Humboldt-Universität zu Berlin.* Innerhalb dieses Forschungsschwerpunktes wird das Projekt **A***rbeits***t***eilung* **E***ntwickler* **O***perateur* (ATEO) durchgeführt. Das erklärte Projektziel von ATEO hat sich mit jedem Phasenwechsel etwas verändert. In früheren Phasen des Projektes (ATEO 1.0, 2004 bis 2007 - ATEO 2.0, 2007 bis 2010) wurde untersucht, „welche internen und externen Ressourcen“ für Entwickler und Operateure „ihre jeweilige Leistungsfähigkeit ausmachen“ [prometei (2006–2012)]. Das heißt, es werden nicht die Leistungen von Menschen mit denen von Maschinen verglichen, „sondern die Leistung zweier Personengruppen zu unterschiedlichen Zeiptunkten empirisch verglichen“ [prometei (2006–2012)]. Verglichen werden also die Leistungen der Entwickler, die zu einem früheren Zeitpunkt die Interaktion von Operateuren mit einem automatisierten System antizipieren und planen müssen, mit den Leistungen der Operateure, die zu einem späteren Zeitpunkt mit den von den Entwicklern implementierten automatischen Systemen in Echtzeit interagieren müssen.

Die aktuelle Projektphase ATEO 3.0 verfolgt „Ansätze kooperativer Automation bzw. dynamischer Funktionsallokation“ [prometei (2006–2012)]. Damit ist gemeint, dass sich Operateur und Automation je nach Situation, Funktion und / oder Aufgabe die Verantwortung gegenseitig übertragen können. Das heißt, in einer bestimmten Situation gibt die Automatik beispielsweise Verantwortung an den Operateur zurück oder in einer anderen Situation überträgt der Operateur der Automatik die Verantwortung (oder einen Teil davon). Durch die geänderte Aufgabenstellung ergeben sich für Entwickler und Operateure neue Herausforderungen. Entwickler müssen nun für eine reibungslose Kommunikation und Kooperation zwischen Operateur und Automation sorgen. Operateure hingegen müssen lernen, mit den neuen Möglichkeiten korrekt umzugehen. Dazu ist zusätzliches Wissen darüber notwendig, wie sich die zur Verfügung stehenden Automationen auswirken und in welchen Situationen es sinnvoll ist, sie einzusetzen.

Um die oben genannten Leistungen von Operateuren mit denen von Entkicklern messen und anschließend vergleichen zu können ([Wandke und Nachtwei (2008)], [ATEO (2008)]), wurde die Versuchsumgebung **S***ocially* **A***ugmented* **M***icroworld* (SAM) entwickelt, in der die entwickelten Automationen zum Einsatz kommen. Bei SAM handelt sich um ein - wie Sommerville in [Sommerville (2011)] es nennt - *soziotechnisches System*, das sowohl aus Hard- und Software-Subsystemen besteht, aber auch menschliche Komponenten enthält. Das Prinzip von SAM besteht darin, dass ein kreisrundes Trackingobjekt eine vorgegebene Strecke entlang manövriert werden soll (vergleiche dazu Abbildung 1). Dieser

Prozess wird von Operateuren und/oder Automatiken steuernd beziehungsweise regelnd begleitet.

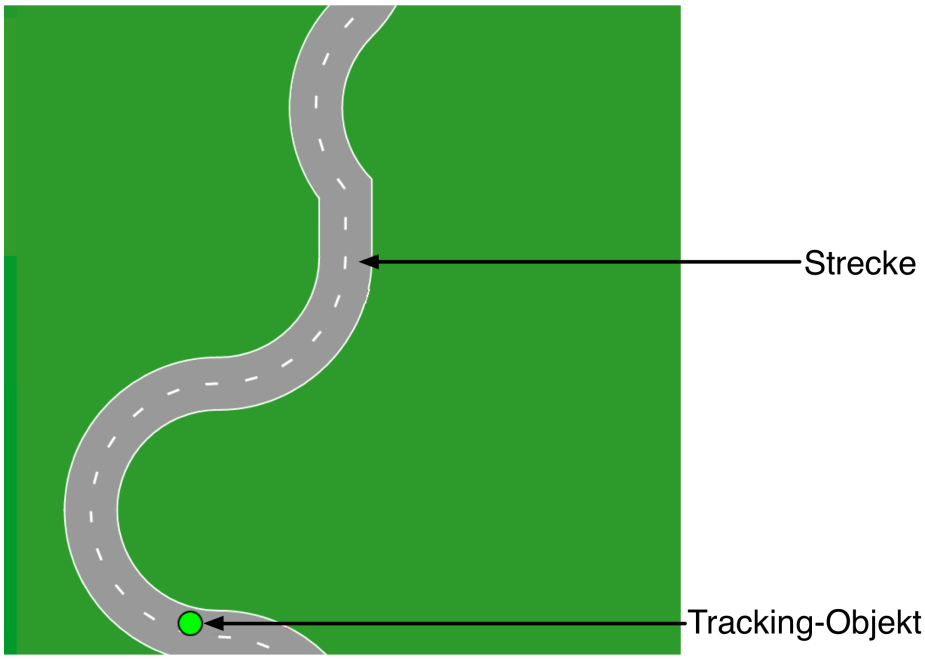

Abbildung 1: Ein Ausschnitt aus dem SAM-Prozess: Zu sehen sind das Trackingobjekt auf einem Streckenabschnitt.

Als Referenz für alle folgenden Versuche wurde ein Single-Tracking durchgeführt, wobei ein (menschlicher) Mikroweltbewohner das Trackingobjekt *ohne* Unterstützung durch den Parcours steuert. Um die Komplexität von SAM zu erhöhen, wurde ein zweiter Mikroweltbewohner dem System hinzugefügt, wodurch es eine nichtdeterministische Note erhielt. Durch den zweiten Mikroweltbewohner wurde die Vorhersagbarkeit des Systems weiter herabgesetzt, da beide Mikroweltbewohner den gleichen Einfluss auf das Trackingobjekt ausüben können, was zum Beispiel bei gegensätzlichem Lenkverhalten dazu führt, dass sich die Eingaben gegenseitig aufheben und sich das Trackingobjekt einfach geradeaus bewegt. Zudem haben beide Mikroweltbewohner unterschiedliche Instruktionen: Der eine hat den Auftrag, zwar schnell, aber dabei besonders genau zu steuern. Der andere hat einen komplementären Auftrag: Er soll genau, aber dabei besonders schnell fahren. Hierdurch wird der Konflikt zwischen den beiden Mikroweltbewohnern noch verstärkt ebenso wie der nichtdeterministische Teil des Systems. Wie die einzelnen Komponenten von SAM miteinander in Beziehung stehen, ist in Abbildung 2 zu erkennen.

1.2 Ziele dieser Arbeit

Das erklärte Ziel dieser Arbeit in erster Linie ist Fehler, die in der Entwicklung von Softwarekomponenten im ATEO Projekt eine Rolle spielen, zu identifizie-

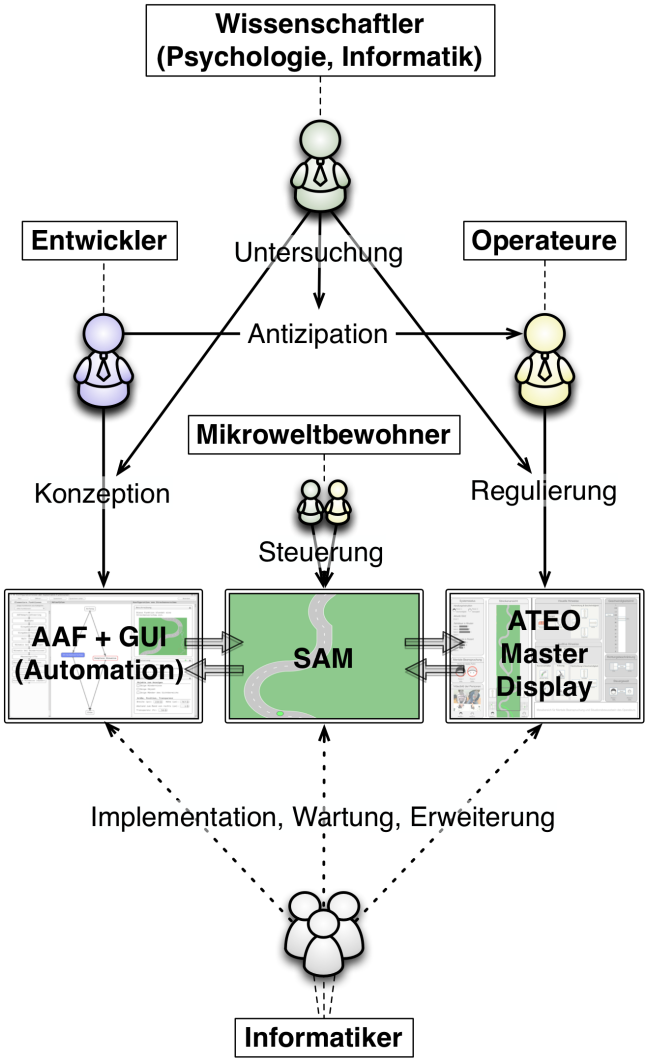

Abbildung 2: Hier ist zu sehen, welche Beziehungen zwischen den einzelnen Rollen untereinander und zu den beteiligten Software-Komponenten von ATEO-SAM (**A***TEO* **A***utomation* **F***ramework* (AAF) / *graphische Benutzungsoberfläche beziehungsweise* **G***raphical* **U***ser* **I***nterface* (GUI), SAM und **A***TEO* **M***aster* **D***isplay des Operateursarbeitsplatzes* (AMD)) existieren.
Entwickler antizipieren Operateur- und Systemverhalten und konzipieren Automatiken zur Unterstützung der Mikroweltbewohner bzw. der Operateure.
Operateure überwachen das System SAM und beeinflussen es über das ATEO Master Display.
Informatiker implementieren, warten, erweitern und verbessern die Softwarekomponenten.
Mikroweltbewohner steuern das Trackingobjekt und erhöhen durch ihr Verhalten die Komplexität von SAM.
Wissenschaftler führen Untersuchungen durch.
(eine weiterentwickelte Grafik aus der Diplomarbeit von [Kosjar (2012)])

Unterstützung	ohne	Operateur	Automatik	Opermatik
SingleTracking	✓ (ATEO 1.0)	—	—	—
MultiTracking	✓ (ATEO 2.0)	✓ (ATEO 2.0)	◯ (ATEO 3.0)	◯ (ATEO 3.0)

Tabelle 1: Diese Tabelle zeigt die Versuchsdurchführungsmatrix. Die mit Kreisen versehenen Versuchskombinationen befinden sich in Planung.

ren, zu klassifizieren und zu analysieren. Dazu gehört auch die Dokumentation der verschiedenen, im Projekt verwendeten, Koordinatensysteme, da diese häufig zu Fehlern oder wenigstens Missverständnissen führten. Außerdem sollen performance-schwache Methoden in SAM erkannt und durch leistungsfähigere Methoden ersetzt werden, die für ihre Ausführung ein geringeres Maß an Zeit benötigen.

1.3 Übersicht über den Aufbau der Arbeit

In Kapitel 2 behandle ich häufige Fehlerklassen und -quellen und stelle Methoden zur Vermeidung beziehungsweise zum Auffinden von Fehlern vor. Anhand einiger besonders interessanter und eindrucksvoller Bugs zeige ich, wie aufwendig es sein kann, Fehler im Nachhinein zu finden und zu beseitigen.

In Kapitel 3 beschreibe ich die verschiedenen Sichtweisen, Bezugs- beziehungsweise Koordinatensysteme, die in Squeak/SAM - speziell für meine anschließende Diplomarbeit [Wickert (2012)] - eine Rolle spielen. Darunter fallen das Standard-Squeak-Koordinatensystem (beschrieben in Abschnitt 3.1), das SAM-Koordinatensystem (beschrieben in Abschnitt 3.2), die Koordinatensysteme für Joystick-Auslenkungen (Abschnitt 3.3) und das Trackingobjekt-Koordinatensystem. Zudem erkläre ich am Beispiel der Hindernisgrafik von SAM, wie die Grafikpositionierung in Squeak/SAM funktioniert.

Im letzten Kapitel 4 dokumentiere ich zunächst ein Werkzeug, mit dem die Ausführungszeit von (Programm-) Blöcken gemessen und je Tick (= eine Iteration von SAMControllerStep processStep) geloggt werden kann sowie zwei Maßnahmen, die ich zur Verbesserung der Leistung von SAM durchgeführt habe.

Zum Abschluss resümiere ich die gesammelten Erfahrungen und diskutiere, inwieweit diese Arbeit einen Mehrwert für zukünftige Entwicklungen in ATEO beziehungsweise SAM hat.

2 Fehler

Moderne Software-Enwicklungsprojekte müssen sich zwangsläufig früher oder später mit Qualitätssicherungsmaßnahmen beschäftigen. Je früher desto kostengünstiger, denn eine späte Fehlerbeseitigung dauert verhältnismäßig länger und hat nicht selten enorme Wartungsarbeiten zur Folge, die die Kosten in die Höhe treiben und etwaige Release-Termine platzen lassen.

Auch das ATEO Projekt hat inzwischen unüberschaubare Ausmaße angenommen: Die geschätzte Anzahl der **L***ines* **O***f* **C***ode* (LOC) liegt etwa um die 100.000 (vergleiche dazu die detaillierte Auflistung in Tabelle 2). Da aber kein festes Personal für die Qualitätssicherung zur Verfügung steht, ist jeder Entwickler innerhalb von ATEO selbst für die Qualität der eigenen Software verantwortlich. Dies führt zwangsläufig zu mehr oder weniger vielen beziehungsweise schwerwiegenden Fehlern. Um deren Zahl aber so gering wie möglich zu halten, ist das schnelle Erlangen von Erfahrungen wichtig. Damit nicht jeder Entwickler, der zum Projekt dazustößt, bei Null beginnen muss, habe ich in diesem Kapitel die häufigsten Fehlerursachen zusammengetragen und biete Möglichkeiten an, wie man sie vermeiden kann.

Kategorie	**Quelldatei**	**LOC**	**Summe**
AAF	AAF	1336	
	AAF-Agents-Concepts-Control	3500	
	AAF-Agents-Concepts-Control-Demo	533	
	AAF-Agents-Concepts-Support	3435	
	AAF-Agents-Concepts-Visual	12931	
	AAF-Agents-Concepts-Visual-Demo	6688	
	AAF-Agents-Deprecated	875	
	AAF-Agents-Dev	6703	
	AAF-Agents-Logs	941	
	AAF-Agents-Logs-Support	33	
	AAF-Events	15715	
	AAF-Graph	1489	
	AAF-GUI	1115	
	AAF-Test	294	
	AAF-Track	4422	
	AAF-TrackData	1366	61376
AAFGT	aafgt.dtd	42	
	AAFGT	1085	
	aafgt.xsd	126	
	AAFGT-Areas	2053	
	AAFGT-Dialog	7300	
	AAFGT-Elements	1481	
	AAFGT-Events	870	
	AAFGT-Events-Configurator	1337	
	AAFGT-GUI	0	
	AAFGT-Test-Cases	205	
	AAFGT-Test-Tools	42	
	AAFGT-VisualHintsDemo	622	
	AAFGT-Widgets	2768	

	AAFGT-Widgets-Agents	2790	
	AAFGT-Widgets-Support	1166	
	AAFGT-XML	381	22268
MW	ATEO-MW-CustomClasses	70	
	ATEO-MW-GUI	282	
	ATEO-MW-Network	67	
	ATEO-MW-Tools	37	
	ATEO-MW-Version	15	471
Network	ATEO-Network	107	
OP	ATEO-OP-CustomClasses	716	
	ATEO-OP-GUI	3008	
	ATEO-OP-Model	335	
	ATEO-OP-ModelData	297	
	ATEO-OP-Network	311	
	ATEO-OP-Tools	35	
	ATEO-OP-Version	15	4824
SAM	ATEO-SAM-Config	249	
	ATEO-SAM-Controller	3175	
	ATEO-SAM-Model	1054	
	ATEO-SAM-Version	15	
	ATEO-SAM-View	1542	6035
Utils	ATEO-Utils	1092	
Tools	ticks for forks (py)	128	
	LFA 1.0	2031	
	LFA 2.0		
	- Checker	230	
	- GUI	1012	
	- LFA Data	412	
	- LFA Item	2603	
	- LFA Item (automatic)	858	
	- output	2246	
	- UnitTests	290	
	- UnitTests (checkertests)	154	11056
Gesamt:			106030

Tabelle 2: Anzahl der Zeilen in den jeweiligen Projektteilen

2.1 Fehlerklassen

Fehler können in bestimmte Klassen eingeordnet werden. In der Bachelorarbeit von Neumann (2008) sind sehr viele dieser Fehlerklassen beschrieben, wovon ein Großteil aber sehr programmiersprachenspezifisch ist. Um nicht den Rahmen dieser Arbeit zu sprengen, beschränke ich mich hier auf eine kleine Auswahl der Klassen, die nicht von der verwendeten Programmiersprache abhängig sind, sondern allgemein Gültigkeit besitzen. Ich lege keinen besonderen Wert auf Vollständigkeit; lediglich die Hauptklassen, mit denen ich mich während der Entwicklungszeit konfrontiert sah, behandle ich in den folgenden Unterabschnitten.

2.1.1 Syntaxfehler

Hierbei handelt es sich um Strukturfehler des Programms. Die *Syntax* (griechisch-lateinische Herkunft) ist die Lehre vom Bau eines Satzes. Als Syntaxfehler bezeichnet man alle Fehler, die einen Verstoß gegen die grammatischen Regeln der verwendeten Programmiersprache darstellen.

Syntaxfehler spielen in der Gegenwart nur noch selten eine Rolle, da sie bereits mit den gängigen Programmierwerkzeugen unmittelbar entdeckt (zum Beispiel durch *Syntaxhighlighting*) und ebenso schnell beseitigt werden können (zum Beispiel durch eine so genannte *Quickfix*-Funktion, die gegebenenfalls eine passende Lösungsmöglichkeit anbietet; vergleiche auch Abbildung 4).

Bei Squeak (Smalltalk) werden Syntaxfehler erst mit dem Abspeichern der aktuellen Änderungen an einer Methode entdeckt. An der fehlerhaften Stelle wird dabei ein Hinweis in den Quelltext eingefügt, der dem Programmierer helfen soll, den Fehler schnell zu beseitigen. Das Beispiel in Abbildung 3 zeigt, was passiert, wenn man ein notwendiges Zeichen im Quelltext vergisst. In Abbildung 4 sieht man, dass Squeak bei falscher Schreibweise einer Variablen gleich Lösungsmöglichkeiten anbietet, den bestehenden Fehler zu beheben.

```
example
    | myValue |

    myValue := 5
    myValue  Nothing more expected ->:= 4
```

Abbildung 3: Syntaxfehler: Der Punkt am Ende des ersten Ausdrucks wurde vergessen.

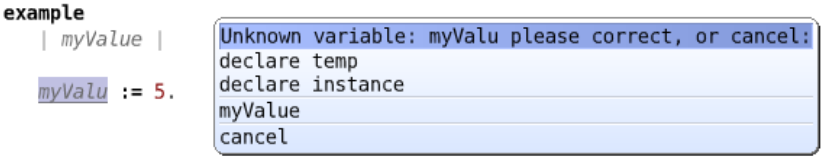

Abbildung 4: Syntaxfehler: Der Variablenname wurde nicht korrekt geschrieben. Erkennbar ist auch ein Beispiel für einen Quick-Fix: Im geöffneten Popup-Fenster ist als vorletzter Eintrag die korrekt geschriebene Variable zu sehen.

2.1.2 Laufzeitfehler

Bei dieser Sorte dreht es sich um Fehler, die während der Ausführung eines Programms auftreten. Vorausgesetzt der Quelltext ist weitestgehend frei von syntaktischen Fehlern, werden Laufzeitfehler dadurch hervorgerufen, indem beispielsweise Feldstrukturen (Arrays) falsch bemessen werden und somit Iterationen, die über diese Felder laufen sollen, zu lange laufen (siehe Abbildung 5).

Eine andere Ausprägung dieses Fehlertyps äußert sich zum Beispiel darin, dass bei typ-unsicheren Sprachen (wie zum Beispiel Smalltalk) der Übergabeparameter einen falschen Typ hat (ein Beispiel ist in Abbildung 6 zu sehen).

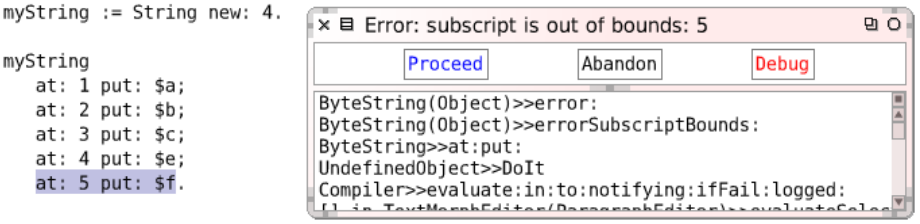

Abbildung 5: Laufzeitfehler: Zugriffsversuch auf eine Feldstruktur an einer nicht definierten Stelle. Der erzeugte String ist nur 4 Felder lang, es wird aber versucht, in das 5. Feld zu schreiben. Entsprechend ist das Verhalten bei Schreibversuchen das gleiche.

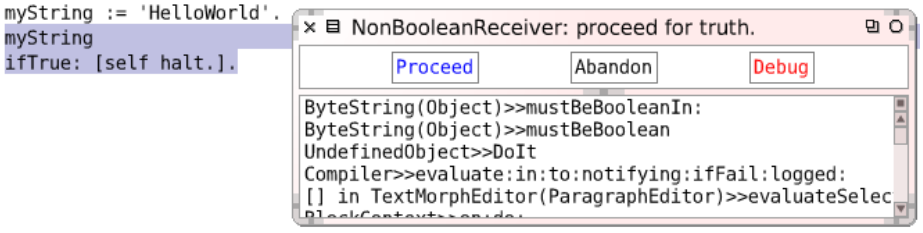

Abbildung 6: Laufzeitfehler: Der erzeugte String wird als Variable des Typs Boolean verwendet. Dies führt zum Fehler 'NonBooleanReceiver: proceed for truth.'.

2.1.3 Semantik- oder Algorithmusfehler

Noch schwerer auffindbar als Laufzeitfehler sind semantische Fehler. Derartige Fehler können nur anhand des Verhaltens eines Programms ausfindig gemacht werden. Der Tester muss Schritt für Schritt den Ablauf des Programms verfolgen und dabei den Programmfluss exakt nachvollziehen. Erst wenn auf diese Art - optimaler Weise - alle möglichen Wege genau überprüft wurden und dabei kein fehlerhaftes Verhalten festgestellt werden konnte, kann man davon ausgehen, dass sich das Programm genau so verhält, wie es konzipiert wurde. Durch solche speziellen und aufwändigen Tests wird bestimmt, ob der zugrundeliegende Algorithmus semantisch fehlerfrei ist. Die größte Herausforderung bei der Suche nach semantischen Fehlern stellt die Auswahl der (geeigneten) Testfälle aus der Menge aller möglichen Testfälle dar. Mindestens aber sollten alle Fälle, die im Einsatzbereich der Software denkbar sind, abgetestet werden, damit wenigstens in allen planmäßigen Situationen ein fehlerfreies Arbeiten des Programms gewährleistet ist. Optimal wäre, wenn alle möglichen Situationen für ein Programm abgedeckt würden. Das ist jedoch oft nicht möglich, da es unter Umständen unendlich viele (oder mehr) Fälle geben kann.

Aufgrund der Vielfalt, in der semantische Fehler auftreten können, gibt es kein Paradebeispiel eines semantischen Fehlers. Jeder semantische Fehler hat seine ganz persönliche Geschichte und ist daher nicht verallgemeinerbar. Hin und wieder gibt es sogar Situationen, in denen das Verhalten eines Programms als fehlerhaft wahrgenommen wird; dieses Verhalten aber tatsächlich so gewollt ist (*„It's not a bug, it's a(n undocumented) feature!“* - Ursprung unbekannt).

Ein semantischer Fehler liegt auch dann vor, wenn sich Softwaremodule nach ihrer Integration gegenseitig „behindern“ beziehungsweise stören oder ganz und gar ihr eigentliches Verhalten ändern, weil sie über ihre Schnittstellen andere Werte erhalten, mit denen sie nicht umgehen können. Siehe dazu auch Unterabschnitt 2.2.4, wo ich am Beispiel unterschiedlicher Agenten verschiedene Fälle erläutere, in denen es zu Problemen oder Fehlverhalten von Automatiken kommt.

2.1.4 Design- oder Konzeptfehler

An dieser Stelle möchte ich auch noch auf Fehler hinweisen, die ihren Ursprung in einem schlechten oder falschen Design bzw. Konzept haben. In einem solchen Fall stimmt die tatsächliche Anforderung an eine Softwarekomponente nicht mit ihrer Funktionsweise überein. Das heißt, das Programm erfüllt nicht die Aufgabe, für die es eigentlich zuständig ist.

In diese Klasse gehören aber auch noch andere Fälle. Ein häufig auftretender „Fehler“ ist *Code-Redundanz*. Dabei werden nur Quelltextstücke kopiert (den Rest des Ursprungscodes benötigt man nicht) und dann an der neuen Stelle unverändert weiterverwendet. Dieses Vorgehen verkürzt zwar die Entwicklungszeit der neuen Softwarekomponente, führt aber auch zu erhöhtem Wartungsaufwand, wenn an genau dem kopierten Quelltext Änderungen durchgeführt werden müssen.

Ein schlechtes Design liegt auch dann vor, wenn zwar die Funktionsweise mit den Anforderungen übereinstimmt, die Art und Weise aber, wie ein Algorithmus zum Ergebnis kommt, nicht optimal ist. Gerade bei Systemen, in denen die Zeit eine Rolle spielt, sind solche Performance- (Leistungs-) Fehler besonders störend. SAM ist ein System, bei dem es zwar einen gewissen zeitlichen Puffer gibt, aber wenn dieser aufgebraucht ist, kommt es zu einer Verzögerung im eigentlichen Prozess, der sehr störend wirkt. Das Problem ist in Unterabschnitt 2.5.5 detailliert beschrieben. Meine Versuche, diesem Problem entgegen zu wirken, dokumentiere ich in Unterabschnitt 4.2.

2.1.5 Systemfehler

Unter einem Systemfehler versteht man jede Form von Fehlern, die von außen auf das Softwaresystem störend einwirken. Dabei spielt es keine Rolle, ob die Fehlerursache direkt (Perepheriegeräte, siehe Unterabschnitt 2.5.1) oder indirekt (Operateur, Mikroweltbewohner) mit dem „Kernsystem“ (bestehend aus Computer-Hardware und -Software) verbunden ist oder sogar noch außerhalb dieses Systems liegt (wie etwa eine ausgefallene Raum-Klimaanlage, die die Leistung des Systems indirekt vermindert). Aber auch interne Fehler, die nicht direkt mit dem Quelltext in Verbindung stehen, werden als Systemfehler bezeichnet (siehe Unterabschnitt 2.5.5).

2.1.6 Fehler durch die Verwendung von Double-Variablen

Die Verwendung von Double-Variablen ist im Allgemeinen sehr verbreitet, führt aber auch häufig zu Problemen. Gerade wenn es darum geht, Double-Werte miteinander zu vergleichen, stoßen viele Interpreter an ihre Grenzen. Besonders bei der Verwendung von Double-Variablen in Squeak sind überaus interessante Phänomene aufgetreten (siehe Unterabschnitt 2.5.6).

2.2 Fehlerquellen

Der Mensch ist nicht unfehlbar. Diese Eigenschaft überträgt sich selbstverständlich auch auf die von Menschenhand gefertigten Produkte. Programme sind davon nicht ausgeschlossen. Aber diese Fehlbarkeit ist nicht die einzige Ursache von Fehlern in Software. Es gibt noch weitere Quellen, welche besondere Aufmerksamkeit verdienen, da sie häufig vermeidbar sind. In diesem Unterabschnitt werden diese verschiedenen vermeidbaren Quellen vorgestellt. Ob und wie sie zu vermeiden sind, wird in Unterabschnitt 2.3 erläutert.

2.2.1 Kopieren, Einfügen, Anpassen

Eine sehr beliebte Art und Weise, um Fehler zu „produzieren“, ist das klassische *copy 'n' paste*. Der Fehlerverursacher beabsichtigt, Teile des Quellcodes wiederzuverwenden, indem er existierenden Code an eine andere Stelle kopiert und für die neuen Zwecke anpasst. Dabei können, je nach Situation, unterschiedliche Fehlerarten auftreten. Je größer der kopierte Codeteil ist, desto größer ist auch die Wahrscheinlichkeit, dass die neue Methode fehlerhaft ist.

In erster Linie treten dabei lexikalische Fehler auf, da Variablen, die im vorherigen Scope (= Umgebungstiefe, Gültigkeitsumgebung einer Methode) gültig waren, im neuen Scope entweder nicht existieren, anders benannt oder sogar schon im aktuellen Kontext in Verwendung sind.

Besonders problematisch kann die Verwendung von Klassen-, Methoden- oder globalen Variablen werden. Enthält der kopierte Codeblock Bezeichner eben solcher Variablen, und beachtet der Programmierer dies nicht, so kommt es mit Sicherheit zu ungewolltem Verhalten. Denn bei diesen Variablenarten ist in der Regel nicht bekannt, welchen Wert sie gerade beinhalten, wenn die neue Methode aufgerufen wird und ob dieser Wert für die neue Methode ebenso sinnvoll ist. Es könnte sein, dass die Variable einen unpassenden Typ hat (siehe nachfolgenden Absatz) oder einen Wert beinhaltet, der nicht im gültigen Wertebereich für die neue Methode liegt. Zudem muss sichergestellt sein, dass Inhalte der Variablen, die innerhalb der neuen Methode geändert wurden, nicht die Korrektheit der Ursprungsmethode beziehungsweise der anderen Methoden, die ebenfalls auf diese Variablen zugreifen, dadurch beeinträchtigt wird. In den meisten Fällen dürfte es spätestens hier zu Problemen kommen. Allerdings sind auch Szenarien denkbar, in denen das beschriebene Verhalten auch beabsichtigt sein könnte.

Sehr häufig kommt es beim copy'n'paste auch zu semantischen Fehlern. Beispielsweise vergessen Entwickler oft die Anpassung der verwendeten Variablentypen. Was für die originale Codepassage korrekt war und fehlerfrei funktio-

niert hat, muss nicht auch für den neuen Einsatzort des kopierten Codeblocks gelten. Speziell der Rückgabetyp und die Parametertypen bedürfen hier besonderer Aufmerksamkeit. Sowohl beim Methodenaufruf selbst, als auch nach der Rückkehr von der neuen Methode zum ursprünglichen Programmfluss müssen die Parametertypen übereinstimmen. Auch hier helfen wieder die Werkzeuge der **I***ntegrated* oder **I***nteractive* **D***evelopment* **E***nvironment* (IDE) und weisen gegebenenfalls auf falsche oder inkompatible Typennutzung hin. Manche Programmiersprachen können sogar selbstständig Casts (= der tatsächliche Typ einer Variablen wird als anderer Variablentyp interpretiert - beispielsweise kann eine Variable vom Typ *Integer* auch als *Float*-Typ betrachtet werden) durchführen. Dies kann unter Umständen ebenso wieder zu einem Problem werden. Denn wenn zu einem späteren Zeitpunkt im Programmablauf der vom Programmierer verwendete Typ explizit verlangt wird, dieser aber gar nicht mehr existiert, weil der Wert der Variablen automatisch uminterpretiert wurde, kommt es zu einem Laufzeitfehler. In diesem Fall ist es besonders anstrengend die fehlerhafte Stelle zu finden, da der gesamte Programmfluss nachvollzogen werden muss. Gerade bei typunsicheren Programmiersprachen sollte hier besondere Sorgfalt gehalten werden, da in der Regel erst zur Laufzeit bekannt ist, welcher Variablentyp zum Zeitpunkt der Ausführung vorliegt.

2.2.2 Method-Stubs - Unfertige Methodenrümpfe

Ansich handelt es sich bei unfertigen Methoden nicht direkt um einen Fehler. Es ist schließlich nicht falsch, wenn eine Methode keine Funktion erfüllt. Es wird erst zu einem Problem, wenn man davon ausgeht, dass diese Methode das liefert, was sie verspricht. Während meiner Zeit im ATEO-Projekt gab es einen Fall, in dem eben diese Annahme getroffen wurde und etwa acht Stunden Fehlersuche investiert wurden, um herauszufinden, an welcher Stelle der beobachtete Effekt seinen Ursprung hatte. Welchen Effekt diese offene Stelle zur Folge hatte und wodurch er verursacht wurde ist ausführlich in den Unterabschnitten 2.5.2 und 2.5.3 dokumentiert.

2.2.3 Quick and Dirty Coding

Wer schnell und hastig programmiert, macht auch irgendwann unweigerlich Fehler. Gerade bei ungeplanten Implementierungen von Funktionen, deren Bedarf erst ad-hoc festgestellt wurde, werden häufig Fehler eingebaut, die bei einer besseren Planung hätten vermieden werden können. Zudem fehlt es derartigen Funktionen oft an Effizenz, da sie hauptsächlich nur einen gewünschten Effekt erzielen sollen. Das heißt, man versucht mit der Implementierung lediglich das verfolgte Ziel zu erreichen, auch wenn der Weg dahin äußerst rechenintensiv ist oder die Ausführungszeit besonders lange dauert. Wer schnell und dreckig programmiert, lässt häufig auch die resultierenden Konsequenzen außer acht und ist sich oft nicht darüber im Klaren, wie weitreichend die Folgen sein können, wenn beispielsweise Instanz- oder Klassenvariablen verändert werden.

2.2.4 Modulintegration

Eine weitere Fehlerquelle kann bei der Integration verschiedener Module zu einem System entstehen. Häufige Ursache von Fehlern während dieses Prozesses ist eine Fehlerhafte Schnittstelle. Das heißt, *Modul A* liefert nicht das, was *Modul B* (in diesem Fall der Nachfolger von *Modul A*) erwartet. Dabei mögen beide Module für sich genommen fehlerfrei arbeiten. Im Fall von SAM könnte es sich beispielsweise um ein Problem mit dem übergebenen Datenobjekt SAMState handeln, das sowohl als Eingabe- als auch als Ausgabeobjekt für alle Automatik-Agenten dient. Denkbar wäre, dass entweder *Modul A* im Objekt einen Wert falschen Typs gesetzt hat und *Modul B* nun nicht mit diesem Eintrag umgehen kann. Oder anders herum: *Modul A* setzt alle Werte korrekt, aber *Modul B* erwartet einen anderen aber falschen Typ.

Während die zuvor genannten Situationen Fehler technischer Natur beschreiben, kann es bei der seriellen Integration von Modulen (die zu integrierenden Module werden nacheinander ausgeführt) auch zu semantischen Problemen kommen. Wie im letzten Abschnitt bereits erwähnt, wird bei SAM-Automatiken der SAMState von Modul zu Modul gereicht. Jedes Modul kann und darf Änderungen an diesem Datenobjekt vornehmen. Daher ist es denkbar, dass zwei Module lesend und schreibend auf die selben Stellen im SAMState zugreifen. Dies kann unter besonders ungünstigen Umständen dazu führen, dass weder der eine Agent noch der andere eine tatsächliche Wirkung im Prozess erzielen, da sich die individuellen Wirkungsweisen gegenseitig aufheben. Weiterhin äußerst problematisch ist eine Situation, in der ein Agent mit bereits veränderten Daten als Voraussetzung arbeiten soll. Eventuell werden dabei für die aktuell gültige Situation von SAM völlig unsinnige Ergebnisse erzielt. Beispiele zu diesem Thema sind in Unterabschnitt 2.5.4 aufgeführt.

2.2.5 Fehler durch Unklarheiten

Oft führen Unklarheiten in Anforderungs-Formulierungen, falsche Informationen über das zugrunde liegende System oder auch nur Missverständnisse in der Kommunikation von Teammitgliedern dazu, dass Fehler auftreten. Dies ist auch während und vor meiner Zeit im ATEO-Projekt der Fall gewesen, weshalb ich mich eingehend mit dem Gegenstand der Missverständnisse und Irrtümer außerordentlich genau auseinander gesetzt habe. Es geht dabei um das Thema der *Koordinatensysteme in Squeak und SAM*. Daher widme ich diesem Bereich ein eigenes Kapitel und verweise auf den Abschnitt 3.

2.3 Fehlervermeidung

2.3.1 Berücksichtigung von Software-Engineering-Richtlinien

Ganz allgemein ist es ratsam sich zumindest ansatzweise an gängige Prinzipien für das Software-Engineering zu halten. Ich werde hier nicht näher auf die einzelnen Themen eingehen, um den Rahmen dieser Arbeit nicht zu sprengen, denn mit Ausarbeitungen über Vor- und Nachteile der diversen Prinzipien kann man ganze Bücher füllen. Nach [Sommerville (2011)] sind besonders die folgenden

Themen zur Fehlervermeidung von besonderer Bedeutung:

- Entwurf und Implementierung:
  - Objektorientierter Entwurf mit UML (Kapitel 7.1)
  - Entwurfsmuster (Kapitel 7.2)
  - Implementierungsaspekte: Wiederverwendung (Kapitel 7.3.1)
- Testen von Software: Entwicklertests (Kapitel 8.1)
- Softwareevolution: Softwarewartung: Präventive Wartung durch Refactoring (Kapitel 9.3.1)
- Qualitätsmanagement:
  - Softwarequalität (Kapitel 24.1)
  - Softwarestandards (Kapitel 24.2)
  - und Reviews und Inspektionen (Kapitel 24.3)

2.3.2 Verwenden von CASE Werkzeugen

C*omputer-***A***ided* **S***oftware* **E***ngineering* (CASE) Werkzeuge unterstützen den Software-Entwicklungsprozess auf vielfältige Weise. Viele dieser Werkzeuge werden schon bei der Eingabe des Programmcodes aktiv, indem sie dabei helfen, die Syntax der Programmiersprache einzuhalten. In erster Linie spielen hierbei *Syntaxhighlighting* und *Codecompletion* die größten Rollen.

Durch Syntaxhighlighting wird dem Programmierer in visueller Form gezeigt, ob er beim Programmieren die Syntax einhält. Macht er einen (Syntax-) Fehler, so wird ihm dies beispielsweise durch ein rotes Symbol angezeigt. Bei Squeak wird der Text an einer fehlerhaften Stelle rot dargestellt und beim Versuch, den Fehlerhaften Quelltext zu speichern, werden dem Fehler entsprechende Beseitigungsvorschläge oder wenigstens der Grund des Fehlers textuell eingeblendet (siehe Abbildungen 3 und 4).

Codecompletion leistet zwar unterschiedliche Hilfen, aber immer nach dem gleichen Prinzip. Wird sie (meist per Tastenkombination) aktiviert, liefert sie dem Programmierer alle ihm an der aktuellen Stelle zur Verfügung stehenden Optionen, wie der Programmcode fortgeführt werden kann. Beginnt der Programmierer den Namen einer Variablen zu schreiben und aktiviert er währenddessen die Codecompletion dann werden ihm alle Variablen aufgelistet, die mit der bereits eingegebenen Zeichenfolge beginnen. Auch Klassennamen und Methodenaufrufe (inklusive der zugehörigen Parameter) lassen sich auf diese Weise vervollständigen. Durch die Nutzung der Codecompletion wird nicht nur die Einhaltung der korrekten Variablen-Schreibweise gewährleistet, sondern auch die Eingabedauer verringert. Dabei entsteht in kürzerer Zeit qualitativ hochwertigere Software. Allerdings setzt dies voraus, dass unter den von der Codecompletion angebotenen Optionen die richtige ausgewählt wird.

Ein weniger häufig verwendetes Hilfsmittel ist das Template. Es handelt sich dabei um ein Stück Code, dass immer wieder verwendet wird. Hilfreich ist seine

Verwendung besonders dann, wenn sich redundanter Code nicht vermeiden lässt. Wenn beispielsweise relativ häufig ein Objekt über diverse Methoden instanziiert werden soll, kann ein Template hierbei helfen. Man muss dann nur noch die Parameter in den Methoden richtig setzen. Beispielsweise kommen Templates bei LaTeX zum Einsatz. Auch sie können mit Codecompletion eingefügt werden. Ich nutze für die Erstellung dieser Arbeit die IDE *Eclipse* mit der Erweiterung *texlipse*. Gebe ich in einem TeX-Editorfenster die Zeichen *fig* ein und aktiviere die Codecompletion so habe ich als Option auch das Template für eine *figure*. Wähle ich diese Option aus, wird der entsprechende Code generiert (siehe Listing 1).

```
%\usepackage{graphics} is needed for \includegraphics
\begin{figure}[htp]
\begin{center}
  \includegraphics[width=figureWidth]{filename}
  \caption[labelInTOC]{figureCaption}
  \label{figureLabel}
\end{center}
\end{figure}
```

Listing 1: LaTeX-Template für eine Figure

Desweiteren unterstützen CASE Werkzeuge auch bei der Testfallerstellung, -generierung, -überprüfung/-durchführung und -organisation. Sie bieten Übersichten, ob Testziele erreicht wurden, welche Softwarekomponenten dabei verwendet wurden, wieviel Prozent des Quelltextes dabei durchlaufen wurde, usw. Das Prinzip des modellbasiertes Testens hilft sowohl dabei den Überblick über das Zusammenspiel der Softwarekomponenten nicht zu verlieren (durch die Darstellung des Programmablaufs als Modell), als auch das automatisieren der Testfallsuche. Mit den Werkzeugen *Conformiq Designer* und *Conformiq Modeler* [Conformiq (2012a)] lassen sich sehr leicht Modelle erzeugen und automatisch (Abdeckungs-) Testfälle generieren, mit denen alle erreichbaren Kanten durchlaufen werden. Der *Conformiq Designer* bietet außerdem die Möglichkeit, zusätzliche Testziele zu definieren, die dann mit ebenfalls automatisch erzeugten Testfällen erfüllt (oder nicht erfüllt) werden. Eine komplette Übersicht über alle Funktionen des Designers erhält man auf der Webseite [Conformiq (2012b)].

2.3.3 Vermeiden von Fehlerquellen

Grundsätzlich ist es ratsam, die in Unterabschnitt 2.2 beschriebenen Fehlerquellen zu vermeiden. Wenn man aber nicht umhinkommt, und dennoch die genannten riskanten Praktiken nutzt, sollte man dabei ganz besondere Sorfalt aufwenden. Ich möchte mit den folgenden Absätzen Tipps geben und sensibel dafür machen, wie diese Sorgfalt zu erreichen ist.

Um Fehler durch *Copy'n'Paste* zu vermeiden, sollten nicht zu große Mengen von Quelltext mit einem Mal kopiert werden. Beim Kopieren darf man nicht den Überblick verlieren, um nicht die genannten Fehler zu provozieren. Das Beispiel in Unterabschnitt 2.5.1 zeigt, was passieren kann, wenn die Nachbearbeitung von kopiertem Quelltext nicht sorgsam genug vollzogen wird.

Oft ist es sinnvoll, bereits frühzeitig leere Methoden (method stubs) für Funktionen anzulegen, deren Implementierung erst später erfolgen soll. Allerdings muss jedem Nutzer dieser Methode auf irgendeine Art mitgeteilt werden, dass die Funktion der Methode noch nicht vollständig oder noch nicht ausgereift ist. Im Optimalfall wird eine Exception (Ausnahmebehandlung) geworfen. Auch wenn eine Methode ihre Bestimmung nur grob oder notdürftig erfüllt, sollte der Nutzer darauf in irgendeiner geeigneten Weise hingewiesen werden. Zwei Fälle, wo dies nicht berücksichtigt worden ist, beschreibe ich in den Unterabschnitten 2.5.2 und 2.5.3.

Selbstverständlich ist es notwendig, jede definierte Schnittstelle stets korrekt zu erfüllen. Es muss immer gewährleistet sein, dass alle Übergabeparameter vom richtigen Typ sind. Ebenso müssen Objekte, die übergeben werden (wie zum Beispiel der SAMState), verifiziert werden. Diese Aufgabe erledigen Modultests, indem sie überprüfen, ob alle Einträge vom richtigen Typ sind und außerdem zulässige Werte besitzen. Dazu ist es sinnvoll, mehrere gültige Instanzen des Übergabeobjektes per Hand zu erzeugen und das Ergebnis des Moduls immer mit diesen Instanzen zu vergleichen. Weicht ein Wert oder Typ von den Vorgaben ab, muss man sich auf die Fehlersuche (siehe Unterabschnitt 2.4) begeben.

Statt 'schnell und dreckig' zu programmieren, lohnt es sich immer ein wenig mehr Zeit für eine vernünftige Umsetzung einer Aufgabe oder eines Problems zu investieren. Selbst wenn es vorher nicht immer absehbar ist, so macht es sich häufig bezahlt, den etwas komplizierteren, dafür aber effizienteren Algorithmus zu implementieren. Zudem habe ich die Erfahrung gemacht, dass es sinnvoll ist, einen großen Algorithmus in kleinere Module zu gliedern, um eine mögliche Wiederverwendung von Teilen des Algorithmus zu erleichtern. Ein Refactoring (Überarbeitung) zu einem späteren Zeitpunkt kostet immer mehr Zeit, als wenn man gleich von Beginn an darauf achtet. Zudem reduziert man durch Modularisierung die Redundanz von Quelltexten. Wenn wirklich etwas wiederverwendet werden kann, so sind die notwendigen Vorkehrungen bereits getroffen. Dazu gehört auch, dass man Methoden und Bezeichner mit kurzen und prägnanten Namen versieht. Zudem hat sich das Kommentieren des Quelltexts immer wieder bezahlt gemacht und eine enorme Zeitersparnis eingebracht, wenn es nötig wurde, 'alten' Code nach langer Zeit wieder genau verstehen zu müssen, um beispielsweise nach Optimierungsmöglichkeiten zu suchen (siehe Abschnitt 4).

Beim erstellen von Automatiken müsste optimaler Weise jeder Agent, der dem Graphen in der ATEO-GUI hinzugefügt wird, dem Architekten der Automatik mitteilen, welche Werte im SAMState benutzt beziehungsweise verändert werden. Damit soll sichergestellt werden, dass die Realisierung der Automatik derart von Statten geht, sodass sich die eingefügten Agenten nicht gegenseitig stören oder bestimmte Effekte, die über die Agenten erzielt werden sollen, nicht eintreten können. Bislang ist eine solche Änderungsinformation nicht im AAF vorgesehen.

2.3.4 Double Variablen

Die Verwendung von Double Variablen ist gängige Praxis in SAM. Dabei ist es jedoch notwendig, dass man gewisse Eigenheiten von Doubles in Squeak kennt und von vornherein vermeidet, in Situationen zu geraten, in denen diese Eigen-

heiten greifen. Welche Schwierigkeiten auftreten können und wie diese umgangen werden können, zeigen die Beispiele in Unterabschnitt 2.5.6.

2.4 Fehlersuche

In diesem Unterabschnitt stelle ich die von mir am häufigsten benutzten Methoden zur Fehlersuche und -beseitigung vor. Viele dieser Methoden dienen dabei nicht nur dazu, Fehler zu finden, sondern sind auch dafür geeignet, die Funktionsweise eines Programms zu überprüfen.

Die erste und einfachste Methode, um das Funktionsverhalten eines Programms zu untersuchen, ist die Ausgabe von interessanten Daten auf ein Ausgabemedium. Bei Squeak ist dies das Transcript-Fenster. Da eine Ausgabe in dieses Fenster aber nur sehr langsam erfolgt, wurde als Entwicklungshilfsmittel ein Ausgabefenster geschaffen, welches die gleichen Methoden unterstützt, wie die Transcript-Klasse, aber wesentlich leistungsfähiger - sprich schneller - arbeitet: Das ATEOTranscript. Eine Übersicht über die meisten (ATEO-) Transcript-Methoden bietet die Internetseite [Squeak (2006)].

Tritt während der Ausführung eines Programms ein Fehler auf oder sorgt eine andere Aktion für diese Reaktion, so wird der Programmablauf unterbrochen und ein Debugging-Fenster in Squeak geöffnet (siehe Abbildung 7). Über dieses Fenster hat der Entwickler beziehungsweise Tester die Möglichkeit, die Entstehung des Fehlers nachzuvollziehen, indem er in der Methoden-Aufrufshistorie rückwärts navigiert; rückwärts heißt abwärts, also ist die oberste Zeile in diesem Bereich die zuletzt aufgerufene (jüngste) Methode. In jeder Methode kann der Entwickler alle Instanz- und Klassenvariablen inspizieren. Im oben genannten Beispiel wurden die Variablen objectsCurrentPosition und objectsNextPosition ausgewählt. Die Pfeile zeigen auf den jeweiligen Variableninhalt, die nun entweder mit bestimmten Sollwerten verglichen oder zumindest auf ihre Konsistenz hin überprüft werden können. Es besteht sogar die Möglichkeit, Programm-Code-Bereich Änderungen vorzunehmen und dann den Programmfluss fortzusetzen. Allerdings muss man dabei gewahr sein, dass durch dieses Vorgehen möglicherweise Effekte auftreten können, die bei einem wiederholten Durchlauf nicht wieder in Erscheinung treten. Wenn die Änderungen also sehr umfangreich sind, sollten diese erst gespeichert werden, wenn das laufende Programm beendet wurde. Mit einem anschließenden Neustart des Programms nach den Änderungsmaßnahmen ist der Entwickler auf der sicheren Seite. Dennoch sind die beiden Workspaces im Debugging-Modus von Vorteil. Man kann sie dazu nutzen, kleine Code-Stücke an der aktuellen Stelle des Programmflusses auszuführen. Im Beispiel habe ich während der Ausführung einer Demonstration des AAF-SideRails-Agenten die beiden oben genannten Variablen miteinander Addiert, indem ich im Workspace das Code-Stück objectsCurrentPosition + objectsNextPosition markiert und dann die Tastenkombination Alt + p (für PrintIt) gedrückt habe. Das Ergebnis ist durch die Textauswahl hervorgehoben: 804.0 @ 17096.47.

Der Programmfluss lässt sich manuell jederzeit durch die Tastenkombination Alt + '.' unterbrechen. Es öffnet sich ein Debugging-Fenster (wie oben beschrieben) an der Stelle, an der sich der Programmfluss aktuell befindet. Nun hat man wieder die Möglichkeit, sich rückwärts durch die Aufrufshierarchie an

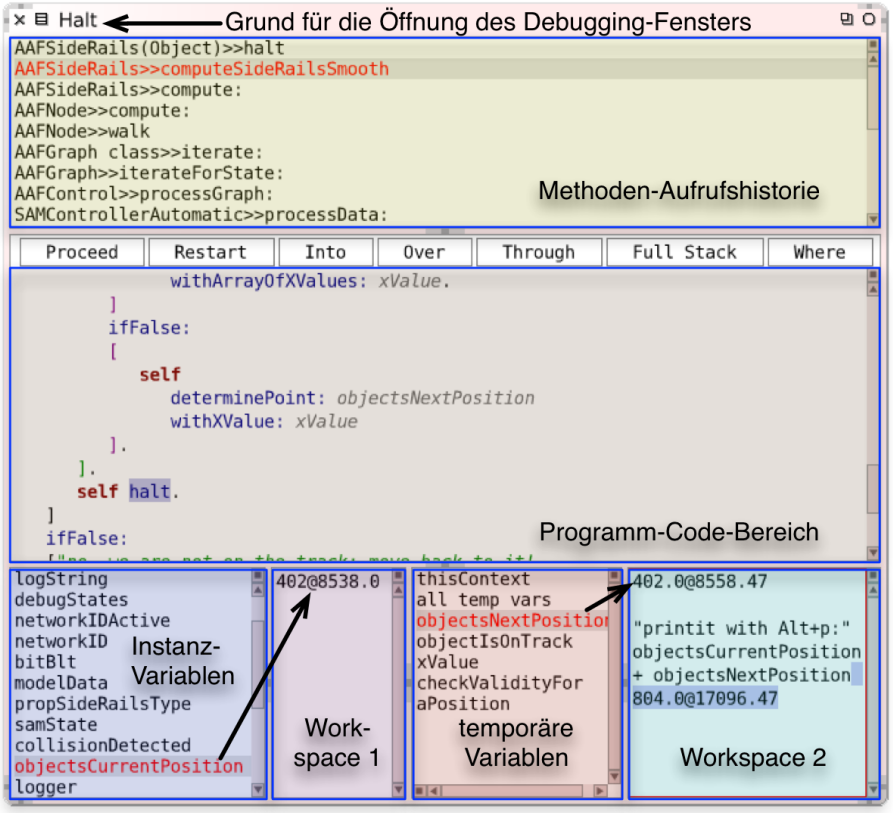

Abbildung 7: Dargestellt ist das Squeak-Debugging-Fenster mit Bezeichnungen für die jeweiligen Bereiche.

eine interessante Stelle zu navigieren, um dort Überprüfungen vorzunehmen.

Möchte man, dass sich an einer ganz speziellen Stelle im Programmfluss ein Debugging-Fenster öffnet, so erreicht man das, indem an die entsprechende Stelle die Codezeile 'SELF HALT.' eingefügt wird (vergleiche Abbildung 7, im Programm-Code-Bereich). Nun hat man wie gewohnt alle Möglichkeiten des Debugging-Fensters zur Verfügung.

2.5 Fehlerbeispiele

In diesem Unterabschnitt dokumentiere ich einige interessante Bugs, die während meines Mitwirkens im ATEO-Projekt aufgetreten sind. Ich habe sie ausgewählt, aufgrund ihrer verschiedenen Herkünfte, um so einige der in Unterabschnitt 2.2 genannten Fehlerquellen auch praktisch vorzustellen.

2.5.1 Unwirksame Joystick-Eingaben

Fehlerbeschreibung:
Während einer Fahrt mit zwei Mikroweltbewohnern wurden Joystick-Eingaben nicht von SAM umgesetzt. Das Trackingobjekt reagierte nicht so, wie es eigentlich durch die Steuerung hätte reagieren sollen. Ignoriert wurden nur die Eingaben von Mikroweltbewohner-2 und auch nur dann, wenn dieser den Joystick ausschließlich horizontal (x-Achse) auslenkte. Statt das Trackingobjekt dadurch in seiner Seitwärtsbewegung zu beeinflussen, bewegte es sich mit 50% Schub geradeaus. Erst wenn der Joystick ein Stück weiter nach vorne oder hinten (y-Achse) ausgelenkt wurde, reagierte das Trackingobjekt wie gewünscht: Es beschleunigte beziehungsweise verlangsamte und auch seitlich (x-Achse) ließ es sich wieder steuern.

Fehlereingrenzung:
Zunächst sind wir von einem Hardware-Defekt ausgegangen und haben den betreffenden Joystick durch einen anderen ersetzt. Aber auch mit dem Ersatz-Joystick trat das selbe Phänomen auf. Damit konnten wir einen Hardware-Defekt ausschließen und den Fehler in der Software suchen.
Als erstes wurde nun die verwendete Automatik deaktiviert. Aber auch ohne aktive Automatik, die Einfluss auf die Joystick-Eingaben hätte nehmen können, konnte der Fehler provoziert werden. Also musste der Fehler in SAM selbst zu finden sein.

Fehlerdiagnose:
Vermutlich wurde durch Kopieren und Einfügen eines Quelltextstücks an einer Stelle (durch gestrichelte Rechtecke in Abbildung 8 hervorgehoben) bei der Nachbearbeitung eine Variable nicht korrigiert (siehe rot eingekreiste Stelle) und dadurch ein ungewollter semantischer Fehler erzeugt. Dieser führte dazu, dass alle Eingaben in x-Richtung immer genau dann auf null gesetzt wurden, wenn keine y-Achsen-Auslenkung beziehungsweise nur eine Auslenkung im *Dead-Zone-Bereich* (siehe Abbildung 13) erfolgte. Im Nachhinein lässt sich nicht genau sagen, wie der Fehler tatsächlich zustande kam, aber sehr wahrscheinlich wurde der Teil im blauen Rechteck kopiert und an der Stelle des orangen Rechtecks eingefügt. Allerdings wurden dann nur die beiden Koordinaten vertauscht (siehe

Pfeile) ohne die notwendige Anpassung durchzuführen.

```
processJoystickPositions
    | joy1Pos joy2Pos |

    joy1Pos := modelData joystickRaw1.
    joy2Pos := modelData joystickRaw2.

    ((joy1Pos x abs) < (SAMConfig joystickThreshold))
        ifTrue: [joy1Pos := (0 @ joy1Pos y)].

    ((joy1Pos y abs) < (SAMConfig joystickThreshold))
        ifTrue: [joy1Pos := (joy1Pos x @ 0)].

    ((joy2Pos x abs) < (SAMConfig joystickThreshold))
        ifTrue: [joy2Pos := (0 @ joy2Pos y)].

    ((joy2Pos y abs) < (SAMConfig joystickThreshold))
        ifTrue: [joy2Pos := (joy2Pos y @ 0).].

    modelData joystick1XAxis: (joy1Pos x);
              joystick1YAxis: (joy1Pos y);
              joystick2XAxis: (joy2Pos x);
              joystick2YAxis: (joy2Pos y).
```

Abbildung 8: Dargestellt ist die Methode processJoystickPositions aus der Klasse SAMControllerInput *vor* der Fehlerkorrektur. Die fehlerhafte Stelle ist mit einem roten Kreis versehen.

Fehlerkorrektur:
Die Korrektur des Fehlers war in diesem Fall besonders leicht, da das rot umkreiste y lediglich durch ein x ersetzt werden musste.

2.5.2 Provisorische Methode I

Fehlerbeschreibung:
Unter Verwendung mehrerer Agenten in einer Automatik reagiert das Trackingobjekt sehr abrupt. Schon bei geringen Auslenkungen der / des Joysticks erreicht die Objektsteuerung bereits maximale Werte sowohl in x- als auch in y-Richtung.

Fehlereingrenzung:
Bei der Verwendung eines einzelnen Agenten gibt es keine Probleme. Auch eine beliebige Anzahl von Agenten in einer seriellen Anordnung im Automatik-Graphen führen nicht zu dem beschriebenen Verhalten. Werden zwei Agenten parallel angeordnet, so halbiert sich der Weg der Joystickauslenkung bis zum Erreichen der Maximalwerte. Fügt man weitere Agenten in paralleler Anordnung hinzu, verstärkt sich der Effekt. Das heißt, je mehr Agenten parallel hinzugefügt werden, desto weniger muss man einen Joystick auslenken, um damit den Maximalwert zu erreichen.

Fehlerdiagnose:

Die parallele Anordnung der Agenten führt zu einer Veränderung der Interpretation der jeweiligen Joystickauslenkung in x- beziehungsweise y-Richtung, die bei der seriellen Anordnung nicht auftritt. Daher konnte der Fehler bei der Vereinigungsmethode der Agentenergebnisse gefunden werden. Der Grund für das Fehlverhalten war ein unausgereiftes Merging-Konzept. Der Algorithmus summierte schlicht alle numerischen Werte der Rückgabe-Objekte der einzelnen Automatik-Agenten und fügte diese Summen dann ins Rückgabeobjekt der Automatik ein. Dadurch entstand der Effekt des skalierten Joystickinputs, wodurch die augenscheinliche Empfindlichkeit der Joysticksteuerung mit steigender Agentenzahl zunahm, da das Inputmaximum durch die Aufsummierung der individuellen Werte immer eher erreicht wurde.

Fehlerbeseitigung:
Ein sogenannter *Quick-Fix* sorgt nun dafür, dass die Summe der Joystickinputs durch die Anzahl der vorliegenden Ergebnisse (Outputobjekte) geteilt wird. Allerdings stellt dies keine akzeptable Endlösung dar, weil dadurch ein anderer ungewollter Effekt eintritt, wie das folgende Beispiel zeigt.

2.5.3 ProvisorischeMethode II

Fehlerbeschreibung:
Die Wirkung eines auf die Objektsteuerung einflussnehmenden Agenten in einer komplexen Automatik ist verschwindend gering.

Fehlereingrenzung:
Der betroffene Agent für sich funktioniert einwandfrei, wie ein Testexperiment mit einer Automatik zeigte, die nur diesen Agenten enthielt. Erst im Zusammenschluss mit anderen Agenten wurde die Wirkung des Agenten verringert.

Fehlerdiagnose:
Die Ursache für den beschriebenen Fehler war wieder die parallele Anordnung der Agenten im Automatik-Graph. Denn in der *Merge-Methode* wird von den Ergebniswerten *aller* Agenten das Arithmetische Mittel gebildet, unabhängig davon, ob die Agenten zuvor Änderungen an den Werten vorgenommen haben. Aufgrund der großen Anzahl an Agenten in der besagten Automatik, von denen ein Großteil keinen Einfluss auf die Objektsteuerungswerte nahm, wurde der Effekt des einen Agenten sehr gering.

Fehlerkorrektur:
An der Merge-Methode wurde zur Fehlerkorrektur nichts geändert, da ein *gutes* Merging mehrerer Ergebnisse nicht trivial ist. Stattdessen wurde vereinbart (obwohl die GUI Graphen mit paralleler Anordnung von Agenten weiterhin erlaubt), dass Agenten in Automatiken ausschließlich seriell - also hintereinander - anzuordnen sind, um das Merging mehrerer Ergebnisse zu vermeiden.

2.5.4 Fehler durch Modulverkettung

Die zuvor beschriebenen Fehler gaben mir Anlass, die Folgen und Auswirkungen zu eroieren, die die Anordnung von Agenten im Automatik-Graphen haben kann. Die Effekte, die mir dabei eingefallen sind, werde ich nun anhand einiger Beispiele erläutern. Die Anordnung der Agenten erfolgt dabei immer hinterein-

ander, um die Effekte des *schlechten* Mergings (siehe vorherige Unterabschnitte) zu umgehen.

Effekt-Aufhebung:
Angenommen zwei Agenten können die Einflussverteilung verändern. Der erste Agent gibt dem schnelleren Fahrer auf einer Geraden +30% (respektive -30% für den genaueren Fahrer) und der zweite gibt dem genaueren Fahrer +30%, wenn sich das Trackingobjekt neben der Strecke befindet. Das heißt, wenn sich das Trackingobjekt neben einer Geraden befindet, hebt sich die Wirkung der Agenten gegenseitig auf. Auch die Reihenfolge spielt hierbei keine Rolle, in welcher die Agenten abgearbeitet werden.

Derartige Beispiele lassen sich beliebig konstruieren. Ein weiteres Beispiel sind Agenten, die in einer Situation gegensätzliche Hinweise (egal ob auditiv oder visuell) geben:

- *langsamer fahren* versus *schneller fahren*
- *links fahren* versus *rechts fahren*
- ...

Effekt-Konflikt:
Ein Szenario, in dem zwei Agenten zwar keine gegensätzlichen Eingaben verursachen, dafür aber in eine Konfliktsituation geraten, die je nach Anordnung der Agenten im Graphen die „Missachtung“ der Ergebnisse des Vorgängers zur Folge hat. Angenommen der erste Agent verhindert das Verlassen der Strecke (zum Beispiel der AAFSideRails-Agent) und der zweite Agent soll Kollisionen mit Hindernissen vermeiden, indem diese mit einem bestimmten Mindestabstand zum Hindernis umfahren werden sollen. Folgt der SideRails-Agent dem Hindernisagent, so kollidiert das Trackingobjekt mit dem Hindernis, bleibt dabei aber auf der Strecke. Wird der Hindernisagent nach dem SideRails-Agent ausgeführt, verlässt das Trackingobjekt zwar die Strecke, kollidiert dafür nicht mit dem Hindernis (siehe Abbildung 9).

Diese Beispiele sollen verdeutlichen, dass Fehlverhalten unmittelbar auf Fehler in einzelnen Komponenten zurückzuführen sind, sondern ihre Ursprünge auch an ganz anderen Stellen haben können. Manche dieser Fehler sind eventuell gar nicht vorhersehbar, da bestimmte Seiteneffekte erst im Zusammenspiel der einzelnen Komponenten auftreten. Lösungen für diese Probleme wären beispielsweise Warnungen, die dann ausgegeben werden, wenn mehrere Agenten auf die selben Werte im SAMState zugreifen. Zudem müssen sich Architekten von Automatiken sehr gut mit den Wirkungsweisen der verwendeten Agenten auskennen und sich den Gefahren der Wechselwirkungen der Agenten untereinander bewusst sein.

2.5.5 Das Peak-Problem

Fehlerbeschreibung:
Unter ganz bestimmten Umständen, die bisher unbekannt aber durch entsprechende Testkonfigurationen reproduzierbar sind, gibt es einen Anstieg (Peak)

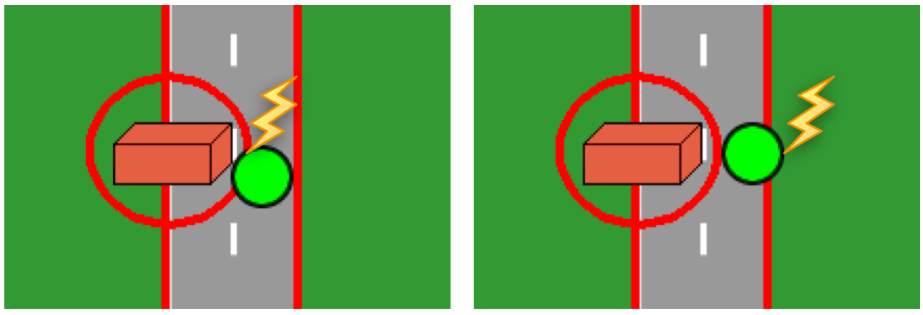

Abbildung 9: In beiden Bildausschnitten ist ein Hindernis-Agent (roter Kreis) und ein Leitplanken-Agent (rote Fahrbahnbegrenzung) angedeutet. Im linken Bildausschnitt verletzt das Trackingobjekt den Sicherheitsbereich des Hindernisses während die Leitplankenfunktion gewahrt bleibt. Im rechten Bildausschnitt wird zwar der Sicherheitsabstand zum Hindernis eingehalten, dafür wird aber der Streckenrand überschritten und die Leitplankenfunktion verletzt.

der Tick-Dauer bis zu einem sehr hohen Maximalwert (gemessen wurden Verzögerungen von über 1000 ms). Ist das Maximum einmal erreicht, fällt die Tick-Dauer wieder auf das Normalniveau von 39/40 ms ab.

Fehlereingrenzung:
Der Peak tritt immer nur in Verbindung mit dem Durchfahren von der Strecke *hauptabschnitt_lang* auf. In einigen Fällen bereits im ersten Durchgang mit *hauptabschnitt_lang*, in anderen Fällen erst im zweiten Durchgang. Wird auf unterschiedlicher Hardware dasselbe Image benutzt (gespeichert auf einer externen USB-Festplatte), tritt der Peak *an unterschiedlichen Stellen* auf. Wird hingegen dasselbe Experiment (also auch die Input-Werte sind identisch) auf einer Hardware mehrfach durchgeführt, so ist der Peak nahezu an derselben Stelle; beobachtet wurde eine Varianz von lediglich 7 Ticks früher beziehungsweise später.

Egal wie oft *hauptabschnitt_lang* durchfahren wird (getestet wurden bis zu 10 Durchgänge hintereinander in einem Experiment), tritt der Peak nur ein einziges Mal auf.

Anfängliche Vermutungen, es könnte sich um die Ausführung des *Garbage Collectors* handeln, der zu einem ungünstigen Zeitpunkt 'den Müll weg räumt', konnten nicht bestätigt werden. Allerdings führte der frühzeitige manuelle Anstoß des Garbage Collectors zu Verbesserungen, hatte jedoch dafür zusätzliche kleinere Peaks zur Folge.

Ebenso konnte ausgeschlossen werden, dass die Ursache das AAF selbst ist, denn auch mit einer Dummy-Automatik, in der kein Agent enthalten ist, ja selbst ganz ohne Automatik, ist der Peak aufgetreten.

Die erste signifikante Verbesserung brachte der Austausch des verwendeten Squeak-Images zusammen mit zusätzlichen leistungsoptimierenden Änderungen (siehe Unterabschnitt 4.2) am Softwaresystem. Allerdings konnte auch danach

der Peak noch mit Hilfe des extra dafür angefertigten Zeitmess-Werkzeugs (Unterabschnitt 4.1) unterhalb der 39/40 ms-Grenze nachgewiesen werden.

Fehlerkorrektur:
Bis zum heutigen Tag konnte die genaue Ursache nicht gefunden werden, die den Peak produziert. Vermutlich handelt es sich aber um eine Reihe von kleineren Ereignissen, die alle zu dem Anstieg der Berechnungsdauer führt. Veranlasst wird diese Vermutung dadurch, dass die Amplituden der Peaks durch einige optimierende Maßnahmen - wie bereits erwähnt - signifikant verkleinert werden konnten, sodass der Effekt des stockenden Prozessablaufs nicht weiter auffällt.

2.5.6 Verwendung von Double-Variablen

Vergleich von Doubles:
Möchte man zwei Doubles miteinander vergleichen, ist es nicht ratsam, dies direkt über den Vergleichsoperator zu implementieren. Man erreicht den Vergleich aber relativ geschickt dadurch, dass man zunächst beide Doubles auf die gewünschte Genauigkeit *normalisiert* und erst dann miteinander in Relation setzt. In meinem Beispiel (siehe Listing 2) wird eine Möglichkeit vorgestellt, wie so eine Normalisierung aussehen kann. Zunächst entscheidet man sich für eine Genauigkeit: Eine 1 gefolgt von einer Anzahl Nullen, welche die Anzahl der Stellen nach dem Komma/Punkt angibt, auf die die Double normalisiert werden soll. Die Normalisierung erfolgt dann in drei Schritten:

1. Multiplikation der Double mit dem Präzisionswert (Kommaverschiebung nach rechts, Zeile 12 in Listing 2)
2. Abschneiden (oder auch Rundung) des Nachkommateils (Zeile 15)
3. Division durch den Präzisionswert (Linksverschiebung des Kommas, Zeile 18) liefert die normalisierte Double.

```
1  normalizeDouble: aDouble precision: aPrecision
2    "if one wants to 'normalize' a double with a special
3     precision, you get back a double with so many di-
4     gits after the dot as the precision has zeros;
5     e. g. one wants 4 digits after the dot, then one
6     chooses:
7     - precision == 10000 and
8     - aDouble == 1.2345678,
9     than one gets: returningDouble == 1.2345"
10
11   "shifting the dot to the right"
12   returningDouble := aDouble * precision.
13
14   "cutting off the rest after the dot"
15   returningDouble := returningDouble integerPart.
16
17   "shifting the dot back to the left"
18   returningDouble := returningDouble / precision.
```

```
19
20    ^returningDouble.
```

Listing 2: Möglichkeit einer Double Normalisierung

Besondere Eigenheiten von Doubles:
Zudem können sich Doubles in Squeak unvorhersehbar verhalten. Wenn man sie beispielsweise mit Integern in einer Schleife verknüpft (etwa durch Addition, Subtraktion, etc.), kann dies einige Durchläufe wie gewünscht funktionieren, aber es kann dabei auch möglicherweise zu einem Rundungsfehler kommen, wie im Transcript-Fenster in Abbildung 10 zu erkennen ist. Das zugrundeliegende Programm, mit dem dieser Fehler provoziert werden kann, ist Listing 3 zu entnehmen.

```
1  d := 1.23456789.
2  d := d * d.
3  Transcript show: d asString; cr.
4
5  4 timesRepeat:
6  [
7    d := d + 1.
8    Transcript show: d asString; cr.
9  ].
```

Listing 3: Code zur Provozierung eines Rundungsfehlers

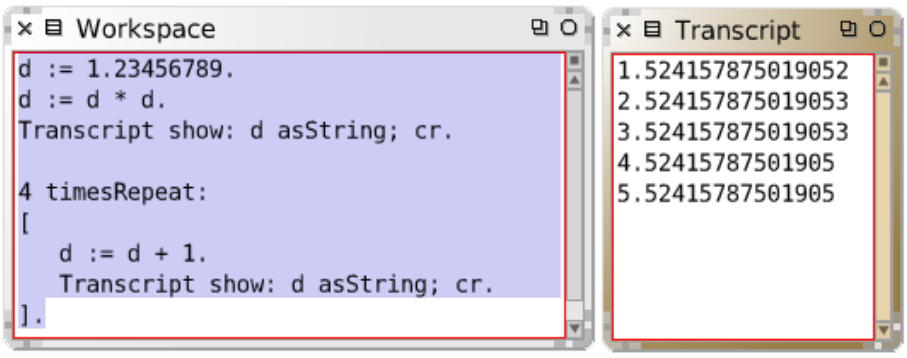

Abbildung 10: Dargestellt sind der Workspace mit dem Programm aus Listing 3 und die Ausgabe des Double-Rundungsfehlers, der bei der Ausführung des Programms entsteht (beachte die letzten Stellen der Ausgaben im Transcript-Fenster). Die erwartete Ausgabe hätte nur an der Stelle vor dem Komma Veränderungen aufweisen sollen.

3 Koordinatensysteme in Squeak und SAM

In diesem Kapitel werden verschiedene Koordinatensysteme in Squeak und SAM vorgestellt. Dazu gehört das Hauptkoordinatensystem von Squeak (Abschnitt 3.1), das SAM-Koordinatensystem (Abschnitt 3.2), das Joystick-Koordinatensystem (Abschnitt 3.3) und das Koordinatensystem des Trackingobjekts. Zudem wird erläutert, auf welche Weise Hindernisse auf der Strecke beziehungsweise der Fahrbahn positioniert werden (Abschnitt 3.5).

3.1 Standard Squeak-Koordinatensystem

In Squeak beziehungsweise SAM gibt es mehrere unterschiedliche Bezugspunkte für Koordinatensysteme. Standardmäßig ist der Ursprung des Koordinatensystems in Squeak die linke obere Fensterecke (siehe Abbildung 11 → Display). Dabei muss darauf geachtet werden, dass die Ordinate (oder auch Funktions- oder Y-Achse) von oben nach unten verläuft.Die Funktionswerte werden somit nach unten hin größer. Die Orientierung der Abszisse (X-Achse) ist - wie aus dem Kartesichen Koordinatensystem gewohnt - von links nach rechts ausgerichtet; die Werte werden also von links nach rechts größer. Dieses System ist die Grundlage für die Anzeige des Streckenausschnitts (sichtbarer Bereich der Streckengrafik) bei SAM.

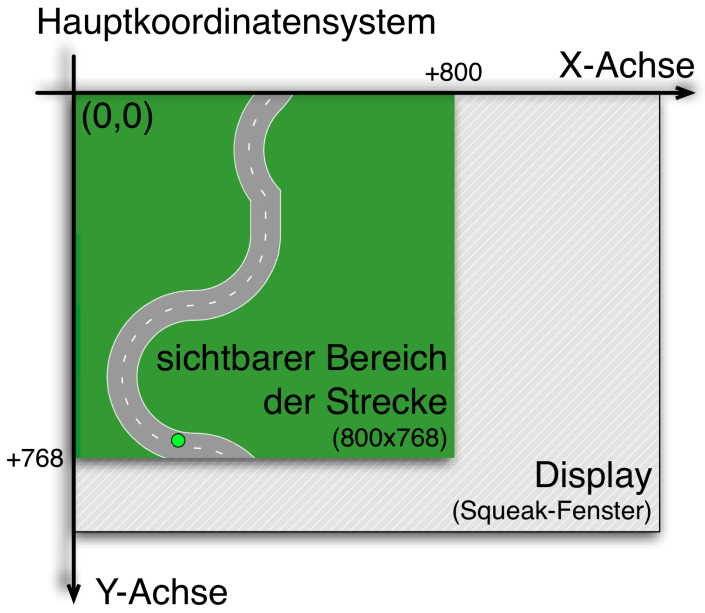

Abbildung 11: Hauptkoordinatensystem in Squeak mit dem sichtbaren Bereich der Strecke (SAM).

Der sichtbare Bereich wird in Form einer Ebene bezüglich des Display-Ursprungs bei $(0, 0)$ eingefügt - daher haben beide Koordinatensysteme denselben absoluten Ursprung. Durch diesen Sichtbereich (800 Pixel breit, 768 Pixel hoch) wird

die Streckengrafik vom Start zum Ziel hindurch geschoben, wodurch der Eindruck des Fortbewegens erzeugt wird.

Die Ausdehnung des Displays hängt in erster Linie von der Größe des Squeak-Fensters ab und wird zudem beschränkt durch die maximale Auflösung des Hardware-Displays. Wenn also das Squeak-Fenster maximiert wird, hat man die größtmögliche Fläche zur Verfügung, auf der man Bilder darstellen kann.

3.2 SAM-Koordinatensystem

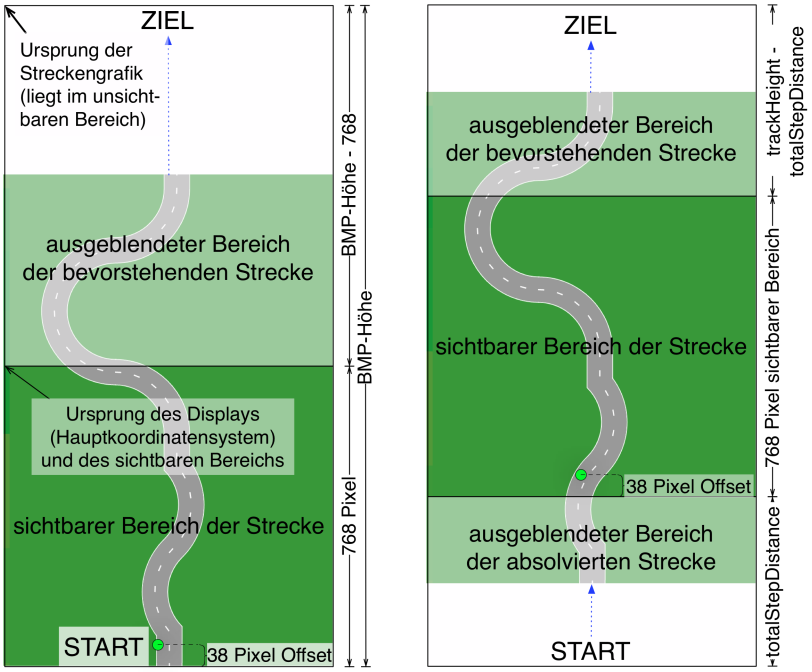

Abbildung 12: SAM-Koordinatensystem, TOTALSTEPDISTANCE und TRACKHEIGHT - links die Ansicht im Startzustand, rechts eine beliebige Position während einer Fahrt.

Speziell für SAM wurde ein Koordinatensystem eingeführt, das dem ersten Quadranten des Kartesischen Koordinatensystems entspricht. Es hat seinen Ursprung in der linken unteren Ecke der jeweiligen Streckengrafik. Die Streckengrafik ist eine aus Kacheln zusammengesetzte Grafikdatei im *Windows* **B***it***M***a***p**-*Format* (BMP). Diese Kacheln liegen ebenfalls im BMP-Format vor und bilden den Bausatz für alle Strecken.

Während das Trackingobjekt bezüglich des sichtbaren Bereichs (und des Displays) immer den selben Y-Wert hat

$$\text{Trackingobjekt}_{Y(Display)} = 768 - \text{Trackingobjekt-Offset}$$

, ändert sich der Y-Wert bezogen auf die Streckengrafik mit jedem Tick, in dem es sich 'vorwärts bewegt'. Diese Vorwärtsbewegung (eigentlich handelt es sich

um das Hinunterschieben der Streckengrafik) wird gemessen und in der Variablen TOTALSTEPDISTANCE im SAMModelData-Objekt gespeichert. Allerdings entspricht die TOTALSTEPDISTANCE nicht dem Y-Wert des Trackingobjekts, sondern nur der Strecke, die die Streckengrafik bereits durch den sichtbaren Bereich geschoben wurde (zu sehen auf der rechten Seite in Abbildung 12). Im Startzustand (zu sehen auf der linken Seite der Abbildung 12) ist diese Distanz null. Um den Y-Wert des Trackingobjekt-Mittelpunkts in diesem System zu bestimmen ($TO_{Y(\text{TOTALSTEPDISTANCE})}$), muss immer noch dessen vertikaler Offset (38 Pixel) zur TOTALSTEPDISTANCE addiert werden:

$$TO_{Y(\text{TOTALSTEPDISTANCE})} = \text{TOTALSTEPDISTANCE} + \text{Trackingobjekt-Offset}.$$

Bezüglich des oberen Randes der Streckengrafik (Ziel) hat das Trackingobjekt den Y-Wert:

$$TO_{Y(Streckengrafik)} = (\text{TRACKHEIGHT} + 768) - \text{Trackingobjekt-Offset}.$$

Für diese Berechnung ist der Ursprung oben links (Koordinatensystem der Streckengrafik). Es handelt sich bei diesem Wert um die Distanz vom Trackingobjekt-Mittelpunkt bis zum oberen Rand der Streckengrafik.

3.3 Joystick-Koordinatensystem

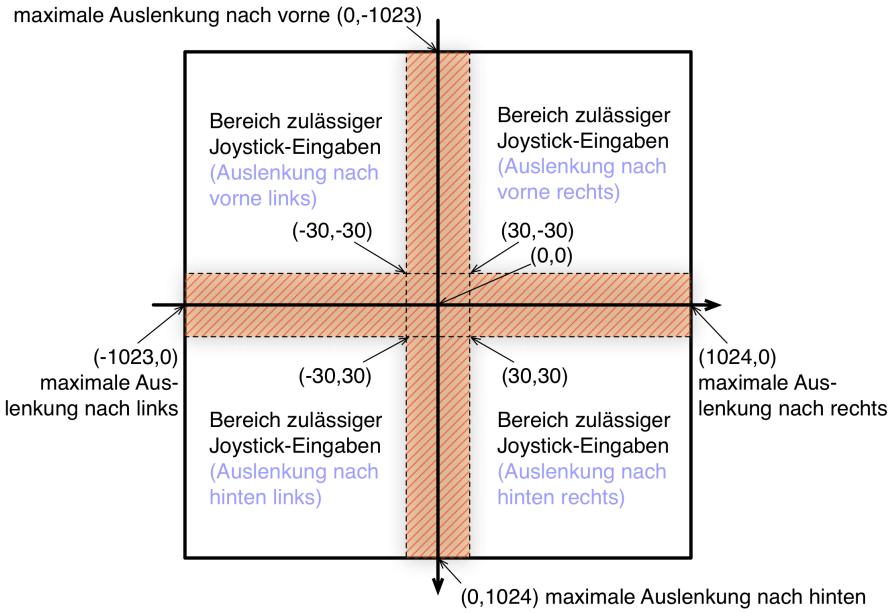

Abbildung 13: Joystickkoordinatensystem, Bereich der dead-zone ist gestreift dargestellt.

Dieses System hat dieselben Achsenorientierungen wie das Hauptkoordinatensystem von Squeak. Allerdings befindet sich der Koordinatenursprung in der

Nullstellung des Joystick (relaxed position). Die Grenzwerte eines Joysticks liegen bei $[-1023, 1024]$ (jeweils auf der X- beziehungsweise Y-Achse). -1023 deshalb, weil pro Achse nur 2048 Positionseinheiten zur Verfügung stehen und die Null Teil dieser Einheiten ist.

In SAM wurde eine so genannte *dead-zone* eingeführt (dargestellt als gestreifte Fläche in Abbildung 13). Alle Eingaben des Joysticks, die kleiner als 30 Positionseinheiten auf jeder Achse in jede Richtung (also jeweils von -30 bis $+30$) sind, werden bei der Interpretation (in SAMControllerInput → processJoystickPositions) per se auf Null gesetzt. Dadurch erhält man eine Ruhe- oder Relaxed-Stellung des Joysticks. In dieser Joystickstellung bewegt sich das Trackingobjekt mit 50% der Maximalgeschwindigkeit geradeaus vorwärts, also ohne Seitenbewegungen. Ohne diese Zone wäre eine Geradeausfahrt nahezu unmöglich, da man einen Joystick nie exakt auf einer Achse auslenken kann, sondern immer ein wenig davon abweicht, was unweigerlich zu einer sehr kleinen Seitenbewegung beziehungsweise einer minimalen Geschwindigkeitsänderung führen würde. Befindet sich eine Joystickeingabe also innerhalb des gestreiften Bereichs, wird entweder die entsprechende Koordinate auf Null gesetzt, oder beide, wenn sich die Eingabe innerhalb der Koordinaten $(30, 30), (-30, 30), (-30, -30), (30, -30)$ befindet (relaxed position, gestricheltes Quadrat mit Mittelpunkt im Koordinatenursprung).

3.4 Trackingobjekt-Koordinatensystem

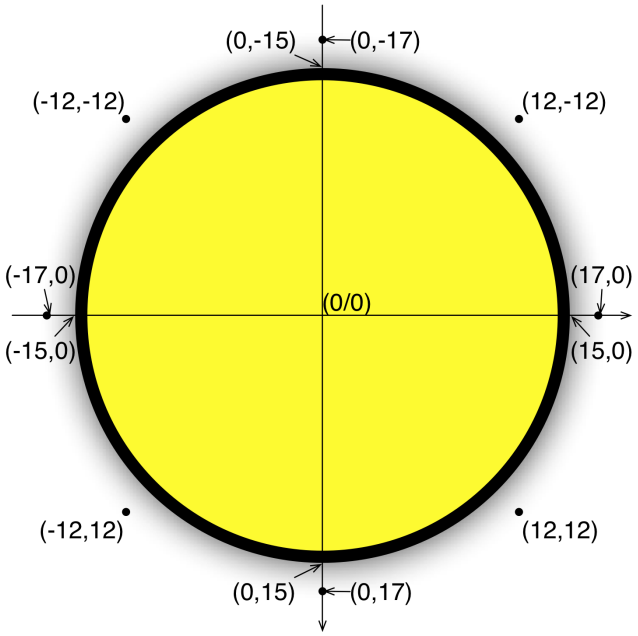

Abbildung 14: Koordinatensystem und Sensoren (schwarze Punkte) des Trackingobjektes.

Ein weiteres wichtiges Koordinatensystem hat seinen Ursprung im Mittelpunkt

des Trackingobjekts. Es dient hauptsächlich dazu, mit Hilfe bestimmter Koordinaten, die als Sensoren genutzt werden (dargestellt als acht schwarze Punkte in Abbildung 14), das Umfeld des Trackingobjekts auf besondere Farbwerte hin zu untersuchen. Insbesondere werden diese Sensoren dazu verwendet, um Zusammenstöße mit Hindernissen zu erkennen. Das Koordinatensystem des Trackingobjekts kann aber auch dazu verwendet werden, Streckenkacheln zu analysieren. Darüber wird ausführlich in der an diese Arbeit anschließenden Diplomarbeit eingegangen. Wie beim Hauptkoordinatensystem in Squeak ist auch bei diesem Koordinatensystem die Orientierung der Ordinate von oben nach unten und die Orientierung der Abszisse von links nach rechts zu sehen.

3.5 Hindernispositionierung

Bei der Positionierung der Hindernisse wurde eine Methode verwendet, für deren Erklärung ein kleiner Exkurs in die BitBlt-Klasse notwendig ist.

Ein BitBlt-Objekt hat eine Quelle (source) und ein Ziel (destination). Die Quelle kann beispielsweise eine externe Grafik oder eine beliebige andere Form (Kreis, Rechteck, etc.) aus Squeak sein. Das Ziel ist in der Regel das Squeak-Display (siehe Unterabschnitt 3.1), kann aber auch eine andere Form sein, zum Beispiel die Ebene des sichtbaren Bereichs.

Soll nun eine Grafik oder andere beliebige Form als BitBlt auf einer anderen Form angezeigt werden, hat man zwei Möglichkeiten diese dort zu positionieren. Zunächst ist es aber wichtig zu wissen, wie das Positionieren überhaupt funktioniert.

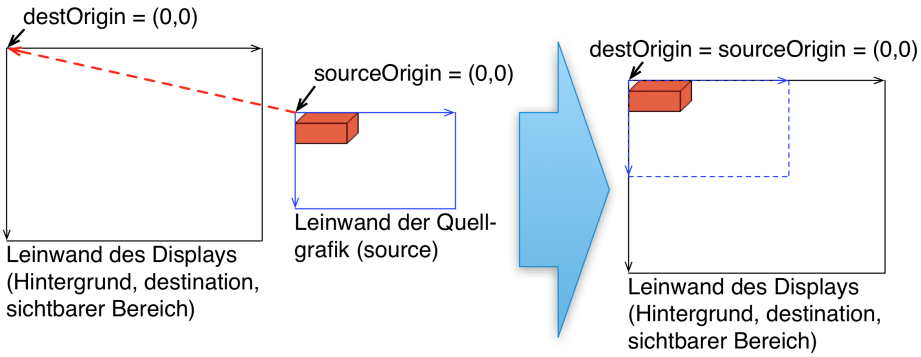

Abbildung 15: Positionierung einer Grafik mit DESTORIGIN = SOURCEORIGIN = $(0, 0)$.

Sowohl die Quelle als auch das Ziel haben eigene Koordinatensystem-Ursprünge. Diese heißen SOURCEORIGIN beziehungsweise DESTORIGIN. Beim Positionieren wird die SOURCEORIGIN über die DESTORIGIN geschoben und die Bilddaten aus der Quelle relativ dazu im Ziel eingefügt. In Abbildung 15 wird die Quellgrafik (in den Beispielen die obstacle.gif), deren SOURCEORIGIN sich bei $(0, 0)$ befindet, auf dem Display in der oberen linken Ecke angezeigt, da auch der Ursprung des Ziels bei $(0, 0)$ liegt. Die SOURCEORIGIN wird an der Stelle DESTORIGIN eingefügt und die Bilddaten relativ dazu auf die Leinwand des Displays kopiert.

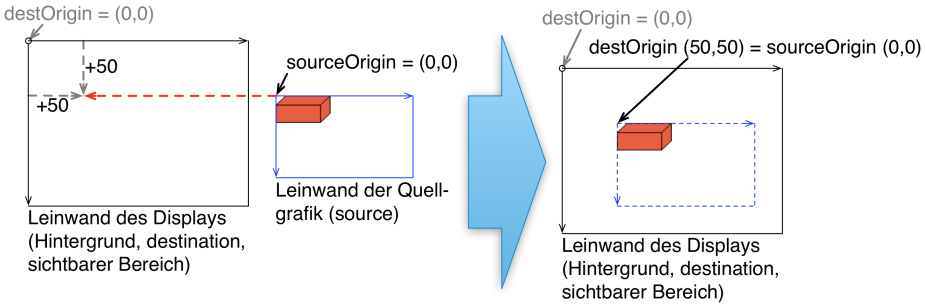

Abbildung 16: Positionierung einer Grafik mit DESTORIGIN = (50,50) und SOURCEORIGIN = $(0, 0)$.

Die erste Möglichkeit, die Quellgrafik an einer anderen Stelle anzuzeigen, ist in Abbildung 16 dargestellt. Als Referenzpunkt für das Einfügen der Quelle wird dieses Mal nicht DESTORIGIN = $(0, 0)$ verwendet, sondern eine andere Position auf dem Display: DESTORIGIN = $(50, 50)$.

Man kann eine Verschiebung aber auch auf eine andere Weise erreichen: Verändert man die SOURCEORIGIN, so wird die *Leinwand der Quelle* bereits vor dem Einfügen in das Ziel verschoben. Die Auswirkungen des Verschiebens der SOURCEORIGIN in unterschiedliche Richtungen sind in Abbildung 17 zu sehen.

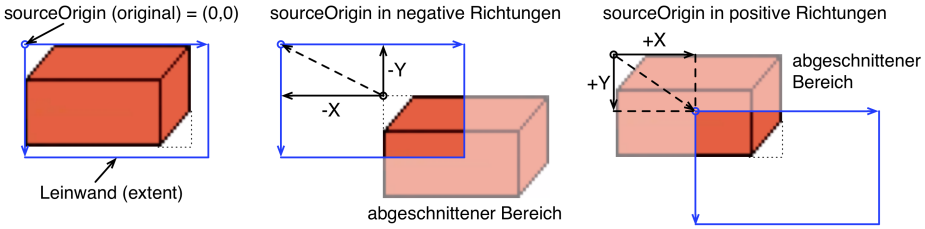

Abbildung 17: Effekte beim Verschieben der SOURCEORIGIN - die Leinwand schneidet je nach dem bestimmte Bereiche der Grafik ab.

Abbildung 18 zeigt eine Verschiebung des Quell-Koordinatensystems in positive Richtungen. Um den Effekt, den man damit erzielt, zu verdeutlichen, wird das Koordinatensystem nur um einen Bruchteil der Maße der Quellgrafik verschoben, nämlich um etwa $(25, 15)$. Der neue Referenzpunkt in der Quelle, der für das Einfügen im Ziel maßgeblich ist, befindet sich nun in der Nähe des Mittelpunktes der Quellgrafik. Wie man in der Abbildung sehen kann, wird so erreicht, dass die Quellgrafik nach oben links verschoben wird. Der Teil der Grafik, der nach der Verschiebung außerhalb der Quellleinwand liegt, wird abgeschnitten und am Ziel nicht mit angezeigt. Nur die Grafikinformationen aus dem Quadranten mit positiven X- und Y-Werten werden auf die Zielleinwand kopiert.

Verschiebt man stattdessen SOURCEORIGIN in negative Richtungen und fügt dann die Grafikinformationen der Quellleinwand bei DESTORIGIN = $(0, 0)$ ein, so wird die Grafik effektiv nach rechts unten verschoben. Allerdings muss man

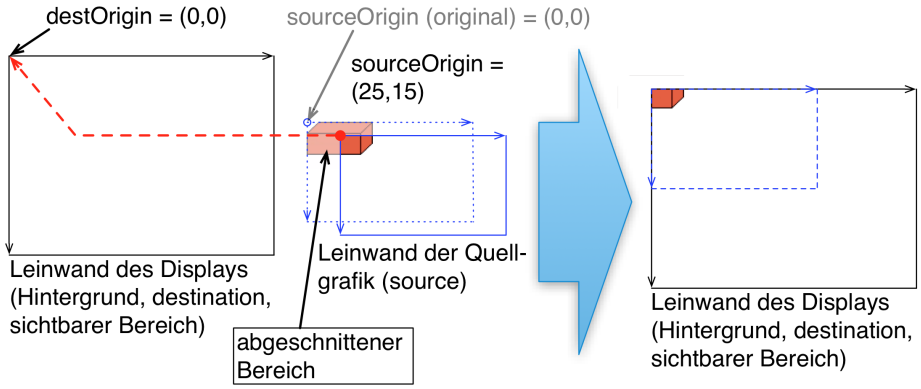

Abbildung 18: Positionierung einer Grafik, deren SOURCEORIGIN um positive Werte verschoben wird.

darauf achten, dass die Ausdehnung (EXTENT) der Leinwand groß genug ist, damit nicht der rechte untere Teil der Grafik abgeschnitten wird (es sei denn, das Abschneiden ist gewünscht; vergleiche auch Abbildung 17). Im Fall von SAM wird die Quellleinwand von der Originalgröße (57×30) auf 800×768 Pixel vergrößert, damit genug Platz zum Verschieben vorhanden ist. Wie diese Verschiebung prinzipiell aussieht ist in Abbildung 19 dargestellt.

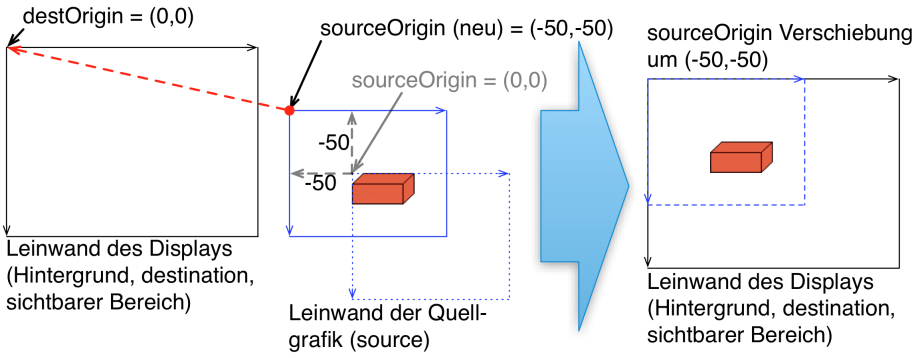

Abbildung 19: Positionierung einer Grafik, deren SOURCEORIGIN um negative Werte verschoben wird.

Quizfrage: Was passiert, wenn man DESTORIGIN = SOURCEORIGIN = $(25, 15)$ setzt?

4 Maßnahmen zur Ausführungszeitoptimierung von SAM

In diesem Abschnitt dokumentiere ich die Ausführungszeitoptimierungen, die ich an der Verwaltung des Distance-Dictionarys vorgenommen habe und das eigens für diesen Zweck konstruierte Zeitmesswerkzeug, mit dessen Hilfe ich die Verbesserungen nachweisen konnte.

4.1 Werkzeug zur Messung von Ausführungszeiten

In Unterabschnitt 2.5.5 habe ich einen Fall vorgestellt, in dem es zu einem drastischen Anstieg der Ausführungszeit eines Durchlaufs des processStep-Loops in der Klasse SAMControllerStep kommt. Um genauere Messungen von Ausführungszeiten durchführen zu können und ob durchgeführte Maßnahmen den gewünschten Zeitgewinn erzielen konnten, habe ich das Werkzeug AAFDurationLog entworfen und implementiert, mit dem man die Ausführungszeit eines Code-Blocks messen und loggen kann.

Um eine Ausführungszeitmessung durchzuführen, müssen einige Dinge beachtet werden. Zunächst muss vor dem Einsatz der Messmethode eine Initialisierung durch die Ausführung von

AAFDurationLog initializeDurationLog.

erfolgen.

Vor jeder Zeitmessung muss die Methode

AAFDurationLog totalTicks: (AAFDurationLog totalTicks) + 1.

aufgerufen werden, damit eine neue Zeile in der Datentabelle hinzugefügt wird. Die Variable TotalTicks enthält also die Anzahl der Messwerte. Sollen periodische Messungen vorgenommen werden - beispielsweise in einer Schleife - , dann muss diese Methode innerhalb der Schleife gleich zu Beginn stehen (siehe Listing 4, Zeile 7 und 8), damit die Werte, die womöglich in der Schleife ermittelt werden, ihre eigene neue Zeile in der Datentabelle erhalten.

Nun können mit zwei unterschiedlichen Methoden die Ausführungszeiten geloggt werden. Zum Einen geschieht das über die Methode

AAFDurationLog logThis: aNumber loggerID: String.

Durch sie kann eine beliebige Zahl (in Millisekunden) in das Logbuch geschrieben werden (wenn bereits über eine andere Methode eine Zeit bestimmt wurde). Der String gibt dabei an, an welche Stelle der Tabelle geloggt werden soll. Wird zum Beispiel die Ausführungszeit eines bestimmten Agenten gemessen, so macht es Sinn, eine Bezeichnung als String-Parameter zu verwenden, mit der man den Agenten hinterher im Logbuch leicht ausfindig machen kann. Das gilt auch für die zweite Methode, die die eigentliche Messfunktion bereitstellt:

AAFDurationLog measureTimeFor: aBlock loggerID: String.

Der als Parameter übergebene Block wird ausgeführt, nachdem die aktuelle Zeit gespeichert wurde. Nach der Ausführung wird die Differenz von der Zeit nach der Ausführung und der Zeit vor der Ausführung gebildet. Das Ergebnis ist die Dauer der Ausführung in Millisekunden und wird an die entsprechende Stelle im Logbuch geschrieben.

Die geloggten Daten lassen sich auf zwei unterschiedliche Weisen wieder ausgeben. Die erste Methode schreibt die ermittelten Daten einfach in ein geöffnetes Transcript-Fenster. Als Beispiel (siehe Listing 4) habe ich ein kurzes Programm gewählt, dass 10 Transcript-Ausgaben erzeugt. Die Dauer jeder einzelnen Ausgabe wird gemessen und anschließend alle gemessenen Zeiten tabellarisch im Transcript angezeigt.

```
"initializing"
AAFDurationLog initializeDurationLog.
"to have more than one Transcript-output:"
1 to: 10 do:
[ :i|
  "counter for the table-rows (initial value is 0):"
  AAFDurationLog totalTicks:
    ((AAFDurationLog totalTicks) + 1).
  "measurement of time:"
  AAFDurationLog measureTimeFor:
    [Transcript show: 'Transcript Output #',
      i asString; cr.]
    loggerID: 'TranscriptTime'.
].
Transcript show: '###DONE###';cr.
"Present the measured values on the Transcript:"
AAFDurationLog writeToTranscript.
```

Listing 4: AAFDurationLog Beispiel

Die Zweite Methode funktioniert nur in Verwendung mit einem SAM-Experiment, da hierbei die Vergabe eines Log-Datei-Namens automatisch erfolgt. Diese Datei wird im selben Verzeichnis abgelegt, in dem auch die normalen Log-Datein des Experiments gespeichert werden. Außerdem wird der gleiche Name des zugehörigen normalen Step-Logs verwendet und nur um einen Zusatz ergänzt. Erstellt wird dabei ebenfalls eine **C***omma*-**S***eparated* **V***alues* (CSV)-Datei.

Zur Zeit existiert eine Einschränkung: Die Aufrufe des AAFDurationLog dürfen nicht geschachtelt werden. Das heißt, dass in dem zu messenden Block keine weiteren Messungen mit AAFDurationLog durchgeführt werden dürfen; auch nicht in einer Methode in einer tieferen Aufrufsebene. Der Grund dafür ist, dass das Werkzeug nicht für diese Art des Einsatzes konzipiert wurde, sondern sollte lediglich für das Messen von Ausführungszeiten aus einer Methodenebene heraus verwendet werden.

4.2 Optimierung der Distance-Dictionary-Verwaltung

Auf der Suche nach dem Grund für das *Peak-Problem* (siehe Dokumentation in Unterabschnitt 2.5.5) wurde von mir die Aktualisierungsmethode für das Distance-Dictionary - einer Sammlung von Entfernungswerten zu den nächsten besonderen 'geographischen' Ereignissen auf der vorausliegenden Strecke - optimiert. Als erstes implementierte ich einen Mechanismus, zur Säuberung des Dictionarys (Unterabschnitt 4.2.2). Allerdings hatte diese Optimierung nur einen geringen Erfolg. Erst durch die zusätzliche Änderung der Methode, mit der die nächsten besonderen geographischen Ereignisse aus dem Dictionary gefiltert werden, wurde eine signifikante Leistungsverbesserung erreicht (Unterabschnitt 4.2.3). Die Ergebnisse der Experimente, die ich in Tabelle 3 aufgeführt habe, diskutiere ich in Unterabschnitt 4.2.4.

4.2.1 Git und Squeak

Um die folgenden Abschnitte leichter zu verstehen, ist es nötig, dass ich einige Begriffe einführe.

Git
Bei der Software *Git* handelt es sich um ein freies, dem *open source* Prinzip unterliegendes Versionen-Kontroll-System für eine verteilte Software-Entwicklung, welches auch im ATEO Projekt zum Einsatz kommt. Wer mehr über Git erfahren möchte, konsultiert bitte die Dokumentation auf der Webseite [git (2012)].

Das Squeak-System
Squeak ist eine IDE für einen Dialekt der Programmiersprache *Smalltalk*. Dieses besteht aus mehreren separaten Komponenten, die im Zusammenspiel das *Squeak-System* bilden, mit dem wir im ATEO Projekt arbeiten. Diese Komponenten und ihre Beziehungen sind in Abbildung 20 dargestellt und werden im Folgenden erläutert.

Die Squeak-VirtualMachine
Sie ist das Herzstück des Systems. Mit ihrer Hilfe wird das Arbeiten mit Squeak erst möglich. Die *SqueakVM* funktioniert aber nur zusammen mit den *Squeak Quellen (Sources)* und einem *Squeak Image* (plus der zugehörigen *Changes-Datei*).

Die Squeak Quellen
Diese Bibliothek beinhaltet die grundlegenden Funktionen von Squeak. Alle Standard-Klassen und deren Methoden sind hierin gespeichert und werden beim Start der Squeak-VM geladen und stehen während des Betriebs zur Verfügung.

Das Squeak-Image
Es stellt eine Art Wechseldatenträger, vergleichbar mit einer virtuellen Festplatte, dar, der von der Squeak-VM eingelesen und verwendet wird. In einem Image wird mit einer Sicherung immer der aktuelle Entwicklungsstand der zu entwickelnden Software gespeichert.

Die externen Quellen
Zusätzlich zu den herkömmlichen Squeak Quellen gibt es im ATEO Projekt auch noch die externen Quellen. Um sie nutzen zu können, müssen sie manuell in das Squeak Image geladen werden. Änderungen, die im ATEO Git veröffent-

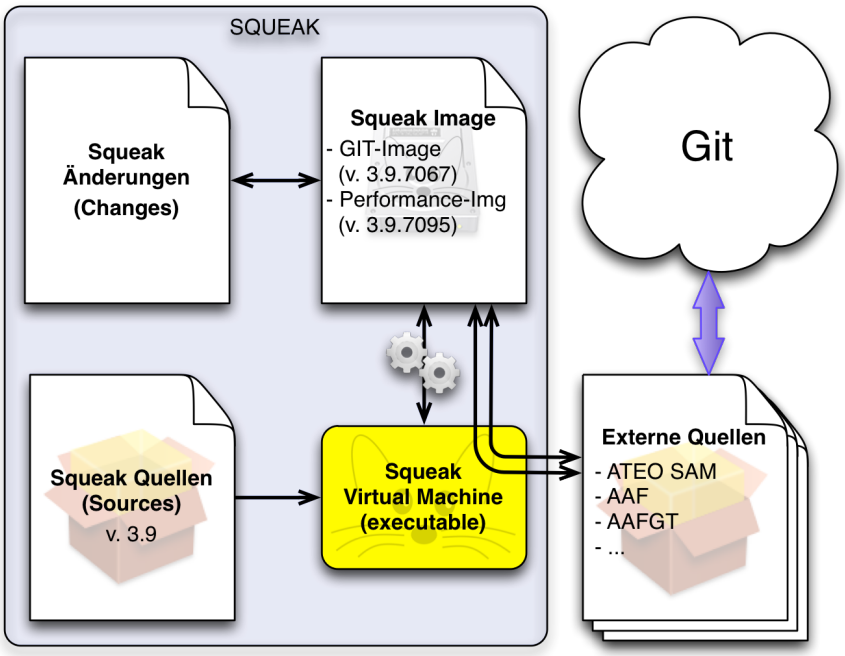

Abbildung 20: Die Komponenten Squeak Virtual Machine, Squeak Quellen, Squeak Image und die Changes bilden zusammen das Squeak-System. In den externen Quellen ist die eigentliche Entwicklungsarbeit enthalten, die im Versionen-Kontrollsystem Git bei allen Entwicklern synchron gehalten wird.

licht werden sollen, müssen zunächst manuell aus dem Squeak-Image exportiert werden. Anschließend sind die Squeak-Programmdateien (erkennbar an der Erweiterung 'st') bereit im Git aktualisiert zu werden.

4.2.2 Entfernen alter Einträge aus dem Distance-Dictionary

Der erste Gedanke für die Optimierung war, den Aktualisierungsalgorithmus so zu verändern, dass er obsolete Einträge aus dem Distance-Dictionary entfernen sollte. Eine kleine Verbesserung wurde dadurch zwar erreicht, aber eine signifikante Änderung leider nicht.

Der Algorithmus vergleicht die aktuelle Position des Trackingobjekts mit den (sortierten) Ende-Werten der Elemente im Distance-Dictionary. Es werden alle Elemente aus dem Dictionary entfernt, deren Ende-Wert kleiner ist, als der y-Wert des Trackingobjekts. Daher wird das Dictionary mit zunehmender Fahrt immer kleiner und die Suche nach den nächsten Elementen erfolgt mit konstanter Geschwindigkeit, da nun die obsoleten Elemente nicht immer wieder mit durchsucht werden müssen, wie es bisher der Fall war. So lässt sich im Übrigen auch die ansteigende Ausführungsdauer (siehe Abbildung 21) für die Methode updateDistances vor den Optimierungen erklären. Denn mit zunehmender Fahrtdauer mussten mehr und mehr Elemente durchsucht werden, die bereits durchfahren wurden; und das in jedem einzelnen Tick (= eine Iteration von SAMController processStep). Durch das Entfernen „abgefahrener" Streckenelemente konnte zwar der Anstieg vermieden werden, jedoch blieb die relativ hohe Ausführungszeit der Methode updateDistances bestehen und die Zahl der kleineren Peaks (bis 60 ms mit Spitzen um die 100 ms) wurde deutlich erhöht.

4.2.3 Effizienterer Algorithmus zum Distance-Dictionary-Update

Eine signifikante Reduzierung dieser Peaks erreichte ich durch ein Refactoring des Algorithmus, der für die Suche nach den nächsten Elementen im Distance-Dictionary zuständig ist. In seiner ersten Version stieß er für jeden Elementtyp einzeln eine Suche durch das Dictionary an, bis das nächste Element des gesuchten Typs gefunden wurde.

Ich habe den Algorithmus so geändert, dass nur noch eine Suche durch das Dictionary erfolgt. Die Suche läuft so lange, bis alle Einträge einer Typ-Checkliste auf true gesetzt wurden oder das Ende des Dictionarys erreicht wurde. Wenn einmal ein Element eines noch offenen Typs gefunden wurde, wird nach keinem weiteren Element diesen Typs gesucht. Dieses Vorgehen hat die Ausführungszeiten erheblich verkleinert (vergleiche Abbildungen 22, 23, 24 und 25).

4.2.4 Diskussion der Ergebnisse

Um meine Optimierungen zu verifizieren, habe ich das Ausführungszeitmesswerkzeug aus Unterabschnitt 4.1 auf die Ausführung der Methode updateDistances angewendet, in der auch die in Unterabschnitt 4.2.2 vorgestellte Methode Anwendung findet. Um zu beweisen, dass meine Änderungen tatsächlich

eine Verbesserung gebracht haben, führte ich fünf Experimente durch, die ich zusätzlich auf zwei weiteren Systemkonfigurationen wiederholt habe. Die einzelnen Versuchskonfigurationen sind in Tabelle 3 aufgeführt.

In jedem der fünf Experimente habe ich drei Fahrten auf der Strecke *hauptabschnitt_lang* durchgeführt. Daher gibt es auch für jedes Experiment drei Diagramme, in denen jeweils die Anzahl der benötigten Ticks auf der x-Achse und die Ausführungsdauer der Methdoe updateDistances auf der y-Achse abgetragen worden sind.

Wie ebenfalls Tabelle 3 zu entnehmen ist, habe ich auch das verwendete Squeak-Image variiert. Frühere Experimente, die durchgeführt wurden, um dem *Peak-Problem* auf die Schliche zu kommen, haben gezeigt, dass auch das in ATEO verwendete Image eine mögliche Ursache für die Peaks ist. Der Einsatz eines absolut frischen Images von der Squeak-Webseite hat bereits Verbesserung gebracht. Daher habe ich auch dieses frische Image mit in meinen Experimenten verwendet.

Experiment	optimiert	Image-Version	Automatik
I	—	Git-Image	—
II	✓	Git-Image	—
III	✓	Performance-Image	—
IV	✓	Git-Image	Konzept 36
V	✓	Performance-Image	Konzpet 36

Tabelle 3: Versuchskonfigurationen der durchgeführten Experimente

Systemkonfiguration für die Experimente:
Für die Versuche habe ich eine Virtual Machine (VM) mit dem Gast-Betriebssystem *Windows 7 Professional x64* installiert. Der VM habe ich 2 CPUs (je 2.8 GHz) sowie 4 GB RAM und 256 MB VRAM zugewiesen. Als VM-Host benutze ich einen Apple iMac 27" mit einem i7 Quad-Core-Prozessor (je 2.8 GHz) und 16 GB RAM und dem Betriebssystem *Mac OS X 10.8.2 - Mountain-Lion*. Die VM habe ich mit dem Programm *Parallels Desktop 7* eingerichtet. Für die Versuche habe ich keine Joysticks verwendet, wodurch als ständiger, für alle Versuche gleicher, Input beider Mikroweltbewohner (-1024@1023) verwendet wurde - also maximale Beschleunigung und maximale Auslenkung nach links.

Zur Bestätigung habe ich Experimente zusätzlich unter einer nativen *Windows 7 Professional x64* Installation auf dem oben genannten iMac sowie in einer VM mit einer CPU (2.8 GHz), 1 GB RAM und dem Gastsystem *Windows XP Professional x86 SP3* (Host-System war in diesem Fall ein MacBook Pro, i7 DuoCore 2.8 GHz CPU, 4 GB RAM und OS X 10.8.2) wiederholt. Die Ergebnisse waren den hier beschriebenen (siehe Abbildung 21 bis 25) extrem ähnlich. Lediglich die zeitliche Position der Peaks war unterschiedlich.

Ergebnisse im Detail - Experiment I
Für dieses Experiment habe ich die ursprüngliche Versuchskonfiguration verwendet, um einen Ausgangswert für meine Vergleiche mit den anderen Experimenten zu erhalten. Das heißt, ich nutzte das ATEO Squeak-Image aus dem Git und die alte updateDistances-Methode. Wie man in Abbildung 21 sieht, stiegen die Ausführungszeiten mit zunehmender Tick-Zahl fast monoton an. Auch die

Peaks hatten erhebliche Spitzenwerte, die nach den Optimierungen nicht wieder erreicht wurden.

Ergebnisse im Detail - Experiment II
Auch hierfür habe ich das alte Image zusammen mit den Verbesserungen verwendet. Deutlich zu sehen ist, dass der Anstieg nicht mehr stattfindet und auch die Ausführungszeiten im Schnitt sehr stark gesunken und relativ konstant sind. Je Fahrt ist nur ein einzelner Peak aufgetreten, der nur knapp oberhalb der 40 Millisekundengrenze liegt.

Ergebnisse im Detail - Experiment III
Ein ähnliches Bild erhielt ich nach Durchführung des dritten Experiments, in dem ich statt des alten Images nun das frische Image verwendete. Die Amplitude der einzelnen Peaks erreicht nun nicht mehr die 40 Millisekunden, in Fahrt 2 kann man sogar gar keinen Peak mehr erkennen.

Ergebnisse im Detail - Experiment IV
Unter Verwendung des alten Images und der Automatik *Konzept 36* treten die Peaks fast gar nicht mehr in Erscheinung. Spitzenwerte von 4 beziehungsweise 6 Millisekunden sind nun signifikant geringer als die Spitzenwerte in den Experimenten zuvor. Ebenfalls ist ein Trend erkennbar, dass die Ausführungszeiten mit zunehmender Tick-Zahl *abnimmt*!

Ergebnisse im Detail - Experiment V
Das Zusammenspiel von Performance-Image und der Automatik *Konzept 36* sorgt für eine weitere Verringerung der Ausführungszeiten. Nun sind 3 bis 4 Millisekunden die Höchstwerte. Auch hier ist zu sehen, dass die Ausführungszeiten mit zunehmender Tick-Zahl geringer werden.

Folgerungen
Wie man gut in (fast) allen Diagrammen sehen kann, wurde das *Peak-Problem* auch durch die optimierenden Maßnahmen nicht vollständig gelöst. Für den Leser ist nicht sichtbar, dass andere Ursachen nach wie vor für Peaks sorgen, die ihren Ursprung nun aber mit Sicherheit nicht mehr im Update-Prozess des Distance-Dictionarys hat. Denn nach den durchgeführten Optimierungen hat sich die Amplitude der Gesamt-Peaks deutlich verringert. Sie gehen nicht mehr bis hinauf in den 1000 Millisekundenbereich, sondern bleiben sogar fast unterhalb der Grenze, die ein Tick normaler Weise maximal dauern sollte: 40 Millisekunden. Daher sind die optimierenden Maßnahmen durchaus als Erfolg zu bezeichnen, wenngleich die Peaks im Allgemeinen existent bleiben.

Was besonders beachtlich ist - und vor allem auch noch durch andere Experimente auf weiteren Systemkonfigurationen bestätigt werden konnte - ist die Tatsache, dass mit dem Einsatz von Automatiken die Peaks kleiner werden! Selbst eine Dummy-Automatik, in der kein einziger Agent verwendet wird, erzeugt höhere Peaks, als eine Automatik mit Agenteneinsatz. Das ist zwar erfreulich, aber dennoch unbefriedigend zugleich, da dafür keine Erklärung gefunden werden konnte.

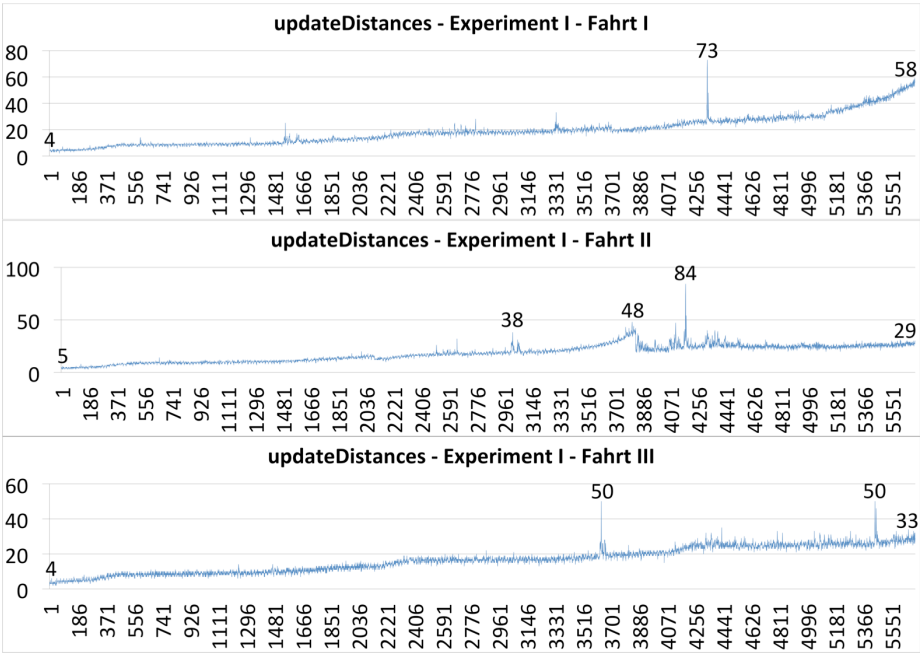

Abbildung 21: Experiment I. Ausführungszeiten der Methode UPDATEDISTANCES vor den optimierenden Maßnahmen mit dem Git-Image selbst ohne den Einsatz einer Automatik.

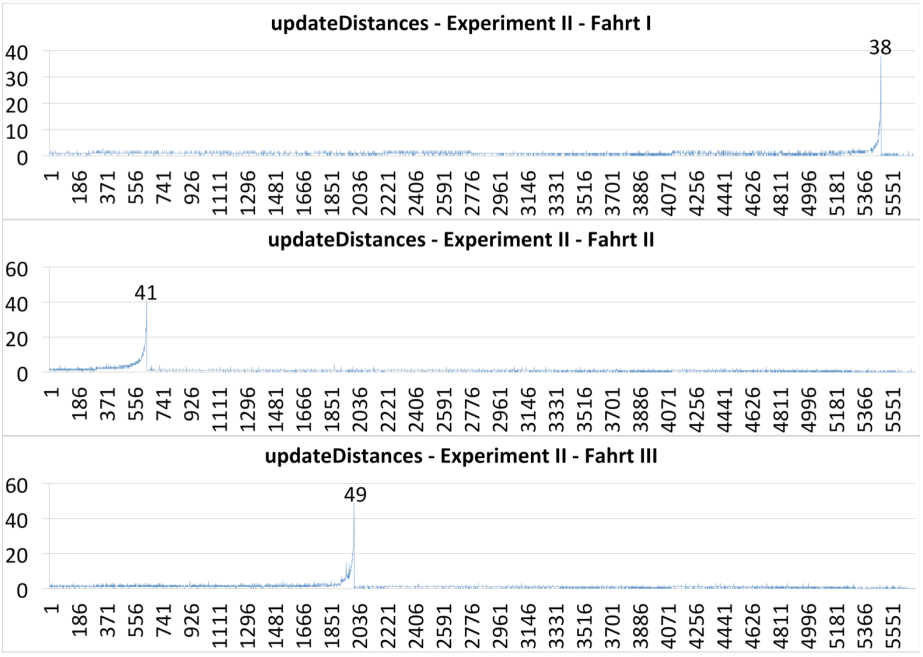

Abbildung 22: Experiment II. Verbesserte Ausführungszeiten der Methode UPDATEDISTANCES nach den optimierenden Maßnahmen mit dem Git-Image und ohne den Einsatz einer Automatik.

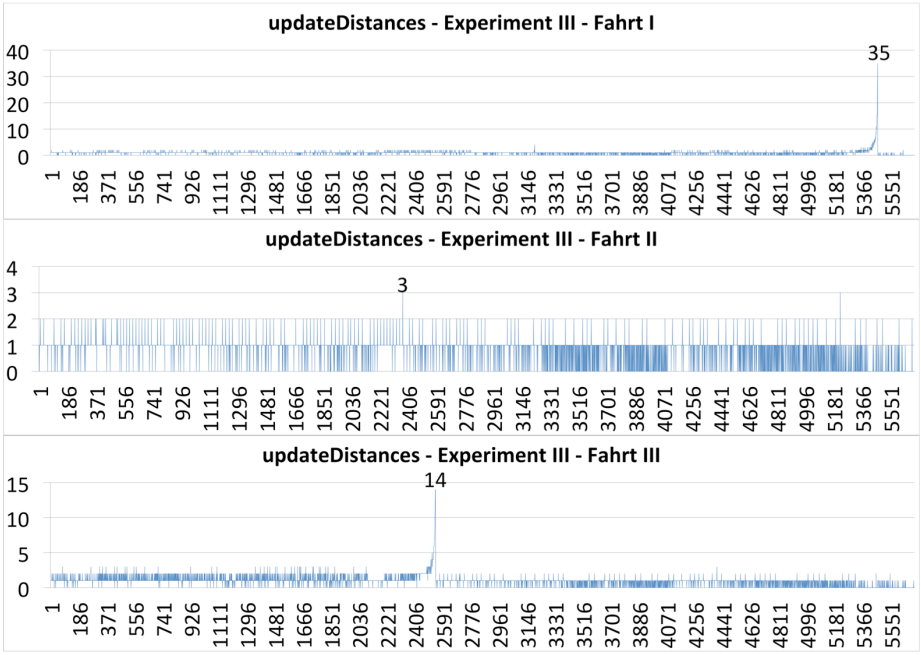

Abbildung 23: Experiment III. Ausführungszeiten der Methode UPDATEDISTANCES nach den optimierenden Maßnahmen im *Performance-Image* ohne den Einsatz einer Automatik.

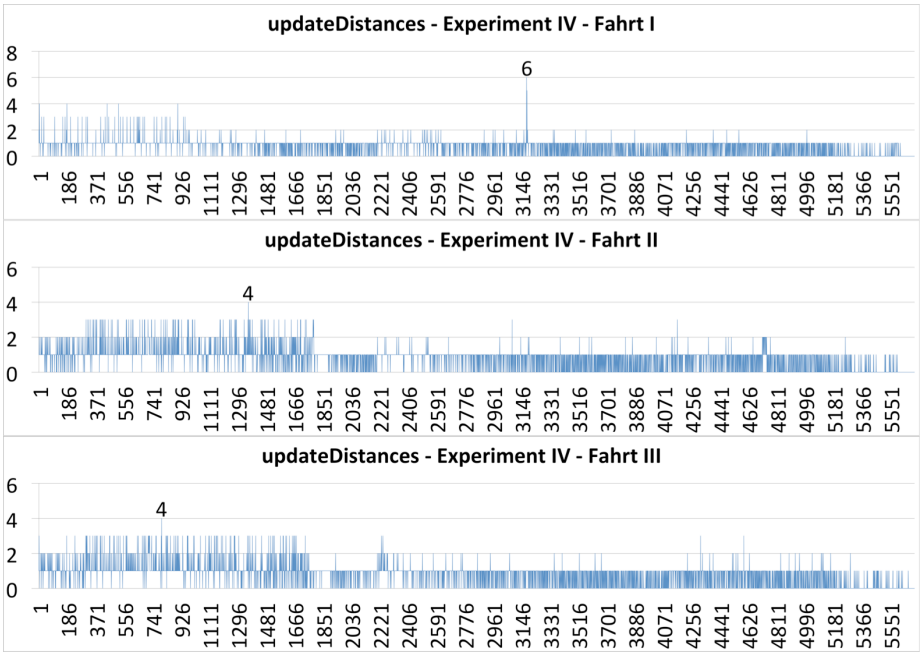

Abbildung 24: Experiment IV. Ausführungszeiten der Methode UPDATEDISTANCES nach den optimierenden Maßnahmen im Git-Image unter Einsatz der Automatik *Konzept 36*.

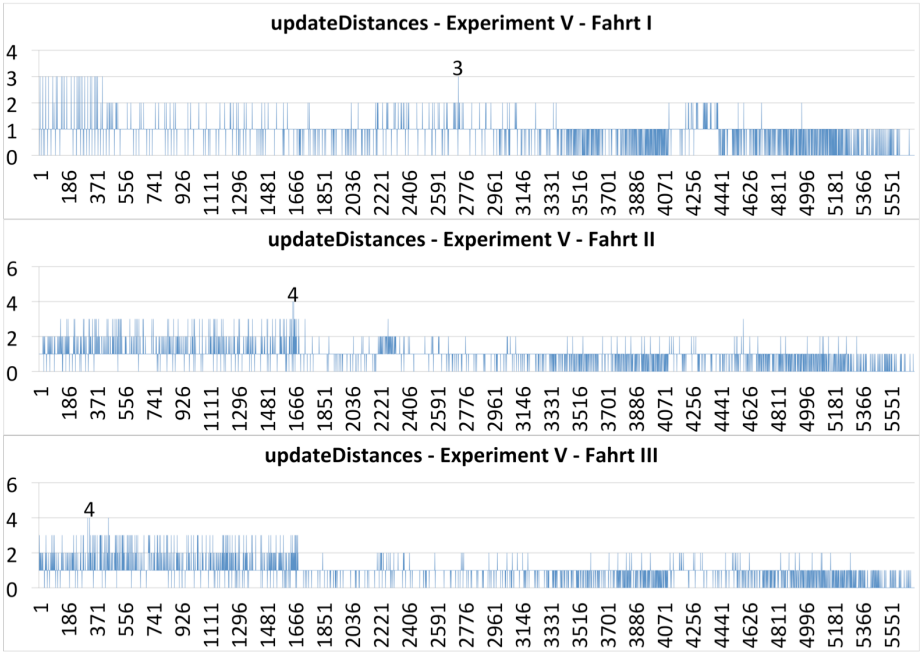

Abbildung 25: Experiment V. Ausführungszeiten der Methode UPDATEDISTANCES nach den optimierenden Maßnahmen mit dem *Performance-Image* unter Einsatz der Automatik *Konzept 36*.

5 Zusammenfassung und Ausblick

Fehler sind da, um gemacht zu werden. Aber Fehler sind auch dazu da, um aus ihnen zu lernen. Ich für meinen Teil habe sehr viel gelernt; gerade auch auf der Suche nach Fehlern, die nicht in meiner eigenen Domäne zu finden waren. Durch die Beschäftigung mit dem Thema *Fehler* bin ich sehr viel sensibler gegenüber den häufigsten Fehlerquellen (Stichwort „Copy 'n' Paste“) geworden und achte sehr viel genauer darauf, korrekten Code zu produzieren. Insbesondere die vielen Beispiele in Unterabschnitt 2.5 sollen in eindrucksvoller Weise demonstrieren, wie sehr sich schon kleine Dinge im Gesamtsystem auswirken können.

Die vielen Unstimmigkeiten beim Thema der unterschiedlichen Koordinatensysteme in Squeak, die es im Verlauf meiner Zeit im ATEO-Projekt gab, haben mich dazu bewogen dafür zu sorgen, dass unsere Nachfolger in diesem Projekt einen leichteren Einstieg finden. Das Kapitel 3 soll jedem einen Überblick und eine Nachschlagemöglichkeit geben, wenn man sich in bestimmten Situationen nicht sicher ist, welches Koordinatensystem gerade relevant ist.

Obwohl heutige Computer-Technik sehr leistungsfähig ist, stellt ein in seinen Ressourcen sehr beschränktes System, wie es bei Squeak der Fall ist, eine ganz besondere Herausforderung für die Entwickler dar. Es hat sich gezeigt, dass es sich lohnt, gleich von Beginn an ressourcenschonende Algorithmen zu entwickeln. Brute-Force-Versuche eignen sich zwar um Machbarkeit zu prüfen, sind aber meist an zu wenig Speicher (die Virtual-Machine von Squeak kann nur mit maximal 512 MB arbeiten) oder zu geringer Leistung (die Squeak-VM unterstützt nur einen Prozessor) gescheitert. Umso wichtiger ist der Einsatz von selbstgebauten Werkzeugen, wie dem Ausführungszeitmesswerkzeug aus Unterabschnitt 4.1. Sie gewährleisten einen genauen Blick ins System und wandeln diese Black-Box in ein offenes Buch, aus dem man sehr viele Informationen gewinnen kann, um noch mehr Leistungssteigerungen zu erwirken.

Literatur

[git 2012] : *Git by The Software Freedom Conservancy*. 2012. – URL http://git-scm.com. – Zugriffsdatum: 2012-11-09

[ATEO 2008] ATEO: *ATEO - ein Beitrag zum Graduiertenkolleg prometei*. 2008. – URL http://www.psychologie.hu-berlin.de/prof/ingpsy/forschung/ateo. – Zugriffsdatum: 2012-07-01

[Conformiq 2012a] Conformiq: *Conformiq Designer and Modeler*. 2012. – URL http://www.conformiq.com. – Zugriffsdatum: 2012-11-01

[Conformiq 2012b] Conformiq: *Conformiq Designer Features*. 2012. – URL http://www.conformiq.com/products/feature-specifications/. – Zugriffsdatum: 2012-11-01

[Kosjar 2012] Kosjar, Nikolai: *Ein Ereignis-System für das ATEO Automation Framework sowie die Implementierung und Testung von auditiven und visuellen Hinweisen*, Humboldt-Universität zu Berlin, Diplomarbeit, 2012

[Neumann 2008] Neumann, J. P.: *Programmiersprachenwahl bei der Entwicklung sicherheitsrelevanter Software*, Hochschule Darmstadt, Diplomarbeit, Feb 2008. – URL http://demod.org/_media/blog/2008/10/bachelorarbeit-neumann.pdf

[prometei 2006–2012] prometei, FSP8: *Graduiertenkolleg prometei; Forschungsschwerpunkt 8: Prof. Dr. Hartmut Wandke*. 2006-2012. – URL http://www.prometei.de/forschungsschwerpunkte/fsp-8.html. – Zugriffsdatum: 2012-11-11

[Sommerville 2011] Sommerville, Ian: *Software Engineering*. 9. Pearson, 2011

[Squeak 2006] Squeak: *Language Notes*. 2006. – URL http://squeak.joyful.com/LanguageNotes. – Zugriffsdatum: 2012-11-02

[Wandke und Nachtwei 2008] Wandke, Hartmut ; Nachtwei, Jens: The different Human Factor in Automation: The Developer behind versus the Operator in action. In: Waard, Dick de (Hrsg.) ; Flemisch, Frank (Hrsg.) ; Lorenz, Bernd (Hrsg.) ; Oberheid, Hendrik (Hrsg.) ; Brookhuis, Karel (Hrsg.): *Human Factors for assistance and automation*. Maastricht, the Netherlands: Shaker Publishing, 2008, S. 1—10

[Wickert 2012] Wickert, Andreas: *Klassifizierung, Umsetzung und Testung von Automatiken für regelnde Eingriffe in die Objektsteuerung von SAM*, Humboldt-Universität zu Berlin, Diplomarbeit, 2012

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel *Fehler und deren Vermeidung bei der Programmierung und Konfiguration von Automatiken im ATEO-Projekt* selbständig und ohne unerlaubte Hilfe verfasst habe.

Berlin, 20. Dezember 2012

Andreas Wickert

Ich bin damit einverstanden, dass die vorliegende Arbeit mit dem Titel *Fehler und deren Vermeidung bei der Programmierung und Konfiguration von Automatiken im ATEO-Projekt* in der Bibliothek ausgelegt wird.

Berlin, 20. Dezember 2012

Andreas Wickert